Atomic Architectural Component Recovery for Program Understanding and Evolution

Evaluation of Automatic Re-Modularization Techniques and Their Integration in a Semi-Automatic Method

Von der Fakultät Informatik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Rainer Koschke

aus Stuttgart

Hauptberichter: Prof. Dr. E. Plödereder
Mitberichter: Prof. Dr. J. Ludewig

Tag der mündlichen Prüfung: 27. Oktober 1999

Institut für Informatik, Universität Stuttgart

2000

Acknowledgments

I would like to thank my thesis advisor, Erhard Plödereder, for his continual support over all my years in his group. He has provided valuable guidance and advice and he has shared many interesting ideas with me. I am thankful to Thomas Eisenbarth, Jean-François Girard, and Georg Schied for many interesting discussions on the topics of my thesis that have broadened my perspective. I appreciate the support of Hiltrud Betz and Thomas Eisenbarth in the implementation of the analyses described in this thesis. I would also like to thank Andreas Zendler for his guidance in finding appropriate statistical tests for the experiment conducted for this thesis. Last but not least I want to thank Hausi Müller and his group for making Rigi available to me.

Table of Contents

List of Abbreviations

ADO	Abstract Data Object
ADT	Abstract Data Type
CPP	C Preprocessor
ExtC	External Connectivity
HC	Hybrid Component
IC	Internal Connectivity
ICS	Institute of Computer Science
IESE	Fraunhofer Institute of Experimental Software Engineering
IML	Intermediate Language
IntC	Internal Connectivity
KLOC	Thousand (Kilo) Lines of Codes
LOC	Lines of Codes
RS	Related Subprograms
RUA	Resource Usage Analysis
RUG	Resource Usage Graph

Abstract

The literature is rich of fully automatic and semi-automatic techniques for component recovery and their number is still growing. The abundance of published methods calls for frameworks to unify, classify, and compare them in order to make informed decisions. This thesis introduces a classification of component recovery techniques based on a unification of 23 techniques. Focussing on structural techniques, 16 fully automatic techniques are classified into connection-, metric-, graph-, and concept-based subcategories and the commonalities and variabilities of these techniques are discussed in depth. Beyond the qualitative comparison, 12 structural techniques are evaluated quantitatively (concept-based techniques were excluded). To that end, an evaluation scheme is introduced that allows to measure recall and precision of component recovery techniques with respect to a set of reference components ascertained by software engineers. Among the evaluated techniques is our new metric-based technique named *Similarity Clustering*. The evaluation scheme based on a set of expected components manually compiled by 5 software engineers for four C systems with altogether 136 KLOC shows that *Similarity Clustering* is among the best techniques for all systems, but it also has more false positives than other techniques. The overall result of this comparison is that none of the fully automatic techniques has a sufficient detection quality.

In order to overcome this problem, a semi-automatic method is presented in this thesis in which computer and maintainer collaborate to detect components. The method is supported by a framework that integrates the existing fully automatic techniques. In this framework, the automatic techniques can be run successively and their results be validated by the user. For this purpose, all the techniques are enhanced to work incrementally. The unification of the automatic techniques makes it possible to implement incremental variants for whole classes of techniques. The results of the techniques can be combined by high-level operators modeled on intersection, union, and difference for fuzzy sets. An alternative way of integration is offered by a voting approach that summarizes the individual agreement of automatic techniques.

Despite of the new ways of combining the automatic techniques, the semi-automatic method inherits weaknesses of the integrated techniques. Future research should investigate whether these weaknesses may be overcome with additional, more precise information gained from dataflow analyses and more domain-oriented information. However, all methods will always have to cope with vagueness and subjectivity of the grouping criteria for components.

Zusammenfassung

In der Literatur findet sich eine große Anzahl voll- und halbautomatischer Techniken zur Komponentenerkennung und ihre Zahl wächst stetig. Die Fülle der publizierten Techniken macht eine Klassifikation und Bewertung notwendig, um begründete Entscheidungen bei der Auswahl einer geeigneten Technik zu ermöglichen. In dieser Arbeit wird eine Klassifikation basierend auf einer Vereinheitlichung von 23 Techniken zur Komponentenerkennung eingeführt. Eine engere Betrachtung von 16 strukturellen Techniken liefert eine Subkategorisierung in verbindungs-, metrik-, graph- oder begriffsbasierte Techniken, deren Gemeinsamkeiten und Unterschiede eingehend vorgestellt werden. Über den rein qualitativen Vergleich hinaus werden 12 strukturelle Techniken quantitativ beurteilt (begriffsbasierte Techniken werden nicht näher ausgewertet). Zu diesem Zweck wird ein Auswerteschema für Komponentenerkennungstechniken vorgestellt, mit dessen Hilfe die Erkennungsqualität hinsichtlich einer Menge von Referenzkomponenten, die durch Software-Ingenieure manuell ermittelt werden, genau bestimmt werden kann. Unter den bewerteten Techniken befindet sich unsere neue metrikbasierte Technik *Similarity Clustering*. Bei der Auswertung anhand des eingeführten Bewertungsschemas und der von Software-Ingenieuren erkannten Referenzkomponenten für vier C-Systeme mit zusammen ca. 136 KLOC befindet sich *Similarity Clustering* bezüglich seiner Wiederfindungsrate stets unter den besten Techniken; allerdings ist auch eine höhere Anzahl unzuordenbarer Komponenten als bei anderen Techniken zu verzeichnen. Als Resultat ergibt sich insgesamt, dass keine der automatischen Techniken eine ausreichende Erkennungsqualität aufweisen kann.

Um diesen Mangel auszugleichen, wird eine halbautomatische Methode eingeführt, in der Computer und Mensch bei der Erkennung zusammenwirken. Die Methode wird unterstützt durch eine Integration der vollautomatischen Techniken, bei der die Analysen sukzessive mit anschließender Validierung durch den Benutzer ausgeführt werden können. Hierzu werden die Techniken zu inkrementellen Techniken erweitert. Die Vereinheitlichung der Techniken erlaubt die einheitliche Implementierung inkrementeller Varianten für ganze Klassen von

Techniken. Die Resultate der Techniken können mittels Operatoren kombiniert werden, denen die Mengenoperationen Schnitt, Vereinigung und Differenz für unscharfe Mengen zugrunde liegen. Eine alternative Art der Integration ist der sogenannte Abstimmungsansatz, bei dem die individuellen Zustimmungen der Techniken zusammengefasst werden.

Trotz der neuen, mächtigen Möglichkeiten, die automatischen Techniken zu kombinieren, wird die halbautomatische Methode durch Schwächen der einbezogenen automatischen Techniken beeinträchtigt. Zukünftige Forschung sollte untersuchen, ob die Schwächen der Techniken mit präziseren Informationen, die durch Datenflussanalysen hergeleitet werden können oder sich durch Vorwissen über das Anwendungsgebiet ergeben, beseitigt werden können. Nichtsdestotrotz werden auch zukünftige Verbesserungen stets mit der Vagheit und teilweisen Subjektivität der Gruppierungskriterien für Komponenten konfrontiert sein.

Part I Prelude

Chapter 1 Introduction

In 1985, Lehman and Belady stated the so-called Lehman's laws. Out of the original five the two "laws" (hypotheses, really) most relevant to the context of this thesis are repeated here:

> (1) **The law of continuing change**: A program that is used in a real-world environment necessarily must change or become progressively less useful in that environment.
>
> (2) **The law of increasing complexity**: As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.

The work presented in this thesis aims at methods and tools to preserve and simplify the structure of a system in order to support program evolution. With **program evolution**, any modification of a software product is meant that takes place after delivery to correct faults, to improve performance or other attributes, to adapt the product to a changed environment, or to add functionality.

1.1 State of the Practice

Software is an increasingly important factor for the expenses and returns of marketed products not only within the traditionally software-dominated domains, such as telecommunication and information systems, but also in other technol-

ogy-oriented lines, such as mechanical engineering, aviation, astronautics, or entertainment industry, whose share of software in production costs is estimated to 30-50 percent. The average fortune-100 company has 35 millions lines of code in operation with a growth of 10 percent per year (Buss et al., 1994).

It is known from diverse case studies that 60-80% of the costs of a software product arise for program evolution (Nosek and Palvia, 1990). Interestingly enough, industry and research have made surprisingly little effort to cope with the problems of program evolution in comparison to the attention experienced by development of new systems. The “year 2000 problem” has put program evolution in the limelight. However, even this example of a mass change has not changed the situation very much (McCabe, 1998).

More than 50% of the time needed for program evolution is spent in understanding the program before the actual change can be designed and realized, as several case studies have shown (Fjeldstadt and Hamlen, 1984). This is because the necessary information for the task at hand is often not completely and correctly documented and therefore has to be derived from the source code. The maintainers, being badly informed and pressed for time, tend to fix the problem only locally, mostly in those subsystems they are familiar with. These local code fixes often disregard the original design and – since they are no real solutions but treat the problem only phenotypically – provoke errors at other sites of the system and complicate future understanding. This is a vicious circle that ends in a non-maintainable system unless preventive measures are taken.

1.2 The Importance of Software Architecture

Large systems are divided into subsystems. These subsystems, also known as components, and the dependencies that exist among the components form the software architecture of a system. The software architecture is a key asset affecting most attributes of a system. An inappropriate or deteriorated architecture can have a disastrous effect on maintainability. Garlan and Perry describe the major impacts of a software architecture on the following aspects of a system with a focus on development of new systems rather than maintenance (1995). I will describe these aspects more from the maintainer’s point of view.

Understandability. The software architecture provides an overview of a system at a higher level of abstraction. This overview exposes the high-level constraints on system design that a maintainer has to observe and allows a more focused search oriented at architectural information. Many original design decisions and the consequences of their disregard become only clear at this level.

Reuse. In the architecture, the maintainer can not only identify the reusable components but also the existing dependencies to other parts of the system that need to be handled before the components can be reused. Current work on reuse generally focuses on component libraries. Architectural design supports, in addition, both reuse of large components and also frameworks into which components can be integrated. Architecture recovery is also an enabling technology for the product line approach in which common parts of architectures of a family of systems are united and generalized into a generic architectural framework for a particular domain; the architectures of the actual systems in this domain can then be realized as instantiations of the general framework (Bayer et al. 1999).

Evolution. The software architecture can be viewed as the skeleton of the system. Having a description of this skeleton enables the maintainer to identify load-bearing and potentially weak parts that need to be carefully addressed when a system is to be evolved. Furthermore, having a clear picture of a component's dependencies allows one to modify the component itself without affecting other parts of the system or to change the dependencies in order to handle evolving concerns about performance, interoperability, and reuse. Errors no longer have to be fixed where they appear but where they were caused by identifying the responsible components or the undocumented dependencies and constraints.

Analysis. If the recovered architecture is specified by a separate architectural description, new opportunities for analyses are provided, including high-level forms of system consistency, conformance to an architectural style, conformance to quality attributes, and domain-specific analyses for architectures that conform to a specific style. Furthermore, the architectural description can be used to check whether changes to the system conform to the design principles of the architecture.

Management. Maintenance assignments can be made on the basis of subsystems. Moreover, the software architecture provides a base for a more rigorous estimation of the costs and risks of a change. The quality of a system can be assessed by judging the load-bearing capacity of its architecture. Weak parts can be identified and measures to overcome these weaknesses can be better examined and aimed. For particularly problematic components, it can be decided whether they should be reengineered or newly developed. Reengineering of large systems is only feasible if it is done subsystem by subsystem. For this incremental migration, the dependencies have to be known and the wrapping of the not yet reengineered parts has to be planned.

Since all these factors are essential for a system's capability to evolve, a description of the software architecture must be recovered when it is lost. Ideally, the documentation should be kept up-to-date with future changes once it was recovered and the need for recovery should never arise again. However, even then it might be necessary to inspect the architecture as built in order to recognize and analyze differences from the documented architecture. Furthermore, the maintainer may need to explore the architecture as built when the higher-level description abstracted from certain details.

Recovering the software architecture and exploring the architecture as built is costly and the only available tool support in practice is far too often a symbolic debugger to trace the system at a very low level.

1.3 Research in Architecture Recovery

Architecture recovery comprises detection of **components** (the computational parts) and **connectors** (the means and points of communication) of systems. It is aimed at supporting the process of program understanding for software maintenance and evolution.

Component recovery. One major research topic in component recovery is detection of subsystems (Schwanke, 1991), another one is recovery of objects and abstract data types. Though abstract data type and object detection is commonly

driven by reuse or object-oriented system migration, it does support architecture recovery at a lower level of components.

Abstract data types and objects consist solely of subprograms, types, and global variables. They are only two examples of architectural concepts we can form with these kinds of base elements. Other examples are sets of related subprograms or hybrid components. We will refer to such low-level components solely built from types, variables, and subprograms as **atomic components**.

Connector recovery. Connectors for concurrent and distributed systems have been the primary target of connector recovery (Harris et al., 1995; Fiutem et al., 1996). However, most systems, especially legacy systems, are sequential and monolithic. Function call is the most primitive and dominating type of connector of such systems. Another common way of communication is via shared global variables. At the next higher level of connectors, we find atomic components. For example, two architectural components may communicate by means of a pipe where the pipe is implemented as an abstract data type. That is, atomic components can be connectors at a higher level of architecture. Detecting them can therefore also aid in understanding of how larger components communicate.

The goal of our research is to find techniques and methods for atomic component detection in the general framework of architecture recovery. In a case study, we have evaluated several published approaches to detect abstract data types and objects (Girard, Koschke, Schied, 1997c). The overall result was that none of the techniques has the needed precision. There are several alternative approaches to overcome this: The techniques can be combined, other sources of information can be considered (for example, dataflow information or domain knowledge), or the user can be integrated into the search. This thesis proposes a semi-automatic method in which computer and maintainer work hand in hand to detect atomic components. Within this interactive framework, the techniques can be combined by simple operations triggered by the user. Due to the complexity, vagueness, and to some degree subjectivity, it is questionable whether we can ever find precise techniques that fit all cases. Therefore, atomic component recovery is a problem that has to be tackled in concert with a maintainer anyway. Hence, how this can be effectively achieved should be investigated first before we search for other sources of information.

1.4 Scientific Questions Addressed in this Dissertation

In more detail, the following questions are going to be addressed in this dissertation (the respective chapters devoted to these questions are given in brackets):

1. What published structural techniques exist and how can they be unified and classified (Chapter 5)?
2. What is the recall rate and precision in atomic component detection of published techniques (Chapter 6)?
3. How can these techniques be improved individually (Chapter 5 and Chapter 7)?
4. How can these techniques be combined (Chapter 8)?
5. How can the user be integrated in atomic component detection (Chapter 9)?
6. Do automatic techniques support a maintainer in atomic component detection (Section 10.3)?
7. Are the techniques and methods for atomic component detection discussed in this work also helpful for other typical maintenance tasks (Section 10.4)?

This thesis focuses on techniques for atomic component detection that leverage only structural information and investigates how far we can get with such methods. Other potential sources of information are the results of control and data flow analyses and domain knowledge. This thesis does not deal with control and data flow analyses, but in the course of the thesis, I will point out where information derived from these analyses could support the structure-oriented methods. Domain knowledge comes into play by the maintainer within the interactive scenario, but automatic ways to leverage domain knowledge are not explored. Still, I give at least one hint on how one of the approaches could profit by the vocabulary of a domain (Section 7.3.5).

1.5 Project Context

The work described in this thesis is embedded in the Bauhaus project. **Bauhaus** is a research collaboration between the Institute for Computer Science of the University of Stuttgart (ICS) and the Fraunhofer Institute for Experimental Software Engineering in Kaiserslautern (IESE). The goal of Bauhaus is to find methods and techniques for architecture recovery, to explore languages to describe recov-

ered architectures, and to investigate analyses to compare architectures as built to the specified architectures.

The first step toward the general goal of Bauhaus was to investigate methods and techniques to recover the architecture based on structural information. The first researchers of this project were Jean-François Girard (IESE), Georg Schied (ICS), and I (ICS). Until mid of 1998, the three of us worked in close co-operation in the field of atomic component detection. All the work of this period was jointly published. When Georg Schied left the Bauhaus project in March 1998, our teamwork was reorganized. Jean-François Girard has concentrated on detection of subsystems since then. This is the reason why my work does not deal with subsystem detection. However, the combinations of the basic techniques that I propose in Chapter 8 are designed so that they are going to work in the presence of recovered subsystems.

My work has continued the detection of atomic components by exploring possible combinations of the basic techniques, evaluating and inventing new techniques, providing incremental variants of the basic techniques, and delving into ways to integrate the user in the process of atomic component detection. To give a complete picture of atomic component detection based on structural information, this thesis does not only report on new improvements since March 1998 but also on previous joint work with Jean-François Girard and Georg Schied that I will explicitly point out in the following.

1.6 Overview of the Following Chapters

This thesis consists of two main parts. The first part deals with automatic techniques and the second part with a semi-automatic method for atomic component detection. Before we get to these two main parts, the terminology and concepts used in this thesis are introduced in Chapter 2 and Chapter 3 and the issues related to the target language are discussed in Chapter 4.

The first main part begins with Chapter 5 that describes published techniques for atomic component detection and suggests individual improvements to these techniques. These techniques are evaluated in Chapter 6. We, namely, Jean-François

Girard, Georg Schied, and I, extended one of the basic techniques described in Chapter 5 in so many ways that it is presented as a technique of its own in Chapter 7.

The second main part proposes ways to combine the basic techniques and shows how the techniques can be modified to work incrementally (Chapter 8). Then it presents a method in which the maintainer uses the incremental versions of the basic techniques to detect atomic components (Chapter 9) and describes a controlled experiment and a case study conducted to evaluate the method (Chapter 10).

The last part discusses related research (Chapter 11) and summarizes the conclusions of this thesis (Chapter 12).

Chapter 2 Terminology

This chapter introduces the terminology used throughout this thesis.

2.1 Reengineering and Software Maintenance Terminology

The following three sections contain standard terminology in reengineering. The definitions for reverse engineering, restructuring, and reengineering were proposed by Cross and Chikofsky (1990). Figure 2-1 sketches the relationships between these terms graphically.

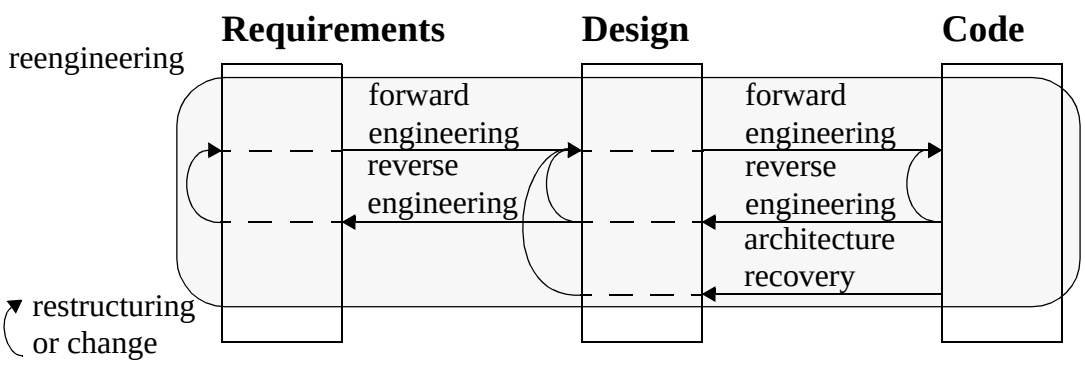


Figure 2-1. **Relationships between terms.**

2.1.1 Forward Engineering

Software engineering was primarily thought of as aiming at the development of new systems though it covers reverse engineering and reengineering as well. To

avoid the connotations of the term software engineering, the term forward engineering is introduced. *Forward engineering* is the process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.

2.1.2 Reverse Engineering

Reverse engineering has the reversed objective of forward engineering. *Reverse engineering* is the process of analyzing a subject system to

- identify the system‘s components and their interrelationships and
- create representations of the system in another form or at a higher level of abstraction.

It is important to note that reverse engineering in and of itself does not involve changing the subject system or creating a new system based on the reverse-engineered subject system. It is a process of examination, not a process of change or replication.

2.1.3 Restructuring

Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system’s external behavior (functionality and semantics). Restructuring is often used as a form of preventive maintenance to improve the physical state of the subject system with respect to some preferred standard.

2.1.4 Reengineering

Reengineering, also known as renovation and reclamation, is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. Reengineering generally includes some form of reverse engineering (to achieve a more abstract description) followed by some form of forward engineering or restructuring.

2.1.5 Program Evolution and Software Maintenance

Reengineering is often seen as part of software maintenance. However, the ANSI/IEEE standard 729-1983 defines **software maintenance** as the "modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment" whereas the goal of reengineering is often to add new functionality to the system, which is not covered by the definition of software maintenance if one interprets its definition narrowly. One may argue that "adapting the product to a changed environment" includes adding new functionality, but as Turski (1981) pointed out it would be a gross abuse of the term *maintenance*: The addition of a new wing to a building would never be described as maintaining that building. That is why I consider adding new functionality as program evolution but not as maintenance. Reengineering is therefore a part of program evolution.

As opposed to reengineering, reverse engineering may in fact be viewed as an activity within software maintenance since its purpose is to recover information that can be used for software maintenance tasks and it does not imply any change.

2.2 Terminology in Software Architecture

There are still debates about the definition of software architecture, but most agree that it should include at least *components* and *connectors* and their hierarchical decomposition. Components are the computational parts and connectors describe the interactions between these components (Garlan and Shaw, 1993; Perry and Wolf, 1992). General examples for components are abstract data types, producer and consumer tasks, or a compiler front end; examples for connectors are procedure calls, shared global variables, pipes, or Unix sockets.

2.3 Architecture Recovery

Architecture recovery is a discipline of reverse engineering that is aimed at recovering the software architecture of a system. It has to be demarcated from design recovery. The term *design recovery* has been introduced by Ted Biggerstaff

(1989) and is the process of recreating design abstractions from a combination of code, existing design documentation, personal experience, and general knowledge about problem and application domains. Biggerstaff argues that design recovery in the broad sense is so inherently unstructured and unpredictable that formal deduction alone is not sufficient and, therefore, fuzzy reasoning should be used as additional way of deriving information. The derived information is used to populate a domain model that is used to understand the software.

Design recovery is distinguished by the sources and span of information it should handle. As Biggerstaff says:

> "*The domain model differentiates design recovery research from such superficially similar efforts as reverse engineering, which automatically abstracts code to a specification level such that the specifications can be modified and revised code can be automatically regenerated. (Biggerstaff, 1989)"*

It is unclear what definition of reverse engineering Biggerstaff had in mind when he wrote this. The definition of reverse engineering by Chikofsky and Cross (Section 2.1.2) would cover design recovery as well, but it came after Biggerstaff's definition of design recovery. Furthermore, the term *design recovery* only suggests that the design is to be recovered in the true sense of the word, whereas Biggerstaff's definition explicitly requires a domain model, which narrows the term unnecessarily. That is why I prefer the more neutral term *architecture recovery.*

2.4 Embedding this Work into Architecture Recovery

This section will present a general framework of architecture recovery proposed by Kazman et al. to accommodate analytical and transformational processes in architecture recovery. Kazman et al.'s model is going to be refined towards a more conceptual framework that is used to show where the work described in this thesis fits in.

2.4.1 The Horseshoe Model of Architecture Recovery

Kazman et al. present a framework that can accommodate analysis and transformation processes in architecture recovery (1998). This framework is called the **horseshoe model** and consists of four different levels:

- **source level**: source code in textual representation
- **code structure level**: the source code in an intermediate representation that enables syntax-aware analyses
- **function level**: relationships among functions, data, and modules, providing a global system overview
- **architectural level**: architectural elements, i.e., connectors and components

Architecture recovery usually starts at the source level. The source code is parsed, syntactically and semantically analyzed, and then represented in some intermediate representation at the code structure level (often by abstract syntax trees). This intermediate representation is further processed by control and data flow analyses. The result is basic information that can be used for deriving the software architecture. However, the represented elements at this level, i.e., the declarations, statements, and expressions of a system and their relationships do not appear in the architectural description (except for global declarations) since this description should give a more abstract overview of the system. Yet, a complete architectural description provides a mapping between the architectural concepts and the implementing statements and expressions.

From the architectural perspective, source and code structure level can be merged into a single **code level**. We can omit a discussion of the code level for the purpose of this thesis. More about suitable intermediate representations for reverse engineering can be found in a separate paper by us (Koschke, Girard, Würthner, 1998).

An architectural description typically suppresses statements, expressions, and local declarations present at code level, but at least global declarations of constants, variables, functions, and user-defined types turn up in such a description since they are architecturally relevant. Constants are used to specify specific aspects permanently while global variables represent state that can change over time by modifying the value of the variables. Global variables can also act as connectors. Global functions are primitive components and user-defined types corre-

spond to higher concepts either of the programming domain (stacks, lists, etc.) or application domain (deposit, person, etc.).

We will refer to these four kinds of entities (global constants, variables, functions, and user-defined types) as **architectural quarks** because they are the building blocks of architectural elements. Conceptually, they belong to both the code level and the architectural level. They form the seam between these two levels, so-to-speak. This seam is exalted in the horseshoe model as the function level.

According to Kazman et al., the function level also shows how the architectural quarks are grouped into modules. Actually, Kazman et al. used in their paper the term *file* instead, but we may assume that they had module in mind and were thinking of older languages in which files are used as a substitute for modules.

Above the function level is the architectural level that consists of the components and connectors of the software architecture.

2.4.2 A Revised Conceptual Model of Architecture Recovery

The horseshoe model is meant as a framework for analytical and transformational processes in architecture recovery and architecture-based development. In this context, it makes sense to distinguish between source and code structure level. At the source level, textual pattern matching based on regular expressions is the only kind of possible analysis whereas at the code structure level syntactic, semantic as well as control and dataflow analyses can be applied. However, there is conceptually no difference between the source and code structure level since the latter is more or less an exact representation of the source code. In a conceptual model, we do not distinguish these two levels.

Any kind of grouping of architectural elements, including that of architectural quarks, is an architecture property. Module organization is one kind of grouping and, therefore, belongs to the architecture domain rather than to the function level as in the horseshoe model. If we exclude module organization from the function level in the horseshoe model, only architectural quarks remain. These, however, also belong to the code level. The function level is therefore rather the seam between the code level and the architectural level than a level of its own. In the revised model, the function level is omitted.

Because of the problems of the horseshoe model just described, I propose a revised conceptual model that consists of the following levels:

- **lower code level**; expressions and statements in function bodies as well as nested functions
- **global code level**; architectural quarks: global constants, global variables, global functions, and user-defined types as well as the relationships among them
- **lower architectural level**; groupings of architectural quarks
- **higher architectural levels**; subsystems and connectors

Above the lower architectural level, there can be several higher levels representing the architecture at different levels of abstraction; the connections among these levels display the hierarchical decomposition of the architectural elements. Both components and connectors can be hierarchical and what is viewed as a component at one level may be a connector at the next higher level.

This thesis is about recovery of smaller groupings at the lower level of architecture that consist solely of architectural quarks, i.e., elements at the global code level. Such groupings will be called **atomic architectural components**, or simply **atomic components** in the following. They are atomic in the sense that they do not consist of further groupings but only and directly of architectural quarks. Atomic components are therefore the smallest components at the architectural level (besides functions which can also - under some circumstances - be considered components). This merely structural definition will be refined in the following section.

Atomic components may be building blocks for larger architectural components at the next higher architectural level. For example, in a case study, we have used dominance analysis to detect subsystems based on atomic components (Girard and Koschke, 1997a). Furthermore, some of the atomic components may even play the role of connectors at a higher level of abstraction, e.g., an abstract data type *Queue* can be used as a pipe between two components. This way, atomic components can be the starting point for detection of connectors and larger components.

2.5 Components

Large systems are decomposed into subsystems that can be managed individually. These subsystems can be again decomposed into smaller subsystems. The smallest decomposition is a **module** that may consist solely of functions, subprograms, and type declarations whereas a **subsystem** is a grouping of modules or lower-level subsystems. Subsystems and modules are **static architectural components** that differ in the degree of granularity. **Dynamic architectural components** are instances of computational units that are created at runtime; e.g., concurrent tasks (with an own thread of control) or queues (without own thread of control). This thesis is about static components only. However, recognizing static components is often a prerequisite for finding dynamic components since the latter are often just instances of the former, such as a queue *X* created on the heap at runtime that is an instance of an abstract data type *Queue* implemented by a static component.

Good design results in a decomposition in which modules, as well as subsystems, have high cohesion and low coupling. The **cohesion** of a module is the extent to which its individual components are needed to perform the same task (Fenton and Pfleeger, 1997). **Coupling** is the degree of interdependence between modules (Yourdon and Constantin, 1979).

There is no standard definition of a module. Yourdon and Constantin (1979), for example, propose the following definition:

> *A **module** is a contiguous sequence of program statements, bounded by boundary elements, having an aggregate identifier.*

This definition, stated in the late seventies when structured design was the proposed design method, sounds nowadays very much like the definition of a function. Today, we have programming languages that support the concept of module. An example is Modula-2 that does not only have the word *module* as keyword in its syntax but also in its name (Wirth, 1985). A module in modern programming languages is a syntactic unit that supports encapsulation. It consists of an interface of the exported parts and an optional hidden implementation. The exported elements are global constants, variables, subprograms, user-defined types, and sometimes nested modules.

In its first design, a system's decomposition may be indeed so that modules reveal low coupling and high cohesion (see Parnas, 1972, on the criteria of modularization, and Parnas et al., 1985, on the modular structure of complex systems), but during continuous maintenance the original decomposition may deteriorate. For example, a function *F* that actually would have belonged to module *A* was put into module *B*. This results in lower cohesion of module *B* and in higher coupling between *A* and *B*, since *F* will need details of the implementation of *A*. Furthermore, the underlying concept of *A* is delocalized because it is also partly realized by *B*. High coupling and low cohesion make changes more difficult. Reengineering has to restructure the system such that the underlying concept of *A* is implemented by module *A* and only by *A* to simplify future maintenance.

The discussion reveals that a real module does not always match its underlying concept, i.e., there is a divergence between the syntactic unit and the logical unit. To distinguish these two kinds of units, we will call the latter **atomic component**. With *module*, we solely mean the syntactic unit further on and follow the typical programming language terminology in doing so.

> A **module** is a syntactic unit that is used to group entities. It consists of an interface and an optional implementation. Entities in its interface are accessible by other modules; the implementation is the module's secret.
>
> A **component** is a group of related elements with a unifying common goal or concept relevant at the architectural level. An **atomic component** is a *non-hierarchical* component that consists of related global constants, variables, subprograms, and/or user-defined types. As opposed to an atomic component, a **subsystem** is a *hierarchical* component consisting of related atomic components and/or lower-level subsystems.

The goal of (re-)structuring a system is to realize an atomic component by one module and one module implements only one atomic component for the sake of maximal cohesion and minimal coupling. In practice, the degree of cohesion of a module can vary. The next section discusses this in more detail.

2.5.1 Cohesion of Modules

Yourdon and Constantin (1979) list the following degrees of cohesion:

- **Functional**: the module performs a single well-defined function.
- **Sequential**: the module performs more than one function, but they occur in an order prescribed by the specification.
- **Communicational**: the module performs multiple functions, but all on the same body of data (which is not organized as a single type or structure).
- **Procedural**: the module performs more than one function, and they are related only to a general procedure affected by the software.
- **Temporal**: the module performs more than one function, and they are related only by the fact that they must occur within the same time span.
- **Logical**: the module performs more than one function, and they are related only logically.
- **Coincidental**: the module performs more than one function, and they are unrelated.

These categories of cohesion are listed from most desirable (functional) to least desirable. This classification was established in the late seventies when the functional paradigm was dominating and structural design was the common design method. The more recent trend towards the object-oriented paradigm - and hence to languages and methods that support data abstraction by modules - at first sight appears to contradict the traditional ideas. A module based on data abstraction may perform several different functions; but all are related in the sense that they characterize the abstract data type, or more generally: the atomic component.

Modules based on data abstraction form a special category, and Macro and Buxton (1987) have extended the cohesion classification to include it. They say, a module has **abstract cohesion** precisely when it is an abstract data type. We can generalize this to: “A module has abstract cohesion precisely when it is an atomic component.” Good design and restructuring should aim at abstractly cohesive modules.

2.5.2 Recovered Atomic Components

This section answers the questions raised by the new definition of abstract cohesion: What exactly is an abstract data type and what are the other kinds of atomic components recovered by the approaches described in this thesis?

2.5.2.1 Abstract Data Types

Liskov and Zilles define an **abstract data type** (**ADT**) as an abstraction of a type which encapsulates all the type's valid operations and hides the details of the implementation of those operations by providing access to instances of such a type exclusively through a well defined set of operations (1974).

Abstract data types may be constrained by global constants. For example, the maximal length of a list may be determined by a constant specific to this type. Such constants are an integral part of the ADT. Furthermore, the implementation of two or more types is sometimes so much interleaved that the types cannot really be separated into distinct ADTs, e.g., a hash table and a hash table entry type. That is to say, an ADT does not necessarily consist of one type only. To sum it up, ADTs consist of a set of types (usually, only one type) and their accessor functions; global constants may also belong to the ADT if they specify aspects of the ADT.

In a modern programming language supporting encapsulation, the data structure of an ADT can be hidden such that only the subprograms that belong to the ADT may access it. All other subprograms may only declare objects of the ADT and call the accessor routines.

An example ADT in a modern programming language. Figure 2-2 shows the interface of an ADT *stack of integers* in the programming language Ada (ANSI/ISO/IEC-8652:1995). The internal parts of the ADT are explicitly declared private (the full type declaration is given in the private section of the package for the purpose of efficient separate compilation).

Only subprograms declared in this package may access these internal parts (actually, in Ada 95, subprograms declared in child units of this package have also visibility to the private part, but this is not important to the discussion here). The interested reader may learn more about support for data abstraction by modern

programming languages in Robert Sebesta's book on concepts of programming languages (1998).

```
package Stacks is
  type Stack is private; -- The ADT stack of Integer
  -- accessor routines
  function Top (S : Stack) return Integer;
  procedure Push (S : in out Stack; Item : Integer);
  procedure Pop (S : in out Stack);
private
  type Stack_Contents is array (1..1000) of Integer;
  type Stack is record  -- the hidden data structure
            Contents : Stack_Contents;
            Stack_Pointer : Natural := 0;
  end record;
end Stacks;
```

Figure 2-2. **An example ADT *stack of integer* in Ada.**

Abstract data types are related to classes in object-oriented programming languages but whether there is a direct correspondence between the two of them depends on the terminology. We will follow the more proper notion of a class as a *set of types*, whereas many object-oriented programming languages use class synonymously to type, e.g., *SmallTalk* and *C++*. In the proper sense, an abstract data type may be an element of a class but no class as such. I want to point out here that the relationships among the abstract data types, i.e., whether they are members of the same class and how they are derived from each other, are beyond the scope of this thesis.

2.5.2.2 Abstract Data Objects

An **abstract data object** (**ADO**) is a group of global variables and constants together with the routines which access them. These clusters are also called abstract objects (Ghezzi et al. 1991) or object instances (Yeh et al. 1995).

ADOs are used to capture state. The state can be manipulated and queried by the accessor routines of the ADO. No other routine may access the variables since the ADO is considered abstract.

An example of an ADO. Figure 2-3 contains an Ada package that implements an abstract data object *stack of integer.* The package interface lists the accessor routines that can be called by a client. The global variables are hidden from clients in the package body.

```
package Stack is
  -- accessor subprograms
  function  Top return Integer;
  procedure Push (Item : Integer);
  procedure Pop;
end Stack;
package body Stack is
  -- global variables
  Stack_Pointer : Natural := 0;
  Contents      : array (1..1000) of Integer;
  ...
end Stack;
```

Figure 2-3. **An example ADO *stack of integer* in Ada.**

This example illustrates the difference of an ADO and an ADT: An ADT is built around a type. This type can be used to create as many instances of the ADT as needed (either by declaring variables of this type or by using dynamic allocation) whereas there is always one instance of an ADO since a client has no instantiation handle for ADOs and there is only one package *Stack*. (In Ada, one could make the package *Stack* generic and then instantiate it many times to get several ADOs *Stack*; however, we do not have generics in older programming languages.)

An ADO could be generalized to an ADT by simple transformation rules:

- a new record type is introduced that contains a component for each global variable of the ADO
- the new record type is added to the parameter list of each ADO accessor routine
- in the body of an ADO accessor routine, accesses to the global variables are replaced by accesses to the components of the new record type; variables of the new record type are passed as actual parameter to the routine
- at the client site, variables of the new record type must be declared that are then passed as actual parameters in calls to the accessor routines

Because of the simple shift from an ADO to an ADT, an ADO is often considered an ADT. Nevertheless, we will distinguish these two kinds of atomic components since there is a conceptual difference and also because the recovery strategies for these two are different. However, the transformation rules mentioned make clear that ADOs are at the same cohesion level as ADTs.

2.5.2.3 Hybrids: State-based ADTs or ADOs with Subordinated Types

In real programs, we often find mixtures of ADTs and ADOs, i.e., atomic components that contain both types as well as variables. There are two different categories of such mixtures. A **state-based ADT** is an ADT having state information by way of global variables. An example is an ADT that counts in a global variable how many instances are created at runtime. An **ADO with subordinated types** is an ADO that contains types that are an integral part of it. An example is an ADO *hash table* that contains two type declarations, one for *hash items* and one for *lists of hash items* (to resolve external collisions). Both types might be so special to the *hash table* that they cannot be reused in other contexts and are therefore no ADTs of their own.

Whether we deal with a state-based ADT or an ADO with subordinated types is often hard to judge. If the distinction is not important, we will refer to both kinds as **hybrid atomic components**, or short **hybrid components**. A hybrid atomic component has abstract cohesion because it can be considered an ADT or ADO.

2.5.2.4 Strongly Connected Components

Strongly connected components are sets of subprograms that call each other recursively. These subprograms form a component because none of them can be omitted without losing a piece of information for the understanding of the other subprograms in the component. They arise from the call graph; that is why programming languages do not need means to specify strongly connected components explicitly.

Strongly connected components do not have abstract cohesion in the sense of Macro and Buxton’s definition since they consist solely of subprograms and are therefore no abstract data types. What kind of cohesion they actually have depends on the logical function they perform, but we can use structural information at least as a clue: It may be that there is only one entry *E* of the cycle from

outside and thus all other parts of the strongly connected components are subordinated to *E*. This is a strong hint that the strongly connected component performs a single function and has therefore functional cohesion. When there is more than one entry, the strongly connected component has at least logical cohesion since the functions within the component depend on each other: one cannot change or remove a single subprogram without affecting the others in the strongly connected component.

2.5.2.5 Other Kinds of Atomic Components

Another relevant type of atomic component are **sets of logically related subprograms** (short **related subprograms**), as, for example, functions of a mathematical library. A construct that could be used as a point of crystallization, as user-defined types for ADTs and variables for ADOs, does not necessarily exist for logically related subprograms and they are therefore harder to detect, especially when they are not even directly connected to each other, i.e., when they do not call each other.

2.5.2.6 Primary Target Atomic Components

Related subprograms can only be found by some of the techniques (e.g., *Similarity Clustering*, *Type-based Cohesion*), though this is not their primary goal. The main targets of the techniques described in this thesis are abstract data types and abstract data objects. They represent a clear concept and are often used in practice (see Section 6.1.1). Less frequently used but still representing a clear concept are hybrid components. Strongly connected components are also useful for program understanding, but whether they also correspond to a specific concept has to be decided for each single component. For example, it could be that parts of the strongly connected component belong to different, yet related atomic components, or it could be that the strongly connected component is a complete part of an atomic component.

As to strongly connected components, some techniques may also be able to detect clusters that belong to different atomic components, for example, communicating parts of related atomic components, – or even form an atomic component which we have no name for yet – and nevertheless contribute to program understanding. This should be borne in mind when the techniques are judged.

2.5.2.7 Abstractness of Atomic Components

The definitions of abstract data types, abstract data objects, and hybrid components provided above describe the ideal situation in which programmers would always be aware and respect the encapsulation of these atomic components. In practice, in languages like C, there is only limited support to hide the implementation details of atomic components and even in modern programming languages in which means to hide implementation details exist, software developers do not always use these means. As a result, the encapsulation of atomic components is often violated by direct accesses which bypass the accessor functions of the atomic components.

In general, we use the adjective **pure** in front of an atomic component to denote that all accesses to its internal parts proceed through its interface and the adjective **permissive** in front of atomic components which suffer from encapsulation violation. We use the convention that, when no adjective is in front of an atomic component, it is a permissive atomic component. This convention was selected because permissive components are much more frequent than pure components among the atomic components identified by a group of software engineers that analyzed several systems manually for our evaluation. Actually, their task involved deciding which function accessing internal elements of a potential atomic component were part of the abstraction and which were not.

The presence of these encapsulation violations should be taken into consideration by reverse engineering techniques which attempt to identify atomic components in an automatic or semi-automatic fashion. In other words, the techniques and methods described in this thesis are aimed at identifying permissive components. These permissive components can then be encapsulated by further automatic program transformations if necessary. Such program transformations are, for example, described by Fanta and Rajlich (1999) but are not further discussed in this thesis.

Chapter 3 Basic Structural Information

As stated in the introduction, the techniques described in this thesis are mainly based on structural information that is directly derivable from source code. This chapter presents the exploited structural information in a programming-language independent manner. This chapter also introduces the means to describe the techniques in a detailed way. The next chapter shows how the abstract model introduced in this chapter can be instantiated for the programming language C.

3.1 Base Entities and Relationships

The leveraged basic information can be described by an entity relationship model whose entities are the smallest significant elements at the architectural level, namely, the architectural quarks: user-defined types, global subprograms, variables and constants. This section describes the relationships among architectural quarks used to detect atomic components. These relationships can be found in most procedural programming languages. How these relationships are derived from C code is described in Chapter 4 and how they are actually used by heuristics for atomic component detection is described in Chapters 5 and 7.

The entity-relationship model used to describe the information leveraged for component recovery makes use of inheritance for both entities and relationships. The model will be extended successively in the following sections both in terms of additional entities and relationships as well as refinement of existing entities

and relationship by means of inheritance. A summary of the final model can be found in Appendix A.

3.1.1 Architectural Quarks and their Relationships

The architectural quarks are summarized by Figure 3-1. Variables and constants are subsumed by an abstract class *object*. (An **abstract class** is a class of which no instances exist and is used to express common properties of its derived classes. Abstract classes will be printed in *italic* in the following inheritance hierarchies.) An *object* should not be confused with an *abstract data object*. An abstract data object is an atomic component that contains objects (variables and constants). Neither should it be confused with an "object" in the sense of object-oriented programming.

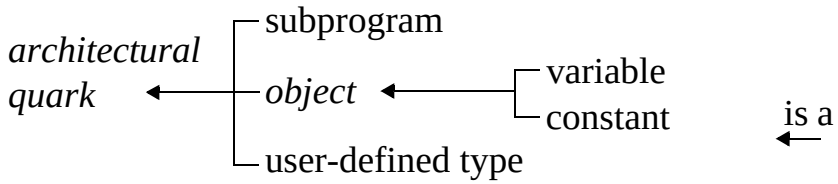


Figure 3-1. **Architectural quarks type hierarchy.**

Figure 3-2 shows the relationships among architectural quarks leveraged by the techniques for atomic components detection as an entity-relationship model. The relationships and the respective roles of the involved entities are listed in Table 3-1. This entity-relationship model is going to be refined in Section 3.1.2.

Two of the relationships of Figure 3-2 can be refined: The *signature-type* of a subprogram is a type that occurs in its signature either as return or parameter type and an object can be *referenced* by using its value or taking its address; a variable can additionally be set. Altogether, we have the relationships summarized by Figure 3-3.

We will define the relationships in Figure 3-3 in the context of the language our analyses are aiming at in Chapter 4; a brief explanation can be found in Table 3-1. Most relationships in Table 3-1 should be fairly self-explanatory, except for the *part type* relationship, which will be explained here.

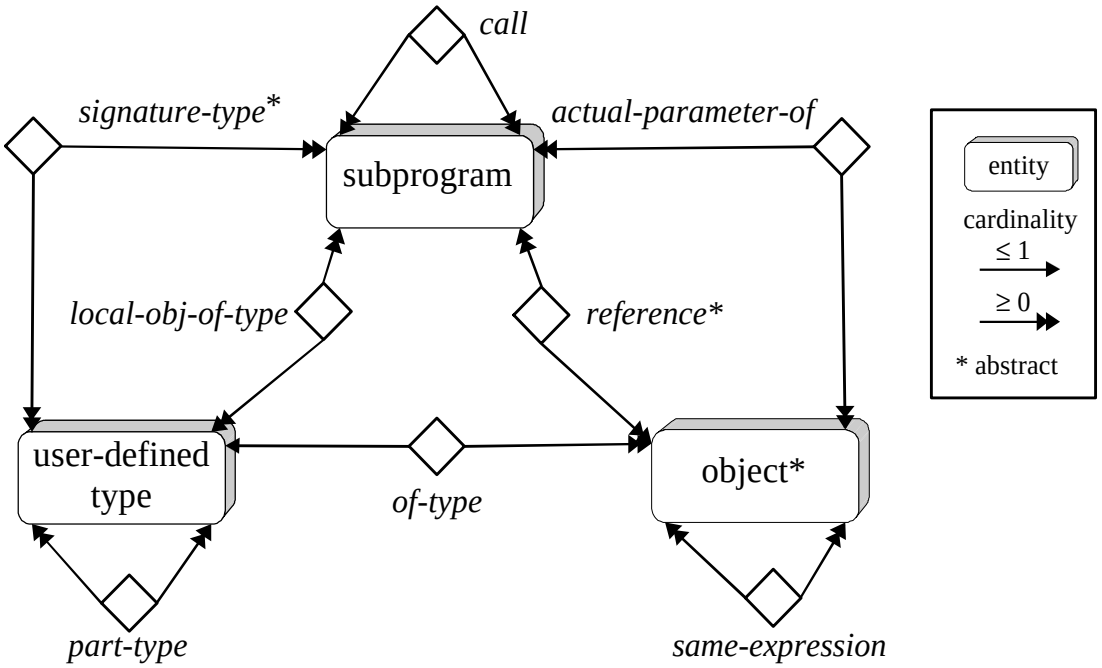


Figure 3-2. **Entity relationship model for architectural quarks.**

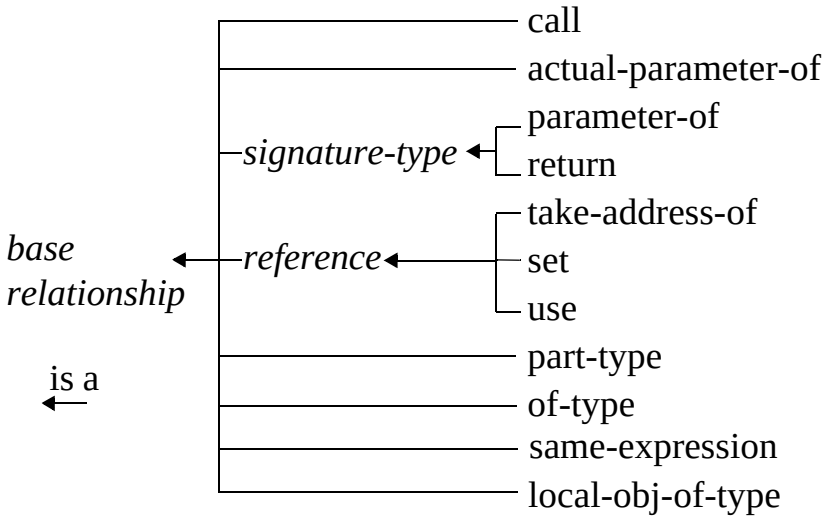


Figure 3-3. **Relationship type hierarchy.**

A type *T1* can be used in the declaration of another type *T2*. In this case, we consider *T1* a **part-type** of *T2* (Ogando et al. 1994). *T2* is the **composite type** of *T1*. For example, in the following C type declarations, *Item* is a part type of *Node*.

Table 3-1. **Relationships among architectural quarks.**

Relationship	Source S	Target T	Meaning
call	subprogram	subprogram	S *calls* T
set	subprogram	global variable	S *sets* the value of T
use	subprogram	object	S *uses* the value of T
take-address-of	subprogram	object	S *takes* the *address of* T
parameter-of	subprogram	user-defined type	S *has* a *formal parameter* of T
return	subprogram	user-defined type	S *returns* a value of T
local-obj-of-type	subprogram	user-defined type	S *has a local object of type* T
actual-parameter-of	object	subprogram	S is an *actual parameter* in a call to T
of-type	object	user-defined type	S is *of type* T
same-expression	object	object	S and T occur in the *same expression*
part-type	type	type	S is a *part type* of T

```
typedef ... Item;
struct Node {Item i; struct Node *next;};
```

The part-type relationship is transitive, i.e. if *T* is a part-type of *S* and *S* is a part-type of *U*, then *T* is also a part-type of *U*.

3.1.2 Record Components

The entity-relationship model in Figure 3-2 contains only the principal constituents that are to be grouped to atomic components by the techniques and the principal relationships between these constituents that are considered for this. Some of the techniques presented in the course of this thesis go beyond these entities and relationships by also considering references to record components of formal parameters, and local and global objects. That is why we enhance the entity-relationship model in Figure 3-2 by explicitly modeling record components.

Record Types. The syntax for record type declarations may vary for diverse programming languages but their essence is to specify the record components of a user-defined record type and their respective types. For example, the following C record declarations:

```
struct Complex
  {float re, im;};
struct List
  { struct List *next;
    struct Complex c1;
    struct Complex c2;
  };
```

define two record types *Complex* and *List*. *Complex* has the record components *re* and *im* of type *float*. *List* is a list of pairs of *Complex* and has, therefore, a *List* pointer to the next element in the list, and two record components *c1* and *c2* for the two complex numbers.

Record Objects. Variables and constants of record types are called *record variables* and *record constants*, respectively; both are referred to as *record objects*. As instances of a record type, they comprise all record components of the type they are declared of plus transitively of all types of the record components. This way accesses to record components across multiple levels of record objects are possible. For example, given the following declaration of a record variable in C (corresponding to the declarations of *List* and *Complex* above):

```
struct List mylist;
```

the following record component access is across two levels:

```
mylist.c1.im
```

Modeling Record Components. Each record object has its own separate set of record components (possibly of multiple levels) as specified by the record components of the record type and its part types. That is, we actually have two kinds of record components:

1. **Record component specifier:** A record component within a type declaration, which defines a part of the structure of all instances of this type.

2. **Record component instance:** An actual record component of a record object that is associated with a memory location.

These two kinds of record components are explicitly modeled by extending the entity type hierarchy in Figure 3-1 as shown in Figure 3-4. Record components are separated from architectural quarks in the extended entity type hierarchy because only the latter may be grouped to atomic components — record components, then, always belong to their enclosing type or object, respectively.

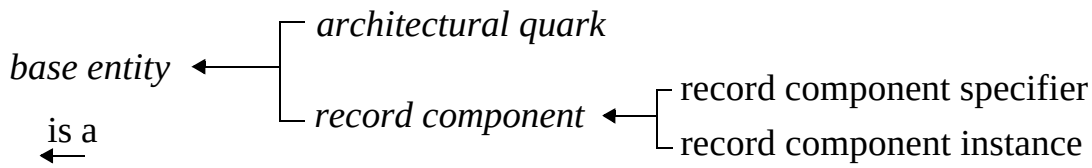


Figure 3-4. **Base entity type hierarchy.**

3.1.2.1 Enclosing Relationship for Record Components

In order to capture where a record component actually belongs to, a new **enclosing** relationship is introduced. If *R* is a record component specifier and *T* is the type in which *R* is declared, *T* is the *enclosing* of *R*. Furthermore, if an object *V* is declared of this type *T*, a new record component instance *R'* is added for *R* whose *enclosing* is *V*. If *R* itself is of a record type *T'*, a record component instance *R"* is added for each record component of *T'* and *R'* is the enclosing of each *R"*. Table 3-2 summarizes the *enclosing* relationship. Note that it is defined for two different domains.

Table 3-2. **Relationships among base entities.**

Relationship	Source S	Target T	Meaning
enclosing	*record component specifier*	*type*	S is *enclosed* by T
enclosing	*record component instance*	*global object* *record component instance*	S is *enclosed* by T

Furnishing each object with its own set of record components allows to distinguish accesses to the same logical record component of different objects and also to make a distinction between an access to a logical record component of a global

object on one hand and to the same logical record component of a local object or parameter on the other hand.

Example. The variable *gl1* in Figure 3-5 has the tree of *record components* as shown on the right hand side of the figure. Note that it inherits the *record components re* and *im* twice because it has two record components of type *Complex*, namely, *c1* and *c2*. Transitively, *mylist* has 7 (partly composite) record components, i.e., there are 7 ways to access the internal components of *mylist*. The type *struct list*, on the other hand, has only three record components reachable by reverse *enclosing* edges.

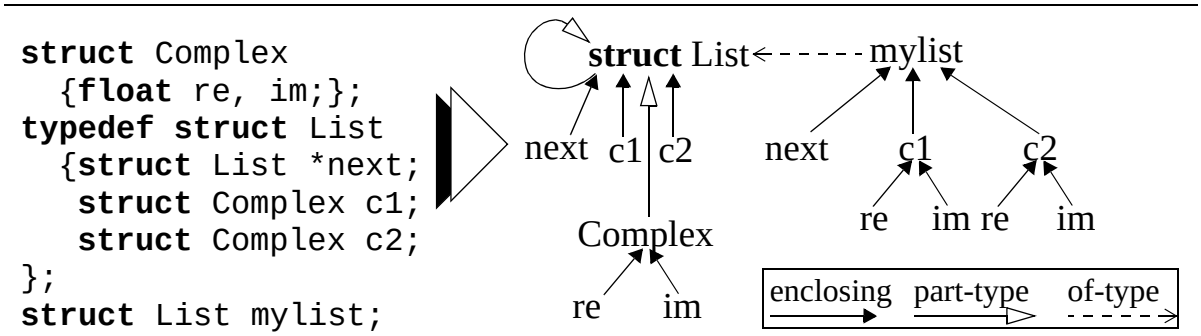


Figure 3-5. **Enclosings of a composite variable.**

With this model, we can distinguish three different record component accesses in the following code as shown in Figure 3-6:

```
struct List gl1, gl2;
foo (struct List pl) { gl1.c1 = gl2.c1 = pl.c1; }
```

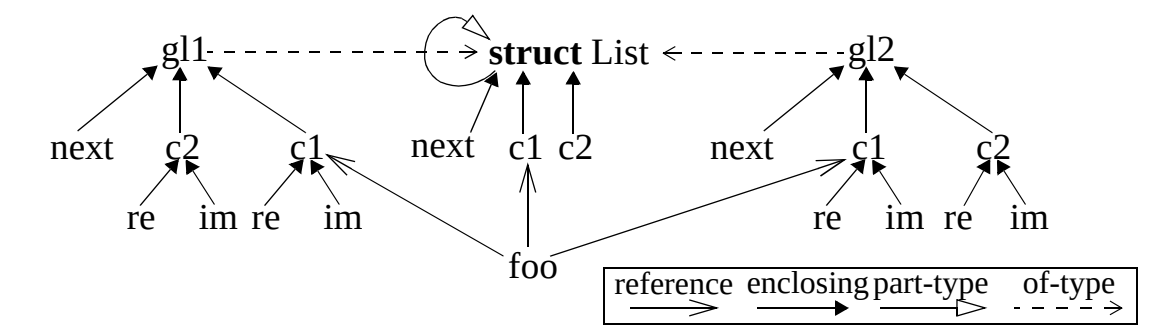


Figure 3-6. **Record component references.**

Note that we do not explicitly model references to local objects and parameters (see Section 3.1.2.3). Note also that Figure 3-6 does not capture the induced references to the global variables *gl1* and *gl2* since we are primarily interested in references to record components in this section. The next two sections describe how references to objects and record components are distinguished.

3.1.2.2 Refining the Reference Relationship for Record Components

There is a substantial difference in an access to a record object as a whole and an access to the record components of such an object: According to the information hiding principle, only accessor routines of the atomic component are allowed to access record components since this requires knowledge of the underlying data structure. In the following example:

```
struct List mylist1, mylist2;
mylist2 = rest (mylist1); /* statement 1 */
mylist2 = mylist1.next;   /* statement 2 */
```

statement 1 may occur within accessor routines of *List* as well as in all other functions whereas statement 2 may only be used within accessor routines according to the information hiding principles.

In order to distinguish references to record components from references to objects as a whole, we refine the *set*, *use*, and *take-address-of* relationships that were introduced as parts of the entity relationship model in Section 3.1.1. The refined model is shown in Figure 3-7. References to an object as a whole are either *obj-address-of, obj-set,* or *obj-use*; references to record components are *comp-address-of, comp-set,* or *comp-use*. A relationship *comp-set (f, c)* has to be understood as "function *f* sets record component *c*". This always implies that *f* also partially sets the enclosing object of *c*, which is explicitly modeled by a corresponding *obj-set* relationship.

In compiler terminology, *comp-set* compares to a partial set. I use another term because, first, the object of a partial set relationship is the composite object and not the record component in compiler terminology and, second, the reference relationship covers only references to record components whereas dereferences of pointers and array subscripts are also considered partial references in compiler terminology.

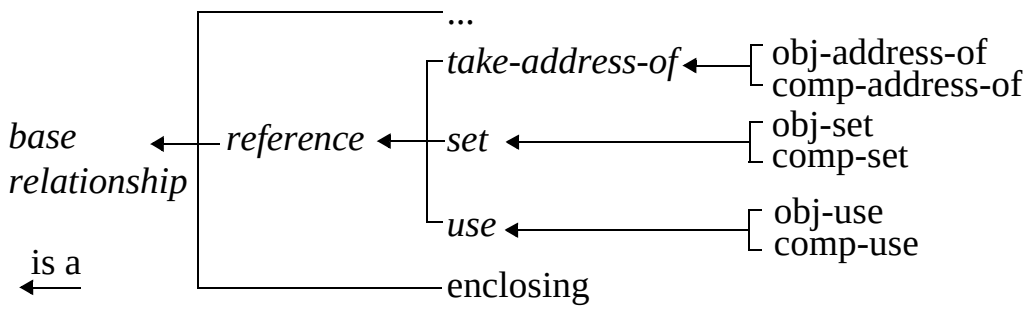


Figure 3-7. **Reference relationships hierarchy.**

Table 3-3 lists the domains and meanings of the new reference relationships. Since it is irrelevant in the context of this thesis to further distinguish references to variables from references to constants, no additional subclasses of object references are introduced in order to simplify the presentation in the following. For reasons of conformity, *obj-set* is used when a global variable is set instead of a more appropriate *var-set*.

Table 3-3. **Reference relationships.**

Relationship	**Source S**	**Target T**	**Meaning**
obj-set	*subprogram*	*global variable*	S *sets* the value of T.
obj-use	*subprogram*	*object*	S *uses* the value of T.
obj-address-of	*subprogram*	*object*	S *takes the address* of T.
comp-set	*subprogram*	*record component*	S *sets* the value of T.
comp-use	*subprogram*	*record component*	S *uses* the value of T.
comp-address-of	*subprogram*	*record component*	S *takes the address* of T.

3.1.2.3 Modeling References

There are basically the following objects that can be referenced by functions: global and local variables, formal parameters, global and local constants (must not be set), and dynamic data structures. Dynamic data structures are anonymous and can therefore not be grouped. However, dynamic data structures are accessible via pointers and these pointers are visible in the source and may be brought in for grouping.

Global variables and constants are explicitly captured by the entity-relationship model proposed in this chapter. References to these can be directly represented by the reference relationships introduced in the previous section, i.e., if a record component of a variable is set, a *comp-set* of the record component instance and an *obj-set* of the variable is added. If the variable is set as a whole, only an *obj-set* of the variable is appended.

Local variables and formal parameters are not captured by the entity-relationship model. They are only used to induce *local-obj-of-type* and *signature-type* relationships. Therefore, references to record components of local variables and formal parameters are re-directed to the respective record component specifier of their type. This may first appear irritating since a record component specifier cannot be referenced — only its instances. However, references to specifiers should be viewed as information how the type is used. This view saves us from representing formal parameters and local variables in the entity-relationship model. Furthermore, if formal parameters and local variables were represented and a function has several parameters of the same type, the information about the usage of the type by the function would be spread over the distinct parameters, though it is not of interest how each single parameter is referenced but only how the type is used altogether.

References to local objects and formal parameters as a whole are not represented as references to their type since the *local-obj-of-type* and *signature-type* relationships are already in place.

Example. Figure 3-8 illustrates the newly introduced concepts. Note that there is no *obj-use* for the parameter *s* because parameters are not explicitly represented.

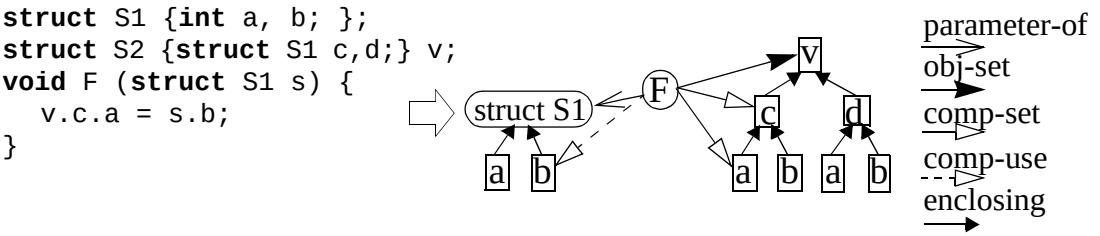


Figure 3-8. **Example representation of record component accesses.**

3.1.3 Entity-Relationship Model for Base Entities

Due to modeling record components, a new entity type and its relationships have been introduced in the previous section. The updated entity-relationship model is given in Figure 3-9.

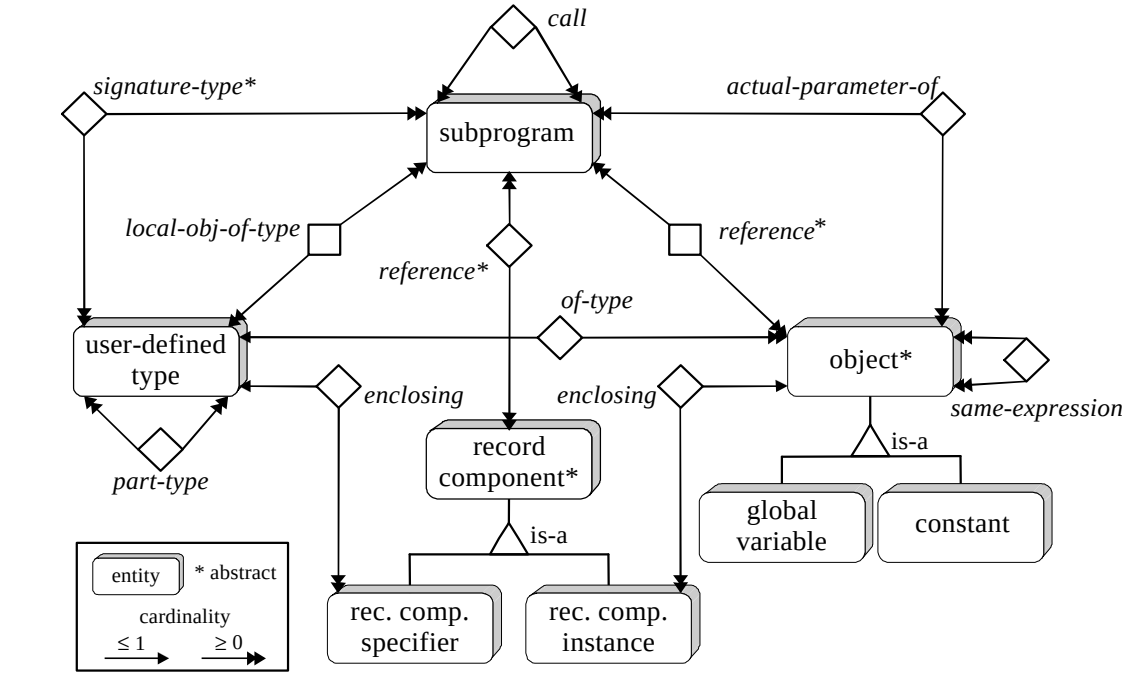


Figure 3-9. **Final base entity-relationship model for component detection.**

It is important to note that record components and the relationships associated with them are only used as additional information for grouping, but that record components themselves are not to be grouped because they always belong to their enclosing object or type, respectively. Nevertheless, in a second analysis once the atomic components have been identified, the constituents of atomic components could themselves be clustered in order to identify cohesive subparts that represent separate interfaces or services of the atomic component. However, this is beyond the scope of this thesis and will not be pursued further.

3.2 Components

As stated in Section 2.5, there are basically two kinds of static components that are to be detected by architecture recovery: Subsystems and atomic components. The two of them differ in their level of granularity: Subsystems may comprise architectural quarks, atomic components, and lower-level subsystems whereas atomic components consist of related global constants, variables, subprograms, and/or user-defined types only. This section discusses how the base entity-relationship model introduced in the last section can be extended to incorporate components.

3.2.1 Atomic Components

An atomic component can be seen as a named set of architectural quarks. In our relational model introduced in the previous section, we can capture this as follows:

- atomic components are represented by a new entity type
- the fact that an entity *E* belongs to an atomic component *AC* is expressed by a *part-of* relationship: *E* is a *part of AC.*

Of course, the *part-of* relationship is equivalent to set membership when an atomic component is regarded as set of architectural quarks. We will use both views in the following depending upon which one is more comprehensible in the given context.

Notation. Graphically, we will picture the two equivalent views as shown in Figure 3-10. The one on the left hand side is the relational view, the other the set view.

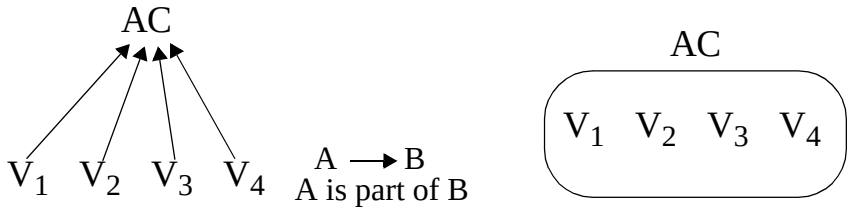


Figure 3-10. **Two equivalent graphical notations for atomic components.**

In the following, identifiers *AC* and AC_i will be used to denote atomic components.

3.2.2 Subsystems

Subsystems are a means to represent hierarchical sets of related elements (architectural quarks, atomic components, and other subsystems) whereas atomic components can be thought of as flat sets of related architectural quarks. Subsystems must contain at least one atomic component — a component with architectural quarks only is considered an atomic component.

We make a clear distinction between a subsystem and its structure. A subsystem as such is an entity in the relational model; the **subsystem structure** is a description of the hierarchical composition of the subsystem. The *part-of* relationship is the spanning relationship for this hierarchy. While an atomic component can only have direct parts, a subsystem may also have indirect parts that are transitively derivable by the *part-of* relationship. Using the *part-of* relationship, one can think of a subsystem structure as a graph whose nodes are the parts of a subsystem and whose edges denote the *part-of* relationship. The root of the graph is a subsystem entity. The graph has the following properties:

- *A subsystem structure is not necessarily a tree*: The *part-of* relationship expresses that an element completely belongs to another element of which it is a part. For a component *C* that is part of another component *C'*, this means that all parts of *C* are also part of *C'*. However, an element can be part of several components, i.e., we allow these hierarchies to overlap since it may not always be definitely clear where an element belongs to.
- *A subsystem structure is acyclic*: Since a subsystem structure should be a real hierarchy, there must not be any cycle in the spanning *part-of* relationship.
- *A subsystem structure does not contain redundant part-of edges in order to be a concise description*: An edge *E* from *A* to *B* is redundant if it is transitively derivable from other edges, in other words, if there is a path from *A* to *B* that does not contain *E*.

The incremental techniques described in Chapter 8 generate subsystems when an entity could be added to more than one existing atomic component (see Section 8.3.1.7). Then, the entity and the atomic components are subsumed under a com-

mon subsystem. Furthermore, the user is allowed to add subsystems. Other than that, this thesis discusses no techniques aimed at detecting subsystems since it is concentrating on the lower level of architecture (as discussed in Section 1.5, detection of subsystems is subject of another thesis within the Bauhaus project). However, taking subsystems into account in the detection of atomic components allows for a future integration with techniques targeted at detecting subsystems.

Notation. The graphical notation introduced for atomic components can be naturally extended to subsystems as exemplified by Figure 3-11. The node *C* of the

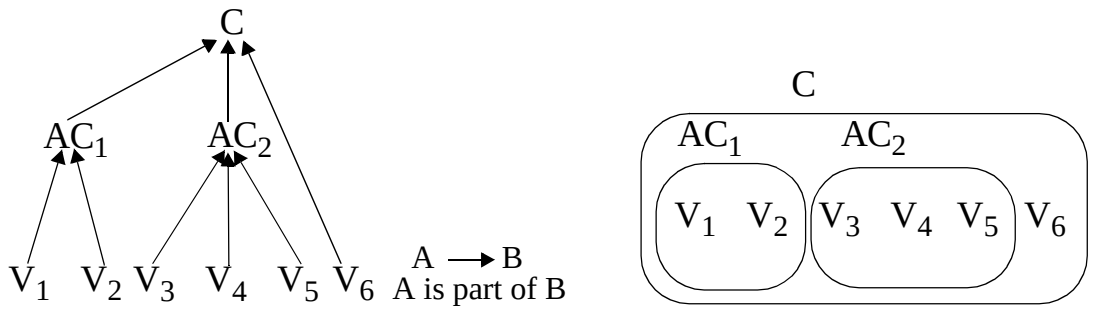


Figure 3-11. **Two equivalent graphical notations for subsystems.**

example in Figure 3-11 is a subsystem. Its subsystem structure is the graph spanned by the *part-of* relationship and rooted by *C* including the nodes V_1, V_2, V_3, V_4, V_5, V_6, AC_1, AC_2, and *C*.

In the following, identifiers *C* and C_i will be used to denote subsystems.

3.2.3 Entity-Relationship Model for Components

Components are not covered by the base entity-relationship model of Figure 3-9 on page 53. The base entity-relationship model, representing the relationships among base entities, is therefore extended in this section. For the purpose of this thesis, it is sufficient to model components and their *part-of* relationship.

Base entities (architectural quarks, more precisely) are the constituents of components. They do not contain other architectural quarks, that is why they are separated from components in the entity type hierarchy of Figure 3-12. Atomic components and subsystems are both architectural components.

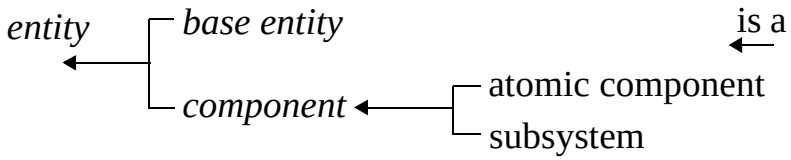


Figure 3-12. **Entity type hierarchy.**

The *part-of* relationship among components and architectural quarks is specified in Figure 3-13. Note that the *part-of* relationship is an n:m relationship, i.e, the

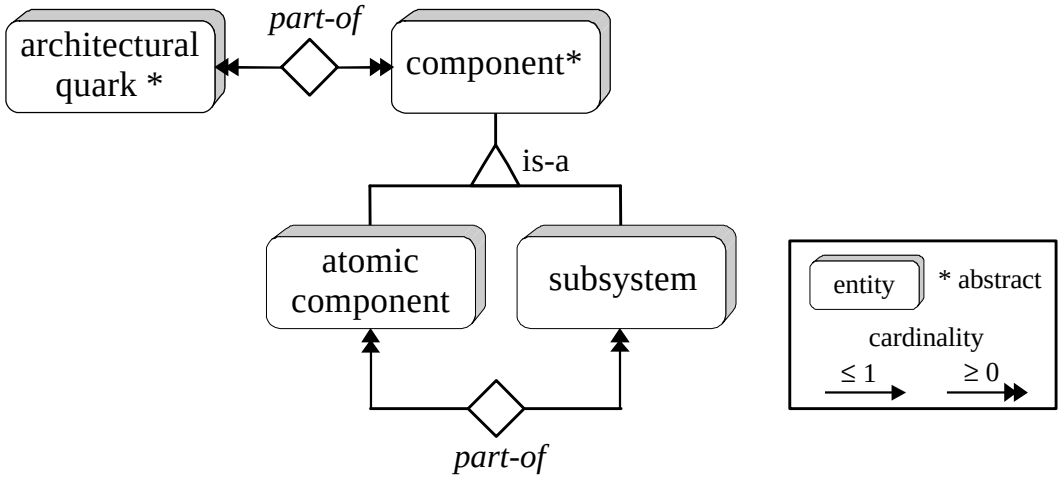


Figure 3-13. **Part-of relationship among architectural quarks and components.**

same entity can be part of different components. This is due to the concession to overlapping components.

3.3 Module Decomposition

Components are used to describe cohesive parts, hence, provide a *logical* view on the system. These components may differ from the *actual* or *physical* decomposition of the system into files. In order to describe the physical decomposition, means similar to those used for components can be used. In order to describe modules a new *module* entity is introduced that is derived from *entity*. Analogous to components, the *part-of* relationship will be used to identify the entities of a module. More precisely, *part-of* (*E*, *M*) holds for an entity, *E*, and a module, *M*, if and only if *E* is declared in *M*.

3.4 Resource Usage Graph

An instance of the entity relationship model introduced in the previous section to capture the basic structural information leveraged for atomic component detection can be represented by a graph. This graph is referred to as **resource usage graph** and is going to be used to describe the techniques for atomic component detection in more detail.

In the reverse engineering literature, the resource usage graph is also often referred to as **resource flow graph** (Müller and Uhl, 1990). However, it may be argued that the term resource flow graph is inappropriate since there is no real flow of resources in the information represented by this graph. The graph contains only very crude control and data flow information as (unordered) calls among subprograms and (directly visible) references of objects by subprograms. Additionally, it describes other aspects like the parameter types a subprogram has or the decomposition of types into other types or record components. Hence, we prefer the term *resource usage graph* in the following.

More formally, a resource usage graph is an attributed typed multi-graph $G = (N, E)$ whose set of nodes, N, consists of architecture-relevant elements and whose edges, E, are relationships among these elements. Edges can be either directed or undirected. The resource usage graph is typed in the sense that we distinguish different types of nodes (entities) and different kinds of edges (relationships). Nodes and edges are refined by means of subtyping and may be annotated with additional information (e.g., a signature edge of a function, F, to a composite type, T, is annotated if F accesses internal components of a parameter of type T).

3.4.1 Nodes

The nodes of the resource usage graph are all entities introduced in the course of this thesis, the edges are the relationships. A summary of all entities and relationships can be found in Appendix A.

3.4.2 Edges

Non-symmetric relationships are represented as directed edges. For directed edges among two nodes, the two involved nodes can be referred to as follows:

Let *e* be a directed edge, then ***source(e)*** is the node at which *e* starts and ***target(e)*** is the node at which *e* ends.

Symmetric relationships, such as *same-expression*, will be represented by undirected edges for which *source* and *target* are undefined. For undirected edges, we will use the term *nodes* to refer to the involved nodes:

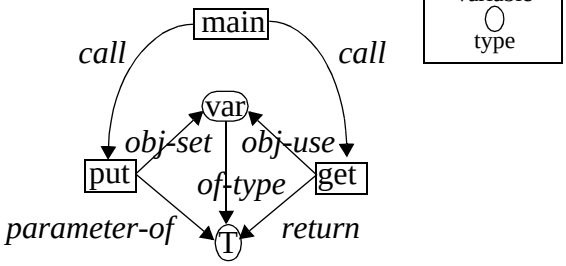
Let *e* be an undirected edge between the nodes n_1 and n_2, then ***nodes(e)*** = $\{n_1, n_2\}$.

The definition of *nodes* can be extended to directed edges as follows (let *e* be a directed edge): *nodes (e) = {target(e), source(e)}*. Note that *nodes(e)* is *{n}* when *e* is connecting the node *n* with itself.

Example. What kind of entities and relationships we actually have depends on the programming language that is to be modeled. Though the exact way of deriving the resource usage graph from C code will only be described in Chapter 4, a simple anticipating example will illustrate the concept. The resource usage graph for the C program in Figure 3-14(a) is shown in Figure 3-14(b).

```
typedef int T;
T var;
void put (T t) {
  var = t;
}
T get () {
  return var;
}
main () {
  put (5);
  get ();
}
```

(a)



(b)

Figure 3-14. **Example resource usage graph.**

3.4.3 Attributes

Nodes and edges of the resource usage graph may be annotated: *x.a* denotes an attribute *a* of node or edge *x*. In the course of this thesis, only boolean attributes are needed. If a boolean attribute is used in an expression as in

$$e = signature\ (S, T) \wedge e.non\text{-}abstract$$

it is to be read as “it is the case that *e* is *non-abstract*”.

3.4.4 Notational Conventions

We will use an *italic serifeless font* for nodes and edges of the resource usage graph to make explicit that we mean a concept of the resource usage graph in a given context as opposed to a programming language concept:

> *Type* is a concept of the resource usage graph, whereas “type” is a concept of programming languages.

This increases the readability as in “There is only one *call* among two *subprograms* though there may actually be many calls between these subprograms” instead of “There is only one call edge among the two subprogram nodes that are used to model the two given subprograms in the C code though there may actually be many calls between the subprograms in the C code.”

3.5 Predicates and Functions for Nodes and Edges

The resource usage graph will be used to describe the techniques to detect atomic components. The descriptions of the techniques contain primitive predicates and functions for nodes and edges that will be introduced in this section.

3.5.1 General Predicates

The information represented by a resource usage graph can be written out using a predicate notation; each relationship is represented by a clause, referred to as edge type predicate in the following. The arguments of a clause are entities. For example, the fact that subprogram, *get*, uses variable, *var*, can textually be repre-

sented as *obj-use (get, var)*. The first argument of a clause is the subject, the second the object of the relationship, or in terms of the resource usage graph: the first is the source of the edge that represents the relationship and the second is the target of this edge. We will use predicate notation within text; in figures, however, we will prefer edges.

Unary clauses will be used for node and edge type predicates. For example, to express that *x* is a subprogram, we can write *subprogram (x)* as a node type predicate. This notation can also be used to specify the type of an edge or relationship, respectively. For example, *set (e)* says that the relationship *e* is a *set* relationship. Given the relationship type hierarchy we introduced before, the predicate *reference(e)* is true whenever *set (e)* holds. Before we can define this more formally, the notation to express the *is-a* relationship among types of entities and types of relationships is introduced.

> Let *T1* and *T2* be types of either entities or relationships. Then *T1 is-a T2* expresses that *T1* is a *T2*, i.e., whatever is true for *T2* is also true for *T1* (but not necessarily vice versa). The *is-a* relation is transitive, i.e., if *T1 is-a T2* and *T2 is-a T3*, then *T1 is-a T3*.

Using the *is-a* relationship, we can define node and edge type predicates:

> *T(x)* holds when *x* is of type *T* or of a type *T'* where *T' is-a T.*

In some contexts, we want to abstract from a specific subclass of a base entity or relationship, yet, without treating all kinds of base entities or relationships alike. For example, we may treat two edges of kinds *parameter-of* and *return* equally because they are both *signature-type* edges and the distinction does not matter in the given context. But we do not want to consider *parameter-of* equivalent to *of-type* because the semantic gap is too wide. Based on the *is-a* relationship, we consider two relationships, *A* and *B*, **equivalent** – denoted by $A \sim B$ – if there is a relation type, *T*, where *T(A)* and *T(B)* and *T is-a base relation* where $T \neq$ *base relation*. Likewise, we consider two entities, *A* and *B*, **equivalent** if there is an entity type, *T*, where *T(A)* and *T(B)* and *T is-a base entity* where $T \neq$ *base entity*.

3.5.2 General Neighbor Functions

The successors and predecessors of a node in a resource usage graph R=(N,E) can be defined as follows:

$$\text{successors}(n) = \{x | \exists (e \in E)(\text{source}(e) = n \wedge \text{target}(e) = x)\}$$
$$\text{predecessors}(n) = \{x | \exists (e \in E)(\text{source}(e) = x \wedge \text{target}(e) = n)\}$$

In the case of successors and predecessors, the direction of the edges is relevant; that is why they are only defined for directed edges. If undirected edges are involved, we can use the concept of neighbors. The set of neighbors of a node *N* is the union of its successors and predecessors plus all nodes that are directly connected to *N* by undirected edges. This can be expressed as follows:

$$\text{neighbors}(n) = \bigcup_{e \in E} \{n' | \text{nodes}(e) = \{n, n'\}\}$$

Note that for the above definition of *neighbors*, *n* is its own neighbor exactly when:

$$n \in \text{neighbors}(n) \Leftrightarrow \exists (e \in E)\text{nodes}(e) = \{n', n\} \wedge n = n'$$
$$\Leftrightarrow \exists (e \in E)\text{nodes}(e) = \{n\}$$

The definitions for successors, predecessors, and neighbors do not take into account by what kind of edges the nodes are connected, which is generally relevant to the techniques to detect atomic components. Furthermore, for some of the techniques, only edges for which a certain boolean edge attribute holds are relevant. Therefore, we will extend the definitions of the neighbor functions above. Let *ST* be a set of type predicates for edges and *a* be a boolean edge attribute:

$$\text{successors}(n, ST, a) = \{x \mid \exists(e \in E)(\text{source}(e) = n \wedge \text{target}(e) = x \wedge \exists(T \in ST)T(e) \wedge e.\text{a})\}$$

$$\text{predecessors}(n, ST, a) = \{x \mid \exists(e \in E)(\text{source}(e) = x \wedge \text{target}(e) = n \wedge \exists(T \in ST)T(e) \wedge e.\text{a})\}$$

$$\text{neighbors}(n, ST, a) = \bigcup_{e \in E \wedge \exists(T \in ST)T(e) \wedge e.\text{a}} \{n' \mid \text{nodes}(e) = \{n, n'\}\}$$

In order to leave out the restricting boolean edge attribute in cases in which it is not needed, the following abbreviations are defined (let *true* be an attribute that is always true):

$$\text{successors}(n, ST) = \text{successors}(n, ST, true)$$
$$\text{predecessors}(n, ST) = \text{predecessors}(n, ST, true)$$
$$\text{neighbors}(n, ST) = \text{neighbors}(n, ST, true)$$

These neighbor functions can be naturally extended to sets of nodes. Let N be a set of nodes:

$$\text{succcessors}(N, ST) = \bigcup_{n \in N} \text{successors}(n, ST)$$
$$\text{predecessors}(N, ST) = \bigcup_{n \in N} \text{predecessors}(n, ST)$$
$$\text{neighbors}(N, ST) = \bigcup_{n \in N} \text{neighbors}(n, ST)$$

Example. For the example resource usage graph in Figure 3-14 on page 59, the neighbors would be as follows:

successors (put) = {var, T}
predecessors (put) = {main}
neighbors (put) = {var, T, main}
successors (put, {obj-set}) = {var}

predecessors (main, {call}) = ∅
neighbors (put, {obj-set, call}) = {var, main}
successors ({put, get}, {reference}) = {var}

It is also useful to define the transitive closure for the neighbor functions. First, we introduce the **n-th application** of a neighbor function NF (successors, predecessors, neighbors):

NF^1 (n, ST) := NF (n, ST)
NF^n (n, ST) := NF (NF^{n-1} (n, ST), ST) where n > 1

The **transitive closure** of a neighbor function NF is then:

transitive_closure (NF (n, ST)) := NF^{∞} (n, ST)

The definition of the n-th application and transitive closure of a neighbor function whose first argument is a set of nodes is analogous.

3.5.3 Neighbor Functions for Base Entities

Specific variants of the neighbor functions will frequently be used in the description of the techniques for atomic component detection and are therefore defined here in Table 3-4 in terms of the neighbor functions defined above or as a relational inverse (denoted by f^{-1}) to another frequently used neighbor function. If *f(x)* is defined as *f(x)=predecessors(x, S, a)*, then the **relational inverse** f^{-1} is defined as:

$f^{-1}(x)=successors(x, S, a)$.

If *f(x)* is defined as *f(x)=successors(x, S, a)*, then the relational inverse f^{-1} is defined as:

$f^{-1}(x)=predecessors(x, S, a)$.

Table 3-4. **Frequently used neighbor functions for (sets of) nodes.**

name	domain	range	definition
caller(s)	subpro-grams	subpro-grams	predecessors (s, {call})
callee	subpro-grams	subpro-grams	$caller^{-1}$
referencing-subprograms(v)	objects	subpro-grams	predecessors (v, {reference})
referenced-objects	subpro-grams	objects	$referencing\text{-}subprograms^{-1}$
signature-types(s)	subpro-grams	types	successors (s, {signature-type})
signature-subprograms	types	subpro-grams	$signature\text{-}types^{-1}$
referred-entities(s)	subpro-grams	objects types	successors (s, {reference, signature-type})
referred-by(s)	objects types	subpro-grams	$referred\text{-}entities^{-1}$

3.5.4 Elements of Components

In the course of this thesis, we often need to refer to the elements of a component, i.e., all its parts. Since atomic components may only consist of architectural quarks, the definition of their elements is straightforward. Because subsystems may contain further lower-level components, elements of a subsystem can be defined with respect to the level of granularity. This section defines the elements of the diverse kinds of components.

3.5.4.1 Direct Elements of a Component

The **direct elements of a component** are all entities that are directly part of this component, in other words, there is a *part-of* edge from the entity to the component:

$$\text{direct-elements}(C) = \{e \mid \text{part-of}(e, C)\}$$

In many contexts, we are interested in particular subsets of the direct elements of a component, namely, in its subprograms, objects, and types:

subprograms (C) = {S | subprogram (S) $\wedge$ S $\in$ direct-elements (C)}

objects (C) = {O | object (O) $\wedge$ O $\in$ direct-elements (C)}

types (C) = {T | type (T) and $\wedge$ T $\in$ direct-elements (C)}

Analogously, the **enclosing components** of an entity, *E*, are the components of which *E* is a part (there can be more than one):

enclosing-components (E) = {*c* | part-of (*E*, *c*)}

If *C* is an atomic component, *direct-elements (C)* denotes all its base entities. However, if *C* is a subsystem, *direct-elements (C)* yields only the direct descendants of *C* and not all elements in a subsystem structure with multiple levels. The definitions in the following section will refine the notion of *elements* for subsystems.

3.5.4.2 Indirect Elements of a Component

Because subsystem structures can have several levels, we can distinguish different kinds of elements depending on how far away they are from the root of the subsystem structure in terms of *part-of* edges (let *C* be a component):

$$\text{elements}^1(C) = \text{direct-elements}(C)$$

$$\text{elements}^n(C) = \bigcup_{\text{part-of}(e,\, C)} \text{elements}^{n-1}(e)$$

Note that $\forall(n \geq 1)\text{elements}^n(C) = \text{elements}(C)$ if *C* is an atomic component.

All elements of a component are the base entities and lower-level components in the transitive closure of the definition above:

$$\text{elements}(C) = \bigcup_{n \geq 1} \text{elements}^n(C)$$

If C is an atomic component, elements(C) = direct-elements(C).

Example. The *elements*^{1} of subsystem C in Figure 3-11 on page 56 are the atomic components AC_1 and AC_2 and the base entity V_6 whereas *elements*^{2}*(C)* = $\{V_1,V_2,V_3,V_4,V_5\}$. All its elements are $\{V_1, V_2, V_3, V_4, V_5, V_6, AC_1, AC_2\}$.

3.5.4.3 Partial Subset Relationship

At several places in this thesis, we have to compare two components with each other to ascertain their degree of congruence. This can be done in terms of their elements. Since the elements of components are basically sets of entities, one important information is whether the elements of one component are a subset of the other's elements. However, this is sometimes too strict. A less strict way of comparison is the following **partial subset relationship** $\subseteq_p$:

$$A \subseteq_p B \text{ if and only if } \frac{|A \cap B|}{|A|} \geq p \text{ where } 0.5 \leq p \leq 1.0.$$

The tolerance parameter p in this relationship can be specified by the maintainer. If set to 1.0, A must be completely contained in B.

This definition still considers a component with elements $\{a, b, c, d\}$ at least a partial subset of a component with elements $\{a, b, d, e, f\}$ when $p \geq 0.75$ though c is not present in the latter set of elements.

Note that the partial subset relationship is not transitive for $p \neq 1$. For example, {a, b, c} $\subseteq_{0.6}$ {a, b, d} $\subseteq_{0.6}$ {b, d, e}, but {a, b, c} $\not\subseteq_{0.6}$ {b, d, e}.

3.6 Views of the Resource Usage Graph

A resource usage graph represents the facts of a system that are leveraged by the atomic component detection techniques presented in this thesis. These facts are either derived from the system directly or by means of further analyses. Often, we need not all information but a specific excerpt. **Views** are parts of a resource usage graph representing special aspects. In the terminology of graphs, they are

subgraphs of a resource usage graph, i.e., all nodes and edges of the view are also in the resource usage graph and, for all edges in the view, both ends are in the view as well. Examples for important views are the call view (or call graph in compiler terminology), which renders the call relationship among subprograms, or the type view, which indicates how types are related.

Views will be used in the description of resource usage graph-based analyses to point out the parts of the resource usage graph that are relevant to the analyses. They will also be used to describe the output of analyses.

Table 3-5 lists some important elementary views that are directly derived from source code and which form the basis for atomic component detection. It also mentions components views that are used to represent the decomposition of components as a result of atomic component detection techniques. Components views are further described in Section 8.2. Whereas the components view describes the logical structure of the system, i.e., identifies the cohesive parts, the module view describes the actual structure of the system as a collection of modules and their declarations as it can be directly derived from the source code. The module view is also called the *physical file* or *module structure.*

Table 3-5. **Common views.**

View Name	Nodes	Edges	Explanation
call view	subprograms	call	call graph
type composition view	types, record components	part-type, enclosing	makes apparent how types are built
signature view	subprograms, types	signature-type	specifies the parameter interface of subprograms
type usage view	subprograms, types, objects	signature-type, of-type, local-obj-of-type	cross-reference for the usage of types
object reference view	subprograms, objects	reference	indicates which objects are set or used or whose address is taken by subprograms
same expression view	objects	same-expression	identifies objects that occur in the same expression

Table 3-5. **Common views.**

View Name	Nodes	Edges	Explanation
actual parameter view	subpro-grams, objects	actual-parameter-of	describes which object is an actual parameter of a sub-program
base view	base entities	base relationship	this is the union of the call, type composition, signature, type usage, object refer-ence, same-expression, and actual parameter views
components view	architectural quarks, com-ponents	part-of	describes the decomposition of one or more components (their part-of relationships)
module view	base entities, modules	part-of	describes the module decomposition; an entity is part-of a module if the entity is declared in the module

Example. The *object reference view* and the *call view* of the resource usage graph in Figure 3-14 on page 59, for example, can be found in Figure 3-15.

object reference view call view

Figure 3-15. **Example views.**

3.7 Identity, Affinity, and Correspondence of Components

At different places in this thesis, we have to compare components to each other. A comparison for atomic components can basically be based on two aspects of a component: Its identity as an entity and its set of elements. In the case of a sub-system, the subsystem structure is a third aspect.

Since components are entities, they have their own identity. In terms of the resource usage graph, two components, or more generally, two entities, A and B, are **identical** if they are represented by the same node, denoted by $A=B$.

If components are not identical, we can still consider them comparable if they share a vast majority of their elements. Such components are said to be **affine**. An exact definition of affinity for atomic components and subsystems follows below.

Two components are said to **correspond** to each other when they are either affine or identical:

$$\text{correspond}(A, B) = \text{affine}(A, B) \vee (A = B) \quad \textbf{(3.1)}$$

If two components do not correspond to each other, they are considered **dissimilar**. The definition of *affinity* follows in the next two sections.

Since base entities do not have further elements, they can only correspond to each other when they are identical.

3.7.1 Affinity for Atomic Components

Atomic components can be compared in terms of their elements using ordinary set relationships: We could require that the two atomic components have to have the same elements in order to be comparable. However, this is often too strict. A less strict comparison based on the set of elements can be made when only a certain degree of overlap is required. Hence, two atomic components A and B are considered **affine** when their degree of overlapping is above or equal to a user-determined **affinity tolerance parameter** Θ (if a maintainer insists on the requirement that the sets of elements must be equal, she can simply set $\Theta=1.0$):

$$\text{affine}(A, B) \Leftrightarrow \frac{|\text{correspondings}(A, B)|}{|\text{correspondings}(A, B)| + |\text{dissimilars}(A, B)|} \geq \Theta \quad \textbf{(3.2)}$$

Where *correspondings* of two atomic components are the elements shared:

$$\text{correspondings}(A, B) = \text{elements}(A) \cap \text{elements}(B) \quad \textbf{(3.3)}$$

Dissimilars denotes the elements that are either in *A* or *B*:

$$\text{dissimilars}(A, B) = \text{elements}(A) \backslash \text{elements}(B) \cup \text{elements}(B) \backslash \text{elements}(A) \quad \textbf{(3.4)}$$

Inequation (3.2) is equivalent to:

$$\frac{|\text{elements}(A) \cap \text{elements}(B)|}{|\text{elements}(A) \cup \text{elements}(B)|} \geq \Theta \quad \textbf{(3.5)}$$

The reason why we prefer (3.2) to (3.5) is that affinity for subsystems will be defined with the same structure as (3.2).

It is obvious that affinity according to (3.2) is symmetric.

3.7.2 Affinity for Subsystems

Affinity for subsystems is more complicated since it cannot just be based on the set of elements. It must also take the structure into account. As an example, consider the two subsystems C_1 and C_3 in Figure 3-16 where *elements* (C_1) = {A, B, C, D, E, F, AC_1, C_2} and *elements* (C_3) = {A, B, C, D, E, F, AC_2, AC_3}. The following observations can be made for this example:

1. $\dfrac{|\text{elements}(C_1) \cap \text{elements}(C_3)|}{|\text{elements}(C_1) \cup \text{elements}(C_3)|} = \dfrac{3}{5} \geq \Theta \qquad \forall \Theta \leq \dfrac{3}{5}$ with respect to (3.2)
2. C_1 and C_3 differ in structure
3. AC_1 and AC_2 are affine but they enter the comparison based on elements according to (3.2) as if they were dissimilar

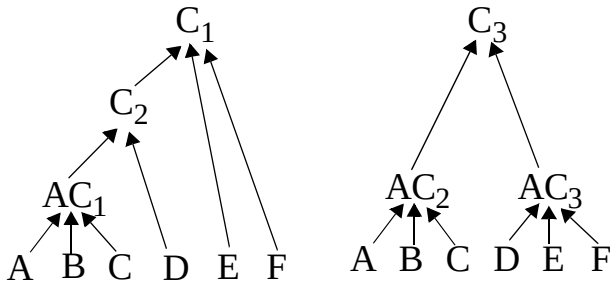


Figure 3-16. **Dissimilar subsystems.**

Because of observation (1), these two subsystems would be considered affine (for $\Theta \leq 0.6$) if the definition of affinity were only based on the set of elements for subsystems according to (3.2). However, since the two of them differ in structure, we do not want to consider them affine. Moreover, the fact that AC_1 and AC_2 are affine should also be taken into account by a definition of affinity for subsystems.

One alternative to accommodate these requirements is to consider two subsystems affine when their subsystem structures are graph-isomorphic. A graph $G_1 = (N_1, E_1)$ is isomorphic to a graph $G_2 = (N_2, E_2)$ if there is a bijective mapping f from N_1 to N_2 where:

$$\forall(n_1, n_2 \in N_1)(n_1, n_2) \in E_1 \Leftrightarrow (f(n_1), f(n_2)) \in E_2$$

An additional requirement is that the bijective function must be the identity for nodes that are in both structures. Otherwise, the bijective mapping could arbitrarily be chosen and therefore C_1 and C_4 in Figure 3-17 would be considered affine though the base entities (i.e., the leaves of the subsystem structure) are not at corresponding positions in C_1 and C_4.

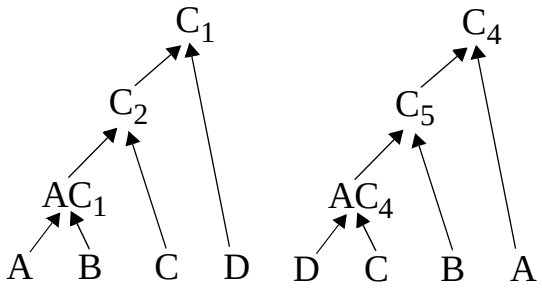


Figure 3-17. **Isomorphic subsystem structures.**

However, graph-isomorphism is too strict because, first, both subsystems must have the same number of nodes and, second, the subsystem structures must be structurally identical. Instead, we want to tolerate small divergences in elements and structure. This can be achieved by a recursive definition based on (3.2) that allows discrepancies by requiring that the direct elements of two subsystems have to be affine only to a large extent (more precisely, not to be *affine* but to *correspond* in order to capture identical elements as well). Since the definition is recur-

sive for the direct elements of a subsystem, a partial correspondence of the entire subsystem structures can be ensured.

The definition of affinity for subsystems is analogous to the one for atomic components. The following definition is therefore a unifying definition for both atomic components and subsystems. It differs from (3.2) only by the replacement of the term *atomic component* by *component* that comprises both atomic components and subsystems: Two components A and B are **affine** if their degree of overlapping is above or equal to a user-determined **affinity tolerance parameter** Θ:

$$\text{affine}(A, B) \Leftrightarrow \frac{|\text{correspondings}(A, B)|}{|\text{correspondings}(A, B)| + |\text{dissimilars}(A, B)|} \geq \Theta \qquad \textbf{(3.6)}$$

However, the definitions of *correspondings* and *dissimilars* for subsystems differ from the ones for atomic components. The definition of these two sets for atomic components were based on the *elements* of the atomic components, while we restrict *correspondings* and *dissimilars* of subsystems to the *direct elements* of the subsystems (*correspond* is defined by (3.1) on page 70):

$$\begin{aligned}&\text{correspondings}(A, B)\\ &= \{(a, b) | a \in \text{direct-elements}(A) \wedge b \in \text{direct-elements}(B) \wedge \text{correspond}(a, b)\}\end{aligned} \qquad \textbf{(3.7)}$$

$$\begin{aligned}&\text{dissimilars}(A, B)\\ &= \{a | a \in \text{direct-elements}(A) \wedge \neg\exists(b \in \text{direct-elements}(B))\text{correspond}(a, b)\}\\ &\quad \cup \{b | b \in \text{direct-elements}(B) \wedge \neg\exists(a \in \text{direct-elements}(A))\text{correspond}(b, a)\}\end{aligned} \qquad \textbf{(3.8)}$$

The definitions of *correspondings* and *dissimilars* were made in such a way that (3.6) is equivalent to (3.2) if A and B are both atomic components. In order to show this, let A and B be two atomic components. Because of *direct-elements (A) = elements (A)* for an atomic component, any $a \in$ *direct-elements (A)* is a base entity. Furthermore, two base entities a and b can only correspond to each other if they are identical, i.e., $\text{correspond}(a, b) \Leftrightarrow a = b$. Hence:

$$
\begin{aligned}
&\left|\{(a, b) \mid a \in \text{direct-elements}(A) \wedge b \in \text{direct-elements}(B) \wedge \text{correspond}(a, b)\}\right| \\
&= \left|\{(a, b) \mid a \in \text{direct-elements}(A) \wedge b \in \text{direct-elements}(B) \wedge a = b\}\right| \\
&= \left|\{(a, a) \mid a \in \text{direct-elements}(A) \wedge a \in \text{direct-elements}(B)\}\right| \\
&= \left|\text{direct-elements}(A) \cap \text{direct-elements}(B)\right| \\
&= \left|\text{elements}(A) \cap \text{elements}(B)\right|
\end{aligned}
$$

Hence, the nominator of (3.6) is equal to the nominator of (3.2). Likewise, the denominators of (3.2) and (3.6) are equivalent when $a = b$:

$$
\begin{aligned}
&\{a \mid a \in \text{direct-elements}(A) \wedge \neg\exists(b \in \text{direct-elements}(B))\text{correspond}(a, b)\} \\
&\quad \cup \{b \mid b \in \text{direct-elements}(B) \wedge \neg\exists(a \in \text{direct-elements}(A))\text{correspond}(b, a)\} \\
&= \{a \mid a \in \text{direct-elements}(A) \wedge a \notin \text{direct-elements}(B)\} \\
&\quad \cup \{b \mid b \in \text{direct-elements}(B) \wedge b \notin \text{direct-elements}(A)\} \\
&= \text{elements}(A) \backslash \text{elements}(B) \cup \text{elements}(B) \backslash \text{elements}(A)
\end{aligned}
$$

Thus, (3.6) is equivalent to (3.2) for atomic components. □

The equivalence allows us to use (3.6) as the general definition for affine components. Using the general definition, we can compare subsystems to atomic components with respect to their affinity. For example, the atomic component *AC* and the subsystem C_1 in Figure 3-18 are affine for $\Theta \leq 4/7$.

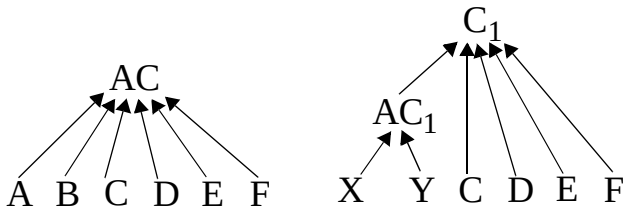


Figure 3-18. **Affine components.**

The example in Figure 3-18 shows another consequence of basing the definition of affinity for subsystems on direct elements only: Affinity as defined by (3.6) is robust against the size and structure of non-affine subcomponents, such as AC_1.

As already said in the beginning of this section, two components correspond if they are identical or affine. Since correspondence of direct elements is required in the definition for affinity of subsystems, correspondence of subsystems is defined recursively. The recursion in the definition of correspondence ends at the leaves of a subsystem structure, which are generally base entities. Since leaves have no further elements, they correspond if and only if they are identical. That is, correspondence of subsystems is well-defined.

The definition of affine subsystems according to (3.6) is more general than the previously discussed alternative definition based on graph-isomorphism. Two subsystems whose structure is graph-isomorphic (and where the bijective function is the identity for nodes that are in both structures) are obviously affine. However, the opposite direction is not necessarily true. This can be shown with the example in Figure 3-19. C_1 and C_2 are affine subsystems, but their structures are not isomorphic since there does not exist any bijective function between them.

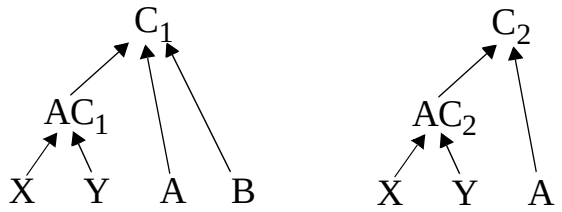


Figure 3-19. **Non-isomorphic, yet affine subsystems.**

3.7.3 Properties of the Correspondence Relationship

Correspondence as defined in the previous sections has the following properties:

- ***reflexive***: $\forall(A,\Theta)$ *correspond* (A,A)
- ***symmetric***: $\forall(A,B,\Theta)$ *correspond* $(A,B) \Rightarrow$ *correspond* (B,A)
- ***not transitive***:
  $\neg\forall(A,B,C,\Theta)$*correspond*$(A,B)\wedge$*correspond*$(B,C) \Rightarrow$ *correspond*(A,C)
- ***ambiguous***: $\neg\forall(A,B,C,\Theta)$ *correspond*$(A,B)\wedge$*correspond*$(A,C) \Rightarrow B$=C

Note that $\neg\forall(A,B,C,\Theta)$ *correspond*$(A,B)\wedge$*correspond*$(A,C) \Rightarrow$ *correspond*(B,C) can immediately be derived from symmetry and non-transitivity (see below).

The correspondence relationship is formally defined as:

$$\text{correspond}(A, B) = (A = B) \vee \text{affine}(A, B)$$

Correspond (A, A) holds because the two arguments are identical and identity is a reflexive relationship. Thus, *correspond* is reflexive. □

Since identity is obviously symmetric, it remains to be shown that affinity is also symmetric. The symmetry of *correspond* can be shown by induction. The induction begins with the smallest components, i.e., atomic components. It cannot begin with base entities since for these, affinity is not defined.

Begin of induction. We have already shown that (3.6) is equivalent to (3.2) for atomic components. Equation (3.2) is obviously symmetric. Thus, affinity for components at height 1 is symmetric.

Inductive step. We assume that the affinity for the direct elements of two subsystems *A* and *B* is symmetric. Then, we have to show that:

$$\text{affine}(A, B) \Leftrightarrow \text{affine}(B, A)$$

which is equivalent to

$$\frac{|\text{correspondings}(A, B)|}{|\text{correspondings}(A, B)| + |\text{dissimilars}(A, B)|} \geq \Theta$$
$$\Leftrightarrow \frac{|\text{correspondings}(B, A)|}{|\text{correspondings}(B, A)| + |\text{dissimilars}(B, A)|} \geq \Theta$$

Because *dissimilars* (*A*, *B*) = *dissimilars* (*B*, *A*) is obviously true, it remains to be shown that the nominators of the fractions above are equal. Because symmetric correspondence of the direct elements of *A* and *B* was assumed, the following holds:

$$\forall(a \in A, b \in B)\text{correspond}(a, b) \Leftrightarrow \text{correspond}(b, a)$$

Therefore:

$$
\begin{aligned}
&|\text{correspondings}(A, B)| \\
&= \left|\{(a, b) \mid a \in \text{direct-elements}(A) \wedge b \in \text{direct-elements}(B) \wedge \text{correspond}(a, b)\}\right| \\
&= \left|\{(a, b) \mid a \in \text{direct-elements}(A) \wedge b \in \text{direct-elements}(B) \wedge \text{correspond}(b, a)\}\right| \\
&= \left|\{(b, a) \mid a \in \text{direct-elements}(A) \wedge b \in \text{direct-elements}(B) \wedge \text{correspond}(b, a)\}\right| \\
&= |\text{correspondings}(B, A)| \qquad \square
\end{aligned}
$$

The counter-example in Figure 3-20 shows that affinity is generally not transitive. For $\Theta = 0.6$, AC_1 is affine to AC_2 and AC_2 is affine to AC_3 but AC_1 is not affine to AC_3. However, for $\Theta = 1.0$ it is indeed transitive because there must not be any dissimilar elements then (without proof). □

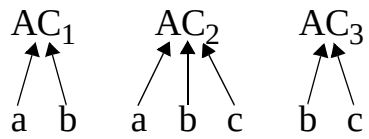


Figure 3-20. **Non-transitively affine components.**

The property $\neg\forall(A,B,C,\Theta)$*correspond*$(A,B)\wedge$*correspond*$(A,C)\Rightarrow$*correspond*$(B,C)$ can be derived from symmetry and non-transitivity by contradiction. Let us assume that $\forall(A,B,C,\Theta)$*correspond*$(A,B)\wedge$*correspond*$(A,C) \Rightarrow$ *correspondence*(B,C), then:

$$
\begin{aligned}
&\text{correspond}(A, B) \wedge \text{correspond}(A, C) \Rightarrow \text{correspond}(B, C) \\
&\Leftrightarrow \text{correspond}(B, A) \wedge \text{correspond}(A, C) \Rightarrow \text{correspond}(B, C)
\end{aligned}
$$

Hence, *correspond* would be transitive, which was shown to be wrong. □

The ambiguity of *correspond* is also obvious: Let $B \neq C$ and *direct-elements* (B) = *direct-elements* (C), then, if there is an A for which *correspond* (A, B) holds, *correspond* (A,C) holds, too, i.e., $\exists(A,B,C,\Theta)$ *correspond*$(A,B) \wedge$ *correspond*$(A,C) \wedge B \neq C$. □

Hence, there may be more than one correspondent to each component.

3.7.4 Correspondence Relationship and Views

Chapter 8 introduces incremental techniques that take a components view as input and produce a components view as output. The techniques may add elements to existing components, that is to say, the output components view may contain components of the input components view (Figure 3-21). When two incremental techniques, T_1 and T_2, are applied to the same input view, V_{input}, the two output components views $T_1(V_{input})$ and $T_2(V_{input})$ may both contain identical components. Hence, the same component can be in the input view as well as in both output views. As a consequence of the application of two different techniques, the same component may have different elements in the two output views. This is so because the two techniques propose to add different elements to the same component. That is why identical components in different components views need not be affine, e.g., the component AC has different elements in $T_1(V_{input})$ and $T_2(V_{input})$ in Figure 3-21. However, identical components can only have different elements with respect to different views.

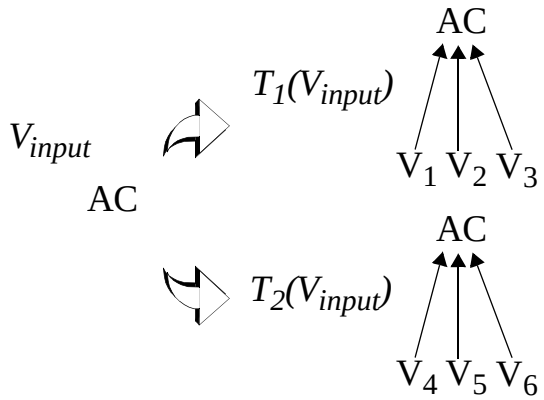


Figure 3-21. **Identical, yet non-affine components in different views.**

Example. The atomic components AC_1 and AC_2 in view (a) and (b) in Figure 3-22 are affine (for $\Theta = 0.5$) but not identical. The atomic component AC_1 in view (a) in Figure 3-22 is obviously identical to AC_1 in view (c), but the component has different elements in (a) and (c) and the two structures are, hence, not affine with respect to these two views. The atomic components AC_2 and AC_1 in view (b) and view (c) of Figure 3-22 are neither identical nor affine.

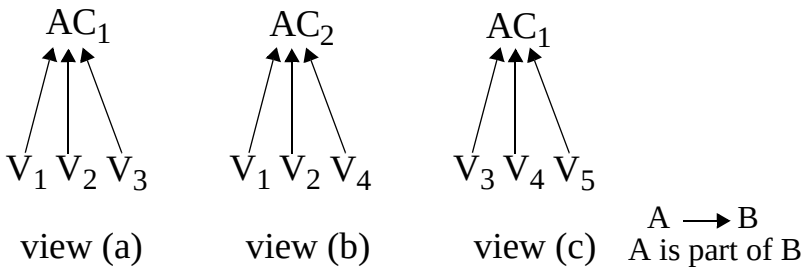


Figure 3-22. **Example atomic components.**

Chapter 4 *Components in the Programming Language C*

As a proof of concept, the approach proposed in this thesis has been implemented for programs in the programming language C. The decision to use C as a target language has practical reasons and reasons that lie in the language as such. Many legacy systems are written in C and many large C systems are available in the public domain. Furthermore, C is widely used as target language in the reverse engineering community, which allows comparable results. C supports abstraction by allowing the user to define his own types and by offering means to hide details of the implementation (see Section 4.3). Yet, the support for information hiding is quite limited and commonly unused such that reverse engineering can make a real contribution to program comprehension of C programs. Despite of the dissemination of the advantages of information hiding, C is still one of the most popular programming languages. Programmers that are acquainted with the ideas of information hiding are trying to simulate the lacking means of expression. However, many programmers do not know these principles or simply ignore them. All that makes C an interesting language from the reverse engineering researcher's point of view: There is abstraction in the language, yet not enough; programs are designed with the ideas of information hiding in mind, yet these ideas are often ignored. Last but not least, C is anything else than a toy language: It has many idiosyncrasies, such as pointer arithmetic, an unsafe type system, or gotos that make analyses of C programs difficult. If an approach works for C, it is likely that it also works for languages that are at a higher level of abstraction than C. Whether this is also true for more primitive languages like Fortran77 is discussed in Section 4.4.

This chapter describes how C is mapped onto the resource usage graph that is henceforth used as a basis to the methods that are going to be proposed in the course of this thesis. An overview of the technical steps of the mapping will be described first. Then, a brief overview of the relevant features of C follows that also shows how these features are mapped to the resource usage graph. Section 4.3 describes the way information hiding could be achieved in C and Section 4.4 discusses the role of the programming language and other factors for atomic component detection in general.

4.1 Analyzing C Code

Before we describe the mapping of features in C to the resource usage graph, we will give a short overview of the technical intermediate steps of this mapping. This is helpful to understand some properties of the mapping.

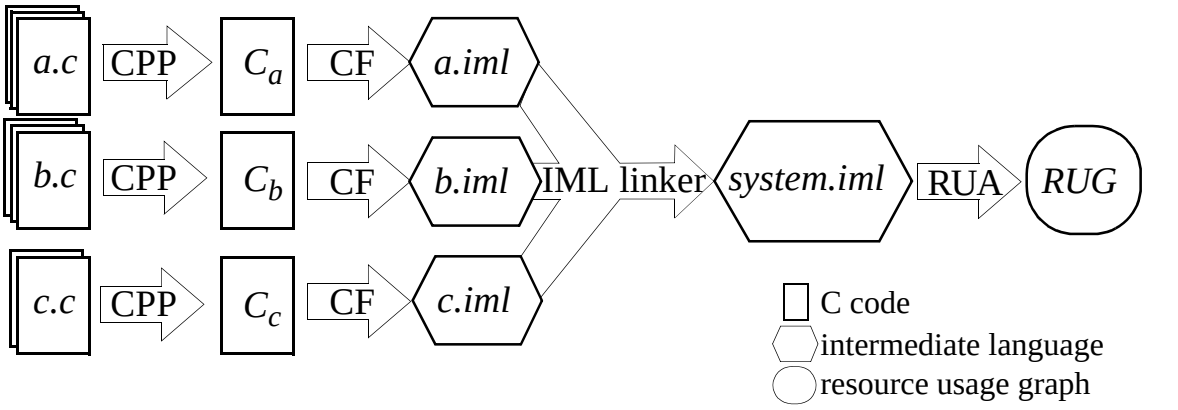


Figure 4-1. **Mapping C code to a resource usage graph.**

Figure 4-1 presents an overview of the intermediate steps. First, the C code is preprocessed by a standard C preprocessor (CPP). As a consequence, macros are replaced before the actual analysis takes place; the discussion of macros is picked up by Section 4.2.1. Only the preprocessed C code is analyzed by our C front end (CF) which generates an intermediate representation (IML) for each analyzed C program unit (Koschke et al, 1998). The individual intermediate representations generated for the individual program units are then linked by an IML linker to a global description of the system. Global external references are resolved at this

stage. Finally, the actual resource usage analysis (RUA) takes place for the global system and maps the C program to the resource usage graph (RUG). All subprograms that do not have a body within the intermediate representation of the global system are assumed to belong to libraries; the same holds for external variables that do not have a definition. Since the preprocessor handles include files and there is no distinction between type declarations and definitions, it is not directly known to the resource usage analysis whether type declarations belong to the system. This information is provided by a command line argument to the resource usage analysis that specifies the library paths. Any declaration of a file that is in one of the given library paths is considered a library unit. Details of the resource usage analysis are described in the next section.

4.2 The Programming Language C

In Chapter 3, we introduced an abstract entity relationship model that is mostly language-independent (we will find these entities and relationships in virtually all procedural programming languages). In this section, we will refine the model for the programming language C by describing the C entities and their possible relationships that we model. The mapping of the information derived from C source to entities and relationships of the resource usage graph will be called **resource usage analysis (RUA)**.

4.2.1 Modules and Macros

There is no concept *module* in C. In order to decompose a system into individually manageable parts, the programmer has to use ordinary files. Programmers often simulate the lacking concept by using one file that contains the code, thus simulating the body of the module, and one file that contains the exported declarations of the module - its specification. The latter is a so-called header file. A convention is to give the two files the same name except for the file suffix; the header file has the suffix "*.h*" whereas the "body" file has the suffix "*.c*". However, this is only a convention that helps a human reader to find the corresponding header file to a given body file. Header files have no meaning for a C compiler.

In order to import declarations of a header file, a preprocessor directive include is used by the importing file. A preprocessor replaces these includes by the actual file and feeds its contents into the C compiler. The C compiler itself is not aware of these files and is therefore not able to cross-check declarations within the system.

Since a preprocessor is needed within a C development environment at any rate, programmers often make use of further preprocessor directives. Macros can be defined whose occurrences are textually replaced by the preprocessor. Conditional directives can be used to ignore or insert text depending upon macro values.

Macros convey important information. For example, they are often used to define constants. When macros are expanded, only meaningless numbers remain where symbolic names were used. In the early days of C, macros were the only way to define constants. Today, the new ANSI standard for C has means to specify constants. However, this is too late for legacy systems and even nowadays, programmers often stick to macros to define constants (they even have to in many cases due to an ANSI rule that excludes constants in static expressions; see Section 4.2.5). Unfortunately, macros are very hard to handle since their usage is virtually unrestricted. The source code could be written in a way that it only obeys the syntax rules of C after the contained macro calls are replaced. This makes analyzing source code with arbitrary macros nearly impossible and is the reason why most reverse engineering tools ignore them, i.e., they analyze the code exactly how it is presented to the C compiler when all macros are replaced. This is also true for the C front end we are using (see Section 4.1). A better and pragmatic solution would be to treat at least simple macros as if they were expressions. On the other hand, syntactic analysis is more difficult to implement when syntax errors can be caused by omitted macro replacements.

4.2.2 Name Spaces

C has three different name spaces: one for enumeration, struct, and union types, one for goto labels, and one for all other kinds of identifier. Within a name space, an identifier can occur only once at the same nesting level. However, the same name can occur in all name spaces. For example, the following is legal C code (note the semantic error in the call to malloc which is likely to occur due to overlaps in name spaces: the argument List of the operator sizeof binds to the formal parameter and therefore sizeof yields the number of bytes needed to encode a

pointer where the space required for struct List was actually meant; the compiler cannot detect such errors):

```
struct List {struct List* next; int value;};
void f (struct List *List) {
  if (List->next == null)
    goto List;
  List->value = 1;
  List: List->next = (struct List *)malloc (sizeof (List));
}
```

Name binding for non-external names is done by the front end and external names are resolved by the linker. However, the different name spaces for enumeration, struct, and union types on one hand and ordinary identifiers on the other hand have to be reflected in the resource usage graph as well (goto labels are not represented in the resource usage graph), e.g., a struct Queue and a type Queue are represented by two separate nodes in the resource usage graph.

4.2.3 Types

In C, the programmer is able to write user-defined types. This way, more abstract programs can be written. However, the type system has many idiosyncrasies that make the resource usage analysis more difficult.

C has numeric (discrete and floating-point) and character base types - where the latter is actually a subtype of int. Boolean types have to be simulated by discrete types. Base types of the programming language C will be ignored by the resource usage analysis since they provide little information useful for architecture recovery.

User-defined types can be built on top of these base types using the common type constructors pointer, array, struct, and union. Structs and unions compare to records and will be discussed in Section 4.2.3.3. In the terminology of C, the type constructors array and pointer (denoted by [] and *) are declarators and only structs, unions, enums, and all base types are considered “real types”. This differs from the terminology used for most other programming languages; we will therefore stay with the common terminology and consider a declaration such as

```
typedef int *My_Integer_Pointers[10];
```

as a declaration of a type `My_Integer_Pointers` that is an *array with range 0..9 of pointers to int*.

4.2.3.1 Typedefs

In other procedural programming languages, there is an explicit construct to introduce a new type. Such type declarations have a name for the new type and describe its data structure. Typedefs in C are similar to type declarations in other languages:

```
typedef struct Node List;
```

This typedef introduces a name *List* that can be used to declare objects whose data structure is actually struct *Node*. However, the semantics of a typedef in C is only to introduce a synonym or abbreviation for another type. Whether one declares an object of type struct *Node* or *List* for the example above does not make a difference.

According to the definition in Section 3.1.1, struct *Node* is a part-type of *List*. However, there is a subtle semantic difference to the kind of part-type relationship introduced in Section 3.1.1. For example, in a record declaration like:

```
struct Node {Item value; Key key; struct Node *next;};
```

Item, *Key*, and struct *Node* are part-types of struct *Node*, i.e., a type can have several part-types (it can even be a part-type of itself), in other words, each part-type describes only a part of struct *Node*. On the other hand, in the case of a typedef, the data structure of the new type is completely determined by the existing type within the typedef declaration. In order to distinguish a typedef from a type declaration that induces several part-types, a *delineate* relationship is added to the entity relationship model of Section 3.1.1. Since the *delineate* relationship is a special kind of the part-type relationship, *delineate* is derived from the part-type relationship (delineate *is-a* part-type):

> **delineate**(*A*,*B*) expresses that *B* is defined in terms of *A* as a synonym or as a new type, in other words, *A* delineates the structure of *B*.

The definition of *delineate* covers two kinds of situations in which typedefs are used in C:

1. to introduce a synonym or abbreviation as in:
   ```
   typedef struct list List;
   ```
   where a programmer can use *List* where struct *list* is meant and save the effort for putting struct in front of *list*
2. to implement a new type by an existing type as in:
   ```
   typedef List Queue;
   ```
   where *Queue* and *List* have the same data structure and a programmer is able to call the accessor functions of *List* with a variable of type *Queue*, but *Queue* is meant to be a new type.

The semantics of typedef in C is really to introduce a synonym, hence, variables of type *List* could also be actual parameters of new accessor functions of *Queue*; they could even be assigned to variables of type *Queue* and vice versa. However, pragmatically, programmers often use a typedef as a type constructor introducing a new type. Whether a typedef is meant as a declaration of a synonym or a new type cannot be decided for C in general. A case like

```
typedef struct list List;
```

is likely a synonym but not necessarily. Even less obvious are declarations like:

```
typedef List list;
```

Due to these ambiguities, the definition of *delineate* does not distinguish between typedefs for synonyms and new types in C. In a language that has means to specify the distinction explicitly, one would derive two subtypes of *delineate* for the two usages. For example, in Ada there are two explicit concepts to distinguish *delineate*:

- In order to introduce a *synonym* to an existing type, a subtype declaration can be used:
  ```
  subtype Queue is List;
  ```
  Queue and *List* are equivalent.
- In order to introduce a *new* type based on an existing type, a new type can be derived by the keyword **new**:
  ```
  type Queue is new List;
  ```

In this case, *Queue* inherits all primitive operations of *List*. However, variables of *Queue* cannot be assigned to variables of type *List*, nor can variables of type *List* be assigned to variables of type *Queue*; the two types are really distinct.

4.2.3.2 Enums

Enumerations of a finite set of discrete values can be expressed by enums in C. The following declaration introduces a new enum type *E* whose range consists of *a*, *b*, and *c*:

```
enum E {a, b, c};
```

In C, the values of an enum type are treated as constants of type int, i.e., assignments between enum and int variables are allowed.

Enums provide little abstraction and atomic component detection techniques often ignore them. However, they are frequently part of a more abstract composite structure and there are indeed examples where they constitute an atomic component, e.g.:

```
typedef enum {true, false} Boolean;
Boolean and (Boolean left, Boolean right);
Boolean or (Boolean left, Boolean right);
Boolean not (Boolean operand);
```

Therefore, we explicitly capture *enums* in the resource usage graph.

4.2.3.3 Structs and Unions

Another way to build a new type is available by declaring struct and union types as in:

```
struct Complex {float re, im;};
```

This declares a new record *Complex* with the components *re* and *im* of type float. As opposed to all other types, the types of struct and union variables must be name equivalent if the variables are to be assigned to each other.

If such a type declaration is to be used, the keyword struct has to be added when the struct name is used to identify the name space. For example, in order to declare a function with *Complex* as parameter, one has to write:

```
void f (struct Complex c);
```

A typedef can be used to introduce an abbreviation:

```
typedef struct Complex Complex;
void f (Complex c);
```

As mentioned in Section 4.2.2, it is legal to use the identifier *Complex* twice in this context due to the different name spaces. However, f could also be specified with the explicit struct Complex in its parameter list and it would not make a distinction from the point of view of the language, but as noted in Section 4.2.3.1, we do make a distinction between the struct Complex and the typedef Complex. They are different *types* in the resource usage graph. Otherwise, all the following functions would be considered similar, whereas we optimistically assume that there is a reason why the programmer introduced three different types:

```
void f (struct Complex c;)
void g (Complex c);
typedef struct Complex Polar;
void h (Polar c);
```

A union is a type that stores values of different types at the same storage location. One can think of it as a struct whose components have an offset of 0. Thus, assigning a value to component *a* of an instance of the following union U will override the value of component *b*.

```
union U {int a; float b;};
```

Unions are used for type conversions or for storage optimizations, when only one of the components of an object is valid at any given time. As opposed to structs, they cannot be used to implement a heterogeneous composite concept, i.e., a concept with several distinct components, because they can only store one component at a time and therefore generally provide little abstraction. That is why most reverse engineering techniques ignore unions. However, we consider both *structs* and *unions*. Unions are, for example, useful to recognize the similarity of two functions when they both are related to a union.

The components of structs and unions are also captured by the resource usage analysis. Each component is modeled by a *record component* no matter whether it is part of a *struct* or a *union*. The relationship *enclosing* identifies the *struct* or *union* to

which the *record component* belongs. The types of the record components are considered *part-types* of the struct or union. More about record components follows in Section 4.2.6.

4.2.3.4 Anonymous Types

One does not need a named type in C in order to declare objects or to specify parameters. The following declarations are legal:

```
int *a[10];
char *f (int b[]);
typedef struct {int a;} T;
```

Types without name will be called **anonymous**. In the last example, the struct has no name. Since it has no name, it cannot be used anywhere else than in this very type declaration of *T*. Therefore, one could treat this declaration as if *T* were the record type. However, anonymous structs can also occur in object declarations. Under such circumstances, a *type* is needed for the object. That is why we introduce a *type* for each struct without name no matter whether the context is an object or type declaration. The latter is done for reasons of uniformity. An artificial and unique name is assigned to the anonymous type. Anonymous unions and enums are treated analogously.

A type that has a name (even if it is artificial as for anonymous structs, unions, and enums) will be called a **named type**; it is either an **intrinsic type** (int, char, float, double and their respective signed/unsigned and long/short variants) or a struct, union, enum, or typedef.

Declarators and hence anonymous types can appear wherever a type is allowed; thus, anonymous types are quite frequent in C programs, though using a telling name would convey more information. There is one specific context in which anonymous types are really appropriate. Parameters are always passed by value in C and changes to the parameters therefore do not have any effect on the actual argument since the parameter is a copy of the passed object. If the value of the passed object is to be changed, call-by-reference must be simulated. Consider the following C code:

```
T t;
void f (T pt) {
  pt = ...;
}
f (t);
```

The assignment to *pt* in the body of *f* does not change the value of *t*. If it is to be changed, the parameter type of *pt* must be changed to a pointer to *T* and the address of *t* must be passed at the call site. Furthermore, all occurrences of *pt* in the body of *f* must be replaced by dereferences of *pt*. Thus, we have to rewrite the code as follows:

```
T t;
void f (T *pt) {
  *pt = ...;
}
f (&t);
```

Now, the type of parameter *pt* has become an anonymous type. Note that the programmer could also introduce a new pointer type for the formal parameter, such as:

```
typedef T *PT;
```

and use *PT* instead of **T* for *pt* in the signature. However, the new pointer type in the signature may be misleading since it suggests *f* to be a function for *PT* as opposed to be a function with a reference parameter for *T*.

Anonymous pointer types are adequate for reference parameters and therefore frequently used in C. This scheme often involves arrays, too, since they are more or less equivalent to pointers in C. On the other hand, it is sometimes not quite clear whether an anonymous pointer type is in fact a reference parameter. For example, given the following type declarations:

```
typedef int Item;
#define MAX 100
typedef Item stack[MAX];
```

We would expect signatures for accessor functions of *stack* as follows:

```
void push (stack s, Item i);   /* example 1 */
Item pop (stack s);
```

However, the programmer could have mistakenly used a type descriptor "*Item **" instead of the type *stack* in the parameter list of a function *pop*:

```
void push (stack s, Item i); /* example 2 */
Item pop (Item *s);
```

Then, *s* in *pop* looks like a reference parameter and not like an accessor function of *stack*. Programmers could even write the program without the type *stack* as follows:

```
void push (Item []s, Item i); /* example 3 */
Item pop (Item []s);
```

or even with anonymous types only

```
void push (int *s, int i); /* example 4 */
int pop (int *s);
```

All these alternative declarations are equivalent in C. However, in the latter two alternatives, the connection of *push* and *pop* as accessor functions of *stack* is no longer visible. In example 3, we could at least assume that *push* and *pop* are related to *Item*. But if only one the two functions mentions *stack* in its signature as in example 2, there is no easy way to find out that "*Item **" actually means *stack*; *s* could also be a reference parameter of type *Item*. In example 4, it is not even clear that *push* and *pop* deal with objects of type *Item*, and one cannot even know whether the two anonymous types "*int **" denote the same concept or whether they are two distinct types with the same structure.

If we knew that *push* and *pop* are accessor functions of the same atomic component, we would be able to conclude that the two anonymous types "*int **" in example 4 are in fact equal. However, because atomic components and their accessor functions are still to be detected, it is not known whether two anonymous types denote the same concept when the resource usage analysis takes place. Furthermore, because there are so many anonymous types in typical C programs that all need to be represented for each occurrence when nothing is known about their meaning, the resource usage graph would contain a huge number of anonymous types - of which most are probably only reference parameters - and, consequently, be of little help to the maintainer. For these reasons, the resource usage analysis ignores all anonymous types and reduces them to their named base types, i.e., we

will treat a parameter declaration as in f (T *t) as if it were the signature f (T t). This makes sense because *t* is probably a reference parameter of type *T*. We will handle a declaration f (T t[]) analogously since it is equivalent to f (T *t) in *C*. More debatable is the fact that we also reduce more complex anonymous types to their named base type, thus f(T **t) is also treated as signature f(T t) by the resource usage analysis. However, since anonymous types are involved, there is no other type this could be attached to. Hence, we rely on the abstraction that programmers add by using named types.

In order to specify more precisely how we handle anonymous types, we define a few terms needed. The **base type** of an anonymous type is the type of object a pointer can point to or the type of elements of an array, respectively. The **named base type** of an anonymous type can then be defined recursively. Let *A* be an anonymous type expression, then:

- if the base type of *A* is a named type *T*, then the named base type of *A* is *T*
- if the base type of *A* is an anonymous type *A'*, then the named base type of *A* is the named base type of *A'*

Example. Let us make an example to illustrate these definitions. Given the following anonymous type:

```
T *a[]
```

The anonymous type of a is an *array of pointers to T*, its base type is a *pointer to T*; this one's base type is *T*, and the named base type of *a* is *T*.

Only user-defined named base types occur in the resource usage graph and the resource usage analysis reduces any occurrence of an anonymous type to its user-defined named base type if there is any, i.e., in declarations as in

```
int *a[];
```

the type of *a* is ignored.

4.2.4 Global Variables

Only global variables are represented in the resource usage graph since local variables are not relevant at the architectural level; nevertheless, local variables are

not completely disregarded as Section 4.2.7 will show. Henceforth, if nothing else is said, *variable* always means *global variable*.

The type of the variable in the resource usage graph is its user-defined named base type, if there is any, referred to by an *of-type* edge. If the variable has no user-defined named base type, no *of-type* edge is introduced.

If two variables (or constants), *A* and *B*, occur in the same expression on the left and right hand side of assignments, in conditions of control statements, or in an actual parameter list of a function call, then *same-expression (A,B)* holds. It is sufficient for them to be once in the same expression, how often they actually occur is not regarded, and they also may occur at different nesting levels of the expression. This definition will be extended in Section 5.5 when the motivation for this relationship becomes clearer.

4.2.5 Constants

In C, there are basically two ways to declare constants: By means of a macro and by means of the keyword const. Using macros is a historical relict from Kernighan and Richie C (1978) where the keyword const did not exist. The only way to introduce symbolic names for constants was to use simple macros such as:

```
#define Max 0
```

In ANSI-C (1989), the same could be expressed as:

```
const int Max = 0;
```

Such constants may be initialized at their declaration and thereafter not be changed anymore. Nevertheless, many programmers are still using macros to define constants partly because constants must not occur in constant expressions (Arnold and Peyton, 1992). C is unlike C++ in this regard.

As said in Section 4.2.1, macros are already expanded for our front end by a preprocessor. That is to say, only constants annotated with const are recognized and represented in the resource usage graph.

There are also objects declared as regular variables that are in fact used as constants, i.e., they are initialized in their declaration but never set elsewhere yet not annotated with const. This may be just for that particular system under investigation in which the subsystem that contains the variable is used or because the programmer forgot to declare the variable explicitly as constant. It is safe to say that a variable is actually a “constant” if it is set only once and its address is never taken. The opposite conclusion is not necessarily true: A variable whose address is taken could still be a constant when no dereference changes the variable’s value. A better approximation could therefore be ascertained by data flow analyses. However, since the automatic techniques for atomic component detection treat constants like variables, we do not need data flow analyses.

4.2.6 Record Components

Record components are used to model the components of structs and unions types. For each record component *R* of a type *T*, a record component specifier for *R* is introduced whose enclosing is *T*. For each global object *V* of type *T*, *V* inherits all transitive record component instances of *T* as described in Section 3.1.2.

4.2.7 Functions

In C terminology, subprograms are called functions no matter whether they return a result or not. A function has a definition that specifies the signature of the function and its body and it has an optional declaration that only specifies its signature. The latter is used when the program is decomposed into several files and the function is to be exported. However, this is not a must. If functions are neither declared nor have a definition, their designated default return type is int and their parameter list is assumed to be variable and unspecified, i.e., any number and type of actual parameters can be passed. From the standpoint of program understanding, little is to learn about such functions. Even worse are contradicting function declarations. A function can be declared in one file as

```
void f(int i);
```

and in another file as

```
int *f(float f; char *s);
```

Since the compiler analyzes always one file (more precisely, the file and its include files; see Section 4.2.1) per compilation, these contradicting declarations will not be detected. The compiler will generate code for the current declaration of f. This can lead to run-time errors that are hard to track down. Our linker performs a global analysis and finds such contradicting function declarations (see Section 4.1).

Functions can only be declared at file level and are either extern or static. Extern functions are globally visible, i.e., they can be called from other files. Static functions are local to a file. If nothing is said, functions are extern per default. Both kinds of *subprograms* are represented in the resource usage graph in the same way.

Not all functions need to have a definition. The code of functions can be linked into the program from libraries. When the resource usage graph is generated by a global analysis (see Section 4.1), functions without definition can be identified. They are considered library units.

In C, there are basically two ways of calling a function: either calling it directly by its name or via a function pointer. Generally in the case of function pointers, several different functions could be called via a given function pointer at a specific location in the source depending on the function pointer's value at run-time. Static data flow analysis can yield an estimation which functions may be called. The set of possibly called functions is usually based on conservative assumptions. At least compilers have to make conservative assumptions to avoid generation of erroneous code. To which degree reverse engineering analyses have to be conservative depends on the task at hand and is still an open question in general. Because we do without data flow information, we have to ignore calls via function pointers. This does not cause any harm, for the contribution of calls to atomic component detection is generally rather limited and calls via function pointers are even less relevant, at least for the main representatives abstract data type and abstract data object. The only approaches that consider *calls* are *Arch* (Section 5.9) and *Similarity Clustering* (Chapter 7) and even for these, a statistical analysis revealed that *calls* have not much weight (Section 7.6.1). However, function pointers can be a problem when subsystems are to be detected as one of our case studies indicates (Girard, Koschke, 1997a).

Actually, there are more than these two kinds of calls. It is also possible to switch between the contexts of functions whose activation record is currently on the runtime stack by means of *setjmp* and *longjmp* under Unix. This is often used to implement exception handlers. However, since this is not part of the language and depends completely on the runtime system, we will ignore such calls.

Functions make use of types by mentioning them in their signature or using them to declare local variables. For the former, we introduce *signature* edges and for the latter, *local-obj-of-type* edges to the named base types of the parameters and function result, or local variables, respectively (see Section 4.2.3).

Types can even occur as type of an expression at any level though they are not explicitly mentioned in the body of the subprogram. For example, in

```
typedef struct S {int a; } T;
T f ();
void g () {
  int i;
  i = f().a;
}
```

function *g* accesses a component of type *T* though it has neither parameters nor local variables of type *T.* Such a type relationship will be reflected as *local-obj-of-type* since it is analogous to declaring a temporary local variable that receives the intermediate result.

4.2.8 References

References to local variables, formal parameters, and global variables are represented as described in Section 3.1.2.2. The references captured by the resource usage analysis are only an approximations of the actual references. This section describes the approximation and discusses the divergent points of view of the fields of compiler construction and reverse engineering on references.

4.2.8.1 Reference Information Approximation

The captured references are relationships between subprograms and declared entities that can be explicitly found in the code. They represent only an approximation of the real sets/uses/takes-address-of relationships. First, it could be that a

reference occurs in a path of the control flow that will never be taken (our approximation is an overestimation then) and, second, an object could be referenced via an alias (an object is **aliased** if it has more than one access path by means of static name binding or dereferenced pointers; in the following, no distinction is made between access paths via different static names or dereferenced pointers). In the case of hidden references via aliases, our approximation is an underestimation since hidden references are not detected by the resource usage analysis. Extensive control and data flow analyses would be needed to detect potential hidden references. However, these analyses can only yield an estimation since possible control flow as well as aliasing are statically undecidable in general because they may depend on values that can only be established at run-time.

Fortunately, it is not a problem for the purpose of atomic component detection when a reference occurs in an unfeasible path. Any reference, no matter whether it is actually executed, creates a relevant static dependency and therefore must not be ignored. Furthermore, hidden references by way of aliasing to an object are generally not relevant to atomic component detection. Remember that relevant objects to the resource usage analysis are either global objects or types. The relationships for the latter are caused by parameters if it is a *signature* relationship or by local objects if it is a *local-obj-of-type* relationship. Therefore, the following combinations of aliasing that could be relevant are possible:

1. aliasing between a global object and a parameter
2. aliasing between a global object and a local object
3. aliasing between global objects
4. aliasing between parameters
5. aliasing between local objects
6. aliasing between parameters and local objects

Ad (1): If a global object is aliased by a formal reference parameter but the subprogram references the global object only by means of the formal reference parameter, then it is quite likely that the subprogram does not have anything to do with the global object. The subprogram is more general and handles different objects, otherwise the programmer would have hardly introduced a parameter. If the subprogram references the global object also directly, then ignoring aliasing may result in undetected additional references to the global object by means of

the formal parameter. However, at least the directly visible accesses are taken into account.

Ad (2): An alias between a global object and a local object can arise if one's address is taken and assigned to the other one. Assigning the address of a local object to a global object will cause run-time errors when the local object's memory location is accessed by way of the global object after the function ended. It could be used to transfer the local object to another function that is transitively called by the current function. Then the activation record of the function would still exist and hence, the local object be defined. Still, there should be a strong argument why the local object was not passed as parameter (maybe because the call went through a library whose functions have a standard signature), otherwise this could not be justified to the maintainer. Therefore, I do not believe this has any significance.

Assigning the address of a global object to a local object is often used by C programmers for looping over global arrays or for shorthanded accesses to components as in:

```
char g1[10];
void f1 () {  /* loop over array */
  char *p;
  for (p = g1; *p != '\0'; p++) *p = 'a';
}
struct {int c;} g2;
void f2 () {  /* shorthanded access */
  int *i = &g2.c;
  *i = 1; /* use *i instead of g2.c */
}
```

These schemes in fact lead to a wrong mapping to the resource usage graph because the modification of *g2.c* by *f2* via the dereferenced local variable *i* will be re-directed to an internal access to the type of *i* since local variables are not represented in the resource usage graph. However, at least the fact that the addresses of the variable *g2* and the record component *g2.c* are taken is captured. Likewise, the fact that *f1* modifies *g1* is not detected; only that *f1* takes the address of *g1* is captured.

Ad (3): Ignoring aliasing between global objects causes a miss of the interrelation between subprograms that access apparently different, yet aliased global objects. However, only if the aliasing is permanent, the miss is really problematic; otherwise these subprograms may indeed be viewed as only weakly coherent. Fortunately, aliasing among global objects is a very rare phenomenon.

Ad (4): Aliasing between parameters would occur if both parameters were reference parameters and the same actual argument is passed to both of them. But again this is not the typical way of using the function, otherwise the function would have only a single parameter instead of two aliased parameters, and therefore the function should be treated in its general case in which the two parameters are not aliased.

Ad (5) and (6): Aliasing between local objects can only take place if the address of a local object is taken and assigned to another local object because subprograms cannot be nested in C. This will not let the references with respect to the type go unnoticed. Remember that the *local-obj-of-type* is a relationship between a function and a type induced by any local object of this type. If such a relationship exists, any reference to a record component of this type will be considered (except for the references to record components of parameters); it does not matter which actual local object it is and whether the type of the actually referenced local object is the one of the *local-obj-of-type* relationship. Similarly, if a local object and a parameter are aliases, the access to the record component of the type will be recorded in all cases either by way of the *local-obj-of-type* or the *signature* relationship.

4.2.8.2 References and Dereferences

From a compiler's point of view, all occurrences of *b* but the first one in the following statements are uses of the variable *b*:

```
struct {int a; } *b;/* 1 */   b = 0;
                    /* 2 */   ...= *b;
                    /* 3 */   *b = ...;
                    /* 4 */   b->a = ...;
```

In the expressions of 2-4, the value of *b* is used to address the designated object of *b*; that is why a compiler considers case 3 and 4 as usages of *b*. However, from a

program understanding point of view, the designated object of *b* is primarily relevant and a programmer will abstract from the difference of *b* and its designated object when reading this code. Consequently, from the point of view of a reverse engineer, cases 3 and 4 can pragmatically be considered settings of *b.* The distinguishing factor between cases 3 and 4 and the usage of *b* in 2 is that *b* occurs on the left hand side of the assignments in 3 and 4.

The resource usage analysis used to produce the results described in this thesis follows the compiler's point of view, hence, captures the actual semantics of dereferenced variables. Future extensions should investigate whether following the reverse engineer's point of view is more appropriate.

4.3 Information Hiding in C

In previous sections, C was criticized as a language that provides only limited support for information hiding. Nevertheless, information hiding can be realized by using the rudimentary means offered by C. Unfortunately, these means are often not used by programmers. This is probably not because the language does not support information hiding in a more direct way, but because programmers ignore these means, which suggests the conclusion that they would neither follow these principles in a more abstract language. This section shows how they could apply the principles of information hiding by the existing means C provides.

The lack of modules in the language must be compensated by header files for the interface and files with the actual code; let us call these **body file**s. The header file lists all extern declarations, i.e., exported declarations, and nothing else. Local function and object declarations can be hidden in the body by the keyword static. Thus, we can easily realize an abstract data object in C by putting the global objects (declared as static) into the body file. Unfortunately, this is not so easy for abstract data types.

Header files are ordinary files without semantics. There is no private section, as in Ada, where we could hide private type declarations. On the other hand, for reasons of efficient separate compilation, the type information has to be imported by any other body file that uses the type; this is needed to ascertain the needed stor-

age for objects of this type. For these reasons, we cannot use arbitrary data structures for abstract data types in C. However, the opaque types of Modula-2 can be simulated by exporting a struct name and a pointer to this struct name which represents the abstract data type. The full declaration of the struct follows in the body file and is therefore hidden for all clients. This is illustrated in Figure 4-2.

List.h

```
struct list;
typedef struct list *List;
extern void insert
  (List l, Item i);
```

List.c

```
struct list {List next;
             Item i; };
void insert (List l, Item i)
{ … }
```

Figure 4-2. **Example abstract data type *List* in C.**

Actually, the type is not really abstract because it is explicitly declared as a pointer with the consequence that assignment and comparison of such abstract data types have always reference semantics. However, clients of this module are not able to access the internal fields of this abstract data type.

If these programming styles are observed and if any simulated module contains only one concept, the concepts and its accessor routines can immediately be found. Unfortunately, these means are rarely used. Current practice is to export global variables and the full type structure, to distribute accessor routines among different modules, and to put several distinct concepts into the same module.

4.4 The Language and Other Factors

It is a trivial realization that the programming language has to provide means to express what we want to detect. However, these means can be very rudimentary. Talented programmers are thoroughly able to simulate higher concepts with only little support by the language. Nevertheless, the degree of support by the programming language does determine the chances of an automatic technique. There are two extremes. At one end, there are assembly languages. None of the techniques that we will get to know in Chapter 5 works for such primitive languages. There should be at least subprograms, objects, and types in the language. How-

ever, even assembly languages have macros, symbolic addresses, and data specifications that could be used as a starting point to find substitutes for subprograms, types, and objects. At the other extreme, there are modern programming languages, such as Ada, Modula-2, and C++, that provide the means for data abstraction and information hiding in general. However, there is no guarantee that these means are properly used by programmers, i.e., even for these languages the techniques presented in this thesis could be helpful.

There are two main atomic components we are searching for: abstract data objects and abstract data types. If we want to detect abstract data objects, there should be ways to express state in the language. Pure functional languages, for example, do not support the concept of state. However, in all procedural languages, we have global variables whose values can be manipulated, and these are the languages that are used in practice. The language need not have a way to specify the relatedness of the global objects; such means are important to abstract data types where the constituents must be explicitly united in a type. Our experiences with the systems we studied indicate quite the opposite. An abstract data object mostly consists of a set of objects that have primitive types. Programmers avoid the effort of introducing a new record type whose components are the constituents of the abstract data object, though this would make their connection obvious.

If an abstract data type is to be detected, there obviously has to be a type in the first place and the user should be able to define new types using type constructors. Among common type constructors, records are the most important ones because they allow to realize heterogeneous concepts. Records have been part of all the most popular programming languages since the early 1960s when they were introduced by COBOL. Notable exceptions are past versions of Fortran. For Fortran77, one has to find the connections of single base types that together form a heterogeneous abstract concept before the heuristics described in Chapter 5 can be brought into play.

The phenomenon of apparently distinct elements together forming an abstract concept is called **tupling** (Koschke et al., 1997). My experiences with the large Fortran77 library *Spicelib* of the Jet Propulsion Laboratory are that programmers use such tuples as if they were records with a fixed order of contiguous components, i.e., if they occur in a parameter list, the components of the tuples are

mostly in the same order and there is no constituent of another tuple within the sequence of components. Other indicators derived from control and data flow analysis could be used to detect tuples, e.g., accesses to some components of the tuple may always be control-dependent on other components of the tuple. Furthermore, domain knowledge is probably a key factor for detecting tuples. Records and likewise simulated records, i.e., tuples, are the most relevant instruments to model concepts from an application domain since most such concepts are heterogeneous. Knowing the domain is often crucial for deciding whether certain constituents together form a unity when this unity corresponds to a concept of the application domain. Tupling is its own field of research and it was only stated as a problem but no solutions have been proposed so far. I will not discuss tupling in this thesis, yet, essential observations of this field are also relevant to atomic component detection and will be summed up here:

- the target of the search is a concept that could convey important information to the maintainer
- the concept is not properly specified because the programming language does not provide the means to model this concept or a programmer has ignored these means
- if the language does not provide adequate means of expression, programmers find ways to simulate these means in part
- there are different sources of information that can be leveraged to find the concept:
  - the typical ways how programmers simulate the lacking means of expression
  - control and data flow analysis
  - domain knowledge

Most techniques described in this thesis mainly follow the first path and try to find atomic components by the way the programmer could have organized them in C. To some degree, they are based on assumptions that could be derived by control and data flow analyses, but these assumptions are only rough approximations and are not validated by the techniques. Two techniques indeed perform control flow analyses on the call graph, namely, dominance analysis and strongly connected component analysis. However, other control flow analyses are not being used. Data flow analyses in particular are not exploited at all. Neither is domain knowledge, other than using the conventions of programmers to convey

the information with the rudimentary means of the language, leveraged by any of the automatic approaches. The method that is proposed in Chapter 9 integrates the user in the detection process and is heading toward leveraging domain knowledge. However, a completely automatic approach incorporating domain knowledge is not yet in sight (see Section 11.2).

Part II Automatic Techniques

Chapter 5 Basic Techniques and Metrics of Component Detection

Many (semi-)automatic techniques for atomic component detections have been proposed in the literature. However, no attempt has been made so far to compare and evaluate these methods. This chapter describes published techniques and some extensions we made. It also unifies and classifies these techniques. An evaluation of the techniques will follow in Chapter 6.

5.1 What All Techniques Have in Common

At a higher level of abstraction, an abstract data type consists of a domain of values for the type and some allowed operations on that type. In an implementation of an abstract data type, the domain of values is implemented by a data structure which is read and set by routines - its operations. The user of an abstract data type can declare objects of that type and pass them as actual parameters to the operations. Consequently, it is a necessary prerequisite for operations of an abstract data type to mention the data type in their signature, i.e., their parameter list or their return type in the case of functions. That is, all routines with a data type *T* in their signature are candidates for an operation of the abstract data type *T*. However, this prerequisite is necessary but not sufficient. Some routines simply pass a value of *T* to other routines and are not true operations of *T*. Many routines have more than one parameter type so that it is necessary to decide which one they belong to. For all kinds of routines which convert one type into another type this can be hard to judge. Sometimes - especially in programming languages that do not provide record types such as Fortran77 - one even has to look at several base

types of the underlying programming language in a parameter list to form one abstract data type. For example, one can have a stack implementation that passes two parameters, one for the stack contents realized by an array and one for the stack pointer implemented by an integer type.

Similarly, an abstract data object represents an abstraction of a state and the operations that manipulate the state. The state is implemented by a set of global objects. These objects are set and used by the operations of the abstract data object. Most of the time, programmers do not make the effort to group the global objects of an ADO together as components of a record structure to make the connection of the objects obvious. In many old programming languages they even would not have a chance to do so because user-defined data types are not supported. Even in programs written in modern programming languages, one often finds accesses to these global objects by routines that do no belong to the ADO because of efficiency considerations. All that makes it difficult to find the objects that together make up the abstract state and the routines that really represent the ADO's operations.

Considering these facts, it is obvious that the naïve approach of grouping types together with all routines whose signature refers to them and of aggregating objects with all routines that set or use them leads to erroneously large candidate components. This strategy is discussed in the next subsection for global object references. In the rest of this section, we present heuristics proposed to avoid erroneously large ADT and ADO candidates by going beyond the simple reference criterion. Some of them can detect ADTs as well as ADOs, some of them are specialized in one type of atomic component. Those that can detect both types of atomic components merge the results of ADO and ADT detection into a hybrid candidate if there is a routine that belongs to both an ADT and an ADO.

5.1.1 Working Example

The C code example in Figure 5-1 will be used to illustrate the various heuristics in the following. It consists of an abstract data type *List of Item* with the accessor routines *empty*, *first*, *rest*, and *prepend* and an abstract data object *stack of Item* with the accessor routines *init*, *push*, *pop*, and *size* whose implementation is based on the abstract data type *List*. Type *Item* is declared in a file of its own, whereas the *List* and *stack* concept are declared in the same file though it would probably

have been better to separate these distinct concepts into two files. Note also the violation of the information hiding principle in the accessor function *size* of *stack*. Because there was no function *length* available for *List* that a programmer could have used, the programmer implemented *size* by a direct access to the component *length* of type *List*. Hence, function *size*, which actually belongs to the abstract data object *stack*, breaks the encapsulation of *List*.

An excerpt of the resource usage graph for this example containing all nodes and most edges is shown in Figure 5-2 (*actual-parameter-of*, *of-type*, *component references* are not shown). The example in Figure 5-2 also demonstrates that resource usage

```
/*--- file list.c ---*/
typedef struct {
    int len;
    Item cont[100];} List;

List empty () {
  List result;
  result.len = 0;
  return result;
}
Item first (List l) {
return l.cont[l.len-1];
}
List rest (List l) {
  l.len--;
  return l;
}
prepend (List *l, Item t) {
  l->cont[l->len] = t;
  l->len++;
}
```

```
/*--- file list.c ---*/
static List stack;
void init () {
  stack = empty ();
}
void push (Item i) {
  prepend (&stack, i);
}
Item pop () {
 Item result = first
(stack);
 stack = rest (stack);
 return result;
}

int size () {
 return stack.length;
}
```

```
/*--- file item.h ---*/
typedef ... Item;
```

Figure 5-1. **Example C program.**

graphs can become quite complicated even for short programs. As a matter of fact, resource usage graphs even for medium size programs at the 30KLoc level cannot reasonably be visualized anymore.

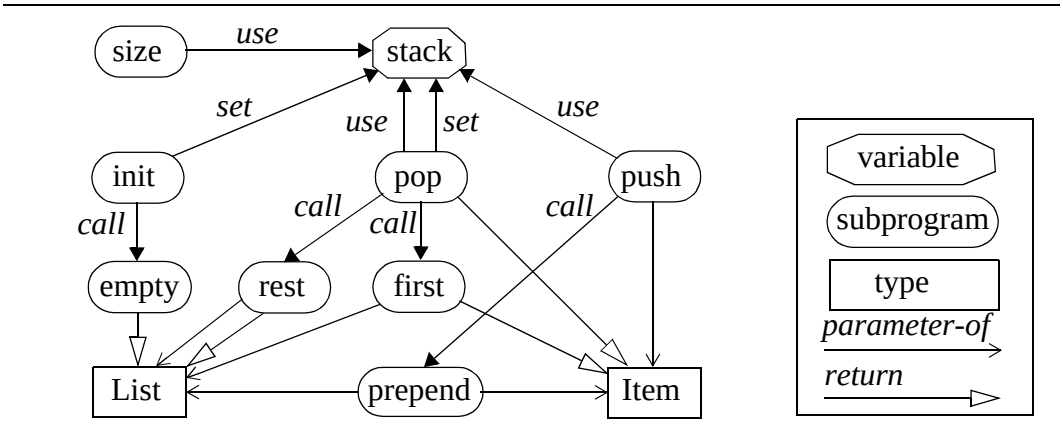


Figure 5-2. **Resource usage graph for the example in Figure 5-1.**

5.1.2 Characteristics of the Techniques

From a mathematical point of view, all the techniques are basically clustering methods that differ in the underlying cluster criterion, the clustering domain and range, and whether the clusters are disjoint. A cluster is basically a set of entities that meet the underlying clustering criterion of the method (the terms cluster, group, and candidate are interchangeable).

Clustering criterion. The clustering criterion is a heuristic approximation of "real" criteria for atomic component composition. Because each technique has its own criterion, this is the most distinguishing characteristic. Other properties of the clustering methods may be shared.

Domain and range. The domain of a clustering method specifies what kind of entities and relationships are considered for clustering. The range, on the other hand, specifies what type of entities can be grouped together at all and hence what kind of atomic components can be detected.

Disjoint clusters. Most methods produce disjoint clusters, but there are exceptions.

In the following description of the clustering methods, we will state the domain, range, clustering criterion, and whether the methods produce disjoint clusters. In doing so, the domain will be specified in terms of the views listed in Table 3-5 on

page 68. The range will be stated as the kind of atomic components the method can detect:

- Abstract data types (ADT) consist of user-defined types and subprograms.
- Abstract data objects (ADO) consist of global objects and subprograms.
- Hybrid components (HC) consist of global objects, user-defined types, and subprograms.
- Related subprograms (RS) consist of subprograms only.

Furthermore, we will give the reference to the originators and specify the extensions or modifications we made. Since the extensions and modifications are part of joint work within the Bauhaus project, they are mainly common ideas of Jean-François Girard, Georg Schied, and me.

5.2 Global Object Reference Heuristic

Name	**Global Object Reference**
Reference	Yeh et al. 1995
Domain	Object Reference View
Range	ADO
Disjoint Clusters	Yes

Clustering criterion. Global objects and all the routines that reference these objects, regardless of where they are declared, are grouped together.

In the case of a global error variable used in many parts of the system, this approach will collapse a large part of the system into one atomic component. Yeh et al. propose to exclude frequently used objects from the analysis to avoid this unwanted effect.

Example. Applied to the example of Figure 5-1, *Global Object Reference* would find the ADO {stack, init, push, pop, size}.

Algorithm. For *Global Object Reference*, we can use the generic algorithm 5-1, which will also be used for other methods that produce disjoint clusters. The algorithm iterates over the subprograms and groups them with their relevant connected entities. What a relevant connected entity is depends upon the respective technique; in terms of the algorithm, this is decided by a generic parameter that yields for each subprogram all entities that have to be part of the same atomic component as the subprogram. This function could also exclude frequently used objects as proposed by Yeh et al. for the *Global Object Reference* heuristic.

Generic parameter:

- function *connected_entities*: Entity → set of Entities

Input:

- input view V

Output:

- disjoint clusters

Algorithm:

1. *put each base entity in V into a set of its own:*
   for each entity E **in** V **loop**
   new_set (E);
   end loop;
2. *clustering*:
   for each entity E **in** V **where** subprogram (E) **loop**
   for each entity E' **in** connected_entities (E) **loop**
   union (find (E), find (E'));
   end loop;
   end loop;
3. *results*:
   each remaining disjoint set is a cluster

Algorithm 5-1. **Generic algorithm to detect disjoint atomic components.**

Algorithm 5-1 uses the union-find implementation for disjoint sets (Hopcraft and Ullman, 1983) where:

- *new_set (e)* defines a new set {*e*}
- *union (s1, s2)* unites the two sets *s1* and *s2*; after the call, the two set identifiers denote the same set, i.e., *s1*∪*s2*.
- *find (e)* yields the set that contains *e*; since all entities will be initially put into a set and since the sets are disjoint, there is exactly one such set

To instantiate the generic algorithm for *Global Object Reference*, the function *referenced_objects* as defined in Table 3-4 on page 65 serves as the actual parameter for *connected_entities*, which returns all global objects referenced by a given subprogram:

$$\text{connected\_entities}(S) = \text{referenced-objects}(S) \quad \textbf{(5.1)}$$

5.3 Same Module Heuristic

Name	**Same Module**
Reference	Girard, Koschke, 1997
Domain	Object Reference View + Signature View
Range	ADO, ADT, HC
Disjoint Clusters	Yes

In the ideal situation, i.e., when the system is properly decomposed, each module contains one single atomic component. When we count on good design, we can group all declarations of a module together to an atomic component which represents the abstract functionality of the module. This is the underlying clustering criterion of the *Same Module* heuristic.

Clustering criterion. All related subprograms, user-defined data types, and global objects that are declared in the same module are grouped together. A subprogram is related to a data type when the data type occurs in the signature of the subprogram. Likewise, a subprogram is related to an object when the subprogram references the object.

This heuristic was first published in a case study by Girard and Koschke (1997). However, the initial idea for this heuristic stems from Erhard Plödereder (1997).

In the case of Ada, a package body and its specification would form a module. In C, modules do not exist, but programmers simulate the lacking concept by a header file *f.h* for the specification and a C file *f.c* for the body. *Same Module* assumes that programmers are disciplined and follow this convention.

The *Same Module* heuristic will fail when parts of the abstract functionality of a module are implemented elsewhere. However, it will still find individual atomic components in very large modules that have many different logical functions unless they are implemented by overlapping sets of subprograms, types, and objects.

Example. For the example of Figure 5-1, *Same Module* would propose one ADT {*List, empty, prepend, first, last*} and one ADO {*stack, init, push, pop, size*} despite of the fact that these declarations are in one common file: The only connections between parts of the stack and parts of the List are call relationships, which are not considered by the *Same Module* heuristic (see Figure 5-2 on page 112). However, if *Item* were declared in the same file, *Same Module* heuristic would have assumed one big hybrid component consisting of the union of the two sets above (including type *Item*) because the stack ADO's accessor routines also mention type *Item* in their signature.

Algorithm. *Same Module* can be implemented by instantiating the generic algorithm 5-1 for disjoint clusters where *referred-entities* (defined in Table 3-4 on page 65) is used for *connected_entities*; *referred-entities* returns the signature types and the referenced objects of a subprogram; the module in which an entity, *X*, is declared, is denoted by *module*(*X*):

$$\text{connected\_entities}(S) = \text{referred-entities}(S) \cap \{X \mid \text{module}(X) = \text{module}(S)\} \quad \textbf{(5.2)}$$

5.4 Part Type Heuristic

Name	Part Type
Reference	Liu and Wilde, 1990
Domain	Signature View + Type Composition View
Range	ADT
Disjoint Clusters	Yes

Often, we find abstract data types that represent some sort of container of other abstract data types. For example, queues as containers of processes, or an account containing data about its owner and the deposited money. For such abstract data types there usually exists an operation that takes an element and puts it into the container. For a process queue, for example, there will be an *insert* routine with two arguments: the process to be inserted and the queue itself. Even though both types are mentioned in its signature, we would not consider *insert* to be an operation for processes but for queues. The *Part Type* heuristic reflects this perception. It is based on the part-type relationship which was already defined in Section 3.5.4.3.

Clustering criterion. *Part Type* groups a routine with those types in its signature that are not a part type of another type in the same signature.

The basic assumption is that a part type is actually used to be put into its container or to be retrieved from it. It does not check this assumption. Data flow analysis could validate this assumption.

Example. This can be illustrated with the following declarations:

```
typedef ... Item;
typedef struct {int len; Item contents [100];} List;
void prepend (List *l, T t);
```

Here, *Item* is a part type of *List*. That is why *prepend* would be an operation of *List* according to *Part Type* and not of *Item* though both types are mentioned in the signature of *prepend*. The detected ADT consists of {*List*, *empty*, *prepend*, *first*, *last*}.

Algorithm. Again, the generic algorithm for disjoint clusters can be used to implement the *Part Type* heuristic. The *connected_entities* of a subprogram are all signature types excluding part types:

$$\text{connected\_entities}(S) = \{T \mid T \in \text{signature-types}(S) \wedge \neg\exists(\tilde{T} \in \text{signature-types}(S))\text{part-type}(T, \tilde{T})\} \quad \textbf{(5.3)}$$

5.5 Same Expression Heuristic

Name	Same Expression
Reference	Unpublished; Rainer Koschke
Domain	Object Reference View + Same Expression View
Range	ADO
Disjoint Clusters	No

Often, the state of an abstract data object is implemented by a set of separate global objects instead of a single record object. For example, an ADO *stack* could be based on the following variable declarations:

```
Item contents [100];
int stackpointer;
```

However, the fact that the apparently separate constituents of an abstract data object contribute to a common purpose often becomes visible when they occur in the same expression, especially when one of the constituents is a composite data structure and the other constituent is used as some kind of selector for the composite data structure. In the above example, we can expect to find an expression such as:

```
contents [stackpointer]
```

The level at which the selector appears does not matter. The stack example could also be as follows:

```
Item **contents;
int stackpointer;
```

```
...(*contents)[stackpointer]...
```

Likewise, common occurrence in a mathematical expression can be a hint for relatedness:

```
printf (“area = %d\n”, length * width);
```

We will also consider objects jointly occurring in the actual parameter list of the same call as being in the same expression, extending the definition of *same expression* given in Section 4.2.4. This extension has turned out be useful in the evaluation of the *Same Expression* heuristic reported in Chapter 6. That is, the three variables in the following code will also be considered to be in the same expression:

```
set_coordinates (window, length, width);
```

As it was already discussed in Section 4.4, the phenomenon of apparently distinct actual parameters together forming an abstract concept is called *tupling* (Koschke et al., 1997). Tupling primarily occurs in older languages, like Fortran77, that do not have record types. However, as our experiences with the systems investigated indicate, examples for tupling can also be found in languages with record types, like C.

Because the *Same Expression heuristic* is aimed at detecting composite abstract data objects consisting of several objects, it ignores clusters with only one object, in which no *same expression* relationship can occur.

Before we can specify the clustering criterion, a definition is needed. A **connected graph component** is a subgraph whose nodes are all (transitively) connected; in other words, each node is reachable from all other nodes within the subgraph where the direction of edges is ignored.

Clustering criterion. All objects that are in the same connected graph component in the *same-expression view* (see Table 3-5 on page 68) are grouped together with all the subprograms that reference at least one of these objects in the object reference view. Subprograms may belong to distinct clusters. Clusters with only one object are ignored.

Algorithm. Algorithm 5-2 computes non-disjoint clusters based on the *same expression* relationship among global objects.

Input:

- input view V

Output:

- a set of non-disjoint clusters

Algorithm:

1. *extract all objects:*
   Vars:= {var | object (var) ∧ var ∈ V};
   Clusters := ∅;
2. *clustering*:
   while Vars ≠ ∅ **loop**
   Var := arbitrary_element (Vars); -- *choose an arbitrary element*
   C := transitive_closure (neighbors (Var, {same_expression}));
   Vars := Vars \ C;
   if | C | > 1 **then**
   Clusters := Clusters ∪ {C ∪ referencing_subprograms (C)};
   end if;
   end loop;
3. *results*:
   Clusters is the result

Algorithm 5-2. ***Same Expression*** **heuristic algorithm.**

Example. In the working example introduced in Section 5.1.1, there is only one global object and, therefore, the *Same Expression* heuristic is not applicable. Let us consider the example in Figure 5-3 instead that contains two abstract data objects; the first one is shaped by a solid line, the second by a dashed line. Object *c* is neither part of the first nor the second abstract data object though it is accessed by accessor routines that belong to these abstract data objects. The example illustrates also that overlapping candidates can result.

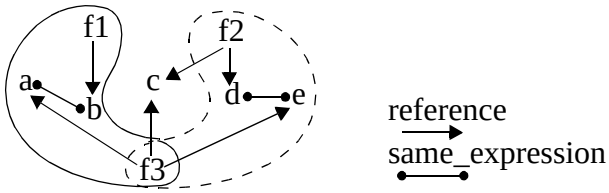


Figure 5-3. **Example of groupings by *Same Expression* heuristic.**

5.6 Internal Access / Non-Abstract Usage Heuristic

Name	**Internal Access / Non-Abstract Usage**
Reference	Yeh et al., 1995 (extensions: Girard, Koschke, 1997)
Domain	Signature View (extensions: Object Reference View) where edges are annotated with internal access information
Range	ADT (extensions: ADO, HC)
Disjoint Clusters	Yes

The purpose of an abstract data type is to hide implementation details of the internal data structure by providing access to it exclusively through a well-defined set of operations. The idealized encapsulation principle entails that all routines that access internal components of the abstract data type are considered to be the data type's operations which is exactly the attitude of the *Internal Access* heuristic. An **internal access** for a record type is any record component selection.

Extensions. Yeh et al. proposed to consider internal access to record types only, but the same principle can be applied to arrays and pointers as well:

- if T is an array then any index subscript is an internal access;
- if T is a pointer then any dereference is an internal access.

Originally, the subprograms are associated with those types in their signature whose corresponding formal parameter is internally accessed in the body of the subprogram. However, if we check only whether parameters are internally accessed, a frequent pattern in the presence of pointers that simulate call-by reference in C will be missed: The value of a simulated call-by reference parameter is

assigned to a local variable which is then used within the rest of the body. Before the function ends, the value of the local variable is assigned to the designated object of the pointer. This way, the programmer does not have to use the dereference operator for each occurrence of the reference parameter:

```
typedef struct {int a;} T;
void f (T *pt) {
  T t = *pt;
  t.a = ...; /* internal access to local variable */
  *pt = t;
}
```

This pattern disguises the fact that the designated object of the parameter is internally accessed. Data flow analysis would reveal the internal access. As an approximation without need for data flow analysis, we count as internal access any access to the internal parts of a local variable of the same type as the parameter. We have found that, for C, our approximation matches reality in most cases.

Though originally only proposed for finding abstract data types, the *Internal Access* heuristic can also be extended to find abstract data objects in cases in which state is implemented by record, array, or pointer variables, i.e., we can also take internal accesses to composite global objects into consideration.

Furthermore, what an internal access basically represents is a non-abstract usage of a type. The data structure, which should have been hidden according to the information hiding principle, is no longer transparent. The same takes place when a predefined operator is applied to an object. In this case, the programmer also exploits the knowledge about the object's data structure. As a conclusion, we can widen the definition of internal access to include applications of standard operators to local and global objects as well as parameters. In doing so, the term internal access is no longer appropriate; that is why we will use *non-abstract usage* instead. To sum it up, **non-abstract usage** includes:

- internal access for composite data structures as defined above
- application of predefined operators to global objects as well as parameters and local objects of a user-defined type

Clustering criterion. The *Internal Access/Non-Abstract Usage* heuristic associates subprograms with all its non-abstractly used objects and signature types (a

type *T* is non-abstractly used when there is a non-abstract usage of any expression – excluding global objects – that has type *T*; this includes references to parameters, local variables, or intermediate expressions).

Note that the clustering criterion does not consider every non-abstract usage of local variables but only of those local variables whose type also appears in the signature of the subprogram. The reason for this restriction is derived from the necessary criterion for accessor functions of an ADT to mention the type in the signature: A subprogram with a local variable of type *T* that does not mention *T* in its signature is a client of the abstract data type *T* but not an accessor function. Relaxing this restriction would lead to erroneously large candidates for systems with limited or no information hiding.

Algorithm. *Internal Access* can be implemented by using the generic algorithm 5-1. For the generic parameter *Connected_Entities*, the instantiation uses function *internally-accessed* (*S*) that yields all internally accessed objects and types of a subprogram, *S*, in the union of object reference and signature view (where we assume that reference and signature-type edges have an attribute *non-abstract* that is true if there is a non-abstract usage by the subprogram S; see also Section 3.4.3 for a description of attributes and Section 3.5.2 for neighbor functions restricted by attributes):

$$\text{connected\_entities}(S) = \text{internally\_accessed}(S) \quad \textbf{(5.4)}$$

$$\text{internally\_accessed}(S) = \text{successors}(S, \{\text{reference, signature-type}\}, \text{non-abstract}) \quad \textbf{(5.5)}$$

Example. In the example of Figure 5-1, *empty, prepend, first,* and *last* would be added by *Internal Access* to the type *List* because of their internal access to it. *Internal Access* would only detect a part of the ADO {*stack, init, push, pop, size*} because the routines *init, push, pop* do not access the internal record components of *stack*. Interestingly enough, the part {*stack, size*} is detected because the programmer breaks the information hiding principle while *init, push,* and *pop* are not considered part of the ADO because information hiding is obeyed.

The two atomic components of Figure 5-1 are an example for layered atomic components, i.e., a component is built on top of another component. Layering is not only used among atomic components but also often within large components.

Frequently within large components, there are a few core functions accessing the internal data structure and several higher-level functions whose services are implemented using the core accessor functions. *Internal Access*/*Non-Abstract Usage* can only identify the core functions within layered components and only the lower-level components when components are built on top of each other.

5.7 Delta-IC

Name	**Delta-IC**
Reference	Canfora et al., 1993, 1996
Domain	Object Reference View (extensions by Rainer Koschke: Signature View where edges are annotated with internal access information)
Range	ADO (extensions: ADT, HC)
Disjoint Clusters	No

High cohesion in the case of an abstract data object *S* implies that each of the subprograms in *S* references many objects of S; low coupling implies that each of the subprograms of *S* references only very few objects that do not belong to *S* and that only few subprograms from outside of *S* reference objects of *S*. The approach proposed by Canfora et al. is heading in this direction. It basically consists of two parts. At first, objects and subprograms are clustered to ADOs according to a specific usage pattern. Then all resulting clusters are rejected whose internal connectivity is below a given threshold. The internal connectivity metric proposed by Canfora et al. is described below.

The clustering pattern and the evaluation metric is defined on the object reference view that describes the usage of global objects by subprograms. They can be explained more easily in terms of the following definitions, given a subprogram *S* and a global object *V* (we are using the concepts introduced in Section 3.1; Canfora et al. describe their approach in a slightly different, yet equivalent way):

subprograms related to S are all subprograms which set or use referenced objects of S:

$$\text{subprograms-related-to (S)} = \bigcup_{e \in \text{ referred-by (S)}} \{F \mid F \in \text{ refer-to(e)}\} \quad \textbf{(5.6)}$$

where *refer-to (e) = referencing-subprograms (e)* and *referred-by = refer-to*$^{-1}$, hence, *referred-by* (*S*) = *referenced-objects (S)*.

The reason why *refer-to*/*referred-by* are introduced here — instead of using *referencing-subprograms/referenced-objects* directly — is that *Delta-IC* is extended to types by re-defining *refer-to* below (*referred-by* is always defined as *refer-to*$^{-1}$).

closely-related subprograms of S are all subprograms which set or use ***only*** referenced objects of S:

$$\text{closely-related-subprograms } (S) = \bigcup_{e \in \text{ referred-by } (S)} \{F \mid F \in \text{ refer-to}(e) \wedge \text{referred-by}(F) \subseteq \text{referred-by}(S)\} \quad \textbf{(5.7)}$$

Example. Given the object reference view of Figure 5-4 and *F* as the subprogram under consideration, then the objects *referred by* F are $\{v_1, v_2\}$, the *subprograms related* to F are $\{F, f_1, f_2, f_3\}$, and the *closely related subprograms* are $\{F, f_1, f_2\}$.

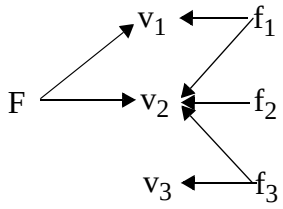


Figure 5-4. **Example objects reference graph.**

The candidate that is proposed as an abstract data object consists of all *closely related subprograms* of the given subprogram *S* plus the objects *referred by S*, i.e., all objects set or used by *S*:

$$\text{candidate-cluster (S)} = \text{closely-related-subprograms (S)} \cup \text{referred-by (S)} \quad \textbf{(5.8)}$$

Example. In the example of Figure 5-4, the candidate cluster is $\{v_1, v_2, F, f_1, f_2\}$ for the given subprogram *F*. Note that the proposed clusters depend upon the given subprogram. Suppose *F* also referenced object v_3, then the cluster for *F* would be $\{v_1, v_2, v_3, F, f_1, f_2, f_3\}$; from the perspective of f_3 we would get the cluster $\{v_2, v_3, f_2, f_3\}$. Thus, clusters can overlap.

The candidate cluster is ranked by the internal connectivity metric and only proposed if this metric yields a value greater than a user-determined threshold. The **internal connectivity measure** (IC) and the **improvement in internal connectivity** (ΔIC) are defined as:

$$\text{IC(S)} = \frac{\sum_{e \in \text{ referred-by (S)}} \left|\{F \mid F \in \text{ refer-to}(e) \wedge \text{referred-by}(F) \subseteq \text{referred-by}(S)\}\right|}{\sum_{e \in \text{ referred-by (S)}} \left|\{F \mid F \in \text{ refer-to}(e)\}\right|} \quad \textbf{(5.9)}$$

$$\Delta\text{IC(S)} = \text{IC(S)} - \sum_{e \in \text{ referred-by (S)}} \frac{\left|\{F \mid \text{referred-by (F)} = \{e\}\}\right|}{\left|\text{refer-to }(e)\right|} \quad \textbf{(5.10)}$$

IC(S) is the portion of references to individual variables of the cluster from subprograms also inside the cluster (closely related subprograms) with respect to the number of all references. If there is no reference from outside the cluster, IC(S) is 1. In the example of Figure 5-4, IC(F) is as follows: (2 + 3) / (2 + 4) = 0.83. The subtrahend in the definition of ΔIC reflects the portion of subprograms that reference only a single variable of the cluster with respect to the number of all references to that variable. In the example of Figure 5-4, the subtrahend of ΔIC is 1/4: f_2 is the only subprogram that accesses a single variable only, namely, v_2, which is referenced by 4 subprograms. Consequently, ΔIC(F) = 0.83 - 0.25 = 0.58.

The underlying intuition of the definition of ΔIC is to have only few references of objects from outside the cluster (this motivates the internal connectivity measure IC) and only few routines in the cluster that reference only one object of the cluster (the second term in the formula for ΔIC). The latter is aimed at clusters whose parts are more tightly coupled. We will discuss this below in more detail.

Clustering criterion. A candidate for a given subprogram *S* is *candidate-cluster*(*S*) where $\Delta IC(S) \geq \Theta$.

Algorithm. The original approach uses the following clustering algorithm (Canfora et al. 1996):

```
repeat
  build object reference view
  for each subprogram S loop
    if  ΔIC(S) ≥ Θ then
      let the user validate candidate-cluster (S)
      if accepted, collapse referred-by (S) into a single representative variable
    else
      slice S using different objects of candidate-cluster (S)
    end if;
  end loop;
until graph contains only isolated subgraphs consisting of an object grouping with one
      or more functions
```

Algorithm 5-3. **Original Delta-IC approach.**

It may be a sign of loose relatedness when a candidate's internal connectivity is below the threshold. The reason may be that the subprograms implement distinct logical functions and therefore reference unrelated objects. The code of such subprograms could be separated into distinct parts that correspond to the distinct logical functions by means of program slicing (Weiser, 1984). This is what Canfora et al. proposed. However, for a pure reverse engineering process, which must not change the system, slicing subprograms is out of the question. Furthermore, if applied only fully automatically and non-iteratively, the nodes *referred-by*(*S*) will not be collapsed. Omitting the slicing and validation steps reduces the outer loop in algorithm 5-3 to one iteration (since the object reference view does not change) and overlapping candidates may result. The subsection on extensions discusses how overlapping candidates can be handled.

The most distinguishing characteristic of this approach is that it first generates clusters and then uses a metric to assess the generated clusters whereas the other techniques simply cluster without rating their candidates. We will take up this idea again in Chapter 9 where we will use several metrics to assess the atomic component candidates within the interactive framework.

Moreover, the threshold used to filter out candidates in this approach offers the user a way to influence the search for atomic components. All other techniques generate always the same candidates; whereas in an interactive environment, the user can play with different thresholds for *Delta-IC* and, hence, get different results. Though the threshold makes the approach more flexible, it also compels the user to search for a reasonable setting. Canfora et al. propose to establish the threshold statistically by a smaller sample of the system. However, it is not yet clear how big the sample has to be in order to allow a usable prediction.

Revisiting the ΔIC definition. The definition of ΔIC consists of two parts. The second part, i.e., the subtrahend of ΔIC, covers substructures of the candidates that consist of only one object and those subprograms that access solely this object. It is motivated by the fact that the clustered variables are collapsed in Algorithm 5-3. Yet, in the first iteration, there are no clustered variables, i.e., each variable stands for itself. In that case, the subtrahend actually represents the internal connectivity of clusters around a single variable that consist of subprograms that only access this variable and no other variable, i.e., if C_v is a cluster that consists of a single variable *V* and all subprograms that only refer to this and no other variable, then the following equation holds (which was neither shown nor mentioned by the original authors):

$$\forall S \in C_V\colon\ IC(S) = \sum_{e \in \text{ referred-by } (S)} \frac{\left|\{F \mid \text{referred-by (F)} = \{e\}\}\right|}{\left|\text{refer-to } (e)\right|}$$

This will now be shown. The set *subprograms-related-to* is the same for all subprograms in C_V. Let *S* be a subprogram of C_v, then *referred-by(S)* = {V} and therefore:

$$\begin{aligned}\text{subprograms-related-to (S)} &= \bigcup_{e \in \text{ referred-by (S)}} \{F \mid F \in \text{ refer-to(e)}\} \\ &= \{F \mid F \in \text{ refer-to}(V)\} \\ &= \text{refer-to}(V)\end{aligned}$$

where *refer-to(V)* depends only upon *V*. Furthermore, it is also true that:

closely-related-subprograms (S)

$$= \bigcup_{e \in \text{ referred-by (S)}} \{F | F \in \text{ refer-to(e)} \wedge \text{referred-by(F)} \subseteq \text{referred-by (S)}\}$$

$$= \{F | F \in \text{ refer-to(V)} \wedge \text{referred-by(F)} \subseteq \text{referred-by (S)}\}$$

$$= \{F | F \in \text{ refer-to(V)} \wedge \text{referred-by(F)} \subseteq \{V\}\}$$

$$= \{F | \text{referred-by (F)}=\{\text{V}\}\} = \{F | F \in C_v\}$$

Thus, we can conclude that:

$$\text{IC(S)} = \frac{\sum_{e \in \text{ referred-by (S)}} |\{F | F \in \text{ refer-to}(e) \wedge \text{referred-by}(F) \subseteq \text{referred-by}(S)\}|}{\sum_{e \in \text{ referred-by (S)}} |\{F | F \in \text{ refer-to}(e)\}|}$$

$$= \frac{|\{F | F \in \text{ refer-to}(V) \wedge \text{referred-by}(F) \subseteq \text{referred-by}(S)\}|}{|\{F | F \in \text{ refer-to}(V)\}|}$$

$$= \frac{|\text{ closely-related-subprograms (S)}|}{|\text{ subprograms-related-to (S)}|}$$

$$= \frac{|\{F | \text{referred-by (F)} = \{V\}\}|}{|\text{refer-to } (V)|}$$

when we have a cluster C_v with one object V and all subprograms that only access V. □

Such substructures around a single variable might be considered a candidate on their own and therefore it could make sense to subtract their internal connectivity from the overall internal connectivity of the composite structure. Yet, this is intuitively not appropriate for the following reasons:

- The decision to consider only subclusters of single variables is arbitrary. Why not considering subclusters with two or more variables?
- Furthermore, one should think that a subprogram that references one variable only and this variable is in the cluster, the subprogram should definitely also be in the same cluster. An example is an abstract data object *stack* based on two global variables *stack_content* (array for the stack content) and *stack_pointer*

(index into *stack_content*) having an accessor function *size* to return the number of elements on the stack; *size* would need to reference *stack_pointer* only and still does belong to the cluster.

Moreover, the metric considers only coupling but not cohesion of the candidate clusters though one would expect that both coupling and cohesion should be taken into account. A minor point of critique is that the term *internal connectivity* is misleading. What the formula for the internal connectivity measures is the fraction *closely-related-subprograms* versus *related-subprograms* with respect to individual variables of the cluster, which is a relationship between the cluster and its environment as opposed to an internal property.

Extensions. The original *Delta-IC* method as proposed by Canfora et al. involves slicing of subprograms that are part of more than one cluster. This, however, means that the system is changed. In the context of Canfora et al.'s work this makes sense since their work is aimed at finding reusable components. Subprograms involved in more than one atomic component fulfill different logical functions for different atomic components. Reusing one atomic component then would imply the need to import other atomic components as well, since the multi-function subprogram shares code with many atomic components.

Our goal is to recover information for program understanding. The system must not be changed during this reverse engineering phase (it may be changed afterwards), i.e., slicing is out of the question. Leaving out slicing may result in overlapping candidates. Merging these overlapping candidates regardless of the degree of overlap is not satisfactory. This approach was taken in an earlier evaluation of the techniques (Girard and Koschke, 1997) in order to get a fair evaluation since the other methods always produce distinct candidates. For the application of the *Delta-IC* method, we can do better: We can merge two candidates when they share a large amount, otherwise they remain distinct and overlapping. In particular, this is the right approach when the user can be consulted. Merging similar candidates frees the user from an overwhelming number of similar candidates: She or he has to judge only critical cases.

The modified *Delta-IC* algorithm 5-4 merges candidates only when they have many elements in common (Step 3). We treat one component *S* as a match of

another component *T* when $S \subseteq_p T$ according to the partial subset relationship introduced in Section 3.5.4.3, which allows for inexact matches. Step 3 merges the overlapping candidates.

Input:

- object reference view V
- ΔIC threshold Θ

Output:

- set of atomic component candidates C

Algorithm:

1. *generate candidates:*
   for each subprogram S **in** V **loop**
   clusters (S) := candidate-cluster (S, V);
   end loop;
2. *filter candidates whose ΔIC is less than* Θ:
   for each subprogram S **in** V **loop**
   if ΔIC (clusters (S)) < Θ **then**
   clusters (S) := ∅;
   end if;
   end loop;
3. *merge overlapping candidates:*
   while ∃ a pair of subprograms {S1, S2} **in** clusters
   where clusters (S1) $\subseteq_p$ clusters (S2) ∨ clusters (S2) $\subseteq_p$ clusters (S1)
   loop
   clusters (S1) := clusters (S1) ∪ clusters (S2);
   clusters (S2) := ∅;
   end loop;
4. *return results*:
   for S **in** clusters'Range **where** | clusters (S) | > 1 **loop**
   C := C ∪ {clusters (S)}
   end loop;

Algorithm 5-4. **Extended Delta-IC analysis.**

Generalization for types. The internal connectivity metric was originally proposed only for abstract data objects. However, we can extend the domain of connectivity to types as well. Before we actually generalize the metric, we state some observations.

There are two different kinds of entities of an abstract data object: variables and constants that we do not want to be accessed from outside of the abstract data object and subprograms that act as public accessor routines. According to these two classes, there are the following different kinds of relationships that we implicitly distinguished above:

1. **non-abstract usage**: an object is directly referenced by a subprogram. There are two categories of non-abstract usage:
   a. the object is non-abstractly used by a subprogram **within** the cluster
   b. the object is non-abstractly used by a subprogram **outside** of the cluster
2. **abstract usage**: an object is not used directly by a subprogram *S* outside of the cluster but by an accessor routine of the atomic component called by *S*, in other words: *S* is accessing the object only by means of the accessor routine associated with the object.

Cases 1.a and 2 conform to the information hiding principle, case 1.b does not. Hence, metrics for cohesion and coupling should penalize 1.b. The metrics for objects in this section are defined with this in mind.

As opposed to objects, we do not want to hide types - they would not be of any use then. Instead, we want to hide the underlying data structure of a type. This corresponds to the idea of the *Internal Access* heuristic. Types should be used abstractly by subprograms outside of the abstract data type. A non-abstract usage of a type is an internal access according to the definition in Section 5.6.

Now that we have a unifying concept *non-abstract usage* for both types and objects, we can generalize the specification of *refer-to* and *referred-by* accordingly. The formulas (5.6) - (5.10) need not be changed. So far, *refer-to(v)* has been defined as *referencing-subprograms(v)* which is defined as *predecessors(v, {reference})* (see Section 3.5.3). Hence, the definition of *refer-to* can be extended as follows in order to include the restricted signature-types relationships (only those

signature types are considered that are annotated by an internal access; see Section 5.6):

$$\begin{aligned}&\text{refer-to(e)} = \\ &\text{predecessors}(e, \{\text{reference}\}) \cup \text{predecessors}(e, \{\text{signature-type}\}, \text{non-abstract})\end{aligned} \quad \textbf{(5.11)}$$

Since *referred-by* is the relational inverse of *refer-to*:

$$\begin{aligned}&\text{referred-by(s)} = \text{refer-to}^{-1}(s) \\ &= \text{successors}(s, \{\text{reference}\}) \cup \text{successors}(s, \{\text{signature-type}\}, \text{non-abstract})\end{aligned} \quad \textbf{(5.12)}$$

Only signature types are considered by the definition above; *local-obj-of-type* edges annotated as non-abstract are ignored for the same reason as for the *Internal Access* heuristic stated in Section 5.6.

By re-definition of *refer-to*, equation (5.10) is now also applicable to types. However, what was found fault for the original definition of *Delta-IC* for abstract data objects, namely, that the definition is aimed at avoiding subclusters consisting of a single object and those subprograms that only access this object, is even more problematic for types. Let us assume information hiding is applied for a component, *C*, consisting of a type *T* and its accessor functions $S_1,\ldots,S_n$ ($n \geq 1$). Then the proposed cluster does not depend upon the chosen subprogram and is always $\{T, S_1,\ldots,S_n\}$. The internal connectivity *IC* for this cluster is exactly 1 since only the accessor functions $S_1,\ldots,S_n$ are referring to the type according to (5.11). However, the subtrahend in the definition of ΔIC in (5.10) is 1, too (let $S \in \{S_1,\ldots,S_n\}$):

$$\sum_{e \in \text{ referred-by (S)}} \frac{|\{F | \text{referred-by (F)} = \{e\}\}|}{|\text{refer-to }(e)|} = \frac{|\{F | \text{referred-by (F)} = \{T\}\}|}{|\text{refer-to }(T)|} = 1$$

Hence, $\Delta IC(S) = 1 - 1 = 0$ for every subprogram *S* referring to *T*. Interestingly enough, ΔIC is also 0 for the following scenario:

- there is a set of subprograms, $\mathcal{S}=\{S_1,\ldots,S_n\}$, and a type, *T*, where $T \in$ *referred-by* $(S_i)\ \forall S_i \in \mathcal{S}$
- only a subset $\mathcal{S}' \subset \mathcal{S}$ refers to *T* only; all other subprograms of $\mathcal{S}$ non-abstractly use other types as well

Then, for any subprogram, $s \in \mathcal{S}'$ (hence, *referred-by* (*s*) = {*T*}) the following holds:

closely-related-subprograms (s) = $\{F \mid \text{referred-by (F)} = \{T\}\} = \mathcal{S}' \subset \mathcal{S}$

and subprograms-related-to (s) = refer-to (T) = $\mathcal{S}$

Hence:

$$\text{IC(S)} = \frac{\sum_{e \in \text{ referred-by (S)}} \left|\{F \mid F \in \text{ refer-to}(e) \wedge \text{referred-by}(F) \subseteq \text{referred-by}(S)\}\right|}{\sum_{e \in \text{ referred-by (S)}} \left|\{F \mid F \in \text{ refer-to}(e)\}\right|}$$

$$= \frac{\left|\{F \mid F \in \text{ refer-to}(T) \wedge \text{referred-by}(F) \subseteq \text{referred-by}(S)\}\right|}{\left|\{F \mid F \in \text{ refer-to}(T)\}\right|}$$

$$= \frac{\left|\{F \mid \text{referred-by (F)} = \{T\}\}\right|}{\left|\text{refer-to } (T)\right|}$$

$$= \sum_{e \in \text{ referred-by}(S)} \frac{\left|F \mid \text{referred-by (F)} = e\right|}{\left|\text{refer-to } (e)\right|} = \Delta IC(S)$$

That is to say, the candidate-cluster for this scenario, in which subprograms outside of the candidate non-abstractly use the type, has the same ΔIC value as a candidate-cluster that represents a pure abstract data type.

Of course, the equation holds also for abstract data objects that contain only one object. However, abstract data types with only one type are much more frequent than abstract data objects with only one object. To sum it up, *Delta-IC* is not really appropriate for atomic components containing only one object or type.

In order to see whether *Delta-IC* is better suited for components with more than one object or type, let us revisit its definition (5.10). The upper bound of ΔIC is 1 and is reached for a cluster that contains at least two types or objects and whose subprograms are referring to more than one type or object in the cluster (hence, the subtrahend of (5.10) is 0) and to no object or type outside of the cluster (*IC* is 1). This makes sense — though it is not clear why highly cohesive clusters with more than one type or object should have a higher ΔIC than highly cohesive clus-

ters with only one object or type. The lower bound of ΔIC depends upon the number of objects in the cluster. Given a subprogram, S, that refers to objects $V_1,\ldots,V_n$ where each object V_i in $V_1,\ldots,V_n$ is accessed by m other subprograms $S_{i,1},\ldots,S_{i,m}$ and *referred-by*$(S_{i,j}) = \{V_i\}$ for $1 \le j \le m$ (see Figure 5-5). Then, $IC(S) = 1$ because of *closely-related-subprograms* (S) = *related-subprograms* (S). However, the subtrahend of ΔIC is as follows:

$$\sum_{e \in \text{ referred-by}(S)} \frac{\left|\{F \mid \text{referred-by (F)} = \{e\}\}\right|}{\left|\text{refer-to } (e)\right|} = n \times \frac{m-1}{m}$$

where $\frac{m-1}{m}$ approaches 1 for large m.

Hence, $\Delta IC(S) \approx 1 - n$ and, therefore, clusters based on $S_{i,j}$ are preferred due to $\Delta IC(S_{i,j}) = 0$ which is what one would expect since S appears like a badly designed initialization function that incorporates initialization code for different components.

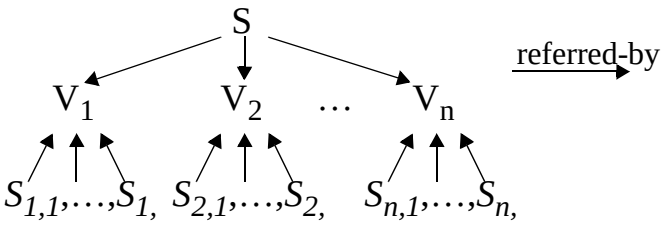


Figure 5-5. **Cluster with low ΔIC.**

Another problematic property of ΔIC is that coupling and cohesion are unbalanced: While the value of IC, which approximates coupling, can only be between 0 and 1, the subtrahend of ΔIC, which approximates lack of cohesion, can be between 0 and n (where n is the number of objects and types in a cluster). It would be useful to adjust the balance between coupling and cohesion with respect to specific system characteristics.

The extensions to *Delta-IC* presented in this section are adaptations of the approach to a pure recovery process that does not allow for changes in the system. The ideas of the original authors are preserved. The critique of the metrics used

for *Delta-IC* leads to a more substantial change of the approach that will be described in the next section.

5.8 Internal and External Connectivity

Name	Internal-External-Connectivity
Reference	Rainer Koschke, unpublished
Domain	Object Reference View and Signature View (where edges are annotated with internal access information)
Range	ADO, ADT, and HC
Disjoint Clusters	No

The general aim of the definition of ΔIC is to minimize external connectivity and to maximize internal connectivity. However, what is called *internal connectivity* (IC) in the *Delta-IC* approach is rather a measurement between the cluster and its environment and, therefore, no internal property.

Moreover, *IC* depends upon the considered subprogram. If one wants to use the metric to rate an arbitrary cluster (that need not to be a cluster in the sense of *Delta-IC*), one has to select one of the subprograms of this cluster to be able to compute the metric and it is generally not clear which one to choose. The metrics defined in this section will only depend on the cluster as a whole. Furthermore, the new metrics allow balancing coupling and cohesion and do not put clusters with only one object or type at a disadvantage.

Internal and external connectivity as defined in this section are primarily proposed in this thesis for assessing candidates (see Section 8.4 and Section 9.3). However, they can also be used as the underlying metric of the *Delta-IC* algorithm 5-4. Moreover, they can also be used to establish a partition of the union of object reference and signature view (including non-abstract usage information) that minimizes external connectivity and maximizes internal connectivity. In order to establish such a partition, genetic algorithms could be used similar to the approach of Mancoridis et al. (1999). Moreover, if the atomic components are

known, one could also use these metrics to assess the degree of information hiding of the system.

First of all, we make a clear distinction between internal and external connectivity, in other words: between cohesion and coupling. The *internal connectivity* (IntC) is only based upon relationships within the atomic component, the *external connectivity* (ExtC) is a measurement between an atomic component and its environment.

The internal connectivity of a cluster is defined as the degree to which the objects and types of the cluster are referred by subprograms within this cluster (*subprograms* was already defined in Section 3.5.4.1 and *refer-to* is defined by (5.11) on page 133):

$$IntC(C) = \frac{1}{|OT(C)|} \times \sum_{e \in OT(C)} \frac{|\text{refer-to(e)} \cap \text{subprograms}(C)|}{|\text{subprograms}(C)|} \qquad \textbf{(5.13)}$$

where $OT(C) = objects(C) \cup types(C)$ is the set of objects and types in C (see Section 3.5.4.1 for the definition of *objects* and *types*).

The motivation of this formula is that subprograms should refer to many objects and types of the cluster to be part of it; thus, we are aiming at a high cohesion. The $IntC(C)$ value is always in the range of 0 and 1. Good design should aim at a high $IntC(C)$ value.

Analogously, we can define external connectivity as the degree to which objects and types of a cluster are referred by subprograms outside of the cluster (in relation to their total usage):

$$ExtC(C) = \frac{1}{|OT(C)|} \times \sum_{e \in OT(C)} \frac{|\text{refer-to}(e) \,/\, \text{subprograms}(C)|}{|\text{refer-to}(e)|} \qquad \textbf{(5.14)}$$

External connectivity corresponds to coupling, i.e., we are striving for atomic components with low external connectivity. The $ExtC(C)$ value is always in the range of 0 and 1. Good design should aim at $ExtC(C) = 0$.

Altogether, in order to minimize external connectivity and maximize internal connectivity, only clusters for which the degree of connectivity

$$\text{connectivity}(C) = \frac{a \times \text{IntC}(C) - \text{ExtC}(C) + 1}{a + 1} \quad \textbf{(5.15)}$$

is above the threshold should be accepted. Factor *a* is used to balance between internal and external connectivity. For normalization, constant 1 is added to the nominator and the term is divided by *a*+1. Because both *IntC(C)* and -*ExtC(C)*+1 yield values between 0 and 1, *connectivity*(*C*) is always between 0 and 1.

5.9 Schwanke´s Arch Approach

Name	Schwanke's Arch Approach
Reference	Schwanke, 1991; Schwanke and Hanson, 1994
Domain	Base View
Range	RS
Disjoint Clusters	Yes

The techniques described above compare pairs of entities by their direct relationships in order to decide whether they belong to the same atomic component. However, a complementary source of information is the environment of the compared entities as stated by Schwanke (1991):

> "If two procedures use several of the same unit-names, they are likely to be sharing significant design information, and are good candidates for placing in the same module."

And not only what kind of entities (unit-names) they commonly use increases their relatedness but also by which common entities they are used. For example, the implementations of a sine and a cosine function will both have a float parameter and result type, but they also will likely be used in the same context, i.e., have common callers. Schwanke's approach takes this into account.

Schwanke's work is aimed at module detection. Subprograms are clustered into modules based on a similarity metric (clustering algorithm 5-5).

```
place each routine in a group by itself
repeat
    identify the two most similar groups
    combine them
until the existing groups are satisfactory
```

Algorithm 5-5. **Similarity Clustering algorithm.**

Clustering criterion. In each iteration, the most similar groups are combined using the similarity metric described below.

Similarity between subprograms. The group similarity used to combine groups in algorithm 5-5 is based on a similarity between subprograms. Given two subprograms *A* and *B*, the similarity metric used during clustering is defined as follows:

$$Sim(A, B) = \frac{Common(A, B) + k \times Linked(A, B)}{n + Common(A, B) + d \times Distinct(A, B)} \quad \textbf{(5.16)}$$

wherein *Common (A,B)* reflects the common features of *A* and *B* and *Distinct (A,B)* reflects the distinct features. *Linked (A, B)* is 1 if *A* calls *B* or *B* calls *A*, otherwise it is 0. The two parameters $k \geq 0$ and $d \geq 0$ are weights given to *Linked* and *Distinct* in *Sim*. They have to be ascertained by experiments on a sample of the subject system. The parameter $n \geq 0$ is used for normalization purposes; it will be considered 0 in the following.

Features of a subprogram *A* are all non-local names that *A* uses including the names of procedures, macros, typedefs, objects, and even the individual record component names of structured types and objects. Record component names of structured types and objects are treated as if they were unique, i.e., if two record types or objects accidentally have record components of the same name, then these names are considered distinct. But not only what *A* uses is a feature of *A*, the fact that *A* is used by another subprogram (i.e., the other subprogram calls *A*) is also considered a feature of *A*. Technically, a feature binding can be expressed

by a pair (*A*, *B*) where *A* uses the non-local name *B*, or, in other words, *A* has feature *B*. The fact that *A* is called by *B* is expressed by (*A*, B^*). The notation B^* denotes a synthetic name derived from *B*. This represents the difference between a name used in a unit and the name of a place where the unit is used. The distinction is made so that (*A*, *B*) and (*A*, B^*) are distinct pairs.

Common and *Distinct* are computed as weighted sums (*features (A)* denotes the features of *A*):

$$\text{Common}(A, B) = W(\text{features}(A) \cap \text{features}(B))$$

$$\text{Distinct}(A, B) = W(\text{features}(A) / \text{features}(B)) + W(\text{features}(B) / \text{features}(A))$$

$$W(X) = \sum_{x \in X} w_x \qquad \text{where } w_x \text{ is the weight of feature x}$$

It obviously makes a difference whether two subprograms have a rare or frequent feature in common. For example, an error object which is used everywhere in the system is less distinctive than an object that is only used by a small portion of the system. Using the Shannon information content to ascertain the individual feature weights takes this into account (Shannon, 1972). It gives frequent features less weight and vice versa:

$$w_x = -\text{ld}(\text{Probability}(x))$$

where *Probability(x)* is the fraction of all entities that have *x* in common. The hypothesis is that rarely used entities are more significant than frequently used entities.

Group similarity. Based on the similarity between subprograms, the similarity for groups is defined as the maximum similarity between any pair of group members (one from each):

$$GSim(A, B) = max(Sim(a, b) | a \in A \wedge b \in B) \qquad \textbf{(5.17)}$$

Extensions. An extension to this approach was proposed by Schwanke himself in joint work with Hanson in 1994. In the extension, they use a nearest neighbor approach to classify components.

The nearest neighbor approach works as follows. The basic similarity measure between individual entities must be monotonic and matching, as described by Tversky (1977). This makes it reasonable to use the similarity measure to compare one entity, *S*, to each of the other entities, T_i, and rank them by their similarity to S. This ranking identifies which of the T_i are the nearest neighbors of *S*. An entity, S, in an existing software system can then be classified by identifying the existing group *G* that contains a plurality of the near neighbors of *S*. Schwanke and Hanson use an arbitrary cutoff by defining "near neighbors" to be the five nearest neighbors (in the following the number of nearest neighbors considered is denoted by *k*). For tie-breaking, points according to nearness are assigned. Then *S* belongs in the group *G* for which the neighbors of *S* that belong to *G* represent the most points. The point values are arbitrary, but observe the following characteristics (let N_i be the *i*'th nearest neighbor): N_1 is worth *k* points, N_2 is worth *k-1*, and so forth; i.e., N_i is worth *k-i+1* points where *i* is in the range of 1…k.

Example. Consider the following example scenarios:

1. Each near neighbor is in a different group: nearest neighbor wins.
2. Group G_1 has N_1, G_2 has N_2 and one of N_3, N_4, or N_5: any two near neighbors are worth at least as much as the single nearest neighbor.
3. G_1 has N_1 and N_2, G_2 has N_3, N_4, and N_5: the two nearest neighbors beat the next three.

Looking at these kinds of scenarios could help decide what point assignments should be used. The "point" is, the weights are chosen to make ordinal comparisons do the right thing.

For this k-nearest neighbor approach, a group similarity measure is not needed and that is why Schwanke and Hanson's paper do not investigate group similarity measures, yet Schwanke suggested in a personal communication a possible group similarity measure based on the same principle (1998):

$$GSim(A, B) = \frac{p(A \cup B) - p(A) - p(B)}{|A \cup B| \times (|A \cup B| - 1)} \qquad \text{where } p(X) = \sum_{S,\, T \in X} p_S(T) \quad \textbf{(5.18)}$$

where $p_S(T)$ is the number of points assigned to T for being a near neighbor of S. That is, this similarity varies directly with the number of near neighbors of members of B that belong to A, and the number of near neighbors of members of A that belong to B. Dividing by the number of relationships in pairs in ($A \cup B$) prevents large groups from getting larger just because more neighbor relationships are involved. Qualitatively, the similarity is the average reduction in *member loneliness* that merging the groups would produce. (Member loneliness is the extent to which the near neighbors of a member belong to other groups.)

The effect of this group similarity measure on clustering can be exemplified as follows. First, the clustering algorithm would pair up all the {S,T} that were mutually nearest neighbors (10 points/2 = 5). Next it would pair up all the remaining {U,V} for which U was nearest neighbor of V, and V was second-nearest for U. (9 points/2 = 4.5).

Schwanke proposes the following exercise to understand the contribution of the group similarity for clustering: compute p({A,B,C}) for all possible neighbor rankings among A, B, and C, and find the cases where the algorithm would prefer forming a triplet from a pair and a singleton rather than forming a new pair out of two singletons. For example,

{a,b,c} := {a,b} + {c} vs.{d,e} := {d} + {e}

The maximum possible value p({a,b,c}) is (5+4+5+4+5+4) = 27. The minimum possible value of p({a,b}), given the values in the previous line, is (5+5)=10, because the algorithm would have formed that cluster first. So, *Sim* ({a,b},{c}) in this case is (27-10)/6 = 2.84 and this triplet {a,b,c} would not be formed until most units were in pairs. Only pairs worth (5+0)/2, (4+1)/2, (3+2)/2, etc., would be formed later. So, almost any unit whose nearest neighbor is not already in a group would be paired with that neighbor before any triplets were formed.

The problem of learning the appropriate weights is addressed by using a feedforward neural network and backpropagating errors (Schwanke and Hanson, 1994).

The network is designed to mirror the model of similarity judgment according to (5.16).

In order to distinguish the original approach from 1991 from the extension of 1994, the former approach is called the **Arch approach** and the latter the **iArch approach** following the terminology Schwanke used in his papers.

The original Arch approach was extended to detect atomic components by Jean-François Girard, Georg Schied, and me in many ways. The enhancements are so manifold that the extension can be considered a new approach. It will be discussed in Chapter 7.

5.10 Type-based Cohesion

Name	**Type-Based Cohesion**
Reference	Patel, Chu, and Baxter, 1991
Domain	Type Composition View + Specific Usage
Range	RS
Disjoint Clusters	Yes

Patel et al. propose an approach similar to Schwanke's similarity clustering, grouping subprograms that share a large amount of types. The main difference to Schwanke's approach is that Schwanke considers any shared non-local features and not just types. We will discuss further differences after the introduction of Patel et al.'s approach.

Patel et al. take into account every type of an expression occurring in the body of a subprogram (both left hand side as well as right hand side expressions) and all types that are used in declarations of the subprogram (declarations of local objects, parameters, and record components as well as type declarations). Each occurrence of a type or record component in any expression or declaration within function *f* is counted as follows:

- If a reference is made to object (or parameter) *V* and *V* has type *T*, then each reference to *V* increments the counter associated with type *T*.

- Moreover, if *T* is a part-type of some other type *T'*, then the counters of all types and components in the path from *T* to the root of the structured type are increased by one.
- If object *V* is a local object of function *f* with type *T*, then the counter of *T* is increased by one.

Example. The approach is based on the type composition view. It will be exemplified by the C code in Figure 5-6 (the example is a translation of Patel's example into C).

```
typedef struct {float real_part; float imag_part;} Complex;
typedef Complex Position_Vector [100];
void f (float rp, float ip) {
  Position_Vector A;
  Complex X;
  int i;

  X.real_part = rp;
  X.imag_part = ip;
  for (i=0; i<100; i++) {
    A[i].real_part = X.real_part;
    A[i].imag_part = X.imag_part;
  }
}
```

Figure 5-6. **Example C code for *Type-based Cohesion*.**

Function *f* is related to the following types: *int* (local variable *i*), *float* (parameters *rp* and *ip*, components *real_part* and *imag_part*), *Complex* (local variable X), and *Position_Vector* (local variable *A*). Furthermore, it references the components, *real_part* and *imag_part*, of record *Complex*. The relations of the types are as in the type composition view in Figure 5-7.

For example, the counter of *int* is 6 because

- *int* occurs in the declaration for *i*
- *i* occurs in the initializing expression of the loop
- *i* occurs in the termination condition of the loop
- *i* occurs as loop increment

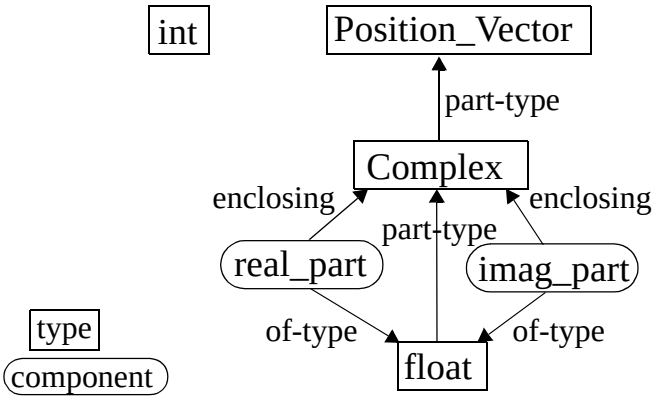


Figure 5-7. **Type composition view for types related to function f in Figure 5-6.**

- *i* occurs twice as index to *A*

The counter for *Complex* is 7 because

- a variable *X* is declared of type *Complex*
- there are two assignments to variable *X*
- there are two uses of variable *X*
- there are two assignments to *A[i]* which is of type *Complex*

The counter for *float* is 10 because

- there are two parameters, *rp* and *ip*, of type *float*
- parameter *rp* is used once
- parameter *ip* is used once
- *X.imag_part* (of type *float*) is used once and set once
- *X.real_part* (of type *float*) is used once and set once
- *A[i].real_part* (of type *float*) is set once
- *A[i].imag_part* (of type *float*) is set once

And finally, the counter of *Complex.real_part* is 3 (likewise for *Complex.imag_part*) because

- *X.real_part* is set once
- *X.real_part* is used once
- *A[i].real_part* is set once

Similarity between subprograms. The relatedness of a function *f* to a sequence *S* of *n* types [T_1, T_2,... T_n] (including record components) can then be represented as an *n*-dimensional vector $R_S(f) = (c_1, c_2, ..., c_n)$ where c_i is the counter associated with T_i computed as described above. Two functions f_1 and f_2 are compared based on $R_S(f_1)$ and $R_S(f_2)$ using the cosine of the angular separation between binary vectors (inspired by work on information retrieval principles; Salton, 1968):

$$Sim(X, Y) = \frac{\sum_{i=1}^{n} x_i \cdot y_i}{\sqrt{\sum_{i=1}^{n} x_i^2} \times \sqrt{\sum_{i=1}^{n} y_i^2}} \quad \textbf{(5.19)}$$

Similarity between groups. The similarity for two groups of subprograms, *A* and *B*, where $A = \{a_1, a_2, ..., a_p\}$ and $B = \{b_1, b_2, ..., b_q\}$ can be defined in terms of the similarity between all subprograms in the union of *A* and *B*. Let $S = A \cup B = \{s_1, s_2, ..., s_n\}$:

$$GSim(S) = \frac{\sum_{N} Sim(s_i, s_j)}{n \cdot (n-1)/2} \quad \text{where } N = \{(i,j) | i, j \in \{1, \ldots, n\} \wedge i > j\} \quad \textbf{(5.20)}$$

Patel et al. propose this as a cohesion metric for subprograms in a module, but it can also be used to cluster similar groups within algorithm 5-5 to find groups of related subprograms.

Differences to Schwanke's approach. Schwanke's and Patel's approaches are similar, but there are substantial differences:

- Patel et al.'s similarity between groups is based on the angular separation of vectors whereas Schwanke proposes to use the maximum of the similarities between two groups or to use the k-nearest neighbor approach.
- The name of an entity counts only once at most in the approach of Schwanke. In the approach of Patel et al., it counts as often as it occurs.

- Direct relations between subprograms (i.e., the two subprograms call each other) are regarded by Schwanke; Patel et al. do not regard *direct* relations between subprograms.
- Schwanke also considers the fact that two subprograms are called by the same subprogram or both call the same subprogram, i.e., *indirect* relations of subprograms other than relations to types.
- All types and names of record components count equally in Patel et al.'s approach; Schwanke weighs them.
- Patel et al.'s similarity is basically a relation between subprograms and types, though this relation may be derived through used objects, whereas Schwanke's similarity is a relation between subprograms and non-local names. This can lead to very different results. An example will make this difference clear. Consider the following C code:

```
T a, b;
f () { a = b; }
g () {T i,j; }
```

  Patel et al.'s metric considers *f* and *g* similar because both *a*, *b*, *i*, and *j* are all of type *T* whereas the similarity between *f* and *g* is 0 by Schwanke's metric.

Patel et al.'s approach can - to some extent - be considered a special case of Schwanke's approach. The major advantage of Schwanke's approach is that similarity is a relation based not only on related types but based on all non-local names. Whether it makes sense to count each occurrence of a type (or record component), as proposed by Patel et al., has to be validated in practice.

5.11 Strongly Connected Components

Name	Strongly Connected Components
Reference	Cimitile, A. and Visaggio, G. (1995)
Domain	Call View
Range	RS
Disjoint Clusters	Yes

Mutually recursive subprograms form an atomic component because none of them can be omitted without loosing a piece of information for the understanding of the other subprogram in the component. Mutually recursive subprograms form a cycle in the call graph which corresponds to the notion of strongly connected components in graph theory. For example, in the call graph in Figure 5-8, we find two strongly connected components {4, 8} and {6, 10, 11}. They can be detected using the linear-time algorithm of Tarjan (1974).

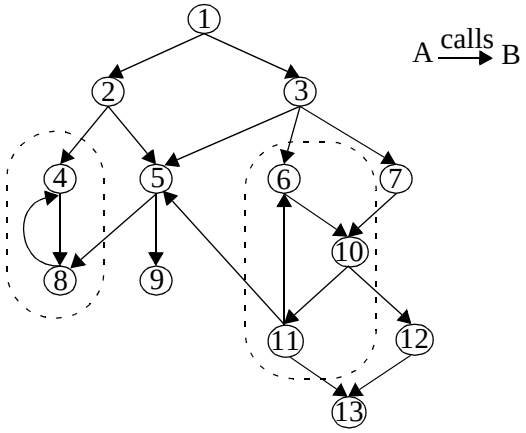


Figure 5-8. **Example of strongly connected components.**

Clustering criterion. All strongly connected components in the call graph form an atomic component.

5.12 Dominance Analysis

Name	Dominance Analysis
Reference	Cimitile, A. and Visaggio, G. (1995)
Domain	Call View, Object Reference View, Type Usage View, and Components View
Range	Yields additional local subprograms/objects.
Disjoint Clusters	Yes

Atomic components often have local subprograms that offer basic services to the accessor routines that constitute the interface of the atomic components. These subprograms do not necessarily mention the type of the ADT in their signature or reference the global objects of the ADO, respectively, and they may remain undetected by all approaches described above. Because these basic service subprograms are local to the atomic components, they are an essential part of them and an atomic component cannot be understood without them. Local in this context means that they are only used by routines in the atomic component. It does not mean that they are local in the sense of nested scopes; quite the opposite: Because they may be used by several other routines in the atomic components, they must be visible to all of them, not to mention that C does not allow nested subprograms. These kinds of local routines can be detected by means of dominance analysis. Locality in the mentioned sense can be viewed as a dominance relation in graph theory. Before we show how, we define the dominance relation:

> A node, *N*, is said to **dominate** another node, *M*, in a directed graph, *G*, if each path from the root of *G* to *M* contains *N*. If *N* is a dominator of *M* and every other dominator *N'* of *M* is also a dominator of *N*, then *N* is called an **immediate** or **direct dominator** of *M*. The dominance relationship can be represented as a **dominance tree** where a node's parent is its immediate dominator.

Cimitile and Visaggio (1995) propose to apply dominance analysis to call graphs to identify candidates for reusable modules. In their approach, cycles (i.e., strongly connected components) are collapsed before dominance analysis is applied. This approach is applied here to detect additional entities local to the components already found. The main differences are that not only cycles are col-

lapsed, but also any atomic component and, furthermore, dominance analysis will here be applied not only to the call view, but to the union of the call, type usage, and object reference view (see Table 3-5 on page 68). This way, objects and types local to components can be detected. The algorithm involves the following basic steps:

1. All members of an atomic component are collapsed to a single node and all edges to and from members are redirected to and from the substituting node (this step is denoted by *Collapse*).
2. Dominance analysis is applied to the collapsed graph resulting from the previous step.
3. In the dominance tree, each component *C* absorbs its (transitively) dominated subprograms that are not dominated by any other component dominated by *C*.

Any (transitive) descendant of node *A* in the dominance tree is local to *A*. That is, we can add all descendants of a component *C* in the dominance tree (i.e., the transitive closure of *C* with respect to dominate edges) to *C*. However, as the example dominance tree in Figure 5-9 illustrates, there may be another component *C'* (transitively) dominated by *C*. Then, the descendants of *C'* in the dominance tree should rather be considered part of *C'* instead of *C* since they are primarily local to *C'*; *C'* can then be considered a part of *C*. For the example in Figure 5-9, S_1 and *C'* are part of *C* whereas S_2 and S_3 are part of *C'*.

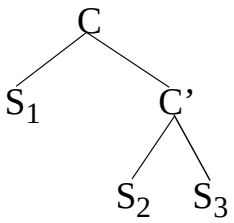


Figure 5-9. **Example dominance tree.**

Clustering criterion. A base entity *N* is considered part of the first component on the path from *N* to the root in the dominance tree. More formally, the entity is added to its primarily dominating atomic component. An atomic component *AC* is said to **primarily dominate** an entity *N* if and only if *AC* dominates *N* and there is no other atomic component dominating *N* that is also dominated by *AC*.

The clustering criterion adds only base entities to components, thus completing existing components. However, the same technique can also be used to detect subsystems by subsuming components under other components according to the dominance relationship. Detecting subsystems by means of dominance analysis was subject of a case study of us (1997a) and is further explored in the thesis of Jean-François Girard.

Algorithm 5-6 adds all local entities to atomic components according to the clustering criterion. The function is started at the root of the dominance tree and recursively traverses the dominance tree in depth-first order.

```
function DFS (Root : Node) return Node_Set is
  Descendants : Node_Set;
begin
  for each D in dominatees (Root) loop
    Descendants := Descendants ∪ DFS (D);
  end loop;
  if Is_Atomic_Component (Root) then
    Add Descendants to Root;
    return ∅;
  else
    return Descendants;
  end if;
end DFS;
```

Algorithm 5-6. **Detecting parts of components in the dominance tree.**

Example. The single steps of the approach can be explained with the example of Figure 5-10. Part (a) shows the input graph that contains two atomic components AC_1 = {4,8} and AC_2 = {6,10,11} detected by previous analyses; they are collapsed in step (1). The result is shown in part (b). The result of applying dominance analysis to part (b) is presented in part (c) of Figure 5-10. Function *DFS* of Figure 5-6 applied to the dominance tree in Figure 5-10(c) adds the two subprograms 12 and 13 and object *Z* to atomic component AC_2 as well as object *X* to atomic component AC_1.

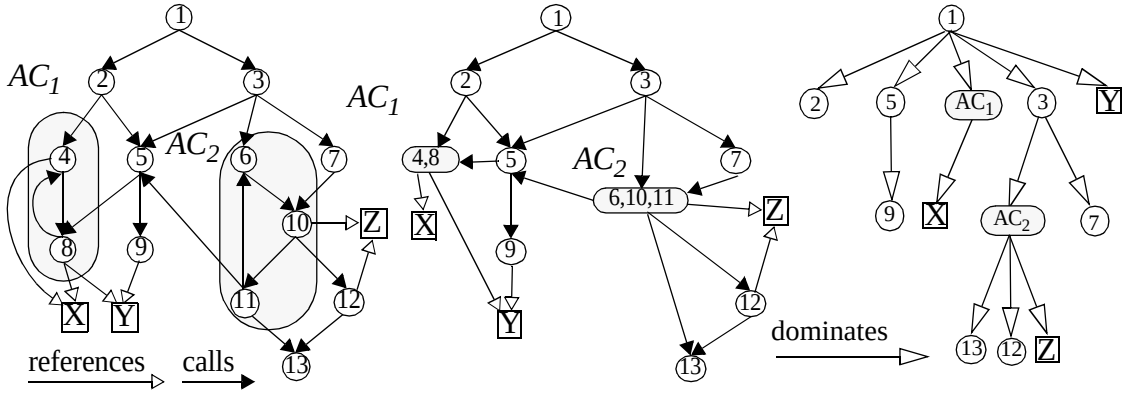


Figure 5-10. **Example of dominance analysis.**

5.13 Preliminary Taxonomy of Basic Structural Techniques

The techniques described in this section are based on structural information only. They neither leverage data flow information nor domain knowledge. They can roughly be classified as follows (this classification will be extended in Section 12.1):

- **Connection-based** approaches cluster entities based on a specific set of direct relationships (and their quality) between entities to be grouped. For example, a routine must have a type in its signature (i.e., the two of them must be directly connected) and the corresponding parameter of this type must be internally accessed by the routine (quality of the relationship) in order to be grouped together. *Global Object Reference*, *Same Module*, *Part Type*, *Internal Access*, and *Same Expression* belong to this category.
- **Metric-based** approaches cluster entities based on a metric using an iterative clustering approach. Schwanke's *Similarity Clustering* and *Type-based Cohesion* fall in this category. The metric-based approaches are based on connections, too, but they differ from connection-based approaches by the degree of freedom that is offered by the metric parameters and the threshold that can be varied to find atomic components with varying confidence.
  Delta-IC is a hybrid within this classification. It is based on a metric but is not really iterative unless one may view it as clustering with only one iteration.

Furthermore, it actually consists not only of the metric but also of a clustering heuristic in the first place: to group closely related subprograms and the referenced objects to a given subprogram. The metric is only used to filter out non-relevant clusters. Nevertheless, I will consider it a metric-based approach because the metric is the predominant factor of this approach and because all connection-based approaches can be viewed as metric-based by expressing their underlying heuristic as a metric (this will be shown in Section 8.4). The opposite direction, i.e., viewing all metric-based approaches as connection-based, does not hold because connection-based approaches always yield the same fixed pattern and there are no parameters to influence clustering.

- **Graph-based** approaches derive clusters from a graph by means of graph-theoretic analyses. The difference to connection-based approaches is that the whole graph has to be considered whereas connection-based approaches regard only direct relationships between entities in order to decide whether they should be grouped. Strongly-*Connected Components Analysis* and *Dominance Analysis* belong to this category.

In this thesis, only the listed connection-based, metric-based, and graph-based techniques are explored. Other techniques are described in Chapter 11.

Chapter 6 Evaluation of the Basic Techniques

In the last chapter, proposed basic techniques for atomic component detection were presented. All but *Same-Expression* have been published and do not originate from us (though we proposed some improvements in the last chapter). However, the original authors rarely described any comparable quantifying evaluation of their techniques. In 1997, we conducted an evaluation of most of these basic techniques (Girard, Koschke, Schied, 1997b) by comparing the atomic components recovered by the approaches presented to those identified by software engineers. The results have already been published (Girard, Koschke, and Schied, 1999; Girard and Koschke, 2000). These results are repeated here. However, this chapter also includes new techniques and systems to achieve a more complete comparison and provides more detail about the method of comparison.

6.1 Reference Corpus

In order to establish a comparison point for the detection quality of the automatic recovery techniques, software engineers manually compiled a list of **reference atomic components** (short *reference components* or *references*) for diverse C systems. These reference atomic components are used for statistical analyses, for calibrating parameters of diverse metrics, and to evaluate the automatic techniques. For the evaluation, we compared the components proposed by automatic techniques, called **candidate atomic components** (short *candidate components* or *candidates*), to the reference components. The sets of references for this com-

parison are called the **reference sets** or **reference corpora**. This section summarizes how the reference sets were obtained and validated.

6.1.1 Systems Studied

The reference components were obtained for several medium size C programs (see Table 6-1 for their characteristics). ***Aero*** is an X-window-based simulator for rigid body systems (Keller, 1995), ***Bash*** is a Unix shell (Ramey, 1994), ***CVS*** is a tool for controlling concurrent software development (Berliner, 1990), and ***Mosaic*** is a world-wide web browser (NCSA, 1997).

Table 6-1. **Suite of analyzed C systems.**

System Name	Version	Lines of Code	# User Types	# Global Objects	# Routines
Aero	1.7	31 Kloc	57	480	488
Bash	1.14.4	38 Kloc	60	487	1002
CVS	1.8	30 Kloc	41	386	575
Mosaic	2.6 (without GUI)	37 Kloc	79	269	564

All figures about program length in terms of *lines of codes* throughout this thesis are ascertained with the Unix tool *wc*, hence include comments and blank lines. Most systems have additional libraries that often encapsulate platform dependencies. These libraries were not investigated. Table 6-1 lists only the size of the core systems that were analyzed by the software engineers.

6.1.2 Obtaining the Reference Atomic Components

The reference components of *Aero*, *Bash*, and *CVS* were compiled by human analysts in 1997. The reference components for Mosaic are the result of the experiment described in Chapter 10. The actual numbers of all major forms of atomic components (abstract data types, abstract data objects, hybrid atomic components) that were identified for each studied system are listed in Table 6-2. The rest of this section gives more detail about how the reference components were established for the respective systems and why they provide a reasonable basis for comparison.

Table 6-2. **Number of atomic components in analyzed systems.**

System	#ADT	#ADO	#Hybrid	#Total
Aero	9	16	1	26
Bash	18	16	5	39
CVS	13	35	6	54
Mosaic	12	28	13	53

6.1.2.1 Reference Components for *Aero*, *Bash*, and *CVS*

We asked five software engineers to identify atomic components in ***Aero***, ***Bash***, and ***CVS***. Table 6-3 summarizes their experience and how the task was divided among them.

Table 6-3. **Human analysts.**

Software Engineer	Programming Experience	System Analyzed
se1	2 years research	***Bash***
se2	2 years research	***Bash***
se3	5 years research	***Bash***
se4	5 years research	***CVS***
se5	> 5 years industry	***Aero***

There was no overlap of their work. They needed between 20 and 35 hours for each system to gather the atomic components of the respective systems. The software engineers were provided with the source code of each system, a summary of connections between global variables, types, and functions, and the guidelines given in Figure 6-1.

Because the analysis of ***Bash*** was distributed among three software engineers, they performed a review of each other's work and came to a consensus on the final reference components.

The guidelines did not exclude overlap between atomic components, i.e., sharing elements among components. There was a small degree of overlap of the reference components of *Aero* and *Bash* and no overlap for *CVS*.

Identify the existing atomic components present in this system. These are abstract data objects (ADO), and abstract data types (ADT), or a combination of both.

- *Here we specified **abstract data types, abstract data objects**, and **hybrid components** by the definitions already presented in Chapter 3.*
- The key difference of ADT and ADO is that an ADT is built around a type and an ADO around a set of simple global variables. This can be decided automatically, so do not waste time writing it down. Just identify the **functions, variables,** and **types which belong together** because they are cohesive and correspond to the idea of **ADO and/or ADT**.
- In practice, programmers sometimes break the encapsulation principle, therefore we widen the definition of abstract data objects and abstract data types to clusters of types or variables, respectively, with their accessor routines. The internal representation of ADTs and ADOs can be public.
- Nota bene: not all functions, variables, or types have to be put into ADO, ADT, and hybrid components.
- In general, your experience and understanding has more value than rules, you are the last judge of what constitutes an ADO/ADT.

Figure 6-1. **Guidelines for human analysts.**

The fact that our reference components used as comparison point were produced by people raises the question whether other software engineers would identify the same atomic components. In order to answer this question, Jean-François Girard performed an experiment on a subset of ***CVS*** containing 2.8 KLOC and composed of the following key files: *history.c*, *lock.c*, *cvs.h*. These source files were distributed along with a cross-reference table indicating the relations among types, global variables, and functions. Four software engineers had the task to identify the atomic components present. He collected a description of the procedure they followed along with their results, then looked for cases where they seemed to have broken their own rules and asked them to refine either their procedure or their results. He also revisited with them those atomic components for

which a comment indicated that they were unsure or something was unclear and corrected their results according to their conclusions.

The four software engineers agreed on the basic principles that characterize an atomic component and proposed very similar components. There were some divergences on the details; for example, one of them added functions to an abstract data type which did not have the type *T* of the abstract data type in their signature, but applied a cast of type *T* to one of their parameters. These divergences occurred rarely.

Jean-François Girard performed a second experiment on a subset of ***Bash*** containing 5.9 KLOC and composed of the following key files: *copy_cmd.c*, *dispose_cmd.c*, *execute_cmd.c*, *make_cmd.c*, *print_cmd.c*, and *command.h*. He followed the same procedure but distributed the subsystem to two software engineers who did not know the system to avoid learning effects.

Finally, in order to assess if these experiment results from a system subset can be generalized to a complete system, one software engineer identified atomic components in the whole ***Bash*** system. The atomic components he identified were compared to those of the reference components used in this thesis (those obtained by consensus).

A quantitative evaluation of the degree of agreement among the software engineers showed first, that the software engineers agreed to a very high degree on the atomic components of these systems and second, that the agreement gained on a smaller subset can indeed be generalized to the rest of the system. Therefore, we may conclude that the reference components for ***Aero***, ***Bash***, and ***CVS*** are a suitable oracle. The procedure used for the quantitative evaluation and its exact results are explained by Girard, Koschke, and Schied (1999).

6.1.2.2 Reference Components for Mosaic

The reference components for Mosaic were obtained by the experiment described in Chapter 10 in which human analysts had to detect atomic components either manually or with tool support. The task of the experimental subjects was to recover as many atomic components for *Mosaic* as possible within 6 hours. In order to obtain comparable results, we reduced the possible search space for

atomic components to a size that could be handled within the given time frame, i.e., all experimental subjects should be able to look at all source files within the available time. Therefore, we excluded the files that are mainly devoted to the graphical user interface, namely, all files whose names begin with the prefix *gui*. The 8 excluded files comprise 15 KLOC, i.e., 40 files consisting of 37 KLOC were to be analyzed. To obtain a common basis of comparison, the atomic components separately detected by each individual were merged and then validated by at least two participants. Only those atomic components were accepted for which a consensus could be reached.

6.2 Comparison of Candidate and Reference Components

Candidate components and reference components are compared using an approximate matching to accommodate the fact that the distribution of functions, global variables, and types into atomic components is sometimes subjective and, pragmatically, we have to cope with matches of candidates and references that are incomplete, yet "good enough" to be useful. "Good enough" means that candidate and reference overlap to a large extent and only few elements are missing. More precisely, we treat one component S as a match of another component T if S is a partial subset of T (denoted by $S \subseteq_p T$) according to the definition of *partial subset* in Section 3.5.4.3. For the results reported in this chapter, $p = 0.7$ is assumed, i.e., at least 70 percent of the elements of S must also be in T. This number is arbitrary, but motivated by the fact that at least three elements of a four-element atomic component must also be in the other atomic component to be an acceptable match.

6.2.1 Classification of Matches

Based on the approximative matching criterion, the generated candidates are classified into three categories according to their usefulness to a software engineer looking for atomic components:

- **Good** when the match between a candidate C and a reference R is close (i.e., $C \subseteq_p R$ and $R \subseteq_p C$). This case is denoted **1~1**.

Matches of this type require a quick verification in order to identify the few elements which should be removed or added to the candidate component.

- **Ok** when the $\subseteq_p$ relationship holds only in one direction for a candidate *C* and a reference *R*:
  - $C \subseteq_p R$, but not $R \subseteq_p C$. The candidate is **too detailed**. This case is denoted as **n~1**.
  - $R \subseteq_p C$, but not $C \subseteq_p R$. The candidate is **too large**. This case is denoted as **1~n**.

  Partial matches of this type require more attention to combine or refine a component. The denotation n~1 and 1~n reflects the fact that multiple Ok matches may exist for a given *R* or *C*.

Altogether, we have three **classes of matches**: 1~1, 1~n, and n~1 where the latter two are both considered Ok.

Example. Consider the example in Figure 6-2. C_1 and R_1 are a good match. Because only partial matches are required, there can be another reference R_4 (with $R_4 \cap R_1 = \emptyset$) that is a partial subset of C_1 (of $C_1 \setminus R_1$, more precisely). C_2 is an Ok match with R_2, and so is C_3. C_2, C_3, and R_2 constitute an n~1 match. That is, the technique has produced finer-grained candidates than what was expected. Note that we cannot necessarily conclude that $C_2 \cup C_3$ and R_2 are a good match because R_2 could be much bigger than $C_2 \cup C_3$. R_3 and C_4 constitute a 1~n match, where no other reference than R_3 can be matched with C_4. C_5 and R_5 do not match at all.

As the example indicates, it is not enough just to count the matches in order to judge the detection quality of a technique. For example, R_3 is a partial subset of C_4 and, therefore, considered at least an Ok match. However, C_4 could be huge and the match just be coincidence. The next section proposes a measurement for detection quality based on multiple aspects that considers this imprecision.

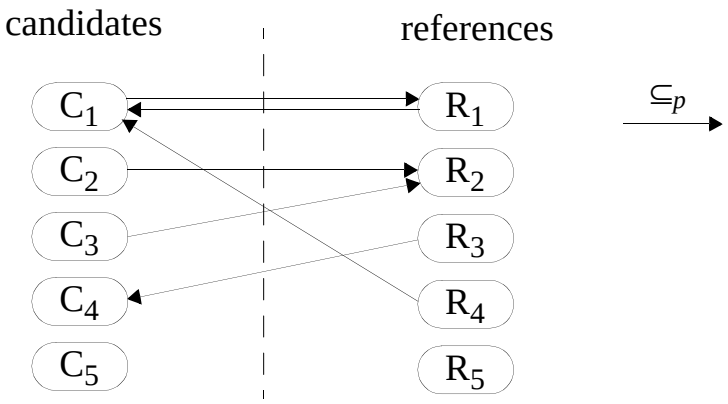


Figure 6-2. **Example correspondences of candidates and references.**

6.2.2 Detection Quality

There are several aspects in a comparison of a set of candidates with a set of references to consider when the matches have been established as described in the last section:

- **Number of false positives**: The number of candidates that neither match a reference nor are matched by any reference, i.e., candidates that cannot be associated with any reference. Technically speaking, these are candidates that are neither involved in a 1~1, 1~n, nor n~1 match. This number should be 0.
- **Number of true negatives**: The number of references that neither match a candidate nor are matched by any candidate, i.e., references that are not even partially detected. Technically speaking, these are references that are neither involved in a 1~1, 1~n, nor n~1 match. This number should be 0.
- **Granularity of matches**: Are the candidates at the right level of granularity? Technically speaking, there should only be good matches and no Ok matches.
- **Precision of matches**: The degree of correspondence between candidates and reference matches. This is discussed in the following in more detail. The precision should approach 1.0.

Since the partial subset relationship is used to establish a match, the matching candidates and references need not be equal. That is, there may be elements of the candidate not in the reference and vice versa: $C \backslash R \neq \emptyset$ and $R \backslash C \neq \emptyset$. In other words, there may be a flaw in a good match; even more so for Ok matches because of (let R be a reference and C_i be candidates for which $C_i \subseteq_p R$ holds):

$$\bigcup_i C_i \subseteq_p R \quad \not\Rightarrow \quad R \subseteq_p \bigcup_i C_i$$

Accuracy for two matching components. In order to indicate the quality of imperfect matches of candidate and reference components, an accuracy factor is associated with each match. The similarity between two components, and thus the accuracy of a candidate vis-a-vis a reference component, is computed using the following formula:

$$\begin{aligned} \text{accuracy}(A, B) &= \text{overlap}(\text{elements}(A), \text{elements}(B)) \\ \text{where } \text{overlap}(X, Y) &= \frac{|X \cap Y|}{|X \cup Y|} \end{aligned} \tag{6.1}$$

In 1~n and n~1 matches — and sometimes even in 1~1 matches — several components may match with one other component. The accuracy as defined above, however, involves only two single components. Therefore, the definition is extended for sets of components as follows.

Accuracy for two sets of components. The overlap for two matching components can be used to ascertain the accuracy of sets of components:

$$\begin{aligned} &\text{accuracy}(\{A_1, \ldots, A_a\}, \{B_1, \ldots, B_b\}) \\ &= \text{overlap}\Big(\bigcup_{i\,=\,1\ldots a} \text{elements}(A_i), \bigcup_{i\,=\,1\ldots b} \text{elements}(B_i)\Big) \end{aligned} \tag{6.2}$$

Accuracy for classes of matches. The accuracy for two sets of components is used to establish the accuracy for a whole class of matches where the two sets $\{A_1,\ldots,A_a\}$ and $\{B_1,\ldots,B_b\}$ are corresponding components in a match within a given class of matches.

More precisely, let the **matching components** of a candidate or reference, X, be defined as follows:

$$\text{matchings}(X) = \{Y \mid Y \subseteq_p X\} \tag{6.3}$$

Using the *matching components*, we can specify the degree of agreement for the diverse classes of matches:

- 1~1 match: accuracy(matchings(C), matchings(R))
- n~1 match: accuracy(matchings(R), {R})
- 1~n match: accuracy({C}, matchings(C))

To put it in words: The accuracy is ascertained based on the united matching components. The way of handling 1~1 matches may first be astonishing, but is motivated by the fact that a 1~1 match does not necessarily mean that there is no other component that is a partial subset of one of the components in the 1~1 match. This was already touched in the example of Figure 6-2 on page 160 where C_1 and R_1 are a 1~1 match and R_4 is still a partial subset of C_1. If there is no such additional 1~n or n~1 match, then:

$$\text{accuracy}(\text{matchings}(C), \text{matchings}(R)) = \text{accuracy}(R, C)$$

That is, we subsume an additional 1~n or n~1 match in a 1~1 match. This is justified because there is a clear 1~1 relationship in the first place and the additional 1~n or n~1 match can only be comparatively small.

Such overlapping matches can also exist for pure 1~n and n~1 matches as the example in Figure 6-3 illustrates. However, the overlap of 1~n and n~1 matches is ignored since there is no dominating correspondence as in the case of overlaps with 1~1 matches. That is, the two overlapping matches in Figure 6-3 are handled as two distinct matches.

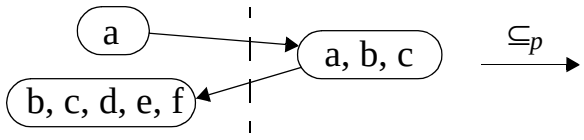


Figure 6-3. **Overlapping 1~n and n~1 matches.**

Now that we have the means to establish the accuracy of a single match with respect to its class (1~1, 1~n, n~1), we can extend the accuracy to the whole class of matches. The classes of matches are defined as follows:

$$GOOD = 1 \sim 1 = \{(\text{matchings}(c), \text{matchings}(r)) \mid c \subseteq_p r \wedge r \subseteq_p c\}$$

$$1 \sim n = \{(\{c\}, \text{matchings}(c)) \mid \text{matchings}(c) \neq \emptyset \wedge \forall (r) r \subseteq_p c \Rightarrow \neg c \subseteq_p r\}$$

$$n \sim 1 = \{(\text{matchings}(r), \{r\}) \mid \text{matchings}(r) \neq \emptyset \wedge \forall (c) c \subseteq_p r \Rightarrow \neg r \subseteq_p c\}$$

$$OK = 1 \sim n \cup n \sim 1$$

Then, the accuracy for a whole class of matches is defined as the average in accuracy of the members of the class (let M be a class of 1~1, 1~n, or n~1 matches):

$$\text{accuracy}(M) = \frac{\sum_{(a, b) \in M} \text{accuracy}(a, b)}{|M|} \quad \textbf{(6.4)}$$

Overall recall rate. In the following, the detection quality of a technique is described by a vector of the number of false positives and true negatives and the average accuracies of 1~1, 1~n, and n~1 matches according to (6.4) along with their respective absolute number to indicate the level of granularity. These figures provide a detailed picture for the comparison of the techniques. However, an additional summarizing value is useful for a quick comparison. The following equation characterizes the overall recall rate (*GOOD* and *OK* are defined above):

$$\text{Recall} = \frac{\sum_{(a, b) \in GOOD} \text{accuracy}(a, b) + \sum_{(a, b) \in OK} \text{accuracy}(a, b)}{|GOOD| + |OK| + |\text{true negatives}|} \quad \textbf{(6.5)}$$

To illustrate the definition of the recall rate, consider the example in Figure 6-4, in which the matching components of each candidate and reference component of Figure 6-2 have been merged for the comparison. There are two *OK* matches and one *good* match. R_5 is not matched at all and, therefore, considered a true negative; likewise, C_5 is a false positive because it does not correspond to any reference. The example also illustrates that the denominator of (6.5) cannot simply be the number of the original references because not only candidates but also references can be united for the comparison, which reduces the number of references actually used for the comparison.

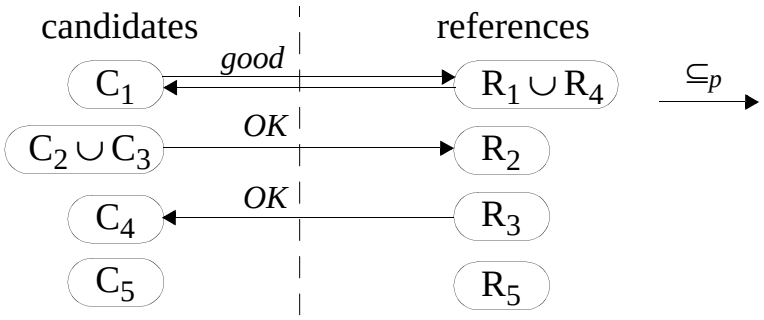


Figure 6-4. **Example merged correspondences of candidates and references.**

The recall rate (6.5) abstracts from the level of granularity — since *good* and *OK* matches are treated equally by this definition — and ignores false positives. The number of false positive is a different aspect and is not captured by this definition because — depending on the task at hand — a higher number of false positives in favor of a higher recall rate may be acceptable.

6.3 Benchmark Results for the Basic Techniques

We applied the techniques listed in Table 6-4 to the reference corpus described in Section 6.1. Note that some of the techniques are only designed for detecting one type of atomic component, some are basically able to detect both ADTs and ADOs; therefore, Table 6-4 summarizes once again what kind of atomic components occurring in the reference corpus are detected by the respective technique (a ⊥ means that the technique cannot detect atomic components of this kind, a ✓ says that the type of atomic component can be detected). Results for hybrid components are not directly reported because there were not enough of them in the reference corpus for a valid evaluation (Mosaic would have been the only exception). Instead, since hybrid components can be viewed as extended ADTs or ADOs, hybrid components are part of the reference set to which the candidates are compared. A technique suitable for ADT or ADO detection can at least partially detect a hybrid component. Hence, the reference set for a technique suitable to detect ADTs contains all reference ADTs and hybrid components, while the reference set for a technique detecting ADOs consists of abstract data objects and hybrid components of the reference corpus. Note that the original *Delta-IC* method was proposed as an iterative approach involving human validation and

Table 6-4. **Evaluated combinations of atomic component detection techniques.**

Method	ADT	ADO
Global Object Reference	⊥	✓
Same Module	✓	✓
Internal Access	✓	✓
Part Type	✓	⊥
Same Expression	⊥	✓
Delta-IC[a]	⊥	✓
Arch (Schwanke)	✓	✓
Type-based Cohesion	(✓)	⊥

a. Previously published evaluations of *Delta IC* (Girard, Koschke, and Schied, 1997b, 1997c, 1999; Girard, Koschke, 2000) are based on a definition of *IC* that differs from the original one; in the evaluation presented in this chapter, the original definition is used.

Table 6-5. **Detected ADTs and ADOs.**

Method	System	ADT						ADO					
		Good		OK				Good		OK			
				too large		too detailed				too large		too detailed	
		#	acc.	#	acc	#	acc	#	acc.	#	acc	#	acc
Global Object Refer-ence	Aero	-	---	-	---	-	---	3	0.91	0	0.00	0	0.00
	Bash	-	---	-	---	-	---	5	0.92	0	0.00	0	0.00
	CVS	-	---	-	---	-	---	8	0.92	1	0.61	6	0.41
	Mosaic	-	---	-	---	-	---	12	0.89	5	0.49	4	0.29
Same Module	Aero	3	0.81	0	0	1	0.34	7	0.88	3	0.10	1	0.27
	Bash	4	0.81	0	0.00	0	0.00	5	0.88	4	0.33	1	0.64
	CVS	6	0.80	1	0.23	3	0.29	25	0.90	4	0.68	10	0.51
	Mosaic	5	0.94	0	0.00	6	0.33	13	0.90	5	0.46	4	0.30
Internal Access	Aero	2	0.87	0	0.00	1	0.60	0	0.00	4	0.28	1	0.11
	Bash	10	0.97	3	0.44	1	0.59	3	0.89	1	0.26	1	0.43
	CVS	4	0.93	1	0.26	4	0.35	6	0.87	0	0.00	15	0.35
	Mosaic	5	0.90	3	0.37	6	0.42	2	0.88	1	0.67	11	0.48

Table 6-5. **Detected ADTs and ADOs.**

Method	System	ADT						ADO					
		Good		OK				Good		OK			
				too large		too detailed				too large		too detailed	
Part Type	Aero	2	0.90	0	0.00	1	0.60	-	---	-	---	-	---
	Bash	9	0.93	1	0.56	0	0.00	-	---	-	---	-	---
	CVS	4	0.95	4	0.37	5	0.26	-	---	-	---	-	---
	Mosaic	6	0.95	2	0.68	5	0.33	-	---	-	---	-	---
Same Expres-sion	Aero	-	---	-	---	-	---	2	0.94	1	0.19	0	0.00
	Bash	-	---	-	---	-	---	4	0.88	0	0.00	2	0.50
	CVS	-	---	-	---	-	---	5	0.79	2	0.18	13	0.50
	Mosaic	-	---	-	---	-	---	1	0.73	0	0.00	5	0.50
Delta-IC	Aero	-	---	-	---	-	---	4	0.81	2	0.25	0	0.00
	Bash	-	---	-	---	-	---	4	0.82	1	0.47	2	0.26
	CVS	-	---	-	---	-	---	9	0.85	0	0.00	15	0.45
	Mosaic	-	---	-	---	-	---	18	0.86	5	0.52	10	0.44
Arch	Aero	1	0.80	0	0.00	5	0.28	3	0.79	6	0.28	1	0.47
	Bash	0	0.00	4	0.39	6	0.41	4	0.70	5	0.28	1	0.21
	CVS	2	0.86	5	0.46	3	0.37	4	0.76	10	0.46	18	0.48
	Mosaic	6	0.79	3	0.49	4	0.36	7	0.72	4	0.47	13	0.47
Type Based Cohesion	Aero	0	0.00	1	0.20	4	0.32	-	---	-	---	-	---
	Bash	2	0.69[a]	0	0.00	7	0.32	-	---	-	---	-	---
	CVS	0	0.00	1	0.14	5	0.34	-	---	-	---	-	---
	Mosaic	0	0.00	0	0.00	8	0.41	-	---	-	---	-	---

a. Note that the accuracy of good matches can also be below the threshold of the partial subset relationship. For example, if *R* and *C* both have 10 elements and 7 elements of *R* are in *C* and 7 elements of *C* are in *R*, then $R \subseteq_{0.7} C \wedge C \subseteq_{0.7} R$ holds and, hence, *R* and *C* are a good match. However, the overlap of *R* and *C* is only $7/13 = 0.54 < p = 0.7$.

slicing of functions. Since we are comparing the technique to non-interactive techniques, we leave out the validation and slicing steps for this evaluation.

Note that *Type-based Cohesion* can only find groups of related routines; it groups neither types nor variables. However, the job of the software engineers was only to find ADTs, ADOs, and hybrid components and, therefore, the reference corpus does not contain sets of related subprograms. Nevertheless, because *Type-based Cohesion* is based on the types the subprograms share, one can attempt to find at

least the accessor functions of an ADT with it. On the other hand, in the evaluation, one should be aware that the type itself is not in the candidate — which decreases the recall rate slightly — and groups of related subprograms are not in the reference corpus — which may increase the number of false positives.

The techniques often produce candidates with less than three elements. Since components of this size are very rare (Aero has one, Bash has none, CVS has five, and Mosaic has three reference components with only two elements), candidates with less than three elements are filtered out. This way, less false positives were produced. The numbers given in the following were established after the filter for small components has been applied. Furthermore, since the largest reference components for the subject systems have less than 50 elements, another filter was applied that ignores candidates with more than 75 elements; i.e., the size of the candidates was not allowed to exceed 50% of the largest reference component. This restriction mainly affected *Global Object Reference* and *Part Type* (in particular, 1~n matches), which both tend to produce very large candidates. Candidates of that size would require too much effort for validation and are therefore of little help to a maintainer.

The detailed information on the number of detected components is shown in Table 6-5, and the number of false positives and true negatives are listed in Table 6-6.

Figure 6-5 and Figure 6-6 summarize the recall rates for ADT and ADO detection, respectively, according to equation (6.5). To be fair, we must also mention that the reference components were used to calibrate *Delta-IC* and *Type-based Cohesion* that both require parameter adjustment. In practice, one does not have the reference components in advance and has to estimate parameters based on a manually extracted list of reference components of a representative sample of the system. That is, in practice the results can be worse for *Delta-IC* and *Type-based Cohesion*.

ADT recall. According to these summaries, the effectiveness of a technique strongly depends upon the system. In the case of the ADTs of Aero, all techniques are similarly effective; *Same Module* is only slightly ahead. Moreover, *Type-based Cohesion* is among the techniques with the least recall rate for all sys-

Table 6-6. **Number of false positives and true negatives.**

Technique		Aero		Bash		CVS		Mosaic	
		false positives	true negatives	false positives	true negatives	false positives	true negatives	false positives	true negatives
Global Reference	ADT	---	---	---	---	---	---	---	---
	ADO	7	14	17	16	4	27	3	13
Same Module	ADT	4	6	8	19	4	8	2	14
	ADO	17	5	24	8	8	1	1	14
Internal Access	ADT	2	7	5	7	9	11	3	10
	ADO	12	10	12	16	5	20	1	27
Same Expres-sion	ADT	---	---	---	---	---	---	---	---
	ADO	9	13	11	15	9	22	1	35
Part Type	ADT	3	7	11	13	8	5	2	11
	ADO	---	---	---	---	---	---	---	---
Delta-IC	ADT	---	---	---	---	---	---	---	---
	ADO	18	11	18	14	10	19	3	9
Arch	ADT	13	4	29	13	36	9	11	10
	ADO	38	6	46	11	42	17	11	15
Type-based Cohesion	ADT	24	5	12	14	17	13	18	17
	ADO	---	---	---	---	---	---	---	---

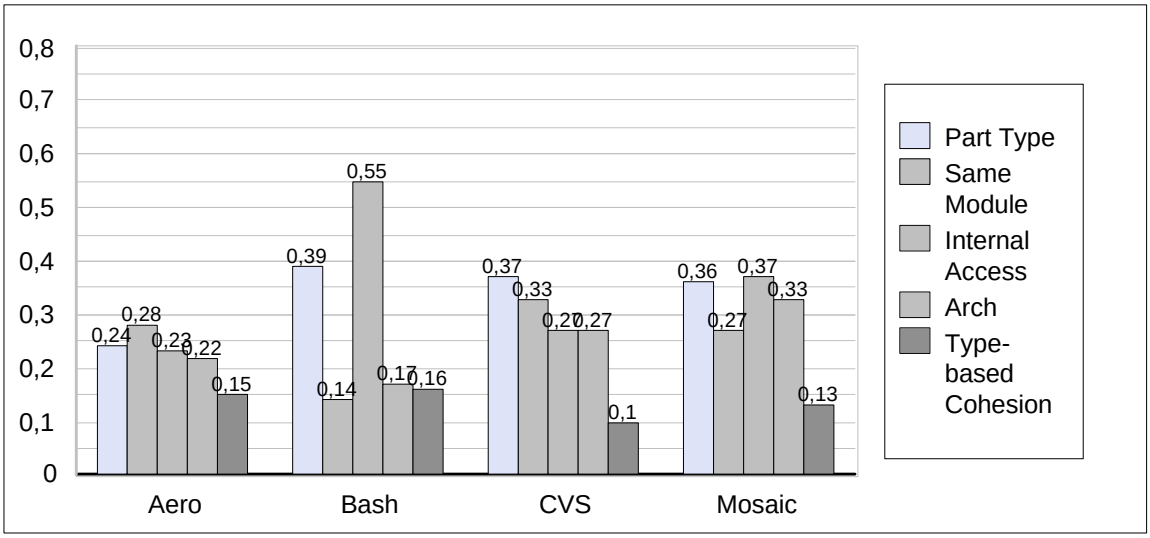


Figure 6-5. **Recall rate for ADTs.**

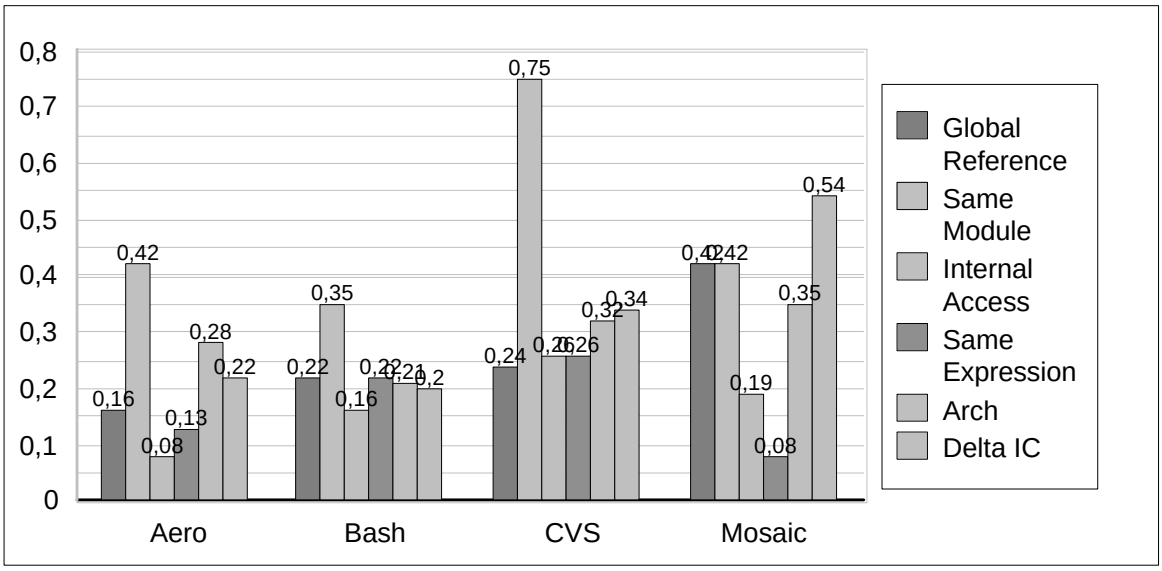


Figure 6-6. **Recall rates for ADOs.**

tems and has also many false positives. In the case of Bash, *Internal Access* is far better than all other techniques. *Internal Access* is also among the best techniques for the other systems. Likewise, *Part Type* has a constantly high recall rate for all systems while *Same Module* fails for Bash. *Arch* is one of the less effective techniques and has most false positives.

ADO recall. *Same Module* identifies more abstract data objects than any other approach (except for Mosaic where *Delta-IC* finds most ADOs). *Arch* has also one of the higher recall rates for all systems but also most false positives. *Global Object Reference* is among the best techniques only for Mosaic. *Internal Access* for ADOs is far less effective than it is for ADTs. The recall rates of *Same Expression* are comparatively low, but *Same Expression* has the fewest false positives (except for CVS where it is on average). *Delta-IC* has comparatively high recall rates for all systems — in the case of Mosaic, it is even clearly the best. The number of false positives of *Delta-IC* is average.

The overall result of this evaluation is that the recall rate of all automatic techniques does not compare to the human recall rate, neither for ADTs nor for ADOs. The best recall rate of *Same Module* for *CVS* is a rare exception. In most cases, the best recall rates are between 20 and 40 percent.

Furthermore, the number of false positives and true negatives is high for many automatic techniques. False positives deserve special attention and, hence, we devote Section 6.4 to their analysis.

6.4 Analysis of False Positives

The techniques proposed some atomic components for which no corresponding reference components existed and, therefore, were classified as false positives. We investigated these false positives to learn more about the weaknesses of the techniques. It turned out that a few false positives are indeed correct positives; many of these were too small to be considered by the software engineers as atomic components, others were simply overlooked by the analysts.

The analysis of false positives was only done for Aero, Bash, and CVS and for the techniques *Same Module*, *Part Type*, *Internal Access*, *Global Object Reference*, and *Delta-IC*. The false positives of *Type-Based Cohesion* and *Arch* were not investigated since the primary goal was to see whether these apparent false positives are really false positives or simply overlooked and to find common patterns of these false positives rather than to detect the specific weaknesses of each technique.

6.4.1 Average Size of False Positives Before Manual Validation

The average size of false positives is an important factor to consider when evaluating approaches, because the time a user of the approach needs to discard a false positive is related to the size of the candidate. For this reason, Table 6-7 reports also the average size of false positives identified by the various techniques on each system (measured in terms of the number of functions, variables, and types of the false positive candidates).

6.4.2 Overlooked Atomic Components

Jean-François Girard, Hiltrud Betz who was one of the original analysts, and I browsed the lists of false positives for the techniques and classified each candidate either as *overlooked positive* when it actually could be regarded as reason-

Table 6-7. **Average size of false positives before filtering overlooked references**

	Technique	Aero	Bash	CVS
ADT	Same Module	3.0	7.3	8.5
	Part Type	5.7	6.2	14.5
	Internal Access	3.0	4,6	8.9
ADO	Same Module	15.0	9.9	6.8
	Global Object Reference	20.0	5.9	5.5
	Internal Access	11.1	4.2	7.0
	Delta-IC	14.7	6.6	7.2
	Same Expression	18.8	8.1	13.2

able atomic component or else as *real false positive*. Figure 6-7 shows the number of real false positives and overlooked positives. The figure reveals that 42% of the ADO candidates and 41% of the ADT candidates originally classified as false positives are indeed overlooked positives.

The presence of such overlooked positives is understandable since browsing a 30 KLOC program manually is a tedious process and even larger components can easily be overlooked. This is interesting because it stresses the importance of automatic support and it shows that an automatic technique does not have to be perfect in order to be useful. Moreover, some of the false positives could be justified from a different point of view (see “Different views” in the following section), i.e., the automatic techniques may provide the maintainer with another perspective.

6.4.3 Common Patterns of False Positives

The analysis of false positives revealed certain common patterns that could be used to filter out false positives in a post analysis after the candidates were proposed by the chosen technique. These patterns were generally found in the set of false positives of any of the examined techniques.

Static local variables. Some global variables are only referenced by one routine; thus, they act as static local variables of this routine but the programmer did not take advantage of the ability in C to express this explicitly. A routine with such

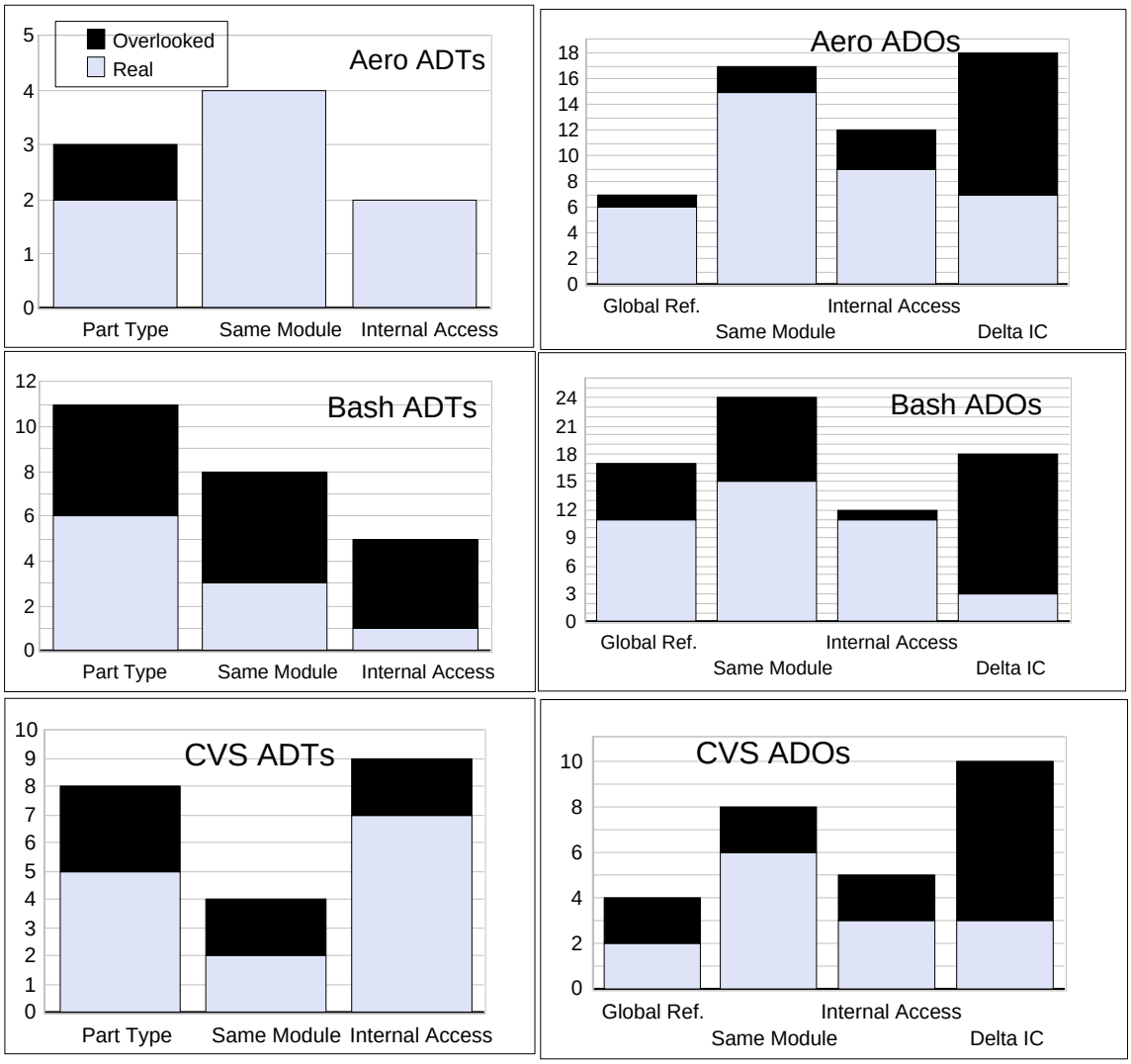


Figure 6-7. **Reviewed number of false positives**

static local variables alone can hardly be considered an ADO in a narrower sense — even though the local variables indeed clearly belong to the subprogram.

Nested routines. Some candidates consist of a few routines among which only one is called from outside and all other routines are only called from routines within the candidate; thus, the latter are local to the routine. Hence, if the variables in the candidate are also only accessed by routines of this candidate, the candidate is rather one routine with nested subroutines (which cannot be expressed in *C*) and some static local variables. Usually, an atomic component has several interface routines and we would therefore not consider this kind of candi-

date a valid atomic component. Such clusters do provide useful information and should be presented to the user — but not as an abstract data object.

Parameter passing. A more complex pattern that we also found in all systems consists of variables used for parameter passing in the presence of call backs. A call back is a call of a function *F* in a component *A* where *F*'s address has been transmitted from another component *B* to *A* as a function pointer. *Aero*, as an X-window-based application, uses this schema in its user interface code. *CVS* has a general recursion handler that traverses directories and has a call back for each file it finds during traversal. The client of this recursion handler does not have to care about the traversal; he only has to code the function that should be applied to each file and to convey it to the recursion handler as a call back function pointer. These functions, designated by function pointers, have to have the same signature. To convey additional parameters from *B* to the call back function, a global variable is used. The variable is set before the call back is done and the call back function then reads the global variable. Figure 6-8 illustrates this kind of parameter passing in the presence of call backs. Variables only used for parameter passing are conceptually different from variables of an abstract data object that model state. Both kinds of variables are needed by the abstract data object, yet — probably due to the conceptual difference — the software engineers did not group "parameter variables" with the abstract data object. It would be helpful to a maintainer to provide additional analyses that characterize the variables of a candidate component as "parameter variables", state variables, or other kinds of usages of a variable. However, in order to detect variables used for parameter passing, control and data flow analyses are needed.

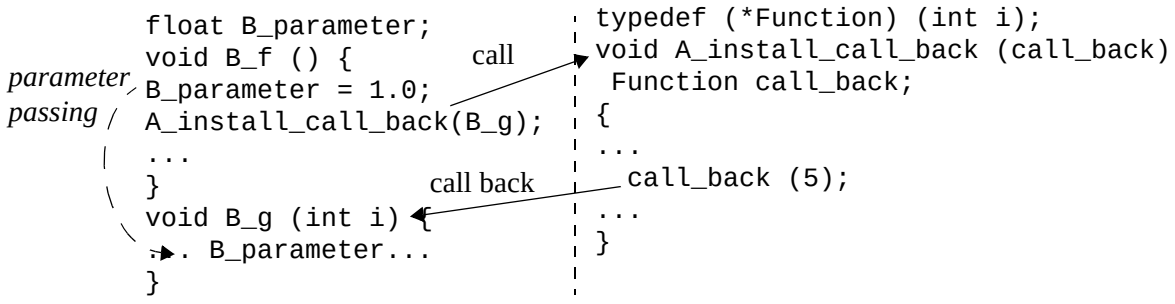


Figure 6-8. **Additional parameter passing in the presence of call backs.**

System parameters. Variables used at many places in the system often represent global system parameters, e.g., variables that indicate whether a certain command line switch was set when the program was invoked. Often, it is recommended to exclude frequently used variables (Yeh et al., 1995). However, simply excluding frequently used variables may also affect variables of an abstract data object that the programmer made public. A more reliable method is to exclude variables that are directly data dependent on the parameter *argv* of the main routine that contains the command line arguments of the invoked program in batch-oriented systems. Other frequently used variables are so-called *mode variables* that indicate the general state of the system as a whole as opposed to the state of an individual abstract data object. A mode variable, for example, may indicate that an error occurred and the system is in recovery mode. It is still not clear how to distinguish these from frequently used public variables of an abstract data object.

Function pointer and enumeration types. Furthermore, the analysis of false positives revealed that function pointer types and enumeration types are generally not helpful for the detection of abstract data types. Function pointer types are often just declared for the purpose of call backs and enumeration types are either used for control variables or are part of more complex abstract data types. Nevertheless, since they are represented in the resource usage graph, they are clustered as any other user-defined type. That is to say, if there is an enumeration or function pointer type, *T*, all subprograms for which *T* is the only user-defined type in the signature may be grouped together by a technique based on signatures (dependent upon the underlying restrictions of the technique) even if the subprograms are otherwise unrelated. In particular, *Part Type* will propose such candidates because the part-type filter for signatures with only one user-defined type cannot have any effect. But also *Internal Access* (if the function pointer parameter is dereferenced or a standard operator is applied to the enumeration parameter, which is legal in C because enumeration values are actually integer values) as well as *Same Module* (if the type is declared in the same module as the subprograms) and the metric-based approaches *Type-based Cohesion* and Schwanke's *Arch* approach (since the subprograms share at least an enumeration or function pointer type) may group these subprograms together.

Different views. Sometimes there were different possible views and one was chosen by the software engineer and the alternative view was chosen by the tech-

nique. Yet, both views could be justified. For example, in Bash, there is a file *print_cmd.c* that provides print commands for different data types. The software engineer decided to group these print routines with the data types, taking an object-oriented view. However, these print routines share global variables that define the current indentation and the increment of indentation. Some techniques grouped the print commands with the global variables taking a more functional view. Interestingly enough, the original programmer of Bash obviously had the functional view in mind, too, since he grouped these routines with the global variables in the same file.

6.5 Qualitative Comparison

After the quantitative comparison described in the last sections, we analyzed divergences between candidate and reference components for the respective techniques.

Global Object Reference. If programmers followed the information hiding principle, *Global Object Reference* would detect all abstract data objects without any false positive. However, this is only the case for a few ADOs. When there is a subprogram that accesses variables of different ADOs, *Global Object References* unites the elements of these ADOs to one single candidate. This could be observed for many ADOs of all systems. Because very large candidates were filtered, *Global Object Reference* achieved a better recall only for Mosaic. Apparently, Mosaic has a better decomposition; probably because of the shorter maintenance history it has, which is also indicated by the good performance of *Same Module* for Mosaic.

Part Type. As opposed to *Same Module*, *Part Type* does not rely on the programmer's distribution of routines into modules. However, it assumes that the parameter of a part type is actually used to be put into its container or to be retrieved from it. Since it does not analyze the actual usage any further, it is going to fail if this assumption is false. Moreover, in most signatures, there is no part type and, therefore, the *Part Type* heuristic equals *Global Object References* for ADOs with its problem of erroneously large candidates.

Same Module. The postulate of *Same Module* is that the programmer structures files according to atomic components. If a programmer puts each routine in a separate file, *Same Module* cannot yield any result. Moreover, for modules with several distinct abstract data types containing conversion routines between each other, this heuristic groups all those routines and data types together in one large component.

Detection of abstract data types with this heuristic did not work well for Bash. Bash has a header file with system-wide type declarations. The routines, however, are implemented in several other C files that include the type declarations. Detection of abstract data objects succeeded better since global variables are never declared in header files (they can only be declared there as external). Moreover, the programmers of the subject systems often take advantage of the means of the programming language C for information hiding of global variables: These variables are often declared static. The limited means for information hiding of ADTs, on the other hand, are rarely used.

Large files can be a problem for *Same Module*. In CVS, for example, we found a type RCS node which encapsulates dependencies on the underlying revision control system (RCS) of CVS. This node type is declared in one huge file where many routines use it as a parameter. Consequently, *Same Module* created a very large atomic component candidate. The group of software engineers has refined this candidate into different aspects of the RCS subsystem.

Same Expression. Evaluation of the false positives revealed that there is in fact a strong semantic relationship among variables that occur in the same expression in most cases. Sometimes these false positives were simply overlooked by the software engineers or were too small to be selected. However, there were a few interesting exceptions. If there are global state or mode variables that occur in the same condition, very large candidates can be proposed. For example, the global variable *interactive_shell* in Bash indicates whether the shell is interactive; only then, certain services are available. In the source code conditions like

```
if ((interactive_shell == 0)&&(def_buffered_input == fd))…
```

are frequent. Often used mode variables lead to a union of otherwise separate ADO candidates of *Same Expression*.

Internal Access. *Internal Access* groups user-defined data types and global variables with the routines that access their internal parts.

For ADT detection, *Internal Access* really checks how the parameter type is used, as opposed to *Part Type.* However, in real programs one often finds the encapsulation principle violated. This frequently happens for reasons of efficiency or convenience in the case of data types of which the programmer is convinced that their representation will never change. If there are many violations of the information hiding principle, *Internal Access* yields very large candidates analogously to *Global Object Reference* or *Part Type*.

This heuristic did badly in ADO detection for at least two reasons. First, if there is a global table, such as an array of error messages, all readers of this table are considered operators of this abstract data object which yields an erroneously large component. Second, it misses all accessor routines that only set or use the variables of the ADO as a whole. In order to group a subprogram with a variable by *Internal Access*, the subprogram must use the variable non-abstractly, i.e., either internally access the variable or apply a standard operator to it. If the variable is of a primitive type and the subprogram only sets or uses the variable, however, the subprogram will not be grouped with the variable. Unfortunately, it is a common phenomenon for abstract data objects to have separate global variables that together form an object. For example, stacks are often declared as two distinct variables, one for the contents and one for the stack pointer. The latter is declared as integer and used as an index to the array implementation for the contents. Then a function returning the size of the stack by returning the value of the stack pointer will not be recognized as part of the abstract data objects, since there is no non-abstract usage. If programmers put the distinct variables together in a record type, the connection among the variable would be obvious and each access to the variables as record components would be a non-abstract usage. However, because abstract data objects have only one instance, programmers do not make the effort.

Delta-IC. A problem of *Delta-IC* is to establish the right threshold. It depends on the specificity of the system studied and cannot necessarily be directly reused. For the systems in this evaluation, these values were as listed by Table 6-8. One practical solution is to take a sample of the system and perform manual recovery by applying the approach on the sample and adjusting the threshold accordingly.

It is necessary to compromise between the effort required to analyze a sample manually and the quality of the results.

Table 6-8. **Selected thresholds of *Delta-IC*.**

System	Threshold
Aero	0.25
Bash	0.38
CVS	0.25
Mosaic	- 0.3

Moreover, the definition of $\Delta_y IC$ is aimed at filtering out candidates that have many accessor functions that only access one single variable of the ADO. As a consequence, any ADO that actually has only one single variable cannot be detected. Furthermore, we often find accessor routines that do access only one single variable. For example, in the stack consisting of two variables for the contents and the stack pointer, a function *size* returning the number of elements on the stack needs only access one variable, the stack pointer. As a matter of fact, the recall rate of *Delta-IC* increases and the number of false positives decreases for Aero, Bash, and CVS when filtering is only based on the internal connectivity as defined by (5.9) on page 126 as opposed to $\Delta_y IC$ as proposed by (5.10) as Table 6-9 shows. In the case of Mosaic, the detection quality is more or less the same.

Table 6-9. **Results of *Delta IC* based on IC only.**

System	Recall	False Positives	Threshold
Aero	0.31	11	0.55
Bash	0.30	21	0.72
CVS	0.39	8	0.53
Mosaic	0.51	3	-0.1

Arch. One problem of the *Arch* approach is to find the right parameters. This is true for all metric-based approaches. However, *Arch* has more parameters than *Delta-IC* and *Type-based Cohesion*. Schwanke and Hanson propose to use neural networks to learn the appropriate parameters (1994), whereas the parameters for *Arch* as evaluated in this thesis were calibrated on the reference components by systematic hand-tuning. In both cases, a set of references is needed to calibrate

the approach. In practice, one does not have these references in advance and, hence, has to compile the components for a sample of the system and calibrate the parameters on this sample. Whether the parameters established on the sample are also appropriate for the rest of the system has to be shown. Furthermore, it is not clear in advance how big the sample should be to find a suitable parameter setting.

The *Arch* metric was originally defined to group related subprograms, i.e., only subprograms had to be compared. However, in our application, the entities to be grouped are heterogeneous since we group also types and objects. Since the metric does not make a distinction among different types of entities, the weight of an entity depends upon its frequency as a neighbor only. There is no way to assign certain types of entities more weight. For example, if we are searching for abstract data objects, variables are the crystallization points, whereas types are secondary. Likewise, because we are searching for specific kinds of atomic components, primarily for abstract data types and abstract data objects, certain relationships are more important than others. *Arch* does not make a distinction whether the neighbor is a type used to declare a local variable or a type in a signature, though it is intuitively clear that *signature* relationship is more important than the *local-obj-of-type* relationship when searching for abstract data types (which will actually be confirmed by a statistical analysis in Section 7.6.1). Another drawback of the approach is that it is not directly recognizable for the maintainer why two entities are in the same candidate since different aspects are considered, namely, common, distinct, and direct relations, which makes validation more difficult, whereas all other techniques have only one simple criterion. On the other hand, considering different aspects is also an advantage. The connection-based approaches, for example, rely on direct relations only and cannot detect groups of related subprograms that may only be detectable when called by the same subprograms.

Type-based Cohesion. *Type-based Cohesion* is very similar to *Arch* (see the comparison in Section 5.10). Its advantage over *Arch* is that it has only one parameter: the threshold that determines whether two subprograms are similar enough to be in the same component. On the other hand, it can only group subprograms based on the portion of types they share. The types themselves, however, are not clustered. Hence, if abstract data types are to be detected, these types

must be added by the maintainer once *Type-based Cohesion* has proposed its candidates. Moreover, *Type-based Cohesion* considers also intrinsic types of the programming language, like *float* and *int*. This may be an additional clue, however, it is rather questionable whether two subprograms are actually related if they share only intrinsic type. Furthermore, the metric of *Type-based Cohesion* does not make a distinction between user-defined types and intrinsic types, nor what the type is used for. Hence, a function, F_1, having a local integer variable has the same similarity to a function, F_2, having a local integer variable and a signature type, *T*, as a function, F_3, that has signature type *T* as well but no integer variable. One would expect that F_3 and F_2 are more similar than F_1 and F_2.

6.6 Distinctive Contribution of Individual Techniques

Before deciding to select one approach or a combination of approaches, one has to consider the additional information provided by the other techniques. As a first estimate of the contribution of each approach, we considered the good candidates from each technique. The first section of Table 6-10 (grey zone in the table) contains the *distinctive contribution* of each technique as the number of reference components identified by only one approach (good match).

Table 6-10. **Number of reference components identified.**

	Aero	**Bash**	**CVS**	**Mosaic**
Global Object Reference		3	1	
Same Module	4	3	15	
Internal Access		2	1	
Part Type		2	1	
Same Expression				
Delta-IC	2	1	1	5
Arch				5
Type-Based Cohesion				
Found by more than one approach	7	13	16	20
Found by no approach	13	15	19	23
Identified by software engineers	26	39	54	53

These data suggest that by combining approaches instead of using a single approach, one would significantly improve the discovery of the reference components. Yet, between 35 and 50 percent of the components could not be matched by a good candidate. (However, they may be matched partially; Table 6-10 contains only good matches.)

6.7 Analysis of True Negatives

In Section 6.5, only the weaknesses of individual techniques have been discussed, which gives insight into why a single technique did not detect certain components. However, as the last section showed, there are even many references that were not detected as a 1~1 match by any of the techniques. To go further into the question why reference components were true negatives for all techniques, I analyzed the references of Aero, Bash, and CVS not detected as a 1~1 match with a candidate component of all techniques. The inspection revealed several reasons:

- **Two-element reference components**: In a few cases, the reference components contained only two elements. Because a filter was used to ignore candidates with less than three elements, these reference components could not be detected by a 1~1 match. If the smallest possible candidate matching a two-element reference contains three elements, the overlap of the candidate and the reference cannot be higher than 2/3. However, 2/3 is below the threshold 0.7 used as tolerance parameter for the partial subset relationship. Hence, such a match is classified as a 1~n match.
  The filter was effective in reducing the number of false positives and is therefore recommended. Reference components with less than three elements are rare. Aero has one, Bash has two, and CVS has five overlooked references with only two elements.
- **Interleaving**: As a consequence of interleaved functions — i.e., functions containing multiple, interwoven strands of computation, each responsible for accomplishing a distinct goal (Rugaber et al. 1995) — clusters are merged that would have been otherwise proposed as separate candidates. An interleaved function, for example, may access variables of two different abstract data objects or internally access two different abstract data types. As a consequence, the interleaved function, *F*, introduces a link between the two clusters built

around the global variables and the composite data structure related to *F*, resulting in a merge of the clusters. On the other hand, considering semantic arguments (that are not captured by purely structure-oriented techniques), the engineer has recognized the distinction among the two concepts and the interleaving of *F* and identified two separate components. Merging two intrinsically different candidates may result in two references detected by a 1~n match instead of two 1~1 matches depending upon the size of the candidates.

- **Lack of abstraction**: A phenomenon, similar to interleaving, leading to 1~n matches is *lack of abstraction*. All techniques assume at least a minimum of abstraction. If all components are permissive, accessed from many different places, and all constituents of components are arbitrarily assigned to modules, large clusters arise and the chances to detect components as a 1~1 match by the proposed techniques are small. They may even not be detected as a 1~n match if the size of the clusters is above the upper threshold for acceptable candidates. Note that interleaving is not the same as lack of abstraction. In the case of lack of abstraction, a function, *F*, may access internal elements of a component, *C*, though *F* does not belong to *C*. On the other hand, in the case of interleaving, the function, *F*, accesses internal elements of more than one component and *F* actually belongs to these components. For example, in the case of two related abstract data types *matrix* and *vector*, the multiplication of a matrix with a vector may be implemented by accessing the internal data structures of both types due to efficiency considerations, and *F* actually belongs to both of these types.
- **Layering**: Interestingly enough, the use of information hiding was also a reason why some references were only partially detected. Some components are structured as layers and only the lowest layer accesses the variables or record components of the atomic component, whereas upper layers deal with user interface issues or implement services on top of the lowest layer. The engineers have sometimes grouped all layers together because there was no finer-grained structuring possible. Hence, *Internal Access* and all other techniques that are based on direct variable accesses can only identify the lowest layer. If these layers are additionally implemented by different modules, which is a reasonable decomposition for large components because each layers constitutes a different kind of service, *Same Module* is neither be able to detect the other layers. *Part Type* may still be able to detect at least abstract data types among layered component (but not abstract data objects). However, because *Part Type* tends to produce very large clusters that are filtered out if they exceed the acceptable

size for candidates, the layered component may happen to be among the filtered clusters. Then, the corresponding reference component is not detected at all.

- **Debatable references**: There are also a few debatable components among the references. Some are incomplete, i.e., there are base entities that could be added to the component; some contain spurious elements, e.g., a function that accesses many variables of another component and that may rather be considered a part of that component. Sometimes it was difficult to make a clear cut between complex clusters of interwoven base entities and a different way of decomposing the cluster could be justified as well.

6.8 Summary

The comparison of automatic techniques with respect to findings of software engineers described in this chapter revealed the following points:

- The effectiveness of a technique depends upon system characteristics, like degree of information hiding, proper module decomposition, and layering.
- None of the investigated techniques has a sufficient recall rate; The best recall rate we obtained was 75% of the abstract data objects (in CVS, as detected by *Same Module*). In the worst case, namely, the abstract data types of Aero, the best technique reached only a recall rate of 28%.
- Many candidates the techniques provide correspond only roughly to the reference components; i.e., elements of these atomic components were superfluous or lacking.
- Combining the automatic approaches instead of using a single approach, one would significantly improve the discovery of the reference components.
- Yet, between 35 and 50 percent of the components still could not be completely and directly found by any of the techniques. However, the components may at least partially be matched.
- In evaluating these automatic techniques, one also has to state what we observed by reviewing the false positives: It turned out that 42% of the ADO candidates and 41% of the ADT candidates classified as false positives could indeed be considered correct positives; they were either too small to be consid-

ered, simply overlooked in the manual process, or represented alternative views.

- We found common patterns of false positives in all systems that could be used to filter out false positives from the set of candidates.
- Moreover, whereas the groups of software engineers needed about 20 - 35 hours to compile the list of atomic components for each of our subject systems, each atomic component produced by the techniques can be checked by software engineers within minutes. To browse the whole list of false positives of all automatic techniques, we needed less than 6 hours per system. The time needed for validation can even be reduced by merging similar candidates of different techniques based on the partial subset relationship because there were many similar false positives among the candidates.

In order to find more components with fewer false positives, we chose the most flexible technique, namely, Schwanke's *Arch* approach, and tried to improve it. The next chapter describes the approach and its evaluation. Furthermore, due to the degree of vagueness of reasonable decompositions and the complex semantic issues involved, the user should be integrated into atomic component detection. For this reason, the second part of this thesis is devoted to effective ways of user integration.

Chapter 7 *Similarity Clustering*

Name	Similarity Clustering
Reference	Girard, Koschke, Schied, 1997
Domain	Base View
Range	RS, ADT, ADO, HC
Disjoint Clusters	Yes

The overall result of the evaluation of the basic techniques described in the last chapter is that none of the techniques reaches human detection quality. In order to achieve better results, we chose the most flexible approach, namely, Schwanke's *Arch* approach (1991), and improved it. This chapter describes the approach and its evaluation.

Schwanke's work is aimed at detecting subsystems using a similarity metric between routines (see Section 5.9). The similarity clustering approach described in this chapter applies the idea of Schwanke's work to atomic component identification by generalizing the similarity metric, adding informal information, edge-dependent weights, and adapting many of its parameters. The two approaches will be contrasted in more detail in Section 7.8.

The enhancements of this technique were joint work with Jean-François Girard and Georg Schied. My improvements to the technique after it has been jointly published in 1997 are explicitly listed in Section 7.8.

Cluster analyses are used in many areas and scientific disciplines in which large amounts of data need to be reduced to few units of meaning that are easy to grasp. Cluster analyses have already emerged in the beginning of the seventies. Steinhausen and Langer, for example, have summarized existing cluster analyses and techniques as early as 1977. The general approach of clustering is therefore well-understood. The challenge in applying cluster analysis to a particular problem is to define an appropriate similarity metric.

7.1 Overview of the Approach

The similarity clustering approach, short *Similarity Clustering*, groups base entities (subprograms, user-defined types, and global variables) according to the proportion of features (entities they access, their name, the file where they are defined, etc.) they have in common. The intuition is that if these features reflect the correct direct and indirect relationships between these entities, then entities which have the most similar relationships should belong to the same atomic component.

Functions, variables, and types are grouped according to the algorithm already outlined in Section 5.9 and repeated in Figure 7-1 for ease of reading. In each iteration of this algorithm, a similarity metric measures the proportion of shared features. The algorithm terminates when “existing groups are satisfactory”; i.e., when the similarity of the most similar groups is below a certain user-determined threshold.

```
place each entity in a group by itself
repeat
    identify the two most similar groups
    combine them
until the existing groups are satisfactory
```

Algorithm 7-1. **Flat similarity clustering algorithm.**

The final result of the clustering algorithm outlined in Figure 7-1 are “flat” sets of similar entities, but the information about the similarity among the set elements is

lost. However, this information is of great interest to the maintainer. Instead of presenting only the final clusters to the maintainer, one should keep a log of the order of combination as an additional information. Since the two most similar groups are combined per iteration, the order of combination can be represented by a binary tree in which the leaves are the initial groups and in which the inner nodes are combinations of groups. The farther a combination is away from the root of the tree, the higher is its degree of similarity. This procedure is called **hierarchical clustering** (Steinhausen and Langer, 1977). Figure 7-2 outlines a hierarchical clustering algorithm.

place each entity in a group by itself
repeat
  identify the most similar groups S_i and S_j
  combine S_i and S_j
  add a subtree with children S_i and S_j to the clustering tree
until the existing groups are satisfactory or only one group is left

Algorithm 7-2. **Hierarchical similarity clustering algorithm.**

If this tree is presented to the maintainer, one can also repeat hierarchical similarity clustering until everything is combined into one single group instead of stopping when a certain minimal similarity is reached. The maintainer can then "climb up the tree" starting at the leaves and stop at inner nodes for which the combination is doubtful.

The similarity metric used in this algorithm to identify the two most similar groups is constructed of three layers:

- The similarity between **two groups of entities** which is defined in terms of similarity between entities across groups.
- The similarity between **two entities** which is a weighted sum of various aspects of similarity.
- Each specific **aspect of similarity** between two entities.

These layers will be discussed in the following sections.

7.2 Similarity Between Groups of Entities

There are several alternatives to define similarity between two groups of entities. The one originally proposed by Schwanke in 1991 is to use the **maximal individual similarity** of elements in the two groups:

$$GSim(A, B) = max(Sim(a, b)) \qquad \forall a \in A, b \in B \tag{7.1}$$

We experienced that this has the effect of creating very large groups which is not very helpful. The same observation was made in other applications of clustering (Steinhausen and Langer, 1977). If we had to group circular structures, i.e., when the cluster is a cycle of pairwise similar elements whose similarity to other elements is otherwise low, basing similarity on the maxima of individual elements would be the right choice. However, this is generally not the case for atomic components; we expect all their elements to be related to each other. To achieve this goal, we decided to define the group similarity as the average of the similarities of all pairs of entities in the two groups (1997):

$$GSim(A, B) = \frac{\sum_{(a \in A, b \in B)} Sim(a, b)}{|A| \times |B|} \tag{7.2}$$

Using the **average group similarity** for groups demands of the elements of a group to be considerably similar to many other elements of the group and, hence, aims at cohesive components. As described in Section 5.9, a newer variant of Schwanke's *Similarity Clustering* uses a k-nearest neighbor approach (Schwanke and Hanson, 1998). The motivation for the k-nearest neighbor approach according to Schwanke and Hanson is that the various factors affecting the weights of features and other terms in the similarity measure prevent it from being validly used to compute ratios or even averages. On the other hand, the nearest neighbor approach may lead to components in which not all parts are strongly related to each other.

7.3 Similarity Between Entities

The similarity between two entities is the weighted sum of various **aspects of similarity** which fall into the following categories:

- direct relations
- indirect relations
- informal information

Direct relations are relations between the two entities compared. Indirect relations are relations with common third entities. Informal information is the information in the program source code which is not captured by the semantics of programming languages, but is used by programmers to communicate the intent of a program (e.g., comments, identifier names, file organization, etc.). Informal information can be used as a complementary source of information as suggested by Biggerstaff (1999).

The various aspects of similarity can be united as follows (the factors x_i are used to adjust the influence of the diverse specific similarities):

$$\begin{aligned} & Sim(A, B) \\ = \; & x_1 \cdot Sim_{indirect}(A, B) + x_2 \cdot Sim_{direct}(A, B) + x_3 \cdot Sim_{informal}(A, B) \end{aligned} \tag{7.3}$$

7.3.1 Features

The following individual aspects of similarity, namely, $Sim_{indirect}$ and Sim_{direct}, are going to be defined in terms of features. A single feature describes a specific relationship of an entity to another entity in its environment. A **feature** has basically three facets:

- the *partner* in this relationship
- the *modality* of this relationship
- the *role* of the entity in this relationship

For example, for a subprogram, it is of interest what global variables it accesses and by what other subprograms it is used (partner). Furthermore, it is relevant whether the accessed variables are set or used and whether the subprogram is

used by directly calling it or just by taking its address (modality). And last but not least, it makes a difference whether a subprogram is the caller or callee in a call relationship (role). The role information is, of course, only relevant to non-symmetric relationships.

We will use the term *feature* as a tuple of these three facets. In terms of the resource usage graph, a **feature** is thus a triple (*n, e, r*) where *n* is a node, *e* is an edge type, and *r* is the role. The node *n* is the partner in a relationship, *e* expresses the modality, and *r* is either *agent* when the entity is the agent in a relationship (technically, the source of the edge), *patient* when it is the patient in this relationship (the target of the edge), or simply *partner* when the relationship is symmetric.

Notation. For a feature (*n,e,r*) of a node *m*, we will use the notation $m \rightarrow_e n$ if *m* is an agent, $m \leftarrow_e n$ if *m* is a patient, and $m \leftrightarrow_e n$ if the relation is symmetric. The predicate $(m \rightarrow_e n)?$ is true if and only if *n* is related to *m* by relationship *e* in which *m* is an agent. Similar predicates are used for the two other kinds of features. In some equations, we will use a place holder for relations: $m \propto_e n$ means that there is a relationship *e* between *m* and *n* where the role of *m* is unspecified. When this place holder is used in a functor or predicate such as $\bigcup_{\propto}$, then the place holder $\propto$ implicitly iterates over $\{\rightarrow,\leftarrow,\leftrightarrow\}$. For example:

$$n(a,b) = \bigcup_{\propto} \{a \propto b | (a \propto b)?\}$$

is equivalent to

$$n(a,b) = \{a \rightarrow b | (a \rightarrow b)?\} \cup \{a \leftarrow b | (a \leftarrow b)?\} \cup \{a \leftrightarrow b | (a \leftrightarrow b)?\}$$

Example. If subprogram *A* calls subprogram *B*, then:

- $A \rightarrow_{\text{call}} B$ is a feature of *A*
- $B \leftarrow_{\text{call}} A$ is a feature of *B*

Moreover, $(A \rightarrow_{\text{call}} B)?$ is true, whereas $(B \rightarrow_{\text{call}} A)?$ is false if B does not call A.

7.3.2 Indirect Relations

Two entities can be considered similar if they use the same entities and if they are used by the same entities, even more so when they use them or are used by them in the same way. Though each individual common relationship to their environment is probably not sufficient to call them similar, the confidence of being similar increases by each one.

Judging two entities by their relationship to the environment is a phenotypic kind of comparison. A genotypic point of view would rather compare the degree of complexity, the number of statements, and so forth. However, these "inner" values are generally of very limited use for the decision whether entities should be grouped together. There is no point in putting subprograms of the same complexity or with the same number of statements in a common module.

Taking only common features into account may distort the result. Two subprograms, for example, may be called by the same subprogram and may set the same variable; but when they are called by many other distinct subprograms and set many distinct variables, we would not consider them similar anymore. Therefore, distinct features must also be borne in mind.

The definition of similarity with respect to indirect relations captures the proportion of features (as introduced in Section 7.3.1) two entities share.

$$Sim_{indirect}(A, B) = \frac{W(Common(A, B))}{W(Common(A, B)) + d \cdot W(Distinct(A, B))} \quad \textbf{(7.4)}$$

where *Common (A,B)* reflects common features of A and B, *Distinct (A,B)* reflects distinct features and $d \geq 0$ is a parameter which regulates the importance given to distinct features, and $W(X)$ is the weighted sum as described below.

Common and *Distinct* can roughly be described as follows (we will later refine this definition):

$$Common(A, B) = features(A) \cap features(B) \quad \textbf{(7.5)}$$

$$Distinct(A, B) = features(A) \oplus features(B) \quad \textbf{(7.6)}$$

The term *features(X)* refers to the set of features of *X* (see Section 7.3.1). The operator⊕denotes the symmetric difference for sets. *W(X*) is the weighted sum of these features:

$$W(X) = \sum_{x \in X} weight(x) \quad \textbf{(7.7)}$$

where $weight(x) \geq 0$ is a weighting factor which allows assigning certain features more influence on the global value of the metric. Weights will be discussed in Section 7.3.3.

The common and distinct neighbors are relevant to the similarity of two entities for obvious reasons. The other facets of features offer additional ways to specify and use *Common* and *Distinct*. These alternatives will be discussed in the following.

Modality. There are several alternative ways of distinguishing the modality of a usage – technically speaking, of considering the edge type. We discuss them by means of the scenarios in Figure 7-1.

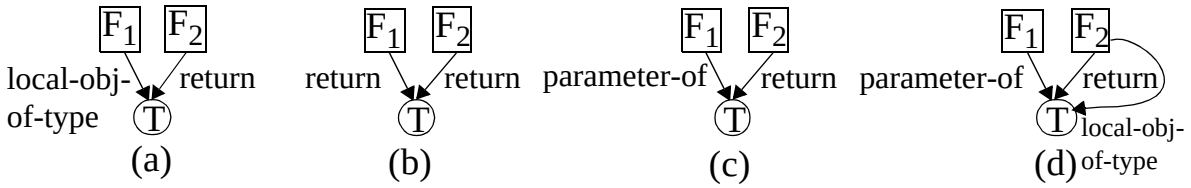


Figure 7-1. **Scenarios for typical relations.**

In Figure 7-1(a), for example, there are two functions related to a type *T*: One by having a local variable of type *T* and one by having a return type *T*. The modality of using *T* is very different for the two functions. It is plausible that the two functions would be more similar when their usage of the type would have the same modality as in Figure 7-1(b). That is the point of view reflected by the first alter-

native (it is even stricter since it does not consider the functions in Figure 7-1(a) to be similar at all):

> *Alternative 1: All nodes that are reachable by the same kind of edge are common features.*

The first alternative obviously goes wrong for scenario Figure 7-1(c) where one function returns a type and the other has a parameter of this type. These two functions could be accessor routines of an abstract data type *T*. The distinguishing factor of this example is that the two edge types are two special cases of the same abstract kind of edge. The second alternative refines alternative 1 by taking subtyping of edges into account (see Section 3.5.1 for the definition of equivalent edges):

> *Alternative 2: All nodes that are reachable by equivalent edges are common features.*

Alternative 2 would not consider the two functions in Figure 7-1(a) similar at all. However, the fact that the two of them share at least type *T*, even if they use *T* differently, is an important information. It can also be a complementary piece of information as in Figure 7-1(d) where we have an additional relationship to *T* by means of a local variable. Frequently, constructors of abstract data types are implemented by using a local variable of this type that is initialized and eventually returned. We should therefore not ignore any kind of edge that leads to a common neighbor. This results in the third alternative:

> *Alternative 3: All nodes that are reachable by any kind of edge are common features.*

Yet, we do like to make a difference between scenarios Figure 7-1(a) and (b). One attempt could be to assign different weights to different edge types. For example, *return* edges have a higher weight than *local-obj-of-type* and therefore a subprogram pair as in Figure 7-1(b) would be preferred to Figure 7-1(a). Still, that does not work because it is rather questionable why two subprograms having both a local variable of type *T* should be more similar than two functions having both return type *T*. The solution is to separate common features into such regarded by alternative 2 and such covered by alternative 3.

Alternative 4: Make a distinction between nodes that are reached via equivalent and non-equivalent edges.

A more formal description of alternative 4 will follow below as soon as we have discussed the influence of roles.

Role. Not only the modality of a relationship to a common neighbor is important; we also have to distinguish whether the roles are identical or not. For example, it makes a difference whether two subprograms both call the same subprogram, *S*, or whether one subprogram calls *S* and the other one is called by *S*.

For these reasons, I propose the following strategy that also takes alternative 4 from above into consideration. We will distinguish two cases:

- $\boldsymbol{Common_{eq}(A,B)}$ is the set of **equivalent features** of *A* and *B*, i.e., the common neighbors of *A* and *B* reachable by equivalent edges and with the same role.
- $\boldsymbol{Common_{ne}(A,B)}$ is the set of **non-equivalent features** of *A* and *B*, i.e., the common neighbors of *A* and *B* that are reachable by non-equivalent edges only or with different roles.

In the case of common features, there are nodes reachable by both entities either via different or same edge types. However, in the case of distinct features, a node is either reached from one node or from the other one such that we need not distinguish features as in $Common_{eq}$ and $Common_{ne}$. Therefore, *Distinct (A,B)* denotes all features of *A* and *B* that are not shared (as originally proposed by equation (7.6)), and thus, equation (7.4) can be refined to:

$$Sim_{indirect}(A, B) = \frac{I_{eq} \times W(Common_{eq}(A, B)) + W(Common_{ne}(A, B))}{I_{eq} \times W(Common_{eq}(A, B)) + W(Common_{ne}(A, B)) + d \cdot W(Distinct(A, B))} \quad \textbf{(7.8)}$$

Parameter I_{eq} in this equation is used to determine the influence of equivalent features. Its value should be greater than one because the same modality of a rela-

tionship to a common neighbor is more significant than just a common neighbor that is related in different ways.

So far, we have used *Common* and *Distinct* in a vague way. Now that we have seen what role they play and how they are used, they can be defined in detail:

$$\text{Common}_{eq}(A, B) = \bigcup_{\propto} \{A \propto_{e_1} C\ ,\ B \propto_{e_2} C \mid (A \propto_{e_1} C)? \wedge (B \propto_{e_2} C)? \wedge e_1 \sim e_2 \wedge C \neq A, B\} \quad \textbf{(7.9)}$$

where $e_1 \sim e_2$ holds when the two edges are equivalent as defined in Section 3.5.

$$\text{Common}_{ne}(A, B) = \bigcup_{\propto} \{X \propto_e C \mid (X \propto_e C)? \wedge X \in \{A, B\} \wedge C \in N(A, B) / \{A, B\}\} - \text{Common}_{eq}(A, B) \quad \textbf{(7.10)}$$

$$\text{where } N(A, B) = \text{Neighbors}(A) \cap \text{Neighbors}(B)$$

Neighbors yields all neighbors of a node as defined in Section 3.5.

$$\text{Distinct}(A, B) = \bigcup_{\propto} \{X \propto_e C \mid (X \propto_e C)? \wedge X \in \{A, B\} \wedge C \in D(A, B) / \{A, B\}\} \quad \textbf{(7.11)}$$

$$\text{where } D(A, B) = \text{Neighbors}(A) \oplus \text{Neighbors}(B)$$

Note that we explicitly exclude direct relations among two entities to compare. Such direct relations will be considered in Section 7.3.4.

Example. Given the resource usage graph in Figure 7-2, the following common and distinct features of F_1 and F_2 can be found (using the abbreviations *actual* = *actual-parameter-of*, *param* = *parameter-of*, *set* = *obj-set*, and *use* = *obj-use*):

$$\text{Common}_{eq}(F_1, F_2) = F_1 \rightarrow_{\text{call}} F_5, F_2 \rightarrow_{\text{call}} F_5, F_1 \rightarrow_{\text{param}} T, F_2 \rightarrow_{\text{return}} T$$
$$F_1 \rightarrow_{\text{set}} V_2, F_2 \rightarrow_{\text{use}} V_2$$

$$\text{Common}_{ne}(F_1, F_2) = F_1 \leftarrow_{\text{actual}} V_1, F_2 \rightarrow_{\text{set}} V_1, F_1 \leftarrow_{\text{call}} F_4, F_2 \rightarrow_{\text{call}} F_4$$

$$\text{Distinct}(F_1, F_2) = F_1 \leftarrow_{\text{call}} F_3, F_2 \rightarrow_{\text{call}} F_6$$

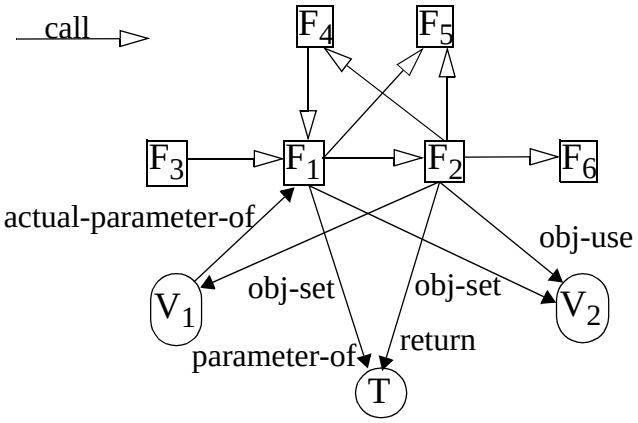


Figure 7-2. **Example for common and distinct features.**

The sets of common and distinct features enter equation (7.8) as a weighted sum $W(X) = \sum_{x \in X} weight(x)$. How features are weighted is discussed in the following section.

7.3.3 Feature Weights

The weight factors are introduced to give more influence to certain features. There are several alternatives for their definition based on the facets of the relationship the feature represents. Remember that a feature consists of three parts (*n, e, r*) where

- *n* is the partner entity of the relationship,
- *e* is the modality of the relationship,

- and *r* is the role the partner plays in the relationship, i.e., whether it is the *agent*. *patient,* or *partner.*

Role. The role information can only make a difference in situations in which an entity can be both *agent* and *patient,* i.e., if we deal with non-symmetric relationships whose arguments are of the same domain, in other words, if they connect equivalent entities. In the entity-relationship model presented in Figure 3-2 on page 45, the relationships *part-type, same-expression,* and *call* are such cases. In Figure 7-3(a), for example, the question is raised whether it is more important that T_2 is a part-type of T_1 than that T_1 is a part-type of T_3 if we compare T_1 to both F_1 and F_2. An argument could be that an abstract data type consisting of T_3 and F_2 needs T_1 whereas an abstract data type consisting of T_2 and F_1 does not need T_1. On the other hand, the abstract data type that includes T_1 is based upon T_2 and therefore $\{F_1, T_2\}$ is closer to T_1. None of the two arguments outdoes the other one. In general, considering the role an influence factor is rather questionable. At any rate, assigning different weights depending upon whether an entity is an agent or patient adds more complexity to this approach and complicates its calibration. Therefore, we do without the distinction.

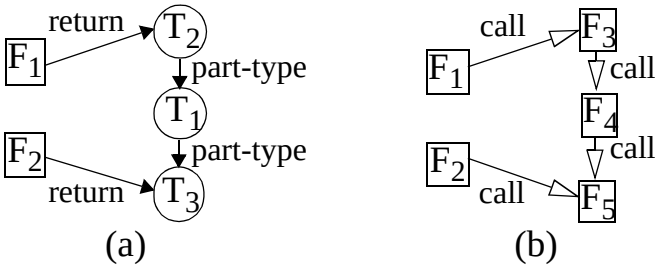


Figure 7-3. **Example for agent/patient difference.**

Note that this is no argument against the roles as such. This information is relevant when we compare relationships of the same class. For example, in Figure 7-3(b) it is plausible that F_4 is more similar to F_2 than to F_1 because both F_2 and F_4 call the same subprogram. The definition (7.9) for $Common_{eq}$ takes this already in account. That is why we have introduced roles in the first place.

Partner. Another facet of a feature is the related entity. This is the only facet Schwanke considered in his approach. He proposed to use **Shannon information content** from information theory as weighting factor *weight(x)* (see Section 5.9):

$$weight(x) = -\text{ld}(\text{probability}(x)) \quad \textbf{(7.12)}$$

Probability (x) is the number of how often the entity *x* occurs in any relationship divided by the number, *N*, of all entities of the system:

$$\text{probability}(x) = \frac{\text{occ}(x)}{N} \qquad \text{where } \text{occ}(x) = \left|\{y | x \propto y \vee y \propto x\}\right| \quad \textbf{(7.13)}$$

The quality of the *Shannon information content* is that it decreases by the frequency of an entity, hence lessening the relevance of frequently used entities. Just to get an impression what the actual weight proposed by the *Shannon information content* is, let us compare the weight for an entity, *x*, that occurs only once and an entity, *y*, whose probability is 0.125. We would consider the latter entity occurring unusually often. The weight ratio of the two of them is as follows:

$$\frac{\text{weight}(x)}{\text{weight}(y)} = \frac{-\text{ld}N^{-1}}{-\text{ld}2^{-3}} = \frac{\text{ld}N}{3}$$

Table 7-1 shows how this ratio evolves when *N* increases. For example, in a system with 65,536 entities, the weight ratio of an entity that occurs only once and an entity that occurs 8,192 times more often is only 16/3. In a system with 1,024 entities, the factor is 10/3 though the other entity occurs only 128 times more often. The reason for this is that the negative gradient of the logarithm rapidly decreases the farther it is from 0. For obvious reasons, we cannot reduce the number of entities in the system, but we may be able to reduce the basis on which the occurrence of an entity is established. We will come back to this below.

Table 7-1. **Example Shannon information content weights.**

N	$2^7 = 128$	$2^{10} = 1{,}024$	$2^{13} = 8{,}192$	$2^{16} = 65{,}536$
occ (x)	1	1	1	1
occ (y)	2^4	2^7	2^{10}	2^{13}
$1/3 \cdot \text{ld}N$	**7/3**	**10/3**	**13/3**	**16/3**

Establishing the weights with *Shannon information content* over all nodes is fine when we do not care for the kinds of relationships that actually occur. In Schwanke's approach, only subprograms are grouped and any usage of a non-local name counts no matter what it means. In our approach, the way of using the non-local entity is relevant. That is why we have to refine the determination of the weights by *Shannon information content*. For example, the subprogram F_3 in Figure 7-4 may be called by thousands of other subprograms and thus, gets a low weight according to equation (7.12) while F_1 and F_2 have a high weight since each is called only once. All these subprograms may set only one single global variable and therefore one should assume they are all equally similar with respect to this variable reference. Nevertheless, F_1 and F_2 are considered more similar than F_1 and F_3 because of the lower weight of F_3.

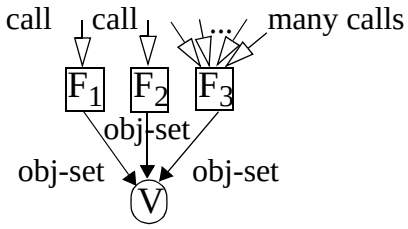


Figure 7-4. **Example for Shannon information content weights.**

Obviously, the way we have introduced to compute the weights is misleading when we take the kind of relationships into consideration (as it was the case in the variant of *Similarity Clustering* published with Girard and Schied in 1997). Equations (7.12) and (7.13) should therefore be refined as follows where we distinguish between different classes of equivalent relationships, *e,* and the roles, *r,* an entity plays in a relationship:

$$weight(n, e, r) = -\mathrm{ld}(\mathrm{probability}(n, E, r)) \qquad \textbf{(7.14)}$$

where E is a representative for all edge types equivalent to e.

$$\text{probability}(n, E, r) = \frac{\text{occ}(n, E, r)}{N_E} \quad \textbf{(7.15)}$$

$$\text{where occ}(n, E, r) = \begin{cases} \left|\{y|((n \rightarrow_e y)? \wedge e \sim E)\}\right| & \text{if } r = \text{agent} \\ \left|\{y|((n \leftarrow_e y)? \wedge e \sim E)\}\right| & \text{if } r = \text{patient} \\ \left|\{y|((n \leftrightarrow_e y)? \wedge e \sim E)\}\right| & \text{if } r = \text{partner} \end{cases}$$

N_E is the number of relationships of type E and all its equivalent relationships in the system (in the case of symmetric relationships, N_E is twice the number of actual relationships), i.e., *probability (n,call,agent)*, for example, tells us the relative frequency of *n* as a caller.

In equation (7.13), the basis on which we established the probability is the set of all nodes. In equation (7.15), on the other hand, the set of a specific kind of edge is used. This may increase or decrease the basis, depending upon the kind of edge. For *part-type*, for example, $N_{part\text{-}type} < N$ (where N is the number of nodes) is quite likely; for *call*, however, $N_{call} > N$ can be expected. As it was discussed above, decreasing the basis makes *Shannon information content* more sensitive.

Shannon information content is a problem-independent way to establish weights that does not take advantage of the kind and quality of the relationships among entities. However, depending upon what kind of atomic components one searches for, different kinds of relationships are of different significance. For example, when looking for an abstract data type, edges connected to user-defined types are more important than call edges, hence should be given more weight. An alternative to *Shannon information content* is therefore to assign fixed edge weights w_e to the kind of edges between entities. This strategy will be referred to as **relational weights**.

$$\textit{weight}(n, e, r) = w_e \quad \textbf{(7.16)}$$

Both ways to establish weights, namely, *Shannon information content* and relational weights, are orthogonal. Their respective strength can be combined by multiplying the two values. This strategy allows tuning clustering for specific patterns

and makes frequently occurring entities less important. This strategy will be referred to as **combined weight strategy**. The results reported in this thesis are based on the combined weight strategy:

$$weight(n, e, r) = -\mathrm{ld}(\mathrm{probability}(n, e, r)) \times w_e \quad \textbf{(7.17)}$$

7.3.4 Direct Relations

Direct relations represent immediate connections between two entities. Such relationships were explicitly excluded in the definition of common and distinct features in order to rate direct relations among two entities differently from relations to common and distinct neighbors. If two entities are directly related, they can generally be considered more dependent than if they were only related by a common third entity.

The contribution of direct relations to the similarity is computed as $Sim_{direct}(A, B)$. In terms of the resource usage graph, this is defined as the weighted sum of edges between *A* and *B*:

$$Sim_{direct}(A, B) = W(Link(A, B)) \quad \textbf{(7.18)}$$

where *Link (A, B)* denotes the actual links between A and B. *Link* can be defined in terms of features as follows:

$$Link(A, B) = \bigcup_{\propto} \{A \propto B | (A \propto B)?\} \cup \bigcup_{\propto} \{B \propto A | (B \propto A)?\} \quad \textbf{(7.19)}$$

This definition has two interesting properties. First, we count each relationship twice since $A \rightarrow B \Leftrightarrow B \leftarrow A$ and $A \leftrightarrow B \Leftrightarrow B \leftrightarrow A$. This is necessary because the weight of a feature depends on the related entity of the feature and it could be that $\mathrm{weight}(A \rightarrow B) \neq \mathrm{weight}(B \leftarrow A)$ due to a different *Shannon information content* of *A* and *B*. Second, the value of equation (7.18) is not normalized as opposed to a previous definition that we proposed in 1997 which divided $W(link(A, B))$ by all theoretically possible links between *A* and *B*. The normalized version promoted relationships that are the only possible ones between cer-

tain kinds of entities (such as the *of-type* relationship between variables and types) in an unjustified manner.

7.3.5 Informal Information

Programmers capture part of the meaning of programs in comments and in the names of functions, variables, and types. This helps them and other programmers to find their way around in a program. Another guide in a program is the file organization: related functions, variables, and types are often put together in one file (as already expressed by the *Same Module* heuristic). Both of these means of communication among programmers are examples of informal information. Usually, informal information is ignored by reverse engineering techniques (a notable exception is Biggerstaff, 1989) which focus on the information derived by a compiler. This section discusses how the information contained in the names of program identifiers and file organization can be relevant to the identification of atomic components.

Names of Identifiers. The naming of functions, variables, and types is an important source of information about a program given to a human reader. It has been observed (Biggerstaff, 1989) that even the author of a program has difficulties in recognizing the purpose of an excerpt from his code once significant identifier names have been replaced by insignificant ones (e.g., *f1* instead of *top_stack*). The naming of identifiers also convey information relevant to the identification of atomic components. For example, in one of the systems investigated, routines that belong to an abstract data type *list* had similar names: *list_insert*, *list_remove*, and *list_create*.

Two naming conventions are widely used for long identifiers built from many words: Separate words with underscore ('_') or start each new word with a capital letter (e.g., InsertWord). The following metric based on the number of common words between two identifiers exploits these conventions:

$$Sim_{words}(X, Y) = \frac{|\text{words}(X) \cap \text{words}(Y)|}{|\text{words}(X) \cup \text{words}(Y)|} \quad \textbf{(7.20)}$$

where *words (X)* denotes the set of words of *X*, i.e., all substrings of *X* separated by underscores or capital letters as delimiters.

An interesting feature of this definition is that it abstracts from the word lengths. Considering the length of the common words is generally not justified. For example, the similarity among the identifier pairs (*list_insert, new_list)* and (*list_insert, setInsert*) is equal according to equation (7.20) because of:

$$Sim_{words}(\text{list\_insert, new\_list}) = \frac{|\{\text{list, insert}\} \cap \{\text{new, list}\}|}{|\{\text{list, insert}\} \cup \{\text{new, list}\}|} = \frac{1}{3}$$

$$Sim_{words}(\text{list\_insert, setInsert}) = \frac{|\{\text{list, insert}\} \cap \{\text{set, insert}\}|}{|\{\text{list, insert}\} \cup \{\text{set, insert}\}|} = \frac{1}{3}$$

If the word length counted, *list_insert* and *setInsert* would be more similar. Actually, as a human, one would expect *list_insert* and *new_list* to be more similar, because *list* is a noun which probably stands for an abstract data type. An alternative, more functional point of view could be to consider *setInsert* and *list_insert* more related, depending upon whether a functional design was preferred to an object-oriented design. Without knowledge of the meaning of words, we cannot make such decisions. An interesting avenue not explored in this thesis is to investigate to which extent the meaning of words could be captured. Because we generally have a very restricted domain of discourse within programs and typical adjectives and verbs, such as *new* and *insert*, are very frequent, taking the vocabulary of a domain into account could be a promising and complementary approach. The purpose of this thesis, however, is to explore to which degree structural aspects can be leveraged.

When such word conventions are not used, identifiers are frequently constructed using a common prefix or postfix. For such cases, the following metric is used:

$$Sim_{suffix}(X, Y) = \begin{cases} 1 & X = Y \\ \dfrac{\text{prefix}(X, Y) + \text{postfix}(X, Y)}{1 + \text{prefix}(X, Y) + \text{postfix}(X, Y)} & X \neq Y \end{cases} \quad \textbf{(7.21)}$$

where *prefix* and *postfix* are the lengths of the common pre- and postfix of their two arguments if the length is longer than three characters; otherwise they are zero.

Organization of Files. The division of a program into files also conveys some information about the meaning of a program. Related functions and variables are often put in the same file or in files with a common substring in their name (e.g., *client-db* and *client-service*). The previous metric for identifier name similarity based on pre- and postfixes is used to compare the names of the files without extensions (i.e., only *file* in *file.c* or *file.h*) in which the entity, *X*, is declared, denoted by *filename*(*X*):

$$Sim_{filename}(X, Y) = Sim_{suffix}(filename(X), filename(Y))$$

Informal knowledge should be seen as a complementary source of information. In an interactive approach, one should always use two modes: One that considers informal information and one that does not. Informal information can be helpful, but may also be misleading.

7.4 Clustering Result

The result of the hierarchical clustering described by Algorithm 7-2 is a **dendrogram**, i.e., a binary tree (or a forest of binary trees if clustering ends before all entities are grouped because all remaining entities are not similar enough) whose leaves are clustered entities and whose inner nodes represent unions of two subclusters or entities (see Figure 7-5). Each inner node is associated with the similarity value of its two subtrees. The farther the nodes are from the root node, the more similar they are in terms of the similarity metric because the dendrogram is generated bottom-up and the most similar subtrees are combined first.

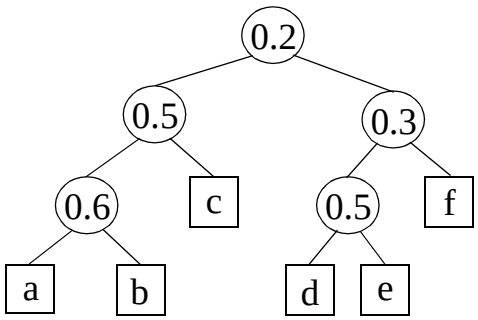


Figure 7-5. **Example dendrogram.**

A dendrogram is a useful information to the user because it shows the order in which entities are clustered and the respective most similar entities. Hence, the result of *Similarity Clustering* should be presented as a dendrogram to the user. The user can then manually select components from the dendrogram. However, in order to retrieve components automatically and also to further process the results of *Similarity Clustering* by other techniques that expect the results as sets of entities (as it will be described in Chapter 8), subtrees of a dendrogram can also be flattened using a user-determined similarity threshold. The threshold determines the minimal acceptable similarity of a subtree to be flattened where the similarity of a subtree is the similarity value associated with its root. As already mentioned, the similarity value of all dendrogram nodes below this root is greater or equal to the similarity value of the root node. In order to retrieve candidates from a dendrogram, Algorithm 7-3 is used.

Consider the example dendrogram in Figure 7-5. If 0.5 is used as a threshold, the algorithm starts at leaf *a* and climbs up until it reaches the node with a parent whose similarity value is below the threshold, namely, the node associated with 0.5. The leaves of the subtree rooted by this node, namely, *a*, *b*, and *c*, are clustered to a candidate {*a, b, c*} and added to the list of candidates. Because *b* and *c* have been marked visited, the next bottom-up traversal starts at *d*. This time, only one step up is taken and the proposed candidate is {*d, e*}. The leaf *f* is not clustered because its parent is associated with a similarity value below the threshold.

7.5 Integration of Other Approaches

The basic connection-based techniques introduced in Chapter 5 can partly be integrated with the general approach of *Similarity Clustering* without major changes. Some of the heuristics are even already a constituent of *Similarity Clustering*, though they are used to yield additional clues among other leveraged aspects rather than in the definite manner of the other approaches. For example, the assumption of *Same Module* that entities declared in the same module are more related than entities in different modules is covered by *Similarity Clustering* as informal similarity aspect $Sim_{filename}$. Moreover, the similarity of objects occurring in the same expression can be increased by assigning more weight to the *same-expression* relationship, hence, the *Same Expression* heuristic is incorpo-

Input:

- dendrogram D
- threshold Θ

Output:

- list of flat candidates C

Algorithm:

1. initialization:

for each leaf L **in** D **loop** visited (L) := false; **end loop**;
C := *empty_list*;

2. retrieval:

for each leaf L **where not** visited (L) **loop**
N := L**;**
while parent (N) ≠ ⊥ **and then** Sim (parent (N)) ≥ Θ **loop**
N := parent (N);
end loop;
if Sim (N) ≥ Θ **then** add {l | l is a leaf of N} to C **end if**;
mark all leaves of N as visited
end loop;

Algorithm 7-3. **Retrieving candidates from a dendrogram.**

rated as well. Likewise, *Global Object Reference* can be simulated by assigning high weights to object reference relationships and setting all other weights as well as informal information parameters to 0.

Part Type. The integration of the *Part Type* heuristic needs more work. The claim of *Part Type* is that a subprogram does not belong to a type of its parameter list that is a part-type of another type in this list. Given the resource usage graph on the left hand side in Figure 7-6, from the point of view of the *Part Type* heuristic, the *parameter-of* from F_1 to T_1 should be eliminated. This could be simulated in *Similarity Clustering* by setting the weight of the relationship for this *parameter-of* edge to 0. However, this means that the weight of a relationship does not only depend upon its type, the related node, and its role but also on the context of a specific instance of this relationship. *Similarity Clustering* can be extended in this

respect. When the similarity metric is computed for the base entities during initialization, the specific contexts can be checked and the weights for such signature edges be lowered or even set to 0. The advantage of *Similarity Clustering* is that is does not necessarily have to set this value to 0; it could also be decided only to reduce the weight. In particular, the weight could be reduced only if there is also an internal access to the container type in this context, because only then the part type could be really put into the container type (or retrieved from it).

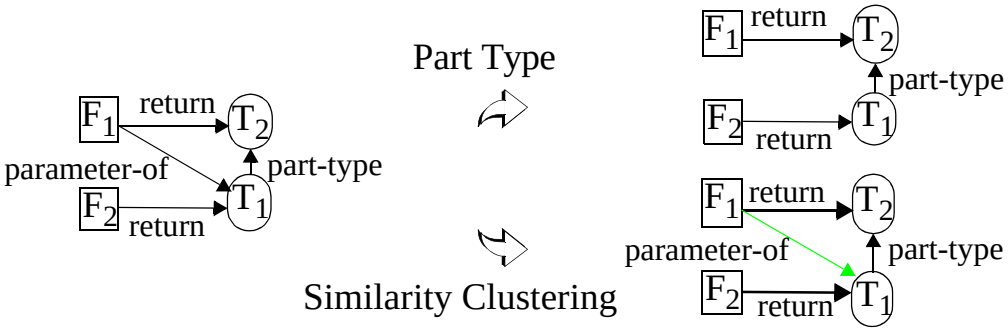


Figure 7-6. ***Part Type* captured in similarity metric.**

Note that even if the weight is set to 0 for *Similarity Clustering*, the approaches need not necessarily yield the same result. *Part Type* iterates over the subprograms and classes them with the related types producing the candidates $\{F_1, T_2\}$ and $\{F_2, T_1\}$ whereas *Similarity Clustering* could first cluster T_1 and T_2 due to the *part-type* relationship (ignored by *Part Type*) and then add the other two functions. However, this is only possible if the *part-type* relationship has a higher weight than the *return* relationship.

Internal Access. The *Internal Access* heuristic groups all subprograms that access a record component of the same type or record variable (see Section 5.6). This may be correct from the information hiding point of view. However, in reality for reasons of efficiency, some of the record components of a type may be accessed from subprograms that do not belong to the type or variable. These are often components of a simple type telling something about the general state of the type or variable. For example, a list data structure has a component *length* of type int that counts the number of elements of the list. Instead of providing a function that returns the value of this component, the programmer may allow access to this component from outside. There may be other components that are not intended to

be used from outside because they are associated with a more complicated control logic. Thus, there can be two kinds of record components: **public** and **private record components**. The *Internal Access* heuristic does not distinguish between them and therefore produces too large components if there are public components.

Since there is no means in the language C to distinguish public from private record components, they cannot easily be distinguished. However, it is likely that a record component is meant to be public when it is frequently accessed. Another indicator is that public record components are generally only read by subprograms that do not belong to the type; writing these components is mostly done by explicit accessor routines of the type because there may be complex consistency constraints to maintain among the record components of this type. *Similarity Clustering* as proposed so far is capable to address these two attributes. Frequently referenced record components have less weight according to the *Shannon information content* used to ascertain the weights of features in Section 7.3.3. In order to distinguish references to record components from references to variables, we have already refined the *set*, *use*, and *take-address-of* relationships in Section 4.2.7 where we distinguish references to record components from references to the object as a whole. Furthermore, we have specified in Section 4.2.6 that each composite variable has its own tree of statically accessible record components according to the type of the variable.

Using these information, *Internal Access* is integrated as follows:

- frequently used public record components have less weights by way of the *Shannon information content*
- accesses to record components have higher weights than ordinary accesses by assigning higher edge weights to *comp-set*, *comp-use*, and *comp-address-of* than to *obj-set*, *obj-use*, and *obj-address-of*
- internal sets have higher weights than internal uses by assigning higher weights to *comp-set* than to *comp-use*

7.6 Establishing the Similarity Metric Parameters

A problem of *Similarity Clustering* in practice is that it has many parameters that have to be adjusted. Research in cluster analyses has produced statistical analyses to improve clustering (Steinhausen and Langer, 1977). However, they cannot be applied to our problem because of the way we had to define our similarity metric.

A similarity metric is normally defined with respect to specific features that are *absolute* to all entities to be grouped. For example, in order to cluster cells in micro biology, we can measure their size, coloring, protein content, and so forth. Each of these features constitutes an absolute scale on which each cell can be measured and hence, we can represent a cell as a vector of such absolute measurements. Similarity among two cells can then be expressed by alternative distance metrics, typically the euclidean metric (see Steinhausen and Langer, 1977, for other metrics). Our similarity metric is defined as a relation directly between two entities rather than to a third absolute point of comparison. That is why we cannot use the statistical methods suggested for traditional similarity metrics.

There are basically three layers at which parameters have to be adjusted:

- edge weights (7.16)
- parameters in individual aspects of similarity, namely, d and I_{eq} in $Sim_{indirect}$ *(7.8)*
- influence factors x_i of the aspects of similarity on the similarity among two entities (7.3)

The similarity for groups of entities does not have parameters. In the following, we are going to discuss how the parameters at the respective layers can be established.

7.6.1 Statistical Analysis of Edge Distribution

As a completion of Schwanke's suggestion to use *Shannon information content* to establish weights of features, we have proposed to assign weights to the kind of relationships, or technically speaking, to the edge types. In order to answer the questions of whether the edge types count at all and how one can find appropriate

edge weights, we investigated the distribution of edge types in the reference components described in Section 6.1.

7.6.1.1 Scope for the Data

What edges we consider for the statistical analysis described below depends on the nodes regarded in the first place. The nodes will be described first before we go into details on how the statistical analysis is performed.

Nodes considered. Because only a portion of all entities in the subject systems really belongs to an atomic component (according to our analysts), considering a global analysis of edge distribution does not make sense. Instead, we regard only entities within **atomic component contexts**. An atomic component context contains any node within an atomic component and any node that is a neighbor of an atomic component element.

Example. There is one atomic component {*f3, f5, f6, v1, v2*} in Figure 7-7. Solid and dashed edges represent calls and references, respectively. *f1*, *f2*, *f4*, and *f8* are neighbors of at least one atomic component element. Therefore, {*f3, f5, f6, v1, v2*} ∪{*f1, f2, f4, f8*} is the atomic component context. *f5*, *f6*, and *f7* do not belong to this context because none of their neighbors belongs to the atomic component.

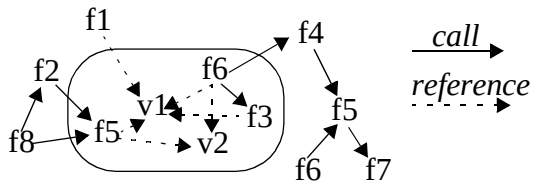


Figure 7-7. **Example for atomic component contexts.**

Edges considered. In the following, we will regard only edges that are relevant to the similarity among nodes in atomic component contexts. These are all edges that occur in *Link, Common,* or *Distinct* of nodes within an atomic component context (where we consider only one atomic component context at a time). If *a* and *b* are within the same atomic component context, *Common(a,b)* and *Distinct(a,b)* can only contain edges with at least one end within the atomic component context according to the definition of *Common* and *Distinct*; however, the other end of the edge could be a node outside of the atomic component context,

like the target of the *call* from *f4* to *f5* in Figure 7-7. For edges in *Link(a,b)*, even both ends must be in the atomic component context because *Link* contains only edges between *a* and *b* which are both in the atomic component context.

Concentrating on atomic component contexts helps us to gather information why some nodes are jointly inside an atomic component and others are not. Therefore, we will count the edge types that occur in *Link*, *Common*, or *Distinct*, respectively, in a comparison of nodes that are both inside the **same** atomic component and nodes of **different** atomic components. Nodes that do not belong to any atomic component at all will be considered an atomic component of their own. The distinction of the edge distribution in same and different atomic components gives us insights what the contribution of an edge type is for two elements being in the same or in different atomic components. If an edge type occurs more often among nodes in the same atomic component than among nodes in different components, it should get more weight and vice versa.

The same edge can occur several times in the statistics because it can occur in *Link*, *Common*, or *Distinct* of different pairs of nodes. Distinguishing among *Link*, *Common*, and *Distinct* can give us additional hints on the weighting of these individual aspects of similarity.

Finally, we have to distinguish among different kinds of atomic components because it is clear from their definition that certain edge types can never occur within specific atomic components. For example, we cannot find a *use* edge within an abstract data type. Otherwise there would have to be a variable within the atomic component and thus, it would no longer be an abstract data type but a hybrid component.

To sum it up, the comparison has the following dimensions:

- edge types
- *Link* / $Common_{eq}$ / $Common_{ne}$ / *Distinct*
- same / different atomic component
- kinds of atomic component

7.6.1.2 Used Data

The data in the following statistical analysis will be ascertained for one atomic component at a time. For reasons of readability, we will not explicitly use indices for individual atomic components in the following presentation of the way the information is computed. For the same reason, we will also do without any index for the kind of atomic component. The following equations should be understood as being applied to individual atomic components of the same kind.

Recall that the edge type weight is multiplied by the *Shannon information content* to obtain the resulting feature weight as proposed by equation (7.17). If we simply counted the occurrence of each edge type, the statistics would not be adequate when Shannon information content is used to balance frequently used entities. Therefore, the statistics has to be based on equation (7.17). That is why the accounting of edge types is by means of

$$W(X) = \sum_{x \in X} weight(x)$$

with

$$weight(n, e, r) = -\mathrm{ld}(\mathrm{probability}(n, e, r)) \times w_e$$

in the following where $w_e = 1$ is assumed for all edge types. If Shannon information content is not considered, one has to compute the following formulas with *weight (n,e,r)* = 1 to get an edge distribution based on pure edge occurrences.

We are going to compute the following figures for each edge type, *e,* for the context of a given atomic component, *A* (assuming $w_e = 1$ for the computation of *W(X)*; furthermore, it is assumed that the nodes are enumerated):

$$\mathrm{Link}_e^s(A) = \sum_{a_i, a_j \in A \wedge i < j} W(\Phi_e(\mathrm{Link}(a_i, a_j))) \qquad \textbf{(7.22)}$$

$$\mathrm{Link}_e^d(A) = \sum_{a \in A \wedge b \in \mathrm{context}(A)/A} W(\Phi_e(\mathrm{Link}(a, b))) \qquad \textbf{(7.23)}$$

where $\Phi_e(X) = \{m \propto_e n \mid (m \propto_e n) \in X\}$ is a filter that sorts out all features not of type *e*. $\text{Link}^s_e(A)$ considers only elements that are both in the same atomic component *A*, whereas $\text{Link}^d_e(A)$ comprises features of entities where only one node is part of *A*.

Given these definitions, we can compute for each edge type and similarity aspect the ratio of edges for nodes in the same atomic component, *A*, and edges between nodes within *A* and nodes outside of *A*:

$$LinkRatio_e(A) = \frac{\text{Link}^s_e(A)}{\text{Link}^s_e(A) + \text{Link}^d_e(A)} \quad \textbf{(7.24)}$$

The average of this ratio over all atomic components is the probability for an edge of kind *e* to link nodes in the same atomic component, in other words, it describes the "natural" portion of edges of type *e* within an atomic component (let $\mathcal{A}$ be the set of reference components used for calibration):

$$LinkRatio_e = \frac{1}{|\mathcal{A}|} \cdot \sum_{A \in \mathcal{A}} LinkRatio_e(A) \quad \textbf{(7.25)}$$

Only if $LinkRatio_e$ is greater than 0.5, the edge type *e* is significant. Hence, the edge weight can be set in correlation to this ratio. We can even use this ratio itself as weight for *e*. If the ratio is greater than 0 but less than 0.5, the weight should not be 0 (or even negative), because at least some edges of type *e* are within detected atomic components. Analogously, the weight for *e* should not be 1 (or greater than 1) if the ratio is greater than 0.5 but less than 1 because then at least some edges of type *e* are between entities not in the same atomic component. Hence, *LinkRatio* represents an appropriate weight.

Definitions analogous to equations (7.22), (7.23), (7.24), and (7.25) can be made for $Common_{ne}$, and $Common_{eq}$. Because *Distinct* is used to discriminate entities as opposed to the other similarities, we use the following formula to measure the

Distinct ratio where a high value indicates that the edge type is an important grouping factor:

$$DistinctRatio_e(A) = 1 - \frac{\text{Distinct}_e^s(A)}{\text{Distinct}_e^s(A) + \text{Distinct}_e^d(A)} = \frac{\text{Distinct}_e^d(A)}{\text{Distinct}_e^s(A) + \text{Distinct}_e^d(A)} \quad (7.26)$$

where

- $\text{Distinct}_e^s(A)$ is defined analogously to (7.22)
- $\text{Distinct}_e^d(A)$ is defined analogously to (7.23)

7.6.1.3 Data for Aero, Bash, CVS, and Mosaic

This section describes the edge ratios for *Link*, $Common_{ne}$, $Common_{eq}$, and *Distinct* as defined in the previous section for the systems Aero, Bash, CVS, and Mosaic. For readability reasons, the bars for *Distinct* actually present *1-DistinctRatio* as defined by (7.26) and the edge type names are abbreviated as listed in Table 7-2.

Table 7-2. **Abbreviations for edge types.**

actual-parameter-of	AP	*comp-set*	CS	*obj-address-of*	OA
call	CL	*comp-use*	CU	*obj-set*	OS
same-expression	SE	*parameter-of*	PA	*obj-use*	OU
local-obj-of-type	LT	*return*	RE	*part-type*	PT
comp-address-of	CA	*of-type*	OT	*delineate*	DE

Ratios for abstract data types. For ADTs the *Link* ratio of *obj-address-of*, *obj-set*, *obj-use*, *same-expression*, and *actual-parameter-of* must be 0 because otherwise a variable would be contained in the component and, therefore, the component be categorized as hybrid. The edge ratios are shown in Figures 7-8, 7-9, 7-10, and 7-11 for the respective system.

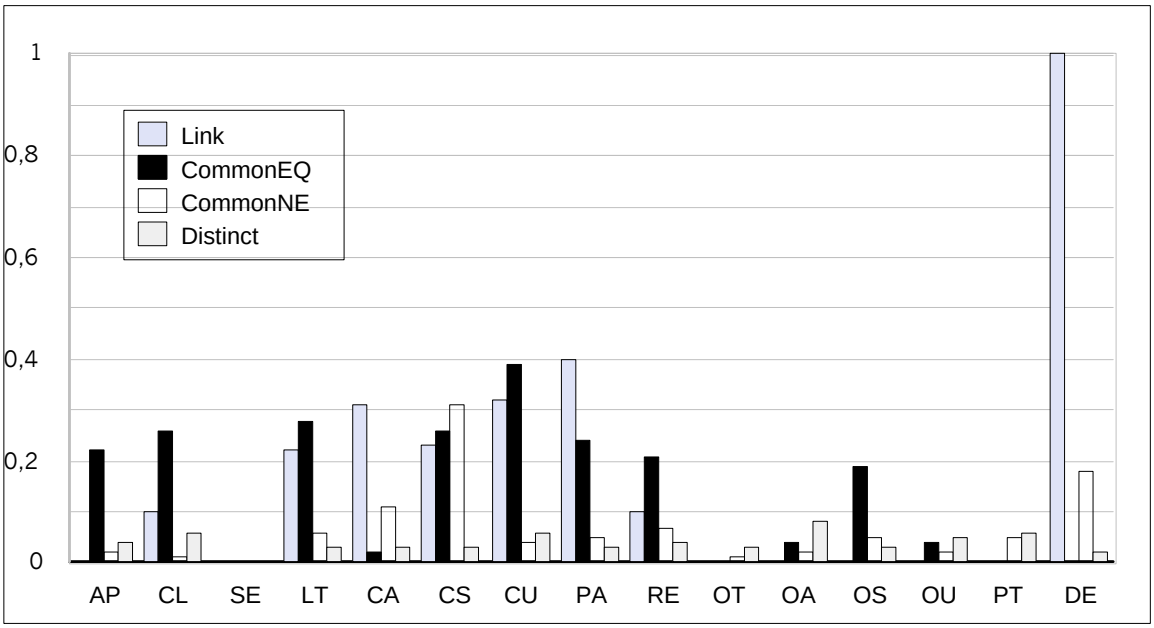


Figure 7-8. **Ratios for ADTs in Aero.**

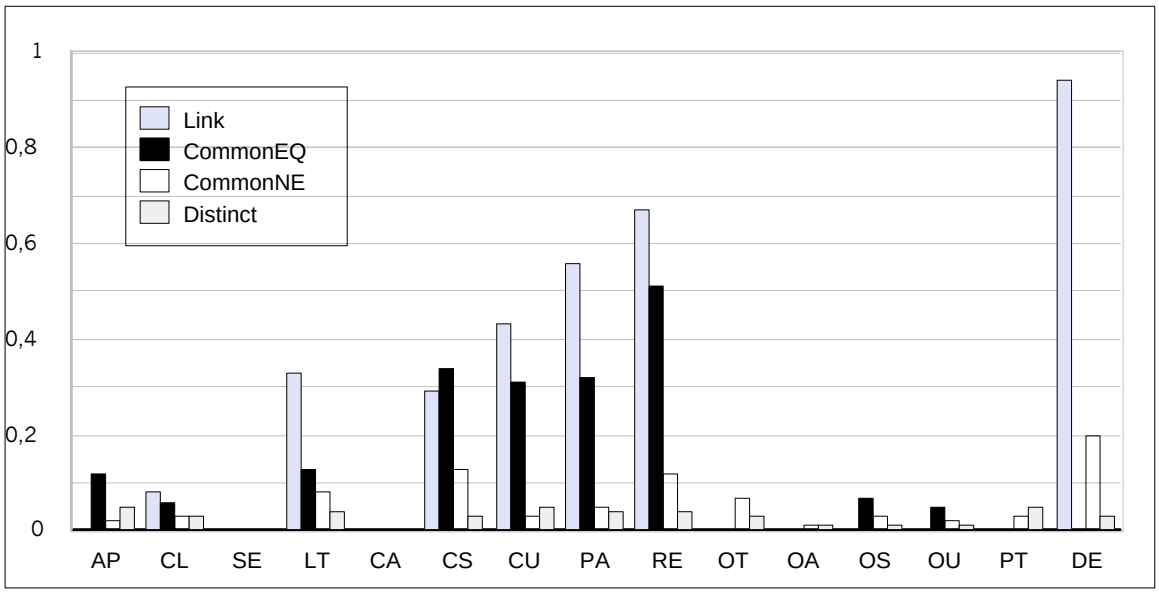


Figure 7-9. **Ratios for ADTs in Bash.**

- **Link**: It is not surprising that *delineate* has such a high significance for *Link*; programmers introduce a synonym for a type and adhere to it. Similarly, the expected higher *Link* ratios for signature and component references can be

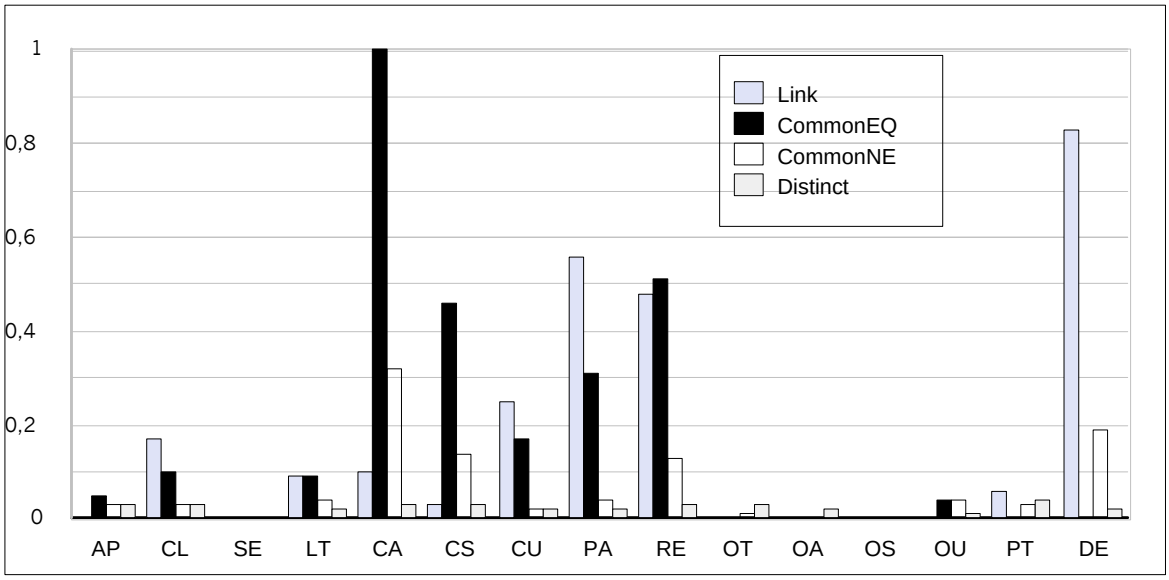


Figure 7-10. **Ratios for ADTs in CVS.**

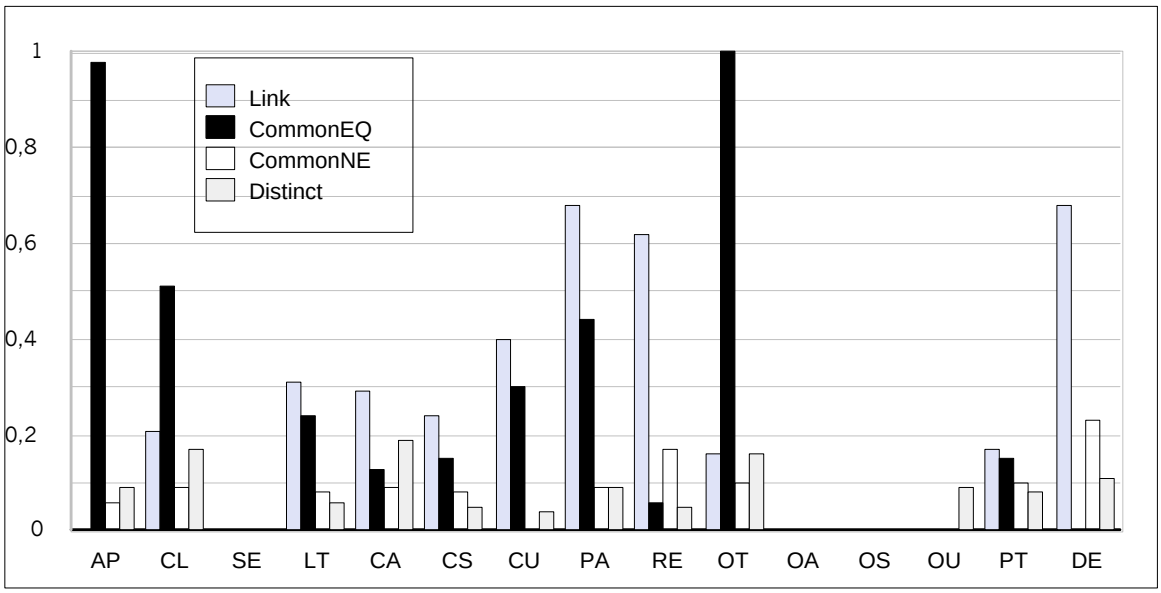


Figure 7-11. **Ratios for ADTs in Mosaic.**

found in the charts; only in the case of *Aero*, *return* are less significant among direct links.

- **Common$_{eq}$**: The ratios for *Common$_{eq}$* is in most cases close to the *Link* ratios for the diverse edge types. The peak of *comp-address-of* in CVS can be explained by the very small number of cases in which the addresses of record

components are taken and which all happened to be by subprograms of the same atomic component. Neither is the peak for *of-type* within Mosaic representative. In comparison to other systems, the value for $Common_{eq}$ is surprisingly high for Mosaic. This means that a variable that is an actual parameter to one accessor function is also an actual parameter to most other accessor functions of the same component.

- **$Common_{ne}$**: The charts for $Common_{ne}$ suggest that non-equivalent features are less significant. The difference between the ratios of $Common_{eq}$ and $Common_{ne}$ supports the distinction of shared features in equivalently and non-equivalently related ones that I proposed as an improvement to the original approach.
- **Distinct**: The data for all systems suggest no significant difference among edge types within *Distinct*. Note that the bars for *Distinct* actually present *1-DistinctRatio*, i.e., the actual *DistinctRatio* as defined by (7.26) is virtually 1 for all edge types.

ADO ratios. In ADOs, *parameter-of*, *return*, and *part-type* cannot appear otherwise we would deal with a hybrid (Figures 7-12, 7-13, 7-14, and 7-15). For ADOs, the remaining reference relationships *take-address-of*, *set* and *use* — in particular, those for global objects — are dominating as expected. Furthermore, *same-expression* is also an important factor which justifies the *Same Expression* heuristic (the peak of *same-expression* in Mosaic is not representative because there is only one such edge). Neither surprising is the fact that *obj-set* is more important than *obj-use* in most cases. The data support our hypothesis that setting objects is often a more critical operation because it may involve checking certain consistency constraints or performing non-trivial algorithms on a complex structure and for this reason, clients of a component avoid changing the state of a component directly. The reason why *comp-set* is less significant than *comp-use*, which apparently contradicts our hypothesis in the case of ADTs, is the way dereferences are handled by the resource usage analysis. As it was discussed in Section 4.2.8.2, an assignment to a record component by means of a dereferenced pointer, like `c->a = 1`, is actually considered a usage of the record component. Because most types of ADTs are dynamic data structures and, hence, accessed by means of dereferenced pointers, many references to record components that would be classified as assignments by a reverse engineer are actually considered

comp-uses by the resource usage analysis that follows the compiler's point of view.

It is also interesting to see how the ratios for all kinds of references change from system to system indicating that Aero and Bash are more permissive than CVS and Mosaic with respect to references to global variables and constants of abstract data objects. The high ratios for $Common_{eq}$ in Bash for *return* edges and in Mosaic for *comp-address-of* edges are due to the very rare number of edges that happened all to be in the same atomic component. These figures are not representative.

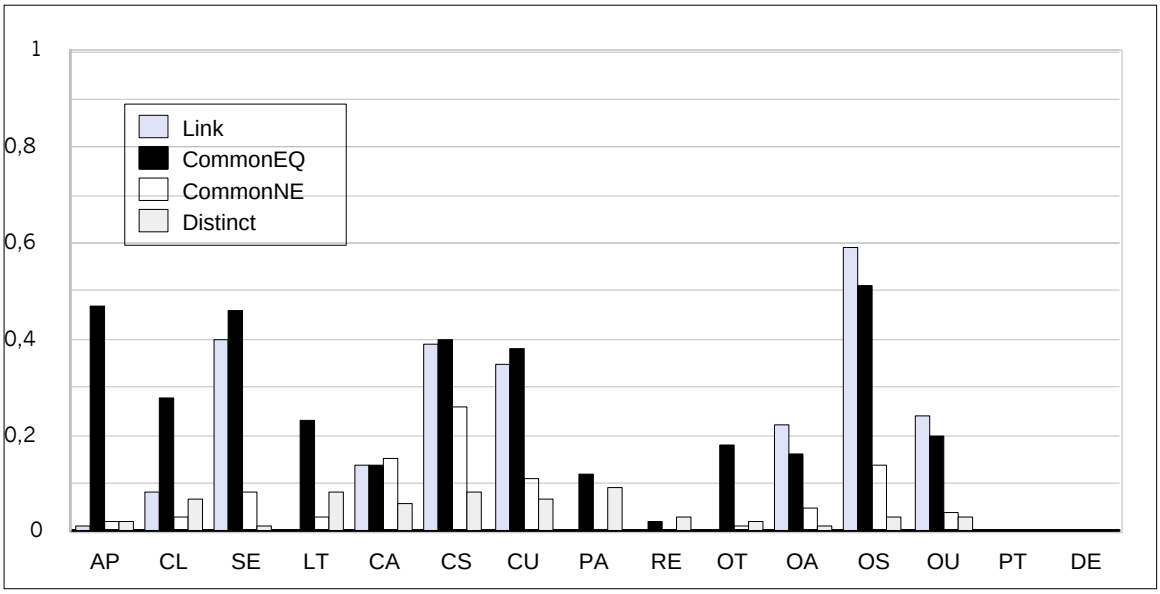


Figure 7-12. **Ratios for ADOs in Aero.**

The overall conclusion drawn from these data is that the actual influence of the edge types in the respective similarity aspects depends on the system. As a consequence, the weights gained for one system cannot necessarily used for a different system. However, in this thesis, we considered very different kinds of systems from different authors. For a family of systems of a common application domain and for programmer teams with established programming conventions, there could be less divergence of edge ratios.

On the other hand, the data clearly reveal differences among the edge types and these differences are similar for all systems — supporting our approach to assign

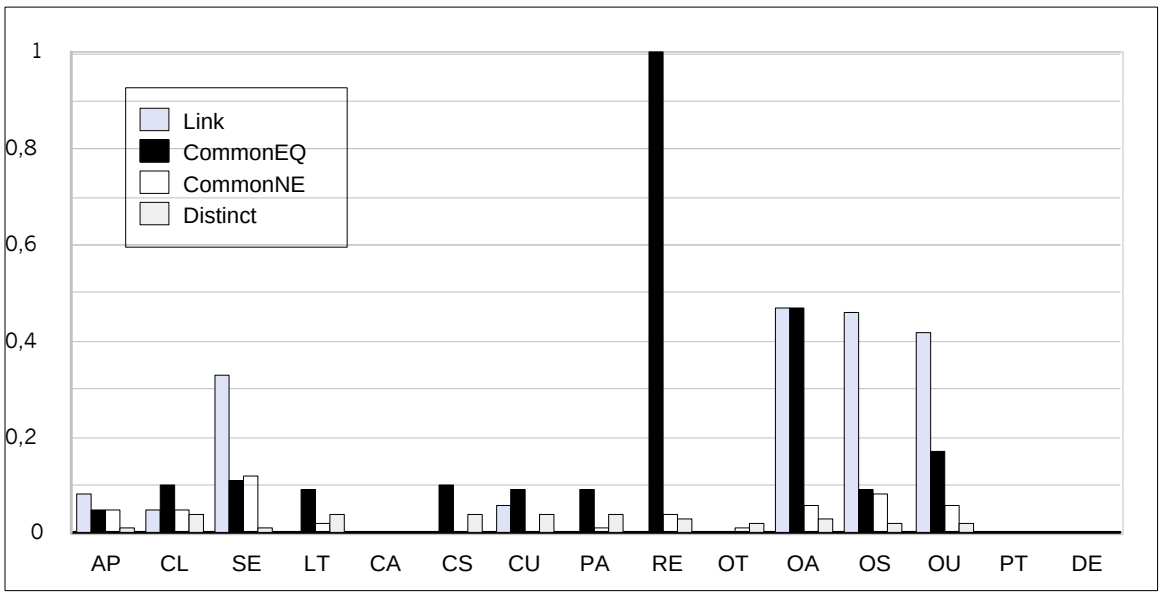


Figure 7-13. **Ratios for ADOs in Bash.**

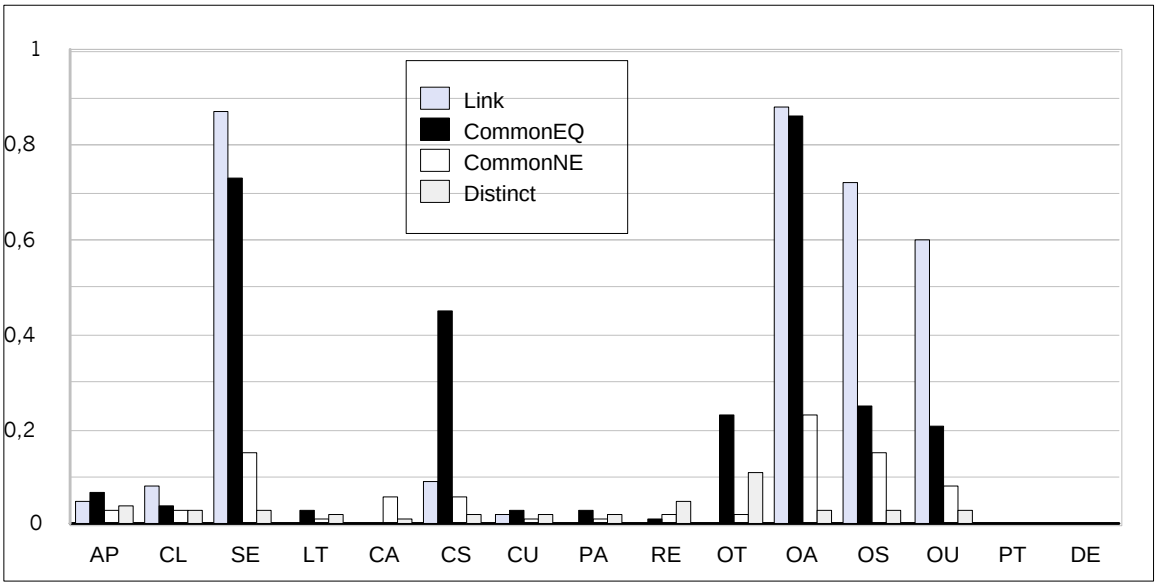


Figure 7-14. **Ratios for ADOs in CVS.**

different weights to different edge types. Furthermore, the different aspects of similarity, namely, *Link*, $Common_{eq}$, $Common_{ne}$, and *Distinct*, actually have different influence; *Link* and $Common_{eq}$ are most important.

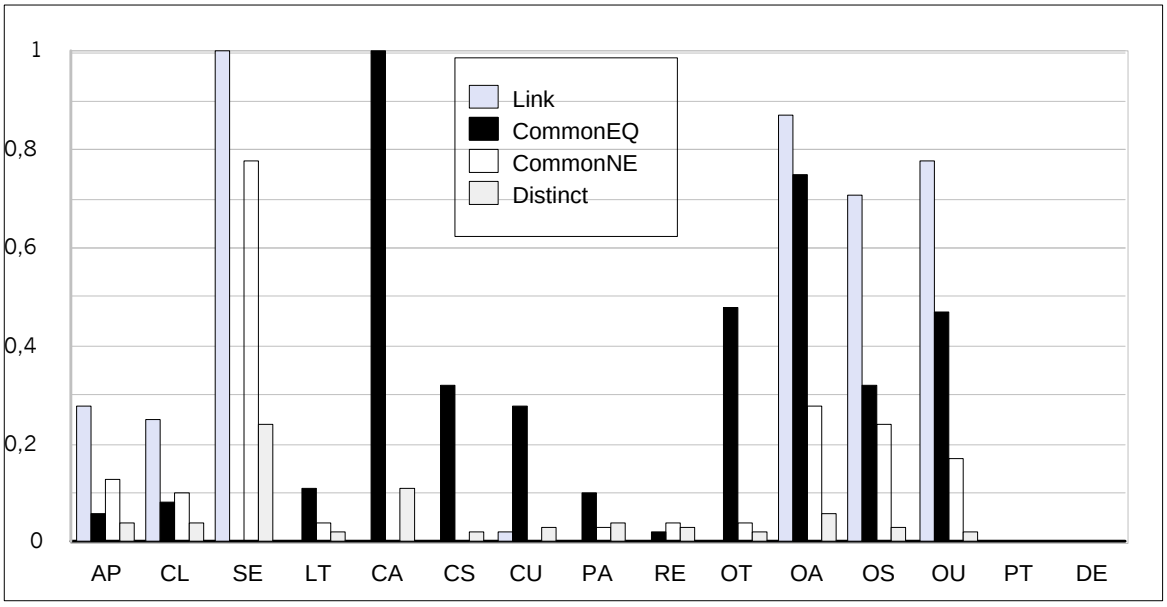


Figure 7-15. **Ratios for ADOs in Mosaic.**

The figures of *Link* for record component and object references also allow insight into the degree of information hiding of the respective systems. There is generally only little information hiding for ADTs for all systems; in the case of CVS, there is even virtually no encapsulation of ADTs at all. On the other hand, information hiding for ADOs ranges from little (Aero) over medium (Bash) to good (CVS and Mosaic). These data indicate that the means for information hiding are more often used for ADOs than for ADTs. However, even for ADOs, there are still many data references that bypass the accessor functions.

7.6.2 How to Find the Parameters

As the statistics in Section 7.6 show, the parameters of *Similarity Clustering* have to be adjusted before it can be applied in practice. This can be done by compiling the atomic components of a sample of the system either manually or by using one of the other techniques. The sample is then used to calibrate the diverse parameters, namely, the edge weights and the individual influence factors of the similarity aspects. With these parameters, *Similarity Clustering* is applied to the whole system. Browsing and selecting the proposed candidates, one adds reference components, which are appended to the previous sample and which can be used for calibration in the next cycle. Of course, not all candidates have to be validated. In

the simplest case, only one candidate component could be treated per iteration. This calibration process is repeated until a parameter setting is found that works well for a good part of the system or until the parameters seem not to change anymore. Fortunately, all steps of this process apart from the validation can be automated. How the detection quality can be judged was already discussed in Section 6.2.2.

Two alternative classes of calibration approaches to find appropriate parameters for a given sample are explored in this thesis. The first one consists of two steps: (1) ascertaining the "typical" edge occurrences in atomic component contexts and (2) finding adequate influence factors with established optimization strategies. The second approach is based on a multi-dimensional contingency table — derived from the sample — that describes the probability for two entities having a certain vector of *Link*, $Common_{eq}$, $Common_{ne}$, and *Distinct* values to be in the same atomic component. Both approaches are presented and evaluated in this section.

7.6.2.1 Traditional Optimization Strategies

Similarity Clustering has a large number of parameters mainly due to the diverse edge types. In order to reduce the problem space of *Similarity Clustering*, we firstly try to find appropriate weights for the edge types and secondly adjust the other parameters.

Edge weights. The edge weights contribute to the direct and indirect similarities between two entities. A high value has the effect of attracting two entities during clustering. Therefore, the likelihood for them to be in the same atomic component increases.

The model described by (7.3) on page 189 assumes equal weights for the same edge type no matter to which similarity aspect the edge type contributes, i.e., whether it appears in *Link*, *Common*, or *Direct*. The influence parameters of the similarity aspects are used to make the difference. That is, for establishing the edge type weights, we do not care about whether there is a different edge type distribution among *Link*, $Common_{eq}$, $Common_{ne}$, and *Direct*; we can simply use the relative frequency of edge types in the context of atomic components as follows:

$$EdgeRatio_E = \frac{\text{inside}_E}{\text{inside}_E + \text{across}_E} \quad \textbf{(7.27)}$$

where $inside_E$ is the number of edges of type E in an atomic component and $across_E$ is the number of edges across the border of atomic components.

It makes good sense to use the edge type ratios as edge weights because they tell something about the "natural" consistence of an atomic component. Using a high edge ratio for *set* edge weights draws many *set* edges into the atomic component while a low edge ratio for *actual-parameter-of* has only little attraction. The edge ratio can then be combined with the Shannon information content as discussed in Section 7.3.3. Note that Shannon information content is based on the frequency of nodes and does not interfere with the edge type frequency.

For the automation of the search for good parameter values, there are several alternatives that will be presented in the following paragraphs. There are many other approaches that are not investigated here. It would go beyond the scope of this thesis to try all of them. The selected ones can be viewed as typical.

Grid Search. Given an interval that defines the search space for each influence factor and a step by which the search progresses, we can search all over a grid of the four-dimensional space spanned by *Link*, $Common_{eq}$, $Common_{ne}$, and *Distinct*. The visited space coordinates of the grid depend on the chosen step. On one hand, the step should not be too wide, otherwise maxima could be skipped; on the other hand, a short step will dramatically increase the time needed for the search. Due to its very large number of iterations, grid search is only feasible for small samples.

Gauß-Seidel Strategy. The Gauß-Seidel strategy follows the strategy of hill climbing by varying a single parameter at a time and analyzing the effect on the detection quality. It first checks for the direction in which the parameter is going to be changed. If the detection quality increases when the parameter is increased, the parameter will be further increased. If the detection quality decreases instead, the parameter will be further decreased. Then the parameter will be changed in

the chosen direction until no improvement can be achieved anymore. The parameter is frozen and the next parameter is adjusted.

Simulated Annealing. Simulated annealing is a widely used algorithm for combinatorial optimization problems. It is based on the following analogy between a combinatorial optimization problem and a physical system (Aarts and Korst, 1990):

- Solutions in a combinatorial optimization problems are equivalent to states of a physical system.
- The cost of a solution is equivalent to the energy of a state.

In condensed matter physics, annealing is known as a thermal process for obtaining low energy states of a solid in a heat bath. The process contains the following two steps:

1. Increase the temperature of the heat bath up to a maximum value at which the solid melts.
2. Decrease carefully the temperature of the heat bath until the particles arrange themselves in the ground state of the solid.

In the liquid phase, all particles of the solid arrange themselves randomly. In the ground state, the particles are arranged in a highly structured lattice and the energy of the system is minimal. The physical annealing process can be modeled successfully by using computer simulation methods from condensed matter physics. The general algorithm is well-known and therefore not presented here (the reader is referred to Aarts and Korst, 1990). Simulated annealing was introduced above as a search for the minimum. Since the maximal detection quality is searched in the sample of reference components, the simulated annealing algorithm is adjusted to find a state of maximal energy. Furthermore, the process ends when the number of maximal iterations (set to 15) is reached or the improvement with respect to the best solution found is less than five percent after at least half of the maximal iterations have been completed.

Evolution Strategies. Another family of optimization techniques that have recently attracted attention are evolution strategies. In order to adopt evolution strategies to find reasonable parameters, one could view parameter settings as

individuals exposed to evolution. However, evolution strategies require a large number of individuals per generation — typically at least 50 — and several iterations. Since each clustering process with *Similarity Clustering* can take several minutes on a Sun Sparc Ultra-60, the time needed with evolution strategies were simply too long. Doval et al. have explored evolution strategies for clustering in more detail (1999).

7.6.2.2 Sample Partitioning

Unfortunately, the optimization approaches described above need several iterations for calibration. The calibration itself involves clustering of the sample and is therefore a rather expensive operation. It would be favorable to have a more direct way to find the parameters.

The optimization approaches above were used to calibrate the individual influence factors of the similarity aspects within equation (7.3) on page 189. The model described so far assumes equal weights for edge types no matter to which similarity aspect they contribute. This need not necessarily be the case but was done to limit the degrees of freedom of *Similarity Clustering*. Adding individual influence factors of the similarity aspects depending on the edge types adds more parameters. However, if there is a sample of reasonable size, these parameters can automatically be ascertained.

In experimental designs, so-called contingency tables are used to test main effects and interactions of factors (Winer et al., 1991). A **contingency table** is an *n*-dimensional table where each dimension $i \in 1 \ldots n$ represents an independent variable V_i with $1 \ldots m_{v_i}$ levels; a cell $p_{v_1 \ldots v_n}$ of the contingency table is the frequency for observed values of the dependent variable at the level v_1 of variable V_1, v_2 of variable V_2, and so forth. For example, if one is interested in whether sex and body height have any effect on the choice of Ada as the favorite programming language, one could take a sample of programmers, divide them into male and female and into short, medium, and tall (where intervals of the exact body height

would have to be specified in order to define these categories). The contingency table would be as follows (the data are fictive):

characteristics		Ada	others	percentage
male	short	20	50	0.29
	medium	40	30	0.57
	tall	30	20	0.60
female	short	10	15	0.40
	medium	30	40	0.43
	tall	10	10	0.50

For this contingency table, statistical tests can be used to validate hypotheses, like "the sex of a programmer does not influence her/his choice of Ada as the favorite programming language" or "tall males prefer other languages". The idea of the contingency table can be adopted to the problem of establishing parameters of *Similarity Clustering*. The independent variables of this adoption are the discretizised similarity aspects. For the dependent variable, we count how often two elements with a certain combination of discretizised values of the dependent variables are in the same atomic component versus those in different atomic components. The percentage is the probability that two elements with a certain combination of similarity aspects belong to the same component and can be used for clustering, as in the example above in which the probability that two male and tall programmers both prefer Ada is 30/(30+20)=0.6.

Before the use of the contingency table is explained in detail, a few principal remarks follow. Given the atomic components for a sample of the system, we could look for the patterns involving two entities in the same component (**positive examples**) or in different components (**negative examples**). These patterns could be very detailed, like "if the two subprograms have five calls to common neighbors, share two types in their signature, but access no variable, then they belong to the same component". If the same pattern occurs more than once but does not always apply to entities in the same component, its validity can be determined as the number of positive examples divided by the number of occurrences. Hence, one can derive the patterns and their degree of confidence from the sample and use this information to try to find the other components in the rest of the whole system. However, if these patterns are too detailed, then similar, yet different

occurrences will be missed. Therefore, there should be some sort of abstraction, in other words, formation of equivalence classes. The similarity metric for two entities is such an abstraction; it abstracts from the exact numbers and kinds of relationships between two entities. However, the abstraction might go too far for at least two reasons:

- Since the similarity metric is the weighted sum over individual aspects of similarity, all similarity aspects should be sufficiently high when two entities are to be grouped. However, it could well be the case that two entities belong together only when one similarity aspect is high while another is low.
- At a lower level, as stated above, the similarity metric assumes that the influence of the edge types does not depend upon which aspect they occur in.

Both of these assumptions need not be valid. Clustering based on single, very detailed patterns as described above does not make these assumptions. However, it requires all needed possible patterns of the whole system to be in the sample. A first abstraction of the very detailed patterns, also without the assumptions of the similarity metric, could be to categorize the *LinkRatio*, $CommonRatio_{eq}$, $CommonRatio_{ne}$, and *DistinctRatio* for all patterns as introduced in Section 7.6.1.2. Recall that these ratios state the probability to find a certain edge type within a similarity aspect (*Link*, $Common_{eq}$, $Common_{ne}$, and *Distinct*) between entities in the same component. A pattern for two entities *A* and *B* can then be characterized by a vector:

$$(Link^{e1}(A,B), Common_{eq}{}^{e1}(A,B), Common_{ne}{}^{e1}(A,B), Distinct^{e1}(A,B),$$
$$Link^{e2}(A,B), Common_{eq}{}^{e2}(A,B), Common_{ne}{}^{e2}(A,B), Distinct^{e2}(A,B),$$
$$\ldots$$
$$Link^{eN}(A,B), Common_{eq}{}^{eN}(A,B), Common_{ne}{}^{eN}(A,B), Distinct^{eN}(A,B))$$

where *e1*, *e2*, …, *eN* are edge types. Then the likelihood for two entities to be in the same component, given a certain vector, can be ascertained as the relative frequency of positive examples for a vector of this kind. Since the similarity aspect ratios are floating numbers, near-misses should be avoided by forming equivalence intervals on the ranges of possible values. That is to say, a floating number is mapped onto a discreet range and all values in this range are handled alike. In other words, a multi-dimensional contingency table is established whose indices

are the discretizised ranges of the similarity aspect ratios and whose entries are as follows:

$$ct(\bar{v}) = \frac{|\text{positive\_examples}(\bar{v})|}{|\text{positive\_examples}(\bar{v})| + |\text{negative\_examples}(\bar{v})|} \quad \textbf{(7.28)}$$

where $\bar{v}$ is a discretizised vector of similarity aspects across all edge types. When the contingency table has been populated with the data drawn from the sample, it can be used to estimate the likelihood that two arbitrary entities of the rest of the system belong to the same component: Only the similarity aspects for the two entities are computed and then the likelihood is looked up in the table.

However, the order of this table is:

$$(\text{range(Direct)} \times \text{range}(\text{Common}_{eq}) \times \text{range}(\text{Common}_{ne}) \times \text{range(Distinct)})^{N}$$

where *N* is the number of edge types and *range* (*X*) is the number of intervals for the disretization of *X* (it is assumed for all edge types that a similarity aspect is discretisized into the same number of subintervals). The fact that a contingency table is very large is not the problem because it is also very sparse which allows an efficient table implementation; the problem is that the number of undefined entries, i.e., entries that neither have positive nor negative examples, increases by the order of the table. In such cases, the contingency table information is not available and it is unclear how to classify the two entities at hand.

A compromise between abstraction and subcategorization is to consider only the top-level similarity aspects, i.e., to classify according to the following kind of vector that does not differentiate among edge types:

(*Link (A,B), Common*$_{eq}$ *(A,B), Common*$_{ne}$ *(A,B), Distinct (A,B))*

The edge type ratios are relevant in so far that they are used to establish the weight of the edge type as in the more detailed model. But then, the edge types are summarized by the top-level similarity aspects and are not used for the index range of the contingency table.

7.6.2.3 Evaluation of the Training Methods

There are two factors by which an optimization approach will be judged:

- the number of iterations to calibrate the parameters on the sample
- the size of the sample needed to find reasonable parameters

On average, Gauß-Seidel optimization took nine iterations and simulated annealing eleven, whereas the approach based on a contingency table needs always one iteration.

Training. For the evaluation, increasingly large subsets of the reference components were used to calibrate the parameters of *Similarity Clustering* with the methods described above. The components used for training are called **training components**. The set of training components is called the **training set**. The set of all references, of which the training set is a subset, is called the **reference set**.

Sizes of the training sets in the range of 0.1 to 1.0 (step = 0.1) were use for calibration where the size of the training set is defined as the sum of the base entities that are part of training components divided by the total number of base entities that are part of the reference set.

The training components were randomly chosen from the reference set and were always complete in order to avoid false information. If a training component were not complete, its atomic component context used for the training may contain relationships erroneously classified as *between two entities of different components* just because one of the two partners of the relationship that are actually in the same original component does not belong to the training component. As a consequence, the following data are only valid for a usage model in which a maintainer identifies single components as complete as possible rather than beginning with several subsets of components in parallel. Furthermore, because the reference components were completely used, the exact share of the training subset may differ from the rated values $i \times 0.1$ ($i \in \{1, \ldots,10\}$).

Because we are interested in the question how big a training set should be to find appropriate parameters, each training for a new subset begins with the same default parameters rather than using the results of the previous training. In a real usage, one would use the parameters of the previous training and try to improve

these. Moreover, in order to get comparable results for the different sizes of subsets, the training components were always selected in the same order from the reference set and the random generator used by simulated annealing was reset for each training. Hence, all methods use the same training sets and each training with simulated annealing iterates over the same random values.

Threshold. *Similarity Clustering* — as a hierarchical clustering technique — returns a dendrogram of similar elements rather than a fixed set of candidates. In order to compare the results of *Similarity Clustering* with those of techniques that yield only sets of "flat" clusters, the dendrograms were flattened as described in Section 7.4. The retrieval of clusters from a dendrogram depends upon a threshold that specifies the minimal acceptable similarity for the flattened clusters. In order to investigate the influence of this threshold, four different thresholds were used.

A high threshold has the effect of retrieving smaller clusters. Whether it also decreases the number of clusters depends. For example, a subtree is split into two clusters when the similarity associated with the root of the subtree is below the threshold but the similarities associated with the children of the root are above the threshold. Then, two clusters are proposed while using a lower threshold would unite these two clusters. On the other hand, a very high threshold generally decreases the number of clusters because only few clusters will be similar enough.

A lower threshold yields larger clusters. Like for higher thresholds, we cannot predict whether a lower threshold also increases the number of clusters: More clusters with lower similarities may be accepted but also the size of acceptable subtrees of the dendrogram increases, hence, more clusters are united.

There is an obvious connection between the number of candidates and the number of false positives: Since the reference set has a fixed size, the more candidates are proposed the more false positives will be created and vice versa. However, as discussed, there is not necessarily a direct correlation between the threshold and the number of candidates and, thus, the number of false positives. Moreover, there is also no simple connection due to the filter used to exclude clusters with less than 3 and more than 75 elements (as it was done in the comparison in Chapter 6).

A low threshold may produce very large clusters that are then filtered out, and a high threshold may produce clusters too small to be proposed as candidates. Nevertheless, as the following figures for the thresholds between 0.1 and 0.4 show, the lower a threshold is, the lower is the number of false positives in general — despite of the fact that the curves in the following charts intersect in some cases. This correlation can be explained by the circumstance that a lower threshold will generally produce larger and, hence, fewer candidates.

In order to see whether the number of false positives increases linearly with the increase of the recall rate as the threshold decreases, a linear regression analysis was performed. The statistical analysis showed that in 7 out of 16 cases the number of false positives linearly increases with the recall rate at a significance level above 0.8. In the other cases, a linear connection could not be shown.

An interesting case of the influence of the threshold is the recall rate for Mosaic using *Gauß-Seidel* optimization shown in Figure 7-18. The thresholds 0.1 and 0.2 yield good results, but then, when the threshold is further increased, the number of candidates above the threshold immediately drops to 0. Fortunately, this phenomenon seems to be rare.

Results. The calibration results for the respective methods and kinds of atomic component are shown in Figures 7-16, 7-17, 7-18, 7-19, 7-20 and 7-21. The charts present the recall rate as defined by (6.5) on page 163 and the number of false positives with respect to the size of the subset used to calibrate the parameters. Each chart contains data for four different thresholds used to retrieve candidates from the tree produced by *Similarity Clustering*. For the *Contingency Table* approach, a different set of thresholds was used to do the charts because the threshold 0.4 did not yield any candidates.

In most cases, there is not a substantial difference between calibration according to *Gauß-Seidel* and *Simulated Annealing*. In few cases, the *Gauß-Seidel* technique yielded worse parameters (ADO detection for Bash; ADT detection for Bash and CVS with threshold 0.1). The *Contingency Table* approach is clearly worse than the other two approaches. Only for Mosaic, fewer false positives were generated at the same recall rate (ADO detection) or a recall rate slightly worse than the other approaches (ADT detection). An overall result of this evaluation is

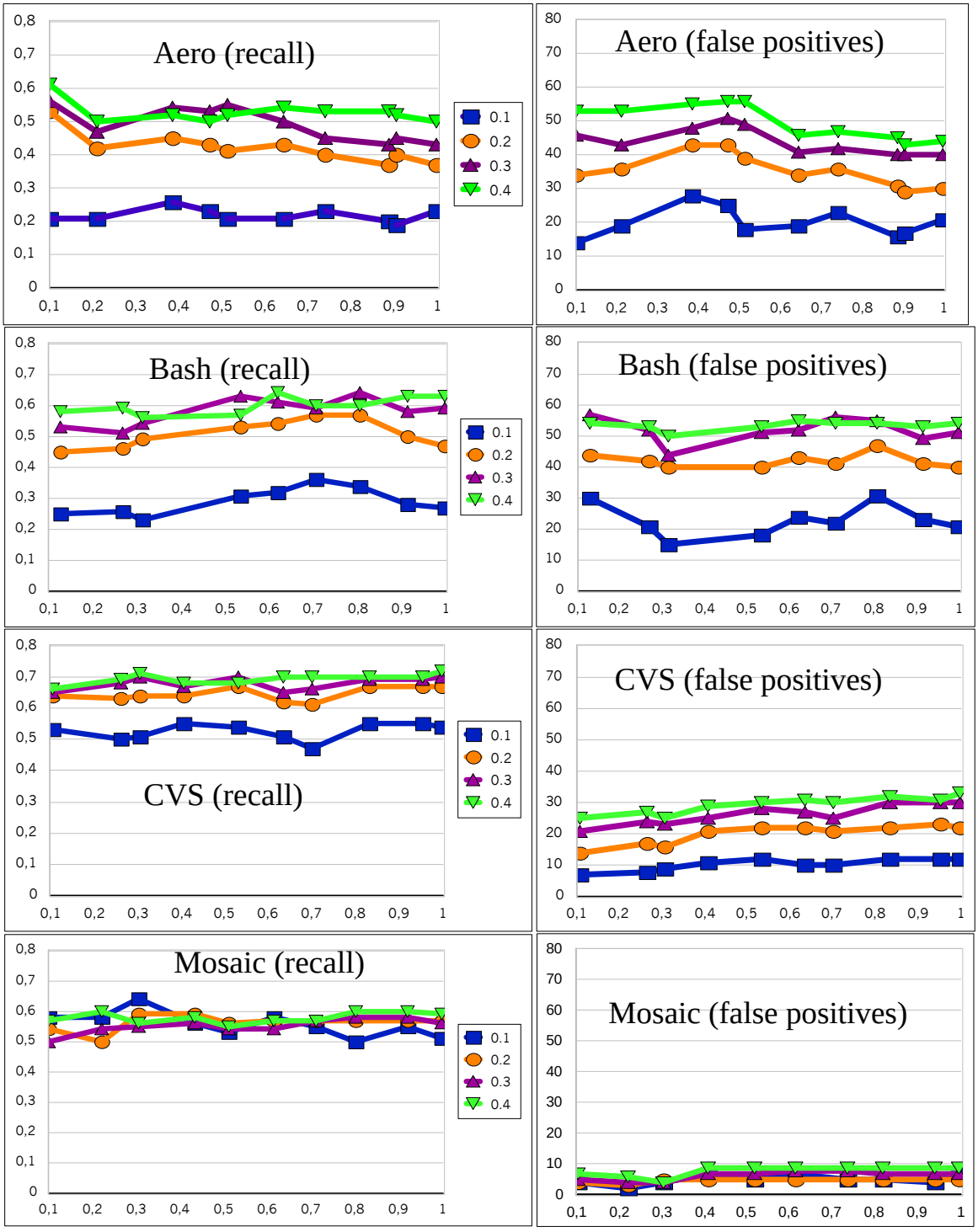


Figure 7-16. **ADO detection with *Simulated Annealing*.**

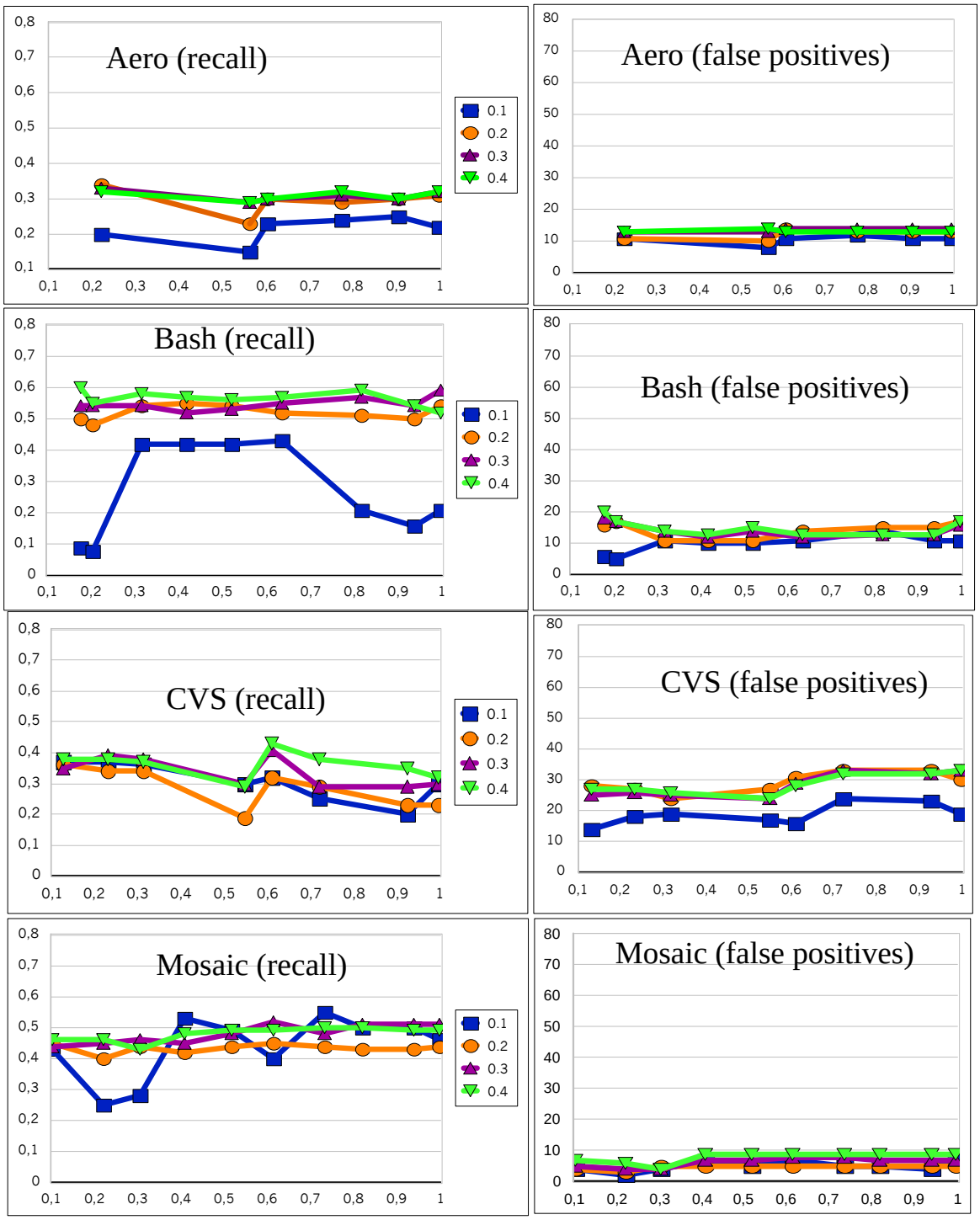


Figure 7-17. **ADT detection with *Simulated Annealing*.**

that the size of the subset does not have any discernible influence on the recall rate and the number of false positives: Sometimes they increase, sometimes they

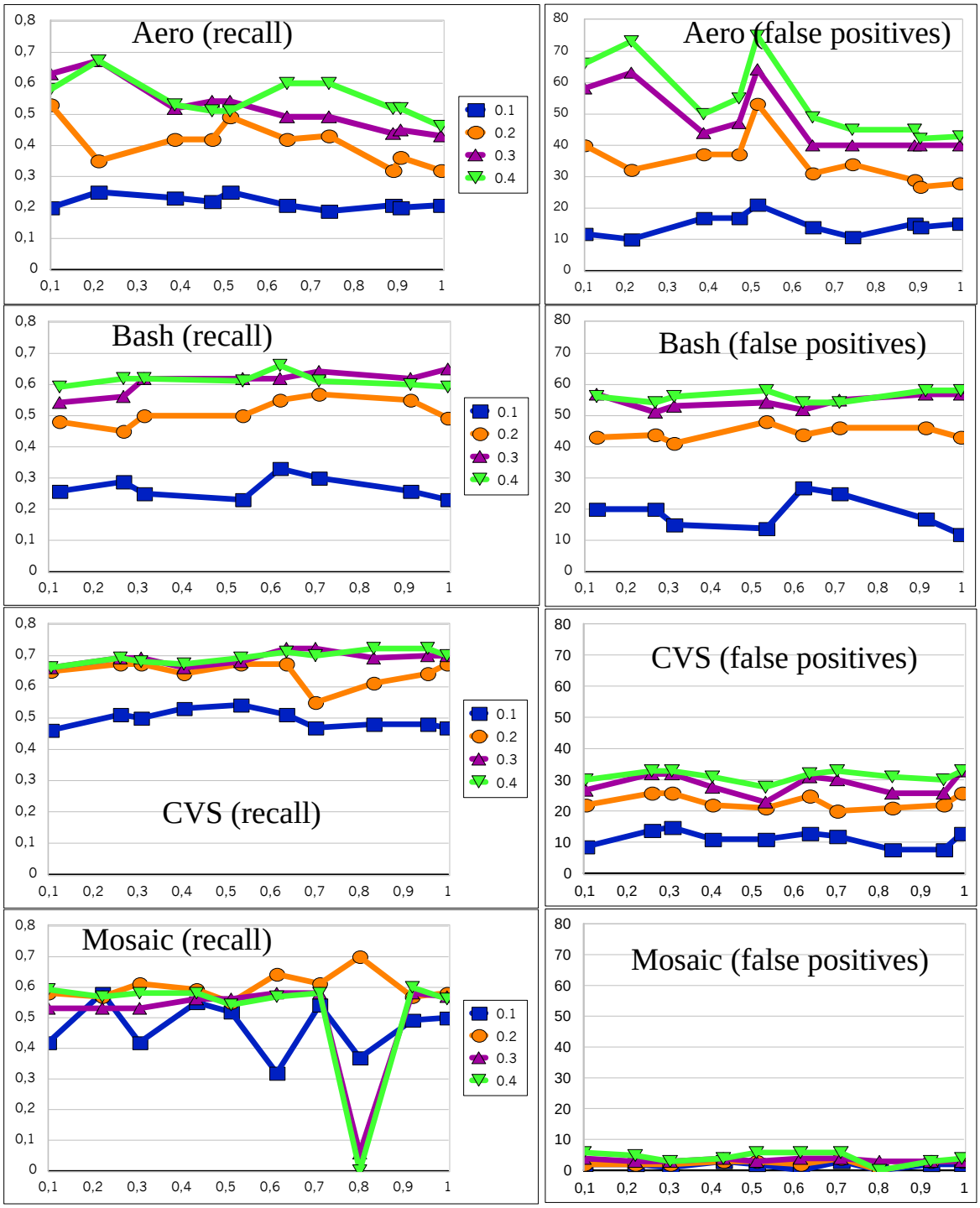


Figure 7-18. **ADO detection with *Gauß-Seidel*.**

decrease. Furthermore, even when the parameters are calibrated on all reference components of the system, the recall rate is far from 1.0. This may partly be due

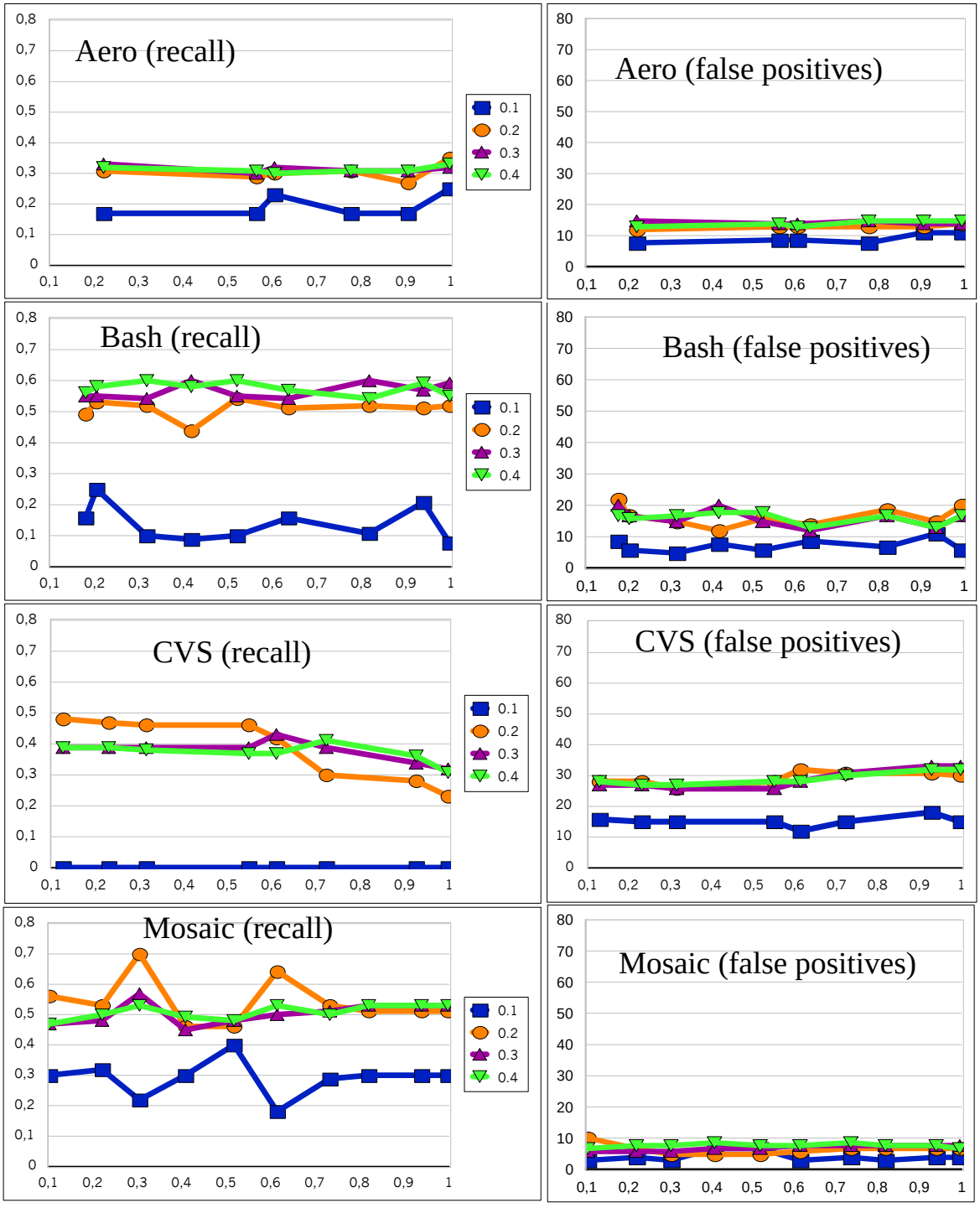


Figure 7-19. **ADT detection with *Gauß-Seidel*.**

to the few iterations of calibration conducted, but is certainly also because the *Similarity Clustering* metric is an abstraction of the actual patterns that is in many

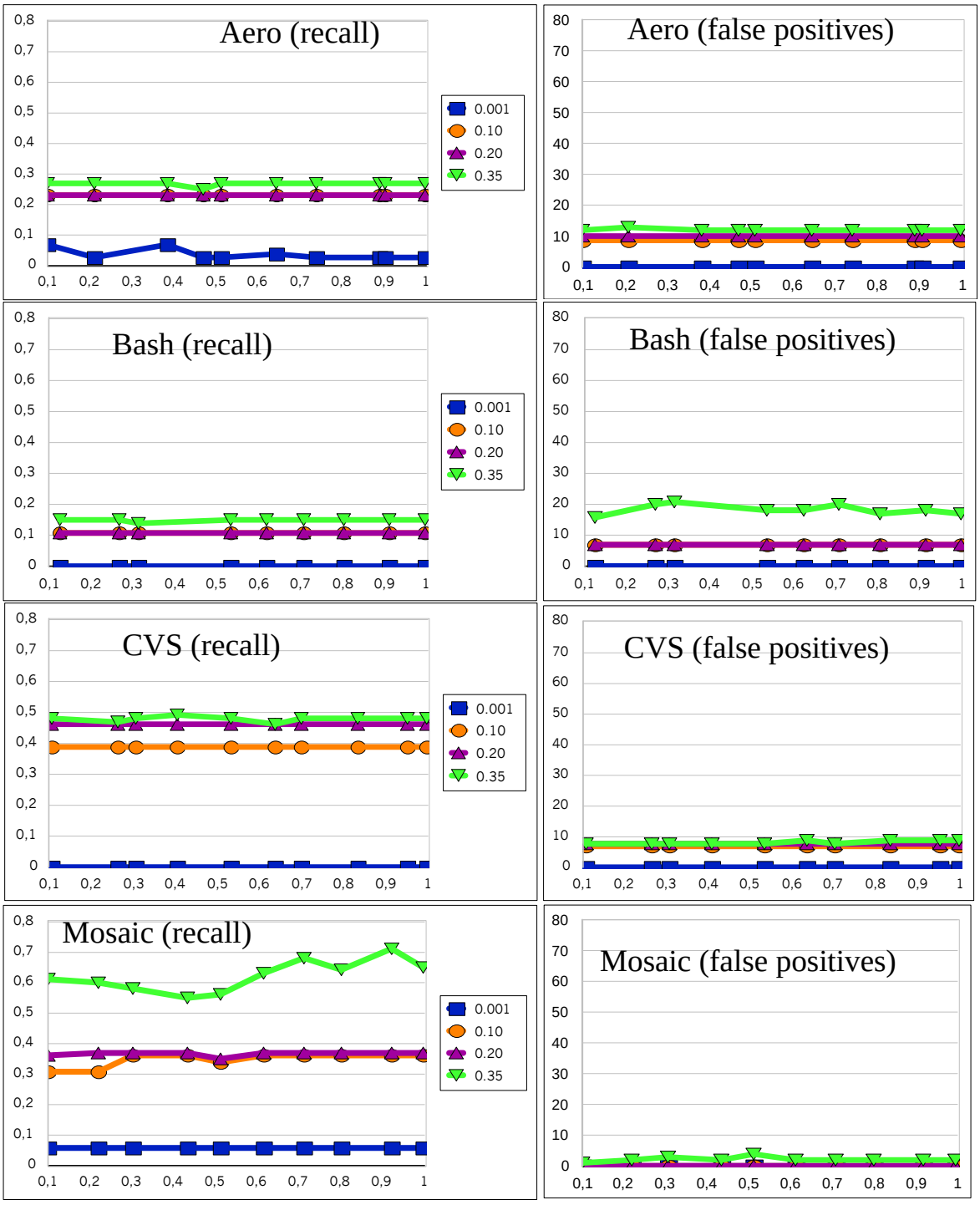


Figure 7-20. **ADO detection with *Contingency Table*.**

cases too coarse. Moreover, for the evaluation, the candidates are derived from the dendrogram using a threshold that is decisive for all subtrees of the dendro-

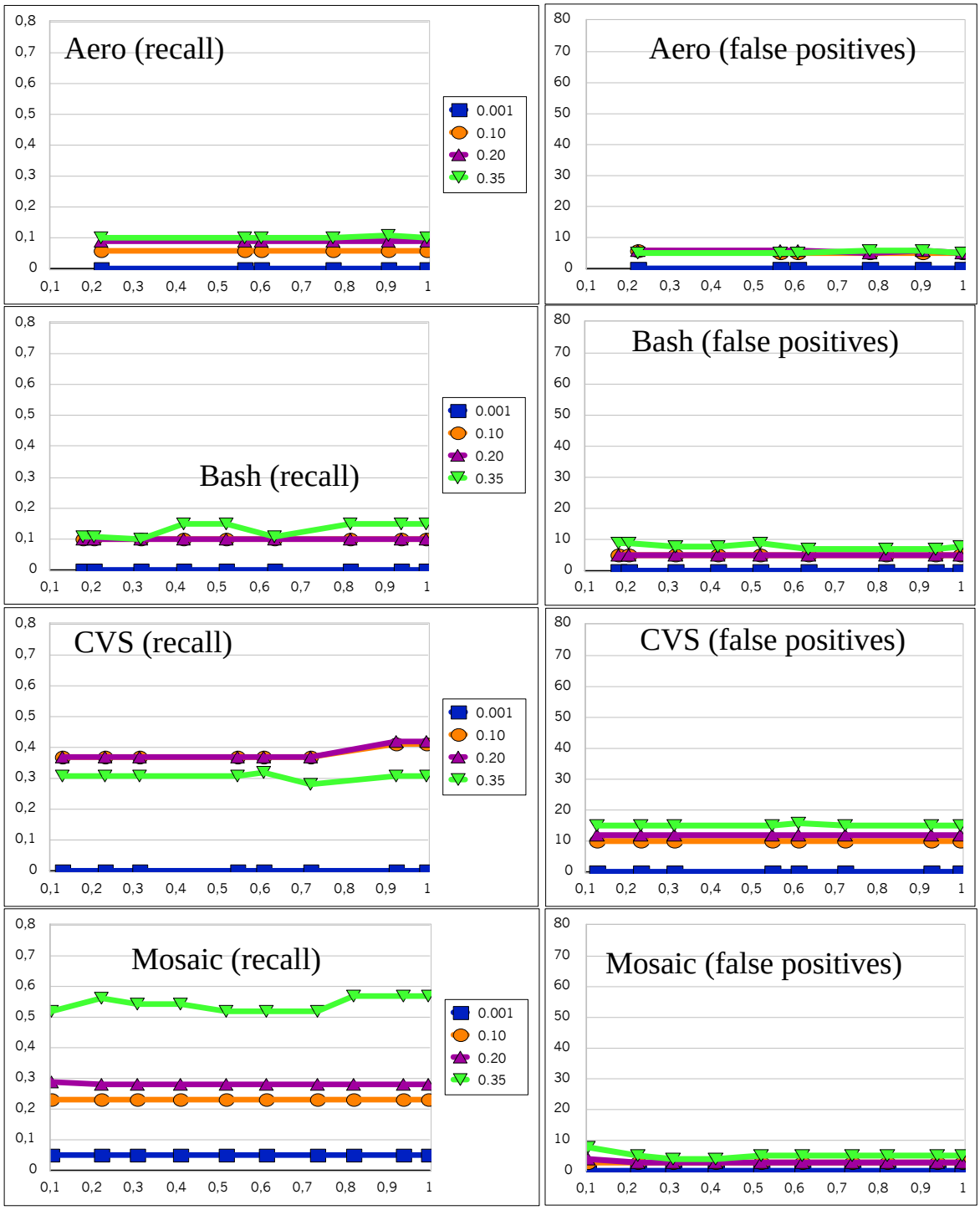


Figure 7-21. **ADT detection with *Contingency Table.***

gram. Hence, it is assumed that the average similarity among elements of components is comparable, which need not be the case; e.g., for one atomic component, a naming convention may be established such that informal information can be leveraged by *Similiarity Clustering*, while for other components, the names of their elements are not similar at all, and consequently, have a lower average similarity.

On the other hand, a positive result of this evaluation is that a subset size of about 20% seems to be sufficient to find suitable parameters.

7.6.2.4 Comparison to Other Techniques

Figure 7-22 and Figure 7-23 contain the recall rate of *Similarity Clustering* in comparison to the other approaches using simulated annealing as the training method and picking suitable thresholds that balance recall rate and number of false positives. Table 7-3 on page 238 contains the accuracies for the 1~1, 1~n, and n~1 matches, where the columns with heading "*th*." contain the minimal similarity thresholds used to retrieve components from the dendrogram. These thresholds were chosen to balance the recall rate and the number of false positives. Table 7-4 on page 238 lists the number of false positives for *Similarity Clustering* and Schwanke's *Arch* approach. The table demonstrates that the number of false positives for *Similarity Clustering* could actually be reduced with respect to its predecessor *Arch*.

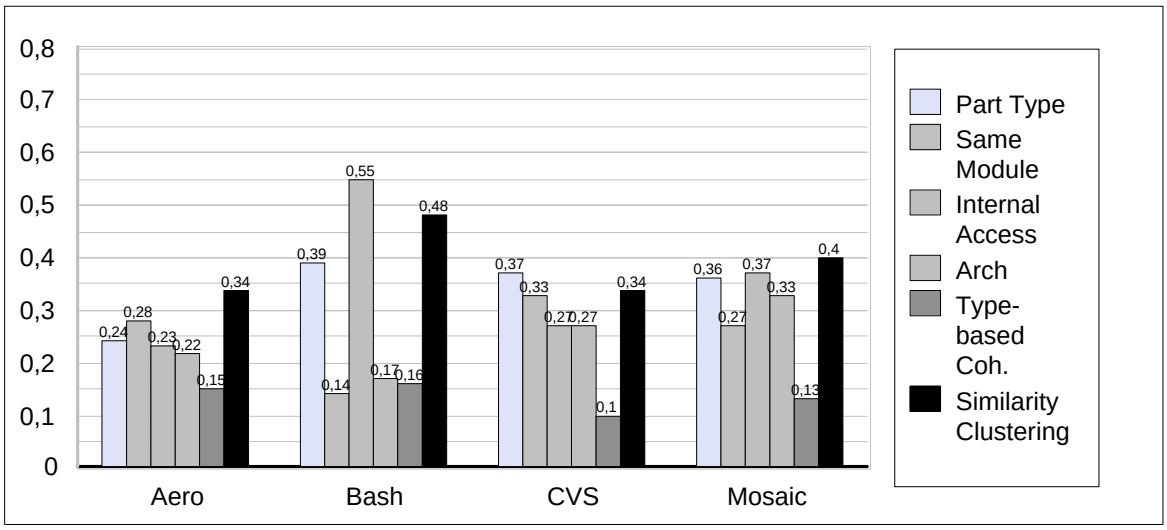


Figure 7-22. **ADT recall rates.**

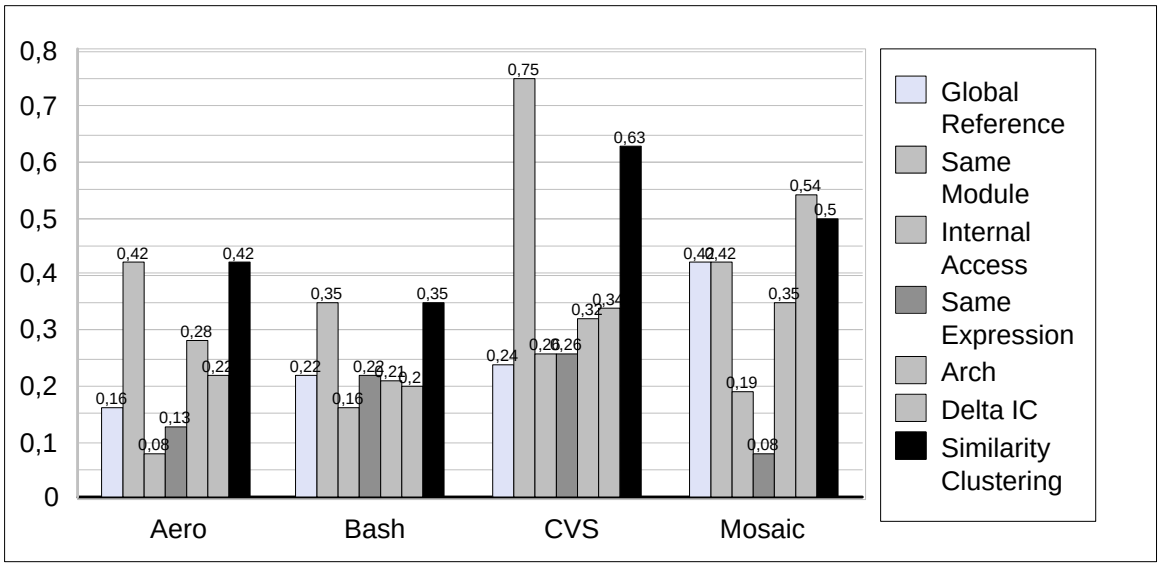


Figure 7-23. **ADO recall rates.**

Table 7-3. **Detected ADTs and ADOs for *Similarity Clustering*.**

System	ADT							ADO						
	Good			OK				Good			OK			
				too large		too detailed					too large		too detailed	
	th.	#	acc.	#	acc	#	acc	th.	#	acc.	#	acc	#	acc
Aero	0.20	3	0.84	2	0.44	0	0.00	0.20	5	0.81	4	0.47	0	0.00
Bash	0.20	4	0.79	8	0.51	0	0.00	0.13	1	1,00	0	0.00	12	0.41
CVS	0.10	5	0.85	3	0.32	4	0.30	0.20	22	0.82	13	0.41	6	0.51
Mosaic	0.40	1	0.86	8	0.53	5	0.41	0.40	13	0.86	12	0.49	4	0.39

Table 7-4. **Number of false positives and true negatives.**

Technique		Aero		Bash		CVS		Mosaic	
		false posi-tives	true nega-tives	false posi-tives	true nega-tives	false posi-tives	true nega-tives	false posi-tives	true nega-tives
Similarity Clustering	ADT	11	5	17	3	27	7	3	4
	ADO	36	5	32	6	17	1	2	8
Arch	ADT	13	4	29	13	36	9	11	10
	ADO	38	6	46	11	42	17	11	15

As Figure 7-22 and Figure 7-23 show, *Similarity Clustering* is always among the techniques with the highest recall rates. However, as Table 7-4 indicates, it has also a higher number of false positives than most approaches. On the other hand, in comparison to its ancestor, the *Arch* approach, the number of false positives was substantially reduced.

Similarity Clustering also needs more computational effort than any of the other approaches. This was the reason why only few training iterations could be done. The next section discusses how time and space complexity can be reduced.

7.7 Implementation Hints

The basic outline of the clustering algorithm was already given in Figure 7-2 on page 187. Because the algorithm suggests to compare each entity with any other entity, the space complexity is $O(n^2)$ where n is the number of entities and the time complexity to compute Sim_{direct} and $Sim_{informal}$ is $O(n^2)$. In the case of $Sim_{indirect}$, the time complexity even seems to be $O(n^3)$ because for each pair all neighbors need to be ascertained and the number of neighbors can be n in the worst case. However, analyzing the similarity metric in more detail shows that at least the computation of Sim_{direct} and $Sim_{indirect}$, which requires the most time, is linear. Computing $Sim_{informal}$ is comparatively cheap. Furthermore, the storage needed for saving the similarity between entities, which is useful because the similarity is computationally expensive and needed more than once, can also be reduced leveraging the fact that most similarities are 0.

In this section, we will refine the algorithm and give some hints on possible optimizations. First, some data structures needed to implement the approach efficiently are described. Then it is shown how Sim_{direct} and $Sim_{indirect}$ can be computed in linear time to initialize the matrix used to store the similarities among individual entities. Finally, the algorithm will be refined and possible optimization to reduce the space needed for the matrix will be discussed.

7.7.1 Used Data Structures

Let N be the set of nodes $\{n_1, n_2, \ldots, n_m\}$ to be clustered. We assume that the nodes in N are enumerated and can be identified by a unique number.

To represent the disjoint clusters (or groups) of base entities, we can use the union-find data structure and algorithms for a partition[1] $S = s_1,\ldots,s_m$ of N by Hopcraft and Ullman (1983) that was already introduced in Section 5.2.

In each step of the clustering algorithm, the pair of clusters with maximal similarity has to be found. Instead of iterating over the whole matrix each time to find this maximum, we will use a priority queue to record the pairs with descending similarity. For the priority queue, the following subprograms are assumed (let p be a pair of set identifiers):

- procedure *insert (p, s)* inserts a pair p of set identifiers into the priority queue with similarity s; if this pair is already present in the queue, the present pair is removed and the new pair is added; only those pairs will actually be added to the queue whose similarity is above the minimal threshold Θ
- function *empty_queue* is true if the priority queue is empty
- function *head* returns the pair of set identifiers with the highest similarity
- procedure *remove_head* removes the pair of set identifiers with the highest similarity

7.7.2 Initialization

Initially, each node of N is put in a set of its own using *new_set*, hence we start with a partition $S=s_1,\ldots,s_m$ where set identifier $s_i = \{i\}$ for all i in $1\ldots m$. Because the similarity relation is computationally expensive and needed more than once, it is computed only once and stored in a matrix sim_N with range $1..m$ where $sim_N(i,j)$ denotes the similarity between i and j according to equation (7.3). In the

1. A partition S of a set $N = \{n_1, n_2, \ldots, n_m\}$ is a set of sets s_i where $i = 1\ldots k$ and $N = \bigcup_{i = 1\ldots k} s_i$ and $\forall(i, j \in \{1\ldots k\}, i \neq j)\ s_i \cap s_j = \varnothing$

following, the capitalized *Sim* denotes the similarity metric, whereas *sim* denotes a similarity matrix.

For the group similarity described in Section 7.2, we can use another matrix sim_S with range $s_1,\ldots,s_m$ where $sim_S\ (s_i, s_j)$ denotes the similarity between s_i and s_j. This matrix is initialized with:

$$sim_S\ (find\ (i), find\ (j)) = sim_N(i, j)$$

Since s_i initially consists only of i, sim_S equals sim_N in the beginning. However, during clustering the values in sim_S change with the following invariant:

> $sim_S\ (s_i, s_j)$ denotes the average similarity between all nodes in s_i and s_j according to equation (7.2) if $s_i \neq s_j$, and is 0 if $s_i = s_j$.

The values in sim_N will not be changed.

Because the similarity metric is symmetric, the similarity relation can be stored in a triangular matrix. Therefore, the two similarity matrices sim_S and sim_N can be stored in one single physical matrix, one in the upper, one in the lower part. Nevertheless, the space complexity for the similarity relations is still $O(m^2)$ where m is the number of nodes to be clustered because we compare any node with any other node. The time complexity for computing the cells of this matrix is $O(m^2 \times k)$ where k is the maximal number of neighboring nodes (in $Sim_{indirect}$ all neighbors of a node pair need to be ascertained). Consequently, the time complexity is $O\ (m^3)$ in the worst case. However, statistical analyses on the number of neighbors per nodes showed that k is a very low constant much smaller than m and neither increases with the system size.

By a closer look at the way similarity between entities is defined, one can see that $Sim_{\text{direct}}(n_i, n_j)=0$ (equation (7.18)) for nodes that are not directly connected, and $Sim_{\text{indirect}}(n_i, n_j)=0$ (equation (7.4)) for nodes that do not have a common neighbor. Let nodes that are direct neighbors of a node n be the **first-degree neighbors** of n and let the nodes that are not directly reachable, but only one node away from n be the **second-degree neighbors** of n, then $Sim_{direct}\ (n, n_j)$ has only to be com-

puted for $n_j \in$ *first-degree neighbors (n)* and $Sim_{indirect}(n, n_j)$ has only to be computed for $n_j \in$ *first-degree neighbors (n)* $\cup$ *second-degree neighbors (n)*. Since there is only a small number of first and second-degree neighbors for each node in practice, Sim_{direct} and $Sim_{indirect}$ is basically computed linearly to the number of nodes, i.e., has time complexity O(m). Because these are the computationally most expensive parts of the similarity between nodes, one can reduce the time needed to compute the similarity matrix immensely. However, computing Sim_{direct} and $Sim_{indirect}$ is only one part of the matrix initialization. In the case of informal information, each entity is compared to each other entity no matter whether the entities are neighbors or not. Therefore, the overall time complexity remains O(m^2) unless one does not consider informal information.

Below, we will also discuss optimizations to reduce the space needed to store the similarity relation leveraging that the similarity relation is usually greater than 0 for only a few pairs of nodes, i.e., the matrix is very sparse.

7.7.3 A Refined Clustering Algorithm

The outline of the clustering algorithm in Figure 7-2 on page 187 can be refined to the one in Figure 7-4. In this refined algorithm, it is still open how the pair with the maximal similarity can be found efficiently.

Instead of iterating over the matrix sim_S each time it has changed and thus ending up with a cubic algorithm, one can use the priority queue introduced in Section 7.7.1 to keep track of the order of similarity. Using the priority queue, one can rewrite the algorithm in Figure 7-4 as in Figure 7-5. Initially, the queue is filled with all pairs in the similarity matrix.

The procedure *recompute* in this algorithm recomputes the similarities of the newly united clusters to all other clusters. During this recomputation, all recomputed pairs will be added to the priority queue; all obsolete similarity values for the recomputed pairs are removed thereby.

Equation (7.2) reduces the similarity of two clusters to the average similarity between their members. Its computation becomes increasingly expensive as the

Input:

- a partition S
- similarity matrix sim_S for the elements in the partition S
- a minimal threshold Θ for the union of clusters

Output:

- a new partition for S
- a tree that describes the hierarchical clustering for S

while $\exists(s_i, s_j)$: $sim_S(s_i, s_j) > \Theta$ **loop**
  let $p = (s_i, s_j)$ **where** $sim_S(s_i, s_j) > \Theta$ is maximal;
  union (s_i, s_j);
  add a subtree with children s_i and s_j to the clustering tree
  recompute (s_i, s_j);
end loop;

Algorithm 7-4. **Refined clustering algorithm.**

while $\neg$ empty_queue **loop**
  let (s_i, s_j) = head;
  remove_head;
  union (s_i, s_j);
  add a subtree with children s_i and s_j to the clustering tree
  recompute (s_i, s_j);
end loop;

Algorithm 7-5. **Optimized refined clustering algorithm.**

involved clusters grow. An equivalent, but less costly approach is to compute the new similarity for the union by the similarity of the two clusters to be united using the following equation:

$$Sim(A \cup B, C) = \frac{|A| \cdot Sim(A, C) + |B| \cdot Sim(B, C)}{|A| + |B|} \quad \textbf{(7.29)}$$

The equivalence of (7.29) and (7.2) can be proven by induction.

Input:

- a pair of set identifiers (s_i, s_j) whose similarities to all other clusters have to be recomputed

Algorithm:

for all s_k **in** S **where** $k \neq i \wedge k \neq j$ **loop**
  sim_S (find (s_i), s_k):= Sim (s_i, s_k) using equation (7.2) on page 188;
  -- *find (s_i) denotes the union of s_i and s_j*
  insert ((s_i, s_j), sim_S (s_i, s_j));
end loop;

Algorithm 7-6. **Algorithm recompute.**

Begin of induction. The equivalence of (7.29) and (7.2) on page 188 is obviously true when *A*, *B*, and *C* are all disjoint sets with only one element:

$$\begin{aligned} GSim(A \cup B, C) &= GSim(\{a, b\}, \{c\}) \\ &= \frac{Sim(a, c) + Sim(b, c)}{|\{a, b\}| \times |\{c\}|} \\ &= \frac{|\{a\}| \cdot Sim(a, c) + |\{b\}| \cdot Sim(b, c)}{|\{a\}| + |\{b\}|} \\ &= Sim(A \cup B, C) \end{aligned}$$

Assumption of induction. To see that this equivalence holds even for disjoint sets with more than one element, let us assume that:

$$Sim(A, C) = GSim(A, C) = \frac{\sum_{a \in A, c \in C} Sim(a, c)}{|A| \cdot |C|} \quad \textbf{(7.30)}$$

$$Sim(B, C) = GSim(B, C) = \frac{\sum_{b \in B, c \in C} Sim(b, c)}{|B| \cdot |C|} \quad \textbf{(7.31)}$$

Step of induction. Inserting equations (7.30) and (7.31) into (7.29) leads to (note that *A* and *B* are always disjoint):

$$Sim(A \cup B, C) = \frac{|A| \cdot \frac{\sum_{a \in A, c \in C} Sim(a, c)}{|A| \cdot |C|} + |B| \cdot \frac{\sum_{b \in B, c \in C} Sim(b, c)}{|B| \cdot |C|}}{|A| + |B|}$$

$$= \frac{\sum_{a \in A, c \in C} Sim(a, c) + \sum_{b \in B, c \in C} Sim(b, c)}{(|A| + |B|) \cdot |C|}$$

$$= \frac{\sum_{u \in U, c \in C} Sim(u, c)}{|U| \cdot |C|} \quad \text{where } U = A \cup B \wedge A \cap B = \emptyset$$

$$= GSim(A \cup B, C)$$

Conclusion of induction. Hence, equation (7.29) indeed computes (7.2) when (7.30) and (7.31) hold. By induction, we conclude that (7.29) computes (7.2). □

The advantage of using (7.29) instead of (7.2) is not only that it is easier to compute, but also that there is no need to keep the original values for the similarities of entities sim_N that were necessary to compute (7.2). Thus, we can save the space for similarity matrix sim_N.

7.7.4 Matrix Representation

Because the similarity between nodes as defined by equation (7.3) is greater than 0 only for a few pairs of nodes (actually, as already discussed, if informal information is not considered, it is definitely 0 for all nodes that do not have a common neighbor and are not directly linked), the matrix to store the similarity relation is very sparse. Instead of allocating a quadratic matrix, it is more reasonable to represent the matrix as linked nodes as outlined in Figure 7-24. Each cell of the matrix whose value is greater than zero is part of two lists: one list for its row and one for its column. A node in this data structure has the following components:

- a real number for the *value* of this cell
- a *column pointer* to the next cell in this column
- a *row pointer* to the next cell in this row

- a *column* and a *row index* of this cell in the original matrix; this is necessary to identify the cell during row or column traversal

The matrix as a whole is then represented by a column and a row vector whose fields point to the first node in the column or row, respectively.

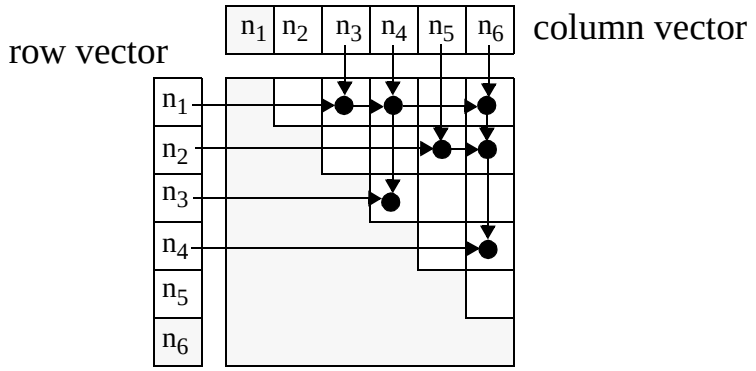


Figure 7-24. **Sparse matrix representation.**

One could argue that the costs of accessing cell values is a high price that we have to pay for this sparse representation. This were true if we randomly accessed the matrix. Fortunately, this is not the case. Only when we merge two clusters, cells have to be accessed to recompute the similarity for the union of these clusters to all the remaining clusters. The last section proposed to use (7.29) on page 243 to recompute the similarity relation for $Sim(A \cup B, C_i)$, which requires to recompute the shaded parts in the matrix in Figure 7-25(a) by reading the values in $sim(A, C_i)$ and $sim(B, C_i)$ for all $C_i \neq$ A, B. After the merge of *A* and *B*, one of them ceases to exist while the other represents the union of *A* and *B* from now on. Let us assume that the columns and rows of *A* are used to store the values of $Sim(A \cup B, C_i)$. Figure 7-25(a) graphically pictures three recomputations of the similarity of $A \cup B$ to C_1, C_3, and C_4 where a complete matrix is assumed. The recomputation is along the columns for *A* and *B* and combines the two values at the corresponding row index for the respective C_i. If a complete matrix is assumed, we would also have to re-compute the rows of *A* and *B* accordingly. However, since it is de facto a triangular matrix without the diagonal, the recomputation actually advances as illustrated in Figure 7-25(b): When the cell for *sim (X,Y)* does not exist, the value of *sim (Y,X)* is used instead.

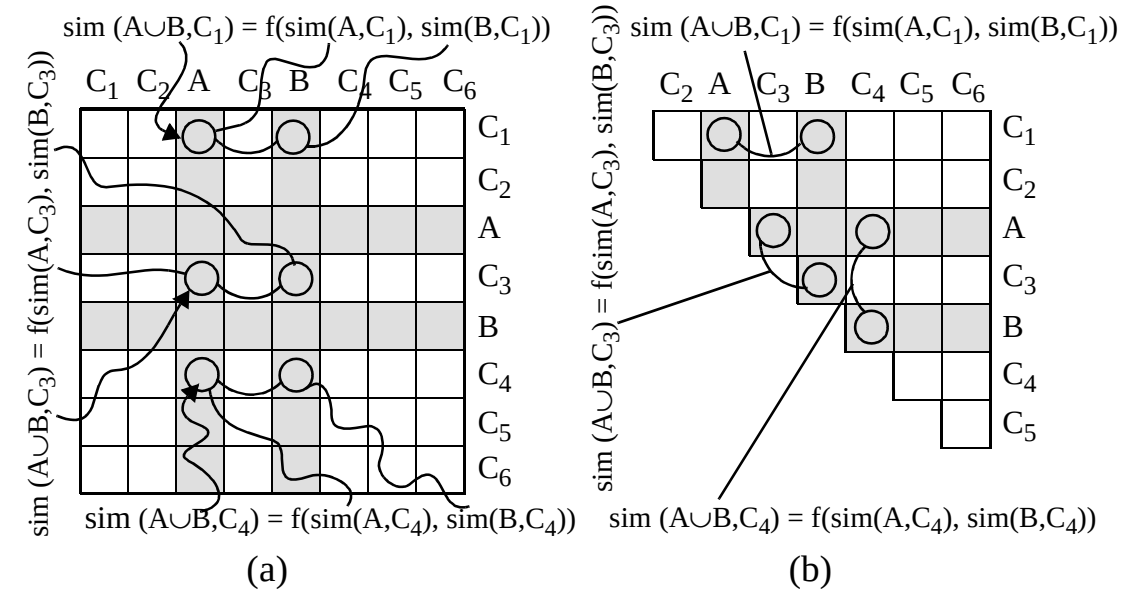


Figure 7-25. **Access pattern for triangular similarity matrix.**

Thus, there is a regular access pattern to the cell contents of a triangular similarity matrix by iterating simultaneously over the rows and columns of *A* and *B* as shown in Figure 7-26(a) that allows an implementation of the rows and columns as linked lists.

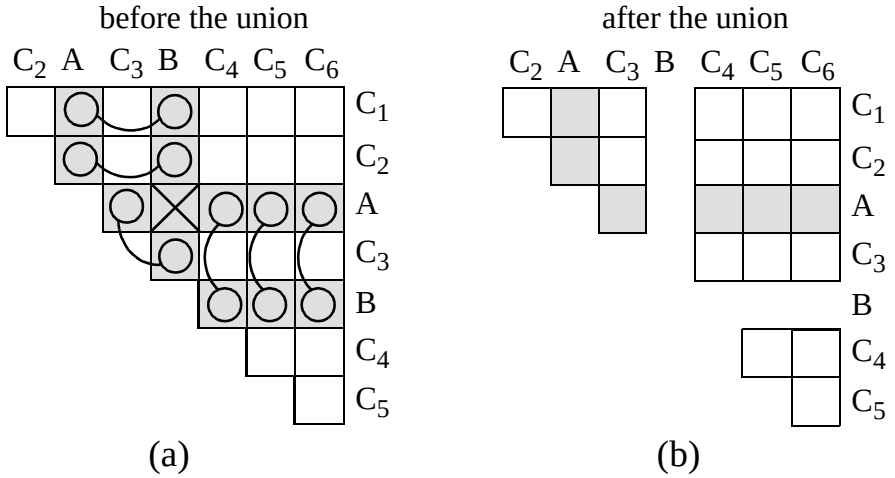


Figure 7-26. **Similarity matrix before and after recomputation due to union.**

We can arbitrarily choose one of *A* and *B* to store the new similarity values for the union of *A* and *B*; rows and columns of the other cluster of the union can be

released. In the example of Figure 7-26(b), we have chosen *A* as the new representative for the union and rows and columns of *B* are removed. This will speed up future traversals.

In the data structure we have chosen for the matrix, each cell node is member in two lists: one for the column, one for the row. Since these lists are only linked in one direction, the nodes cannot be easily removed from the list. We reject doubly linked lists due to the space overhead. The easiest solution to this problem is just to the set the cell value to 0.0 and to remove cells with a 0.0 content during future traversals on the fly.

7.7.5 Implementing the Priority Queue

The priority queue can be implemented as an integral part of the similarity matrix by a list of nodes in the matrix as illustrated by Figure 7-27. The head of the list represents the cell with the highest value.

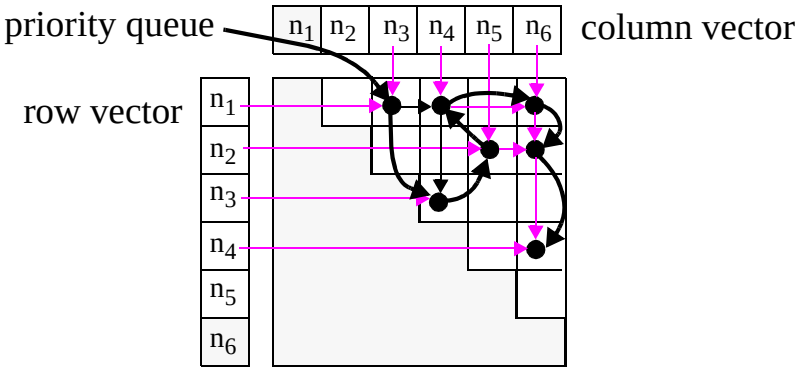


Figure 7-27. **Sparse matrix representation combined with priority queue.**

When two clusters are united, one can sort the newly computed nodes and then traverse the priority queue replacing the changed nodes by their newly computed values from the sorted list of nodes. The traversal of the priority list stops when all newly computed nodes are at their appropriate place. This way only one (partial) traversal of the priority queue is necessary.

7.8 Differences from Previous Approaches

Similarity Clustering differs from the other base techniques described in Chapter 5 in that it generates candidates using a metric which provides a more refined view on the relations between program entities. The previous approaches compute connected components or other fixed patterns in a graph. In contrast, whether these patterns are present or not, *Similarity Clustering* deals with continuous values which reflect the degrees of similarity.

Schwanke defines a similarity metric used to group functions into modules (1991, 1994). Our new similarity clustering approach exploits these nuances, but extends the metric in the directions summarized in Table 7-5. Further refinements that took place after Jean-François Girard, Georg Schied, and I jointly developed and published these extensions in 1997 are described in column “My Additions”.

7.9 Summary

Similarity Clustering is the most general approach described in this thesis. It can detect abstract data types, abstract data objects, hybrid components, as well as groups of related routines. All connection-based techniques described in Chapter 5 can be subsumed under *Similarity Clustering*. *Similarity Clustering* goes beyond other approaches in that it also considers relations to common third entities and informal aspects.

Similarity Clustering can be used in two different modes: Search for specific user-defined patterns and search for similar patterns of already found atomic components. Whereas connection-based techniques always yield the same candidates, *Similarity Clustering* can be adjusted by the maintainer to different search patterns by changing its edge weights. The adjustable parameters of *Similarity Clustering* offer more flexibility. On the other hand, when the maintainer wants to search for atomic components similar to those already found, these parameters can be automatically calibrated by the set of known components using traditional optimization techniques, such as simulated annealing or Gauß-Seidel optimization. The sample used to calibrate *Similarity Clustering* can be ascertained with other techniques or with *Similarity Clustering* after specific user-defined adjust-

Table 7-5. **Differences from previous approaches.**

	Schwanke's Approach	**Our Approach 1997**	**My Additions**
clustering method	hierarchical clustering algorithm	non-hierarchical clustering algorithm	hierarchical clustering algorithm
domain	routines	routines, user-defined types, global variables	
weights	Shannon info. for non-local names	Shannon info. for partners (globally established) edge weights reflecting the modality	refined Shannon info. (established considering modality and partner)
features	usage of non-local names including macros	type relationships reference relationships call relationships no macros	roles references to record components same-expression
indirect links	common and distinct	common and distinct	common separated into $Common_{eq}$ and $Common_{ne}$
direct links	values = 0 or 1	continuous values between 0 and 1	continuous values ≥ 0
informal information	not used	tokens in identifiers pre- and postfix in identifiers organization of files	
similarity between groups	maximum similarity between elements (1991) k-nearest neighbors (1994)	average similarity between elements	
establishment of feature weights	neural networks	systematic hand tuning (grid search)	contingency table Gauß-Seidel approach simulated annealing

ments have been applied. Experiments with calibration methods for the subject systems used in this thesis indicate that a sample of 20-30% of the groupable base entities of a system grouped to atomic components is a sufficient training set (a base entity is said to be *groupable* if it actually belongs to an atomic component; recall that not all base entities were grouped to components by the software engineers for our subject systems). However, the data have not shown that one could improve the recall rate of *Similarity Clustering* by using larger samples. This is probably because of the diversity of characteristics among atomic components. For example, some atomic components may be properly encapsulated such that high weights for record components and variables references will yield good results. Some others may be permissive atomic components for which higher weights for record components and variables references will also add many non-accessor functions that break the information hiding principle.

Similarity Clustering is one of the most effective techniques as far as the recall rate is concerned. On the other hand, it has also more false positives than other approaches (except for *Arch* which has more false positives). In earlier variants, the number of false positives was even worse (Girard et al. 1997c). My additions described in Table 7-5 in the last section resulted in a substantial reduction of the false positives in comparison to results previously published.

Another advantage of *Similarity Clustering*, as a hierarchical clustering method, is that it yields a dendrogram of clustered entities instead of a set of flat candidates. This is in particular useful for validation. In the quantitative evaluation of *Similarity Clustering*, branches of the dendrogram were cut and converted into candidates using the same similarity threshold. However, this assumes that the same threshold is suitable for all components. Using a single threshold was necessary in a fair comparison to other automatic techniques; in an interactive approach, one does not need a threshold at all. Hence, less false positives can be expected for a hierarchical view.

There are also some drawbacks of *Similarity Clustering*. For all techniques other than *Similarity Clustering*, there is one single criterion used for clustering. Hence, the reason why a technique has grouped entities together is obvious. This is less obvious for *Similarity Clustering* when the similarity metric considers several

aspects at the same time. This complicates validation of candidates proposed by *Similarity Clustering*.

When *Similarity Clustering* considers informal information, it may happen that entities are clustered that are not even transitively connected to each other just because they have similar names. This may be useful when groups of related subprograms are to be detected. However, for ADT and ADO detection, the entities are always at least transitively connected via call, type, or reference relationships. Fortunately, unconnected entities can be easily filtered from candidates if this is necessary. In the quantitative comparison reported in this chapter, a filter for candidates with unconnected entities was not used. Using such a filter will probably lead to less false positives. Furthermore, future extensions of *Similarity Clustering* should try to use informal information only for nodes that are either first or second-degree neighbors. Then, informal information would only be an additional hint but not a sufficient criterion for two nodes to be in the same component. This would also reduce the time complexity for computing the similarity matrix as discussed next.

The computational effort needed for *Similarity Clustering* is higher than for all other techniques. This is mainly due to establishing the similarities among the entities while clustering as such is comparatively fast. Section 7.7 gave hints on how the complexity can be reduced. It turned out that time and space complexity for *Similarity Clustering* is basically linear to the number of entities, n, when informal information is excluded (assuming an upper constant limit of neighbors an entity can have). However, when informal information is used, each entity has to be compared to any other entity resulting in a time complexity of $O(n^2)$. If the proposal above to use informal information only for first and second-degree neighbors is put into action, however, the complexity can be reduced to $O(n)$.

Part III The Semi-Automatic Method

Chapter 8 *Combined and Incremental Techniques*

In the previous chapters, the basic techniques and their evaluation were described. The evaluation has shown that none of the basic techniques has the detection quality that compares to human judgement. Therefore, further improvements are necessary. Advances can be expected by combinations of the basic techniques. Another avenue to progress in atomic component detection is to integrate the maintainer in the detection process. Furthermore, new techniques could be invented or existing techniques be refined by means of control and data flow analyses, for example. However, before new techniques are tackled, possible improvements by combinations of existing techniques should be explored first. Moreover, the maintainer has to validate the candidates proposed by automatic techniques at any rate because we can rarely expect perfect recall of any automatic technique due to the vague criteria and sometimes subjectivity of the rules for constituting atomic components. That is, integrating the maintainer is needed in any case.

For these reasons, this thesis presents only generic ways of combining existing techniques and leaves new techniques to further research. The combinations are designed with applicability to a semi-automatic detection process with human intervention in mind. Chapter 9 proposes a possible semi-automatic method that uses the extensions discussed in this chapter.

8.1 Ways of Combinations

There are basically two ways of combining the heuristics described in Chapter 5: They can either technically be integrated such that the underlying heuristics of several basic techniques are implemented by one combining analysis — or the results of the techniques and not the techniques as such are combined. The latter strategy has the advantage of more flexibility in both practical operation and implementation: The user can decide on his own on the selected heuristics as well as the order of their application and the developer of new analyses can write the analyses independently; otherwise, adding a new analysis to an existing suite of *N* analyses would require to develop *N* combinations of the new analysis with the existing ones if pairs of combinations are considered. If more than two techniques are to be combined, the implementation effort is even worse.

That is why I propose to combine the results of the techniques instead of the techniques themselves. Manifold combinations are possible by means of a few combining operators such as union, intersection, and difference operators for components views. Chapter 9 describes a semi-automatic method in which the maintainer uses these operators to combine and tailor the basic methods for his own use. The operators are simple enough that the maintainer can compose them by simple mouse clicks.

One of the operators, namely, the composition operator, applies two techniques successively. The output of the first technique is fed into the second technique. As a consequence, the second technique has as input a description of the system that does not only contain the base entities and their relationships but also a set of atomic components already detected by the first technique. The second technique must be prepared for this, i.e, we need incremental versions of the basic techniques.

Incremental analyses are not only useful for intermediate steps in combinations of techniques but also for the interactive method described in Chapter 9. The maintainer validates the candidates proposed by the diverse techniques and in doing so produces a partial description of the systems's atomic components. The techniques then find new atomic components based on this partial description. Chapter 9 will elaborate more on that. In this chapter, however, we discuss how the

information provided by the user can be captured and used by the incremental analyses.

This chapter will also describe a voting approach as a complement to the combining operators, in which the agreement of each technique with a given component, i.e., its underlying heuristic, is expressed as a metric. An atomic component candidate can then be assessed by summing up the agreements of the individual techniques to it.

8.2 User Information

Validating the candidates, the user has several choices: He or she can complete, accept, and reject partially and entirely. All this information has to be kept for the next iteration of an iterative detection process because confirmed information should not be presented for validation once more. For the purpose of representing components accepted by the user we can adopt the means introduced in Section 3.2.1 and Section 3.2.2 to represent atomic components and subsystems proposed by techniques. In this section, we will also introduce a few extensions to capture other aspects of user information.

8.2.1 User Information and Components Views

To distinguish candidates from reference components, we can make use of the resource usage graph views introduced in Section 3.6. To capture the point of view of the user, we use one dedicated view (per user), called the **user view** in the interactive method described in Chapter 9. The user view contains all information added by the user, inclusive acceptance of results of automatic analyses. Therefore, in order to ascertain whether something has already been established or rejected, we can simply consult this view.

A view that shows the decomposition of components is called **components view** (see Section 3.6). The user view is one example; all the results of the basic techniques can likewise be represented by components views. There are basically two kinds of information contained in a components view (possibly added by the user): positive and negative information. **Positive** information is any decision of the user that certain elements *belong*

together; it is considered **negative** information when the user decides that certain elements *do not belong together.*

We can use components views and the formalism introduced in Section 3.2.1 and Section 3.2.2 to represent positive information: The fact that the elements belong together is simply added to the user view in a relational fashion. However, we have not introduced means to express negative information yet. We will represent this fact by a new symmetric *mutually exclusive* relationship:

> ***mutually-exclusive (a, b)*** expresses that entity *a* and entity *b* must not be direct elements of the same component, nor may one be a direct part of the other one.

Note that *a* and *b* may also be components, i.e., atomic components or subsystems, in the definition above. Note also that the definition above does not exclude the example in Figure 8-1 in which *a* and *b* are both transitive parts of *C* (*a* is even a direct element of *C*) though *a* and *b* are mutually exclusive. The definition above specifies only that *a* and *b* must not be part of the same component; otherwise, *mutually-exclusive* edges could not be used to separate elements within the same subsystem. The user could, for example, decompose the whole system into subsystems with one single root; in this case, no *mutually-exclusive* edges could be used at all.

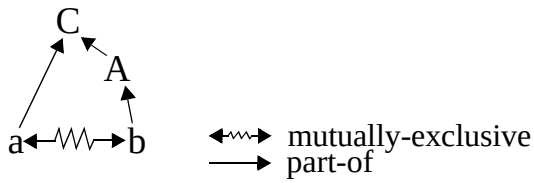


Figure 8-1. **Example for *mutually-exclusive*.**

A components view will be used as additional input parameter to the combining operators and incremental analyses in order to describe the currently established component decomposition. Likewise, the result of operators and analyses is also a components view. Since the operators and analyses do not have to care whether the input components view stems from the user or was generated by previous analyses, we can define the operators and analyses in the following in a uniform way, i.e., we need not care whether they are applied to the user view or in a com-

position of techniques. Since all analyses produce components views, we can easily combine the results by operators described below.

The next section describes the assumptions we are going to make about components views that have to be maintained by the operators and incremental analyses.

8.2.2 Constraints for Components Views

Components views were already introduced in Section 3.6 (Table 3-5 on page 68). Here, we extend components views to incorporate *mutually exclusive* edges. Therefore, the following additional assumptions will be made about components views:

- *Nodes*: A components view may contain subsystems, atomic components, and base entities.
- *Edges*: A components view may contain *part-of* and *mutually exclusive* edges among base entities, atomic components, and subsystems.
- *Constraints*:
  - A base entity is only in the components view if it is part of a component in the view or if it is an end of a *mutually exclusive* edge.
  - The constraints for subsystem structures as stated in Section 3.2.2 (may overlap, is acyclic, and has no redundant *part-of* edge) hold.

8.2.3 User Actions

Altogether, the user can manipulate a components view in the following ways:

- **creation**: create a new component
- **assignment**: add an entity to a component
- **rejection**: remove a bound entity from its component (which does not imply that the entity and its component are mutually exclusive from now on)
- **exclusion**: mark two entities as mutually exclusive (which implies rejection if one is part of the other one)
- **confirmation**: confirmed information is added to the user view

Note that the user is offered two ways of removing an entity from a component. If the entity is only rejected, the entity may be re-added to the component by subse-

quent analyses that take the current components views as input, which is useful when the user is unsure and wants to see whether another technique would also add the entity to the component. If the user is sure that the two entities do not belong together, he can mark the entities as mutually exclusive.

All user actions, excluding confirmation, affect only the components views manipulated by the user. Only on confirmation, information is transmitted to the user view and may affect newly started analyses.

8.3 Combining Operators

The heuristics we have described in Chapter 5 use diverse strategies to find atomic components. Some of them may be particularly powerful for certain kinds of atomic components, but rather weak for others. Some of them are not even able to detect certain kinds of atomic components at all; for example, the *Part Type* heuristic is not able to detect ADOs. It makes sense to combine these techniques in order to leverage individual strengths of the techniques and to compensate their weaknesses.

The base techniques are basically functions that take a view as input and produce a components view containing the atomic components detected, denoted as:

> T: $\mathcal{V} \rightarrow \mathcal{V}$ (where *T* is a technique for atomic component detection and $\mathcal{V}$ is the set of views) and the application of a technique is denoted by *T(V)* (this will be refined in Section 8.3.1).

Therefore, we can use functional *composition* to combine the techniques: The result view of one technique is the input view of another technique. Since the results are basically sets of atomic components, we can use the *union* and *intersection* as another way of combining them. Thus, the following operators can be used for the combination of the results of the basic techniques (let *V*, V_1, and V_2 be components views and T_1 and T_2 be techniques):

- union $(V_1, V_2) = V_1 \cup V_2$
- intersection $(V_1, V_2) = V_1 \cap V_2$

- composition (V, T_1, T_2) = T_2 (T_1 (V))

These are the core operators. In the following sections, we will discuss them in more detail. Here we give only a brief overview. The union operator is useful for techniques that produce very different kinds of atomic components and is, therefore, especially suited to combine techniques that are restricted to one class of atomic component. For example, *Delta IC* can only detect ADOs and *Part Type* only ADTs. Applying the union operator to these two heuristics allows the detection of both kinds of atomic components.

The intersection is used to reveal the agreement of two techniques: Only atomic components detected by both techniques will be present in the resulting view. This gives us a higher confidence about the resulting components. A good example is the intersection of *Part Type* with *Internal Access*. Remember that *Part Type* assumes that the parameter of the part type in the signature is put into or retrieved from the parameter of the container type. This can only be the case when the parameter of the container type is internally accessed.

The composition of techniques is the consecutive application of two techniques. The application of the second technique tries to group base entities that were not grouped by the first technique. The second technique can create new atomic components or add not yet bound entities to atomic components detected by the first technique.

The exact definitions of these combining operators will follow. At this point, we only point out that applying the set operators *union* and *intersection* is more than just composing, uniting, and intersecting the sets of atomic components represented by views in the sense of set theory. For the intersection, for example, we can hardly expect that we find two exactly equal atomic components by both techniques. A simple definition of intersection according to set theory would therefore yield empty result views in most cases.

Composition, intersection, and union are the core operators to combine the techniques, but there are other useful operators. Because some techniques consider variables and types, it may be desirable for their combination with other techniques to restrict them to one kind of base entity. For example, one might want to

apply *Internal Access* to detect ADTs only and to apply *Delta IC* to detect ADOs. Therefore, we add the following unary operator:

- restriction (V, k) = V_k where V_k is the view V restricted to entities of kind k

Since it generally does not make sense to exclude subprograms for the combination of techniques, k denotes either variables or types; the resulting view will always contain all the subprograms of the non-restricted view. *Internal_Access (restriction (V, Type))*, for example, will apply *Internal Access* to the View V_{Type} that contains all the subprograms and types of V but not the variables of V. In order to apply *Delta IC* to detect ADOs and *Internal Access* to detect ADTs in view V, for example, one can compute:

- union (Delta-IC (restriction (V, Variable)),
  Internal-Access (restriction (V, Type)))

Finally, a difference operator may also be useful in an interactive approach. It allows the maintainer, for example, to inspect the difference between his own point of view, i.e., the set of atomic components he has specified, and the set of candidates generated by an analysis:

- difference (V1, V2) = V1\V2 ∪ V2\V1

These operators allow powerful yet simple combinations of base techniques. Yet, not all possible combinations may be sensible; in particular, one has to be aware that multiple intersections may result in empty sets of atomic components and multiple unions may produce many overlapping atomic components. In the following sections, we will discuss the combining operators in more detail.

8.3.1 Composition

The composition operator applies two techniques consecutively. The output of the first technique is the input of the second technique. The second technique may add left free entities to the existing component (and create new components as well). Hence, the composition operator presumes incremental techniques. The base techniques as introduced in Chapter 5 have only one input parameter, namely, the base view. Recall that the base view consists of all base entities and their relationships that can directly be derived from source code (see Table 3-5 on

page 68). The incremental versions of these techniques have an additional parameter: the components view that describes the atomic components that have already been found. The result is a components view meeting the constraints listed in Section 8.2.2. Hence, the application of an incremental technique, *T*, can be described by a function:

> T: $\mathcal{V} \times \mathcal{V} \rightarrow \mathcal{V} \times \mathcal{V}$ (where $\mathcal{V}$ is the set of views) and the application of a technique is denoted by $T(V_1, V_2)$ where V_1 is the specific input view for the technique that describes the base entities and relationships considered by T and V_2 is a components view. Note that $T(V_1, V_2) = (V_1, V_2')$ holds, i.e., the first input view is not changed.

We will call base entities **bound** if they are already part of a component in the input components view. All other entities are considered **free**.

8.3.1.1 Restrictions Imposed by the Interactive Employment

The composition operator is used to support the semi-automatic method that is going to be described in Chapter 9. The interactive employment of the operator imposes further restrictions on the allowable operations on components views:

> (R1): Only free entities may be grouped.
>
> (R2): If a free entity, *E*, is to be clustered with a bound entity, *F*, entity *E* is to be added to the enclosing components of *F*.

The restrictions will be motivated in the following.

Ad (R1). There are basically two kinds of scenarios for the application of the composition operator:

1. the operator is applied to the user view
2. the operator is applied to an intermediate view (a view that results from analyses or combining operators)

When applied to the user view, the operator must preserve all information contained in the user view; in other words, the operator must not remove bound entities or reorganize existing components because the user has previously approved the elements of the user view. In principal, it would also be useful for a more gen-

eral usage within maintenance to allow the analyses to restructure existing components; the analyses would then propose alternative decompositions of existing components. However, the focus of this thesis is on *recovery* rather than *restructuring*. Nevertheless, the thesis provides building blocks to support restructuring. A simple way to support restructuring would be to allow clustering of all entities of the user view; then, the difference operator can be applied to the user view and the resulting view in order to investigate discrepancies. Another way to analyze an existing decomposition is to let the analyses rate the components as described in Section 8.4.

Because the user has previously approved the elements of the user view and because the focus of the operators described in this thesis is on recovery, an incremental technique may only create new components comprising free entities and add free entities to existing components, i.e., a bound entity must not be removed from its components nor may it be added to another component (because this would be another kind of restructuring).

The arguments against allowing an incremental technique to regroup or remove bound entities hold also for the second scenario in which the operator is applied to intermediate views. The operator is only applied by the user and if she selects a components view to be refined by the composition operator, we may assume that she agrees to the decomposition within the intermediate view, otherwise she would have modified the intermediate view by hand or rejected the intermediate view as a whole.

Ad (R2). Furthermore, during clustering, an analysis may decide to group a free entity, *E*, with another entity, *F*. If *F* is also a free entity, the two of them may be used to create a new atomic component when all free entities are clustered. However, if the input view of the composition operator is not empty, *F* may already be bound. Because bound entities cannot be removed from their enclosing components (as it was discussed above), there are basically two options on how to deal with a free entity, *E*, and a bound entity, *F*, that the analysis wants to be grouped together:

1. *E* and *F* are used to create a new component (but F remains still also a part of its enclosing component)
2. *E* is added to the enclosing component of *F*

The first option is not possible due to restriction (R1) which implies that a bound entity must not be added to another component. However, the second option is not only preferred because the first option is excluded by (R1). First, if restriction (R1) were not in place and *E* and *F* were added to a new component, the new component would overlap with the enclosing components of *F*, which would force the user to the check the addition of *F* to the new component though she has already decided on *F*.

Second, during the course of clustering, an incremental technique whose input components view is empty (i.e., all entities are free) puts *E* and *F* into the same new cluster if they are related. The same would be expected of an incremental technique whose input component view is not empty. For example, if the base entities *a*, *b*, and *c* in Figure 8-2(a) are related in the sense of the heuristic underlying a technique, *T*, that is to be applied, *T* will create a new atomic component, *AC*, that consists of *a*, *b*, and *c* as sketched in Figure 8-2(b). If the user decides to reject *b* in the result of technique *T* (Figure 8-2 (c)), *AC* will no longer contain *b* (Figure 8-2 (d)). However, because *b* was only rejected temporarily and not excluded (see Section 8.2.3), one would expect that *T* will add *b* again to *AC* when applied to Figure 8-2 (d). This effect is only achieved by the second alternative that adds *b* to the enclosing component of *a* and *c*.

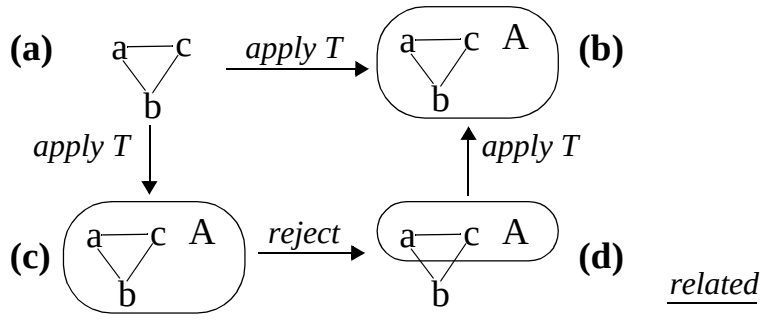


Figure 8-2. **Successive application of the same technique.**

The second alternative is also appropriate if two different techniques are applied successively. Consider the example in Figure 8-3 in which a composition of two techniques *S* and *T* is computed based on the second alternative. Only *a* and *c* are related with respect to technique *S* and therefore grouped by *S* into *AC*. When *T* is applied to the intermediate result in Figure 8-3(b) and *b* is related to *c* according to the clustering criterion of *T*, *c* is already bound to *AC*. Following the second

alternative, *T* groups *b* to *AC* as well. The second alternative allows to successively completing an atomic component using different criteria, i.e., different techniques can complement each other, while the first alternative would always generate a new component and assumes that one criterion is sufficient for entities to be in the same component. For these reasons, the second alternative is preferred.

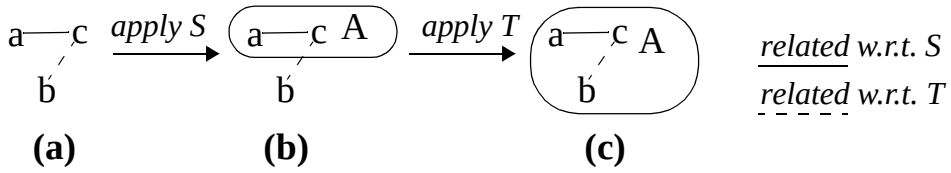


Figure 8-3. **Successive application of different techniques.**

8.3.1.2 Algorithm for the Composition Operator

The composition can be organized in the following steps (the term *cluster* is used here to make clear that the individual techniques generate sets of related elements that only become components in the last stage of the composition):

1. Iterate over the free entities and cluster them. The result are clusters with *related-to* information (see below).
2. Split all clusters so that there are no mutually exclusive entities in the same subcluster where each subcluster contains only entities that are (transitively) related to each other.
3. Transform clusters into components.

The first step depends on the analysis. It is described as composition in the following sections. The second and third steps are identical to all analyses and are explained in Section 8.3.1.6 and Section 8.3.1.7. An advantage of organizing the composition this way instead of letting the analyses be in charge of mutually exclusive entities is that the analyses do not have to take care of negative information. Other consequences of this decision will be discussed in Section 8.3.1.6.

We will discuss the diverse classes of approaches, namely, connection-based, metric-based, and graph-based techniques (as described in Section 5.13) separately. The clusters generated by all techniques are sets of related entities which

are to be grouped together. More precisely, a cluster is represented as a connected graph, so-called **related-to graph**, whose nodes are the entities to be grouped together and whose edges represent the symmetric *related-to* relationship that specifies which entities are related to each other in the sense of the underlying heuristic of the technique (the related-to relationship is a virtual relationship defined by each technique in its own specific way in the following and, therefore, not represented in the resource usage graph; one can think of the related-to graph as a separate data structure distinct from the resource usage graph). All elements of a cluster are at least transitively related to each other; otherwise, the techniques would not have grouped the elements together in the first place. However, the *related-to* annotation is needed when the cluster is split into subclusters due to mutually exclusive entities. Then, a subcluster may only contain related entities and no mutually exclusive entities. The *related-to* information also plays a role when clusters are transformed into components by step 3 as it will be explained in Section 8.3.1.7.

8.3.1.3 Composition for Connection-based Techniques

Connection-based approaches cluster entities based on a specific set of direct relationships in between entities to be grouped. They differ only in the types and characteristics of the relationships they consider. These relationships are present in the base view (or in a subview of the base view, respectively). The techniques iterate over the free entities in the base view and collect the connected entities that could be grouped. These connected entities themselves may be either bound or free. Free connected entities will belong to the same cluster of the entity that is under consideration. Bound connected entities cannot be grouped again because they already belong to an atomic component. Instead, the entity under consideration should belong to the same atomic component the connected bound entity belongs to (or the set of atomic components if the connected entity is bound to more than one). This will technically be solved in algorithm 8-1 by adding the enclosing atomic component(s) in lieu of the connected entity.

Initially, each entity is put into a set of its own using the disjoint sets algorithm already introduced in Section 5.2. For simplification, we assume that all entities (bound or free) and atomic components are enumerated from 1 to *Last_Entity.* The generic parameter *Connected_Entities* of this algorithm is specific to each connection-based approach and yields all the base entities that could be grouped

Generic Parameter:

- Connected_Entities : Entity → Set of Entities

Input:

- base view B
- components view A

Output:

- set of disjoint clusters with *related-to* information

Algorithm

1. *Put each free entity and component in a set of its own*:

```
for E in 1..Last_Entity where not Is_Bound (E, A) or Component (E) loop
   New_Set (E);
end loop;
```

2. *Iterate over free entities and cluster connected entities*:

```
for E in 1..Last_Entity where not Is_Bound (E, A) and not Component (E)
loop
   for C in Connected_Entities (E) loop
      if not Is_Bound (C, A) then
         Union (Find (E), Find (C)); add related-to (E, C);
      else
         for AC in Enclosing_Components (C, A) loop
            Union (Find (E), Find (AC)); add related-to (E, AC);
         end loop;
      end if;
   end loop;
end loop;
```

3. *Result*:

Each disjoint set represents a cluster that constitutes a candidate.

Algorithm 8-1. **Generic incremental connection-based clustering.**

according to the underlying heuristic of the approach. The function *Enclosing_Components* returns a set of atomic components and subsystems to which the given entity belongs. When two entities *A* and *B* are grouped together by *Union*, the *related-to* information is added.

The result of this algorithm is a set of disjoint clusters with *related-to* information. A cluster can consist of base entities and components. How the result is interpreted and converted into components is described in Section 8.3.1.7.

Algorithm 8-1 enhances all connection-based techniques to work incrementally. The techniques themselves need not be changed, i.e., the specification for *Connected_Entities* remains unaffected.

8.3.1.4 Composition for Metric-based Techniques

Delta-IC. As previously said, *Delta-IC* is a hybrid of metric-based and connection-based approaches. The actual clustering is connection-based while the metric is only used to remove bad candidates. Hence, an incremental extension of *Delta IC* can be organized analogously to connection-based approaches. A single step in the incremental connection-based techniques basically consists of two substeps: (1) select a cluster and (2) replace bound entities within these clusters by their enclosing components. This scheme can be adopted for *Delta-IC* as follows. At first, the clusters are identified according to the procedure explained in Section 5.7 and then bound entities are replaced within these clusters as described by algorithm 8-2 (only step 4 and 5 are new, all other steps were already explained in algorithm 5-4 on page 131). The reasons for replacing bound entities by their enclosing components were given in Section 8.3.1.1.

If a cluster is split into subclusters due to mutually exclusive entities, a subprogram, *S*, should be in a subcluster that contains an object, *O*, referenced by the subprogram. That is why a *related-to*(*S*, *O*) annotation is added to the cluster in step 4 of algorithm 8-2. If an entity, *E*, is related to an entity, *E'*, that is already bound, the induced *related-to* information with respect to the enclosing components of *E'* is added as part of step 5 of algorithm 8-2.

Input:

- object reference view V
- components view A
- ΔIC threshold Θ

Output:

- non-disjoint *clusters* with *related-to* information

Algorithm:

1. - 3.
   -- *See algorithm 5-4 on page 131*.
   -- *clusters is an array of clusters where* ΔIC (clusters (C)) ≥ Θ ∀*C*

4. add related-to information:

```
for each C in clusters'Range loop
   for each subprogram S in clusters (C) loop
      for each object O in referenced-objects (S) in V loop
         add related-to (S, O);
      end loop;
end loop;
```

5. *handle bound entities:*

```
for C in clusters'Range loop
   for each element E in clusters (C) loop
      if Is_Bound (E, A) then
         clusters(C):=(clusters(C)\{E})∪Enclosing_Components(E,A);
         -- clusters (C) is a set of entities, i.e., a single cluster
         for each E' in clusters (C) where related-to (E, E') loop
            for each AC in Enclosing_Components (E, A) loop
               add related-to (E', AC); -- induced related-to information
            end loop;
         end loop;
   end loop;
end loop;
```

Algorithm 8-2. **Extended incremental *Delta IC*.**

Other metric-based techniques. Both *Type-based Cohesion* and *Similarity Clustering* use the same hierarchical clustering algorithm that was proposed in Chapter 7; they only differ in the underlying metric. In an incremental approach, atomic components have already been detected and yet unbound entities have to be clustered. This compares to a snapshot of the clustering algorithm after a few runs when there are already some clusters and yet more iterations ahead. Considering this, the clustering algorithm can easily be modified to work incrementally. Only a pre- and a post-processing phase is necessary. In the pre-processing phase, the similarity relation is computed among all atomic components and all unbound entities using the group similarity (equation (7.2) on page 188). The clustering algorithm then clusters all atomic components and free entities based on the similarity relation just computed. The result of the clustering algorithm is a dendrogram whose subtrees are flattened into clusters using a user-determined threshold for the acceptable similarity among the elements of the subtrees as described in Section 7.4. If atomic components have been added in lieu of their elements in the pre-processing phase, the dendrogram may contain both components and base entities and, hence, the retrieved clusters contain components besides entities, too.

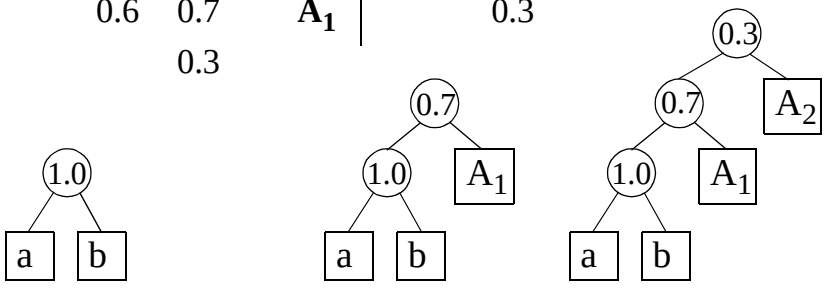
Note that it need not necessarily be the case that the similarity among *all* elements of a cluster retrieved from a dendrogram is above the threshold. Because the average as defined by (7.2) on page 188 is used as group similarity, a less similar entity may be balanced by strongly similar entities. Consider the example similarity matrix in Figure 8-4 that contains two base entities *a* and *b* and two atomic components A_1 and A_2. According to Figure 8-4(a), $\{a\}$ and $\{b\}$ are the most similar groups and are therefore united. The similarity of the union of $\{a\}$ and $\{b\}$ to all other elements is re-computed using the group average. The resulting similarity matrix is Figure 8-4(b) in which $\{a,b\}$ and $\{A_1\}$ are the two most similar groups. The final dendrogram for the similarity matrix Figure 8-4(a) is shown in Figure 8-4(c). In this example, *a* and A_2 are in the same cluster if a threshold $\Theta_{\ddot{y}} \geq 0.35$ is used to retrieve clusters from the dendrogram in Figure 8-4(c) even though the similarity among *a* and A_2 is only 0.1 according to Figure 8-4(a).

In this example, the high similarity among *b* and A_2 has lead to the inclusion of A_2 into the group.

	b	**A_1**	**A_2**
a	1.0	0.9	0.1
b		0.6	0.7
A_1			0.3

	A_1	**A_2**
a,b	0.75	0.4
A_1		0.3

	A_2
a,b,A_1	0.35



(a) **(b)** **(c)**

Figure 8-4. **Example dendrograms.**

To sum it up, the group similarity is used by hierarchical clustering to generate a dendrogram and the threshold is used to derive flat clusters from the dendrogram. The threshold is determined by the user and expresses his degree of tolerance about when two elements are similar enough to be in the same cluster. Using the average group similarity, an entity is added to a group that contains very similar entities even though there may also be a few entities for which the similarity is less than the threshold. The less similar entities are outweighed by the very similar entities. If the clusters are split into subclusters due to mutually exclusive entities, the resulting subclusters should contain those entities that are similar enough according to the user, i.e., whose individual similarity is above the threshold. Hence, the cluster is annotated with *related-to* (*a*, *b*) for all elements *a* and *b* of the cluster for which *Sim* (*a*, *b*) $\geq \Theta$ according to equation (7.3) on page 189 holds. The clusters derived from the dendrogram are then treated in a post-processing phase as described in Section 8.3.1.7.

8.3.1.5 Composition for Graph-based Techniques

Dominance Analysis was already presented as an incremental analysis: *Dominance Analysis* is applied to the collapsed base view and a base entity is added to

its primarily dominating component. The **collapsed view** for a view, *V,* results from the following transformations:

1. A bound subprogram, variable, and type in *V* is replaced by its enclosing atomic component.
2. Edges between unbound entities are kept whereas edges to a bound entity are replaced by edges to its atomic component and analogously, edges from a bound entity are replaced by edges from the atomic component.

Recall that base entities can directly be part of a subsystem. The fact that these base entities are part of a subsystem expresses only that they are in some way related but not related enough that they could be grouped into an atomic component. That is why we collapse only base entities that are part of an atomic component.

There are basically four kinds of possible combinations to consider when edges are to be added to the collapsed view (see Figure 8-5):

1. The edge is among entities that are not part of an atomic component; in this case, both entities will be added to the collapsed view including the edges among them.
2. The source of the edge is a free entity, *S,* and the target is a bound entity that is part of an atomic component, *A,* then *S* and *A* and only the induced edge from *S* to *A* will be added to the collapsed view.
3. The target of the edge is a free entity, *S,* and the source is a bound entity that is part of an atomic component, *A,* then *S* and *A* and only the induced edge from *A* to *S* will be added to the collapsed view.
4. The edge is among entities that are parts of atomic components A_1 and A_2, respectively; then A_1 and A_2 and an induced edge between A_1 and A_2 (only if $A_1 \neq A_2$) are added to the collapsed view.

There are two consequences for the collapsed view one should be aware of. First, collapsing may result in dependencies that were not present before, and second, collapsing overlapping atomic components is problematic.

The examples in Figure 8-6 illustrate the first property. In example (a), a cycle results because *x* and *y* are collapsed into one node though there was previously

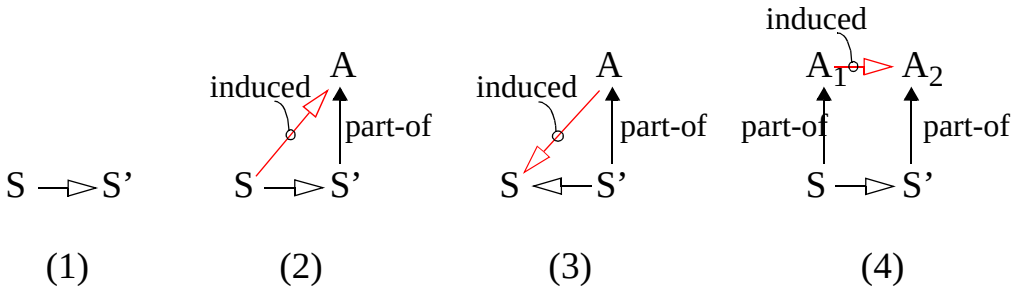


Figure 8-5. **Induced relationships.**

no such cycle. In example (b), collapsing *x* and *y* has as result that *b* is reachable from *a* though it was not reachable from *a* before. (The discussion of the effect of collapsing nodes will be continued in the discussion of *Strongly Connected Components Analysis* below.)

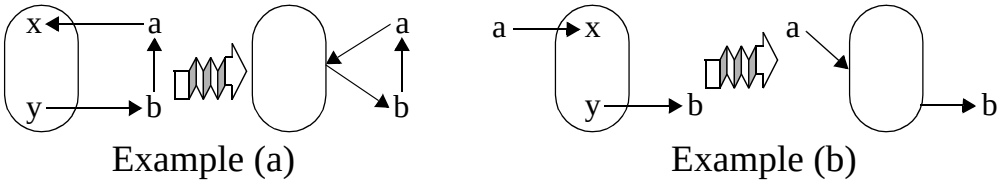


Figure 8-6. **Examples for changed dependencies in the collapsed view.**

The problem with collapsing overlapping atomic components is that all involved atomic components inherit the dependencies of the overlapping parts. This is illustrated by the example in Figure 8-7 where AC_1 and AC_2 overlap in *y* and *x*. Both AC_1 and AC_2 are assumed to call *a* and to be called by *b* in the collapsed view. Furthermore, because AC_1 contains *x* which calls *y* that is also contained in AC_2, AC_1 is assumed to call AC_2 and vice versa. Hence, the ambiguity that was present in the overlapping atomic components continues in the resulting collapsed view and the user is, therefore, urged to limit overlapping in the atomic components.

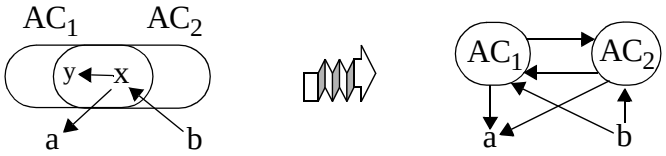
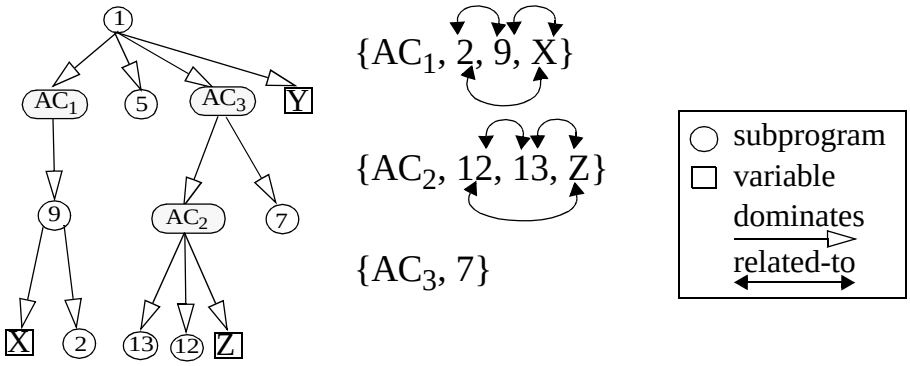


Figure 8-7. **Examples for collapsing overlapping atomic components.**

In the dominance tree for the collapsed view, the clusters proposed by the *Dominance Analysis* heuristic consist of an atomic component and all its primarily dominated entities. If such a cluster is to be split due to mutually exclusive entities, an entity should be added to those subclusters that contain its dominator, its dominatees, or its siblings in the dominance tree since these are the entities most related according to the dominance relation. As an example, consider the dominance tree in Figure 8-8. The clusters proposed by *Dominance Analysis* for the dominance tree Figure 8-8(a) are shown in Figure 8-8(b) including the *related-to* information. The *related-to* information for the atomic component is omitted for readability reasons.



***(a)** dominance tree* ***(b)** clusters*

Figure 8-8. **Dominance example.**

Strongly Connected Components Analysis detects cycles in the call view, or in other words, mutually recursive subprograms. There are basically two avenues for an incremental *Strongly Connected Components Analysis*:

- **Strongly Connected Collapsed Component Analysis**: One can apply strongly connected components analysis to the collapsed view as it was already described for dominance analysis, i.e., atomic components are collapsed first and only then, the analysis searches for cycles.
- **Strongly Connected Base Component Analysis**: One can apply strongly connected components analysis first to the call view and then try to add the cycles to existing atomic components.

The first approach may yield artificial cycles due to the collapsing of nodes as illustrated by the examples in Figure 8-6 and Figure 8-7. In Figure 8-6(a), the

cycle is introduced because *x* and *y* are treated as one node. Recall that strongly connected components are considered a relevant kind of atomic component because all the parts in the cycle are necessary to understand the component. This is not only so for mutually recursive subprograms. This also applies to artificial cycles as in Figure 8-6(a): Because *x* and *y* are part of the same atomic component, their respective comprehension is very much related.

But what about the cycle in Figure 8-7 that causes the two overlapping components to be considered a strongly connected component? Because of the ambiguity in the overlapping part, the two atomic components are obviously very much related and the *Strongly Connected Component Analysis* does not reveal more than what was already present. If the user does not want this effect, a better way of modeling the relatedness of two overlapping components and the unclear classification of the ambiguous part is offered by means of the subsystems introduced in Section 3.2.2. A better representation for Figure 8-7 is given in Figure 8-9 where AC_1 and AC_2 as well as the overlapping part *x* and *y* are all distinct parts of a subsystem. This avoids the artificial cycle in the collapsed view.

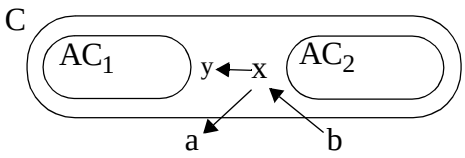


Figure 8-9. **Better representation for overlapping atomic components.**

The second approach that applies *Strongly Connected Component Analysis* to the call view yields only real cycles. On the other hand, it is therefore not able to detect atomic components that are mutually dependent, though this is an important information to the maintainer: One cannot reuse one of these atomic components in another context without taking on all of them. Both approaches have their advantages and disadvantages, so it is only pragmatic to offer both to the user.

The two approaches can technically be reduced to the incremental solution for connection-based approaches where the elements of the strongly connected component act as connected entities, i.e., in order to instantiate algorithm 8-1 on page 268, the following definition of *Connected_Entities* can be used:

$$\text{Connected\_Entities}(S) = \{s \mid s \in \text{called}(S) \wedge S \in \text{called}(s)\} / \{S\} \quad \textbf{(8.1)}$$

where $\text{called}(s) = \text{transitive\_closure}(\text{successors}(s, \{\text{call}\}))$.

In the case of *Strongly Connected Base Component Analysis*, *Connected_Entities* is established on the call view, whereas *Strongly Connected Collapsed Component Analysis* ascertains *Connected_Entities* on the collapsed call view.

When cycles are split due to mutually exclusive entities, an entity should be in the subcluster that also contains its direct neighbors in the cycle since these are the elements of the cycle most needed to understand the entity. Hence, the proposed clusters are annotated by *related-to* (*a*, *b*) for all elements *a* and *b* of the cluster that are direct neighbors in the cycle.

8.3.1.6 Partitioning Clusters with Mutually Exclusive Elements

The previous sections explained how the individual techniques can be adapted to work incrementally. They all yield clusters of related entities that are to be transformed into actual components. The clusters may contain mutually exclusive entities since the techniques as described above do not take negative information into consideration. Clusters with mutually exclusive entities must be split into subclusters without conflicting entities, which is uniformly handled by a separate step after clustering. This section explains how.

The input to the partitioning stage are the clusters that were formed by the previous step and are annotated with *mutually-exclusive* and *related-to* information, i.e., these clusters are associated with an **interference graph** that describes the entities that mutually exclude each other and the **related-to graph** that describes which entities are related.

Generally, not all entities in a cluster are mutually exclusive and, therefore, the cluster is not completely senseless. Instead of throwing away the whole cluster, the cluster should be partitioned into subclusters without mutually exclusive entities. A second requirement for a reasonable splitting is that the subclusters should be as large as possible. Subclusters with only one element obviously do not have any conflict, but they are not very helpful either. Unfortunately, we are facing the

NP-complete graph coloring problem here, i.e., for an optimal solution, we may need exponential time.

The graph coloring problem is to assign a minimal number of colors to nodes in a graph where no two neighboring nodes may have the same color. This is equivalent to partitioning a graph's nodes into sets of nodes where no two neighboring nodes are in the same set. This problem can be tackled heuristically as follows:

1. Remove nodes from the interference graph in the order of least to most conflicts and put them onto a stack. When all nodes are on the stack, the nodes with most conflicts are at the top of the stack.
2. Then, the stack's nodes are popped and assigned to a partition such that no neighboring nodes are in the same partition.

Algorithm 8-3 implements this strategy. Function *Choose_Node* selects a node with minimal conflicts and function *Partition_Number* yields the minimal partition number that can be assigned to a given node *N* where the corresponding partition does not contain a node that is in conflict with *N*.

Using the algorithm in the described way yields subclusters that do not contain mutually exclusive entities. The number of generated subclusters is a local optimum, that is to say, there may be better solutions with less, hence, larger subclusters, but the found solution is generally a good approximation. Finding the optimal solution would require exponential time in the worst case.

However, the algorithm used in the described way can yield subclusters of unrelated entities. For example, given the cluster in Figure 8-10, this algorithm may yield two subclusters $\{T_2\}$ and $\{T_1, F_1, F_2, F_3\}$ though F_1 and F_3 are not related to T_1.

In order to avoid unrelated subclusters, function *Partition_Number* needs to be provided with a preference rule: The entity is preferably added to a cluster that has most related entities (and no mutually exclusive one). Because the answer depends on the heuristic that produced this cluster in the first place, the detection techniques annotated the generated clusters with *related-to* information.

Input:

- Cluster C

Output:

- A set of clusters C_1, C_2,..., C_n where:

$$C = \bigcup_{i = 1 \ldots n} C_i \wedge (\forall (i, j \in \{1 \ldots n\}) i \neq j \Rightarrow C_i \cap C_j = \varnothing)$$
$$\wedge \forall (i \in \{1 \ldots n\}) \neg \exists (a, b \in C_i) \text{mutually\_exclusive(a,b)}$$

Algorithm:

1. *Initialization*:
   C' := C;
2. *Build stack; top most element has most, bottom element has least conflicts*:
   while not Is_Empty (C') **loop**
   Node := Choose_Node (C');
   Remove_Node (C', Node);
   Push (Node_Stack, Node);
   end loop;
3. *Partitioning (graph coloring)*:
   while not Is_Empty (Node_Stack) **loop**
   Node := Pop (Node_Stack);
   Assign (Node, Partition_Number (Node, C));
   end loop;

Algorithm 8-3. **Partitioning clusters with mutually exclusive elements.**

However, modifying *Partition_Number* solves the problem of unrelated entities only partly because the algorithm 8-3 treats all elements equally when putting them onto the stack. For example, F_1 and F_3 could be above T_2 in the stack (Figure 8-10) when F_1 and F_3 have conflicts, too. Then the top element, say F_1, would be assigned to a subcluster first. When F_3 is to be added, there is no connection to F_1 and, hence, F_3 could be added to a separate subcluster. Only then T_2 is to be

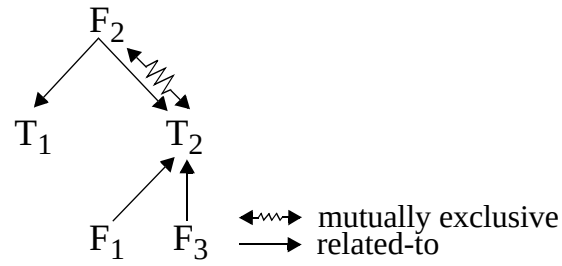


Figure 8-10. **Cluster with mutually exclusive entities.**

handled and may either be added to the subcluster of F_1 or F_3 resulting in one of F_1 and F_3 becoming orphaned.

In general, types as well as variables constitute a crystallization point for clustering and, therefore, should be distributed to subclusters before subprograms. Algorithm 8-3 can be adjusted to this strategy by first putting all subprograms onto the stack and only then the remaining types and variables. This way, types and variables are above all subprograms in the stack and, therefore, get partitioned first. The only necessary modification is to adapt *Choose_Node* accordingly.

The resulting subclusters can then be transformed into components as described by the following section.

8.3.1.7 Transforming Clusters into Components

The (sub-)clusters generated by the previous step as described in the last section contain related entities that are not mutually exclusive. These clusters are basically sets of entities and are now to be transformed into actual components.

In general, clusters with only one element are not useful and are, therefore, rejected. Likewise, huge clusters may also be rejected. In an interactive approach, the user is able to specify lower and upper bounds for reasonable components (the upper bound should only be set when the user does not plan to refine the candidate components by a second analysis).

Principally, the clusters generated by the incremental techniques fall into one of the following categories:

1. The cluster contains only base entities. Such clusters represent completely new atomic components.
2. The cluster contains a single atomic component, all other elements are base entities. This is the case when an incremental technique wants the base entities of the cluster to be added to an existing atomic component. Remember that the enclosing atomic component is added to a cluster in lieu of a bound entity.
3. The cluster contains more than one atomic component. This happens when at least one base entity can be added to more than one existing atomic component.

For the first type of cluster, we can create a new *atomic component* that contains all the elements of the cluster (Figure 8-11(a)). In the second case, we add all elements to the *atomic component* contained in the cluster (Figure 8-11(b)).

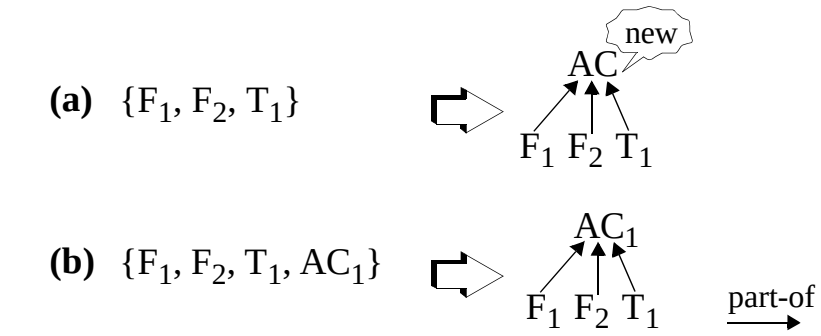
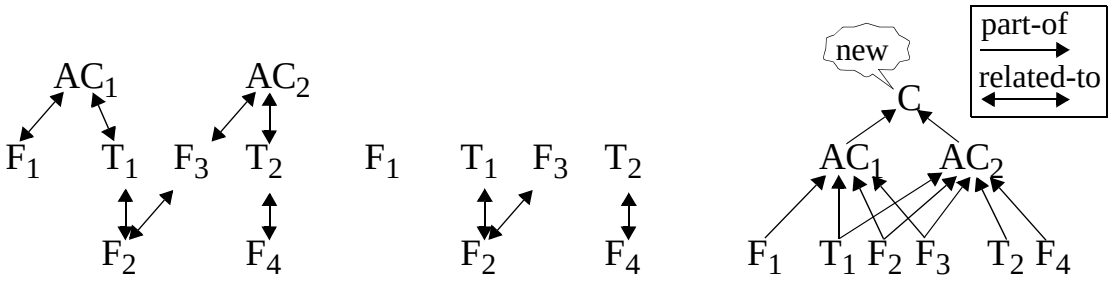


Figure 8-11. **Transforming clusters into components.**

If clusters contain more than one atomic component, it is not clear to which atomic components the base entities of the cluster should belong. Therefore, such clusters have to be presented to the user as a whole and he or she has to decide. To achieve this, a new *subsystem* is introduced and all atomic components of the cluster are considered part of the subsystem. The *related-to* annotation determines the possible elements of the atomic components. These elements can be derived as follows:

1. The **related-to graph** is ascertained whose nodes are entities and whose edges represent the *related-to* relationship (Figure 8-12(a)).
2. The **restricted related-to graph** is derived from the related-to graph by omitting components, i.e., the restricted related-to graph contains base entities only (Figure 8-12(b)).

3. The direct elements of an atomic component, *A*, are all members of connected subgraphs (in a connected graph, each node can be reached from each other node) of the restricted related-to graph that contain at least one node, *N*, for which *related-to* (*N*, *A*) holds in the original related-to graph (Figure 8-12(c)). Furthermore, all atomic components are part of a new subsystem node.



(a) *related-to graph* ***(b)*** *restricted related-to graph* ***(c)*** *component decomposition*

Figure 8-12. **Handling clusters with more than one component.**

Restricting the related-to graph before identifying the related entities is necessary in order to add only those entities to a component that are (transitively) related to the component itself. Otherwise, entities would be added to components that actually belong to other components, e.g., T_2 is primarily related to AC_2 and not to AC_1. The fact that basically all elements are related in the cluster is expressed by subsuming them under the same subsystem.

As the example in Figure 8-12 illustrates, overlapping components may be proposed using this strategy. However, the overlap arises from the actual *related-to* information. It is the user's decision how the overlap is to be resolved. Moreover, if the deep intersection operator is applied to the resulting components view, the overlap may also be resolved by the judgement of another technique.

An alternative strategy to assign base entities to atomic components in clusters with more than one component would be to use dominance analysis applied to the related-to graph assuming a subsystem node as root to which all atomic components of the cluster are related. Then, entities would be added to their dominating component. However, if an entity is (transitively) related to more than one component, the entity would be dominated by the root node and not by one of the atomic

components, i.e., the information about the overlap would be lost. For this reason, the first strategy is preferred.

8.3.2 Set Operators for Combining Components Views

The former section explained one of the core operators to combine components views, namely, the composition. This section describes further operators that are modeled on set operations.

Components views are basically sets of components and, hence, can be combined by set operations. In order to perform the set union, intersection, and difference for components views, we have to identify matching components of the two views to be combined; e.g., the intersection operator yields only those components that are in both views. This raises the question how do we determine whether two components match? Remember that in our relational perspective of components, a component has two faces: A component is characterized by a component entity and a set of its parts. Thus, a component is actually a named set. As a consequence, we can compare two components by name as well as by their elements. A comparison by name would consider the two atomic components AC_1 and AC_2 equal when they are identical, i.e., $AC_1 = AC_2$, in other words, when they are represented by the same entity; a comparison by elements, on the other hand, would consider AC_1 and AC_2 equal when their elements are the same, i.e., *elements* (AC_1) = *elements* (AC_2). The comparison by name makes sense when two components views are the result of different incremental techniques applied to the same input view. Then, the same component may occur in both views if it is present in the input view. On the other hand, when two incremental techniques are applied to disjoint views, identical components cannot occur. Then, we want to compare components in terms of their contents rather than name. For example, because AC_1 and AC_2 in Figure 8-13 have the same elements, they can be considered equal from the set perspective though AC_1 and AC_2 may be different entities in the relational perspective.

However, using set operations for combining views by elements is still too simplistic to be useful for two reasons. First, small differences of the elements lead to duplication or removal of similar components for union and intersection, respectively, and second, the comparison based on elements does not come up to the

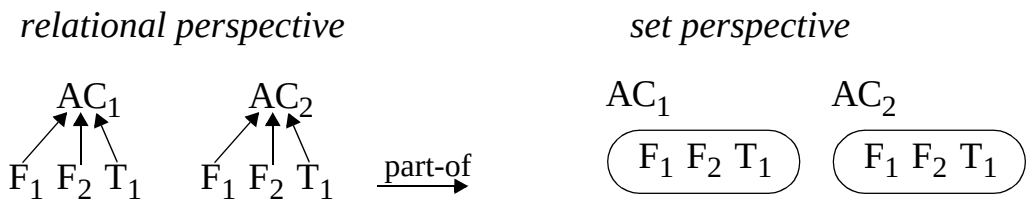


Figure 8-13. **Two equal components AC_1 and AC_2 from different perspectives.**

hierarchical structure of subsystems. In the following, these problems will be discussed in detail.

The purpose of a union operator is to combine two techniques that detect more or less distinct entities, i.e., their results are mainly disjoint sets of disjoint candidates such that the union operator can be interpreted as a union of sets. However, there is no guarantee that the results really do not overlap, and if they do overlap, using the simple union for sets based on elements may lead to almost duplicated candidates. For example, in Figure 8-14, the simple union, so-called **shallow union**, of the sets of candidates *Result 1* and *Result 2* treats any component as an atomic set member and produces five candidates where four of them are very similar. If this is presented to the maintainer, she has to check almost equal components twice.

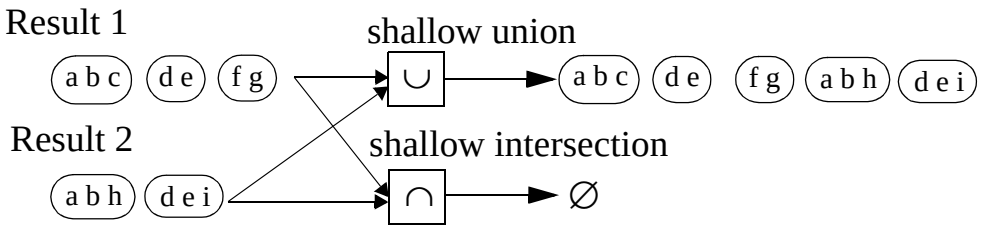


Figure 8-14. **Shallow union and intersection based on elements.**

The outcome for intersecting the two components views in Figure 8-14 is even worse since the resulting view is empty. For set intersection, a component of one view is only in the resulting view when it has an exact match in the other view.

Beside the problem of similar, yet not equal components, the common set operators do not come up to the hierarchical structure of subsystems. In the case of atomic components, we can consider their elements in order to find out whether a comparable atomic component exists in the other view; but considering only the

elements is not appropriate for subsystems since subsystems with the same set of elements may still differ in structure.

In the following, ways of combining two views that are modeled on set union, intersection, and difference are described that overcome the deficiencies of the shallow set operator semantics. In particular, the proposed operators

- are based on both comparison by name and by elements,
- tolerate divergences of the elements of the components,
- consider the structure of subsystems,
- and also combine the parts of subsystems at all levels.

In order to support comparison by name and by elements as well as to tolerate divergences, the matching criterion of the new combining operators for two components is the correspondence as defined in Section 3.7: We consider two components corresponding when they are identical or affine where affinity is associated with a tolerance parameter that allows inexact matches. The correspondence of components is defined at all levels such that a component in one view may be matched with a component in the other view that is a subcomponent of a larger subsystem. For example, C_1 and C_2 of the components views V_L and V_R in Figure 8-15 are a match where C_2 is a part of C_3; C_4 and C_5 — that are transitive parts of C_1 and C_2, respectively — are a match, too. The corresponding components are then combined by uniting, intersecting, or building the difference of their direct elements depending on the respective operator. Only the direct elements of corresponding components are combined since only these were the basis on which correspondence was established. Moreover, if there are corresponding transitive parts of two subsystems, they will likewise be combined at their respective level. The combination of C_1 and C_2 of Figure 8-15, for example, combines the direct elements of C_1 and C_2 as sketched on the right hand side of Figure 8-15 by applying the underlying set operator to their direct elements α_1 and α_2. Furthermore, the corresponding parts of C_1 and C_2, namely, C_4 and C_5 will also be combined analogously. Whether C is also in the result view despite of the fact that there is no correspondent, depends upon the operator (see below).

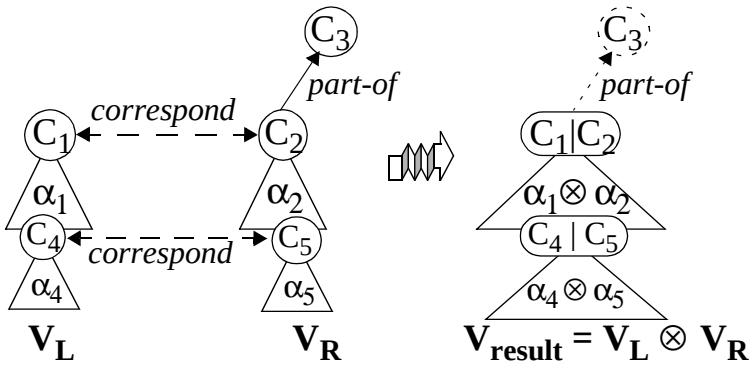


Figure 8-15. **Deep set operator outline.**

Because not only components views as such are united or intersected but also corresponding components contained in these views, these operators are called **deep union, deep intersection,** and **deep difference**. Beside corresponding components, the deep set operators also determine how nodes that do not have a corresponding node, so-called **singles**, have to be treated. In the case of the intersection operator, for example, they have to be ignored; in the case of the union operator, they have to be added to the result view.

The deep set operators basically consist of the following three steps and differ only in the actual *set operation* ⊗ that is to be applied:

1. Find the corresponding components in the two components views.
2. Handle correspondents: Apply the *set operation* ⊗ to the direct elements of all corresponding components.
3. Handle singles.

How these steps can be implemented in a general manner for all deep set operators is explained in this section (see algorithm 8-4). The following sections describe the specific parts of the concrete deep set operators.

Handling correspondents and singles. The generic algorithm 8-4 has three generic parameters. *Handle_correspondents* combines two corresponding components. The other two generic parameters are used to handle singles. There are two generic parameters for singles because the symmetric difference operator treats singles in the two argument views differently. The view that is given as first

Generic

- *handle_correspondents*: View × View × Entity × Entity × View
- *handle_left_single*: View × Entity × View
- *handle_right_single*: View × Entity × View

Input

- components views V_L, V_R

Output

- components view V that contains the result of the deep set operator

Algorithm

```
-- Identify corresponding components
for each L in reverse-topological-order (V_L) loop
   if ∃ (R ∈ V_R) identical (L, R) then
      add R to correspondents (L); add L to correspondents (R);
   end if;
   for each R in Potentially_Affine (L) loop
      if affine (L, R) then
         add R to correspondents (L); add L to correspondents (R);
      end if;
   end loop;
end loop;
-- Handle correspondents and singles
for each L in reverse-topological-order (V_L) loop
   if correspondents (L) = ∅ then
      handle_left_single (V_L, L, V);
   else
      for each R in correspondents (L) loop
         handle_correspondents (V_L, V_R, L, R, V);
      end loop;
   end if;
end loop;
for each R in reverse-topological-order (V_R) where correspondents (R) = ∅
loop
   handle_right_single (V_R, R, V);
end loop;
```

Algorithm 8-4. **Generic algorithm for deep set operators.**

argument to the operator is called **left argument view**, the one given as second argument is called **right argument view**.

Finding correspondents. Before the set operation can be applied, the corresponding components have to be identified. Correspondence of base entities can immediately be established by identity; identity is their only way of correspondence since they do not have further elements. Correspondence of atomic components can also be quickly established based on their direct elements. Ascertaining corresponding subsystems is more difficult since subsystems can have several levels and the correspondence at one level depends on the correspondence at the deeper levels, i.e., the correspondents of the parts of a component have to be established before one can determine its own correspondents. Since a subsystem structure is an acyclic directed graph, the subsystem structure in one components view can be traversed in reverse topological order (inverting the *part-of* edges) where the visited nodes are compared to nodes in the subsystem structures of the other components view. Traversal of acyclic graphs in reverse topological order ensures that a node is only visited when all its parts have been visited. As a consequence, we can assume that the corresponding nodes of its parts have already been established.

When a node is visited, its corresponding nodes in the other components view can be determined by identity or affinity. Whether there is an identical node in the other components view can be decided immediately. In order to identify affine nodes, however, there has to be a component in the other view whose direct elements correspond to the direct elements of the currently visited node. Fortunately, we do not have to compare all possible pairs of components in the two components views in order to find affine components. According to the definition of affinity given in Section 3.7.2, a node *A* can only be affine to a node *B* if

$$\exists(a \in \text{direct-elements}(A))\exists(b \in \text{direct-elements}(B))\text{correspond}(a, b)$$

That is, the principal components that could be affine to *A* are as follows:

$$\begin{aligned}&\text{Potentially\_Affine}(A) = \\ &= \{B \mid a \in \text{direct-elements}(A) \wedge \text{correspond}(a, b) \wedge b \in \text{direct-elements}(B)\} \\ &= \{B \mid \text{part-of}(a, A) \wedge \text{correspond}(a, b) \wedge \text{part-of}(b, B)\}\end{aligned}$$

Since all constituting conditions for membership of this set (i.e., *part-of* and *correspond* when it is saved once computed) can be checked in constant time and the number of parts can be assumed to be below a constant limit, this set can be computed in constant time. This set allows only to identify the potentially affine components; by iterating over its elements and checking each element for affinity with *A* as defined in Section 3.7.2, the actual affine components can be identified. Note that each component may have more than one correspondent as it was shown in Section 3.7.3. That is why *correspondents (A)* in Figure 8-4 denotes a set of entities.

One could argue that traversing only the left argument view is not sufficient because the correspondence information of nodes in the right argument view is not yet available. But this is not the case. Let us consider the scenarios in Figure 8-16. When a node *A* of the left argument view is visited, this node can either be a leaf or an inner node in a subsystem structure. Because leaves do not have further elements, the only possible way of correspondence with a node *B* in the other view is by identity (Figure 8-16(a) and (b)). Hence, the direct elements of *B* are irrelevant to the correspondence with *A*.

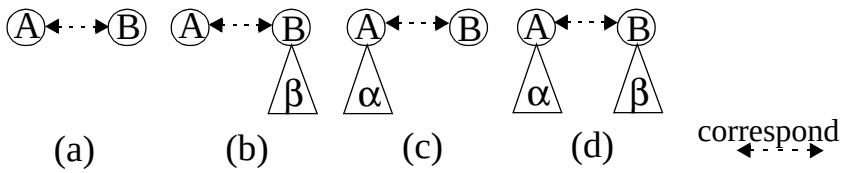


Figure 8-16. **Constellations of correspondence.**

Now, let us assume *A* is an inner node, i.e., it has direct elements in the components view. Again, *B* can either be a leaf or an inner node. If *B* is a leaf, the only possible way of correspondence is the one by identity again (Figure 8-16(c)). Thus, the only situation in which correspondence can be by affinity is when both *A* and *B* are inner nodes (Figure 8-16(d)). However, if *A* is an inner node, its parts have already been visited and their correspondents are all known. Hence, all needed information to find out whether *A* and *B* correspond is available.

Algorithm 8-4 traverses the left argument view twice. This is needed because the following descriptions of the actual procedures to handle correspondents and singles assume that the correspondence information has been established for all

direct elements of its argument entities. To illustrate that actually two traversals are needed, consider Figure 8-17. Let us assume that the nodes 1, 2, 3, and 4 have already been visited and AC_1 is the currently visited node during the first traversal in reverse topological order. At this point, it is known that AC_1 and AC_3 correspond to each other. Moreover, the singles and nodes with correspondent among the direct elements of AC_1 are known as well. However, it is not necessarily known for the direct elements of AC_3 that do not have a correspondent among the direct elements of AC_1 whether they do not have a correspondent at all. For example, the direct element 5 of AC_3 actually has a correspondent, but its correspondent in V_L has not yet been visited and is therefore not yet known. If *handle_correspondent* (AC_1, AC_3) were called during the first traversal, *handle_correspondent* would consider node 5 a single though it actually has a correspondent.

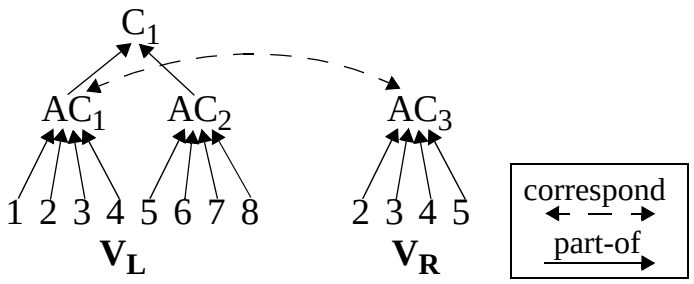


Figure 8-17. **Example traversal of subsystem structures.**

The generic algorithm 8-4 implements the steps to combine two views. The deep union, intersection, and difference are instantiation of this general scheme by putting the generic parameters into concrete terms. The generic parameters represent the underlying shallow set operation that is to be applied to the direct elements of corresponding components. For example, the deep union operator unites all direct elements of corresponding components using the regular set union (by providing an adequate implementation for *handle_correspondents*). Since the corresponding components do not depend on the deep set operation and, therefore, the regular set operation is applied to the same pairs of entities independent from the actual deep set operation, the following equation holds:

$$\text{deep-union}(A, B) = \text{deep-intersection}(A, B) \cup \text{deep-symmetric-difference}(A, B)$$

on the analogy of set theory:

union(A, B) = intersection(A, B) $\cup$ symmetric-difference(A, B)

We do not prove this equation since it becomes plausible by the working example that is used to explain the deep set operators in the following.

Working example. Figure 8-18 contains two subsystems C_1 and C_2 that are to be compared. The figure already connects the corresponding components, assuming as tolerance parameter $\Theta = 3/5$. Corresponding base entities are not explicitly highlighted for ease of readability. Since base entities correspond by identity, the pairs of corresponding base entities are obvious. Here, we will explain the generic algorithm 8-4 by describing a traversal and listing the generic subprogram parameters that are called during this traversal. The following sections, in which the concrete subprograms to instantiate the generic algorithm for the actual deep set operators are described, will only present the result for the respective instantiation.

One possible reverse topological order of C_1 (among others) is to visit the nodes in the order of 0, AC_0, 1, 2, 3, 4, AC_1, 5, 6, 7, 8, AC_2, 10, C_1. Base entity 0 has no identical node in the other view and, therefore, *handle_left_single*(V_L, 0, V) is called. Likewise, AC_0 has no identical node, nor has its direct element 0 has a correspondent and, therefore, AC_0 has no correspondent. That is why *handle_left_single* (V_L, AC_0, V) is called. 1 has no correspondent (*handle_left_single* (V_L, 1, V) is called), but 2, 3, and 4 have identical nodes in the other view, which results in calls *handle_correspondents* (V_L, V_R, 2, 2, V), *handle_correspondents* (V_L, V_R, 3, 3, V), and *handle_correspondents* (V_L, V_R, 4, 4, V). AC_1 has no identical node but its direct elements 2, 3, and 4 have corresponding entities whose enclosing component is AC_3. Therefore, *Potentially_Affine* (AC_1) is $\{AC_3\}$. AC_1 and AC_3 are in fact affine for every tolerance factor $\Theta \leq 3/5$ and, hence, *handle_correspondents* (V_L, V_R, AC_1, AC_3, V) is called. Then, *handle_correspondents* is called for the pairs of identical nodes (5,5), (6,6), (7,7), and (8,8). AC_2 has an identical component in the other subsystem structure; thus, *handle_correspondents* (V_L, V_R, AC_2, AC_2, V) is called.

Note that the two identical components are not affine for Θ = 3/5 because their degree of overlap according to condition (3.2) on page 70 is only 2/5. 10 does not have a correspondent, which leads to a call to *handle_left_single* (V_L, 10, V). Eventually, C_1 is visited. Its direct elements are AC_0, AC_1, AC_2, and 10. AC_1 and AC_2 have corresponding components in the other view whose enclosing component is C_2. Hence, *Potentially_Affine* (C_1) is {C_2}. The corresponding direct elements of C_1 and C_2 are (AC_1, AC_3) and (AC_2, AC_2). Distinct elements of C_1 and C_2 are AC_0, 10, 6, AC_4. The degree of overlap for C_1 and C_2 is, therefore, 2/(2+4) < Θ = 3/5. Thus, C_1 and C_2 are distinct and *handle_left_single* (V_L, C_1, V) is called.

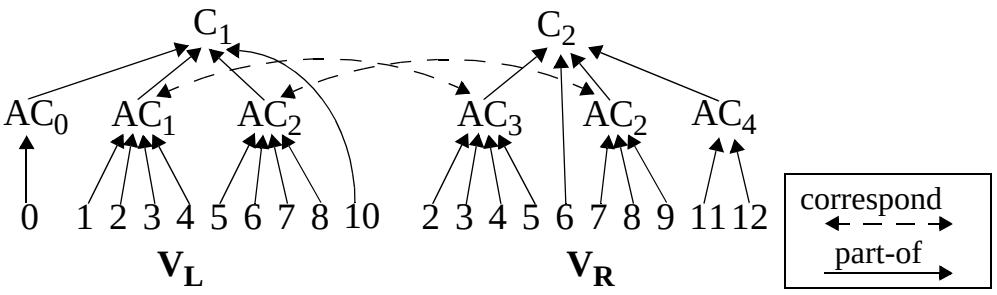


Figure 8-18. **Working example of partly corresponding components.**

After the second reverse topological traversal of V_L, *handle_right_single* is called for each remaining single of the components view given as second argument of the deep set operator. To sum it up, the generic operations are executed in the order given in Table 8-1.

Notation. The nodes that represent the combination of two corresponding nodes L and R are represented in the result of the combination as a pair (L, R) if $L \neq R$. If $L=R$, only the single node L is added to the output view in order to preserve its identity for future combinations. A single entity L is equivalent to a pair (L, L).

8.3.2.1 Deep Union

The deep union operator unites corresponding components of its argument components views in order to avoid duplicated similar components. All entities that do not have a correspondent are added to the output components view, too, since this is the expected semantics of a union. That is, *handle_correspondents* will

Table 8-1. **Executed generic subprogram parameters.**

handle_left_single (V_L, 0, V)
handle_left_single (V_L, AC_0, V)
handle_left_single (V_L, 1, V)
handle_correspondents (V_L, V_R, 2, 2, V)
handle_correspondents (V_L, V_R, 3, 3, V)
handle_correspondents (V_L, V_R, 4, 4, V)
handle_correspondents (V_L, V_R, AC_1, AC_3, V)
handle_correspondents (V_L, V_R, 5, 5, V)
handle_correspondents (V_L, V_R, 6, 6, V)
handle_correspondents (V_L, V_R, 7, 7, V)
handle_correspondents (V_L, V_R, 8, 8, V)
handle_correspondents (V_L, V_R, AC_2, AC_2, V)
handle_left_single (V_R, 10, V)
handle_left_single (V_R, C_1, V)
handle_right_single (V_R, 9, V)
handle_right_single (V_R, 11, V)
handle_right_single (V_R, 12, V)
handle_right_single (V_R, AC_4, V)
handle_right_single (V_R, C_2, V)

unite its two arguments and *handle_left_single* / *handle_right_single* simply add their argument to the output view by taking on their parts.

Handling correspondents. In more detail, for any pair of corresponding entities <L, R> of the two views, a new entity (*L*, *R*) is added to the output view that represents the union of the two corresponding entities *L* and *R* (algorithm 8-5). The direct elements of (*L*, *R*) are the pairs of correspondents among the direct elements of *L* and *R* plus all direct elements of *L* and *R*, respectively, that do not have a correspondent. Consider the example in Figure 8-19. The two components *L* and *R* are to be united at an intermediate stage. Their direct elements have already been visited, hence, pairs (*l*, *r*) have been added to the output view when *l* is a direct element of *L* and *r* is a direct element of *R* and *l* and *r* correspond. If they do not have a correspondent, they have been added as non-combined nodes. In the

example, b_1 corresponds to b_2 and c_1 corresponds to c_2 and c_3 while *a* and *d* do not have any correspondent. Thus, the nodes *a*, (b_1, b_2), (c_1, c_2), (c_1, c_3), and *d* have been added. These and only these nodes are the direct elements of the new node *(L, R)* that represents the union of *L* and *R*. Algorithm 8-5 implements this step and is used as *handle_correspondents* in the instantiation of the generic algorithm 8-4.

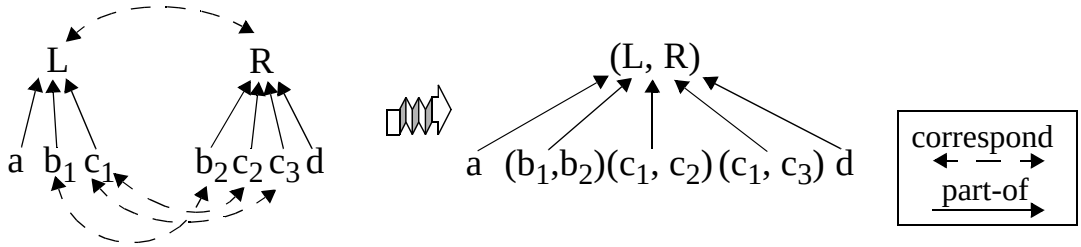


Figure 8-19. **Union for corresponding entities.**

Handling singles. The entities that do not have a correspondent have to be added to the output view in the case of the union operator. When a single is added to the result view, all its direct elements in the original view become its direct elements in the result view, too. Due to the traversal in reverse topological order, all elements of a single have already been added to the output view. Thus, all what has to be done is to enrol the direct elements of the component in the input view as its direct elements in the output view.

When direct elements are added, direct elements with and without correspondent have to be distinguished. For the former, pairs of correspondents have been added, for the latter, only plain nodes. The example in Figure 8-20 illustrates this.

Algorithm 8-6 implements the union for singles and is used as parameter *handle_left_single* to instantiate the generic algorithm 8-4. The implementation of *handle_right_single* differs from *handle_left_single* only by the order of elements of corresponding pairs (*e*, *e‘*): The first element of such a pair is assumed to be contained in the left argument view and the second element of the pair is assumed to be contained in the right argument view; i.e., within the body of *handle_right_single*, we would replace (e, e‘) by (e‘, e).

Input

- Components views V_L, V_R
- Entity L in V_L
- Entity R in V_R
- Output components view V

Algorithm

add (L, R) to output view V;
for each l ∈ direct-elements (L in V_L) **loop**
  if correspondents (l) ≠ ∅ **then**
    for each r ∈ correspondents (l) ∩ direct-elements (R in V_R) **loop**
      add part-of ((l,r), (L,R)) to output view V;
    end loop;
  else
    add part-of (l, (L,R)) to output view V;
  end if;
end loop;
for each r ∈ direct-elements (R in V_R) **where** correspondents (r) = ∅
loop
  add part-of (r, (L,R)) to output view V;
end loop;

Algorithm 8-5. ***Handle_correspondents*** **for deep union.**

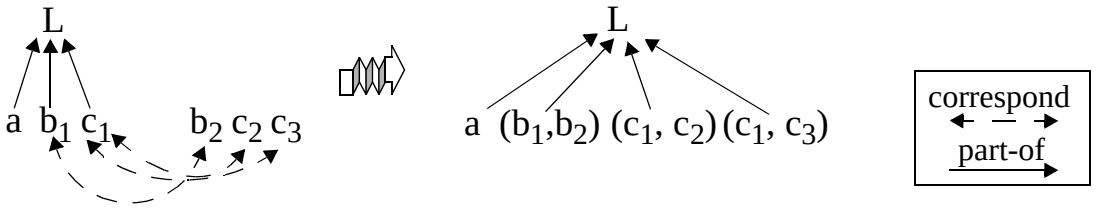


Figure 8-20. **Union for singles.**

The result of the deep union operator applied to the working example in Figure 8-18 on page 292 is shown in Figure 8-21.

Input

- Components view V_E
- Entity E in V_E
- Output components view V

Algorithm

```
add E to output view V;
for each e ∈ direct-elements (E in V_E) loop
   if correspondents (e) ≠ ∅ then
      for each e‘ ∈ correspondents (e) loop
         add part-of ((e,e‘), E) to output view V;
      end loop;
   else
      add part-of (e, E) to output view V;
   end if;
end loop;
```

Algorithm 8-6. ***Handle_left_single* for deep union.**

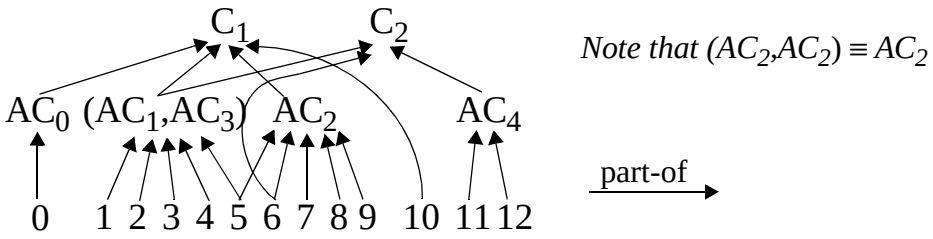


Figure 8-21. **Result of deep union.**

8.3.2.2 Deep Intersection

In the case of the union operator, correspondents and singles have to be added to the result view whereas singles have to be ignored by the intersection operator since only entities to which both techniques agree — in other words, entities with correspondent — may be in the output view of the intersection.

Handling correspondents. The parameter *handle_correspondents* for the instantiation of the generic algorithm 8-4 is a simplified version of

handle_correspondents for the union operator, in which the part that handles direct elements without correspondent is omitted (see algorithm 8-7).

Input

- Components views V_L, V_R
- Entity L in V_L
- Entity R in V_R
- Output components view V

Algorithm

add (L, R) to output view V;
for each l ∈ direct-elements (L in V_L) **loop**
  for each r ∈ correspondents (l) ∩ direct-elements (R in V_R) **loop**
    add part-of ((l,r), (L,R)) to output view V;
  end loop;
end loop;

Algorithm 8-7. ***Handle_correspondents* for deep intersection operator.**

Handling singles. The specifications of *handle_left_single* and *handle_right-single* is trivial: A node without correspondent must not be in the output view. Thus, these two subprograms do nothing.

The result of the deep intersection operator for the working example in Figure 8-18 on page 292 is shown in Figure 8-22. As one can observe, the result of the deep intersection operator is a subset of the result of the union operator as one would expect of the common union and intersection operations.

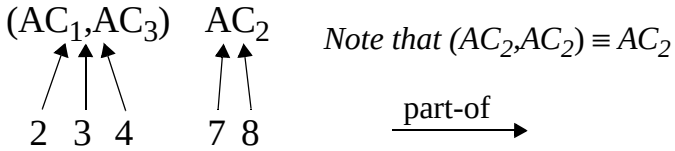


Figure 8-22. **Result of deep intersection.**

8.3.2.3 Deep Symmetric Difference

The *deep symmetric difference operator*, or simply *difference operator*, is used to investigate the discrepancy between two views. There are basically two scenarios for its usage (let V_L be the left argument view and V_R be the right argument view):

1. One wants to see what has been added to a view after an analysis has been applied to it. Then, the simple difference $V_R \backslash V_L$ of the original components view V_L to the components view V_R resulting from applying a technique to V_L is of interest. Because the incremental techniques as presented so far can only add entities to a components view and do not remove entities, V_L is a subset of V_R. In an interactive environment, the simple difference operator will be most often applied with respect to the user view.
2. One wants to compare components views generated by different techniques. In this case, differences can be additions as well as removals. Given a components view V_L to be compared to a components view V_R, $V_R \backslash V_L$ denotes the additions and $V_L \backslash V_R$ denotes the removals. Since we are interested in both $V_R \backslash V_L$ and $V_L \backslash V_R$, we are actually interested in the symmetric difference of V_L and V_R denoted by $\mathrm{V_L} \oplus \mathrm{V_R} = V_R \backslash V_L \cup V_L \backslash V_R$.

In this section, the symmetric difference for components views instead of the simple difference is introduced since the former is more general and also appropriate for the first scenario: If V_R results from an incremental technique applied to V_L, then $V_L \subseteq V_R$ holds and, hence, $\mathrm{V_L} \oplus \mathrm{V_R} = V_R \backslash V_L \cup \varnothing = V_R \backslash V_L$, which is exactly what is of interest in the first scenario.

Handling correspondents. As opposed to a shallow symmetric difference, the deep symmetric operator also yields the symmetric difference of the corresponding components. If *L* and *R* are corresponding components, the differences between *L* and *R* can be (let *L* be in the left and *R* in the right argument view):

1. a direct element of *L* has no correspondent in *R*
2. a direct element of *R* has no correspondent in *L*

Because the first argument of the difference operator is the view to which the other view is to be compared, case 1 is considered a *removal* and case 2 an *addition*. The resulting view has to contain these differences tagged accordingly. The

entities that are in both *L* and *R* — more precisely, direct elements of *L* with a correspondent in *R* — will be ignored since they do not represent a difference. Algorithm 8-8 computes the symmetric difference for two corresponding components with respect to their direct elements. Because *L* and *R* are both present in the input views, (*L*, *R*) is actually no difference between the input views. However, adding (*L*, *R*) to the output view is needed since the entities in *direct-elements*(*L*)*direct-elements*(*R*) and *direct-elements*(*R*)*direct-elements*(*L*) have to be attached to it — it is the part-of edge that makes the difference and an edge needs both source and target. In order to make explicit that (*L*, *R*) is no real difference, it is added non-tagged.

Input

- Components views V_L, V_R
- Entity L in V_L
- Entity R in V_R
- Components view V

Algorithm

add (L,R) to V non-tagged;
add each element in
$\{r \mid \text{part-of}(r, R) \in V_R \wedge \neg\exists(l)\,\text{part-of}(l, L) \in V_L \wedge \text{correspond}(l, r)\}$
to ((L,R)) in V with tag *added*;
add each element in
$\{l \mid \text{part-of}(l, L) \in V_L \wedge \neg\exists(r)\,\text{part-of}(r, R) \in V_R \wedge \text{correspond}(l, r)\}$
to ((L,R)) in V with tag *removed*;

Algorithm 8-8. ***Handle_correspondents*** **for the deep symmetric difference.**

Handling singles. Above, differences between the direct elements of correspondents were classified as removals or additions. Likewise, singles contained of the left argument view of the difference operator are considered removed and singles contained in the right argument view are considered added. In both cases, the singles and their direct elements have to be enrolled in the output view. In doing so, we again have to make a distinction between direct elements with and without correspondent since the former were added as pairs and the latter as single enti-

ties. The distinction was already explained for the deep union operator. Algorithm 8-9 implements *handle_left_single* for the deep symmetric difference. The algorithm for *handle_right_single* is analogous where nodes and edges are tagged *added* instead.

Input

- Components view V_L
- Entity L
- Output components view V

Algorithm

add L to output view V tagged as *removed*;
for each l ∈ direct-elements (L in V_L) **loop**
  if correspondents (l) ≠ ∅ **then**
    for each r ∈ correspondents (l) **loop**
      add part-of ((l,r), L) to output view V tagged as *removed*;
    end loop;
  else
    add part-of (l, L) to output view V tagged as *removed*;
  end if;
end loop;

Algorithm 8-9. ***Handle_left_single*** **for deep symmetric difference.**

The result of the deep symmetric difference operator applied to the working example in Figure 8-18 on page 292 is shown in Figure 8-23. Note that both nodes and edges need to be tagged (nodes that are present in both input views and, therefore, do not represent a difference are shown in italic font).

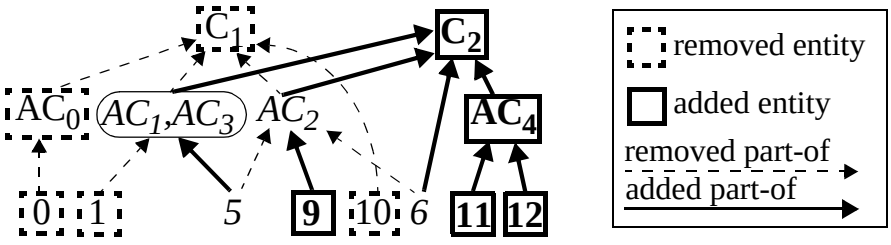


Figure 8-23. **Result of the deep symmetric difference.**

8.4 Voting Approach

In the previous sections, we have discussed combinations of techniques based on operators modeled on set operations. This section presents an alternative approach for combining the techniques.

When a yet free entity it to be bound during clustering, the question arises to which atomic component this entity should be added. So far, this has been answered by running one of the base techniques separately. However, what if we do not want to rely on a single heuristic? Then, we could run several analyses, compare the results, and add the entity to the atomic component to which most techniques agree. Agreement among techniques can be established by means of the intersection operator. The intersection operator is defined for two arguments but could easily be extended to an arbitrary number of components views. However, the likelihood that the resulting components view is empty increases by the number of techniques that have to agree because *all* techniques have to subscribe. A more practical approach when several techniques are considered at the same time is to accept atomic components when a certain number of techniques agree but not necessarily all. Furthermore, the agreement of a technique is only binary for the intersection operator though the actual degree of certainty of the technique may lay somewhere in between 0 and 1.

8.4.1 Summarized Agreement

Dropping total agreement and allowing continuous degrees of agreements between 0 and 1 lead us to the so-called **voting approach**: Given a base entity E that is to be added to one of the atomic components in $SAC = \{AC_1, \ldots, AC_n\}$ and a set of techniques $ST = \{T_1, \ldots, T_m\}$, E is added to the atomic component $AC_j \in SAC$ for which the agreement among techniques of ST is maximal, i.e., quantitatively,

$$\text{total-agreement}(E, AC_j) = \left(\sum_{i=1}^{m} x_{T_i} \times \text{agreement}_{T_i}(E, AC_j)\right) / \sum_{i=1}^{m} x_{T_i} \qquad \textbf{(8.2)}$$

has to be maximal, where x_T is the weight of technique T used to give more trusted techniques more influence and $agreement_T\ (E, AC)$ is the individual agreement of technique T that E should be added to AC.

The allowable range of values for the individual agreement is between -1 and 1, where values in the range of -1 and 0 express varying degrees of disagreement and values greater than 0 are considered agreements. However, as it will be discussed in the following sections, the actual range of agreement of most techniques is between 0 and 1. This is the case when the underlying clustering criterion is inherently positive, i.e., when the clustering criterion does not specify attributes to *exclude* entities.

Because the individual agreements are normalized to yield a value between -1 and 1, *total-agreement* is also normalized. Normalization simplifies the application of the voting approach for the user. Adjusting the weights x_T, i.e., the influence of individual techniques, is easier when the range of agreement of a technique is known to be between -1 and 1. Furthermore, the *total-agreement* is easier to interpret because it is a relative value between -1 and 1, whereas the left and right limits of *total-agreement* would be unspecified if *total-agreement* were not normalized.

The individual agreement is specific to each technique. Before the individual agreement is defined for each technique, different applications of the voting approach are discussed. The agreement of some techniques is not applicable to all possible usage scenarios of the voting approach. The following sections will point out techniques not applicable to a certain kind of application.

8.4.2 Ways of Using the Voting Approach

Before we go into detail of defining the level of agreement for each technique, we discuss some usages of the voting approach. There are basically four kinds of scenarios in which the voting approach is helpful:

- **single entity assignment**: one entity it to be added to *existing* atomic components
- **multiple entities assignment**: several entities are to be added to *existing* atomic components
- **clustering**: free entities are to be grouped to new or existing atomic components
- **assessment**: existing atomic components are to be assessed

These scenarios will be discussed in the following.

Single entity assignment. The voting approach was already motivated as an alternative to the intersection operator for determining the corresponding atomic component of a single free entity when several techniques are to be consulted. The entity is added to the atomic components for which the *total-agreement* of all techniques is maximal. This kind of usage of the voting procedure is useful during clustering, but a similar situation arises when the maintainer implements a new function of the system after the atomic components have been detected: The voting techniques could then be used to determine to which atomic component — or module, respectively — the function should be assigned (this is known as the orphan problem; Tzerpos and Holt, 1997). N.B.: *Same Module* cannot be polled because the new function has no module yet.

Multiple entities assignment. Assignment of several entities to existing atomic components is a straightforward extension of the latter scenario. However, one has to be aware that the assignment depends on the order in which the entities are handled, i.e., the addition of an entity to an atomic component, *AC*, can influence the agreement for another entity with respect to *AC* — the agreement can increase or decrease. That is why the entities should be assigned in the order of the current agreement as described by algorithm 8-10. This way, the voting procedure can be used in a clean-up phase when the atomic components have been detected to a large extent and the remaining free entities are to be added to the found atomic components.

Clustering. In the previous two scenarios, entities were to be added to existing atomic components. When new atomic components are to be created, we can use the voting procedure as a regular clustering method by using *total-agreement* as defined by equation (8.2) as a similarity metric for the clustering algorithm described for *Similarity Clustering*.

Total-agreement was defined above with respect to an entity and an atomic component. However, in the beginning, when no atomic components are present, one can treat each entity as an atomic component of its own.

Input:

- set S of free entities
- set SAC of atomic components

Algorithm

while $S \neq \emptyset$ **loop**

ascertain $E \in S$, $AC \in SAC$ **where**

$\forall E' \in S, AC' \in SAC$: total-agreement(E', AC') $\leq$ total-agreement(E, AC)

add E to AC

$S := S \setminus \{E\}$

end loop;

Algorithm 8-10. **Multiple entities assignment.**

Assessment. Techniques may yield a large number of candidate components. In an interactive approach, it is useful to rank these candidates when presenting them to the user. The user can then validate the most promising candidates first. The voting procedure can be used to rate these components. It makes sense to include the vote of the technique that has produced the candidates, too, since the technique need not necessarily be entirely convinced, hence, its agreement may be below 1.

Furthermore, the voting procedure can also be used to assess an existing decomposition of a system into files. It can be applied under the assumption that each file constitutes an atomic component. This way, those modules may be identified that need to be restructured and atomic component detection can be applied more goal-oriented. The assessment of the voting procedure can then also be used to detect the mavericks that cause low cohesion of some modules. N.B.: It is clear that *Same Module* cannot be used for an assessment of existing modules.

In order to identify mavericks of a module, we can use equation (8.2) directly by iterating over the elements of an atomic component and computing the *total-agreement* to the chosen entity, *E,* with respect to its atomic component, *AC,* (excluding the element) and to all other atomic components. The element *E* is a **maverick** when the entity should actually be in a different atomic component, i.e., if there is an atomic component *AC*' for which

total-agreement (*E*, *AC*') > *total-agreement* (*E*, *AC*-{*E*})

In order to rate an atomic component as a whole, one can look at the absolute and relative number of mavericks (with respect to the number of elements in the atomic component) as well as compute the average on *total-agreement* over all elements in the atomic component *AC*:

$$\text{average-agreement}(AC) = \left(\sum_{E \in AC} \text{total-agreement}(E, AC - \{E\}) \right) / |AC|$$

Modules with a high absolute and relative number of mavericks or a low *average-agreement* may represent candidates for restructuring in preventive maintenance. On the other hand, atomic component candidates with a low absolute and relative number of mavericks or a high *average-agreement* should be validated first in an interactive atomic component detection process.

8.4.3 Agreement of Individual Techniques

The definition of *total-agreement* according to equation (8.2) is based on the individual agreement of the basic techniques. In this section, this agreement will be defined for the connection-based, metric-based, and graph-based techniques listed in Chapter 5.

The agreement has to be defined as a function of an entity and an atomic component. It has to express the degree of certainty of the technique that the entity belongs to this atomic component. That is, the agreement is high if and only if the technique would add the entity to the atomic component during clustering. Therefore, the following definitions of agreement are modeled on the actual action of the techniques when they would have to cluster this element. In other words, the definitions reflect one single clustering step.

In order to obey the information that has been contributed by the user, there is an overriding rule for the definitions following in the next sections: If the agreement for an entity, *E*, and an atomic component, *AC*, is to be computed, the agreement is -1 if there is an entity, *E*', in *AC* where *E* and *E*' are mutually exclusive.

8.4.3.1 Agreement of Connection-based Techniques

The principal procedure of clustering is the same for all connection-based techniques and can be described by the generic algorithm 8-1 on page 268. These techniques differ only in the actual specification of the generic function parameter *Connected_Entities* of the algorithm that yields all the connected entities to which a given entity is to be grouped. That is, the definition of agreement for all connection-based techniques can be uniformly made based on the abstract function *Connected_Entities*.

Among the elements yielded by *Connected_Entities* to be grouped with the entity may be bound and free entities. We are going to ignore free entities in the following definition of agreement because the question for the voting approach is to which existing atomic component a given entity belongs. However, if free entities must not be ignored, e.g., for general clustering, we can consider each free entity as an atomic component of its own. The following function is a filter for free entities:

$$\delta(S) = \{s | s \in S \wedge \text{Is\_Bound}(s)\}$$

Applying this filter to the result of *Connected_Entities* leaves the bound entities. As described by algorithm 8-1 on page 268, the incremental connection-based approaches add the entity to the enclosing atomic component of the bound entities. That is, it is sufficient for an atomic component to have one single entity that is related to the entity under consideration (in the sense of the heuristic) to be a candidate for receiving this free entity. This rule is adequate for the incremental techniques since their results are intended to be presented to and validated by the user. The user is then the final judge sifting all possible alternatives. However, in this section, we have to define a finer gradation of the agreements of the techniques than just *yes* or *no*. The definition of agreement will, therefore, be based on the number of entities of an atomic component that are a reason for the given entity to be added to the atomic component. It is the fraction of entities of the atomic component connected to the entity under consideration, relative to all connected entities. More precisely,

$$\text{agreement}(E, AC) = \frac{|\{x | x \in \delta(\text{Connected\_Entities}(E)) \cap \text{direct-elements}(AC)\}|}{|\delta(\text{Connected\_Entities}(E))|} \quad \textbf{(8.3)}$$

where δ(*Connected_Entities* (*E*)) are all bound entities to which *E* can be grouped.

So far, the actual functions used as *Connected_Entities* for the specific techniques have been defined for subprograms only. However, the entity could also be a variable or type. In these cases, we simply use the relational inverse of the original function for *Connected_Entities*.

Example. Consider the example in Figure 8-24. *Connected_Entities (E)* = {x_1, x_2, x_4, x_5, x_6} where x_1 and x_2 are both free, hence, δ(*Connected_Entities (E)*) = {x_4, x_5, x_6}. Because AC_1 has two elements to which *E* is connected, the *agreement (E, AC_1)* is 2/3, while AC_2 has only one connected entity and whose *agreement (E, AC_2)* is, therefore, only 1/3.

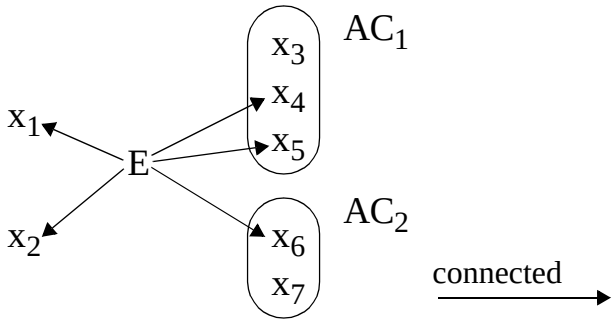


Figure 8-24. **Connected entities example.**

Note that the definition of agreement is only relative to the number of connected entities and does not depend on the size of the involved atomic components. However, the more elements an atomic component has, the likelier it is that an atomic component contains many connected entities. In other words, since a component with *N* elements cannot have more than *N* connected entities, smaller components are at a disadvantage. This is undesirable for clustering because it leads to few large components. The definition of agreement above was made from the perspective of the entity: It expresses the connection of the entity to the atomic component relative to all other components to which the entity could belong. For clustering, we can flip over the perspectives and define the agreement as the extent to which the entity fits into the atomic component. This can be

achieved by defining it relatively to the number of elements of the atomic component as follows:

$$\text{agreement}(E, AC) = \frac{|\{x | x \in AC \wedge x \in \delta(\text{Connected\_Entities}(E))\}|}{|\text{elements}(AC)|} \quad \textbf{(8.4)}$$

For clustering and assessment, we will prefer equation (8.4), for entity assignment, however, we will prefer (8.3).

8.4.3.2 Agreement of Metric-based Techniques

For the definition of agreement for metric-based approaches, we can use their metrics. However, *Delta IC* is an exception. As it was already discussed in Section 5.13, *Delta IC* is a hybrid of strictly connection-based and metric-based approaches.

Delta IC. *Delta IC* consists of two steps: cluster formation and cluster filtering. The metric is only used for filtering. Moreover, the metric is defined for subprograms only and has other disadvantages (see Section 5.7). Two other metrics, namely, *Internal* and *External Connectivity*, have been proposed to overcome the restrictions of the *Delta IC* metric. (The definition of agreement for *Internal* and *External Connectivity* follows below.) Due to the restrictions of *Delta IC* and the fact that *Internal* and *External Connectivity* are too similar to the *Delta IC* metric — and hence, similar characteristics would enter *total-agreement* twice — the agreement of *Delta IC* will be based on its primary criterion for cluster formation.

The actual cluster comprises the closely related subprograms and the referred entities of a subprogram (see equation (5.8) on page 125):

$$\text{candidate-cluster (S)} = \text{closely-related-subprograms (S)} \cup \text{referred-by (S)}$$

Therefore, given a subprogram, S, we can define the agreement of *Delta IC* analogously to agreement of connection-based techniques as the fraction of entities of the atomic component that are also in the cluster:

$$\text{agreement}(S, AC) = \frac{|\{x | x \in AC \wedge x \in \delta(\text{cluster}(S))\}|}{|\delta(\text{cluster}(S))|}$$

However, what if the entity is not a subprogram but a type or object? For the connection-based approaches, we used the relational inverse of *Connected_Entities*. A similar approach can be used for *Delta IC*. The relational inverse of *referred-by* is *refer-to*. *Closely-related-subprograms*, on the other hand, was defined as follows:

$$\text{closely-related-subprograms}(S) = \bigcup_{e \in \text{referred-by}(S)} \{F \mid F \in \text{refer-to}(e) \wedge \text{referred-by}(F) \subseteq \text{referred-by}(S)\}$$

An inverse analogon for types and objects is:

$$\text{closely-related-entities}(E) = \bigcup_{s \in \text{refer-to}(E)} \{e \mid e \in \text{referred-by}(s) \wedge \text{refer-to}(e) \subseteq \text{refer-to}(E)\}$$

A cluster for a type or an object, *E*, can then be defined as:

$$\text{candidate-cluster}(E) = \text{closely-related-entities}(E) \cup \text{refer-to}(E)$$

Hence, we can use *candidate-cluster* as *Connected_Entities* in (8.3) and (8.4), respectively, to specify the agreement of *Delta-IC*.

Internal and external connectivity. Internal and external connectivity are two different aspects; this is why agreement is defined for each one separately in this section rather than defining a combined agreement in terms of connectivity as defined by (5.15) on page 137.

Internal and external connectivity are defined for a given atomic component whereas the agreement is based on an atomic component and an entity. However, we can use them for the definition of agreement for internal and external connectivity, respectively, by assuming the given entity were part of the atomic component and measure the difference of connectivity with and without the entity as follows:

$$\text{agreement}(E, A) = \Delta IntC(E, A) = IntC(A \cup \{E\}) - IntC(A) \qquad \textbf{(8.5)}$$

$$\text{agreement}(E, A) = \Delta ExtC(E, A) = ExtC(A) - ExtC(A \cup \{E\}) \quad \textbf{(8.6)}$$

Equation (8.5) is used for the agreement based on internal connectivity and equation (8.6) is used for the agreement based on external connectivity. Note the two different orders in the differences (8.5) and (8.6). The definition of $\Delta IntC$ is aimed at rewarding an increase in internal connectivity, whereas the definition of $\Delta ExtC$ promotes a decrease of external connectivity. Furthermore, the value of internal and external connectivity as defined by (5.13) and (5.14) on page 137, respectively, is between 0 and 1. Hence, the differences (8.5) and (8.6) are in the range of -1 and 1.

Type-Based Cohesion, Schwanke's Arch Approach, and Similarity Clustering. The agreement of the similarity-based techniques *Type-based Cohesion*, Schwanke's *Arch* approach, and *Similarity Clustering* can be based on the underlying metric of these techniques. In the case of Schwanke's *Arch* approach, and *Similarity Clustering*, the group similarities as defined by (5.18) on page 141 and (7.2) on page 188, respectively, can be used treating the entity to be compared to the component as a group of its own. This compares to a real clustering step when the elements of A have been grouped together and the similarity of A and the group that contains the single entity is to be re-computed. More precisely, given a component, A, and an entity, E, the agreement can be defined as follows:

$$\text{agreement}(E, A) = GSim(\text{direct-elements}(A), \{E\})$$

The group similarity for *Type-based Cohesion* as defined by (5.20) on page 146 cannot be used to define the agreement of *Type-based Cohesion* because the group similarity unites the two groups to be compared and computes the average similarity over all pairs of the union. Hence, the similarity among the elements that were already in component A to which the entity E is to be compared would also contribute to the agreement. As a consequence, if this group similarity were used and the component A as such has already a high agreement by *Type-based Cohesion*, the resulting agreement of *Type-based Cohesion* for $A \cup \{E\}$ would also still be high even if E has nothing to do with A. That is to say, the agreement should measure the similarity of E to all other elements in A but not the similarity among elements of A. In the course of this thesis, two alternative group similarities have been introduced. Schwanke proposed to use the maximal similarity between the two groups whereas *Similarity Clustering* uses the average similarity

between elements of different groups. Both of these group similarities can also be used for the agreement of *Type-based Cohesion*. However, because the group similarity based on the maximal individual similarities was less effective in practice, the average group similarity is used instead. Hence, the following definition of agreement for *Type-based Cohesion* is used (the formula is simplified leveraging that $\{E\}$ contains only one element):

$$\text{agreement}(E, A) = \frac{1}{|\text{elements}(A)|} \times \sum_{e \in \text{ elements}(A)} Sim(E, e)$$

Sim is the similarity between two base entities specific to *Type-based Cohesion* and is defined by (5.19) on page 146.

8.4.3.3 Agreement of Graph-Based Techniques

Cycles in the call view are rare and most constituents of an atomic component are not dominated by the atomic component. Moreover, the atomic component must be known to a large extent before *Dominance Analysis* can be applied in the first place. Therefore, *Strongly Connected Component Analysis* is commonly used only in the beginning of the detection process whereas *Dominance Analysis* is preferably used at the end of the process. That is, these two graph-based techniques play a minor role in the actual clustering process (other than in a preparation and a clean-up phase) in which the voting approach has its place. Nevertheless, their agreement may be an additional piece of information and is, therefore, defined here.

As discussed in Section 8.3.1.5, *Strongly Connected Components Analysis* clusters all entities of a cycle. Technically, this is achieved by using the generic algorithm 8-1 on page 268 for connection-based approaches where the entities in a cycle are returned by the actual parameter for *Connected_Entities*. Likewise, we can reduce the definition of the agreement of *Strongly Connected Components Analysis* to the agreement for connection-based approaches as defined by equation (8.3) on page 306 for entity assignment or equation (8.4) when assessment or clustering is required. Function *Connected_Entities* therein is the set of entities within a cycle as defined by (8.1) on page 277. Consider Figure 8-25 as an example. Given the entity, *E*, for which the agreement is to be ascertained, the cycle containing *E* consists of *a*, *b*, *c*, and *E*. According to the definition (8.1) of

Connected_Entities, the connected entities of *E* are *a*, *b*, and *c*. Entity *a* is part of A_1 and the other two entities *b* and *c* are part of A_2. Thus, *agreement* (E, A_1) = 1/3 and *agreement* (E, A_2) = 2/3 according to (8.3).

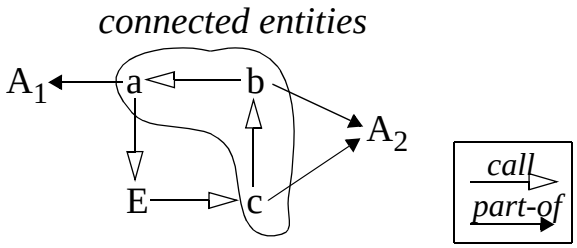


Figure 8-25. **Example for *Strongly Connected Components*.**

Dominance Analysis adds an entity to its primarily dominating atomic component (see Section 5.12). If the primarily dominating atomic component exists, it is always unique. Hence, as opposed to the other approaches, there is maximally one atomic component to which the entity is added. Therefore, the agreement of *Dominance Analysis* is binary, either 1 if the given atomic component is the primarily dominating atomic component or 0 otherwise.

8.5 Summary

In this section, high-level operators were introduced that allow manifold combinations of the basic techniques in a flexible way. These operators are preferred to a technical integration of the diverse heuristics since integration of new techniques in the operator framework is straightforward. These operators are especially suited for combinations triggered by a user and are, therefore, used in the semi-automatic method presented in the next chapter. In order to support compositions of techniques, the basic techniques have been extended to work incrementally.

This section also proposed a voting approach that allows to combine the heuristics on the basis of their agreement overcoming the shortcomings of the intersection operator. Though the intersection operator could principally also be used to establish agreement among techniques, it is too strict when more than two heuris-

tics are to be combined. The voting approach can be used for entity assignment, clustering, and assessment. When a new technique is added to the atomic component detection framework, its heuristic must be cast into a metric that reflects the agreement of the technique that a given entity belongs to a certain atomic component. This agreement is a value between -1 and 1.

The basic techniques introduced in Chapter 5 and Chapter 7 were evaluated in Chapter 6 and Section 7.6.2.3, respectively. The combinations of the basic techniques as described in this chapter are not evaluated with respect to the evaluation framework of Chapter 6 because first, there is an infinite number of possible combinations and, second, these ways of combinations are primarily thought as being used interactively in order to support the semi-automatic method described in the next chapter. Chapter 10 describes experiments conducted to evaluate the semi-automatic method that will also partly allow conclusions for the combined techniques.

Chapter 9 *A Semi-Automatic Method to Detect Components*

The techniques introduced in Chapter 5 are all fully automatic which is desirable especially for large systems. However, their evaluation revealed that none of them has the detection quality that compares to human detection. There are basically two ways to improve the detection quality process. We can search for more sophisticated techniques or include the user in the detection process. The purpose of this thesis was to see to which extent structural information can be leveraged. Future research toward using data flow information and domain knowledge may produce more powerful techniques. However, even then, the user will remain the final judge. Due to the complexity, vagueness, and to some degree subjectivity, it is questionable whether we can ever find precise techniques that fit all cases. Therefore, atomic component recovery is a problem that has to be tackled in concert with a maintainer at any rate.

This chapter describes a method in which human and computer interact to detect atomic components. It depicts how the process can be split into tasks assigned to either the computer or the maintainer. The outcome of each task carried out by one of the two partners, human or computer, is used for the other partner’s next task.

9.1 Method Overview

This section presents an overview of the method with the main tasks of both computer and human and the way and order of their interaction. A detailed description of the individual parts will follow after this overview.

Figure 9-1 contains the main steps of the method. The inner cycle, consisting of analysis application, metric ranking, presentation, and bookkeeping of detected atomic components, is the core of the detection process. The user controls the detection process by selecting analyses and metrics and by validating the candidates proposed by the automatic techniques. The task of the computer comprises the automatic analyses, computation of the metrics for the proposed candidates, presentation of the results, and bookkeeping of the user decisions.

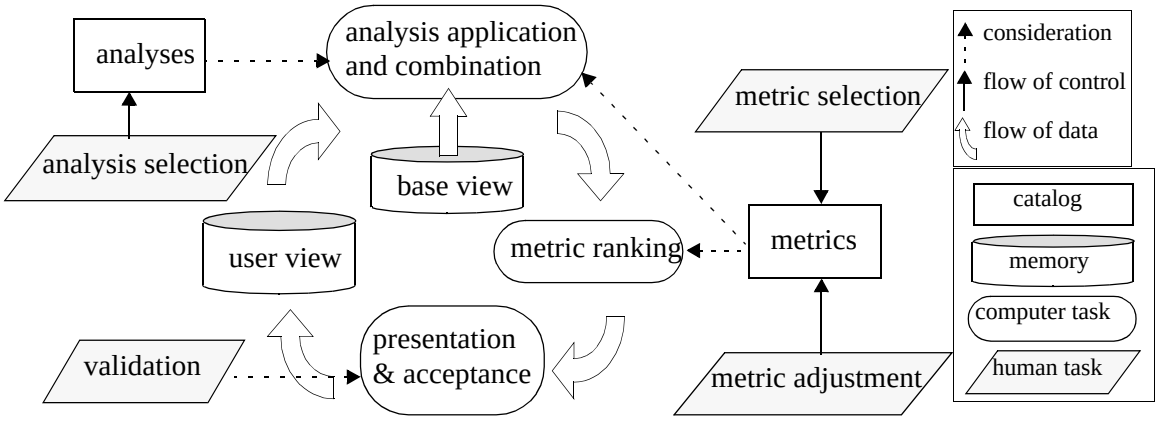


Figure 9-1. **Semi-automatic method for atomic component detection.**

The base view contains the base entities and their relationships needed for component detection and is automatically derived from source code. The so-called user view logs the information contributed by the user. It records the atomic components that have been detected and confirmed so far. In the beginning, when no atomic component is known, the user view is empty. The user selects an analysis that is to be applied. The analysis takes into consideration the components that were confirmed by the user (in the first iteration there are none). Thus, the analyses are applied incrementally. Chapter 8 already discussed how the analyses introduced in Chapter 5 can be modified to work incrementally by clustering only those base entities that have not been clustered before and by forming new atomic

components or adding free base entities to existing atomic components. Generally, the techniques propose many candidates. The user should not be swamped with all of them. Instead, the candidates should be presented in their presumed quality. Metric ranking is supported by letting the user select and adjust certain metrics. After the candidates have been ranked by user-selected metrics, the candidates are presented to the user for acceptance. The presentation is a crucial and non-trivial task. It must be in such a way that the user's validation can be as quick as possible. Additional information the user may need has to be provided on demand. For example, the maintainer will probably also want to inspect the source code. The user validates the candidates and those atomic components he or she accepts enter the atomic component memory, i.e., the user view.

In each iteration, the user selects and combines different analyses to find components that could not be found by previous analyses. The process ends when the found components are sufficient for the task at hand or no further component can be found anymore. The user does not have to select, apply, and validate only one analysis at a time. Instead, several analyses can be selected and applied in parallel. Then, the intersection, union, and differences of these analyses can automatically be ascertained and the user can investigate and validate these. Particular large candidates of some techniques can be refined by applying other techniques to these individually.

Because the typical maintenance task usually does not require to find all atomic components of a system but only a few relevant ones in a specific part of the system, the domain of search can be restricted to certain modules.

We are using Rigi for presentation and interaction with the user (Müller, 1994). Rigi is a customizable graph editor developed to support reverse engineering and offers many useful capabilities such as:

- support for annotateable nodes and edges of different types
- hierarchical nodes and views
- direct linkage to the correspondent source code by clicking on nodes
- automatic layout and context-preserving browsing capabilities
- filter and selection mechanisms
- Rigi command language for customizations

Rigi was extended in many directions to adapt it to our needs. The adaptations were opportunistic; not everything what might have been useful could be worked into Rigi, e.g., an undo mechanism would have been helpful. But all of our major requirements were more or less easy to fulfill with Rigi.

The following sections go into more detail of the individual steps of the method.

9.2 Analysis Selection and Application

Currently, all connection-based, metric-based, and graph-based techniques listed in Chapter 5 and Chapter 7 are supported. The user selects a subset of the basic analyses and specifies the order in which they should be applied. This is necessary since for many basic analyses, the order of composition is relevant.

The analysis takes into account the atomic components that have been detected so far, i.e., that have been confirmed by the user. Chapter 8 anticipated what kind of information the user can add: **Positive information** expresses that a base entity (variable, type, or subprogram) belongs to a given atomic component. **Negative information** conveys that two entities do not belong together. Every analysis must preserve all positive information, that is to say, an analysis may only add to the components a user has confirmed and never remove any of their elements, and likewise, an analysis must not cluster entities that were not supposed to be grouped together.

The metric-based approaches use a metric to measure the similarity of entities that are to be grouped together. These metrics can be used for both the assessment and clustering of candidates. That is why the analysis application in Figure 9-1 is not only controlled by the selected analysis but also by the used metrics and their actual parameter settings.

The analyses can be selected, combined, and started from within Rigi by means of list boxes and menus. It was important to us that the selection and combination is easy to do with simple mouse clicks such that the user need not learn a complex language. The result of an analysis is represented by a single hierarchical analysis

node that is the root of the actual candidates. This makes it possible to further process the results of an analysis by direct manipulation. For example, the difference to the currently accepted atomic components can be shown, it can be intersected or united with the results of other analyses (deep intersection, deep union), or the next kind of analysis can be applied to it (composition). For the interactive approach, a variant of the composition operator is useful that can be applied to individual components. The so-called **individual composition** is a form of composition in which only the elements of a single component may be clustered.

9.3 Metric Selection, Adjustment, and Ranking

Metrics are used to assess and rank the candidates that have been proposed by the analysis. There is a catalog of metrics that the user can choose of. The catalog comprises the metrics of the metric-based approaches and metrics that express the underlying heuristics of the other non-metric-based techniques as described in Section 8.4 for the voting approach. Established intra-modular and inter-modular metrics, such as number of lines of code, McCabe or Shepperd complexity (Fenton and Pfleeger, 1997), could also be integrated to measure additional aspects of the candidates but were not implemented for the current prototype. The metric used for ranking is a composite metric that is the normalized weighted sum over the individual metrics of the techniques.

The composite metric is used to guide the user through the large set of candidates. The metric is computed once and then a threshold is used to control the presentation. All candidates above the threshold come to the fore. The next section will discuss how this can actually be done. Independent of the way of presentation, we would start with a high threshold that is decreased step by step. In each step, the candidates above the chosen threshold are validated by the maintainer. Once the elements of a candidate have been accepted, they are not clustered again since the user has already decided where they belong to.

Some metrics have parameters that need to be adjusted. Altogether, there are, hence, three dimensions of variability:

- the influence factors for the basic metrics within the composite metric,

- the inherent parameters of the basic metrics,
- and the filtering threshold.

These parameters can be adjusted by the user and the presentation updated accordingly. Several distinct metric settings can be tried without need to rerun the automatic clustering; only the metrics have to be re-computed.

9.4 Presentation, Validation, and Acceptance

For the presentation of the results of automatic techniques, we are using the means offered by Rigi, mainly flat and hierarchical perspectives of the component decomposition. Base entities and components are expressed as nodes, their relationships as edges. We are using a special node type, so-called **analysis node**, to represent the results of an analysis. Introducing an analysis node gives the user a handle for direct manipulation of analysis results. For example, the user can select two analysis nodes and apply the intersection operator to them.

The *analysis node* can be unfolded. Then, the actual subsystem and atomic component candidates are shown. The user can browse these candidates by clicking on the nodes or viewing the node hierarchy as a whole. The node hierarchy is especially interesting when the results of *Similarity Clustering* or *Type-Based Cohesion* are viewed. These two clustering approaches return a tree that indicates the order in which elements were grouped together and, therefore, immediately show what is more similar and what is less. The maintainer can then “climb up the tree” starting at the leaves and stop at any inner node for which the combination is doubtful. Direct validation is possible in any view.

The composite metric mentioned in the last section can be used for the purpose of presentation of the candidates either to filter candidates below a threshold or to emphasized candidates above the threshold. An alternative way of presentation would be to color the nodes or to set the size of the nodes according to their metric value. This way, the user could see all candidates at once and yet immediately find the most promising candidates. Unfortunately, this would have been more difficult to achieve with Rigi.

The returned candidates can be accepted or rejected individually or as a whole by direct manipulation. The atomic components can be renamed by the user to give them a meaningful name. Single base entities within candidates can be accepted or rejected. Cut and paste capabilities are available to move single or whole sets of base entities as a group at once from one atomic component to the other. Any base entity can be added to a candidate. The maintainer is also able to create her own atomic components.

Everything confirmed by the user is moved in the user view. The user view, basically, is represented by the same means as any other analysis view such that most commands available for analysis results are also available for the user view in a uniform manner. Only those that make no sense for the user view were excluded, such as accepting nodes or viewing the difference to the user view.

9.5 Detection Strategy

The framework described in the previous sections has many degrees of freedom. Therefore, some guidelines should be given on how to use it successfully. The recommended strategy for component recovery consists of two main parts: Detection of atomic components and then identifying the relationships among the atomic components.

Component detection. Detection of atomic components can be done in the following phases (each phase may consist of several iterations):

1. Apply all connection-based analyses and *Strongly Connected Base Component Analysis* in parallel. Use deep union, deep intersection, and composition to combine the techniques (for reasonable combinations see below). If you want to poll the agreement of more than two techniques, use the voting approach instead of the intersection (the result may be empty otherwise). Add negative information during validation when you find entities that should not be grouped together.
   Rationale: One gets only few promising candidates in the beginning which can form points of crystallization for the subsequent analyses. The added mutually exclusive information will break up larger candidates in the subsequent runs of the analyses.

2. Apply connection-based approaches once again but one at a time. This time, they will leverage the information the user has contributed and return different clusters (containing no entities that have already been grouped and no mutually exclusive entities). Validate the non-intersected results (thus, using less strict criteria). If particular large candidates occur, refine them with other techniques by means of intersection or individual composition. If the results of one technique have been validated, run the next analysis.
   Rationale: The crystallization points of the first step are extended and new atomic components are built that were dropped out by the intersection in the first step. Applying the techniques successively guarantees that all validation information is respected by the analyses.
3. The metric-based approaches are associated with parameters whose values may not be known in advance. Fortunately, the previous two steps lead to a set of components that can be used to calibrate the metric-based approaches. Varying this calibration reveals further clusters. In the case of *Similarity Clustering*, the parameters should be set specific to one kind of atomic component. For example, if abstract data types are to be detected, one gives *signature* and *local-obj-of-type* relationships more and *variable reference* relationships less weight. If a hierarchical clustering is used, one starts the validation of the results at the leaves and then climbs up the tree toward the root until a metric value is reached that does not indicate sufficient confidence in the candidate anymore.
   Rationale: The connection-based techniques are based on fixed patterns and, therefore, will always yield the same candidates. The metric-based techniques allow more variability by changing their parameters.
4. Finally, *Dominance Analysis* can be applied to find local utility functions of atomic components.
   Rationale: The dominance analysis for atomic components can only be applied when the atomic components exist in the first place. Hence, it can only be used late in the detection process. *Dominance Analysis* will detect the local entities that might not be detectable by other approaches. For example, a subprogram that provides a special service for one abstract data type only need not have any recognizable relation to the type (signature relation or a local variable of this type) other than being called by the functions of this abstract data type. Therefore, none of the connection-based techniques will cluster it to the abstract data type since they all ignore calls. Likewise, the chances that *Similarity Clustering*

finds it are low since the calling relationship alone is usually not very significant.

Combining components views. In the first phase of the method above, the diverse techniques are combined by means of the combining operators described in Section 8.3, namely, restriction, composition, deep intersection, and deep union.

It is often helpful to restrict the search to one kind of atomic component at a time because the search criteria are mostly different. For this reason, it is possible to restrict all the analyses to a particular set of entity types. For example, if one searches for abstract data types, global variables can be ignored.

If a technique was restricted to certain entity types, entities of other types may be left. Then, another technique can be applied in a composition to the result of the previous technique in order to cluster free entities. Composition can also be used to refine the results of possibly too large components of one technique by another technique.

When the intersection is applied to components views generated by techniques that propose very distinct atomic components, the resulting components view is likely to be empty. This is, in particular, the case when techniques are combined that consider different kinds of entities. By a look at Table 9-1, which summarizes what kind of base entities are considered by the respective base technique, one can quickly decide which intersections do make sense. A ✓ in Table 9-1 means that a certain kind of base entity is considered, ⊥ means that it is not considered. A sensible intersection can be expected of those techniques that consider a common set of base entity kinds.

Interestingly enough, even if the results of techniques that are not compatible in the sense of Table 9-1 are intersected, the results need not necessarily be empty. All techniques that have been introduced so far consider at least subprograms. So, even when *Delta IC*, which considers variables and subprograms, is intersected with *Part Type*, which considers types and subprograms, a few groups of subprograms may remain. The result may, however, not be very useful because the reason why the subprograms were brought together is not clear anymore.

Table 9-1. **Domains of the basic techniques.**

Technique	Subprograms	Variables	Types
Global Variable Reference	✓	✓	⊥
Same Module	✓	✓	✓
Part Type	✓	⊥	✓
Same Expression	✓	✓	⊥
Internal Access	✓	✓	✓
Delta IC	✓	✓	⊥
Similarity Clustering	✓	✓	✓
Type-Based Cohesion	✓	⊥	⊥
Strongly Connected Components	✓	⊥	⊥
Dominance Analysis	✓	✓	⊥

The deep union operator is useful to join together the results of two different techniques for further processing by a third analysis. The union operator may produce overlapping components (the intersection and composition do not yield overlapping atomic components other than those already produced by the applied techniques). Overlapping candidates are a problem when presented to a maintainer for validation because all overlapping candidates have to be investigated to decide where a given entity (in the overlapping part) belongs to. In the case of non-overlapping components, the maintainer can simply accept or reject the entity at hand. However, this is only a problem when the final result contains overlapping components. For intermediate results during combination, overlapping components are useful. This way, several alternative candidates can be investigated in parallel until a decision is made in the course of combination.

Detection of relations among components. Once the atomic components have been detected, their relationships can be analyzed by applying *Strongly Connected Component Analysis* and *Dominance Analysis* to the base view in which the atomic components are collapsed (see Section 8.3.1.5). *Strongly Connected Component Analysis* yields sets of atomic components that mutually depend on each other and *Dominance Analysis* reveals whether atomic components are local to each other.

The recovered components are documented by the maintainer and saved for future maintenance. They can be used to explain the system at a higher level of abstraction above the code level and are candidates for reuse. Ideally, the module decomposition of the system will be restructured to conform to the atomic component structure, i.e., a module contains exactly one atomic component. If this is not immediately possible, the programmer should regard the system at the atomic components views rather than at the module view (see Table 3-5 on page 68). The source location of each base entity can be used as a mapping from the components view to the module view.

If the system is changed during further maintenance, the captured decomposition of the system into components has to be adapted. If entities are removed from a component, the voting approach can be used to analyze whether the system should be restructured. If an entity is added, the voting approach can be used to give hints to which atomic component or module, respectively, the entity should be assigned (see Section 8.4).

9.6 Extensibility of the Framework

The framework supporting the semi-automatic method can be extended in many ways:

- The way how combination of techniques is organized allows for quick addition of new analyses.
- Further metrics can be brought in to rank the candidates, in particular, traditional intra-modular software engineering metrics, such as lines of codes, complexity and information flow metrics.
- The way how results are presented to the user can be changed independently from the analyses. Coloring the candidates according to their metric ranking is just one example.
- Multiple user views could be enabled instead of the view of one single maintainer in order to collect the components for large systems by the joint work of many maintainers. From the point of view of one maintainer, the user view of other maintainers is like any other analysis. The combining operators can then be used to reach a consensus among the maintainers or to analyze divergences.

Or the different views may indeed be used as different perspectives on the systems for different programmer groups.

The method could also be applied with no automatic analyses at all. One could, for example, start with the actual module decomposition and restructure the modules by hand without any proposal by automatic techniques. The framework would then only be used to keep track of the manual findings and as a convenient cross-reference tool. Whether the automatic analyses are helpful at all is yet an open question. The next chapter reports on an experiment conducted to look into this.

Chapter 10 Experiments to Evaluate the Semi-Automatic Method

This chapter describes an experiment conducted to empirically evaluate the impact of the automatic techniques for atomic component detection within the semi-automatic method and a case study to see whether our tools for atomic component detection may also help maintainers in other typical maintenance activities.

10.1 Goals of the Experiments

In Chapter 6, the detection quality for atomic component recovery of diverse techniques was evaluated. The results indicate that none of these techniques reaches the human recall rate. However, the techniques did find many relevant atomic components very quickly and, therefore, can support the maintainer. Chapter 9 proposed a method in which maintainer and computer work hand in hand to find atomic components. The automatic techniques are to support the user in the detection process. However, it is not per se clear that a maintainer using the automatic techniques is faster than one without the analyses. For this reason, I conducted an experiment to examine the aid of automatic techniques.

The framework described in the last chapter is primarily aimed at atomic component detection. Finding atomic components is a prerequisite for migration of a legacy procedural system to an object-oriented system, it supports identification of reusable components, and may help in program understanding by providing a more abstract view of the system above the code level. Beyond component recov-

ery, the prototype provides the maintainer also with many other base facilities that may support more typical maintenance tasks. Among these are cross-referencing, search mechanisms, automatic derivation of exact interfaces, and so forth. In order to gather experiences on how useful they actually are and what extensions would be necessary for a broader support of maintenance, we performed a case study in which students were asked to perform typical maintenance tasks with or without the prototype.

There were several constraints for the experiments that I want to point out before the experiments are described in the following sections. Only very few companies exist that are willing to spend time and money in an experiment whose outcome is unclear. Since this was our first experiment, we could not approach industry by showing previous success stories. Therefore, I decided to conduct the experiment with students. Unfortunately, it was very difficult to find even students for this task. Only few students volunteered. Due to their limited available time, the experiments including training and discussion could not exceed 20 hours. For this reason, it was not affordable to do more than one system per experiment since systems of a realistic size were to be used and the experimental subjects should have enough time to do more than a superficial analysis.

Since the experimental subjects were students and since their number was low and also because only one system per experiment was analyzed, we refrain from generalizing the results too far. The general objective of these experiments was not to yield a definite empirical proof for the usefulness of the semi-automatic method for all kinds of systems and settings but to learn about the strengths and weaknesses of the method and to investigate where further research should be directed. Furthermore, the detailed description of the experimental layout of this pilot study and its statistical analysis in the following sections can be used to repeat the experiment in an industrial setting and with a larger number of experimental subjects for other systems.

10.2 Experimental Subjects

Nine students volunteered for the experiment. Since we chose a two-block design, we asked the system administrator of our department to participate in order to get

equal group sizes. The system administrator was not involved in programming for our project.

At the time of the experiment, the students were studying computer science at the University of Stuttgart (six at the graduate level, three at the undergraduate level). All of them had at least two years of programming experience and were familiar with the programming language C. See Table 10-1 for their individual profile.

Table 10-1. **Profile of the experimental subjects.**

experimental subject	# semester	programming experience
S_1	11	good
S_2	9	good
S_3	9	good
S_4	3	good
S_5	5	average
S_6	9	good
S_7	3	good
S_8	3	average
S_9	9	good
S_{10}	professional	good

10.3 Experiment to Evaluate the Semi-Automatic Method

The experiment described in this section addresses the impact of the automatic techniques within the interactive and incremental method.

10.3.1 Hypotheses

The general hypothesis is that the semi-automatic method as described in Chapter 9 yields more atomic components than a pure manual process. By manual search, we mean that only the cross-reference capabilities of our atomic component detection framework as well as common textual pattern matching tools, such as

grep, are used, whereas for the semi-automatic method, all automatic analyses are available.

The independent variable is therefore:

- *semi-automatic method* versus *manual search*

The dependent variable is:

- the recall of atomic components

The null hypothesis and the two-sided and single-sided alternative hypotheses are as follows:

Null hypothesis H_0: There is no difference in the recall rate for the semi-automatic and manual method.

Alternative hypothesis H_1: The recall rates for the semi-automatic method and the manual search differ (for two-sided tests).

Alternative hypothesis H_2: The recall rate for the semi-automatic method is greater than the one for manual search (for single-sided tests).

10.3.2 Experimental Materials

The task of the experimental subjects was to recover the atomic components for *Mosaic* (see Table 6-1 on page 154). In order to obtain comparable results, we reduced the possible search space for atomic components to a size that could be handled within the given time frame, i.e., all experimental subjects should be able to look at all source files within the available time. Therefore, we excluded the files that are mainly devoted to the graphical user interface, namely, all files whose names begin with the prefix *gui*. The 8 excluded files comprise 15 KLOC, i.e., 40 files consisting of 37 KLOC were to be analyzed. None of the experimental subjects was familiar with the implementation of the system.

Mosaic was selected for several reasons. First, the students were all acquainted to the application domain of web browsers. Second, its implementation involved several programmers such that different programming styles could be expected.

Third, Mosaic is used in many other reverse engineering studies which allows results more comparable to other approaches.

10.3.3 Tool Support

The tools used for the experiment were as follows:

- **standard tools**: the editors *emacs* and *vi*, the Gnu C compiler *gcc*, and *grep* (the Gnu tool for pattern matching based on regular expressions); these tools are widely used on Unix platforms and can be considered standard tools for maintainers (all participants were familiar with these tools before the experiment)
- **plain Rigi**: the graph editor Rigi without the analyses that was used as a graphical cross-reference tool and to capture the results of the search; plain Rigi can be considered a representant of source code browsing tools; even though there are similar tools of this type available on the market (mostly for object-oriented systems primarily to browse the inheritance hierarchy and often only text-based), these tools are rarely available to the typical maintainer
- **extended Rigi**: the graph editor Rigi with an integration of the analyses described in Chapters 5, 8, and 9

Limitations of extended Rigi. At the time of the experiment, the extended Rigi had some limitations. Some of the ideas presented in previous chapters were only inspired by the experiment. The restrictions were as follows:

- Among the combining set operators, only deep intersection was available. Composition was implemented in the form of incremental analyses but not as an operator that could be applied to individual components.
- Neither was the voting approach implemented.
- *Similarity Clustering* could not be run in parallel and, therefore, one had to wait up-to ten minutes for the results (for this reason, it was rarely used by the experimental subjects during the experiment that was limited to six hours).
- Moreover, it was not possible to accept components in the hierarchical view returned by *Similarity Clustering*. The hierarchical view had to be flattened before components could be accepted. Hence, an advantage of *Similarity Clustering* could not be leveraged.

- There were also problems with Rigi's (both plain and extended Rigi) update strategy for visualized components views: Whenever a component was accepted, the layout of all windows was lost and people had to re-order their components views.
- Furthermore, hierarchical subsystems were not supported. Interestingly enough, people in both groups complained that there was no higher grouping mechanism on top of atomic components which would have allowed them to group related atomic components. This shortage motivated the generalization of the combining operators for atomic components to subsystems as described in Chapter 8.

10.3.4 Experimental Design

The experimental subjects were randomly assigned to two groups that differed in the tools available to the search for atomic components:

- **Group SAM (semi-automatic method)**: extended Rigi
- **Group MS (manual search)**: standard tools and plain Rigi

Table 10-2 shows how the experimental subjects were randomly assigned to the two groups. The size of each group was five persons.

Table 10-2. **Experimental groups.**

group	experimental subjects
SAM	S_1, S_3, S_5, S_6, S_8
MS	S_2, S_4, S_7, S_9, S_{10}

In order to avoid too much variance in the set of experimental subjects, all experimental subjects were jointly trained as follows:

1. The diverse kinds of atomic components that were to be detected were explained and exemplified. (30 minutes)
2. The available tools (*emacs*, *vi*, *grep*, and *plain* and *extended Rigi*) were introduced. The available analyses in the *extended Rigi* were even introduced to members of group MS though they would not use the extended Rigi. This was done to avoid a margin of the SAM group since introducing the analyses (which was necessary to use them at all) means teaching the heuristics that are

associated with the analyses.
(1 hour)

3. The experimental subjects were trained with an example system (the unix spread sheet calculator *sc* with about 10 KLOC). The subjects had to detect as many atomic components as possible within a fixed period of time. The available tools for this task were emacs, vi, grep, and plain Rigi for all trainees. Members of group MS could also use the extended Rigi. (3 hours)
4. The result of the training was jointly discussed in order to achieve an agreement on the notion of atomic components among the subjects. This discussion revealed a consensus about the general notion of atomic components among the experimental subjects. (1 hour)

In the actual experiment after the training, every subject had to analyze Mosaic for 6 hours. The system was already preprocessed, i.e., the resource usage graph for the analyzed system was available for the experimental subjects.

10.3.5 Measurement of the Dependent Variable

Since the number of atomic components for Mosaic was not known in advance, a termination criterion for the search for atomic components did not exist. That is why we stopped the search after 6 hours. Limiting the available time makes the experiment even more realistic since in an industrial setting, one can generally not afford to spend unrestricted time on a problem.

Two distinct ways of measuring the dependent variable were chosen:

1. using the absolute number of clustered elements for each subject (**individual absolute recall**, short **IAR**)
2. comparing the components of each individual to the joint set of components of all individuals (**reference corpus recall**, short **RCR**)

The first alternative does not require agreement among the experimental subjects, hence measures only how many elements were clustered in a given time by each individual.

For the second alternative, the individual results were joined and the result of each individual was judged with respect to the joined result. The joined list of

components was individually reviewed by the experimental subjects. The review work was distributed among the subjects so that each atomic component was at least reviewed by two persons. In order to reduce the effort for the experimental subjects to review the reference corpus, each subject had only to review a part of the reference corpus. The reviewing subjects could accept components as a whole or in parts as well as add elements to the components. Overlapping atomic components were allowed. The set of accepted atomic components formed the **reference corpus** to which the proposed atomic components of each subject were compared. The comparison followed the method described in Section 6.2. The dependent variable was measured as the recall rate defined in Section 6.2.2 with respect to the reference corpus.

Because not each individual reviewed the whole reference corpus, it may have happened that people would not agree to certain parts. That is why both *reference corpus recall* as well as *individual absolute recall* will be evaluated in the following.

10.3.6 Experimental Results

The recall rate of each experimental subject with respect to the reference corpus and the individual absolute recall are listed in Table 10-3. The numbers are listed in descending order; this order does not correspond to the order in which the experimental subjects are listed in Table 10-1 for reasons of anonymity.

Table 10-3. **Results for Mosaic.**

		x_{*1}	x_{*2}	x_{*3}	x_{*4}	x_{*5}	$\sum x_{*i}$	$\bar{x}_{*i} = (\sum x_{*i})/5$
RCR	**SAM**	0.42	0.28	0.28	0.20	0.16	1.34	0.27
	MS	0.48	0.35	0.27	0.24	0.12	1.46	0.29
IAR	**SAM**	433	400	290	204	248	1575	315
	MS	498	275	275	326	150	1524	305

Both the reference corpus and individual absolute recall are approximately the same for both groups.

10.3.7 Statistical Analysis

The common way to evaluate statistical data of controlled experiments is to apply analysis of variance (ANOVA). The *F statistic,* for example, may be used to test the hypothesis that the population means for the two groups are equal (Winer et al., 1991). However, the *F statistics* and other statistical tests of ANOVA assume a certain distribution of the population or themselves approach a normal distribution only for large samples. However, normal distribution cannot be assumed for our experiment. There have not been any large-scale experiments on the recall rate of programmers in finding atomic components yet and, hence, the actual distribution is unknown. Furthermore, the size of our sample is too small to evaluate it with the *F statistics*. There are other statistics, so-called non-parameterized statistics, that do not assume any distribution and are applicable to small samples. The power of these tests is generally better than the power of parameterized tests. According to Lienert, there are basically two kinds of statistics appropriate for the design chosen for this experiment (1973): The exact U-test by Mann and Whitney (1947) and the exact Fisher-Pitman randomization test for two independent samples (Pitman, 1939). These two methods differ in the leveraged scaling information of the data. The exact U-test assumes data at an ordinal scale, i.e., the data can only be compared in terms of a greater/lesser relationship, whereas the exact Fisher-Pitman test is based on interval information. Since the recall rate is actually at an interval scale, Fisher-Pitman test seems to be the appropriate test. However, it assumes that the samples are an exact image of the whole population which cannot really be justified since we dealt almost only with students. Therefore, both tests are used for the evaluation.

10.3.7.1 Exact U-Test

The exact U-Test consists of the following steps (Mann and Whitney, 1947; Lienert, 1973):

1. The data of both groups are united and ordered.
2. Each value of SAM is compared to all other values of MS. Let G_i be the number of elements of MS that are smaller than element i and L_i be the number of elements of MS that are greater than i.

3. Summarize the numbers: $G = \sum_{i \in SAM} G_i$ and $L = \sum_{i \in SAM} L_i$. The smaller figure is the **observed *U* value**.
4. The **expected value of *U*** under the null hypothesis is $\mu_U = (N_{SAM}+N_{MS})/2$ where N_{SAM} is the size of group SAM and N_{MS} is the size of group MS. In our experiment, $N_{SAM} = N_{MS} = 5$. The more *U* differs from μ_U the less likely does the null hypothesis hold.
5. The likelihood to get the observed *U* value or a value smaller than the observed *U* value is determined by the number Z_U of those combinations out of the

   $$\binom{N_{SAM} + N_{MS}}{N_{SAM}} = \binom{N_{SAM} + N_{MS}}{N_{MS}}$$

   possible permutations of SAM and MS recall rates that yield a *U* value not greater than the observed *U* value (for a single-sided test):

   $$P = Z_U / \binom{N_{SAM} + N_{MS}}{N_{SAM}}$$

   and for a two-sided test: $P = (Z_U + Z_{U-1}) / \binom{N_{SAM} + N_{MS}}{N_{SAM}}$
6. Reject the null hypothesis if *P* is less than a certain threshold.

The united and ordered data of Table 10-3 are listed in Table 10-4 and Table 10-5. For the reference corpus recall, *L*=12 and *G*=13, hence *U*=*L*=12; and for the individual absolute recall, *L*=13 and *G*=12, hence *U*=*G*=12.

Table 10-4. **Ordered reference corpus recall rates for SAM and MS.**

rank	1	2	3	4	5	6	7	8	9	10
recall	0.12	0.16	0.2	0.24	0.27	0.28	0.28	0.35	0.42	0.48
group	MS	SAM	SAM	MS	MS	SAM	SAM	MS	SAM	MS
G_i		4	4			2	2		1	
L_i		1	1			3	3		4	

For the probability *P*, one could either lookup *P* in the table provided by Owen (1962, Table 11.3) or write a small program that computes Z_U by evaluating all

Table 10-5. **Ordered individual absolute recall for SAM and MS.**

rank	1	2	3	4	5	6	7	8	9	10
recall	150	204	248	275	275	290	326	400	433	498
group	MS	SAM	SAM	MS	MS	SAM	MS	SAM	SAM	MS
G_i		4	4			2		1	1	
L_i		1	1			3		4	4	

permutations of MS and SAM members. For $U = 12$, this program will compute Z_U=126 and Z_{U-1}=106, hence:

$$P = (126 + 106) / \binom{5+5}{5} = 0.92 \text{ for a two-sided test.}$$

Since we chose U=L=12 for the reference corpus recall and U=G=12 for the individual absolute recall, the null hypothesis holds for both measurements with a probability of 0.92. In other words, a positive effect of the automatic analyses for atomic component detection could not be shown.

10.3.7.2 Exact Fisher-Pitman Test

As opposed to the exact U-test, the exact Fisher-Pitman test leverages the interval information of the recall rate data (Pitman, 1939; Lienert, 1973). However, its assumption is that the sample is an exact image of the whole population. The exact Fisher-Pitman test is based on the following observations: The whole sample (i.e., the union of SAM and MS) of the $N_{SAM} + N_{MS} = N$ values may be split into two single samples with N_{SAM} and N_{MS} values in

$$\binom{N}{N_{SAM}} = \binom{N}{N_{MS}}$$

different ways. Each of them is equally likely with respect to the null hypothesis. If we use the difference of the locations (means or medians), i.e., difference $D = \bar{x}_{SAM} - \bar{x}_{MS}$, as the value against we test, then we can compute the D values for all $\binom{N}{N_{SAM}}$ two-sample combinations and determine whether the observed test value D is among the $Z + z$ highest D values (Z is the number of

two-sample combinations whose value is the observed *D* value and *z* is the number of two-sample combinations whose value is less than the observed *D* value). If so, we reject the null hypothesis. An equivalent but simpler evaluation strategy is as follows:

1. The sum *S* of the recall rates of the smaller probe is the value against which we test (in our case, both samples are at the same size):

   $$S = S_{SAM} = \sum_{i=1}^{N_{SAM}} x_{SAMi}$$

   Let S_{MS} be the sum of the recall rate for MS, i.e., $S_{SAM} + S_{MS} = T$, then:

   $$D = \frac{S_{SAM}}{N_{SAM}} - \frac{S_{MS}}{N_{MS}} = S_{SAM} \cdot \left(\frac{1}{N_{SAM}} - \frac{1}{N_{MS}}\right) - \frac{T}{N_{MS}}$$

   $$\Leftrightarrow S = S_{SAM} = \left(D + \frac{T}{N_{MS}}\right) \cdot \frac{N_{SAM} \cdot N_{MS}}{N_{SAM} + N_{MS}}$$

   Since T/N_{MS} and $N_{SAM}N_{MS}/(N_{SAM} + N_{MS})$ are both constants, the test distribution of S_{SAM} is functionally connected to the test distribution of *D*. Therefore, we can use the more easily computable distribution of S_{SAM} instead of *D*.

2. The test whether an observed *S* value is sufficient to reject the null hypothesis is as follows. First, we compute all two-sample combinations whose *S* value is either smaller than or equal to the observed *S* value; let the former be denoted by *z* and the latter by *Z*. Then, for a single-sided alternative hypothesis, the probability of the null hypothesis is as follows:

   $$P = (Z + z) / \binom{N}{N_{SAM}}$$

Since the distribution of *S* depends upon the samples and, therefore, differs from test to test, it cannot be tabulated. One may write a program that computes *Z* and *z* over the possible two-sample combinations of the values of S_{SAM} and S_{MS}.

For the data given in Table 10-4, *Z*=154 and *z*=2 and therefore:

$$P = (154 + 2) / \binom{10}{5} = \frac{156}{252} = 0.62$$

In other words, the likelihood that the alternative hypothesis H_2 holds is 38 percent.

For the data in Table 10-5, Z=110 and z=2 and, hence, P = (110 + 2) / 252 = 0.44. Therefore, the likelihood of the alternative hypothesis H_2 is 0.56.

In comparison to the exact U-test, the likelihood of the alternative hypothesis has increased both for the reference corpus recall and the individual absolute recall. Yet, these results are still at a low significance level and, therefore, are not sufficient to reject the null hypothesis.

10.3.8 Summary

A positive effect of the automatic analyses could not be shown by the experiment on the evaluation of the semi-automatic method. However, the conclusions should not be generalized too far for the following reasons:

- The subject system used for the experiment, Mosaic, is well-structured. The experimental subjects were allowed to use the module view that consists of the modules of the system and their respective contents. The module view corresponds to the *Same Module* heuristic without the restriction of *Same Module* that the elements of a component have to be (transitively) connected to each other. Since *Same Module* - and hence, the module view if the modules are not too large - is one of the most effective techniques for well-structured systems, the advantage of people with automatic support was only marginal.
- Beside the fact that the subject system was in a better shape than many legacy systems, the experimental subjects may also not be considered typical average programmers. All participants were computer science students with grades better than the average. A future experiment should investigate whether less talented programmers would profit more from automatic analyses.
- Furthermore, even members of group MS could use automatic support to some degree. They could use plain Rigi as a cross-reference and browsing tool and to

log their findings. Mostly, programmers can only use very simple search tools, like *grep*, in practice.

- One must also note that at the time of the experiment, the framework did not offer all the functionality described in Chapter 8 and Chapter 9. Section 10.3.3 describes the limitations of the extended Rigi used for the experiment.

However, even for worse decomposed systems and less talented programmers, for which automatic analyses may be more helpful, we should be aware that the support of the automatic analyses is limited to gathering candidates. This part, even done by hand, is comparatively small to the time needed for validation by the maintainer who will always have to look at the source code. As long as there are no absolutely precise techniques whose candidates can be accepted without checking, there will be a constant human factor in the process of component detection that cannot be eliminated. The goal for automatic analyses must be to be as reliable, flexible, and quick as possible for an interactive application. Here, the framework inherited in parts the weaknesses from the used automatic techniques. As the evaluation in Chapter 6 has shown, the techniques may produce bad candidates and do not find all components either. Future research should be aimed at finding more precise techniques considering other sources of information, like data flow information or domain knowledge. The techniques described in this thesis leveraged only structural information.

The experiment described in this chapter could not show any positive effect of the analyses (nor a negative effect). On the other hand, experiences with the overlooked false positives described in Section 6.4 indicate that automatic analyses may be useful when larger parts of a system are to be analyzed. In the experiment with Mosaic, the search was limited by time (6 hours). On the other hand, in gathering the references in Aero, Bash, and CVS used to evaluate the automatic techniques, which took between 20 and 35 hours per system, 42% of the ADO candidates and 41% of the ADT candidates proposed by the automatic techniques and formerly categorized as false positives were indeed overlooked or alternative components.

10.4 Case Study for Maintenance Support

In order to find out whether the framework could also be used to support maintenance tasks other than atomic component detection, a case study with the students of the previous experiment was performed. The goal of this case study was also to learn what other types of analyses would be useful for maintainers.

All but one of the experimental subjects of the last experiment participated in this case study. The participants were in the same groups as in the previous experiment. The size of group SAM was four and the size of group MS was five. Members of group SAM could use the extended Rigi, while members of group MS were only allowed to use the standard tools *grep*, *vi*, *emacs*, and a C compiler, i.e., they could not use plain Rigi.

The system used for the case study was XCoral (version 3.2), an X-window-based editor consisting of 73 KLOC of C code. The editor also contains a large subsystem SMAC that implements an interpreter for a subset of the programming language C. The structure of the system is highly deteriorated, information hiding is disregarded, and many function clones (duplicated and slightly modified functions) exist.

The assignments for the participants involved typical maintenance problems like change of data structures, lifetime analysis of variables, interface identification, concept recognition, and clone detection. The tasks were as follows (the comments in italic font were not mentioned to the participants):

1. All clients of the ADT *List* directly access the record components *next* and *previous* of *List*. These components should now be hidden, i.e., no function — except for the accessor functions of *List* — may access internal components of *List*. Instead, an iterator for *List* should be provided and used at the client site. What changes are necessary?
   This task was aimed at impact analysis of changes to a data structure. However, the primary question was whether the participants would find the already existing iterator for ADT List that was implemented in another file and then avoid the re-implementation of the found iterator.
2. The subsystem SMAC has a memory leak at its hash table because the destructor *HashTable_Empty* is never called. Insert this function call at the sites where the lifetime of the hash table ends. If the lifetime of the hash table lasts till the

end of the program, it does not need to be explicitly released; in this case, it will be implicitly released with the program termination.
This task was aimed at lifetime analysis of variables. SMAC has 8 global hash tables, one of them not even used at all, and no local hash tables (except for one in the constructor for hash tables). The lifetime of all global variables ends with the program. As a matter of fact, in the narrower sense, there was no memory leak of the table itself. But there was one call to HashTable_Empty freeing the entries of the hash table that was removed from the source. Hence, the memory allocated for the entries was not released, which in turn would lead to an error in the application because the table needed to be empty at some points.

3. What is the exact interface between SMAC and XCoral, i.e., what parts of SMAC are used within XCoral and what parts of XCoral are used within SMAC?
   As a matter of fact, there is one file smacXCoral that is supposed to be the interface between XCoral and SMAC. However, there are a few other declarations of SMAC used in XCoral not listed in smacXCoral. Interestingly enough, there are also declarations of XCoral used within SMAC.
4. File *file_dict.c* implements an ADT *FileRec* as a list of filenames. Moreover, this file contains a global variable *file_dict*. What is the concept behind these declarations? How could this concept be reused? And where (within the system)?
   On the surface, this task was aimed at concept recognition. The actual purpose was to detect existing clones of this concept.

The total time available for all these tasks together was six hours. The tasks were to be handled in the given sequence. Participants were allowed to skip a task when they could not find a solution. Actually, there was another task aiming at the understanding of a central data structure *Text* that consists of many “inlined” subconcepts whose accessor functions are implemented in different files. However, only three participants arrived at this task within the given time. The individual times needed for these assignments are shown in Table 10-6. The figures in brackets are the time spent on a task before a participant gave up (these numbers are not considered for the average and median).

Table 10-6. **Needed time in minutes for task 1-4.**

		Task 1	Task 2	Task 3	Task 4
SAM	**SAM_1**	85	115	60	70
	SAM_2	90	75	(40)	95
	SAM_3	60	90	75	65
	SAM_4	60	30	120	60
Average SAM		*74*	*78*	*85*	*73*
Median SAM		*73*	*83*	*75*	*68*
MS	**MS_1**	35	80	130	40
	MS_2	125	60	75	30
	MS_3	110	50	(45)	120
	MS_4	80	80	37	40
	MS_5	74	56	92	54
Average MS		*85*	*65*	*84*	*57*
Median MS		*80*	*60*	*84*	*40*

Table 10-6 shows a high variance among the participants for most tasks in both groups. Members of group SAM needed less time than members of MS for task 1, similar time for task 3, and more time for tasks 2 and 4.

10.4.1 Task 1 - Change of Data Structure

The variance of the time needed for task 1 can partly be explained by the varying degree of detail in the answers of the participants. Participant MS_1, for example, identified only the files that are affected by the proposed change, while participant MS_2 even listed all source positions that would have to be changed and described how these sites should be changed.

Members of group MS used *grep* to identify the places where a *previous* and *next* occur. This strategy would have been more difficult if there had been another data structure with components named alike. One of the members of group MS even noted that certain sites where *previous* and *next* occurred would not have to be

changed because they belong to a global variable *Memo*. However, the participant overlooked that *Memo* is of type *List*.

The extended Rigi was able to support task 1, but only in two steps. For this task, the *Internal Access* heuristic could be used to gather all functions that access the internal components of global variables and parameters of type *List*. However, internal accesses to local variables are not reported by this analysis. Subprograms having local variables of type *List* could be identified by the cross-reference facility and then, their source code could be inspected. This explains why members of group MS still needed so much time. *Internal Access* could be enhanced toward fully automatic support of maintenance jobs like task 1. An enhanced *Internal Access* analysis would then report reliable results within a minute where primitive tools, like grep, yield only a very rough approximation that has to be validated by the maintainer.

In both groups, there were three people that did not realize the existence of an iterator for *List* in the source. The assumption was that members of group SAM would overlook the iterator more often because they would rely on the results of *Internal Access* and, therefore, not look at the source code. However, members of MS who were far more attached to the source code overlooked the iterator equally often.

10.4.2 Task 2 - Lifetime Analysis

The support for lifetime analysis by the extended Rigi is very limited. Finding the hash tables in the system could be done by the cross-reference facilities. But the available control and dataflow information in the resource usage graph is very rough. The call view abstracts from the exact order of function calls and contains only directly visible calls; limited data flow information is only available as the *set* and *use* relationships for global variables. Likewise, people solely using *grep* could identify the global hash tables. Most people in both groups concluded from the scope of the global variables that the lifetime of the variables would end with the program, others were unsure. None realized the missing call to the function that empties the hash table. One participant of group MS proposed to check for a defined hash table in certain initialization routines and then to release the hash table when already defined before calling the constructor. However, firstly, this would not be a remedy since the initialization routines are only called once in the

beginning and, secondly, this could even result in an error when the hash table were aliased and the initialization functions were called more than once.

10.4.3 Task 3 - Exact Interface Identification

Members of the MS group looked at the include statements in order to identify the exact interface. Those files were closely investigated that included a file from the other subsystem. Since all files of SMAC are in a subdirectory of the XCoral source directory, the files of XCoral used in SMAC could be identified by include statements like "../file.h". This strategy did not work for files of SMAC used by XCoral since there was no include statement like "Smac/file.h". Instead, explicit extern declarations for SMAC elements were used within XCoral. Looking only at files that are explicitly included may also result in overlooking usages of routines for which not even an extern declaration within the using file exists since routines are implicitly declared in C when no declaration can be found.

In Rigi, there is a built-in function that allows the extraction of exact interfaces of composite nodes. The most simple solution would have been to collapse all files of SMAC into one node (identified by their path) and all other files into a second node. Then the exact interface could have been ascertained for both nodes. This would have been work of few minutes. However, members of SAM did not see this ability. They rather used the cross-reference capabilities.

Of both groups, one participant gave up at this task.

10.4.4 Task 4 - Concept Recognition and Clone Detection

There are approaches that try to automatically assign concepts to pieces of code using typical coding patterns for data structures and algorithms (Wills, 1992; Quilici, 1997). The extended Rigi does not offer such capabilities. The only support it provides for this kind of task is to find groups of related elements without assigning any meaning to these groups, and offering a higher level of abstraction by visualizing global declarations and their relationships only. The actual goal of this task, however, was to see whether a more abstract view would help or hinder in realizing function clones. On one hand, providing a more abstract view may suppress important details because the source code is not immediately visible and the abstracted information may not be sufficient to recognize function clones. On the

other hand, merely looking at the source code may lead to get lost in details. This case study did not provide evidence for either hypothesis. In both groups, there was one member that did not see the function clones. Both members recommended to use another more general data structure in the subsystem SMAC instead.

10.4.5 Summary

The case study described in this section was headed at the ability of the extended Rigi to support maintenance tasks other than those it was originally designed for, namely, atomic component detection. The goal of this case study was also to learn what other types of automatic analyses would be useful for maintainers.

The average time needed for the diverse maintenance tasks performed in this case study was at least 1 hour. Cases in which participants needed 2 hours were not rare. However, at least three of these tasks could easily be automated (to some degree).

Tasks 1 and 3 fall into the same category of static name binding for a given set of declarations at the atomic component and subsystem level which can reliably be done with a global semantic analysis as far as visible declarations are concerned. In the case of interfaces, there may also be hidden dependencies that are harder to track down. For example, there could be an external file which one part of the system writes into and the other part reads from. Such hidden dependencies may be partly derived by control and data flow analyses, e.g., by means of constant propagation. However, it need not necessarily be recognizable or decidable that the same external file is meant at two different sites of the system.

Problems of the kind of task 2 deal with lifetime and protocols of components (a protocol is a specification on the allowed order of actions associated with a component). How long a component exists is often undecidable statically; for example, when the component is allocated on the heap, one would have to find out whether there is no pointer to the component left in order to ascertain the end of the lifetime of the component. The protocol of the hash tables in the above assignments would require to call *HashTable_Empty* at the end of the lifetime of each hash table. Since the end of the lifetime is often undecidable, this protocol specification cannot always be checked statically. On the other hand, if points-to analy-

ses or user assertions exclude aliasing of the hash table, the lifetime of the hash table may in fact be derived and it can be checked whether *HashTable_Empty* is properly called.

In order to support clone detection, as requested in task 4, several automatic approaches have been proposed. Baker uses pattern matching techniques (1995), Mayrand et al. compare values of certain metrics in order to identify pieces of code that perform similar functions (1995), and Baxter et al. match abstract syntax trees (1998).

Part IV Finale

Chapter 11 Related Research

This chapter summarizes research in architecture recovery related to atomic component detection.

11.1 Other Automatic Component Detection Techniques

Most automatic techniques for component detection have already been presented in Chapter 5. Others not considered in this thesis follow in this section.

11.1.1 Metric-based Approaches

There have been several clustering techniques for module and subsystem detection proposed that are based on a similarity metric. Schwanke's work (1992) and the work of Patel, Chu, and Baxter (1991) have already been discussed in Section 5.9 and Section 5.10, respectively.

11.1.1.1 Belady and Evangelisti

Belady and Evangelisti's approach groups related subprograms using a similarity metric based on data bindings (1982). A data binding is a potential data exchange via a global variable. Several kinds of data binding can be distinguished according to the following levels of accuracy: A *potential data binding* is defined as an ordered triple <*p*,*x*,*q*> where *p* and *q* are procedures and *x* is a variable within the static scope of both *p* and *q*. A *used data binding* is a potential data binding where *p* and *q* either set or use *x*. An *actual data binding* is defined as a used data bind-

ing where *p* assigns a value to *x* and *q* uses *x*. A *control flow data binding* is defined as an actual data binding where there is a "possibility" of control passing to *q* after *p* has had control. A possibility is said to exist whenever either (1) there exists a chain of calls from *p* to *q* or vice versa, or (2) there exists a procedure *r* such that there are chains of calls from *r* to *p* and from *r* to *q* and there exists a path in the directed control flow graph connecting the call chain to *p* with the call chain to *q*. The similarity metric is based on the percentage of the bindings that connect to either of the two components and are shared by the components. Varying reliability can be achieved by selecting different degrees of data bindings.

11.1.1.2 Hutchens and Basili

Hutchens and Basili extend Belady and Evangelisti's work by using a hierarchical clustering technique to identify related subprograms and subsystems (1985).

11.1.1.3 Girard and Würthner

Girard and Würthner's work is aimed at identification of functionally cohesive components and subsystems (Eisenbarth et al. 1999). Functionally cohesive components are groups of routines that together implement a certain functionality. Candidates of functionally cohesive components may be identified as subtrees of the dominance tree for the call view. The root of the subtree is the interface function, all other functions of the subtree are local service functions to the interface function. In order to retrieve functionally cohesive components from the dominance tree, two heuristics were proposed. The first heuristic is based on the size of the subtree. During bottom-up traversal of the dominance tree, a subtree is selected whose size (in terms of number of nodes) is not larger than a user-determined threshold. However, selecting subtrees by size does not necessarily say anything about the cohesion of a component and, therefore, another heuristic based on shared variables is proposed. The underlying assumption of the second heuristic is that a subcomponent, *S*, that has a different functionality than a component, *C*, dominating *S* may be distinguished by its references to global data. If *S* implements a functionality that is only needed in the context of *C*, it is quite likely that *S* references the same data as *C**S* (the set of subprograms in *C* without those in *S*). If *S* implements a functionality that differs from the one of *C*, *S* probably needs further data. Hence, a dissimilarity can be defined as:

$$\text{dissim}(S, C) = \frac{|V(S) \setminus V(C \setminus S)|}{|V(C)|}$$

where $V(X)$ is the set of variables referenced by subprograms in X. That is, the dissimilarity measures the ratio of the variables additionally used by S. An incremental algorithm can be used to cut the dominance tree so that dissimilarity among components is maximized. Once the functionally cohesive components have been identified, they can be grouped to subsystems based on the variables they share. Likewise, variables can be clustered based on the functionally cohesive components that reference these variables. This approach is particularly suited for a language like Fortran in which global variables are very frequent.

11.1.1.4 Mancoridis et al.

Mancoridis et al. propose measurements very similar to internal and external connectivity as defined in Section 5.8 for clustering modules to subsystems (1999). They propose a genetic algorithm based on these metrics for finding a partition of the module view that minimizes external connectivity and maximizes internal connectivity. The approach was applied to modules only, hence, very few nodes — compared to the number of base entities in a system — are to be clustered. Whether the use of genetic algorithms scales to a finer granularity has to be shown. Furthermore, external connectivity as defined by Mancoridis et al. ignores the actual number of dependencies existing among modules, i.e., modules with only one dependency have the same connectivity as modules with hundreds of dependencies.

11.1.2 Concept Analysis

Concept analysis is also based on structural information; yet — as it will be shown in this section — it is a class of approaches quite different from those presented in Chapter 5 and represents its own field of research. For this reason, it has not been explored further in this thesis. This section acquaints with the basic ideas of concept analysis and summarizes existing applications of concept analysis to atomic component detection. Because concept analysis is also based on structural information, yet not investigated in this thesis, the diverse approaches based on concept analysis are presented in more detail than other techniques within related research.

Concept analysis provides a way to identify groupings of entities that have common attributes. Its mathematical foundation was laid by G. Birkhoff in 1940. Gregor Snelting introduced concept analysis to software engineering in 1998. Since then it has been used to evaluate class hierarchies (Snelting and Tip, 1998), explore configuration structures of preprocessor statements (Krone and Snelting, 1994; Snelting, 1996), and to perform atomic component detection (Lindig and Snelting, 1997; Siff and Reps, 1997; Sahraoui et al., 1997; Graudejus, 1998; Canfora et al., 1999).

11.1.2.1 Mathematical Background

Concept analysis is based on a relation $\mathcal{R}$ between a set of objects O and a set of attributes $\mathcal{A}$, hence $\mathcal{R} \subseteq O \times \mathcal{A}$. An object in the sense of concept analysis can be anything, not only *objects* as defined by Section 3.1.1. Within this section on concept analysis, the term *object* will be used in the sense of concept analysis.

The triple $C = (O, \mathcal{A}, \mathcal{R})$ is called a **formal context**. For any set of objects $O \subseteq O$, the set of **common attributes** is defined as

$$\sigma(O) = \{a \in \mathcal{A} \mid \forall (o \in O)(o, a) \in \mathcal{R}\}$$

Similarly, for any set of attributes $A \subseteq \mathcal{A}$, their set of **common objects** is

$$\tau(A) = \{o \in O \mid \forall (a \in A)(o, a) \in \mathcal{R}\}$$

As an example, let us consider the fictitious binary relation described by Table 11-1. An object O_i has attribute A_j when there is a ✕ in row i and column j in Table 11-1. In this table, for example, the following two equations hold:

$$\sigma(\{O_1\}) = \{A_1, A_2\} \qquad \tau(\{A_7, A_8\}) = \{O_3, O_4\}$$

The two functions σ and τ form a **Galois connection**, i.e., a pair of two antimonotone functions:

$$O_1 \subseteq O_2 \Rightarrow \sigma(O_2) \subseteq \sigma(O_1) \text{ and } A_1 \subseteq A_2 \Rightarrow \tau(A_2) \subseteq \tau(A_1)$$

Table 11-1. **Example relation.**

	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8
O_1	×	×						
O_2			×	×	×			
O_3			×	×		×	×	×
O_4			×	×	×	×	×	×

and both $\sigma \circ \tau$ and $\tau \circ \sigma$ are closure operators: e.g., $\sigma \circ \tau(O)$ determines the biggest set of objects that have the same attributes as *O*.

A pair (*O*, *A*) is called a **concept**, if $A = \sigma(O) \wedge O = \tau(A)$, i.e., all objects share all attributes. For a concept $c = (O, A)$, *O* is the **extent** of *c*, denoted by *extent*(*c*), and A is the **intent** of *c*, denoted by *intent*(*c*).

Informally, a concept corresponds to a maximal rectangle of filled table cells modulo row and column permutations. For example, Table 11-2 contains the concepts for the relation in Table 11-1.

Table 11-2. **Concepts for Table 11-1.**

C_1	$\{O_1, O_2, O_3, O_4\}, \varnothing$
C_2	$\{O_2, O_3, O_4\}, \{A_3, A_4\}$
C_3	$\{O_1\}, \{A_1, A_2\}$
C_4	$\{O_2, O_4\}, \{A_3, A_4, A_5\}$
C_5	$\{O_3, O_4\}, \{A_3, A_4, A_6, A_7, A_8\}$
C_6	$\{O_4\}, \{A_3, A_4, A_5, A_6, A_7, A_8\}$
C_7	$\varnothing, \{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8\}$

The set of all concepts of a given relation forms a partial order via

$(O_1, A_1) \leq (O_2, A_2) \Leftrightarrow O_1 \subseteq O_2$ or, equivalently

$(O_1, A_1) \leq (O_2, A_2) \Leftrightarrow A_1 \supseteq A_2$.

If $c_1 \leq c_2$, then c_1 is said to be a **subconcept** of c_2 and c_2 is a **superconcept** of c_1. For example, $(\{O_2, O_4\}, \{A_3, A_4, A_5\}) \leq (\{O_2, O_3, O_4\}, \{A_3, A_4\})$ in Table 11-1.

The sets of concepts and the partial order form a complete lattice, called **concept lattice**:

$$\mathcal{L}(C) = \{(O, A) \in 2^{O} \times 2^{\mathcal{A}} \mid A = \sigma(O) \wedge O = \tau(A)\}$$

In this lattice, the **infimum** (or **join**) of two concepts is computed by intersecting their extents:

$$(O_1, A_1) \wedge (O_2, A_2) = (O_1 \cap O_2, \sigma(O_1 \cap O_2))$$

Note that $A_1 \cup A_2 \subseteq \sigma(O_1 \cap O_2)$, as $O_1 \cap O_2$ has at least common attributes $A_1 \cup A_2$. Thus, an infimum describes the set of attributes common to two sets of objects.

Similarly, the **supremum** (or **meet**) is computed by intersecting the intents:

$$(O_1, A_1) \vee (O_2, A_2) = (\tau(A_1 \cap A_2), A_1 \cap A_2)$$

Again, $O_1 \cup O_2 \subseteq \tau(A_1 \cap A_2)$. Thus, a supremum describes a set of common objects which fit to two sets of attributes.

Graphically, the concept lattice for the example relation in Table 11-1 can be represented as a graph whose nodes are the concepts in Table 11-2 and whose edges denote the < relationship as shown in Figure 11-1 (a).

The graph for a concept lattice would be difficult to read when each node showed its complete concepts, i.e., if the nodes C_i were replaced by their contents according to Table 11-2. Fortunately, there is a better strategy for labelling nodes. A graph node in Figure 11-1(b) is labelled with attribute $a \in \mathcal{A}$ if it is the largest concept having a in its intent; it is labelled with an object $o \in O$ if it is the smallest concept having o in its extent. The (unique) lattice element labelled with a is thus:

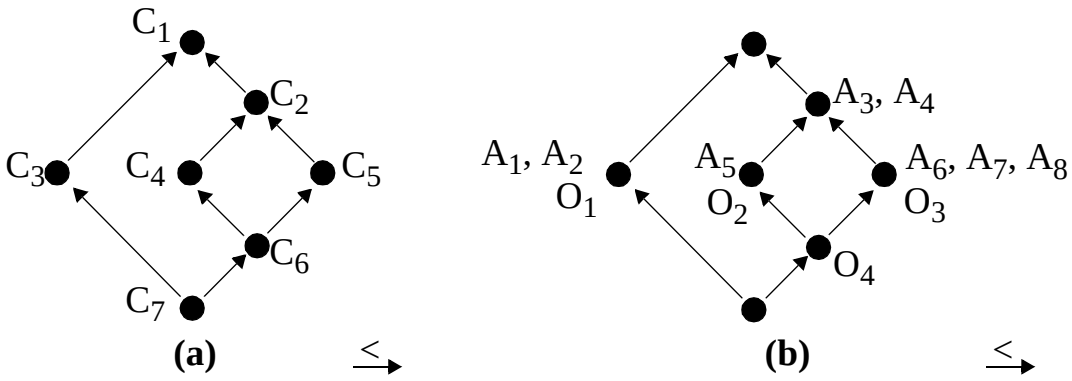


Figure 11-1. **Example lattice.**

$$\mu(a) = \vee \{c \in \mathcal{L}(C) | a \in intent(c)\}$$

The element labelled with *o* is

$$\gamma(o) = \wedge \{c \in \mathcal{L}(C) | o \in extent(c)\}$$

The equivalent graph for Figure 11-1(a) using this labelling strategy is shown in Figure 11-1(b). A concept represented by a node *N* in this graph consists of all objects at and below *N* and of all attributes at and above *N*. For example, the concept labelled with O_2 and A_5 in Figure 11-1(b) is ($\{O_2, O_4\}$, $\{A_3, A_4, A_5\}$).

The concept lattice and the table, *T*, originally used to represent the relation are two equivalent ways to represent the relation, i.e., they can be reconstructed from each other, formally stated as:

$$(o, a) \in T \Leftrightarrow \gamma(o) \leq \mu(a)$$

However, the concept lattice is a much more comprehensible representation allowing direct insight into the structure of the original relation. For example, we can immediately derive from Figure 11-1 that there are two disjoint sets of objects: O_1 that has attributes A_1 and A_2 and objects O_2, O_3, and O_4 share the other attributes. Furthermore, among O_2, O_3, and O_4, O_4 has all attributes of O_2 and O_3 but not vice versa. A_3 and A_4 is common to all objects O_2, O_3, and O_4. This information can, of course, also be derived from the table, but for larger tables, this is more difficult.

In the following sections, it is discussed how concept analysis can be applied to detect atomic components. The approaches differ by the kinds of objects and attributes considered, the way interferences are handled, and how the concept lattice is interpreted.

11.1.2.2 Lindig and Snelting's Approach

Name	Concept Analysis by Lindig and Snelting
Reference	Lindig and Snelting (1997)
Domain	Object Reference View
Range	ADO
Disjoint Clusters	Yes

A formal context for concept analysis in the approach of Lindig and Snelting (1997) consists of subprograms as objects and global variables as attributes. The underlying relation is the variable reference, i.e., a subprogram *S* has attribute *V* if and only if *S* references variable *V.* An atomic component candidate is a concept, i.e., shows up as a maximal rectangle in the table of the variable reference relation. For example, if Table 11-1 represents a variable reference relation, then ($\{O_1\}$, $\{A_1, A_2\}$) forms a candidate. The maximal rectangles in the table, however, need not be completely filled — not every subprogram in a component uses all variables, and not all variables are used by all procedures — as long as it corresponds to a independent sublattice in the concept lattice. An **independent sublattice** is connected to other sublattices only via the top and bottom elements; it has a single entry (from the bottom element) and a single exit (to the top element), so to speak. The concepts associated with the nodes $<O_4>$, $<O_2, A_5>$, $<O_3, A_6, A_7, A_8>$, and $<A_3, A_4>$ in Figure 11-1, for example, form an independent sublattice. $<O_1, A_1, A_2>$ forms another independent sublattice. Given an independent sublattice, the constituents of the proposed candidate are all objects (subprograms) and attributes (variables) of the concepts in the sublattice. Hence, $\{O_2, O_3, O_4, A_3, A_4, A_5, A_6, A_7, A_8\}$ and $\{O_1, A_1, A_2\}$ are two candidates detected by concept analysis.

Ideally, the variable reference relation is horizontally decomposable, i.e., the corresponding lattice consists of independent sublattices only. However, in reality, there are often subprograms in one component that directly access variables of

other components, which results in interferences in the concept lattice that prevent us from identifying components as independent sublattices. The subprograms *S* in Figure 11-2, for example, accesses two variables V_1 and V_2 that belong to different sublattices. As a consequence, the concept labelled with *S* causes an interference among two sublattices. Snelting also notes that such interferences can automatically be detected and then — in principle — be removed by program transformations. For example, the access to V_1 by *S* could be replaced by a call to an appropriate accessor function of the atomic component containing V_1. Snelting notes that even in the case of many interferences the system can be modularized by means of so-called block relations (also called *weak congruencies*) that correspond to rectangle shapes in the table and induce a factor lattice (see Lindig and Snelting, 1997, for details). However, even though interferences are automatically detectable, the user has to be called on or further heuristics have to be used to decide how the interferences should actually be handled.

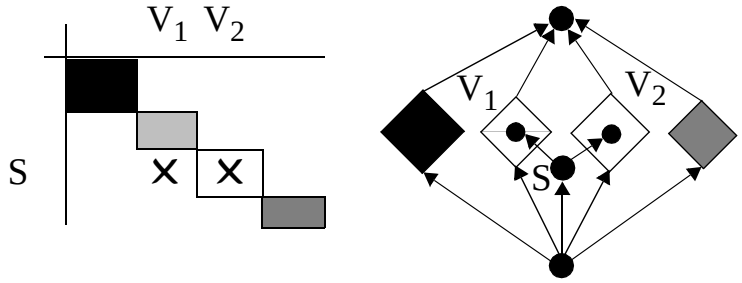


Figure 11-2. **A horizontal decomposition and an interference.**

Lindig and Snelting report on a case study in which concept analysis was applied to a 20-year old aerodynamics system used for airplane development after several manual restructuring efforts had failed (1997). The program was written in Fortran and consisted of about 100 KLOC, 317 subroutines, and 492 global variables in 49 COMMON blocks. The concept lattice for this program contains 2249 elements and is full of interferences such that it could not be horizontally decomposed. Several attempts to restructure at least some parts of the system and to use block relations and other approaches failed, too. It was decided to cancel the reengineering project and to develop a new system from scratch.

11.1.2.3 Siff and Reps's Approach

Name	Concept Analysis by Siff and Reps
Reference	Siff and Reps (1997)
Domain	Signature View and non-abstract usage information
Range	ADT
Disjoint Clusters	No

Siff and Reps use concept analysis to detect abstract data types. The objects are subprograms as in the approach of Lindig and Snelting while the attributes are signature relationships and accesses to record components as opposed to variable references. A subprogram, *S*, has attribute *param T* if and only if *signature-type (S, T)* holds. *S* has attribute *access T* if and only if:

$$S \in \text{predecessors}(T, \{\text{local-obj-of-type, signature-type}\}, \text{non-abstract})$$

Siff and Reps also consider disjunctions of attributes and negative attributes. Disjunctive attributes allow the user to specify properties of the form "*A or B"*, like "internally accesses type *stack* or type *queue*". This may be useful when the user is aware of the similarity of two types. Negative attributes are a means to leverage the absence of an attribute in order to eliminate interferences. Consider the example in Figure 11-3, which describes in table (a) that subprograms S_1 and S_2 have parameter type T_2 and both internally access an instance of type T_2; likewise, subprogram S_3 has parameter types T_1 and T_2 and internally accesses an instance of type T_1. Table (a) is decomposable in three concepts ($\{S_1, S_2\}$, $\{param\ T_2, access\ T_2\}$), ($\{S_3\}$, $\{param\ T_1, access\ T_1, param\ T_2\}$) and ($\{S_1, S_2, S_3\}$, $\{param\ T_2\}$) as shown by the shaped rectangles. Suppose the programmer breaks the information hiding principle and subprogram S_3 also internally accesses an instance of type T_2. Then, a concept ($\{S_1, S_2, S_3\}$, $\{param\ T_2, access\ T_2\}$) replaces the concepts ($\{S_1, S_2\}$, $\{param\ T_2, access\ T_2\}$) and ($\{S_1, S_2, S_3\}$, $\{param\ T_2\}$) and the concept ($\{S_3\}$, $\{param\ T_1, access\ T_1, param\ T_2, access\ T_2\}$) replaces ($\{S_3\}$, $\{param\ T_1, access\ T_1, param\ T_2\}$). The replacing concepts overlap such that the two abstract data types that originally corresponded to distinct concepts can no longer be easily detected from the concept lattice. The introduction of a negative attribute $\neg$ *access* T_1 reveals the original atomic component consisting of S_1, S_2, and T_2 again

as shown in table (c): Because S_1 and S_2 do not internally access an instance of type T_1, a new maximal rectangle shows up in the table corresponding to the concept ($\{S_1, S_2\}$, $\{param\ T_2, access\ T_1, \neg\ access\ T_2\}$). However, adding negative attributes can only lead to new concepts in the lattice but never to removal of concepts. Hence, concept ($\{S_1, S_2, S_3\}$, $\{param\ T_2, access\ T_2\}$) is still present.

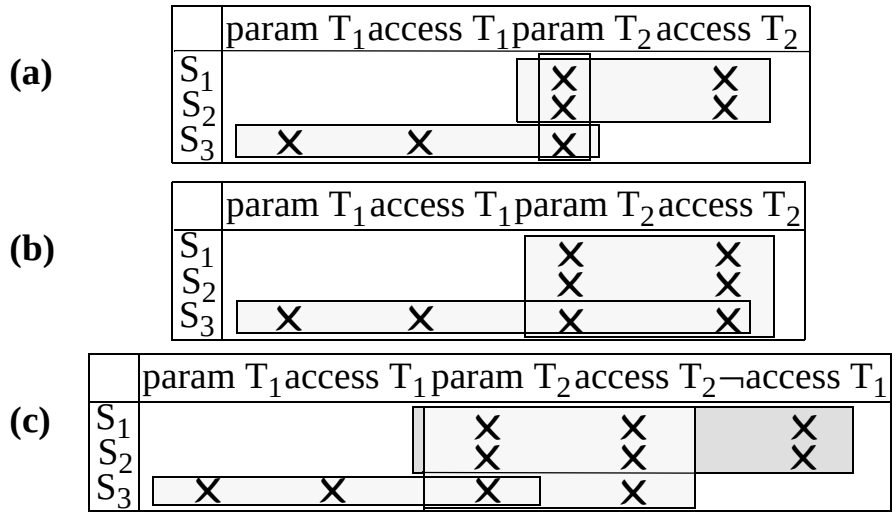


Figure 11-3. **Example for negative attributes.**

Not every negative attribute does help to reveal relevant concepts. For example, adding a negative attribute $\neg\ access\ T_2$ fails to detect the component consisting of S_1, S_2, and T_2 in the example above since S_1, S_2, as well as S_3 internally access T_2. Furthermore, negative attributes can lead to many additional concepts, i.e., to many new potential candidates. Siff has, therefore, refined the first idea of adding negative attributes for all attributes (**fully complemented context**), published in 1997, by adding them only when they can be used to make a difference between two overlapping concepts and when there is a negative attribute that can be used to distinguish a concept from all other concepts (Siff, 1998); the latter is called the **uniquely complemented context**.

Whereas Lindig and Snelting consider only independent sublattices as candidates, Siff and Reps treat each concept as a potential candidate. The number of concepts, however, can be huge and not all possible combinations of concepts actually make sense. Overlapping atomic components are rather rare according to

our experience with components compiled by software engineers for several systems described in Section 6.1.1. For this reason, Siff and Reps propose to partition the concept lattice into non-overlapping concepts; in other words, each subprogram is assigned to exactly one candidate. Formally, given a formal context $(O, \mathcal{A}, \mathcal{R})$, a **concept partition** is a set of concepts whose extents form a partition of O. That is, $P = \{(O_1, A_1), (O_2, A_2), \ldots, (O_k, A_k)\}$ is a concept partition if and only if

$$O = \bigcup_{1 \le i \le k} O_i \wedge \forall (i \neq j)\ O_i \cap O_j = \varnothing$$

Each concept partition is a possible system decomposition and can be validated by a maintainer. In the simple example of Figure 11-1 on page 357, there is only one possible concept partition $(\{O_1\}, \{A_1, A_2\}), (\{O_2, O_3, O_4\}, \{A_3, A_4\})$ since the two concepts labelled with O_2 and O_3 both contain O_4. But generally, there can be a large number of possible partitions.

The extent of each concept in a partition identifies the subprograms that belong to an atomic component candidate. However, what are the types that belong to the candidate? Though not explicitly said in their paper, we may assume by the examples Siff and Reps are using that they add those types to the candidate that are associated with the attributes in the intent of a concept. Recall that the attributes Siff and Reps consider are not types directly but the kinds of type usages, such as *accesses record components of T, has return type T,* and *has parameter T.* That is, the candidate formed for a concept $(\{S_1, S_2, S_3\}, \{access\ T_1, return\ T_2\})$, for example, is $\{S_1, S_2, S_3, T_1, T_2\}$.

Selecting candidates from concept partitions as proposed by Siff and Reps has the following disadvantages:

1. A type may be assigned to more than one candidate within a given partition.
2. Not all types are assigned to candidates.
3. There are many possible partitions.
4. The 1:1 relationship between a concept and a candidate is often too strict.

Ad (1): Because Siff and Reps partition the objects, i.e., the subprograms, rather than the attributes, i.e., the types, candidates may result that overlap in their sets of types. For example, if the two concepts in the lattice of Figure 11-4 labeled by T_2 and T_3 are selected, the resulting candidates overlap in type T_1. As a matter of fact, it is even worse because Siff and Reps do not only consider types as such as attributes but the way a type is related. That is to say, because the two distinct attributes *param T* and *access T* can be in different concepts, the type is added to different components. In the systems we have investigated, types do always belong to one component at most. Overlap in subprograms is more frequent — yet rare.

Ad (2): Consider the example lattice in Figure 11-4. The concept labelled by T_1 can be selected for a partition. Then, its (transitive) subconcepts cannot be selected for the partition because their extents are subsets of the concept labelled by T_1. As a consequence, the types T_2 and T_3 that are associated with subconcepts will not be assigned. They may only be assigned in a different partition, i.e, one that selects these subconcepts. Hence, it is not sufficient to look at a single partition in order to assign all types.

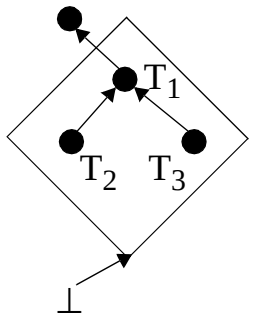


Figure 11-4. **Lattice interpretation.**

Ad (3): Siff and Reps report for the fully complemented context of a smaller program consisting of 372 subprograms and 8 user-defined types altogether 34 concepts and 63 possible partitions of the lattice. For another program with 26 subprograms and 3 user-defined types, 28 concepts were detected and 153 possible partitions of the fully complemented lattice were proposed. The high number is mainly due to the introduction of negative attributes in a fully complemented context. However, even for a uniquely attributed context according to the refine-

ments discussed above, the approach was not usable for systems at the 30KLOC level because of the huge number of identified partitions (Siff, 1998). For this reason, Siff and Reps propose to select the partitions interactively. An improvement was proposed by Graudejus (1998). He suggests to select only the concepts located directly below the top element of the concept lattice and achieved better results. Concepts directly under the top element in the lattice comprise all subprograms of their transitive subconcepts and consist of only a few types (the higher a concept the more objects and the less attributes it has).

Ad (4): Forming a candidate by a single concept is very strict because then all subprograms must be related to all types in the concepts (positively or negatively) otherwise it would not be a concept. This constraint is acceptable for abstract data types because an abstract data type usually consists of one single type only and in fact all accessor functions of an abstract data type should be related to this type. For abstract data objects, however, this is too rigid. Not necessarily all subprograms of an abstract data object reference all of its variables.

11.1.2.4 Sahraoui et al.'s Approach

Name	**Concept Analysis by Sahraoui et al.**
Reference	Sahraoui et al. (1997)
Domain	Object Reference View
Range	ADO
Disjoint Clusters	No

Sahraoui et al. apply concept analysis to the object reference view in order to detect abstract data objects. Their approach differs from Lindig and Snelting's approach by distinguishing different kinds of variable references and by the interpretation of the concept lattice.

Variable references are distinguished into setting and using a variable. The way a variable is referenced can help in two cases:

1. A global variable that is never modified can be considered a constant and be removed from the variable reference view. Sahraoui et al. note that this decision is not easy to make when pointer arithmetic is used. However, why pointer arithmetic for global variables should be a problem is unclear. What they prob-

ably meant was that the variable could be modified when passed as a (simulated) reference parameter to a function that modifies the dereferenced parameter. A conservative decision without need for data flow analysis is to exclude global variables that are never modified directly and whose address is never taken.

2. The specific kind of reference can also be used for method identification (see below).

Sets and *uses* of variables could also be used as two different attributes for concept analysis, i.e., one could make a distinction between subprograms that only use a variable and subprograms that set a variable. Though not explicitly said but suggested by the example they give, Sahraoui et al. do apparently not distinguish *sets* and *uses* for concept analysis as such but have only one attribute for referencing a variable.

The identification of atomic components in the concept lattice is divided into three steps:

1. Identification of sets of variables.
2. Merging of overlapping sets of variables.
3. Identification of subprograms.

Identification of sets of variables. The first step is based on the concept lattice for the variable reference relation. In the paper of Sahraoui, the concept lattice is computed for the inverse variable reference relation. This does not have any influence on the outcome since attributes and objects are interchangeable for concept analysis (the concept lattice is flipped upside down then). However, we prefer to follow the conventional way of treating the subprograms as objects and the variables as attributes.

In order to identify sets of variables from the concept lattice, a set, *NS*, that contains the not-yet-selected variables is used. In the initial state, *NS* contains all variables. The iterative identification process starts at the top element of the lattice and stops when *NS* is the empty set, hence, when all variables have been assigned to a candidate. In the case of equal cardinality of subprogram sets, concepts with fewer variables are preferred in order to avoid large objects. If these two criteria are still not sufficient to decide what concept should be selected, a

priority is given to the cluster C that has the higher cardinality of the set *intent* (C) $\cap$ NS. Each time a group is selected, the variables it contains are removed from NS. Groups with *intent* (C) $\cap$ $NS = \varnothing$ and groups consisting of only one variable of a basic type are ignored.

Merging overlapping sets of variables. The previous step can lead to overlapping sets of variables. These overlapping sets may be merged. In order to detect overlapping sets of variables, concept analysis is applied to the formal context ($\mathcal{O}$, $\mathcal{A}$, $\mathcal{R}$) where $\mathcal{O}$ consists of the sets of variables, $\mathcal{A}$ consists of the global variables, and (o, a) $\in \mathcal{R}$ if o contains a. Two sets of variables overlap when they have a common superconcept other than the top element. Sahraoui et al. merge two sets of variables only if they have at least two common variables. As a consequence, two candidates may overlap in one variable.

Method identification. Once the sets of variables for the candidates have been established, the subprograms are identified. Interestingly enough, Sahraoui et al. do not consider the concept lattice in order to assign subprograms to candidates. Instead, they propose three rules based on the following definitions:

Let V be the set of global variables, $SV \subseteq V$ be the set of variables identified by the previous step, and F the set of functions. For each function $f \in F$, two sets *referenced-by*(f) and *set-by*(f) are defined as follows:

referenced-by(f) = referenced-objects(f) $\cap$ SV
set-by(f) = set-objects(f) $\cap$ SV where set-objects(f) = $\{v \mid \text{obj-set}(f, v)\}$
(*referenced-objects* is defined in Table 3-4 on page 65 and *obj_set* is defined in Table 3-3 on page 51)

Note that *set-by*(f) is a subset of *referenced-by*(f). Based on possible categories for values of *referenced-by*(f) and *set-by*(f) relevant to assign a function to a candidate, three rules can be stated:

- If | *referenced-by*(f) | = 1, then f is assigned to the unique set of variables in *referenced-by*(f).
- If | *referenced-by*(f) | > 1 $\wedge$ | *set-by*(f) | = 1, then f is assigned to the set of variables identified by the previous two steps within the concept lattice that con-

tains the single variable in *set-by*(*f*) since a modification suggests a higher coupling than just reading the value of a variable.

- If | *referenced-by*(*f*) | > 1 ∧ | *set-by*(*f*) | > 1, then *f* cannot be clearly assigned to one set of variables. Sahraoui et al. propose to slice this function since their approach is aimed at migrating to an object-oriented language.

Note that these rules do not guarantee that no function is assigned to more than one candidate because the sets of variables merged by the previous step may still overlap in one variable and if the function happens to set only one variable that is in more than one set of variables, the function will be added to all these candidates. Moreover, in a pure reverse engineering approach, it is unclear what to do with functions in the third category. An attempt not explored by the authors could be to assign the function to the set of variables with the maximal number of variables referenced by the function. This strategy could also be used in the second category. It is questionable whether a single modification of a variable in one set of variables outweighs many uses of variables in another set of variables.

Sahraoui et al. motivate the top-down traversal of the concept lattice for component detection by the fact that the higher a group is in the lattice the higher is the cardinality of its subprogram set (the extent) and their hypothesis is that a set of variables can be considered forming an abstract data object if these variables are simultaneously accessed by a larger number of subprograms. However, counter-examples against this hypothesis are frequently used system state or mode variables, such as an error variable. Another consequence of the top-down traversal is that smaller sets of variables are preferred. The argument of the authors is that they want to avoid atomic components with a large number of variables. However, this is counter-intuitive in the context of concept analysis. The definition of a concept requires of all subprograms to reference all variables and since variables are rarely referenced by the same set of subprograms in reality, concepts with a huge number of variables do virtually not exist. Thus, the concepts we can expect to find do not comprise a very large number of variables and concepts with many variables represent higher cohesion among the subprograms in the concept than concepts with a lower number of variables.

11.1.2.5 Canfora et al.'s Approach

Name	Concept Analysis by Canfora et al.
Reference	Canfora et al. (1999a)
Domain	File Usage View (see below)
Range	Persistent Objects
Disjoint Clusters	Yes

In a case study, Canfora et al. used concept analysis to detect persistent objects, consisting of data files and their accessor routines, in a COBOL system for distributed system migration (1999a). The objects are COBOL programs and the attributes files. The relationship is defined as $(o, a) \in \mathcal{R}$ if program *o* accesses file *a*.

Data files and file accesses have not been captured by the resource usage graph used so far. The extensions, however, are straightforward:

- adding new entity types *data file* and *program*
- adding a new relationship type *file access*
- adding a new view *file usage view* that describes which programs access data files

The authors used concept analysis in this case study only as the main driving method; they also used ideas underlying other techniques to reduce the number of interferences in the concept lattice, which distinguishes their approach from the approaches discussed in previous sections. Some of the interferences could be eliminated by

1. grouping synonymous files, where synonymity was detected manually (Canfora, 1999b),
2. considering only files corresponding to application domain objects,
3. excluding programs accessing only one file, and
4. excluding programs with many file accesses after a manual inspection of the lattice and the source code.

The persistent objects were detected in the concept lattice as independent sublattices, i.e., sublattices only connected to the bottom and top element in the lattice.

The resulting concept lattice was simple enough to be inspected manually and to find further persistent objects.

Though a first step toward a combination of existing techniques, this approach is still limited. Firstly, two of the ideas, namely, 1 and 3, are rather trivial. Secondly, observing the second idea leaves many programming domain objects undetected. Furthermore, the approach relies heavily on manual inspection of the lattice. This may be possible at the level of programs and files. However, at the level of global declarations, at which atomic components are detected, the lattices are much bigger.

11.1.2.6 Summary of Concept Analysis

Godin et al. (1995) showed that the worst case space complexity of the concept lattice is $O(2^k \times |O|)$ for the finite upper bound k on the number of attributes for an object in the formal context. Thus, the space complexity is basically linear. Our own experiences with the resource usage graphs for the systems we investigated indicate that k is small when we equate objects with entities and attributes with relationships. But this is not so when negative attributes are considered. Negative attributes are the complement of a relationship and since the plain relationship is sparse, its complement has many members. This effect became already visible in the discussion of the approach of Siff and Reps.

On the computational side, concept analysis has an exponential time complexity in the worst case. Experiences for atomic component detection indicate that it takes cubic time on average (Snelting, 1999).

Concept analysis has a sound mathematical background and the insight into the relationships among system components that it can offer makes it an interesting technique for atomic component detection. On the other hand, its demanding time complexity jeopardize scalability of concept analysis for larger systems. Furthermore, it is still not clear whether the computational costs for concept analysis really pay. Horizontally decomposable sublattices can also be identified by the simple *Global Reference* heuristic at much lesser effort using a union-find algorithm. The problem of concept analysis are the interferences in the concept lattice — as for any other of the basic structural techniques. Moreover, *Delta IC* is very similar to concept analysis. The set of closely related subprograms corresponds

either to a single concept or — if not all subprograms in the set access all variables — to a set of concepts that are in a subconcept relationship. *Delta IC* tolerates interferences by means of the quotient of the numbers of *closely-related-subprograms* and *related-subprograms* (see Section 5.7).

The overview and analysis of the diverse approaches using concept analysis in this section has revealed that detecting atomic components by means of concept analysis is a field of its own that still requires a lot of research. This field is too wide to be explored in this thesis in concert with the other techniques. For this reason, concept analysis was not investigated in this thesis further.

11.1.3 Dataflow-based and Domain-based Approaches

Only few techniques exist that leverage data flow information and there is only one approach — to my knowledge — that is based on information about the application domain.

11.1.3.1 Valasareddi and Carver

Valasareddi and Carver identify objects in a variant of the program dependency graph that describes control and data dependencies (1998). A node in the program dependency graph represents a statement, an edge represents either a control or data dependency. The dependency graph is restructured by merging nodes with high cohesion. Within the resulting dependency graph, each node represents a potential accessor function and the variables referenced by the node constitute an object.

The advantage of this approach is that it may identify components below the global code level, i.e., at the level of statements within functions. However, because no evaluation of this approach is given, one cannot judge whether meaningful components can be identified at this lower level. Moreover, the authors do not specify how they deal with nodes whose sets of referenced variables overlap. If these sets overlap, overlapping candidates are proposed.

11.1.3.2 Gall and Klösch

Gall and Klösch's approach is based on data flow information as well as on domain information. The approach starts at dataflow diagrams (domain informa-

tion) and then follows part-type relationships among data store entities of the dataflow diagrams and user-defined records and pursue data dependencies among record components to identify objects with application semantics (1995). Any found data store (file) is considered because information that is stored in a file seems to be essential for the program; a user-defined record that (transitively) depends upon data stores is considered because it is related to something already considered application-related. The accessor functions of atomic components are identified as subprograms that reference files and record variables selected.

This approach is primarily aimed at abstract data objects that have application semantics in order to recover an application-oriented object model. It falls short of identifying components that rather belong to the programming domain, like stacks and queues. Though these kinds of components are at a lower level, they are necessary to understand the system as a whole and are often better suited for reuse than application-related components. Moreover, it is not clear what the coverage of this approach is: Are all and only application-related objects found? In analogy to program slicing where slices often cover about 80% of the program, it may be the case that more or less all record types are considered application-related. Hence, the question arises whether the approach excludes programming-domain components effectively.

11.2 Semi-Automatic Methods

This section describes further semi-automatic approaches that integrate the user in the detection process.

11.2.1 Müller et al. (Rigi)

Müller et al. point out that architecture recovery cannot be fully automated (1993); thus, the role of the human software engineer constitutes a central and integral part of architecture recovery and, consequently, tools for architecture recovery should integrate the user. On the other hand, as much as possible should be automated. Therefore, the tool supporting their approach, namely, Rigi allows human intervention and offers automatic operations for subsystem detection, too. The available operations for subsystem detection are removal of omnipresent

entities as well as composing by interconnection strength, common client/suppliers, centricity, and name. Furthermore, metrics can be used to assess the structure of the subsystem decomposition and exact interfaces of subsystems can automatically be derived. Moreover, because many analyses for architecture recovery are system-dependent, Rigi offers a scripting language that allows a maintainer to write his or her own clustering techniques.

Because the semi-automatic method proposed in this thesis uses Rigi for visualization and user interaction, Müller's and my approach have a great deal in common. However, my work focuses on atomic component recovery and, therefore, offers a wider selection of atomic component recovery techniques, while Müller's work is also aimed at hierarchical subsystems. Moreover, the only way of combining different techniques in the original Rigi is to apply the techniques successively, whereas the new Rigi supporting my semi-automatic method, namely, extended Rigi, handles alternative results in parallel and offers deep set operators and composition to combine these results.

11.2.2 Kazman and Carrière (Dali)

Another extension of Rigi is Dali, developed by Kazman and Carrière (1997). Dali uses an SQL database to store information about systems to be reverse engineered. As a consequence, SQL can be used to specify clustering patterns. Kazman and Carrière distinguish two different levels of patterns: application-independent patterns and application patterns. Application-independent patterns are used to aggregate declarations according to the language semantics, like local variables with their enclosing function or data and function members with their class (another type of low-level application-independent patterns filters noise introduced by insufficient tools for parsing and semantic analysis; these patterns are not needed when a capable C++ frontend is available). The purpose of application patterns is to group functions and classes to subsystems. These patterns leverage naming conventions or are enumerations of related elements.

The advantage of Dali is its support for SQL queries to specify clustering patterns; patterns can be written in a declarative manner and in a language that is widely used. The patterns can be viewed as a specification of the system structure and be used to re-generate the visualization of the system when the system has changed (in which case, the patterns have to be updated accordingly). Moreover,

the patterns can be employed to group elements quickly instead of grouping them manually. In the case of patterns that are mere enumerations, however, this advantage is only limited. On the other hand, there is no analytic capability in Dali. Hence, the patterns have to be written by someone already familiar with the system and cannot be reused for other systems in many cases. Refinement of the results of a pattern is difficult because the user cannot interact with the system by direct manipulation since the refinement has to be done in the pattern. For the same reason, combinations of results are difficult. In principle, combining operators like deep union and intersection could be written in SQL as well using temporary tables and logical *or* and *and* operators, but the result (probably refined during validation) would have to be expressed as an SQL query afterward to obtain the advantages of a separate specification of the system structure as SQL patterns.

11.2.3 Yeh et al. (ManSART)

The extended Rigi has also much in common with ManSART, a tool developed at MITRE that supports architecture recovery (Yeh et al. 1997). ManSART visualizes different views of the system that are directly derived from source code, such as task-spawning views, dataflow between procedures and data files, and abstract data type views. In order to combine different views for presentation purposes, several operators are offered to the user. The purpose of the operators is to connect distinct views at different levels of abstractions (e.g., a task-spawning view with the abstract data type view), whereas the extended Rigi offers operators to find agreements and differences of component views or to unite two views where all views are at the same level of abstraction. In order to establish correspondence among concepts in different views when views are combined, a containment relation is used by ManSART that is based on source positions of statements implementing the concepts. When the extended Rigi combines two views, it can consider the base entities contained in components because the corresponding components are at the same level of abstraction.

11.2.4 Gall, Klösch, and Weidl

All approaches that have been presented in this section so far, including the one underlying extended Rigi, are primarily bottom-up approaches. The search starts at the level of declarations extracted from source code, which are then grouped

together by automatic techniques and human judgement. Gall, Klösch, and Weidl complement bottom-up clustering by a top-down approach (1998). They also use bottom-up clustering heuristics as described in Section 11.1.3. However, they go beyond a pure bottom-up process by using a domain model built by a human engineer (e.g., using the unified model language UML). The domain model describes the application concepts and their relationships. Part of the recovery process is to bind the domain concepts to the components found by the bottom-up phase. In order to establish this mapping, a similarity metric is used based on similarity of names of domain concepts and source code identifiers and on similarity of types (Weidl and Gall, 1998). Because it is not always possible to establish the mapping using the similarity metric only, the user is integrated in the binding process.

The domain model may be used to make the recovery process more goal-directed and may increase the chance to find components with application semantics. On the other hand, other programming concepts may be missed that are also necessary to understand the system or could be reused in another context. Moreover, the explicit domain model and a mapping from domain concepts to components in the source implementing these domain concepts is a valuable documentation. The bottom-up approaches discussed above also use a domain model — but it exists only in the head of the maintainer. However, building a domain model needs additional effort and the necessary degree of detail of a useful domain model is not known in advance.

11.2.5 Murphy, Notkin, and Sullivan (Software Reflexion Model)

Another top-down approach is the Software Reflexion Model by Murphy, Notkin, and Sullivan (1995). The Software Reflexion Model is to capture and exploit the differences that exist between the source code organization and the designer's mental model of the high-level system organization. An engineer defines a high-level model of the structure of the system and specifies how the model maps to the source. A tool then computes a software reflexion model that shows where the engineer's high-level model agrees with and where it differs from a model of the source. The primary purpose of this technique is to streamline the amount of time it takes for someone unfamiliar with the system to understand its source code structure.

11.3 Plan Recognition Techniques

A research area that would directly profit from atomic component detection is plan recognition. A plan is a stereotypical coding pattern for algorithms or data structures used to implement frequently used concepts, comparable to design patterns at the design level. For example, a *stack* is often implemented by a stack pointer and an array; *push* adds its argument to the array and increases the stack pointer. Such known coding patterns can be used to identify the pieces of codes that implement the concept. For example, van Deursen, Woods, and Quilici used plans for leap year calculations in order to find code pieces that would incorrectly handle the leap year 2000 (1997). There are many approaches to plan recognition that can be distinguished by the intermediate form they work on (source code, abstract syntax tree, lambda expression, control flow graph, or data flow graph), the underlying recognition technique (textual pattern matching, tree matching, graph parsing), and the classes of plans (control concepts, data structure concepts) (Wills, 1992; Harandi and Ning, 1990; Hartman, 1991; Quilici et al. 1997). A detailed classification of existing plan recognition techniques was compiled by Wills (1992).

The difference to atomic component detection is that plan recognition does not only identify the pieces of a concept (functions, variables, types) but is also able to assign a concept to these pieces. However, automatic plan recognition is computationally demanding. Atomic component detection techniques can be used as pre-analyses that identify the cohesive elements; then, plan recognition techniques could investigate each single atomic component individually, which would reduce the search space by an order of magnitude.

11.4 Connector Detection Techniques

Research for techniques to detect architectural connectors has mostly concentrated on connectors for parallel and distributed systems, such as spawning processes, remote procedure calls, socket calls, and so forth. These kinds of connectors can often be identified as a set of function calls of libraries implementing synchronization and communication means for parallel and distributed systems. Plan recognition techniques can be used to find these patterns. Harris et

al. identify such function calls within the abstract syntax tree using tree pattern matching (1995). Fiutem et al. additionally exploit data flow information in order to establish communication channels between different parts of a system and propose a language to specify plans for connector detection (1996).

However, most systems are sequential and even parallel systems have a large sequential part and would also profit from connector detection. Simple connectors, like procedure calls, are trivially detected. Calls via function pointers need points-to analyses. Actual data bindings via shared variables would also require control and data flow analyses. Moreover, atomic components can often be considered connectors between different parts of the system; e.g., one parts fills a queue and the other reads from the queue. I have no knowledge of any approach that investigates the role of atomic components as connectors.

Chapter 12 Conclusions

This chapter summarizes the contributions and conclusions of this thesis and proposes further research directions.

12.1 Conclusions

Section 1.4 has stated the scientific questions addressed in this work. This section briefly summarizes the answers to these questions given in previous chapters.

What published structural techniques exist and how can they be unified and classified?

Chapter 5 described a taxonomy for structural techniques for atomic component detection. The class of concept-based approaches was added to this taxonomy in Section 11.1.2. The class of structural techniques comprises:

- **Connection-based** approaches cluster entities based on a specific set of direct relationships (and their quality) between entities to be grouped.

  All the connection-based approaches can be unified by using the same generic algorithm for clustering based on a function *Connected_Entities* that yields the elements that are to be grouped with an entity. The approaches differ only in the exact specification of this function.
- **Metric-based** approaches cluster entities based on a metric using an iterative clustering approach. The metric-based approaches are based on connections, too, but they differ from connection-based approaches by the degree of free-

dom that is offered by the metric parameters and the threshold that can be varied to find atomic components with varying confidence.

- **Graph-based** approaches derive clusters from a graph by means of graph-theoretic analyses. The difference to connection-based approaches is that the whole graph has to be considered whereas connection-based approaches regard only direct relationships between entities in order to decide whether they should be grouped.
- **Concept-based** approaches use concept analysis to compute a lattice of concepts. They use heuristics to detect atomic components within this lattice. The approaches differ in the objects and attributes considered and the way the concept lattice is interpreted to retrieve components.

Beyond structure-based techniques, there are other automatic techniques based on data flow information and domain information (Section 11.1.3). Semi-automatic techniques integrate the user in the detection process. Most of the semi-automatic techniques are bottom-up approaches that start at the global code level (Chapter 9, Sections 11.2.1 and 11.2.2). Top-down approaches start at a high-level model of the system and try to map the concepts in the high-level model to the source (Sections 11.2.4 and 11.2.5). Combined semi-automatic techniques tackle the problem of component recovery from both sides. An overview of the taxonomy and the respective techniques for atomic component detection is shown in Figure 12-1. Techniques that are in more than one class of the taxonomy are marked by ⇨.

What is the recall rate and precision in atomic component detection of published techniques?

In Chapter 6, an evaluation of the basic techniques was described in which the results of the automatic techniques were compared to components found by software engineers. The evaluation has revealed the following points:

- The effectiveness of a technique depends upon system characteristics, like degree of information hiding, proper module decomposition, and layering.
- None of the investigated techniques has a sufficient recall rate. The best recall rate we obtained was 75% of the abstract data objects. In the worst case, the best technique reached only a recall rate of 34% (*Similarity Clustering* for ADTs of Aero; Figure 7-22 on page 237).

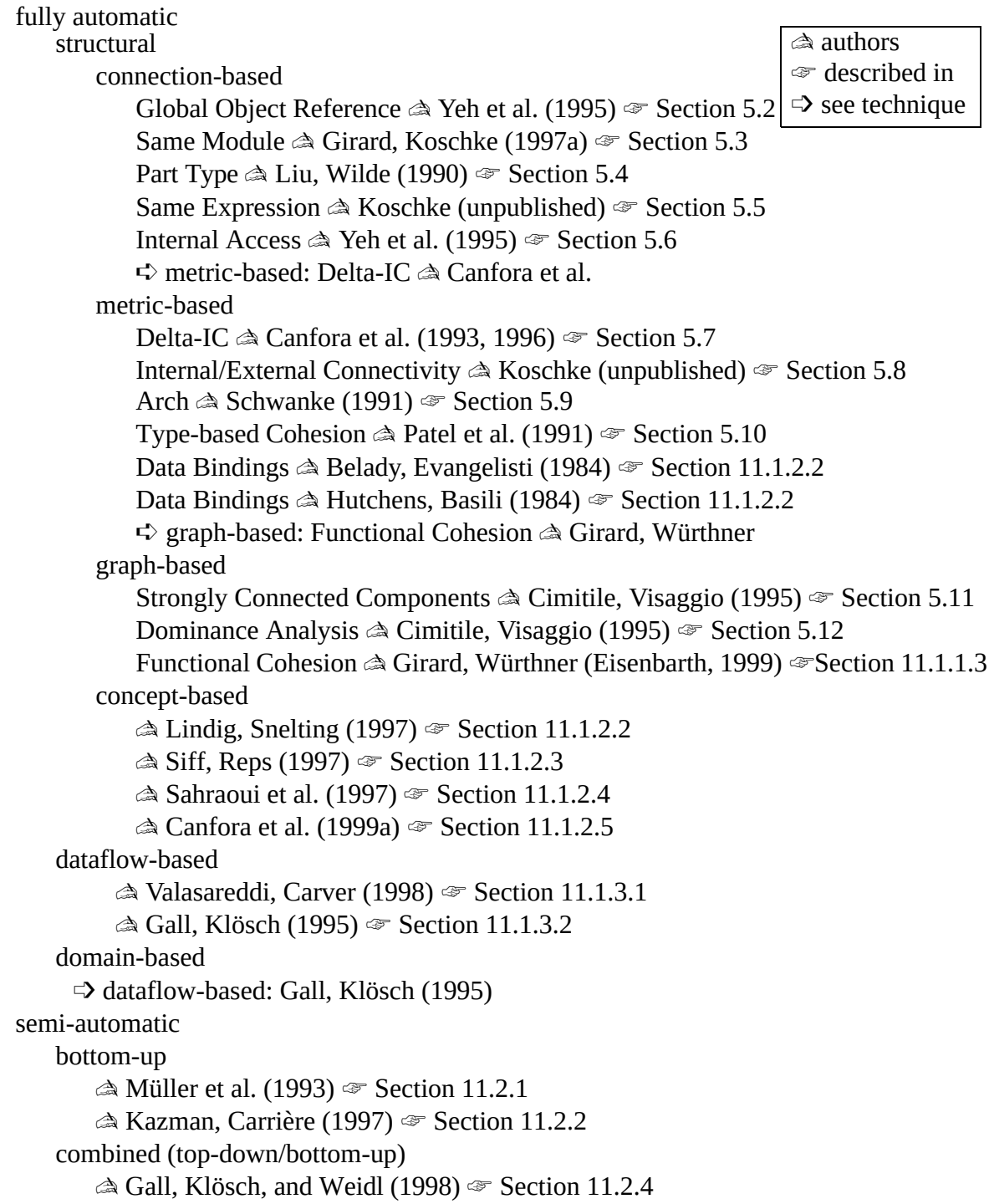


Figure 12-1. **Taxonomy for atomic component detection techniques.**

- Many candidates the techniques provide correspond only roughly to the reference components; i.e., elements of these atomic components were superfluous or lacking.
- Combining the automatic approaches instead of using a single approach, one would significantly improve the discovery of the reference components.
- Yet, between 35 and 50 percent of the components still could not be completely found even by uniting the results of the techniques. However, the components may at least partially be matched.
- It turned out that 42% of the ADO candidates and 41% of the ADT candidates classified as false positives in the evaluation could indeed be considered correct positives after an analysis of the false positives for Aero, Bash, and CVS and a selection of automatic techniques; they were either too small to be considered, simply overlooked in the manual process, or represented alternative views.
- We found common patterns of false positives in all systems that could be used to filter out false positives from the set of candidates.
- Whereas the groups of software engineers needed about 20 - 35 hours to compile the list of atomic components for each of our subject systems (except for Mosaic, which was used in the experiment), each atomic component produced by the techniques can be checked by software engineers within minutes. To browse the whole list of false positives of all automatic techniques, we needed less than 6 hours per system. The time needed for validation can even be reduced by merging similar candidates of different techniques based on the partial subset relationship because there were many similar false positives among the candidates.

How can these techniques be improved individually?

Chapter 5 proposed several smaller improvements for published techniques. One of these techniques, namely, Schwanke's Arch approach, was enhanced in so many ways that it can be considered a new technique. The extended technique, *Similarity Clustering*, is described in Chapter 7 in detail. The properties of *Similarity Clustering* are summarized here:

- *Similarity Clustering* is the most general approach described in this thesis. All connection-based techniques can be subsumed under it. It can detect all kinds of atomic components. It goes beyond other approaches in that it also considers relationships to common third entities and informal aspects.

- *Similarity Clustering* can be used both to search for specific user-defined patterns and search for similar patterns of already found atomic components.
- The adjustable parameters of *Similarity Clustering* offer more flexibility. If atomic components similar to those already found are to be searched for, the parameters can automatically be calibrated on the known components using traditional optimization techniques, such as Simulated Annealing or Gauß-Seidel optimization.
- *Similarity Clustering* is one of the most effective techniques as far as the recall rate is concerned. On the other hand, it had also more false positives than other approaches in the evaluation (except for Aero, which has more). However, this is partly because the same threshold was used for all atomic components when the candidates were retrieved from the tree of clustered entities generated by *Similarity Clustering*. In an interactive application, the maintainer would begin the validation at the leaves and move up the tree toward the root until a node is reached for which the component membership is doubtful; i.e., thresholds individual to components would be used.
- For all techniques other than *Similarity Clustering*, there is one single criterion used for clustering. Hence, the reason why a technique has grouped entities together is obvious. This is less obvious for *Similarity Clustering* when the similarity metric considers several aspects at the same type, which may complicate validation of candidates proposed by *Similarity Clustering*.
- Time and space complexity for *Similarity Clustering* is basically linear to the number of entities considered when informal information is excluded (assuming an upper constant limit of neighbors an entity can have). However, when informal information is used, each entity has to be compared to any other entity resulting in a time complexity of $O(n^2)$.

How can the techniques be combined?

Chapter 8 described high-level operators like deep intersection, union, and composition offered to a maintainer in order to combine the results of the techniques instead of combining the methods technically. This way, new techniques can be added with very little effort and the maintainer has all flexibility to combine the analyses. The specification of the combining operators was extended to hierarchical subsystems as a consequence to the wish of many participants of the experi-

ment described in Chapter 10 to have a grouping mechanism beyond atomic components in order to group related atomic components.

Another way of combining the techniques was presented as the voting approach in which the agreement of each technique is polled, weighted, and summarized to a total agreement whether a given cluster is a promising candidate. In order to add a new technique to the voting approach, its underlying heuristic has only to be expressed as a metric.

How can the user be integrated in atomic component detection?

In Chapter 9, an incremental semi-automatic method was described that integrates the user into atomic component detection. The analyses, selected by the user, are used to propose candidates that are then validated by the user. The information added by the maintainer is used by the techniques in the next iteration. In order to realize this iterative process, the techniques had to be enhanced to work incrementally. Chapter 9 gives also some advice in which order the analyses should be applied.

Do automatic techniques support a maintainer in atomic component detection?

Section 10.3 has presented a controlled experiment in order to find out whether the automatic techniques are helpful within the semi-automatic method. The evaluation of the experiment could neither show a positive nor a negative effect on atomic component detection. However, the following restrictions of the experiment restrain us from generalizing the results too far:

- The system used for this experiment was well structured such that the experimental subjects using automatic techniques had only little advantage.
- The extended Rigi used within the experiment did not offer all the functionality proposed in this thesis. Many improvements were only inspired by the experiment.
- The experimental subjects were all well trained students of computer science and had grades above the average. Less talented programmers might profit more from automatic analyses.

Despite of these considerations, there are two fundamental realizations one has to be conscious of: Firstly, the semi-automatic method can only be as good as the underlying analyses and secondly, the effort of validating candidates is a constant factor that cannot be eliminated. Consequently, a way to improve atomic component detection with the semi-automatic method is to use more reliable and covering analyses. More reliable techniques are needed to reduce the number of candidates to be validated; more coverage is needed to find as many components as possible in order to avoid manual search. The prototype supporting the semi-automatic method used structure-based techniques only and inherited their weaknesses that are described in Chapter 6 and Chapter 7. However, even with more sophisticated automatic support, the effort for component recovery can only be reduced to the constant factor of validation. Validation will always be necessary because I doubt that we can ever find absolutely reliable techniques since the criteria for cohesive components are vague to some degree and legacy systems rarely employ information hiding.

However, if more reliable techniques are available, the semi-automatic method and its realization in the extended Rigi is flexible enough to integrate these new techniques quickly. The method as such and the way how user interaction is supported would not have to be changed.

Are the techniques and methods for atomic component detection discussed in this work also helpful for other typical maintenance tasks?

In Section 10.4, a case study was described that was performed to investigate the ability of the extended Rigi to support maintenance tasks other than those it was originally designed for, namely, atomic component detection. The goal of this case study was also to learn what other types of automatic analyses would be useful for maintainers. At least two of the four tasks assigned to the participants in this case study and dealing with global name binding are supported by the extended Rigi or could easily be solved by simple enhancements. These analyses could return the results within a minute where participants needed up-to two hours. For the other tasks, at least Rigi's cross-reference and browsing capabilities were helpful. In order to find function clones or answer lifetime and protocol questions, more rigorous analyses are needed.

12.2 Future Research

This sections proposes future research directions based on the results of this thesis.

12.2.1 Data Flow Information

The evaluation of the automatic structural techniques has shown that their recall rate does not compare to human detection and in the experiment for the semi-automatic method, a positive effect of the structure-based techniques could not be shown. This is partly due to system characteristics; if programmers obeyed to the rules of information hiding, there are structural techniques, like *Internal Access* for ADTs and *Global Object Reference* for ADOs, that would reliably identify all atomic components. However, because programmers do break information hiding principles very often in practice, distinct concepts in the source code are merged to single candidates by the techniques. This is because structural techniques leverage only coarse information about the relationships among types, variables, and subprograms. Control and data flow information might be an avenue to come to finer-grained analyses with more reliability.

It was already mentioned that the assumption of *Part Type* that the part type is put into or retrieved from the container type could be checked by data flow analysis. *Same Expression* could be extended to group variables that are in the same program slice (a program slice comprises all statements that contribute to the value of a variable). Slicing techniques could also be used to check whether there are independent slices in a subprogram with respect to different global variables; if so, the subprogram likely performs multiple functions and should be excluded from clustering and be presented to the user instead. Excluding subprograms with multiple purposes avoids merging of candidates. Control flow analysis could reveal control dependencies among variables, which could be used as another grouping criterion.

On the other hand, data flow analysis is expensive both in terms of computational costs and the costs to build an infrastructure for data flow analyses. The advantage of structural approaches is that they are comparatively simple and fast.

12.2.2 Domain Knowledge

Programmers understand programs not only in terms of programming semantics; they also base their understanding on their knowledge about the domain. Actually, understanding a program involves to establish a mapping between the concepts in the domain and the concepts found in the code. In the semi-automatic method, domain knowledge comes into play by the user who selects appropriate analyses and validates their results. The structural techniques, however, do not use domain knowledge. On one hand, the analyses can therefore be used in different contexts. On the other hand, domain knowledge may improve automatic component recovery. A first approach by Gall et al. (1998) toward using domain knowledge for component recovery was already discussed in Section 11.2. In order to establish a mapping between domain model and recovered candidates, Gall et al. propose a similarity metric based on name similarity and type similarity. This approach still needs human intervention. Earlier approaches, like the one of DeBaud and Rugaber, were purely manual (1995). More research is necessary to explore further ways to establish the mapping. Meanwhile, at least some elements of a (partial) domain model would be helpful. For example, a domain model defines the used vocabulary. A vocabulary would be useful in order to let *Similarity Clustering* (which uses naming conventions) know that *Create_Account* is more similar to *Release_Account* than to *Create_List*. Moreover, if the implementation characteristics for certain domain concepts are known, plan recognition techniques could be used to find the concepts as implemented in the source code. Hence, atomic component detection and plan recognition could complement each other.

Furthermore, only in rare cases, a domain model exists, for example, created as part of a process toward a product line. If a domain model is only created for atomic component detection, it is not per se clear whether the additional effort for setting up the domain model really pays off. One may argue that domain models have additional benefits beyond component recovery, like documentation or support for reuse. However, at least for documentation purposes, a domain model may be too general since it describes a whole domain and not just the system at hand. Further research should be devoted to evaluate the costs and benefits of domain models, where the costs and benefits should not only be measured with respect to component recovery but also to other aspects like the value added to documentation and reuse.

12.2.3 Research Directions Concerning the Method

The semi-automatic method could be further improved by allowing more interaction. For example, the user should be able to annotate variables and record components as public once the constituents of the atomic component have been found; the automatic analyses would then exclude public entities from consideration. Another example for more interaction is that, in the case of abstract data type detection, the user should be able to pick a single data type for which the accessor functions are to be detected, i.e., only one candidate is created at a time. Currently, many candidates are created and consequently, distinct components are merged to a single candidate when there is a subprogram that apparently belongs to both components. Clustering only one abstract data type at a time would hence avoid erroneously large components. However, this would not solve the problem of merged components in abstract data object detection because ADOs mostly comprise several variables and not just one type as it is the case for most abstract data types. Alternatively, one could allow overlapping candidates instead of merging them (subprograms in more than one candidate should be highlighted then).

The experiment to evaluate the semi-automatic method should be repeated. The experimental design and the statistical analysis described in Section 10.3 can be reused. However, future experiments should take the structure of the system into account. That is to say, instead of a single independent variable, one would use two independent variables: tool-support and the shape of the system. Furthermore, more experimental subjects and preferably common programmers should be involved.

12.2.4 Role Identification

The future research directions proposed above are aiming at atomic component detection as such. Other research paths should take atomic component detection as a starting point. These paths should aim at the semantics of atomic components. While techniques for atomic component detection group related entities, more knowledge is necessary to understand the purpose, or role, of the components. Two levels of roles are associated with atomic components whose identification would help in program understanding:

- **intra-component roles**: roles of base entities within the atomic component
- **inter-component roles**: roles of atomic components within the system

12.2.4.1 Intra-Component Roles

The constituting base entities of a component can be distinguished into public and private elements. Public elements can be used by clients of the component, whereas private elements are hidden within the component. The decision of whether an entity should be considered private or public has to be made by the maintainer. Automatic analyses may only identify the current state.

Accessor functions of an atomic component can further be classified as follows:

- **constructor**: creates a new (instance of the) component (hints: call to memory allocation routines like *malloc*, setting record components with literals)
- **destructor**: releases an existing (instance of the) component (hints: call to memory deallocation routines like *free*)
- **selector**: returns information about an existing (instance of the) component without changing it (hints: the data of the component are not changed)
- **modifier**: changes the state of an existing (instance of the) component (hints: the data of the component are changed)

This is no strict classification because in real-world programs there are often routines that fall into more than one category, for example, due to efficiency considerations, a routine may act as modifier and selector. The categorization of a routine into the above classes describes a part of the impact of a routine call (only a part because a routine could also have side-effects) and is a useful information to a maintainer.

Further analyses using plan recognition techniques could reveal more detailed characteristics of individual accessor functions. For example, Hartman has developed a technique to recognize so-called control concepts like “do-loop”, “read-process loop”, “succeed-fail loop”, and “bounded linear search” (1991). Restricting recognition to control concepts allows for efficient detection. The characteristics of accessor functions in terms of their underlying (control) concepts could be used as indices in a framework that supports reuse. These characteristics can be used to retrieve the accessor functions from the base of reusable components.

12.2.4.2 Inter-Component Roles

Inter-component roles of atomic components describe their association to other components and their purpose within the system.

One major motivation for atomic component detection is migration of procedural programs to object-oriented programs (Fanta, 1999). In order to leverage the expressiveness of object-oriented languages and to simplify the system structure by removing redundant code, inheritance relationships should be identified for this kind of migration. Most legacy systems, however, are not designed based on the object-oriented paradigm and, therefore, inheritance cannot be recognized. On the other hand, in three of the five systems we have used to evaluate the automatic techniques and the semi-automatic method, namely, Aero, Bash, and XCoral, we have found atomic components that are actually so similar to each other that they could be implemented as types of the same class. Inheritance relationships may be identified via variant records (in C, as unions of structs), record types with record components of similar names and/or types and similar accessor functions. Here, several different reverse engineering techniques could be integrated: atomic component detection to identify cohesive components, concept analysis for identifying similar types (where record component names and/or types would act as attributes), and clone detection techniques to identify similar functions.

Another kind of relationship useful for documentation of data models and explicitly represented in object-oriented modeling languages is aggregation. Actually, aggregation is already present in the resource usage graph as the part-type relationship. Other associations may be identified as calls to accessor routines or references to data of other components. These associations may be a starting point for gathering more specific information about the role of associated components, like:

- one component C_1 is a wrapper to another component C_2 if all accesses to C_2 are via C_1
- one component C is a data store if C contains some kind of buffer and there are read and write accesses to C, but C does not access other components unless they are themselves data stores

- one component C is a connector between two components C_1 and C_2 if C is a data store and C_1 writes into C and C_2 reads from C (in Belady and Evangelisti's terminology, $<C_1, C, C_2>$ is a data binding via C)

The examples above represent simple design patterns (Gamma et al. 1994). Future research could investigate to which degree more complicated design patterns can be recognized. This would not only be useful for program understanding but also for validation whether a design pattern has actually been implemented correctly.

Recognition of most design patterns require control and data flow analyses, in particular, when non-global objects as instances of abstract data types are investigated that may pass through chains of function calls as actual parameters.

12.2.5 Protocols

The *exact interface* of an atomic component, *C*, can be identified as the declarations of *C* used outside of *C* and the declarations *C* uses from its context (Müller, 1993). The former constitutes the actual *syntactic interface*. Though deriving the actual syntactic interface provides interesting information, it falls short of a client's need for information on how to use the component. The interface of a component is conjoined with a protocol that specifies the allowable order of actions associated with the component. An action associated with a component is a call of one of its accessor functions or a reference to its data. For example, a client has to know whether it is necessary to call a constructor or a destructor and under which circumstances a call to a modifier or selector is allowed.

Assuming the protocol of a component is observed, hints on its form can be derived from the actual usage of the component within a given program. For example, one attempt toward protocol recovery can be made by extracting the actions associated with a component as regular path expressions from the program (Tarjan, 1981) and presenting them as finite state machines to the user. The user can then use the extracted order of actions in order to specify the protocol. Extracting protocol information as actions visible outside of the component considers the component a black-box. A complementary glass-box approach would look inside of the component in order to identify pre- and post-conditions of indi-

vidual accessor functions to get hints on the admissible order of actions. Both black-box and glass-box approaches are necessary: Black-box approaches support protocol recovery only for actual, given usages of the component and there may be more allowable usages. On the other hand, in many cases, the assumption that a certain subprogram of the component has been called is (undocumented) part of a precondition. Hence, the actual order of actions established by black-box approaches would be of benefit to glass-box approaches.

12.2.6 Subsystem Detection

Related atomic components themselves may be grouped to lower-level subsystems and lower-level subsystems can be grouped to higher-level subsystems in order to derive a hierarchical decomposition of the system as-built. In particular for large systems, subsystems are an important grouping mechanism for understanding, maintenance, and management purposes.

Atomic component detection is a starting point for subsystem detection. Techniques similar to atomic components detection can be used to detect subsystems. For, example *Same Module* could be extended to *Same Directory, Similarity Clustering* could be used with a similarity metric based on atomic components instead of base entities, *Dominance Analysis* can be used as described in Section 11.1; if atomic components to be hidden are known, *Internal Access* could be extended, and so forth. The main difference to atomic component detection is that a hierarchy of components is to be detected instead of a list of flat components.

Techniques for subsystem detection could easily be integrated into the extended Rigi since the combining operators have already been defined for hierarchical clusters.

12.2.7 Architectural Conformance

Recovering architectural information from source code is not only necessary for system understanding. It is also necessary to validate architectural specifications unless the code is automatically generated from the specification and the generation itself is reliable. A software architect may specify aspects like the structure of the system (atomic components, subsystems, hidden parts etc.), protocols of components, or configurations, e.g., as design patterns. In order to validate these spec-

ifications, it is necessary to recover the architecture as-built and compare it to the specification. Hence, three major aspects need to be addressed by research in architectural conformance:

- Specification:
  - What is to be specified in software architecture?
  - What are suitable methods, notations, and tools for specifying software architecture?
  - What kind of analyses are supported by these methods and notations?
- Recovery:
  - How can we retrieve architectural information that needs to be validated?
- Validation:
  - How can the architecture as-built be checked for conformance to the specification?

12.3 Final Remarks

This thesis has contributed to architecture recovery by evaluating, improving, and combining automatic techniques for component recovery and integrating these techniques in a general framework supporting an interactive and incremental process. The information gathered by the methods and techniques described in this thesis are helpful for program understanding, other reverse engineering and reengineering techniques, as well as software validation. This thesis is a stepping stone toward automatic support for architecture recovery. The need for automatic support will increase as the size and complexity of systems as well as projects increase. However, the summit is not yet reached. As this chapter has discussed, there is still a lot of work to do in component recovery and architecture recovery in general.

Pour soulever un poids si lourd,
Sisyphe, il faudrait ton courage!
Bien qu'on ait du coeur à l'ouvrage,
L'Art est long et le temps est court.

— Charles Baudelaire: Les fleurs du mal.

Appendix A *Entity-Relationship Model for Basic Structural Information*

This chapter summarizes all entities and relationships introduced in the course of this thesis. The entity type hierarchy is shown in Figure 1-1 (abstract types are printed in italics); non-abstract entity types are explained in Table 1-1. The relationship type hierarchy is presented in Figure 1-2; the non-abstract relationships are explicated in Table 1-2. Figure 1-3 is the entity-relationship model consisting of the entities in Table 1-1 and their relationships in Table 1-2.

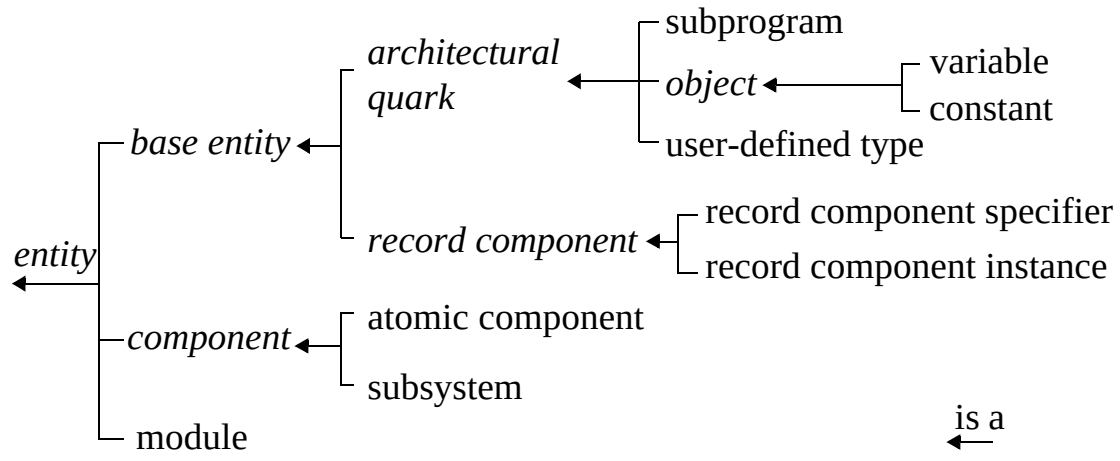


Figure 1-1. **Entity type hierarchy.**

Table 1-1. **Alphabetic list of non-abstract entity types.**

Entity	Meaning	Section
atomic component	named flat set of cohesive base entities	3.2.1
constant	a global object whose value cannot be changed	3.1.1
module	a syntactic unit that is used to group base entities	3.3
record component instance	an actual record component of a record object that is associated with a memory location.	3.1.2
record component specifier	a record component within a type declaration, which defines a part of the memory layout of all instances of this type.	3.1.2
subprogram	a function or procedure of a program	3.1.1
subsystem	hierarchical sets of related elements (architectural quarks, atomic components, and other subsystems)	3.2.2
user-defined type	a type introduced by a programmer	3.1.1
variable	a global object whose value can be changed	3.1.1

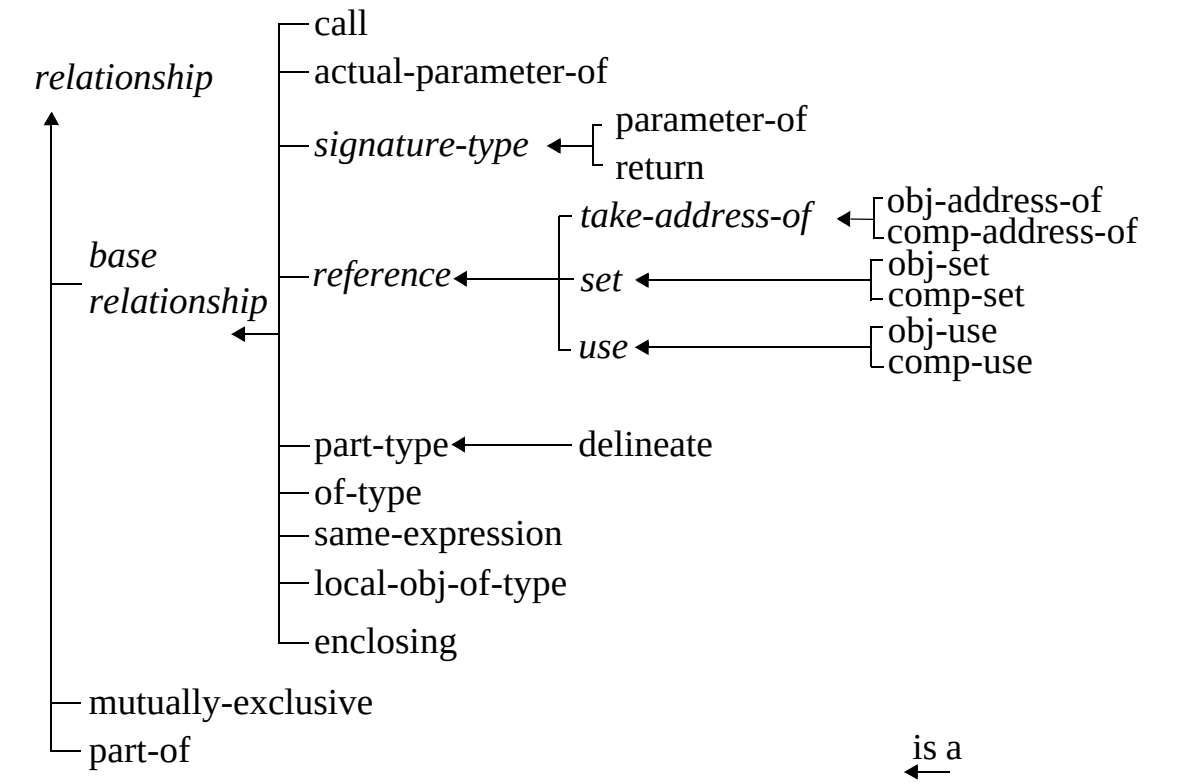


Figure 1-2. **Relationship type hierarchy.**

Table 1-2. **Alphabetic list of non-abstract relationships.**

Relationship	Source S	Target T	Meaning	Section
actual-parameter-of	object	subprogram	S is an *actual parameter* in a call to T	3.1.1
call	subprogram	subprogram	S *calls* T	3.1.1
comp-address-of	subprogram	record component	S *takes the address* of T.	3.1.2.2
comp-set	subprogram	record component	S *sets* the value of T.	3.1.2.2
comp-use	subprogram	record component	S *uses* the value of T.	3.1.2.2
delineate	type	type	T is defined in terms of S as a synonym or as a new type	4.2.3.1
enclosing	record component specifier	type	S is *enclosed* by T	3.1.2.1
enclosing	record component instance	object record component instance	S is *enclosed* by T	3.1.2.1
local-obj-of-type	subprogram	user-defined type	S *has a local variable of type* T	3.1.1
mutually-exclusive	arch. quark component	arch. quark component	S and T must not be in the same component and one must not be a part of the other one	8.2.1
obj-address-of	subprogram	object	S *takes the address* of T.	3.1.2.2
obj-set	subprogram	global variable	S *sets* the value of T.	3.1.2.2
obj-use	subprogram	object	S *uses* the value of T.	3.1.2.2
of-type	object	user-defined type	S is *of type* T	3.1.1
parameter-of	subprogram	user-defined type	S *has* a *formal parameter* of T	3.1.1

Table 1-2. **Alphabetic list of non-abstract relationships.**

Relationship	Source S	Target T	Meaning	Section
part-of	arch. quark. component	component	S is *part of* T	3.2.1
part-type	type	type	S is a *part type* of T	3.1.1
return	subprogram	user-defined type	S *returns* a value of T	3.1.1
same-expression	object	object	S and T occur in the *same expression*	3.1.1

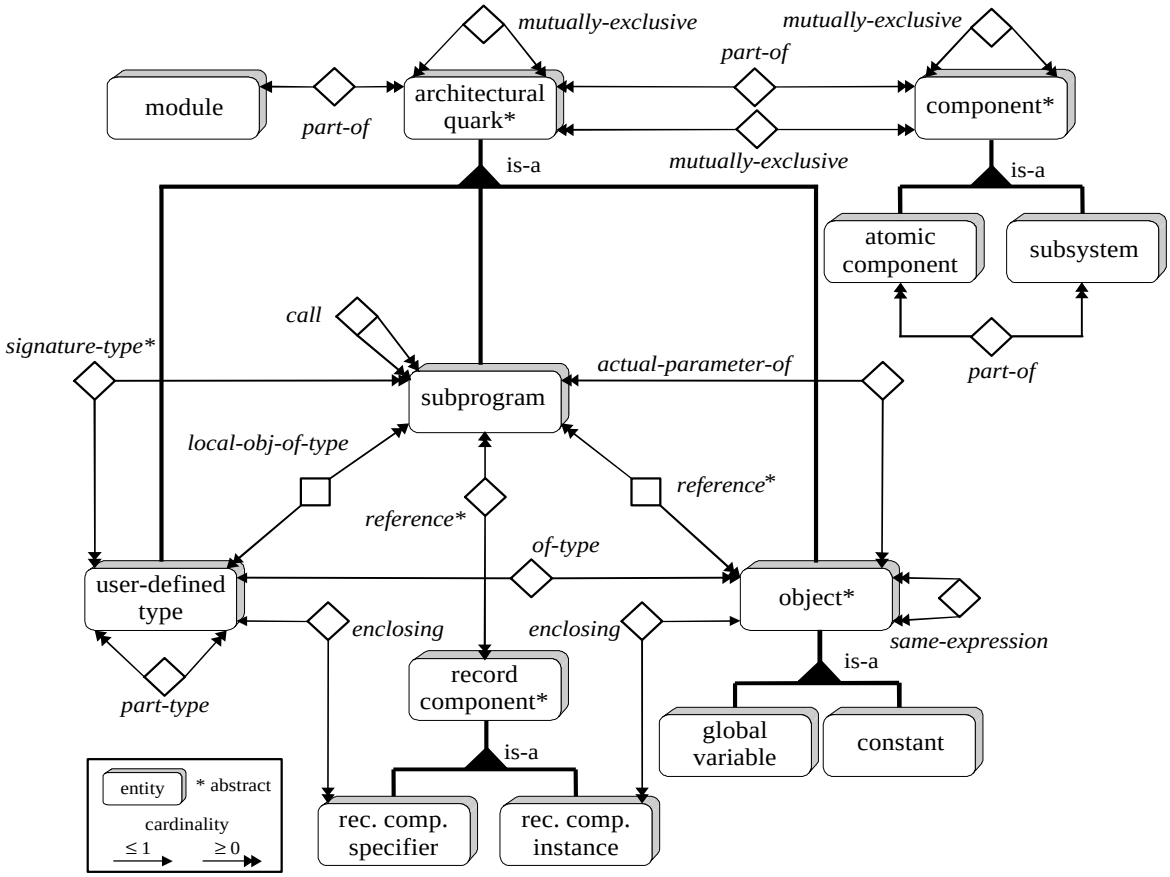


Figure 1-3. **The entities and their relationships.**

Bibliography

Aarts, E. and Korst, J. (1990), *Simulated Annealing and Boltzmann Machines*, Wiley - Interscience Series in Discrete Mathematics and Optimization.

Ada 95 Reference Manual (1995), ANSI/ISO/IEC-8652:1995.

Aho, A.V., Sethi, R., and Ullman, J.D. (1986), *Compilers: Principles, techniques and tools*, Addison Wesley.

American National Standard Programming Language C (1989), ANSI X3.159-1989. American National Standards Institute, New York.

Arnold, K. and Peyton, J. (1992), *A C User's Guide to ANSI-C*, Addison-Wesley.

Baker, B. (1995), 'On Finding Duplication and Near-Duplication in Large Software Systems', *Proceedings of the Working Conference on Reverse Engineering*, pp. 86-95, IEEE Computer Society Press.

Baxter, I., Yahin, A., Moura, L, Sant'Anna, M, and Bier, I. (1998), 'Clone Detection Using Abstract Syntax Trees', *Proc. of the Int. Conference on Software Maintainence*, pp. 368-377, IEEE Computer Society Press.

Bayer, J., Girard, J.-F., Würthner, M, Apel, M., and DeBaud, J.-M. (1999), 'Transitioning Legacy Assets - a Product Line Approach', *Proceedings of the SIGSOFT Foundations of Software Engineering*, Toulouse, pp. 446-463, Association of Computing Machinery.

Belady, L.A. and Evangelisti, C.J. (1982), 'System Partitioning and its Measure', *Journal of Systems and Software*, vol.2, no. 1, February, pp. 23-29.

Berliner, B. (1990), 'CVS II: Parallelizing Software Development', *Proceedings of 1990 Winter USENIX Conference*,Washington, D.C.

Biggerstaff, T. (1989), 'Design Recovery for Maintenance and Reuse', *IEEE Computer*, pp. 36-49, July.

Binkley, A.B. and Schach, S.R. (1998), 'Validation of the Coupling Dependency Metric as a Predictor of Run-Time Failures and Maintenance Measures', *Proc. of the Int. Conference on Software Engineering*, pp. 452-459, April, IEEE Computer Society.

Booch, G., Rumbaugh, J., and Jacobson, I. (1997), *Unified Modeling Language Reference Manual*, Addison-Wesley.

Buss, E., De Mori, R., Gentleman, W. et al. (1994), 'Investigating Reverse Engineering Technologies for the CAS Program Understanding Project', *IBM Systems Journal*, vol. 33, no. 3, pp. 477-500, February.

Canfora, G., Cimitile, A., Munro, M., and Taylor, C.J. (1993), 'Extracting Abstract Data Types from C Programs: A Case Study', *Proceedings of the International Conference on Software Maintenance*, pp. 200-209, IEEE Computer Society Press, September.

Canfora, G., Cimitile, A., and Munro, M. (1996), 'An Improved Algorithm for Identifying Objects in Code', *Journal of Software Practice and Experience*, vol. 26, no. 1, pp. 25–48, January.

Canfora, G., Cimitile, A., De Lucia, A., and Di Lucca, G.A. (1999a), 'A Case Study of Applying an Eclectic Approach to Identify Objects in Code', *Workshop on Program Comprehension*, pp. 136-143, Pittsburgh, IEEE Computer Society Press.

Canfora, G. (1999b), personal communication at the Workshop on Program Comprehension, Pittsburgh, May 4th-6th.

Chikofsky, E.J., Cross II, J. H. (1990), 'Reverse Engineering and Design Recovery: A Taxonomy', pp. 13-17, *IEEE Software*, January.

Choi, S.C., Scacchi, W. (1990), 'Extracting and Restructuring the Design of Large Systems', *IEEE Software*, vol. 7, no. 1, pp. 66-71, January.

Cimitile, A. and Visaggio, G. (1995), 'Software Salvaging and the Call Dominance Tree', *Journal of Systems Software*, 28, pp. 117–127.

DeBaud, J.-M., Rugaber, S. (1995), 'A Software Re-Engineering Method using Domain-Models', *Proc. of the Int. Conference on Software Maintenance*, IEEE Computer Society Press, pp. 204-213, October.

Doval, D., Mancoridis, S., Mitchel, B.S, Chen, Y., and Gansner, E.R. (1999), 'Automatic Clustering of Software Systems using a Genetic Algorithm', http://www.mcs.drexel.edu/~serg/Projects/Bunch/publication/index.html.

Eisenbarth, T., Koschke, R., Plödereder, E., Girard, J.-F., and Würthner, M. (1999), 'Projekt Bauhaus: Interaktive und inkrementelle Wiedergewinnung von SW-Architekturen', *Konferenzband zum Workshop Reengineering*, pp. 17-25, Bad Honnef, Universität Koblenz.

Fanta, R. and Rajlich, V. (1999), 'Restructuring Legacy C Code into C++', *Proc. of the Int. Conference on Software Maintenance*, pp. 77-85, IEEE Computer Society Press.

Fenton, N.; Pfleeger, S. (1997), *Software Metrics: A Rigorous and Practical Approach* Pws Pub.

Fiutem, R., Tonella, P., Antoniol, G., and Merlo, E. (1996), 'A Cliche-based Environment to Support Architectural Reverse Engineering', pp. 319-328, *Proc. of the Int. Conf. on Software Maintenance*.

Fjeldstadt, R.K., and Hamlen, W.T. (1984), 'Application Program Maintenance Study: Report to Our Respondents', Proc. GUIDE 48, IEEE Computer Society Press, April.

Gall, H. and Klösch, R. (1995), 'Finding Objects in Procedural Programs: An Alternative Approach', *Proc. of the Second Working Conference on Reverse Engineering*, pp. 208-216, IEEE Computer Society Press.

Gall, H., Klösch, R., and Weidl, J. (1998), 'Resolving Uncertainties in Object-Oriented Re-Architecturing of Procedural Code', *7th int. Conf. on Information Processing and Management of Uncertainty in Knowledge-Based Systems*, pp. 726-733, July.

Gamma, E., Helm, R, Johson, R., and Vlissides, J. (1994), *Design Patterns*, Addison-Wesley.

Garlan, D., Perry D.E. (1995), 'Introduction to the Special Issue on Software Architecture', *IEEE Transactions on Software Engineering*, Vol. 21, No. 4, pp. 269-274, April.

Garlan, D, Shaw, M. (1993), 'An Introduction to Software Architecture', *Advances in Software Engineeering and Knowledge Engineering*, Volume 1, World Scientific Publishing Company, New Jersey.

Girard, J.F., Koschke, R. (1997a), 'Finding Components in a Hierarchy of Modules: a Step Towards Architectural Understanding', *International Conference on Software Maintenance*, pp. 58-65, IEEE Computer Society Press.

Girard, J.F., Koschke, R., and Schied, G. (1997b), 'Comparison of Abstract Data Type and Abstract State Encapsulation Detection Techniques for Architectural Understanding', *Work-*

ing Conference on Reverse Engineering, pp. 66–75, Amsterdam, The Netherland, October, IEEE Computer Society Press.

Girard, J.F., Koschke, R., and Schied, G. (1997c), 'A Metric-based Approach to Detect Abstract Data Types and Abstract State Encapsulation', *Conf. on Automated Software Engineering*, Lake Tahoe, pp. 82-89, IEEE Computer Society Press.

Girard, J.F., Koschke, R., and Schied, G. (1999), 'A Metric-based Approach to Detect Abstract Data Types and Abstract State Encapsulation', *Journal on Automated Software Engineering*, no. 6, October, pp. 357-386, Kluwer Academic Publishers.

Girard, J.F. and Koschke, R., (2000), 'A Comparison of Abstract Data Type and Objects Recovery Techniques', *Journal Science of Computer Programming*, vol. 6, issue 2-3, March, pp. 149-181, Elsevier Science.

Ghezzi, G., Jazayeri, M., and Madrioli, D. (1991), *Fundamental Software Engineering*, Prentice Hall International.

Godin, R., Missaoui, R., and Aloui, H. (1995), 'Incremental Concept Formation Algorithms Based on Galois (Concept) Lattices', *Computational Intelligence*, 11(2).

Graudejus, H. (1998), *Implementing a Concept Analysis Tool for Identifying Abstract Data Types in C Code*, Diplomarbeit, University of Kaiserslautern, Germany.

Harandi, M. and Ning, J.Q. (1990), 'Knowledge-based Program Analysis', *IEEE Software*, January, pp. 74-81.

Harris, D.R., Reubenstein, H.B., and Yeh, A.S (1995), 'Recognizers for Extracting Architectural Features from Source Code', *Proceedings of the Working Conference on Reverse Engineering*, pp. 227-236, Toronto, IEEE Computer Society Press.

Hopcraft, Ullman (1983), *Data structures and algorithms*, Addison-Wesley.

Hutchens, D.H. and Basili, V.R. (1985), 'System Structure Analysis: Clustering with Data Bindings', *IEEE Transactions on Software Engineering*, vol. SE-11, no. 8, pp. 749-757, August.

Karypsis, G., (Sam) Han, E.-H., and Kumar, V. (1999), 'Chameleon: Hierarchical Clustering Using Dynamic Modeling', *IEEE Computer*, pp. 68-75, August.

Kazman, R. and Carrière, S. J., (1997), 'Playing detective: reconstructing software architecture from available evidence', *Technical Report* CMU/SEI-97-TR-010, ESC-TR-97-010, Software Engineering Institute, Pittsburgh, USA.

Kazman, R., Woods, S., and Carrière, S. (1998), 'Requirements for Integrating Software Architecture and Reengineering Models: CORUM II', *Working Conference on Reverse Engineering*, Hawaii, pp. 154-163, October, IEEE Computer Society Press.

Keller, H., Stolz, H., Ziegler, A., and Bräunl, T. (1995), 'Virtual Mechanics Simulation and Animation of Rigid Body Systems with Aero', *Simulation for Understanding*, vol. 65, no. 1, pp. 74–79, July.

Kernighan, B.W., Ritchie, D.M. (1978), *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ.

Koschke, R., Rugaber, S., Steirwalt, K., and Wills, L. (1997), 'Tupling', Unpublished.

Koschke, R., Girard, J.-F., and Würthner, M., (1998) 'An Intermediate Representation for Integrating Reverse Engineering Analyses', *Working Conference on Reverse Engineering*, Hawaii, October, pp. 256-267, IEEE Computer Society Press.

Krone, M., Snelting, G. (1994), 'On the Inference of Configuration Structures From Source Code', *Proc. of the Int. Conference on Software Engineering*, pp. 49-57, May, IEEE Computer Society Press.

Lehman, M.M., Belady, L. (1985), 'Program Evolution', *Processes of Software Change*, Academic Press, London.

Lienert, G.A. (1973), *Verteilungsfreie Methoden in der Biostatistik*, Verlag Anton Hain, Meisenheim am Glan, Germany.

Lindig, C. and Snelting, G. (1997), 'Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis', *Proc. of the Int. Conference on Software Engineering*, pp. 349-359, Boston.

Liskov, B., Zilles, S. N. (1974), 'Programming with Abstract Data Types'. *SIGPLAN Notice*, vol. 9, no. 4, pp. 50–60, April.

Liu, S.S. and Wilde, N. (1990), 'Identifying Objects in a Conventional Procedural Language: An Example of Data Design Recovery', *Int. Conf. on Software Maintenance*, November, pp. 266-271, IEEE Computer Society Press.

Livadas, P.E. and Johnson, T. (1994), 'A New Approach to Finding Objects in Programs', *Journal of Software Maintenance: Research and Practice*, no. 6, 249-260.

Macro, A. and Buxton, J. (1987), *The Craft of Software Engineering*, Addison-Wesley, Reading, MA.

Mann, H.B., and Whithney, D.R. (1947), 'On a Test of Whether One of Two Random Variables is Stochastically Larger than the Other', *Annuals of Mathematical Statistics*, 18.

Mayrand, J., Coallier, F. and Merlo, E. (1996), 'Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics', *Proc. of the Int. Conference on Software Maintenance*, pp. 244-253, IEEE Computer Society.

McCabe, T. (1998), keynote address at the Working Conference on Reverse Engineering, Hawaii, October.

Müller, H., Uhl, J.S. (1990), 'Composing Subsystem Structures Using (K,2)-Partite Graphs', *Proceedings of the Conference on Software Maintenance*, pp. 12-19, IEEE Computer Society Press.

Müller, H., Wong, K., and Tilley, S. (1992), 'A Reverse Engineering Environment Based on Spatial and Visual Software Interconnection Models', *Proc. of ACM SIGSOFT Symposium on Software Development Environments*, pp. 88-98, December.

Müller, H., Orgun, M., Tilley, S., and Uhl, J. (1993), 'A Reverse Engineering Approach to Subsystem Structure Identification', *Journal of Software Maintenance: Research and Practice*, vol. 5, no. 4, pp. 181-204, December.

Murphy, G., Notkin, D., and Sullivan, K. (1995), 'Software Reflexion Models: Bridging the Gap between Source and High-Level Models', *Proc. of the ACM SIGSOFT Symp. Foundations of Software Engineering*, pp. 18-28.

NCSA (1997), 'NCSA Mosaic Home Page', National Center for Supercomputing Applications, http://www.ncsa.uiuc.edu/SDG/Software/Mosaic.

Nosek, J.T. and Palvia, P. (1990), 'Software Maintenance Management: Changes in the Last Decade', *Journal of Software Maintenance*, 2(3), pp. 157-174, Sept.

Ogando, R.M., Yau, S.S., and Wilde, N. (1994), 'An Object Finder for Program Structure Understanding in Software Maintenance', *Journal of Software Maintenance*, vol. 6, no. 5, pp. 261–83, September-October.

Owen, D.B. (1962), *Handbook of Statistical Tables*, Addison-Wesley.

Parnas, D.L. (1972), 'On the Criteria To Be Used in Decomposing Systems into Modules', *Communications of the ACM*, Vol. 15, No. 12, pp. 1053-1058, December.

Parnas, D.L., Clements, P.C., and Weiss, D.M. (1985), 'The Modular Structure of Complex Systems', *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 3, pp. 408-419, March.

Patel, S., Chu, W., and Baxter, R. (1992), 'A Measure for Composite Module Cohesion', *Proc. of the 14th Int. Conf. on Software Engineering*, pp. 38-48, Melbourne, May, Association of Computing Machinery.

Perry, D.E., Wolf, A.L. (1992), 'Foundations for the Study of Software Architecture', *ACM SIGSOFT*, vol. 17, no. 4, pp. 40–52, October.

Pitman, E.J.G., (1937), 'Significance Test Which May be Applied to Samples From Any Population', *Journal of the Royal Statistics Society* 4.

Plödereder, E. (1997), personal communication, Stuttgart, Germany.

Quilici, A., Woods, S,. and Zhang, Y. (1997), 'New Experiments With a Constraint-Based Approach to Program Plan Matching', *Working Conference on Reverse Engineering*, pp. 114-123, IEEE Computer Society.

Ramey, C. (1994), 'Bash - The GNU shell', *Linux Journal*, issue 3, August, Specialized Systems Consultants.

Rugaber, S., Stirewalt, K., and Wills, L. (1995), 'The Interleaving Problem in Program Understanding', *Proc. of the 2nd Working Conference on Reverse Engineering*, pp. 166-175, Toronto, July, IEEE Computer Society Press.

Sahraoui, H., Melo. W, Lounis, H., and Dumont, F. (1997), 'Applying Concept Formation Methods to Object Identfication in Procedural Code', *Proc. of Conference on Automated Software Engineering*, Nevada, pp. 210-218, November, IEEE Computer Society.

Salton, G. (1968), *Automatic Information Organization and Retrieval*, McGraw-Hill, New York.

Salton, G. (1975), *Dynamic Information and Library Processing*, Prentice-Hall, Englewood Cliffs, N.J., pp. 80.

Schwanke, R. W. (1991), 'An intelligent tool for re-engineering software modularity', *International Conference on Software Engineering*, pp. 83–92, May.

Schwanke, R.W., Hanson, S.J. (1994), 'Using Neural Networks to Modularize Software', *Machine Learning*, 15, pp. 136-168.

Schwanke, R.W. (1998), personal communication by electronic mail, November.

Sebesta, R. (1998) *Concepts of Programming Languages*, Addison-Wesley Publishing Company, Inc.

Shannon, C. E. (1972), *The mathematical theory of communication*. Urbana University of Illinois Press, ISBN 0-252-72548-4.

Siff, M. and Reps, T. (1997), 'Identifying Modules via Concept Analysis', *Proc. Int. Conference on Software Maintenance*, Bari, pp. 170-179, October, IEEE Computer Society.

Siff, M. (1998), *Techniques for System Renovation*, Ph.D. Thesis, University of Wisconsin - Madison.

Snelting, G. (1997), 'Reengineering of Configurations Based on Mathematical Concept Analysis', *ACM Transactions on Software Engineering and Methodology* 5, 2, pp. 146-189, April.

Snelting, G. and Tip, F. (1998), 'Reengineering Class Hierarchies Using Concept Analysis', *Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering*, November, pp. 99-110.

Steinhausen, D., Langer, C. (1977), *Clusteranalyse*, Walter de Gruyter, Berlin, New York.

Tarjan, R. (1972), 'Depth-First Search and Linear Graph Algorithms', *SIAM*, vol. 1, no. 2, pp. 146–160.

Tarjan, R. (1974), 'Finding Dominators in Directed Graphs', *SIAM Journal of Computing*, vol. 3, no. 1, pp. 62-87, March.

Tarjan, R. (1981), 'Fast Algorithms for Solving Path Problems', *Journal of the Association for Computing Machinery*, vol. 28, no. 3, July, pp. 594-614.

Turski, W. (1981), 'Software Stability', *Proc. 6th ACM Conf. on Systems Architecture*, London.

Tversky, A. (1977), 'Features of Similarity', *Psychological Review*, vol. 84, no. 4, July.

Tzerpos, V. and Holt, R. (1997). 'The Orphan Adoption Problem in Architecture Maintenance', *Proceedings of the Fourth Working Conference on Reverse Engineering*, pp. 76-82, Amsterdam, October, IEEE Computer Society Press.

Valasareddi, R.R. and Carver, D.L. (1998), 'A Graph-based Object Identification Process for Procedural Programs', *Proc. of the Fifth Working Conference on Reverse Engineering*, Honolulu, pp. 50-58, October, IEEE Computer Society Press.

Van Deursen, A., Woods, S., and Quilici, A. (1997), 'Program Plan Recognition for Year 2000 Tools', *Proc. of the Fourth Working Conference on Reverse Engineering*, pp. 124-133, IEEE Computer Society Press.

Weidl, J. and Gall, H. (1998), 'Binding Object Models to Source Code: An Approach to Object-Oriented Re-Architecturing', *Proc. of the 22nd Computer Software and Applications Conference*, IEEE Computer Society Press, August.

Weiser, M. (1984), 'Program Slicing', *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352-357, July.

Winer, B.J., Brown, D.R., and Michels, K.M. (1991), *Statistical Principles in Experimental Design*, 3rd edition, McGraw-Hill Series in Psychology.

Wills, L.M. (1992), *Automated Program Recognition by Graph Parsing*, Ph.D. Disseration, MIT, Cambridge, MA.

Wirth, N. (1985), *Programmieren in Modula-2*, Teubner, third edition.

Yeh, A.S., Harris, D., and Reubenstein, H. (1995), 'Recovering Abstract Data Types and Object Instances From a Conventional Procedural Language', *Second Working Conference on Reverse Engineering*, pp. 227–236, July. IEEE Computer Society Press.

Yeh, A.S., Harris, D.R., and Chase, M.P. (1997), 'Manipulating Recovered Software Architecture Views', *Proc. of the Int. Conference on Software Engineering*, pp. 184-194, Association of Computing Machinery.

Yourdon, E. and Constantine, L.L. (1979), *Structured Design*, Prentice Hall, Englewood Cliffs, NJ.

Index