NOTE TO USERS

This reproduction is the best copy available

UMI
SCALABLE PROGRAM RECOGNITION FOR KNOWLEDGE-BASED REVERSE ENGINEERING

A Thesis Submitted to the College of Graduate Studies and Research in Partial Fulfillment of the Requirements For the Degree of Doctor of Philosophy in the Department of Computer Science University of Saskatchewan Saskatoon, Saskatchewan

by

Srinivas Palthepu

August 1998

© Copyright Srinivas Palthepu, 1998. All rights reserved.
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-37907-8
UNIVERSITY OF SASKATCHEWAN
College of Graduate Studies & Research

SUMMARY OF DISSERTATION
Submitted in partial fulfillment of the requirements for the

DEGREE OF DOCTOR OF PHILOSOPHY
By

Srinivas Palthepu
Department of Computer Science
University of Saskatchewan
Fall 1998

Examining Committee:

Dr. L.G. Watson                Dean's Designate, Chair
College of Graduate Studies and Research

Dr. Gordon McCalla            Co-Supervisor, Department of Computer Science

Dr. Jim Greer                 Co-Supervisor, Department of Computer Science

Dr. J. Paul Tremblay          Department of Computer Science

Dr. Carl McCrosky             Department of Computer Science

Dr. Robert Gander             Department of Electrical Engineering

External Examiner:

Dr. W. Lewis Johnson, Director
University of Southern California
Information Science Institute
4676 Admiralty Way
Marina del Rey, CA 90292-6695
U.S.A.
Scalable Program Recognition for Knowledge-Based Reverse Engineering

Reverse engineering is the process of reconstructing high-level design information from program code. Reverse engineering and re-engineering have become pressing needs for many organizations as existing legacy code no longer meets their needs. Program understanding plays an important role in any reverse engineering activity since the user (typically a maintainer) needs to reconstruct the cognitive conceptualization of the programmer to be able to understand and make any changes to the existing system. Attempts in Artificial Intelligence (AI) to automatically "understand" a program in terms of a set of pre-defined plans have encountered many problems of scalability and brittleness when applied to real world problems such as reverse engineering.

Software engineering is different from traditional engineering disciplines. A software system is a cognitive artifact and is intimately tied to the conceptualizations of the human programmer. Hence techniques such as formal analysis, originally developed in traditional engineering disciplines do not adapt well to software engineering. It is necessary to acknowledge the fact that software engineering is primarily a cognitive process to develop more effective tools and techniques that address the problems of software engineering.

In this thesis, we present a human-centered software reverse engineering environment using a scalable, robust program recognition technique based on granularity. The granularity-based formalism as used in SCENT is extended by adding additional types of constraints and context modifiers to make the program recognition efficient. Granularity-based program recognition
overcomes many of limitations of the traditional approaches by allowing the human expert using
the system to be always "in-the-loop" of problem solving. The agenda-based recognition
method presented here is flexible to be able to use various sources of information to guide the
system. We implemented a prototype system called KARE (Knowledge-based Assistant for
Reverse Engineering) using this approach. We conducted three different experiments with
KARE by using it on two different real-world software systems. The results from our experiments
show evidence that the powerful granularity mechanisms such as the context modifiers along
with appropriate human interventions help make KARE scalable. The thesis describes KARE
and our experiences of using KARE on real-world software systems along with some
experimental evidence which demonstrates the usefulness of the techniques.

BIOGRAPHICAL

September, 1962
August, 1984
January, 1986
December, 1987
January 1988 to December 1993
Born in Boenpally, AP, India
B.E. in Civil Engineering
Osmania University, Hyderabad, India.
M.E. in Structural Engineering
Indian Institute of Science, Bangalore, India.
M.Tech. in Computer Science
Jawaharlal Nehru University, New Delhi, India.
Staff Scientist
National Center for Software Technology, Bombay (Mumbai), India.

HONOURS

University of Saskatchewan Graduate Scholarship (1994).
Jawaharlal Nehru University Graduate Fellowship (1986).
Permission to Use

In presenting this thesis in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
University of Saskatchewan
Saskatoon, Saskatchewan, Canada
S7N 5A9
Abstract

Reverse engineering is the process of reconstructing high-level design information from lower-level information such as program code. Reverse engineering and re-engineering (a reverse engineering step followed by a forward engineering step) have become pressing needs for many organisations as existing legacy code no longer meets their needs. Program understanding plays an important role in any reverse engineering activity since the user (typically a maintainer) needs to reconstruct the cognitive conceptualisation of the programmer to be able to understand and make any changes to the existing system. There have been some attempts in artificial intelligence (AI) to automatically “understand” a program in terms of a set of pre-defined plans. Most of these attempts have encountered many problems of scalability and brittleness when applied to real world problems such as reverse engineering.

Software engineering is different from traditional engineering disciplines. A software system is a cognitive artifact and is intimately tied to the conceptualisations of the human programmer. Hence techniques such as formal analysis, originally developed in traditional engineering disciplines do not adapt well to software engineering. Attempts to remedy the problems faced by software engineering, such as the “software crisis”, yielded only limited results. It is necessary to acknowledge the fact that software engineering is primarily a cognitive process. We need to develop tools and techniques that address the problems of software engineering from this perspective.

In this thesis, we present a human-centered software reverse engineering environment using a scalable, robust program recognition technique based on granularity. The granularity-based formalism as used in SCENT is extended by adding additional types of constraints and context modifiers to make the program recognition efficient. Granularity-based program recognition overcomes many limitations of the traditional approaches by allowing the human expert using the system to be always “in-the-loop” of problem solving. The agenda-based recognition method presented here is flexible to be able to use various sources of information to guide the system. A prototype system called KARE (Knowledge-based Assistant for Reverse Engineering) is implemented using this approach. Three different experiments were conducted with KARE by using it on two different real-world software systems. The results from the experiments show evidence that the powerful granularity mechanisms such as the context modifiers along with appropriate human interventions help make KARE scalable. The thesis describes KARE and our experiences of using KARE on real-world software systems along with some experimental evidence which demonstrates the usefulness of the approach.
Acknowledgements

My sincere thanks to my supervisors Professor Gordon McCalla and Professor Jim Greer for helping me and providing guidance during the entire period of my thesis. Working with Gord and Jim has been not only a great learning experience, but also a pleasurable one.

I would like to thank Professor W. Lewis Johnson for serving as the external examiner on my thesis committee and providing valuable and insightful suggestions which helped me to improve my thesis.

I want to thank Professor J. Paul Tremblay, Professor Carl McCrosky and Professor Bob Gander for serving on my committee and providing valuable feedback and suggestions on my thesis.

I would like to thank the staff members of the department of computer science at the University for providing necessary support that enable me to carry out my work.

Thanks are also due to many members of ARIES Laboratory who directly or indirectly helped me in my work.

Life would have been difficult without friends like Vive Kumar who has been always there when I need him for whatever the reason. I would like to express my special thanks to him. I also want to thank him for helping me in tying some loose ends on my thesis draft at the final stages when I was away from the campus.

I am fortunate to have a wonderful wife, Lakshmi whose love and support helped me to re-charge my mental energy when I am down.

Finally, I would like to thank the University of Saskatchewan for providing me the financial support in terms of the graduate scholarship, and my supervisors Gord and Jim for providing me additional financial support.
Dedication

To my mother, Yashoda, who always has faith in me, encourages me to do my best, and takes pride in what I do.
# Table Of Contents

Permission to Use ................................................................. i

Abstract ................................................................................. ii

Acknowledgements ................................................................. iii

Dedication ................................................................................. iv

Table Of Contents ..................................................................... v

List of Tables ........................................................................... x

List of Figures .......................................................................... xi

1 Introduction ............................................................................. 1

1.1 Motivation and Broad Goals of the Thesis ................................. 2

1.2 Specific Goal and Contributions of the Thesis ............................ 5

1.3 Guide to the Rest of the Thesis ................................................. 6

2 The Reverse Engineering Problem ............................................ 7

2.1 What is Reverse Engineering? .................................................. 7

2.1.1 Terms and Their Meaning ................................................... 8

2.1.2 Scope of Reverse Engineering ............................................. 10

2.1.3 Why is Reverse Engineering Hard? ................................... 12

2.2 Approaches to Reverse Engineering ........................................ 12

2.2.1 Language Parsing and Compiler Technology ....................... 15

2.2.2 Program Transformational Approach ................................. 17
2.2.3 Program Organisation/Visualisation ................................... 19
2.2.4 Other Code Analysis Tools ........................................... 23

2.3 Knowledge-Based Approaches for Software Analysis ........................................... 24
2.3.1 Program Recognition Approach ......................................... 25
   2.3.1.1 Intention-Based Program Analysis .................................... 27
   2.3.1.2 Program Recognition as Graph Parsing .............................. 28
   2.3.1.3 Other Approaches ..................................................... 29
2.3.2 Other Knowledge-Based Approaches ......................................... 31
2.3.3 Problems in Program Recognition for Reverse Engineering .... 32

2.4 Towards a Human-Centered Approach to Knowledge-Based Systems ........................................... 34
2.4.1 Human-Computer Symbiosis ............................................. 35

2.5 Summary ................................................................. 37

3 Granularity-Based Representation and Reasoning ........................................... 38
3.1 Roles of the Human User .................................................. 40
3.2 What is Granularity? ........................................................ 40
3.3 Granularity-Based Representation .......................................... 41
3.4 Granularity-Based Recognition in SCENT .................................. 44

4 The KARE Approach to Granularity-Based Cliché Recognition ........................................... 47
4.1 Granularity-based Program Recognition for Reverse Engineering ........ 49
   4.1.1 Distributed Knowledge: Contexts, Constraints, and Controls ........ 50
      4.1.1.1 Contexts and Context Modifier Operators ......................... 50
      4.1.1.2 Constraints ......................................................... 54
      4.1.1.3 Controls ........................................................... 58
   4.1.2 Sharing the Knowledge: Attributes and Attribute Translations ........ 60
      4.1.2.1 Attributes ........................................................ 60
      4.1.2.2 Attribute Translation Rules ...................................... 62
4.1.2.3  Attribute Constraints ........................................ 67
4.1.3  Flexible Control: Agenda-Based Recognition .................. 69
  4.1.3.1  What Is an Agenda? ........................................... 70
  4.1.3.2  The Recognition Algorithm ................................... 71
4.2  KARE: The Reverse Engineering Environment ...................... 77
  4.2.1  KARE Interface .................................................. 77
  4.2.2  Creating Granularity Hierarchies in AROMA .................. 79
  4.2.3  REFINE and KARE ............................................... 82
  4.2.4  A Recognition Example .......................................... 83
4.3  How KARE Can be Used ............................................ 87

5  Using KARE .................................................................. 90
  5.1  Experimental Setup .................................................. 90
    5.1.1  Target Software System ....................................... 90
    5.1.2  Knowledge-Base ................................................. 92
    5.1.3  Experimental Methodology ..................................... 92
    5.1.4  The Human-in-the-Loop ....................................... 93
    5.1.5  Controlling the Experiments ................................... 94
      5.1.5.1  Human Intervention ....................................... 94
      5.1.5.2  Context Modifiers ......................................... 95
      5.1.5.3  Code Size .................................................. 96
    5.1.6  Cliché Engineering for the Experiments ..................... 97
  5.2  The Experiments .................................................... 97
    5.2.1  Experiment 1: Scalability and Human-in-the-Loop .......... 98
      5.2.1.1  No Intervention ........................................... 100
      5.2.1.2  File Scoping ............................................... 100
      5.2.1.3  Thresholding .............................................. 101
      5.2.1.4  Reordering ................................................ 104
      5.2.1.5  All Interventions ........................................ 107
      5.2.1.6  General Observations and Implications ................ 107
5.2.2 Experiment 2: Transferability .......................... 109
5.2.3 Experiment 3: Versatility and Generality ............... 111
5.2.4 Instance Combinations and Computation Time .......... 113

5.3 Typical Situations for Using KARE ....................... 116
5.4 User's Time vs System's Time ............................ 116
5.5 Comparison With Other Approaches ....................... 119
  5.5.1 KARE vs. Johnson's PROUST .......................... 119
  5.5.2 KARE vs. Wills' Recognizer ......................... 120
  5.5.3 KARE vs. SCENT .................................... 122
  5.5.4 KARE vs DECODE .................................. 122
  5.5.5 KARE and Constraint Satisfaction .................... 123
  5.5.6 KARE and Syntactic Methods ......................... 124
  5.5.7 KARE and Simple Searching Tools .................... 125

5.6 Concluding Remarks .................................... 126

6 Conclusions and Future Directions ........................ 127
  6.1 Contributions ........................................... 127
    6.1.1 Contributions to Software Engineering ............... 127
      6.1.1.1 Robustness .................................. 128
      6.1.1.2 Human-In-The-Loop ............................ 128
      6.1.1.3 Scalability .................................. 129
      6.1.1.4 Supporting Human Activity .................... 132
    6.1.2 Contributions to AI and Granularity ................ 133
      6.1.2.1 Practical AI Application Development ........... 133
      6.1.2.2 New Techniques for Program Recognition .......... 133
      6.1.2.3 Scalability of AI Techniques ................... 134
      6.1.2.4 Old Issues Re-cast in New Setting ............... 134
  6.2 Future Directions ...................................... 135
    6.2.1 Extensions to KARE ................................ 135
    6.2.2 Program Recognition Based Reverse Engineering Environment 138
List of Tables

4.1 Source code operators ........................................ 57
4.2 Status indicators of a granularity object. ............... 72
4.3 Overall recognition status of a granularity object. ....... 73

5.1 Effect of various human interventions for using KARE on the linked list cliché. ........................................ 99
5.2 Effect of human intervention and CMs for the searching cliché .... 109
5.3 Date and time cliché: All interventions ..................... 112
5.4 Computation time for various instance combination. ....... 114
List of Figures

2.1 Re-engineering triangle ........................................ 9

3.1 A granularity representation for *Loops* in programming. ........ 42
3.2 A granularity representation for *Linked List* in programming. .... 43

4.1 An architecture of the KARE system. ............................ 48
4.2 An example of occurrence constraints. ............................. 55
4.3 Ordering constraints. ........................................... 56
4.4 Object attributes and attributes translation rules. ................ 61
4.5 Attributes unification algorithm. ................................ 65
4.6 An example of Attribute Value Modifier. .......................... 66
4.7 Attribute constraints: an example. ................................ 67
4.8 Attribute constraints: another example. ........................... 68
4.9 Recognition agenda. ............................................ 69
4.10 An example of an agenda item. ................................... 71
4.11 A module interaction diagram showing caller-callee relationships among.
    major functions in the KARE recognition algorithm (actual control
    flow is not shown). ............................................... 74
4.12 Top-level agenda-based recognition algorithm. ..................... 74
4.13 Algorithm for recognising an object. .............................. 75
4.14 Algorithms for initialising and unfolding agenda items. ............ 76
4.15 A snapshot of the interface to KARE environment. ................. 78
4.16 AROMA hierarchy engineering environment. ........................ 80
4.17 KARE and REFINE relationship with each other. .................. 82
4.18 A part of the C source code in Mosaic. ........................... 85
4.19 An instance of cliché HASH-TABLE underlying the code implement-
ing user history. ..................................................... 86

5.1 Effect of file scoping with CM. ............................... 101
5.2 Effect of file scoping without CM. ......................... 102
5.3 Effect of thresholding with CM ............................. 103
5.4 Effect of thresholding without CM .......................... 103
5.5 Effect of reordering with CM ............................... 105
5.6 Effect of reordering without CM ............................ 106
5.7 Effect of all interventions on LINKED-LIST cliché ....... 108
5.8 Effect of human intervention and CMs for the Searching Cliché. .... 110
5.9 Instance combinations vs time. ............................. 114

6.1 Scalability and the role of the human user in KARE ............. 130

B.1 AST representation of some of the declarations in Mosaic .... 160
B.2 AST representation of some of the declarations in Mosaic (contd.) . 161
B.3 AST representation of some of the functions in Mosaic ........... 162
B.4 AST representation of some of the functions in Mosaic (contd.) .... 163
B.5 AST representation some of the functions in Mosaic (contd.) ....... 164
B.6 AST representation some of the functions in Mosaic (contd.) ....... 165
B.7 AST representation some of the functions in Mosaic (contd.) ....... 166
Chapter 1

Introduction

Software reverse engineering is the process of extracting high-level design information from program code.\textsuperscript{1} Reverse engineering helps in understanding a software system by mapping parts of the code to the functionality of the system in its domain. It brings out the complex relationship between goals and objectives of the system and their implementation in the code. Program understanding is an essential part of software maintenance. Before attempting to modify a software system, it is necessary to understand the relationships between existing code elements and the intent of the programmer in terms of the design objectives. Thus, reverse engineering of a program requires understanding the existing program by mapping its code to its implementation strategies, and its implementation strategies to its behaviour and intended objectives. This is essentially the reverse engineering process.

Automated program understanding attempt to make a computer system able to understand a given program similar to the way a human would understand it. The computer can exhibit this understanding by various means, e.g., by answering questions about the program. This effort is ambitious in its goals. Program understanding has been studied in the field of artificial intelligence (AI) for many years resulting in development of various techniques. These techniques generally use the knowledge of typical strategies programmers employ in implementing a software sys-

\textsuperscript{1}Although software reverse engineering can also be done from any low-level information to any high-level information (for example, from design to specification), it is usually done from the source code to design. One of the reasons for this is that the code is the most reliable source of information about the current state of the system.
tem. This is known as *knowledge-based program recognition*. Here a given program is recognised as a combination of a predefined set of programming strategies.

Most of the efforts in reverse engineering are limited for two fundamental reasons:

- Software reverse engineering in particular, and software engineering in general, have tended to ignore the cognitive nature of a software system. It is a fact that any human endeavour has some element of cognitive processes, but the software being a non-physical artifact, that is flexible, fluid and constantly changing, needs to be interpreted in conceptual terms for a human to be able to understand it and maintain it. Knowledge-based approaches for program understanding potentially can help in providing such interpretations of software systems.

- Technology has been *limited* and *brittle* when it comes to cognitive domains. It has been limited in the sense that most of the AI systems work well in a very limited domain on a small scale and would run into various unforeseen problems when used in a real-world setting on large input (this is also called the problem of scalability). It has been brittle in the sense that any unexpected deviation of the real-world input data can cause an AI system to fail in a sudden and disastrous fashion (this is also called the problem of brittleness). AI techniques for program understanding similarly suffer from the same problems of brittleness and scalability.

In this thesis, we present a knowledge-based program recognition method that is scalable to be able to handle large real-world systems and robust (i.e., less brittle) to be able to work in situations where there is incomplete knowledge.

### 1.1 Motivation and Broad Goals of the Thesis

**Artificial Intelligence Needs a Human in the Loop**

Artificial Intelligence (AI) has been a fertile field of research and development. Research efforts in various sub-disciplines of AI yielded many techniques that address
specific problems of AI. Unfortunately, many of these techniques prove difficult to apply in real-world applications. That is why there are relatively few AI applications deployed and working in real-world settings, when compared to the body of research in AI that generated these techniques.

There are many reasons for this state of affairs, some of them are listed below:

- **Toy domains:** Many of the AI techniques work in small domains in which they are developed and they do not scale up to work in large real-world domains.

- **Isolated techniques:** Many of the AI techniques work well in solving a problem in isolation, but any real application requires many techniques working together to handle complex real-world problems.

- **Full automation:** Many of the AI techniques try to solve a problem automatically, and ignore the human expert using the system.

- **Usability:** Many systems are developed in isolation from the place where the system is supposed to be deployed. The user is frequently overlooked in the process of designing systems.

When it comes to cognitive domains where human conceptualisations play an important role in problem solving, and where available techniques are not sufficient to solve a problem, the human expert using the system should be given a complementary role that facilitates him or her to cooperatively solve the problem together with the system.

*The first goal of the thesis is to show that placing the human expert using an AI system in the centre of the problem-solving process makes it possible for an AI-based system to be successful in the real world. This requires designing tools and techniques that will allow the human user to coexist symbiotically with the system, where system and user complement each other’s abilities to solve a problem.*
Software Engineering Needs Cognitively Oriented Methods

The field of software engineering is unique because of the very nature of the artifact viz., the software system, dealt with in the field. People in the field of software engineering try to build cognitive artifacts in contrast to traditional engineering disciplines such as civil engineering or mechanical engineering, which focus on building physical artifacts. A software system as an artifact is different from a physical artifact such as a machine or a bridge. Engineering disciplines that try to build physical artifacts have developed an array of methodologies for various activities such as analysis, design, and maintenance. There are some well-developed formal techniques that help analyse physical artifacts. There are some representational schemes for communicating information about a physical artifact. For example, most of the information about a bridge designed by a civil engineer can be communicated by means of blueprints of design drawings. The representation has all the necessary information about the physical artifact. In short, a physical artifact can be understood by simply analysing its representation, without much help from outside.

In contrast, a cognitive artifact such as a computer program is a conceptual entity in the minds of the people who developed the programs. The fact that it can be executed on a machine, or the fact that it can be printed on a physical media does not make it a physical artifact. A program listing, or any other physical representation does not communicate all the information about a software system. The practice of placing comments in a program listing is one of the methods people have invented to bridge this "conceptual gap" between the physical representation and the conceptual understanding of the programmer. A program encodes enough information for a computer to be able to carry out the instructions. A program cannot readily communicate the conceptual intent of the programmer to other human beings for the simple reason that it does not have all the information. Most of the conceptual information is lost while mapping design to code. As a result, program code cannot be understood in isolation. Additional information such as comments
and external documentation may help to some extent but they are incomplete and incorrect more often than not.

Most of the formal tools and techniques developed for analysing a software system are useful for obtaining information peripheral to the software system. They do not capture its essence: the cognitive conceptualisation of the programmers and designers. Hence, any techniques for designing and analysing a software system have to take into account the fact that software is different from other engineering products and try to develop cognitively-oriented methodologies for this unique discipline.

The second goal of the thesis is to show that by deploying cognitively oriented methods such as knowledge-based program recognition into the realm of software engineering, more useful and successful tools can be developed.

1.2 Specific Goal and Contributions of the Thesis

This thesis attempts to achieve the goals listed above in the context of a specific problem, the reverse engineering problem, where a software maintainer interprets the program code of a software system and tries to reconstruct its design. In particular, we would like to demonstrate the ability of granularity-based techniques for reverse engineering to meet these goals. This thesis attempts to demonstrate that granularity is well suited for real-world applications, and directly addresses the problems of brittleness and scalability. Granularity provides a robust and flexible means of representing and reasoning about software systems, while at the same time taking advantage of the human expert using the system. This is made possible by placing the human in the loop of the problem-solving process. We introduce a granularity-based program recognition tool to demonstrate these abilities and apply it to reverse engineering of real-world software systems.

This thesis makes many contributions to the field of software engineering as well as to the field of Artificial Intelligence. To the field of software engineering, the thesis offers a new program cliché recognition tool for reverse engineering that is scalable and can work on large real-world software systems. It also offers a perspective of
developing software engineering tools that takes into account the unique cognitive
nature of software artifact. To the field of AI, the thesis offers a way of making
AI techniques robust and practical by placing human user in the loop of problem
solving. It also presents a robust program recognition technique that is flexible to
make use of information from various sources.

1.3 Guide to the Rest of the Thesis

This thesis is organised as follows. Chapter 2 discusses the reverse engineering
problem in detail. It first introduces the problem and describes why it is hard.
The chapter then ends with a description of some of the traditional approaches for
reverse engineering to give an idea of previous attempts to solve the problem.

Chapter 3 gives a detailed background of granularity-based representation and
how such a representation can be used in program recognition.

Chapter 4 presents the extended granularity formalism as used in our system,
KARE (Knowledge-based Assistant for Reverse Engineering). It gives a detailed
description of the approach to tackle the problem of program recognition in large
real-word software systems. It then gives a detailed account of the methodology
including representations and algorithms with examples.

Chapter 5 describes some experiences in using KARE on real-world software
systems. It also gives some results that show evidence of KARE's ability to work
on large reverse engineering problems.

Chapter 6 presents some of the research contributions of the work. The chapter
also discusses further research directions and some open questions.
Chapter 2

The Reverse Engineering Problem

_The answers are in the source code_

−Mark Weiser, IEEE Computer, 1987 [85].

With the increase in the number and the size of large legacy software systems\(^1\), many companies are facing the problem of maintaining these systems. Also, with the rapidly changing present-day business environment, the requirements of software systems used by companies and businesses are also changing constantly. Since it is prohibitively expensive to develop new systems each time a new set of requirements comes along, it has become increasingly necessary to re-engineer old systems to meet new specifications. But re-engineering requires understanding of the existing system: solving the problem of extracting high-level descriptions of the system is a prerequisite for any re-engineering process. In addition, software maintenance also requires some amount of program understanding and reverse engineering. The problem is compounded, since, for the most part, the people who do the re-engineering and maintenance are different from the people who developed the system in the first place.

2.1 What is Reverse Engineering?

_Software Reverse engineering_ is defined as the process of reconstructing high-level

---

\(^1\)Legacy software systems are those that are rather old (15-25 years) and often in poor condition because of prolonged quick-fix types of maintenance.
design information from program code [16]. Reverse engineering in traditional engineering disciplines is an extraneous activity as the artifacts of these disciplines are not fluid and are not modified without explicit changes to their designs. But in the field of software engineering it is an integral part of software maintenance. A maintainer needs to understand the software system by mapping parts of the code to the goals of the system and the domain functions the system is performing. This is the first step for any kind of maintenance activity ranging from simple bug-fixes to large scale system enhancements. In addition, the reverse engineering process is used to extract reusable software components, to recover business rules out of a software system, and to generate documentation from the code.

2.1.1 Terms and Their Meaning

Chikofsky and Cross [16] give a summary of different concepts and terms used in reverse engineering and design recovery which are briefly described here. The term forward engineering is the traditional engineering activity where the system is synthesised from requirements. The term reverse engineering, on the other hand, refers to the process that goes in the opposite direction, i.e., from the software system towards the specification. The term re-engineering refers to a process that is a combination of reverse engineering and forward engineering. The first step in the process of re-engineering is reverse engineering, i.e., to understand the existing system. The second step in re-engineering is forward engineering from new specifications and the knowledge of the old system obtained from reverse engineering to obtain a new system. A graphical representation of these terms is given in Figure 2.1.

Figure 2.1 shows three levels demarcated as specification, design and implementation (i.e., code) that correspond to respective phases in the software development life cycle. These divisions are well accepted in software engineering [64]. But as described in Section 2.1.3, these divisions are arbitrary when we try to view them from the standpoint of the human software engineer involved in software development.

Chikofsky and Cross [16] distinguish two types of abstractions in software sys-
Figure 2.1: Re-engineering triangle

tems. Abstractions from one stage of the system life cycle to another stage (e.g., specification to design), and abstractions within each stage (a detailed description of a design document compared to an abstract description of the same). Reverse engineering generally involves extracting design artifacts and building abstractions that are less implementation dependent. There are two main classes of reverse engineering activities, called Re-documentation and design recovery. Re-documentation basically provides alternative but semantically equivalent views of the system at the same abstraction level. Examples of such systems are pretty printers, formatters, indentation tools that prints out the source code in a form that brings out the structure of the program more clearly, and diagram generators that represent code in terms of flow graphs. Design recovery, on the other hand, tries to use different sources of knowledge to recover the higher-level design information from the code. These sources of information could come from domain knowledge, specification documents, design documents, comments in the code, and/or other external information. This kind of reverse engineering, most of the time, should be able to deal with incomplete and insufficient information and should be able to recognise the design patterns approximately. Restructuring is another term used in this field.
which involves transformation of the system into another representation at the same level of abstraction. Unlike in re-documentation, here the changes to the artifact are substantial but it does not change the behaviour or meaning of the system. Examples of such subsystems are restructuring tools which transform programs with GOTO statements to programs without GOTOs.

It should be noted that all these terms and corresponding activities do not presume that these activities are done automatically by a computer. They might well be done by humans, and in fact many of them are now done by people. These terms are equally valid in either case.

Byrne gives a conceptual basis from which the problem of software re-engineering can be understood more clearly [13]. He presents various levels of abstractions of software systems in which the static code forms the lowest level of abstraction and specification and goals form the highest level of abstraction. Re-engineering can be done at any of these levels, and types of activities at each level can be different. He also describes some of the properties of these activities such as separation of concerns at various abstraction levels, and information inclusion where information included at any level of abstraction influences only levels below it.

2.1.2 Scope of Reverse Engineering

There are various types of activities that come under the rubric of reverse engineering and re-engineering. In a broad sense, any type of software maintenance requires some amount of program understanding and reverse engineering. Even before one can attempt a simple type of maintenance, like fixing a bug, one needs to understand the relevant part of the code and how it is performing a particular task in the domain. Some simple bugs can be fixed locally without understanding the rest of the code, but most of the time various parts of the code of a software system are inter-linked in a complex way and one should make sure that the changes being made will not affect the functionality of the rest of the system.

Reverse engineering is done in many situations. What is achieved as a result of
any reverse-engineering activity varies depending on what is intended to be done in the next step with the information extracted from reverse engineering. Reverse engineering involves some or all of the following activities:

- **Program comprehension** generally means understanding a program by a person by figuring out what it does and how. Hence program comprehension is program understanding by a human.

- **Re-documentation and/or document generation** involves generating documentation of a software system from the source code so that others can understand its functionality. The documentation may be either created for the first time, or derived from the existing documentation corresponding to some older version of the software system which underwent changes over time. In the latter case, the existing documentation serves as the starting point.

- **Design recovery** is the process of recovering high-level abstractions that were coded into a program in the process of implementation. It should be noted that there might not have been any design abstractions created in the first place and the implementation might have evolved without any explicit changes to the software design. The process of design recovery tries to extract the design abstractions, even in situations where there were no explicit designs that were responsible for the implemented system.

- **Recovering reusable components**: where pieces of software are extracted that can be isolated from the system and re-used in some other place in the same software system or in some other application: that is altogether different.

- **Recovering business rules.** A software system embodies various kinds of business practices and rules of the organisation implicitly or explicitly. It is useful to recover such rules from the software so that they can be used either for implementing a new software system or simply as a documentation aid for the organisation.
The above list is not an exhaustive list of activities of software reverse engineering. The list is intended to provide a glimpse of the typical objectives people generally have when they attempt reverse engineering and re-engineering activities.

Depending upon a specific situation, reverse engineering may include some or all of the above activities. Whatever may be the objective of the reverse engineering effort, it consists of understanding an existing software system. It starts with existing information such as original design documents, user manuals, and the code. Although various sources of information are available, the actual code is the ultimate description of the current state of the system. Hence, the crux of the reverse engineering process is the problem of program understanding.

2.1.3 Why is Reverse Engineering Hard?

Software engineering is a human activity. A program is intimately tied to the cognitive conceptualisation of the programmer. If we remove the programmer from the system, the static code cannot sufficiently communicate the intent of the programmer to other human beings. While good documentation can fill this communication role, sufficient documentation may not exist or it may not have been updated as the code evolved. In many situations the program needs to be understood in the absence of the person or persons who created it. The code can communicate the computational intent to some extent, but it cannot communicate the full conceptual intent of the programmer, nor can it communicate how the conceptual intent is related to the system's objectives.

2.2 Approaches to Reverse Engineering

There have been many efforts directed towards reverse engineering in the software engineering community. These activities fall under the umbrella of software maintenance. There are various types of information that can be extracted from the static code. Each type of information requires different kinds of and different degrees of analysis of the code. Various code analysis techniques have been developed
in different branches of computer science. Most of these techniques, however, can be classified according to the following three dimensions:

- **Formality of algorithm used:** Some approaches employ formal algorithms to analyse the code. Such approaches typically use programming language parsing technology [6, 4]. Some of them also use other kinds of algorithms which are not traditional language parsing techniques, but are derived from similar methods. For example Wills [87] uses a graph parsing approach for program recognition that maps program fragments onto common programming clichés found in programming. Similarly, Ward [82, 83] uses formal algorithms to extract information from code. Some of the approaches [34, 52, 21] use less formal (heuristics-based) algorithms for recognition.

- **Properties of programs analysed:** One can have systems that extract simple properties of a program, or one can also have an extensive set of properties analysed and extracted from a program. For example, a simple cross referencer like the C information abstraction system [15] can extract caller/callee relationships of different functions. This kind of system, however, can analyse only the static properties of a program. One can also have a system that extracts information like data and control flow graphs that capture how the program would behave under execution. Although fully modelling the exact nature of a program requires a complete semantic model (e.g., denotational semantics), one can have varying degrees of dynamic analysis of a system. Müller’s system, Rigi [53], gives more information than a conventional cross referencer, but its ability to model the dynamic properties of a program is less than that of Letovsky’s [44]. Complete modelling of the meaning of a program using formal techniques will be difficult for any real-world application, as the complexity is prohibitive.

- **Amount of domain knowledge used:** Understanding a program fully requires knowledge of the domain in terms of its structural components and
their functions and their relationships with one another. Also, for recognising what a program fragment is achieving, one has to map its components onto the domain concepts. This is called the concept assignment problem [8]. A system can also be analysed in terms of task-independent computational structures such as commonly used algorithms like sorting, searching, or data structures like stacks and queues. Knowing the domain of a software system and its typical functionalities, generates expectations as to what kind of domain-independent computational algorithms or data structures might be present in the code. A system that does not use domain knowledge, like Rigi, provides only limited information for understanding a program but can deal with very large programs. A system that uses more domain-specific knowledge in the form of clichés like PROUST [34] or SCENT [52] can give a better description of the program, but can deal with only very small programs. Clichés are typical programming strategies that people use in implementing a software system. They embody the conceptual knowledge of programmers. Complete program understanding requires encoding all the possible programming strategies in a knowledge-base and recognising a software system as a whole as a combination of these strategies. Cliché recognition, on the other hand, tries to recognise individual clichés that are sufficiently large to provide useful information about the system. Cliché recognition can be thought of as a midway between complete understanding of a system on the one hand and purely syntactic analysis of a system that does not use any conceptual knowledge on the other.

Most of the tools and techniques in software maintenance and re-engineering can be classified along these three dimensions. Some of the tools described below may use more than one technique. In such situations, the tool should be evaluated from the point of view of each of the techniques it employs. The following sections give a brief account of some of the approaches to reverse engineering.
2.2.1 Language Parsing and Compiler Technology

Many techniques developed for programming language compilers are useful in program analysis, specifically for maintenance and re-engineering. A compiler shares many features with a reverse engineering and re-engineering system. Both start with a program in some programming language\(^2\). Both need to analyse programs and extract semantically equivalent language-independent representations. Both compute the global properties of the program like data flow and control flow from the code. So, it is not surprising that many techniques developed in compilers and parsers are carried over to software re-engineering.

In its simplest form, a compiler can be considered as a reverse engineering tool since it finds computational structures such as loops, statements, procedural blocks etc., from the static code. But this may not be of much use from the perspective of a person trying to understand the program. Many software maintenance tools like Rigi [53] use techniques like data-dependency analysis that are normally used by compilers for finding various dependency characteristics of modules. Rigi also employs aggregation relationships to infer from the program what parts of the code can be combined to form an aggregate structure. In other words, a set of smaller sub-systems or modules can be combined to form an aggregate module.

Software Refinery tools (such as REFINE [69]) provide a rich environment and a set of powerful tools based on syntactic models of programs. REFINE provides a tool called DIALECT in which one can specify the syntactic elements of the programming language such as statements and expressions and also specify the grammar of the surface syntax. DIALECT then generates a parser and un-parsing (also called printer) for the specified language. The generated parser can be used to analyse any program written in the language. The parser generates an abstract syntax tree (AST) representation of the program into an object-base. An AST of a piece of program code represents the syntactic structure of the code explicitly in the form of a tree. The nodes of AST correspond to syntactic elements in the grammar.

\(^2\)Here we assume that the product that is being reverse engineered is program code, even though these ideas can be extended to reverse engineer other artifacts like design [16]
of the programming language in which the program is written. The leaves of an AST represents the lexical elements in the corresponding program. The parser also generates some extra information such as cross-reference information. The printer associated with the parser can be used to generate the exact representation of the program in terms of its surface syntax from the abstract syntax tree. REFINE provides an API (application programming interface) that allows one to build other tools based on REFINE which will perform further processing of the code from AST. REFINE also allows more than one grammar to coexist at the same time and the grammars are organised in a hierarchy where a new grammar can inherit from an existing grammar. This helps in creating various dialects of a programming language which only differ slightly. A REFINE generated parser also allows parsing of parts of programs such as individual statements independently of their context. This helps in specifying transformation rules in which any source code segment matching a grammatical pattern can be transformed into target code segments.

Tools like Rigi and REFINE employ techniques which are efficient and well understood to generate useful information about a software system. Dependency relationships and various metrics generated by Rigi are useful in a very coarse-grained "understanding" of a system. But Rigi provides almost no support to understanding how individual functions and modules are implemented and what strategies, data structures and algorithms are employed by programmers to implement a domain-level functionality. A REFINE generated abstract syntax tree provides a lot of useful information such as cross references and caller-callee relationships. But such information in itself is not sufficient to make sense of the software system. This information has to be used to arrive at higher-level conceptual information which helps a human understand the system. A knowledge-based system that can make use of syntactic information provided by systems like Rigi and REFINE can be more useful.
2.2.2 Program Transformational Approach

There have been many program transformation efforts that have resulted in the development of techniques relevant for reverse engineering. Program synthesis is a process of transforming a specification into program code. Reverse engineering on the other hand is the process of going backwards from a program to design and specification. In a sense, both of these activities can be viewed as program transformation techniques.

Kozaczynski et al. [39] presents a method that combines program transformation techniques with concept-recognition techniques to aid in the understanding and maintenance of programs. They identified four types of transformations depending upon four different levels of viewing a program:

- The first level is called the text level, which deals with string transformations.

- The second level, called the syntactic level, allows one to specify syntactic contexts as part of the pattern for a transformation to be carried out. At this level the patterns are specified on the elements of an abstract syntax tree rather than on the program text. Thus the effects of surface syntactic variations are avoided in the transformation.

- The third level is the semantic level. In this level the semantics of the programming language is considered while writing transformations. For example one could specify a constraint such that the transformation would be applied only to a specific part of the code where certain control and/or data would flow. For example, one can specify that only the condition part of an if statement that occurs in a while loop should be considered.

- The fourth level is known as the concept level. This is the most sophisticated level, in that it allows one to specify the constraints and transformation patterns that include conceptual knowledge. Concepts can be of many types. They can be concepts of the programming language, or they can be
language-independent programming concepts. The language-independent abstract concepts in turn can be of many types [40]. For example, they can be programming concepts such as sorting and searching, or they can be architectural concepts associated with interfaces to the execution environment such as operating systems, databases and networks. Some of the concepts come from the application domain and are independent of any programming language. For example, a business data processing application has concepts like payroll and wage structure. These concepts are similar to clichés described earlier. Recognising such clichés from source code can be very useful for understanding a software system.

A software system contains all these types of concepts.

Recognising language concepts is relatively easy. A parser for a programming language can be thought of as a recogniser for language-level concepts. Using a parser, one can recognise concepts such as assignment statements or conditional statements. Recognising abstract concepts is not easy. Kozaczynski [40] uses a concept model to suggest concepts to look for during recognition. His concept model is essentially an ISA hierarchy of concepts 3 which is a traditional knowledge representation formalism in AI. For example, CUSTOMER-RECORD ISA COMPANY-DATA. Recognising abstract concepts is difficult for various reasons. One of the reasons for this difficulty is that some parts of the code fragments constituting a concept may not be located in one place as consecutive text in the program. Syntactic information must be supplemented with other constraints for recognising abstract concepts. Each concept contains two types of information that help in recognition. One type of information is about different parts that make up a concept and the other type of information is a set of constraints on these parts. A CUSTOMER-RECORD might contain information such as name of the customer or type of the customer. In many situations the concepts can only be partially recognised. This is mainly because the concept model

---

3The ISA relationship between objects in semantic networks [42] has the meaning that if A is related to B by an ISA relationship, it means that an object of type A is a specialisation of the object of type B, in other words, the set of instances of A is a subset of the set of instances of B.
may be incomplete. In the approach by Kozaczynski [39] it is necessary to define in advance all possible/necessary transformations that contain the concepts to be recognised. Concept recognition which is purely based on patterns cannot partially recognise the concepts if the description of the concept is incomplete or alternative implementations of the same concept are possible. This approach is useful only in situations where we are looking for predefined bugs. The transformation mechanism will produce a version of the code with the bugs corrected. General purpose maintenance and reverse engineering should ideally also make use of information supplied by design and specification documents or by human experts.

2.2.3 Program Organisation/Visualisation

Various tools that assist in organising and visualising large software systems come under this category. They range from simple display tools that allow selective browsing of programs to sophisticated program animation tools.

Roman and Cox [72] describe five different criteria that help in classifying program visualisation tools. They are:

1. scope,
2. abstraction level,
3. specification method,
4. interface, and
5. presentation.

A program can be characterised by its code, data/control states, and execution behaviour. Scope determines which aspect of the program is visualised. The second criterion helps in answering questions like “What kinds of information are conveyed by the visualisation tool?” At the lowest level, one can simply highlight parts of the code and show the components. Alternatively, code can be abstracted into high-level modules and/or concepts and displayed. The third criterion looks at the mechanism
that allows an animator to construct a visualisation. The fourth criterion looks at the facilities the system provides for visual presentation of information. This includes the graphical vocabulary used by the animator as well as the interactions used by the viewer. The last criterion is based on how the system conveys information to a user.

Shneiderman et al. [74] give a list of strategies for displaying programs to facilitate browsing. Some of the interesting strategies are:

- **Fusion**: is a method of showing continuous text on multiple windows for the sake of continuity. This technique is useful when the program to be viewed is large and should be considered as one unit.

- **Synchronised scrolling**: When two or more related files are displayed in separate windows, and when a user scrolls (i.e., moves the text in) one window, the other windows are scrolled automatically. This helps in keeping correspondence among the information displayed in many windows.

- **Information on demand**: Any symbols like variables or procedure names, and data types can have properties associated with them. On user demand, the system can display the related properties of the symbol pointed to by the cursor. For example, if the symbol is a variable, its type information can be shown on demand.

- **Hierarchical browsers**: This technique makes the structure of a program code explicit in displaying it to a user. For example, class browsers in object orientated systems such as SmallTalk, Eiffel, and CLOS, use hierarchical browsing.

Stephan et al. [91] present a logic-based representation of information about software components and their interconnections. They model the interconnections using directed graphs. Once the interconnection information is represented as a set of axioms in logic, various validity checks can be performed on them and theorems can be proved about the components, using automated theorem-proving techniques.
Cordy et al.'s TuringTool [17] allows multiple views of a source program text via hierarchical source elision. TuringTool allows selective viewing of a program text at a particular level by hiding details of lower levels. It also has some non-structural elision mechanisms. One such mechanism is called rule-based elision. In rule-based elision, some parts of a program text can be selectively viewed based on semantic rules specified by the user rather than on the structure of the code. For example, one can view the part of the code that changes a particular variable. These kinds of elision rules allow one to encode different views of a program. It is possible to encode a wide variety of views including the views computed from dependency graphs. In essence, TuringTool is a code browser. Each view is essentially a projection of source code through the elision rules. Derived projections can be obtained by combining two or more elision rules. Rules can be combined by using one of the operations: (a) union, (b) intersection, (c) subtraction or (d) inversion. TuringTool also has a facility to save these derived rules and retrieve them on demand. These rules can also be parameterised. During the course of a session, a user may derive multiple projections corresponding to various alternative possibilities he/she is pursuing. These sets of derived elisions form a tree of projections. The branches of such a tree represent trails of the user’s reasoning about the program. These “audit trails” can be stored and retrieved for later use. These kinds of code browsers, though useful at a first level of maintenance for getting a feel for the code, will not take one very far in terms of understanding the functionality and semantics of the code. Nor do they provide the kind of feedback and hints that require subtle knowledge of what the program is doing.

Oman and Cook [58] suggest a different approach for organisation and visualisation of programs. They suggest that there are many parallels between a book and a computer program, and suggest that it will be helpful if a program is organised in the form of a book with a preface, a table of contents and an index. They point out that the initial description of a program is like the preface of a book and chapters are like program units (or modules, or packages). A cross reference listing in a program is like an index at the end of a book. They developed a system called BookMaker
which assists in producing a book format listing of programs from the source code text. They also conducted a controlled experiment to see the effectiveness of their principles of typographic formatting using 53 senior graduate-level students to recreate a missing procedure from a program listing. The results showed that the people who used the book format listing performed on average 10-20% better over the people who used a traditional listing. Also, they did the work in 68% of the time as compared to the programmers using a traditional listing. A similar experiment with professional programmers also showed that the people who used the book format scored better in a comprehension test.

Jerding et al.,[32] developed a method of visualising dynamic interactions in object-oriented program execution. They discovered some interaction patterns that can be used as abstractions in the understanding process. They implemented two prototype systems: one that computes overviews of large event traces and allow information to be filtered and highlighted, and another that embodies the notion of interaction pattern that can be compared with design-level execution scenarios.

The Rigi [53, 54] system developed by Hausi Müller's group is a classic example of attempts to provide a tool that can help re-engineer large-scale programs. It has an interesting collection of code analysis techniques neatly integrated into a user-friendly graphical environment. Rigi performs an exhaustive static analysis of source code to extract information such as dependency of different modules on each other, call-graphs, and data-dependencies. All this information is then stored in a database. Rigi has a good graphical interface where a person doing reverse engineering/re-engineering can view all this information as a set of interconnected nodes and links. For example, one can view the system at subsystem level, module level and so on, down to the individual function-level. Also, Rigi's interface allows a user to selectively filter out certain types of nodes/links from the graphical display. Rigi also allows manipulation of system description documents and the establishment of links among documents. One can document different views of different parts of the system and save these views for later use. In essence a software description is presented using an “electronic atlas” metaphor where the system is viewed from
“whole to part” (big-picture to details). Rigi also computes various metrics that help a user in the re-engineering process.

Later work on Rigi [80] proposed a method of user customised re-targetable domain-specific reverse engineering tools. They propose a flexible and scalable approach that can allow incremental compilation, tailorable interfaces and user-defined multiple views. They aim for the tools that can be used for reverse engineering of large software to the tune of a million lines of code.

2.2.4 Other Code Analysis Tools

A slice of a program is a collection of statements in a program that collectively achieve a common objective. A linear text of a program generally consists of many slices running in parallel through the same code. Program slicing [84] is a method of separating these slices to help understand the program. Weiser’s method works by iteratively solving data-flow equations based on program flow-graphs. If we can successfully separate the slices in a program, a software engineer can understand the program flow by understanding each slice separately. Also, tools such as program recognition can be applied on each slice separately. Hence, program slicing techniques show a good potential to be useful in program recognition to reduce complexity and increase efficiency. Recent work in program slicing has led to various techniques that can extract slices from source code. Korel [38] proposed a technique that can work in unstructured programs. He computes dynamic slices by separating the parts of the program that contribute to the computation of a variable value and the parts of the program that do not contribute. Lanubile and Visaggio [41] describes a method of using slices to extract reusable functions from unstructured source code.

Jarzabek [31] describes a program query language (PQL) that can be used by a software engineer for interactively understanding a software system by querying. CIA [14] and OMEGA[47] store structural information of programs in a relational database so that SQL queries can be used to extract information.
Baniassad and Murphy [7] proposes a technique called the *conceptual module querying*. A conceptual module is a set of lines of source code that is to be treated as a logical unit. Baniassad and Murphy provide a tool that allows a software engineer to iteratively define conceptual modules and analyse data and control flow interactions between a conceptual module and the rest of the code, as well as between two different conceptual modules. They also provide a way of querying these modules using a query language.

Kontogiannis [37] presents some experiments in using software metrics for detecting programming patterns in source code. He uses data and control flow metrics such as McCabe’s complexity measure as a signature for a code fragment.

Soni [78] proposes a new type of recognition as part of the development of a Maintainer’s Assistant. She tries to recognise *guidelines* that express global relations among design elements. One such relation could be the relation between data structure slots and how they may be accessed or updated.

Almost all the tools and techniques discussed here perform only syntactic-level analysis. But, as described in the next section, reverse engineering cannot be performed unless the program is “understood” to the extent of browsing what the program or program fragments are doing. This requires a different kind of program analysis that takes into account how programmers conceptualise various parts of the program and recognise such parts in the code.

### 2.3 Knowledge-Based Approaches for Software Analysis

There is no disputing the fact that it is necessary to understand the existing software system before any reverse engineering can be achieved. This understanding can be achieved at different levels and by various methods. Many of the tools and techniques described above compute interesting and useful properties from the code and help in the process of reverse engineering. But this methodology has its limita-
tions. It is increasingly being felt that static analysis needs to be augmented with knowledge-based program analysis to extract human-oriented conceptual information to help humans understand the code (see [8]). Most of the time documentation is not available; even if it is available, it is either insufficient or incomplete. Also, software systems change over time, and it is very hard to keep the documentation up to date with the system. This results in out of date documentation that cannot be trusted any more. This can be attested by the recent approaches to software documentation such as the I-DOC system, taken by Johnson [35, 36]. Johnson argues that documentation should be generated from the code on demand. In short, the code is the ultimate authority regarding the correctness of any interpretation of a software system. Hence, it is necessary to start with the source code and build higher-level interpretations from there. One may use other auxiliary information to help with the process of this interpretation, but code is the primary source. This requires program understanding and/or program recognition. Some of the recent programming environments, such as EiffelCase, advocates similar approaches where class documentation should be generated from the class diagrams and other descriptions.

2.3.1 Program Recognition Approach

Understanding programs and modelling how humans develop programs has been one of the favourite topics for artificial intelligence (AI) research. Major AI activities in this field fall into two groups\(^4\). The first group is known as *program synthesis* (also known as automatic programming, or knowledge-based program generation or synthesis). In this category, attempts have been made to create a system that will synthesise a program from a specification of its goals using a repertoire of explicitly-represented knowledge of expert programmers [70]. The second group is known as

\(^4\)There is actually a third group of efforts which are not "knowledge-based", but generally falls into the realm of AI. In this group, people have been trying to generate (or synthesise) programs from formal specification of software components such as functions. Efforts in this category use mathematical techniques such as theorem proving, or a combination of knowledge-based and formal methods [75, 22, 76].
program understanding, or program recognition. In this category, efforts have been made to create a system that will, given a program, "understand the program" and exhibit its understanding by various means, e.g., by answering questions about the program, or by finding semantic bugs in the given program [34, 87, 88]. In the case of program synthesis, the system is trying to do the job of an expert programmer; whereas in the case of program understanding, the system is trying to do the job of a person trying to understand a program for the purpose of maintenance or debugging. Although the two streams have different goals, they share many similarities. Both of them require a detailed knowledge of programming activity. Also, they both require a large store of commonly used programming patterns (or plans, clichés, or strategies). If a program can be represented as a hierarchical design where the top-level is the specification of its goal and the bottom level is the implementation in the form of code, then the process of understanding is going bottom-up from the code to specification, whereas the synthesis is going top-down from the specification to the code. Program synthesis and program recognition roughly correspond to the conventional software forward engineering and software reverse engineering, respectively.

Biggerstaff [8] gives an interesting classification of different approaches to program understanding using two dimensions. In one dimension he considers the formality and rigour of the method used and in another dimension, the amount of generality. On one extreme, there are methods which use very formal algorithms to analyse programs, and on the other extreme, they are heuristically-based procedures. Similarly, in the other dimension, on one end there are very general, domain-independent methods applicable to all domains and on the other end, very special methods that are applicable to a particular domain. Biggerstaff’s classification, although not considering many efforts in the field of intelligent tutoring systems, is still valuable and illuminating.

Program understanding systems have taken one of two broad approaches [65]. The first approach, known as the top-down approach, starts with the knowledge of the goals of the system being reverse engineered and a set of predefined program-
ming concepts variously known as clichés [70], programming plans [34], or abstract concepts in a conceptual model [12, 40, 39]. The system then tries to find out which of the plans in the library, combined in a particular way, will achieve the goals of the program under consideration. There may not be a single plan capable of achieving the desired set of goals. In such cases there might be some techniques for modifying the plans and/or combining more than one plan to achieve the objectives. Once the plans are identified, they are connected to the actual program code fragments. The second approach, known as the bottom-up approach [45, 88, 87], also has a library of programming plans. But in this case the analysis starts with the actual program and tries to find out which of the plans it matches. From these matched plans, the system infers higher level goals of the program being reverse engineered.

In addition to the efforts in the field of artificial intelligence there has been a considerable amount of work in the related field of intelligent tutoring systems (ITS). In the case of ITS, the goal is to create systems that "understand a program" created by a student to be able to find out what kind of strategies or plans she/he is using in writing the program. In the process, the system can also find out the wrong strategies, and give an effective feedback that helps a student to learn programming. We will look at some of these approaches to program recognition in detail.

2.3.1.1 Intention-Based Program Analysis

One of the early program understanding systems was PROUST, developed by Johnson and Soloway [34, 33]. The method employed by PROUST is called intention-based program analysis. PROUST analyses programs written by students and finds out their correctness and reports any possible bugs. Here the program is assumed to be syntactically correct and PROUST only looks for semantic bugs. PROUST represents programming knowledge at three levels. These three levels roughly correspond to three stages a student goes through in the program development process.

The first level, called the goal level, represents the goals of the program as described in the requirements of a students' assignment. In addition, PROUST also knows about some goals that are not directly stated in the problem statement.
The next level in PROUST is called the plan level where implementation-level strategies and plans for achieving the goal are specified. The plan library is indexed by the goals achieved by each plan. PROUST also has a library of buggy plans to help detect buggy plans in a student program.

The lowest level in PROUST's representation is called the code level. This level encodes the knowledge that allows any discrepancies in the recognises plan to be resolved by means of transformation rules and buggy rules. This type of knowledge helps when plans are not completely recognised, or there are some ambiguities to deal with.

PROUST uses a recognition algorithm that generates hypothesised plans or plausible plans that might be present in the code. This is done by selecting goals generated from the specification including any inferred goals. There are some heuristics that suggest the most promising order in which goals should be tried.

Although PROUST's method successfully recognised plans in student programs, it is not well suited for recognising clichés in large software systems. PROUST's recognition method is rigid and does not allow human input into the recognition process.

2.3.1.2 Program Recognition as Graph Parsing

Wills [87, 71] describes a method of using a graph parsing algorithm for recognising program plans. Wills uses flow graphs for representing programs as well as program patterns, called clichés, in the library. This reduces the problem of syntactic variations in implementing any particular plan. First, any given program is converted into an equivalent representation in a plan calculus formalism and then translated into flow graphs. Semantically equivalent programs that have some implementation difference map to the same representation in flow graphs. This helps in solving the problem of implementation variability discussed in Section 2.1.3. Graph grammars are used as a means to recognise clichés in the program represented in plan calculus. Hence, the problem of program recognition becomes a graph parsing problem. Wills’ system, called Recognizer, produces a description of a given program in terms
of clichés in the library and the actual information from the program such as names of variables and procedures. Recognizer uses a graph parser which is a derivative of chart parsers developed by Brotsky [11] with some features such as flexibility of control adopted from Lutz's algorithm [49].

Recognizer uses the concept of overlays\(^5\) that maps a higher-level specification of a cliché in terms of its implementation using lower-level clichés. Overlays allow Recogniser to work correctly even if there is an overlapping implementation of two or more clichés. An overlapping implementation occurs when a piece of code is shared by two or more clichés.

Wills' method uses data-flow graphs constructed from the source code. This method has advantages as well as disadvantages. One advantage of the data-flow representation of programs is that many implementation details are normalised to a common data-flow graph that captures the essence of the code. But data-flow graphs are not sufficient to accurately characterise a program. The flow graphs are augmented with control environments and other information. Another problem with this representation is that data-flow graphs may be easy to generate from programs in languages like LISP, but it is difficult to get such information in a reasonable way from programs written in languages like C where pointer and address manipulations are very common. Moreover, the flow graphs become really messy with complex control environments when applied to C programs.

2.3.1.3 Other Approaches

Program Recognition as Constraint Satisfaction

Woods and Yang [90] have proposed an approach for program-plan recognition which casts the problem into a pure constraint satisfaction problem (CSP). This method seems very interesting since a large body of knowledge exists in the traditional field of constraint satisfaction including some problem solving systems such as constraint logic programming (CLP). But pure constraint satisfaction does not help

\(^5\)An overlay is composed of two plans and a set of correspondences between their parts.
exploit the locality of constraints properly. Neither do constraints per se allow incremental progression towards understanding with the human-in-the-loop. Also, it is difficult to incorporate any heuristics into traditional CLP systems that might be useful in exploiting the structure of the special domain of programming. However, CSP can provide a formal basis on which program-plan recognition can be studied. Woods and Quilici [89] and Quilici et al. [67] have performed experimental studies of constraint-based program plan recognition techniques from the scalability point of view. They found that many well-known techniques from CSP such as constraints ordering, variable sorting, and forward checking have some useful effect on the performance of these algorithms. Mackworth et al.[50] presented an attempt to exploit the structure of the problem domain to increase the efficiency of constraint-satisfaction algorithms.

Memory-Based Program Recognition

Quilici [65] describes a system that extracts design knowledge from C programs useful for translating such programs into C++ programs. He studied the way students understand a program and came up with some observations: People do not match every plan in the library with the code, but have explicit indexes from a combination of program actions to entries in the library. This indexing phenomenon is not really new. It has been studied well in the field of natural language understanding where indexing plays the role of “getting reminded”, triggered by certain features of current natural language sentences being understood [73]. Also, people tend to try specialised plans first before trying alternative plans (a kind of depth-first search in an abstraction hierarchy). They also use some predefined knowledge about how plans are modified and look for these modified plans before moving on to the other pre-defined plans. Quilici’s approach is very similar to general constraint satisfaction with an additional indexing facility. This indexing allows one to quickly limit the number of plans to be considered for recognition in a bottom-up approach. Plans in Quilici’s method are not related to each other hierarchically, and hence it is not easy to naturally travel along the abstraction dimension when refinement or abstractions
are needed. Also, Quilici’s method does not address the problem of brittleness of the system in the light of insufficient information. It does not have any provision for the human software engineer to intervene and guide the system.

Although most of the above approaches provide interesting and useful techniques, none of them really address the problems of scalability (to large real world systems), and flexibility (in the face of insufficient knowledge). The granularity-based approach described in the next chapter tries to address these issues by providing the unique approach of putting the human in the loop for overcoming the limitations of the system and representing plans at various levels of detail so that they can be recognised at any level.

2.3.2 Other Knowledge-Based Approaches

Biggerstaff [9, 10] describes pattern-extraction methods with aid from a rich domain model. This model consists of design expectations for a particular domain including information such as typical terminology and typical module structures associated with particular problem domains. The recognition is done based on the correlation between conceptual structure and mnemonic procedure and variable names including the words in program comments. A grep-like pattern recognition is performed on program text including comments to cluster program elements.

Ning’s [56, 27] system, PAT, uses a bottom-up, rule-based inference engine to recognise clichés. PAT can recognise delocalised as well as overlapping plans. It also has rules that search for “key” events that should be searched for first.

Abdel-El-Hafiz and Basili [2, 1, 26] use a formalised knowledge of program structure and a library of “plans” to model the loops. Plans are organised as a tree structure with antecedent at the root that should match one of the main loop event and the edges of the tree corresponds to control conditions that correspond to the appropriate consequent.

Hartman’s UNPROG[29] proposes a recognition that is based on a specialised class of clichés called control concepts. UNPROG models the control flow by decom-
posing large control flow graphs and recognises them by graph matching.

2.3.3 Problems in Program Recognition for Reverse Engineering

There are many aspects of a software system that makes the job of program understanding and/or recognition very hard. In addition to the inherent difficulty of the problem, there are other problems that arise when program recognition is attempted on large real-world systems. Here we discuss some of these problems.

- **Size:** Many real-world software systems are extremely large, running into millions of lines of code. Understanding programs of such a scale is inherently difficult. Abstracting essential features of a large system and getting a global view of the system is also very difficult. One has to cut through a maze of details to find out what is going on in the program.

- **Complexity:** Many real-world systems are very complex and deal with a variety of entities. Each module is connected to other modules via various dependencies. These complex inter-relationships make it difficult to understand the code.

- **Non-local interaction:** Various parts of the program might have complex interactions with other parts of the code located far away. This kind of non-local interaction (called de-localised plans [46, 77]) makes it difficult to see the effects of a change in any part of the code will have on other parts of the code.

- **Poor/Non-existing/Out-of-date documentation:** Although documentation is crucial for any maintenance, most of the time there is not enough reliable documentation. There are many reasons for this state of affairs. Most projects are started without proper specifications. Even if there is a specification document, either it is incomplete or it might not reflect the actual implementation. This is so because the requirements change and the specification document
may not be updated to reflect the new changes. Other documentation such as
design documents and external/internal code documentation may not reflect
the actual current state of the code, since most of the time modifying code to
meet new and constantly changing requirements is given higher priority than
updating the corresponding documentation.

- **Lack of standardised styles:** Program writing is a lot like literary com-
  position in the sense that there are many ways of writing code to essentially
  achieve the same objective. Also, there is a lot of freedom to choose from dif-
  ferent alternatives at different places right from choosing variable/procedure
  names to choosing various algorithms. Personal tastes and styles have great
  influence coding. Since there are no standards for coding style, it is difficult
  for a person to understand the code written by other people. Many companies
  have their own guidelines for coding styles, but it is difficult to enforce these
  guidelines.

- **Programmer idiosyncrasies:** As mentioned in the previous problem, each
  programmer has his/her own ways of saying the same thing in any program-
  ming language. This makes it difficult for others to follow. Also, when some-
  body writes a program, his/her main objective is to make the program work
  rather than to make the code understandable to other people. In other words,
  a programmer gives more priority to communicating with the computer as
  compared to communicating with other people.

- **Poor organisation:** Even large systems could be understood better if they
  had a good organisation of their components. If different parts of a system
  that humans see as belonging together are put into one place and are properly
  interfaced to other modules, the effort required to understand the software
  system is reduced. Unfortunately, many programs are not well organised.
  Understandability has not been one of the key criteria for software design.
  This is especially true in the case of legacy systems. Some of the recent
  object-oriented software development methodologies addresses these issues to
some extent.

- **Aging**: As the system gets used over the years, many changes are made to it. Some of these changes might be minor bug fixes and some of the changes might be large-scale enhancements. Since changes are made to achieve the immediate goal, people generally do not give much thought to what is happening to the system’s overall organisation and understandability. Hence, it becomes more and more difficult to understand the system as a software system ages. The problem of aging is particularly true of legacy software systems.

### 2.4 Towards a Human-Centered Approach to Knowledge-Based Systems

One of the themes of this research is that the central role of the human expert using the system can help reduce the limitations of AI techniques and make AI problems tractable. Traditional AI approaches seem to ignore the expertise available from the human user. Hence these approaches result in systems that try to achieve every goal automatically, an impossible task in most cognitive domains. That is one of the reason why technology has been limited when applied to cognitive domains like software engineering. In contrast, we propose to make the user and the system work hand-in-hand to complement each other’s abilities to achieve common goals. We call this approach *human-in-the-loop* [59, 61, 60]. Because of the central role of the human user, we need to study the issues of human-computer interaction more closely. The rest of this chapter discusses these issues in the context of the specific application, KARE (Knowledge-based Assistant for Reverse Engineering), the reverse engineering system that is the major contribution of this thesis.

The idea of a human user assisting the system in problem solving is not new. It has been proposed in various forms by various people. In natural language processing and machine translation, humans pre-process the input data and/or post-process the output generated by the system. Pre-processing may involve simplifying complex
sentences so that the system will be able to understand/translate easily. Post-processing may involve making corrections to the translated text and/or completing the portions of the translation that the system could not handle. This has been known as human-assisted AI. There are some ideas known in the literature such as human-centered system design where the human-aspects of the system use are given central importance. This is mainly in terms of the usability of the system as to how effectively the user can utilise the functionality of the system without cognitive over-load.

Wills [87] proposed the idea of agenda-based cliché recognition with the facility for the human user to be able to manipulate the agenda. Quilici's DECODE system [66] also utilises human users' capabilities to augment the system's performance. Our ideas are very closely related to the ideas proposed by Wills and Quilici. We took this idea that is peripheral in other approaches, and designed KARE around this central notion of human-in-the-loop.

2.4.1 Human-Computer Symbiosis

One of the important issues to be considered while designing a system is the "division of labour" between the system and the user. Both the system and the user bring their own knowledge and expertise which can be shared profitably. At the same time, the interaction and cooperation should be natural and should not be intrusive. This thesis attempts to investigate these issues in detail. A list of the areas of expertise that each agent involved in the interaction can bring to the common platform is given follows.

- The computer system can provide:

  * A recognition engine for finding programming clichés in a software system.
  * Constant feedback so that the user knows what is happening in the system.
* High-level segmentation of code for search to reduce the search space.
* Context knowledge about which cluster to select for searching at any granularity object.
* Presentation and visualisation of results of recognition.
* An interface where the user can force success or failure of any node.
* An interface where the user can browse through recognised clichés.
* An interface where the user can satisfy some constraints and provide some controls in the process of searching for clichés.
* An interface where the user can interrupt the recognition process and then continue.

- The human user can perform the following activities (in KARE):
  * Specify where to look in the code (segment relevant chunks of the program).
  * Specify what to look for (select relevant program clichés).
  * Specify when to stop looking by interrupting.
  * Specify desired cliché recognition order using local controls.
  * Reorder the search agenda to change the priorities.
  * Augment partial, automatic recognition by recognising the remaining parts of a cliché by hand.
  * Monitor and intervene in the KARE’s recognition process to preempt a futile path.

In this thesis, we attempted a study of some of the above issues and how they can affect the performance and the usability of the system. Chapter 5 gives some of the results of our experiments with these notions. Most of these issues need further study to find out how to facilitate the user and the system to perform these functions effectively and naturally and communicate with each other effectively.
2.5 Summary

The problem of reverse engineering to extract high-level information from code is becoming increasingly important to maintain many large existing legacy systems. Reverse engineering requires program understanding, but the static code does not contain enough information for understanding its functionality since such information is lost in the process of translating the design goals of the system into programs in implementation. There have been some attempts to understand programs via program recognition, but they are limited to understanding small programs and do not scale-up to real-world applications. Since a program is a cognitive artifact created by the programmer, it is necessary to understand these cognitive connections between program code elements and design objectives. This can be achieved only by incorporating cognitively-oriented models into the realm of program understanding and reverse engineering. In the next chapter, we present a more detailed discussion of the cognitive nature of software engineering and the software artifact. Chapters 3 and 4 describe a method for program understanding based on the granularity formalism that addresses these limitations.
Chapter 3

Granularity-Based Representation and Reasoning

KARE uses a granularity-based formalism for representing program knowledge [23]. Common programming plans, known as clichés, are encoded in the granularity formalism as granularity hierarchies. The granularity formalism has many features that make it an excellent mechanism for representing programming knowledge. Some of these features are listed below.

- **Human-oriented strategic knowledge**: Granularity represents human strategic knowledge at various grain-sizes. Different ways (strategies) of achieving a goal can be effectively represented using granularity via its features such as objects and clusters (See Section 3.3).

- **Explicit representation of grain-sizes**: Granularity represents information at different levels of detail. Depending upon the requirement of a particular application and/or the task the user is interested in, recognition of clichés can be done at a very high level, or it can be refined to a more detailed level. This type of ability to travel naturally among various grain-size levels is useful for reasoning effectively in the face of incomplete information.

- **Approximation**: Granularity algorithms do not fail in a brittle way, as is often the case with many other AI representations and algorithms. Depending upon the availability of information and/or the ability of the algorithm, cliché recognition can occur at more refined levels, or only at a very high level.
• **Localised knowledge:** Almost all the knowledge in granularity hierarchies is distributed among various objects. Thus an object needs to know only about other objects that are in one of its associated clusters. This localisation of knowledge is one of the key aspects of granularity that helps in the knowledge engineering activity.

• **Distributed knowledge:** Control knowledge used by the recognition algorithm is distributed among different objects via various mechanisms such as contexts, controls, and constraints.

• **Flexible recognition method:** The granularity representation is very flexible and various recognition algorithms can use the representation. For example, SCENT [23] has a recognition engine that uses a restrictive, but efficient, top-down approach where the context is imposed top-down on an object that is to be recognised. The recognition algorithm used in KARE described in Section 4.1.3 uses a flexible control method that combines bottom-up and top-down methods.

• **Human input:** Granularity has some mechanisms that allow the designing of tools to place the human user in the loop of solving problems. For example, in our agenda-based recognition using granularity, the user can intervene and effect the recognition by pruning some partially recognised objects.

• **Communication:** The objects in granularity hierarchies represent programming clichés that are naturally understood by the human user, thus facilitating communication between the system and the user during and after the recognition.

In addition, granularity permits encoding of various types of heuristic knowledge into observers which help in reducing the search space while locating an occurrence of an object in the environment.
3.1 Roles of the Human User

In this thesis we talk about the human user using the KARE system in various roles at various places. This section describes these roles in detail. It is not necessary that different user plays each of these role. It may be that the same human user might play different roles presented here in different situations.

- **Software reverse engineer**: A software reverse engineer is a person who is performing the reverse engineering activity by using the KARE system on a target software system.

- **Knowledge engineer**: A knowledge engineer is a person who creates library of clichés (knowledge-base) using the AROMA knowledge engineering tool (See Section 4.2.2 for more details on AROMA). A knowledge engineer can be a the same person who is performing the reverse engineering task, or he/she may be a different person.

- **Software engineer**: A software engineer is a person who is performing a software maintenance/reverse engineering task on a target software system. Sometimes he plays the role of a software reverse engineer and some other times he may also play the role of a knowledge engineer.

- **Reverse engineer**: Sometimes the term reverse engineer is used as a short hand to refer to a software reverse engineer.

3.2 What is Granularity?

The idea of explicitly studying granularity was first proposed by Hobbs [30] as a means to model human reasoning that takes place at different levels of detail (grain sizes). This idea was formalised and extended by McCalla and Greer [23, 51]. They developed a knowledge representation formalism called the granularity hierarchy and used it for representing knowledge of strategies used by students in an intelligent tutoring system called SCENT [24].
The central idea of granularity is that human reasoning occurs at different levels of detail depending on the problem at hand. This idea can be utilised in knowledge representation for problem solving. A concept can be described/represented at various levels of detail. The task at hand will determine what details are relevant at what level. For example, when making a travel plan, a road is simply a one dimensional line and only the length of the line is sufficient. But when the same road is considered for the purpose of paving it, we need to consider other details such as the width of the road. Granularity formalises these notions for representing knowledge at various levels.

3.3 Granularity-Based Representation

Granularity hierarchies are directed graphs where nodes represent strategies. Strategies are connected to each other via two distinct types of relations or links. One type of link represents the abstraction relation and the other type of link represents the aggregation relation. The abstraction link provides for ISA and approximation links among strategies whereas the aggregation link provides for a part-whole relationship among strategy nodes. The abstraction relation and the aggregation relation are also called abstraction dimension and aggregation dimension respectively.

Figure 3.1 shows an example granularity hierarchy representing a common programming concept, LOOP. Figure 3.2 shows another example of a granularity hierarchy which describes the common cliché in programming called a linked list. In the figures, abstraction links are shown with solid lines and aggregation links are shown with dotted lines. The aggregation dimension specifies the constituent parts of a strategy object. All such parts of an object are grouped into K-clusters. An object can be decomposed into parts in many ways; so there can be many K-clusters for an object.

Using the example in Figure 3.1, let's illustrate these concepts in more detail. The concept of a loop (LOOP) in a program typically consists of a termination condition (TERMINATION-COND) and a body (LOOP-BODY) represented as a K-cluster. This type
Figure 3.1: A granularity representation for *Loops* in programming.

of decomposition of a concept into K-clusters is called *articulation*. Any concept can ultimately, through successive articulations, be decomposed into component concepts until we reach a level at which the concepts are directly recognisable by *observers* which look at the real-world. In Figure 3.1, strategy objects are shown as rectangles and observer objects are shown as ovals.

Similarly, along the abstraction dimension, an object can be classified into specialisations in many ways using *L-clusters*. This type of specialisation is called *refinement*. A loop can be refined along the abstraction dimension in many ways. In the example in Figure 3.1, the concept loop can be specialised in three different ways, giving rise to three different L-clusters:

1. **TOP-TESTED-LOOP** and **BOTTOM-TESTED-LOOP**, 

2. **SEARCH-LOOP** and **COMPLETE-LOOP** 


Each L-cluster specifies an equivalence relation on the set of instances of the object. Thus, the children of an L-cluster provide an exclusive-or relationship. An
Abstraction, aggregation
S-objects, Observers
K-clusters, L-clusters
Contexts, Context modifiers
Attributes and translations

Figure 3.2: A granularity representation for Linked List in programming.
object can be an instance of only one child of an L-cluster. Different ways of classifying an object are defined by different L-clusters.

Each strategy object in granularity (also called S-object) can have various types of constraints and controls for specifying how the parts are aggregated or decomposed and how they are abstracted or refined [20]. Constraints specify conditions that the members of a cluster need to satisfy to construct an S-object. On the other hand, controls specify conditions under which various actions such as refinement (recognising a refined version of an object), propagation (propagating recognition to a parent of an object) are to be performed. They provide mechanisms that allow some amount of customisation of the generic recognition algorithm. Some of these will be discussed later in this chapter. More details on the granularity formalism, and how it can be used in recognition can be found in [51, 25, 23].

Once concepts are represented in the granularity formalism, they can be used by reasoning algorithms that require knowledge of clichés. For example, we can use granularity hierarchies to recognise plans/clichés in program code using a program recognition algorithm. The following section describes one of the algorithms used for program recognition.

3.4 Granularity-Based Recognition in SCENT

SCENT [25, 24] is an intelligent advising system that helps students solve introductory programming problems by providing a strategy-level critique of their programs. SCENT uses granularity to represent strategies employed by students learning programming in LISP. SCENT has various tools that work on granularity-based representations. One of the tools is the recognition tool which works by trying to recognise the student program as an instance of one in the library of strategies used by students in programming a given problem. Hence the program is “understood” in terms of the represented knowledge. Once a student’s program is recognised, another set of tools is used to generate appropriate feedback to the student.

For every object successfully recognised or attempted for recognition, there is
an associated context. So when we say that an object is recognised, it means that the object is recognised in a particular context. Context roughly corresponds to the location in the program code where the object is found/ried. Section 4.1.1.1 contains more details about the notion of context used in granularity.

The SCENT granularity-based recognition algorithm works as follows: It starts by assuming that the student's program corresponds to the top-level S-object of the granularity hierarchy. This is a very important assumption and helps a great deal in dealing with the problem of alignment, i.e., the problem of finding out which object in the hierarchy corresponds to which part of the code. The SCENT recognition algorithm works by trying to recognise the root of the granularity hierarchy in the top-level context of the program. This in turn generates recursive calls to the recognition algorithm for the children of the object in order to meet the requirements of one of its K-clusters. When recursive calls to the recognition algorithm for children are performed, a corresponding set of possible contexts is generated by applying context modifier functions to the context of the parent object. When these recursive calls ultimately reach an observer object with a set of possible contexts, the observer is tested to see if it is satisfiable in any of the contexts. This test is performed by an observer function associated with the observer. Once all such observers needed for an S-object are recognised in the way described above, the S-object is also recognised provided that the additional constraints of the corresponding K-cluster are satisfied.

There are various types of recognition in SCENT, such as guided recognition, strict recognition, and partial recognition. We will not go into further details of these recognition approaches. More comprehensive treatment of these approaches can be found in other places [25, 19, 20].

Although the method described above works well in its intended domain (the domain of analysing students programs for advising), it is not suited for the problem we are attempting to solve in the thesis. Its primary weaknesses are the problems of scalability and flexibility. SCENT works in a domain of students' small recursive LISP programs. In the domain of reverse engineering, the programs to be recognised and the potential number of clichés are very large. There need to be considerable
enhancements to the SCENT approach, if it is to be able to work on large-scale programs. Another problem with the SCENT recognition algorithm is flexibility. SCENT employs a restrictive control for the recognition algorithm that imposes context from top-down. This method is simply not feasible for the domain of reverse engineering. Reverse engineering deals with large programs and it is not known beforehand where exactly in the source code a particular part of a cliché is likely to be found. In fact, one of the goals of program recognition is to find such locations and search for specific information at these locations. Thus, the context should ideally be generated as a byproduct of the recognition process, since the contextual information is not known beforehand.

We propose a new recognition algorithm that addresses precisely these issues of scalability and flexibility. The following chapter describes the method used in KARE.
Chapter 4

The KARE Approach to Granularity-Based Cliché Recognition

In this chapter we describe in detail our approach to program cliché recognition based on granularity. We also describe a system called Knowledge-based Assistant for Reverse Engineering (KARE) based-on the recognition technique. A software engineer can use KARE to recognise clichés in program code and display the instances of these clichés. He/she can also browse through the recognised clichés to see where in the source files the parts of the cliché are implemented. Figure 4.1 shows a high-level system architecture of KARE along with its components and data-flow connections.

As shown in the Figure 4.1, at the heart of KARE is a recognition engine based on granularity. KARE uses granularity-based representation of clichés in the form of granularity hierarchies. The source code is first parsed and converted into Abstract Syntax Trees (ASTs). The recognition agenda contains the clichés the user is interested in finding in the code. The recognition engine works on these main inputs (i.e., granularity hierarchies, ASTs, and agenda) and finds the occurrences of the clichés on the agenda in the source code. As a result of the recognition process, KARE creates instance hierarchies that correspond to each occurrence of the clichés on the agenda. Throughout the process, the user can interact with the system through KARE’s interface by manipulating various things such as the agenda, and the source code (via context). A more detailed account of how the human user interacts with KARE is described in Section 5.1.4. The following sections describe
Figure 4.1: An architecture of the KARE system.
the KARE's agenda-based recognition technique in detail.

4.1 Granularity-based Program Recognition for Reverse Engineering

As described in the previous chapter (Chapter 3), granularity-based recognition has been successfully used in the domain of intelligent tutoring system for recognising clichés in students' programs. The type of programming in intelligent tutoring systems situations is called programming-in-the-small as these programs tend to be rather of small size and typically implement one well-defined concept such as recursion. Program recognition needed in the context of reverse engineering must cope with programming-in-the-large. This thesis attempts to extend these techniques for recognising programming plans to large legacy software systems for the purpose of reverse engineering. The task in the case of programming-in-the-large becomes harder for two primary reasons:

- **Size**: The programs that need to be recognised in the domain of tutoring systems are rather small. The number of strategies contained in such programs is also small. But the size of the program to be analysed for reverse engineering task is quite large. The number of potential strategies that are likely to be present in a software system is also very large. The methods that work well in tutoring domains will encounter difficulties when applied to a large real-world software system.

- **Context**: A student's program defines one single global context. The program as a whole implements one target concept that a student is required/asked to implement. This will greatly restrict the amount of effort needed to recognise the program. In contrast, a software system implements a large number of concepts scattered in different functions and different files. Often there are many instances of the same concepts/clichés in a system. This poses a unique problem that is not directly addressed in student program recognition.
Many of the problems that occur in recognising programs in the small become intractable for large programs because the sheer size and complexity leads to a combinatorial explosion of possible clichés. In addition, there are many problems discussed in Section 2.1.3 that add to the complexity of the task of program recognition in the large.

In an attempt to solve these problems, and to facilitate a flexible, human-centred interaction with the recognition algorithm, we propose to extend the granularity mechanism in various ways. The following sections describe these extensions in detail.

4.1.1 Distributed Knowledge: Contexts, Constraints, and Controls

One of the key aspects of granularity is localisation of knowledge. Almost all the knowledge in granularity hierarchies is distributed among various S-objects and observers. Objects need to know only about other objects in their attached clusters. There are various mechanisms through which knowledge is distributed among objects and groups of objects.

4.1.1.1 Contexts and Context Modifier Operators

The logical as well as the physical place in the source code where a program plan is located is characterised by a context. The notion of context is very important in recognition. Context serves many purposes that can be grouped into three categories as described below.

1. **Before an object is recognised:** Contexts provide a natural place to put information about possible locations in the code where an instance might be found. This information could be obtained by various means. For example, the user might provide some clues (e.g., a file name, a directory name etc.) telling where to look in a large program. Alternatively, there might be some
information provided/propagated by other related objects that help in recognising the current object. All such information is stored in context objects utilised during the recognition process.

2. **After an object has been recognised:** After an object is recognised, the context associated with the object holds information that allows us to distinguish this instance from other instances of the same object. So when there are different instances of an object, they are found in different contexts. In short, a context encodes information that is needed to identify an instance of an object in real-world data.

3. **During the time an object is being recognised:** Context provides a place holder for any temporary information, such as information about a partially recognised object.

A *context* in KARE is represented as a 4-tuple \(<A, B, C, D>\) where:

- A is a set of global variable declarations,
- B is a set of global function definitions,
- C is a set of source file names, and
- D is a context data item.

A *context data item* (item D) is an element of the real-world data on which the recognition is attempted. This real-world data can vary depending upon the problem being solved. In SCENT, the real-world data consists of LISP symbols that are present in a student’s program. In reverse engineering, the input data to the recognition algorithm is the source code of the software system being reverse engineered. In KARE, we do not perform recognition directly on the source code text of a program. Instead, we pre-process the source code text using REFINE [69] to generate an abstract syntax tree (AST) representation of the program. The recognition algorithm works on the generated AST. Thus, in KARE, the real-world data consists of the nodes of the AST. When we refer to “source code” throughout
the reminder of this thesis, we mean the AST representation that corresponds to the actual source code. See Section 4.1.3 for more details.

It should be noted that to uniquely define a context, all four components of the 4-tuple representing a context need not all be present at any given time. In other words, one or more of these components can be empty at a particular time, depending upon the values of other components.

A context essentially represents a portion of the program code. Initially, the code can be any part of the source code. Once an object is recognised, the context corresponding to the recognised object represents the specific parts of the code that make up the instance of the object.

Contexts are related to one another in many complex ways. We have identified a few relations which are useful in recognition and are described below:

$C_1$ Equals $C_2$ : The relation $C_1$ Equals $C_2$ is true if the contexts $C_1$ and $C_2$ represent the same program elements.

$C_1$ Contains $C_2$ : The relation $C_1$ Contains $C_2$ is true if the program fragment represented by context $C_1$ contains the program fragment represented by the context $C_2$ in one of its sub-blocks. This is a general relation. There are specific relations which are sub-relations of the relation Contains.

$C_1$ Contains$_{true}$ $C_2$ : For this relation to be applicable, the program segment represented by $C_1$ should be a conditional statement such as an IF statement. The relation $C_1$ Contains$_{true}$ $C_2$ is true if the program represented by $C_1$ contains the program represented by $C_2$ in its success part of the program segment. Clearly the relation Contains$_{true}$ is a sub-relation of the relation Contains.

$C_1$ Contains$_{false}$ $C_2$ : Again, for this relation to be applicable, the program segment represented by $C_1$ should be a conditional statement such as an IF statement. The relation $C_1$ Contains$_{false}$ $C_2$ if the program represented by $C_1$ contains the program represented by $C_2$ in its failure part of the program segment. The relation Contains$_{false}$ is also a sub-relation of the relation Contains.
$C_1$ \text{Equal}_\text{alt} \ C_2$: The relation $C_1 \text{Equal}_\text{alt} \ C_2$ is true if the program segments represented by contexts $C_1$ and $C_2$ are part of alternative contexts for a selection statement such as a switch statement or a conditional statement.

$C_1$ \text{Equal}_\text{block} \ C_2$: The relation $C_1 \text{Equal}_\text{block} \ C_2$ is true if the program segments represented by $C_1$ and $C_2$ occur in the same block at the same level. A block in a C program is a sequence of statements that are delimited by open and closed curly braces "{" and "}".

Since contexts represent portions of the source code of a program, the context relations are relations on the source code fragments they represent.

\textit{Context modifier functions} (CMFs) are functions that manipulate context objects. Each CMF takes a context object and returns a modified context object. CMFs also take an additional argument that indicates the direction. The direction can be either top-down or bottom-up. CMFs are associated with K-clusters as well as L-clusters. Every cluster contains one CMF for each parent-child pair in the cluster. So, if a cluster consists of one parent and three children, there are three CMFs in the cluster. When the second argument for a CMF is top-down, then the CMF takes the parent context as an argument and returns the corresponding child context. Analogously, when the second argument is bottom-up, the CMF takes the context of the child object and returns a corresponding context for the parent.

CMFs are very useful in propagating information from one place to another. For example, if one of the children of an object is recognised in some context, the instance of that object will have a definitive physical location associated with it. We can then apply the CMF associated with the object and its parent to the context of the instance recognised to generate the context for the parent. Thus, the parent will get a more definitive (specific) context than it had before. This reduction in the scope of a context will help in reducing the scope of the potential context for the rest of the children of the object.

For example consider the fragment of granularity hierarchy shown in Figure 4.2. The figure shows a cluster which has two child objects and one parent object. Before
any object is recognised, the context for recognising any child object is all decla-
rations in the source code. But, if the child object LINKABLE-NODE is recognised,
then this information can be used to reduce the source code that is searched for
recognising the other children in the cluster, namely the object LINKED-LIST-HEAD.
Now the scope for searching for the object LINKED-LIST-HEAD contains only those
declarations whose type specifier is the same as the type-specifier of the object
LINKABLE-NODE. CMF operators essentially perform these types of context modifi-
cation which help in reducing the search space.

This reduction of the scope of a context will reduce the amount of search required
by recognition. Once a parent has a possible context computed by applying the
CMF, the context for other children of the parent can be computed by applying their
corresponding CMFs in top-down mode. In the current implementation of KARE,
these CMFs are encoded as Common LISP functions. Many of these functions can
be easily encoded using a predefined set of source code operators (SCO). These
operators are described in more detail in the next section under context constraints.

4.1.1.2 Constraints

Constraints allow restrictions to be placed on recognition. Constraints are placed on
a cluster of an object in a granularity hierarchy representing a cliché. For example,
constraints placed on a K-cluster specify how instances of different child objects
in the cluster should be organised to make it an acceptable instance of the parent
object in the cluster. Similarly, L-cluster constraints would specify conditions that
should be satisfied when one wants to propagate recognition from an L-cluster child
to its parent and vice versa. A constraint always belongs to a cluster (a K-cluster
or a L-cluster). There are four main types of constraints.

- Occurrence constraints

- Ordering constraints

- Context constraints
• Attribute constraints

Occurrence Constraints

Occurrence constraints of a K-cluster are unary constraints which specify how many instances of a child object are needed to enable one instance of a parent object. In many situations this constraint specifies that there should be just one instance, but in some situations it may require more than one instance of the same object. A K-cluster contains one occurrence constraint for each of the child objects in the cluster.

![Diagram](image)

Figure 4.2: An example of occurrence constraints.

As an example, consider a portion of the cliché shown in Figure 4.2. The figure shows only a part of the larger cliché LINKED-LIST. The S-object LINKED-LIST-DATA has one K-cluster with two child S-objects, LINKABLE-NODE and LINKED-LIST-HEAD. Hence, the K-cluster for the object LINKED-LIST-DATA has two occurrence constraints: one for each of these child objects. Here both occurrence constraints need exactly one instance of each object. In general, an occurrence constraint can specify that more than one instance required by a cluster.

An object that satisfies an occurrence constraint is a potential candidate for belonging to a cluster. There are other types of constraints that children of a cluster
need to satisfy before a cluster can be formed. These are various types of binary constraints discussed in the following sections.

**Ordering Constraints**

Ordering constraints of a K-cluster are relations between two child objects of the cluster. They specify a precedence relationship regarding the physical location in the source code of the instances of these objects. For example, if a K-cluster $C$ contains two children $O_1$ and $O_2$ then an ordering constraint may specify that the instance of a child object $O_1$ should appear before the instance child object $O_2$ according to the physical location of these instances in real-world data. It should be noted that these constraints are very simple in the sense they do not know anything about the semantics of the program elements and their scope rules. These are physical constraints. Any constraints more complex than these have to be specified by means of context constraints and attribute constraints.

![Diagram](image)

**Figure 4.3:** Ordering constraints.

Figure 4.3 shows an example of an ordering constraint. The fragment of cliché shown in Figure 4.3 shows one K-cluster for the object LINKED-LIST-DATA. The two child objects in the cluster have one order constraint. The order constraint specifies that the child object LINKABLE-NODE should occur before the object LINKED-LIST-HEAD in the physical location of the code in a file(s). It should be emphasised that the
Table 4.1: Source code operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARENT</td>
<td>Maps an object to its parent</td>
</tr>
<tr>
<td>CHILDREN</td>
<td>Maps an object to set of children</td>
</tr>
<tr>
<td>SIBLINGS</td>
<td>Maps an object to its siblings</td>
</tr>
<tr>
<td>TYPE-SPECIFIER</td>
<td>Maps an object to its type as declared in the source code</td>
</tr>
<tr>
<td>HEAD-OF-EXPRESSION</td>
<td>Maps a complex reference to its head</td>
</tr>
<tr>
<td>TAIL-OF-EXPRESSION</td>
<td>Maps a complex reference to its tail</td>
</tr>
</tbody>
</table>

constraints as shown in the Figure 4.3 specify only simple relations about the physical location of the instances of objects involved. Other examples of order constraints are before, after, in-the-same-function, in-the-same-file, etc. More complex constraints need to be specified using context and attribute constraints as described in the following sections.

**Context Constraints**

Context constraints specify complex relationships among context objects. Some of the relationships among context objects are described in Section 4.1.1.1. More complex context constraints can be specified by means of Source Code Object (SCO) operators. SCO operators are simple one-argument functions defined in the domain of objects that are nodes in the abstract syntax tree representation of a source program. They are similar in spirit to the source code algebraic operators proposed by Paul and Prakash [62] to help query the source code. SCO operators are functions that transform one node of the syntax tree to one or more other related nodes. Some of the SCO operators are listed in Table 4.1 along with their meaning. The operators that return a set of objects, such as SIBLINGS, take a predicate as an additional argument. This predicate can be any single argument function that takes an object and returns TRUE or FALSE. When this predicate is supplied, the corresponding SCO operator will return only those objects that satisfy this predicate. Hence, the predicate acts as a filter on the set of result objects of the operator.
These basic operators can be combined to achieve more complex transformations. For example, consider a complex reference involving a pointer which is an element of a record.

```c
record->element = 10;
```

If we want to find out the type of the record element, we can apply TYPE-OF (TAIL-OF-EXPRESSION (X)) which maps the reference X to the type of record item to which the variable element is a pointer.

**Attribute Constraints**

Attribute constraints are constraints on the attributes of the instances of two objects. Attribute constraints are defined between an instance of a parent object and an instance of one of its children in a cluster, or between attributes of instances of two different children in the cluster. Attributes are properties of an object that uniquely identify one instance of an object with another instance of the same object. When an object is recognised in a piece of code, its attributes are instantiated, i.e., assigned values. The values of these attributes are elements of the source code that make up the object. In Figure 4.4, the object LINKABLE-NODE has three attributes, viz., NODE-NAME, LINK-NAME, and INFORMATION. Attribute constraints are constraints placed on the attributes of one object and the attributes of another object. There are various types of attribute constraints. Section 4.1.2 discusses attributes and attribute constraints in more detail.

**4.1.1.3 Controls**

*Controls* encode various types of information that allow a reasoning algorithm (e.g., a recognition tool) to be supplied with local control knowledge. Controls allow the knowledge engineer to specify information that should override decisions taken by the general decision-making procedure built into an algorithm. Just like constraints, controls are associated with a cluster. There are various types of local controls associated with a cluster:
- Propagation controls
- Refinement controls
- Unfolding controls
- Local recognition order controls

Propagation controls specify whether or not recognition should be propagated to the abstraction parents of an object after it is recognised. By default, recognition is always propagated, but one can specify a control predicate that will determine dynamically if propagation is to be performed. Similarly, refinement controls specify if a recognition is to be refined towards the abstraction children of the object after it is recognised. Refinement of a recognised object results in a more refined version of the object that contains more details. There may be situations where a more detailed description of the concept is needed, in which case the refinement should be attempted. There may be other situations where there is no need to find more detailed information of the concept that is recognised. In such cases, refinement only adds extra unnecessary work to the recognition tool and can be avoided safely. For example, if an object representing the linked list operation is recognised (see Figure 3.2), one can refine the concept by attempting to recognise its abstraction children, namely, singly linked list and doubly linked list. If a situation is such that it does matter whether a linked list is a doubly linked list or not, then the refinement should be attempted. If the situation does not require a specific version of a linked list, there is no need to refine.

Unfolding controls specify control information that helps in the process of unfolding. Unfolding is the process of expanding a cluster of an S-object to place its children on the agenda. See Section 4.1.3.2 for more details on the unfolding operation. For example, an unfolding control can specify which cluster should be selected next from the set of available clusters of an object for recognition.

Local recognition order controls specify an order in which the children of a cluster should be attempted for recognition. Sometimes it is necessary to impose a
sequential recognition ordering some of the children so that they can be recognised efficiently.

4.1.2 Sharing the Knowledge: Attributes and Attribute Translations

As described earlier, most of the knowledge in granularity hierarchies is distributed among various objects and clusters. This makes it efficient and convenient to use the granularity hierarchy since at each stage one needs to consider only the information local to that object and/or cluster. But, sometimes local knowledge alone will not be sufficient for using granularity. We also need to access non-local information, i.e., information from objects/clusters that are not directly connected to the present cluster/object. This section describes a mechanism which allows non-local information to be propagated from one place in the hierarchy where the information is collected to other places in the hierarchy where the information is needed/used. Attributes and attribute translations help achieve the sharing of non-local information. Attributes also help in defining some of the fine-grained constraints in a better way.

4.1.2.1 Attributes

Attributes are properties associated with an object in a granularity hierarchy. Attributes capture essential features of a real-world instance of a granularity object. Every granularity object has a list of attribute names associated with it. When an observer object is recognised in the real world, an instance of the object is created and its attributes are assigned values from the data corresponding to the object being recognised. On the other hand, when an S-object is recognised, its attributes get their values from the values of the attributes of other objects (usually the children objects of the S-object) which are responsible for recognising the object under consideration. This assignment of values to attributes is called instantiation. Attribute values allow us to distinguish one instance of an object recognised in a context (See
Section 4.1.1) from another instance of the object recognised in a different context.

Figure 4.4 shows examples of objects, their attributes and corresponding values for the attributes. Different instances of an object will have different values for their attributes. In Figure 4.4, the object LINKED-LIST-DATA has four attributes viz. NODE-NAME¹, LINK-NAME¹, INFORMATION¹, and HEAD-VARIABLE¹. Similarly the object LINKABLE-NODE has three attributes viz. NODE-NAME², LINK-NAME², and INFORMATION². Attributes help in many ways to define granularity hierarchies. First, they facilitate differentiation of various instances of the same object. Second, they help us to propagate information about one object to other objects. This is achieved by means of attribute translation rules. Attribute translation rules provide a method of transferring attribute values of an object to its parent and/or to its children. Attributes and attribute translations also act as a set of constraints.

![Diagram of object attributes and translations](image)

Figure 4.4: Object attributes and attributes translation rules.

The idea of attributes and attribute translations is not new. For example, compilers use attribute translation grammars in which each symbol (terminal, non terminal) has a set of attributes associated with it. When a non-terminal symbol is recognised using a reduction, its attributes get instantiated [3, 81]. Attributes are also adopted by at least one other program understanding/recognising system called
Recogniser [87] by Wills. See Section 2.3.1.2 for more details on Wills' graph parsing approach to program understanding used in Recogniser. Wills calls the rules that specify translation of attribute values from one object to another as *attribute translation rules or attribute transfer rules*.

There are two different types of attributes. In the field of compilers they are called a) inherited attributes, and b) synthesised attributes [3, 81]. *Synthesised* attributes are attributes which get their values from the child objects in one of its clusters. In other words, synthesised attribute values travel upwards from child to parent. *Inherited* attributes are attributes whose values get propagated from parent to child. Hence, values for an inherited attribute comes from a parent to its child. The distinction between synthesised and inherited attributes will become more clear in a subsequent section.

### 4.1.2.2 Attribute Translation Rules

Attribute translation rules (ATRs) are a set of rules that describe how the attributes of child objects are related to the attributes of their parent object and vice versa. ATRs are associated with a cluster. Hence, each cluster potentially can have a different set of ATRs for the same parent-child combination. ATRs in KARE are represented in such a way that they can be used in both ways, top-down as well as bottom-up. While going from a child object to its parent object, the rules are used in bottom-up mode. Similarly when going from an object to its children, the same set of rules are used in top-down mode. In fact ATRs can be viewed/used many ways. For example, they can be viewed as:

1. **Computing a parent's attributes:** Given a set of attributes along with their values for an instance of an object, we can use ATRs to compute the attributes values for an instance of a parent of the object. In this case, the result of applying ATRs would be a set of attribute values for the parent object. (similar to synthesised attributes)
2. **Computing child attributes:** Given a set of attributes along with their values for an instance of an object, we can use ATRs to compute the attribute values for an instance of a child of the object. In this case, the result of applying ATRs would be a set of attribute values for the child object. (similar to inherited attributes)

3. **Constraints:** Given a set of attributes and their values of an instance of an object and a set of attributes and their values of an instance of a parent of the object, we can use ATRs to check if the two sets of attribute values satisfy the constraints specified by ATRs. The result in this case would be either yes or no.

4. **A combination of 1 and 3 above:** This case arises when some of the parent’s attributes have values and other parent attributes are uninstantiated. The result of applying translation rules would then be assigning values for some of the attributes that do not yet have values in such a way that the existing attribute values are consistent with the newly assigned attribute values according to the constraints specified by ATRs.

5. **A combination of 2 and 3 above:** In this case the rules are applied in top-down mode and hence some of the child object attributes will be assigned values as a result of applying ATRs.

Figure 4.4 shows a portion of a granularity hierarchy consisting of a K-cluster with object attributes and corresponding attribute translation rules. Object LINKED-LIST-DATA is an aggregation parent of the objects LINKABLE-NODE and LINKED-LIST-HEAD. The two objects form one K-cluster. Object LINKABLE-NODE has a list of attributes: NODE-NAME, LINK-NAME, and INFORMATION. These three attributes capture the essence of an instance of the object LINKABLE-NODE. When an instance of LINKABLE-NODE is recognised, these three attributes are instantiated, i.e., assigned definite values from the real-world data in which the recognition is attempted. In the case of recognition of a cliche in a program, these three attributes
will be given values from the program text (or elements from its equivalent representation such as nodes of the AST). Similarly, object LINKED-LIST-HEAD has two attributes: NODE-NAME and HEAD-VARIABLE. Object LINKED-LIST-DATA has the list of attributes: NODE-NAME, LINK-NAME, INFORMATION, and LINKED-LIST-HEAD.

It should be noted that the attributes of a parent object need not be a simple union of attributes of its aggregation children. In fact, the attributes of a parent can be completely different from the attributes of its children. In any case, values for parent attributes are always computable from those of the children, using attribute translation rules. Since ATRs are always associated with a cluster, we will use the term the cluster to refer to the cluster through which the current parent and its children are attached to each other. So it makes sense to say the parent of a cluster to refer to the parent object, and children of the cluster to refer to the set of children that are attached to the parent via the cluster. Figure 4.4 shows the format of simple translation rules. In this format, each rule consists of a pair of symbols. The first symbol in each pair is the name of an attribute of the parent of the cluster under consideration. The second symbol in each pair is the name of an attribute of one of the children of the cluster. This is the simplest form of an ATR. This basically says that the set of parent's attributes are mapped one-to-one to the union of the sets of children's attributes. Also, in this particular example, the names of the attributes of the parent are the same as the names of the attributes of the children.

The algorithm for applying ATRs closely resembles the unification algorithm used in Prolog [48]. The following is a simplified version of our algorithm, named the attributes unification algorithm, used for applying ATRs on the set of attributes of an object. There are two different versions of the algorithm, one for applying while going bottom-up from a child to a parent, another while going from a parent to a child. Section 4.1.3 describes in detail how these algorithms are used in recognition.

When we talk about applying ATRs, we often use terms such as source object and target object. A source object is the object which has attribute values assigned to it and a target object is the object whose attribute values we want to compute using ATRs. So, when we go bottom-up, the source object is a child object and
ATTRIBUTES-UNIFICATION-ALGORITHM
   (RULES P-ATTRS P-ATTR-VALUES C-ATTRS C-ATTR-VALUES)
   -- bottom-up version
1. For each RULE in RULES, repeat the following steps
2. Let LHS = first (RULE) and RHS = second (RULE)
3. If LHS is in P-ATTRS then
   /* Parent needs this attribute */
   3.1 Let P-A-VALUE = RETRIEVE-VALUE (LHS, P-ATTR-VALUES)\ and
       C-A-VALUE = RETRIEVE-VALUE (RHS, C-ATTR-VALUES)
   3.2 If (NOT-NULL P-A-VALUE) and (NOT-NULL C-A-VALUE) then
      /* Child also do not have value, fail */
      3.2.1 If P-A-VALUE =/= AVM(C-A-VALUE), fail
   3.3 Else If (NOT-NULL C-A-VALUE)
      /* Child has value, propagate up */
      3.3.1 add LHS and AVM(C-A-VALUE) to P-ATTR-VALUES
4. Return P-ATTR-VALUE

Figure 4.5: Attributes unification algorithm.

the target object is a parent of the source object. When we go top-down, the above
roles of the objects get reversed, i.e., the parent object becomes the source object
and the child object becomes the target object.

Figure 4.5 shows the bottom-up version of the attributes unification algorithm.
The algorithm is designed in such a way that if the parent object as well as the
child object already have values for their attributes corresponding to a rule, the
unification algorithm acts as a verifier. In this case, it will check if the assignment
of values to the attributes is consistent with the rules. If the assignment of attribute
values is not consistent with the rules, the algorithm will return a failure. Otherwise,
it returns a success. On the other hand, if a parent attribute corresponding to a rule
does not have a value yet (i.e., the attribute is uninstantiated), then the algorithm
assigns a value to the corresponding parent attribute in such a way that the rules
are satisfied. The algorithm for going top-down is similar.

The format of ATRs described in Figure 4.4 is simple. These ATRs transfer the
values of the attributes of a source object to the target object and vice versa with a
possible renaming of the attribute names. It is not always possible to describe the relationship among attributes of child objects and a parent object in such a simple manner. Sometimes we need to modify the value of an attribute of the source object to obtain the value for the corresponding attribute of the target object. This can be achieved by means of *Attribute Value Modifier* or AVM operators. An AVM operator is a simple one argument function that takes an attribute value of an object and returns the value of the corresponding attribute of the target object. In most cases an AVM operator is the *identity* function that simply returns its argument. In such a case, the algorithm becomes simple, as shown in Figure 4.5. But this is not necessarily always true. If the AVM operator is not an identity function, then the operator is applied to the value of an attribute of the source object before it is compared against the value of the target object in the algorithm of Figure 4.5.

![Diagram](image)

**Figure 4.6: An example of Attribute Value Modifier.**

Figure 4.6 shows an example of an AVM operator. One of the translation rules shown in the figure relates the attribute `NODE-PTR-DECLARATION` of the object `LINKED-LIST-INSERT` to the attribute `NODE-POINTER-REF` of the object.
ALLOCATE-LINK-NODE. It also specifies the AVM operator DECLARATION-OF. That means when the attribute translation rules are used to compute parent attribute NODE-PTR-DECLARATION, the corresponding AVM operator is applied on the value of the child attribute NODE-POINTER-REF. This example also shows a non-trivial translation rule where the names of the child attributes can change when they get transferred to its parent and *vice versa*.

### 4.1.2.3 Attribute Constraints

In Section 4.1.1.2, we discussed various types of constraints that are used in defining a granularity hierarchy for a cliché. Attribute constraints are constraints placed on attributes of objects in a hierarchy. In the last section we discussed attribute translation rules (ATRs). ATRs define constraints on attributes, because any child object instance that does not satisfy the ATRs of its cluster is rejected. Some times the constraints imposed by ATRs are not sufficient. There are two different types of attribute constraints:

a) Constraints placed on two child objects of a cluster,

b) Constraints placed on a parent object and one of its children.

![Diagram of attribute constraints](image)

**Attribute Constraint:**  
\[
\text{NOT-CONTAIN} \ (\text{NODE-DECLARATION}, \text{HEAD-VARIABLE})
\]

Figure 4.7: Attribute constraints: an example.
Constraints specified by ATRs belong to type b) above, as they impose constraints on an object and one of its children. Type a) constraints cannot be specified by ATRs. They have to be specified as explicit constraints similar to occurrence constraints and ordering constraints as described in Section 4.1.1.2.

Figures 4.7 and 4.8 illustrate the two types of constraints. The constraint NOT–CONTAIN says that the head variable given by the attribute HEAD–VARIABLE should not be within the scope of the declaration of the node itself which is given by the attribute NODE–DECLARATION. This constraint is needed to prevent link declarations of the LINKABLE–NODE to be mistaken for the head variable declaration as both are of the same type: pointers to the structure of type NODE–NAME. This is an example of a constraint on two child objects.

As an example of a constraint defined between a parent object and a child object, consider the fragment of the cliché hierarchy shown in Figure 4.8. The figure shows that the object LINKED–LIST–INSERT has one K-cluster which has four child objects. The constraint SAME–DECLARATION in the figure is defined between an attribute of one child and an attribute of another child. On the other hand, the constraint TYPE–OF specifies a constraint on one attribute of parent (NODE–NAME)
and an attribute of child (NODE-POINTER-REFERENCE-1). These two attributes of the constraint SAME-DECLARATION, are synthesised attributes which get propagated upwards from children to parent. The children of the objects ALLOCATION-CALL, NULL-TEST, INSERT-NULL-LIST, and INSERT-NON-NULL-LIST are not shown in Figure 4.8. The values for the attributes (for example, attributes ALLOCATION-CALL, NODE-POINTER-REFERENCE-1) of these nodes come from their children. On the other hand, the attributes NODE-NODE, LINK-NODE, and HEAD-VARIABLE for the object LINKED-LIST-INSERT comes from its parent. Hence they are called inherited attributes.

Attributes and attribute constraints provide a powerful mechanism to share knowledge among various objects. Together with context modifier operators, they provide a very effective means to constrain search by propagating partial recognition among other objects.

4.1.3 Flexible Control: Agenda-Based Recognition

The agenda in KARE serves many purposes. First, it provides a focusing mechanism where the recognition algorithm is directed/re-focused based on what is on the
agenda. Second, it provides a basis for the user to direct/guide the recognition algorithm. Figure 4.9 gives a schematic view of the agenda along with its relationships to various other parts of KARE.

The following section describes the agenda used in KARE. Section 4.1.3.2 describes the general algorithm for recognition and Section 4.2.4 gives an example illustrating how the algorithm works.

4.1.3.1 What Is an Agenda?

An agenda is a data structure that contains a list of clichés (called agenda items, or simply items) that are to be worked on by the recognition algorithm. Each agenda item exactly corresponds to an object in the granularity hierarchy. An item stores a pointer to one of the objects in the hierarchy, as well as a pointer to a list of instances of the object. An agenda item also contains a pointer to a context in which it is to be searched. But once some instances of the object associated with the item are recognised, more specific contexts are created that correspond to each of the instances. An agenda item also contains some information such as if this item has been reordered on the agenda by the user, or if there is a break point set on this item. The instance associated with an agenda-item can be in any one of the :start, :unfolded, :partial, or :strict states depending upon its recognition status. The meanings of each of these states is described in detail later in this section. The instantiated object also has information about the context in which the instance was found. Figure 4.10 shows an example agenda item.

The agenda is organised as a multi-level priority queue. The first level, (i.e., the top-level) contains the clichés that are either specified explicitly by the user, or derived by the system. Each agenda item also contains a sub-agenda, local to itself. The local agenda for an item is generated by the system while trying to recognise the cliché corresponding to the item. When recognition is attempted on this item, if the local agenda is empty, all the subgoals needed to satisfy this item have been recognised, so the item is recognised by checking various constraints. If the local agenda is not empty, an item from the local agenda is selected and recognition is
<table>
<thead>
<tr>
<th>Object Name</th>
<th>Linked List Insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic S-object</td>
<td>The object LINKED-LIST-INSERT</td>
</tr>
<tr>
<td>Current Cluster</td>
<td>K-CLUSTER (1) LINKED LIST INSERT</td>
</tr>
<tr>
<td>Contexts</td>
<td>(CONTEXT-1 CONTEXT-2... )</td>
</tr>
<tr>
<td>Instances</td>
<td>(INSTANCE-1 INSTANCE-2... )</td>
</tr>
<tr>
<td>Local Agenda</td>
<td>(AGENDA-ITEM-1 AGENDA-ITEM-2... )</td>
</tr>
<tr>
<td>Local agenda modified</td>
<td>YES (or NO)</td>
</tr>
<tr>
<td>Break after this item</td>
<td>YES (or NO)</td>
</tr>
</tbody>
</table>

Figure 4.10: An example of an agenda item.

attempted on it. As recognition proceeds, items are removed from the agenda and moved to a recognition cache and (possibly) more items are added to the agenda. A local agenda holds the objects that correspond to sub-goals generated by the system in order to satisfy one of the main goals of recognising an agenda item. The following section describes the algorithm in detail.

4.1.3.2 The Recognition Algorithm

Recognition in KARE happens in two phases. In the initial phase, the system makes sure that agenda items are grounded in a specific context in the source code. A context is a physical location in the source code where an object might occur. Depending upon the object being recognised, a context can be a program statement, or a variable declaration or any other portion of the source code. Actual recognition happens in the second stage. Some of the objects placed on the agenda may have a context specified by the user (i.e., the software engineer). This can be done by the user by selecting parts of the source code in which she/he wants to search, and attaching it to the agenda item. In such cases, KARE attempts recognition in the context with the reduced scope for that object as specified by the user. If no such context is specified, then the entire source code is searched for occurrences of the object.
When an object is being attempted for recognition, the recognition algorithm makes many decisions such as whether to unfold an object, or attempt recognition, or propagate recognition etc. Many of these decisions are based on the current status of the object and the information provided by the user interactively. An instance of a granularity object can be in any of various possible states during the course of the recognition process. The overall status information of an object is determined by two primary status indicators: a) parts status \((\text{parts-status})\), and b) propagation status \((\text{prop-status})\). Parts-status of an object indicates the result of attempting to recognise the object by means of recognising its aggregation parts. Similarly, prop-status indicates the result of attempting to recognise the object by means of recognising one of its abstraction children. The overall status of an object is a combination of these two status indicators. The two status indicators of an object can take values from the set of possible status values. They are \(\text{:start, :unfold, :partial, :strict or :fail}\). The meaning of each of these states is described in Table 4.2.

<table>
<thead>
<tr>
<th>Indicator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>:start</td>
<td>The object has just been placed on the agenda</td>
</tr>
<tr>
<td>:unfold</td>
<td>The object is unfolded and its children have been placed on the agenda</td>
</tr>
<tr>
<td>:partial</td>
<td>The object is partially recognised, i.e., some of its children are recognised</td>
</tr>
<tr>
<td>:strict</td>
<td>The object is completely recognised, i.e., one of its clusters is recognised with all its associated constraints satisfied</td>
</tr>
<tr>
<td>:fail</td>
<td>The object could not be recognised</td>
</tr>
</tbody>
</table>

In addition to the values listed in Table 4.2, an object’s status indicators can have one of the two additional values viz., \(\text{:parts and :prop}\), depending upon the type of recognition that is being attempted. If an object’s prop-status has value \(\text{:parts}\), then we should look at its parts-status for the over all status of the object. Similarly, if an object’s parts-status has value \(\text{:prop}\), then we should look at its
prop-status for the over all status of the object. Table 4.3 describes these rules of how the two status indicators are combined to give an overall status for an object. The horizontal top-row in Table 4.3 indicates the value of parts-status of an object and the vertical first row specifies the prop-status. To find out the overall status of an object, the values of prop-status and parts-status are used to index into this table. The corresponding entry in the table gives the current overall status of the object.

<table>
<thead>
<tr>
<th>prop-status</th>
<th>parts-status</th>
</tr>
</thead>
<tbody>
<tr>
<td>:prop</td>
<td>:start</td>
</tr>
<tr>
<td>:start</td>
<td>:unfold</td>
</tr>
<tr>
<td>:partial</td>
<td>:strict</td>
</tr>
<tr>
<td>:fail</td>
<td>:fail</td>
</tr>
</tbody>
</table>

The top-level algorithm for the agenda-based recognition is shown in Figure 4.12. Figure 4.11 shows the caller-callee relationships among the functions given in Figure 4.12. In each recognition cycle, an agenda item is selected and recognition is attempted on that item. When an object on the agenda is recognised, it is moved to the recognition cache. The recognition cache contains all the objects that have been recognised completely. When an object is moved from the agenda to the cache, its recognition is propagated to its parents via both the abstraction and aggregation dimensions. While propagating, if an object is fully recognised, it will be moved from agenda to cache and its parents will be placed on the agenda.

Recognising an object depends upon whether it is an observer object, which can be recognised directly, or an interior object (S-Object), which needs to be recognised indirectly. Recognising an object is achieved by the three algorithms RECOGNISE, RECOGNISE-observer, and LOCAL-RECOGNITION shown in Figure 4.13. Initial grounding of an object in a specific context is done by the two algorithms
Figure 4.11: A module interaction diagram showing caller-callee relationships among major functions in the KARE recognition algorithm (actual control flow is not shown).

Algorithm AGENDA-BASED-RECOGNITION

1. Initialise the agenda (Algorithm INITIAL-SWEEP)

2. loop until all items are recognised or cannot proceed further
   
   2.1 Select an item from the agenda
   2.2 Recognise the object corresponding to the item (Algorithm RECOGNISE)

3 Stop

Figure 4.12: Top-level agenda-based recognition algorithm.
Algorithm RECOGNISE (ITEM)

1. If the ITEM’S object has a direct observer then
   1.1 Call RECOGNISE-OBSERVER
   1.2 propagate recognition and the context
      a) to abstraction parents
      b) to aggregation parents
   Return
2. If the object is not unfolded? then
   call UNFOLD
3. If the local agenda is empty then
   Return
   Else call LOCAL-RECOGNITION

Algorithm RECOGNISE-OBSERVER

1. Extract the contextual information from the object.
2. Run the observer function on the context to generate a
   list of instances
3. Remove the item from the agenda

Algorithm LOCAL-RECOGNITION (AGENDA-ITEM)

1. IF LOCAL-AGENDA is empty and current cluster is a fail,
   1.1 call EXPAND-ONE-STEP
   1.2 Goto step 3
2. ELSE If LOCAL-AGENDA is empty
   Return failure
3. Pop the first element of the LOCAL-AGENDA(AGENDA-ITEM)
4. Check all the constraints.
   4.1 If the constraints are satisfied
      Remove the item from the agenda
   4.2 Else, generate all the remaining children yet to be recognised
         place them on agenda.
   4.3 Return

Figure 4.13: Algorithm for recognising an object.
INITIAL-SWEEP, and UNFOLD shown in Figure 4.14.

Algorithm INITIAL-SWEEP (ITEM)

1. If ITEM has a direct observer, then return
2. Else
   2.1. Unfold the item by calling UNFOLD (ITEM)
   2.2. Return

Algorithm UNFOLD (ITEM)

1. If the top of the local agenda of ITEM has a direct observer, then return.
2. Else
   2.1. Pop the top item off the local agenda
   2.2. If there are aggregate children then select a K-cluster
       -- Place all the children of the K-cluster on the local agenda
   2.3. If not, then select a L-cluster
       -- Place all the children in L-cluster on the local agenda
   2.4. call UNFOLD (AGENDA-ITEM)

Figure 4.14: Algorithms for initialising and unfolding agenda items.

The unfolding operation involves taking clusters of an object and generating all the children for each of the clusters and placing them on the agenda. The unfolding operation also applies CMF operators (See section 4.1.1.1) while unfolding to generate contexts for the child objects.

In each cluster, the order in which the children of the cluster are to be recognised is specified explicitly. This order, called the local recognition order, is encoded explicitly using local controls. This is necessary to make sure that some objects are recognised before others are attempted. In addition, there are various types of constraints such as context constraints, ordering constraints and attribute constraints that place restrictions on how child objects aggregate to make a parent object. The next section gives an example of recognition using the KARE system.
4.2 KARE: The Reverse Engineering Environment

KARE uses an extended granularity formalism as described above for representing programming plans. It is implemented in Common LISP. In addition, KARE uses two major tools to implement two of its components. The first tool, AROMA [18], is a knowledge engineering tool. The second tool, REFINER[69], parses C source code and generates corresponding abstract syntax tree. These tools are described below. A screen shot of the KARE environment is shown in Figure 4.15.

4.2.1 KARE Interface

As shown in Figure 4.15, the KARE screen is divided into five main areas. The top-left corner pane is used to display the list of items currently on the global agenda. The pane immediately below shows the local agenda of the current global agenda item. The middle pane is used to display the source code. This is useful for the user to browse the recognition and also to attach the context for an object. The user can click on an instance of an object that has been recognised to get KARE to highlight the code corresponding to the recognised instance. The two panes on the extreme right display a list of S-objects and a list of observer objects, respectively. Finally, there are two panes at the bottom. The left pane at the bottom is a control pane that displays the status of various user-controlled flags such as single-stepping (enabled or disabled) and thresholding (enabled or disabled). The user can change the status of these flags any time to affect the recognition process. The right pane at the bottom is used to display a log of status messages of the recognition process such as which object is being recognised, which cluster is being expanded, and when a recognition is being propagated.

Almost all the objects displayed on the screen are mouse-sensitive and allow various actions to be performed with various gestures of mouse-events combined with keyboard events. Items on the agenda can be moved around by dragging and dropping them from one position to another position in the agenda. Alternatively, the user can place an item on the top of the agenda by simply clicking on it with
Figure 4.15: A snapshot of the interface to KARE environment.
the left mouse button. By default, the order in which items from an agenda are attempted is determined by the status of the items. However, if a user forces an item to be at the top of the agenda, KARE will simply take the top item from the agenda and recognition is attempted on the item and KARE will not try to determine priority from the status of agenda items. The user can obtain a description of any object displayed on the screen in a pop-up window by clicking the middle mouse button.

4.2.2 Creating Granularity Hierarchies in AROMA

AROMA [18] is a tool which allows creation of granularity hierarchies. Using the AROMA's graphical interface, a knowledge engineer can create, test, and manage granularity hierarchies for various clichés. The original AROMA has been extended to handle various new types of constraints, attribute and attribute translations. With the modified AROMA, a knowledge engineer can create the topology of the hierarchy, various clusters, and other constraints. He/she can also specify context modifier functions and attach them to the clusters. Figure 4.16 shows a screen shot of the AROMA hierarchy engineering environment.

For more details on AROMA itself, please consult the AROMA project documentation [18]. For the purpose of the rest of this section, we say that cliché hierarchies can be engineered using AROMA. This knowledge engineering task is separate from the main task for which the hierarchies are used, i.e., reverse engineering, and can be done separately long before the reverse engineering task is undertaken (probably by some other person). Some of the hierarchies are specific to an application domain, and some of the hierarchies are domain/application independent. See section 6.1.1.3 for a more detailed discussion about knowledge engineering.

Creating good granularity hierarchies is not an easy task. There are many things one needs to worry about while creating these hierarchies. There are four primary steps a knowledge engineer needs to perform for creating a cliché. These steps are:

- Creating the topology of a granularity hierarchy,
• Attaching attributes to objects,

• Encoding constraints such as constraints for clusters, and constraints for attributes, and

• Encoding context modifiers and observer functions.

The most crucial part of creating a granularity hierarchy is encoding the context modifier functions and observer functions. There is a tradeoff between how much work is done by CMs and how much work is done by the recognition engine in terms of satisfying various constraints. If CMs are encoded in such a way that the context is reduced before the observer is recognised, then KARE generates fewer of spurious instances which are to be discarded later while satisfying constraints. On the other hand, if CMs are simpler, then there are more instance combinations of child instances that will be tried. There are no hard and fast rules about creating granularity hierarchies. The best way of learning about creating them would be to try creating a few hierarchies. There are some guidelines in the AROMA documentation [18] that provide some hints for beginners.

From our experience, we can say that with some practice and experience, a knowledge engineer who is knowledgeable about the granularity formalism and KARE can create a good cliché in a few weeks. It is not necessary to create these hierarchies every time KARE is used. Many of the cliché hierarchies that are generic can be created by an expert knowledge engineer in advance which can be used by a typical software engineer. If a software engineer wants to specialise some of these clichés or create some clichés specific to the application and/or the reverse engineering task at hand, then he/she needs to learn how to create the hierarchies. As the third experiment in the next chapter demonstrates, even a small, simple cliché that is created in a day, can be useful for extracting useful patterns from the code.
4.2.3 REFINE and KARE

KARE is currently tuned to reverse engineer C code. KARE does not directly work on raw code. Instead, KARE works on an abstract syntax tree (AST) representation of a C program with annotated cross reference information. The AST representation has an advantage in that the structure of the code in terms of syntactic elements is explicitly represented in the form a tree. It will be easier to process these syntax trees, find specific nodes of interest, find relationships among various nodes by traversing the syntax trees.

![Diagram of KARE and REFINE relationship](image)

Figure 4.17: KARE and REFINE relationship with each other.

KARE uses REFINE (from Reasoning Systems. Inc. [69]) to convert C programs into abstract syntax trees. The parser in REFINE takes a set of C code files, parses them into ASTs, and stores them into an in-memory database. ASTs generated by REFINE are not directly usable by KARE. Unlike KARE, REFINE uses a specialised object system to represent the objects in AST. KARE uses the standard CLOS object system to define its hierarchy and context objects. Hence we need to convert the AST object-base from REFINE into the AST object-base of KARE.
Figure 4.17 shows a schematic diagram of how REFINE and KARE are related. The REFINE C parser converts raw C code into an AST, and then the translator program translates this AST object-base into KARE’s AST object-base which is saved into a file. The AST Code Object Base is also stored as Common LISP code which when loaded into a running LISP system will create the ASTs in memory. As shown in the figure, the REFINE world and the KARE world do not have any dynamic interaction. The dashed line separating these two worlds indicates this fact. All the work in the REFINE world is done off-line in batch mode. From KARE’s point of view, the existence of REFINE is immaterial, as along as there is a way to obtain the AST version of the source C code in the form of Code Object Base (COB).

4.2.4 A Recognition Example

In this section we present a simple example that illustrates many aspects of KARE. The next chapter (Chapter 5) gives various experiences using KARE on a real-world software system.

As discussed earlier, a parser generated using the REFINE system is used for generating syntax trees. These trees are stored in a database in a form that can be used by KARE. Since this processing step does not require the user’s intervention, it can be done off-line so that the user of KARE need not wait while the parser is generating syntax trees. Figure 4.18 shows a fragment of a C program relevant to our example. This program segment is taken from a real-world system, the well-known WWW browser NCSA Mosaic.

After the ASTs are generated, the first step a user (the software reverse engineer) must take is to define a task by selecting the Define Task menu and giving a name to the task. KARE allows a user to maintain many tasks at the same time. Once tasks have been defined, the user has to choose one of the tasks to work on. The user will be prompted for the name of the database file in which the compiled C code (i.e., AST object base) is stored. The next step is to initialise the agenda.

83
This can be done in one of two ways. Either the user can add items to the agenda by explicitly selecting objects from the list of objects displayed, or she/he can let the system initialise the agenda using a set of heuristics. Heuristics can be used to encode any guidance a user may want to give to the recognition system. The default heuristic the system uses is to find all the objects that have direct observers attached to them and put them on the agenda.

Once the agenda is initialised, the user can start the recognition with the Start Recognition menu item. This will begin the process of searching for the clichés placed on the agenda in the source code.

The agenda is displayed in a separate mouse-sensitive window (See Figure 4.15). The user can rearrange the agenda by simply clicking and dragging various items into different positions in that window. Also displayed in another window is a list of S-objects. By clicking on these objects, they can be placed on the agenda with a default global context as the scope.

Selecting items from the top-level agenda is one of the important decisions that has to be made during a recognition cycle. We encode various heuristic rules that help determine priority values for these items. Also, a user can override the priority by physically changing the order of the items in the agenda. After each cycle, priorities are normalised and the agenda is sorted according to their priority values with the highest priority item being on the top. The recognition algorithm always selects the top-most element of the agenda.

For example, let us consider a specific reverse engineering task of understanding how NCSA Mosaic stores the history of Web addresses visited during a Web session. Initially, the software engineer does not know the details of how Mosaic is handling history. As a first step, the software engineer can look at the set of source code files and isolate a possible subset of files that might contain code related to handling the user history. This is a first attempt by the software engineer to reduce the search space. She/he can find out that the file globalhist.c is one such file. Now the job is to find out what kind of clichés the file globalhist.c might define. Since the person doing the reverse engineering is expected to be an experienced software engineer,
```c
#define HASH_TABLE_SIZE 200
typedef struct entry
{
  char *url;
  cached_data *cached_data;
  struct entry *next;
} entry;
typedef struct bucket
{
  entry *head;
  int count;
} bucket;
static bucket hash_table[HASH_TABLE_SIZE];
static int hash_url (char *url)
{
  int len, i, val;
  if (!url)
    return 0;
  len = strlen (url);
  val = 0;
  for (i = 0; i < 10; i++)
    val += url[(i * val + 7) % len];
  return val % HASH_TABLE_SIZE;
}
static void dump_bucket_counts (void)
{
  int i;
  for (i = 0; i < HASH_TABLE_SIZE; i++)
    fprintf (stdout, "Bucket %03d, count %03d\n", i, hash_table[i].count);
  return;
}
static void add_url_to_bucket (int buck, char *url)
{
  bucket *bkt = &hash_table[buck];
  entry *l = (entry *)malloc (sizeof (entry));
  l->url = strdup (url);
  l->cached_data = NULL;
  l->next = NULL;
  if (bkt->head == NULL)
    bkt->head = l;
  else
    {
      l->next = bkt->head;
      bkt->head = l;
    }
  bkt->count += 1;
}
```

Figure 4.18: A part of the C source code in Mosaic.
she/he should know the possible aggregate data-structures one can potentially use for such a purpose and can select a list of such data-structures and place them on the agenda. In our example, the software engineer selects the clichés LINKED-LIST, and HASH-TABLE and places them on the task agenda of the current task.

```
HASH-TABLE
  DATA
    ARRAY
    MAX-SIZE
      HASH_TABLE_SIZE
      bucket hash_table[...]
    HASH-FUNCTION
      int hash_url(char *url)
    BUCKET
      typedef struct bucket
        { 
          ....
        }
    ENTRY
      typedef struct entry
        { 
          ....
          struct entry *next
        }
  OPERATIONS
    ADD-ENTRY
    INITIALIZE
    INSERT-LIST
    SENTINEL-LOOP
    LOCATE
      l = (entry *)malloc(...)
    CREATE
      l->next = bkt->head
      bkt->head = l
    ADJUST-LINKS
```

Figure 4.19: An instance of cliché HASH-TABLE underlying the code implementing user history.

Once the current task agenda is initialised, the recognition engine then starts the recognition process on the code associated with the task by following the algorithms described in Section 4.1.3. The recognition process stops either when all the agenda items are recognised, or some parts of the agenda items are recognised and the recognition process cannot proceed further. The latter case happens when the code implements the clichés in ways that are not captured in the granularity representation of these clichés. In any case, the software engineer now can browse through the result of the recognition. This will allow him/her to find out which parts of the cliché are implemented by which parts of the program. Figure 4.18 shows the
source code in our example and Figure 4.19 the corresponding cliché hierarchy.

If the recognition process stops (or if the software engineer decides to interrupts it), the user can browse through the result of the recognition process (potentially partial) using the browse menu option to find out what parts of the cliché are missing and are yet to be recognised. While browsing, the parts of the code corresponding to the parts of the cliché that are recognised are highlighted in the code window as shown in Figure 4.15. In a situation where the software engineer is browsing the partial recognition, she/he can complete her/his understanding by using personal knowledge of clichés and code. Alternatively, she/he can fill in the next step of recognition by instantiating some of the parts by hand and attaching them to the partial recognition and then restarting the recognition process. The agenda-based recognition tool and the granularity-based representation allows such natural co-operation between human and system.

4.3 How KARE Can be Used

In the previous section we described KARE, a reverse engineering environment which provides recognition engine that can recognise clichés in C code. In this section we present a brief description of typical situations where KARE can be useful.

KARE is a cliché recognition system that can find instances of cliché’s in a program code. Therefore, for using KARE to perform reverse engineering, any reverse engineering task has to be formulated as a problem of finding clichés. Cognitive studies of program understanding by humans have revealed that people understand a software system by first forming hypotheses about the code and then looking for evidence to prove or disprove the hypotheses [43]. This process fits well with the working of KARE. As a part of the reverse engineering process, a software engineer first forms an hypothesis for existence of cliché’s. He/she then uses KARE to find the instances of cliché’s in the part of the code for which he/she formed hypothesis. Thus, a cliché recognition tool such as KARE can be used as one of the generic tool in the over all process of program understanding and reverse engineering.
Recognising programming concepts and patterns in terms of clichés is only one of the steps in any reverse engineering and maintenance activity. Here we list some examples of typical reverse engineering situations where KARE can be used effectively.

- How does Mosaic handle the history of urls (web site addresses) visited by the user?

This type of questions can arise when a software engineer is trying to understand how Mosaic keep its history of user’s visits to Web sites. It is needed, for example, if a software engineer is trying to modify the Mosaic history handling mechanism so that it can also keeps track of additional information about users’ actions, such as date and time of previous visits and frequency of visits to each site.

- Where are all the Linked Lists (or any similar data structures) implemented in this system?

This type of question arise in situations, for example, where a software engineer found out that there is a generic linked-list implementation in a library that can be used in place of the one implemented in Mosaic. If he/she can locate where are all the linked-list clichés are implemented, this can help in replacing the code with a call to appropriate library functions.

- Where in this file is the code that implements searching operation?

This type of questions can arise, for example, in situations where a software engineer is assigned a task of optimising search and retrieval operations to reduce the response time. If he/she can find out what type of search operations are implemented, he/she can think of ways to make changes to the code to increase the efficiency. One such change can be to replace a linear search algorithm by a binary search. This type of change sometimes also require appropriate changes to the related data structure declaration.
Are there any date manipulation clichés in this module?

This type of questions can arise in situations such as the well-known year 2000 problem where a software engineer needs to find out if a piece of software can work properly when the year changes form 1999 to 2000. It should be noted here that the system such as KARE does not identify all the year 2000 problems as such. But if the system can help locate the places where the date and time operations are performed, it can help a software engineer to find out if any of these operations have some hidden assumptions that make the operation not work correctly when the year changes to the 21st century.

In Section 5.3 we will describe in detail how such questions can actually be handled using KARE. Here they are listed to give a general idea of what sort of questions KARE can help answer about a software system.
Chapter 5

Using KARE

In the last chapter, we presented a detailed description of program cliché recognition in KARE. In this chapter, we describe our experience in using KARE on a real-world software system.

5.1 Experimental Setup

5.1.1 Target Software System

The software system that we used for this experiment is a once popular Web browser named NCSA Mosaic Version 2.6, from the National Centre for Supercomputing Applications, University of Illinois at Urbana-Champaign. There are many reasons why Mosaic has been selected for the purpose. It was one of the earliest browsers for the Web. It has been used by many people and its functionality is known and understood by users. Mosaic has been under development/maintenance over many years. Its development started from the early days of the Web. Mosaic also is sufficiently large and qualifies as a real-world system. Finally, the source code of Mosaic is freely available.

There are some problems when using a real-world system for the purpose of evaluating a reverse engineering environment like KARE. For example, we cannot precisely control various properties of the system, such as the relative composition of various types of statements in the system, and the number of possible clichés occurring in the system. To counteract these problems, Quilici [89] in his experiments
used a synthetic program generated randomly so that various parameters could be controlled precisely. But we prefer to work with a real system, because we feel that KARE's complete range of capabilities (including the human-in-the-loop aspects) can only be effectively demonstrated on a real software system.

Mosaic is a C program consisting of approximately 130,000 lines of C code. The system is divided into 10 modules that are placed in separate directories. They are as follows:

- src:
- libdtm:
- libXmx:
- libwww2:
- libnut::
- libhtmlw:
- libnet:
- auth:
- Makefiles:
- platform-configs:

From the names of these directories, it can be inferred easily that the two directories, Makefiles and platform-configs are not part of the core functionality. These files are used for specifying platform specific information that helps in building the system for each of the various hardware/software combinations. The module auth consists of a set of shell programs for authenticating the distribution, and thus is not part of the core of the software system. The actual functionality of the Mosaic system is, therefore, distributed among the seven remaining modules. The core of Mosaic is implemented in the module src. Other modules are implemented as
libraries, each one handling a specific functionality. We use primarily the module src for our purposes.

In the next sections we demonstrate the working of KARE on the Mosaic source code recognising the clichés. We focus on the issues of scalability and identifying how KARE reacts to changes in various parameters. In addition, a different target system other than Mosaic (namely, one of the popular mail readers PINE) is also studied. This is to demonstrate that KARE is in fact capable of working on wide variety of software written in C and much of the knowledge-base is transferable across different applications.

5.1.2 Knowledge-Base

The knowledge-base (i.e., the cliché library) we have chosen for conducting the experiment consists of three large clichés:

- Linked List: A typical dynamic data structure for storing elements that are varying in number and length.
- Date and Time operations: Data structures and routines that perform operations on dates and times.

Appendix A gives a simplified listing of granularity objects for these clichés.

5.1.3 Experimental Methodology

There were three different experiments conducted, each with its own specific goals. The first experiment was conducted to investigate primarily the scalability aspects of KARE using the Linked List cliché. Specifically, it was designed to study the performance of KARE with increasing sizes of source code. This experiment also included a study of how various human interventions can affect scalability. There are two different aspects of scalability. When the code size gets large, any recognition
algorithm should be efficient in the sense it is able to naturally scale up to perform
in a reasonable amount of time. Another aspect of scalability is the scalability
of the knowledge-base. In order to be able to use any tool in large systems, the
knowledge representation scheme should be expressive enough to cover a wide variety
of situations. There should be a way of increasing the coverage of the knowledge-base
as needed.

The second experiment is concerned with transferability: i.e., it was intended to
see whether the performance can be maintained on a second cliché of a similar type
to the first, the Searching cliché.

The third experiment is designed to test the versatility of granularity and KARE
for a markedly different type of cliché, namely the Date and Time cliché.

It should be noted that these experiments are not meant to be viewed as tightly
controlled experiments but rather viewed as an exercise to demonstrate the capabili-
ties of the approach and the abilities of the prototype system KARE. It is, essentially,
a proof-of-concept experiment. The rest of this chapter presents experiments that
are designed to investigate these aspects.

5.1.4 The Human-in-the-Loop

The human software engineer helps the KARE system in many ways. The following
are some of the activities the software engineer can perform:

- Agenda management: He/she can affect the recognition process by changing
  the recognition order of various items on the agenda. That is, the software
  engineer can change the recognition sequence to promote certain parts of the
  cliché hierarchy.

- Context Management: He/she can affect the regions of the source code in
  which a search can happen. That is, the software engineer can specify the
  context (the scope of the code) of the cliché to be recognised in terms of the
  files that should be considered for recognition.
• Cliché management: He/she can affect the types of clichés that need to be searched for. That is, the software engineer can specify the parts of the cliché library that are of interest to his/her task.

• Assist in the recognition: He/she can force a recognition by manually satisfying some constraints. He/she can also prune some partially-recognised instances of an object. That is, the software engineer can remove some of the parts of partially recognised clichés that he/she thinks are not useful for his/her task.

All these types of human intervention can help in increasing the efficiency of KARE’s recognition, thus allowing it to work on larger source codes. KARE provides various mechanisms by which the human user can perform the actions listed above. The following section describes some of these mechanisms.

5.1.5 Controlling the Experiments

In this section, we describe various factors that affect the performance of the KARE system’s recognition engine. Performance experiments are done by selectively controlling these factors by using the mechanisms that allow the types of human interventions discussed above, as well as by enabling or disabling the context modifiers and varying the code size.

5.1.5.1 Human Intervention

We have conducted the experiments by selectively varying levels of user intervention. The specific factors that control human intervention are:

• File-Level scoping (enabled or disabled). This is achieved by selecting the set of files which are relevant for the cliché that is being searched. This affects the context of the search performed by KARE.

• Agenda reordering (enabled or disabled). The user can change the order of the items on the agenda by simply clicking on the agenda item which he/she
wants to be attempted next. In the absence of the user's action for selecting an item, KARE will use its heuristics to select the item from the agenda.

- Pruning some recognition (enabled or disabled). This is also called thresholding. This is achieved by assigning a threshold level for each agenda item. Whenever the number of partially or fully recognised instances for this item exceeds this threshold, KARE stops and the human user can browse through the results of the recognition and eliminate (or prune) some of the instances that he/she thinks are not relevant.

- Forcing a recognition (enabled or disabled). This type of human intervention can be used when the system fails to find any instances of an item. By forcing an item as recognised, he/she instantiates the attributes associated with this item. KARE then proceeds as if it had been recognised by KARE.

We conducted the experiments with combinations of each of the first three features listed above. The last feature, i.e., the human forcing a recognition is not used in the experiment. This feature is mostly used for making the recognition process succeed when the system alone cannot find some parts of a cliché. This feature does not directly affect the efficiency of the system. We also investigated the experiment with all the human interventions enabled at the same time.

5.1.5.2 Context Modifiers

Context management in KARE is a very powerful mechanism that helps in reducing the search done by the recognition engine. As described in Section 4.1.1.1, context modification is performed by KARE to reduce the scope of search for remaining objects to be recognised using information gained from the instances of objects that have already recognised. We conducted the experiments by enabling and disabling this feature to see how the presence of context modifiers (CMs) actually affect the computation. This feature is orthogonal to the factors listed in the previous section, and can be changed independently of them.
5.1.5.3 Code Size

As described in Section 5.1.1, the target software system used in our experiment (NCSA Mosaic) is a C program consisting of approximately 130,000 lines of C code. We considered three different subsets of src files for the purpose of the experiment to see how the various mechanisms of KARE help in the scalability of the system. They are as follows:

- **Size 1**: 1320 lines, consisting of one file:
  - globalhist.c

- **Size 2**: 5300 lines, consisting of five files:
  - globalhist.c
  - hotlist.c
  - hotfile.c
  - grpan.c
  - hotlist.c

- **Size 3**: 10909 lines, consisting of ten files:
  - src/globalhist.c
  - grpan.c
  - history.c
  - hotfile.c
  - hotlist.c
  - prefs.c
  - newsrc.c
  - pan.c
  - annotate.c
  - main.c
5.1.6 Cliché Engineering for the Experiments

As described in the previous chapter (Section 4.2.2), granularity hierarchies are created using the AROMA knowledge engineering tool. The three clichés created using AROMA for use in the experiments varied in size, complexity and nature.

The Linked List cliché and the Searching cliché consists of about 60 objects. The Linked List cliché was engineered over a couple of months time and the cliché went through considerable amount of tuning. The development of KARE itself and the engineering of the Linked List cliché went in parallel.

Creating the Searching cliché took a little over a month even though it was of the size similar to that of the Linked List cliché. This was possible because the experience gained during the creation of the first cliché helped in the creation of the second cliché.

The third cliché, namely the Time and Date cliché is relatively a small cliché, consisting of about 10 objects. The first two clichés are related to typical data structures and algorithms used in programming, whereas the third cliché is more conceptual in nature. This cliché was created in a couple of days.

5.2 The Experiments

The experimental results are measured in terms of the number of instance combinations that are tried by KARE in recognising a cliché. The combinations are counted as follows: For an observer object, the number of instance combinations is the number of potential syntactic objects from the source code that are evaluated for finding instances of the object. For an S-object, the calculation of instance combinations is a bit elaborate. Let us assume that an S-object $S$ has $n$ children, and each of these $C_i (i = 1, \ldots, n)$ children has $m_i (i = 1, \ldots, n)$ instances respectively. Further, assume that the K-cluster constraint on the object $S$ specifies that each instance of object $S$ needs $k_i$ instances of child $C_i$ respectively. Then the number of instance combinations for the object $S$ is given by the following formula:
\[ I(S) = C_k^{m_1} \times C_k^{m_2} \times C_k^{m_3} \times \cdots \times C_k^{m_n} - X \]

Where \( X \) is the number of instances ruled out by the propagation of constraints and of the attribute values by the attribute translation rules. In other words, the number of instance combinations for an object is all possible combination the instances of its children less the number of combinations eliminated through constraints and attribute translation rules.

In the following sections, we describe three different experiments, each with its specific goals.

5.2.1 Experiment 1: Scalability and Human-in-the-Loop

Experiment 1 used the Linked List cliché. The cliché is shown in Appendix A.

The goals for this experiment are:

- to study the effect of various human interventions on cliché recognition,
- to study the scalability of KARE for large programs, and
- the interaction between these two.

By running the cliché recognition algorithm on the following three code sizes, we found the following instances of the Linked List cliché.

- Code size 1320 has one file and has one instance of a linked list, which was correctly recognised.
- Code size 5300 has five files in all and also has only one linked list which was correctly recognised.
- Code size 10909 has two different linked lists, a linked list of user visited web sites and a linked list of Mosaic windows. It also has some parts of a third linked list which was partially recognised. Some parts of the third linked list are defined in another file which was not part of the analysed code.
<table>
<thead>
<tr>
<th>Code Size</th>
<th>No Intervention (A)</th>
<th>File Scope (B)</th>
<th>Threshold (C)</th>
<th>Reordering (D)</th>
<th>All Interventions (E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM On</td>
<td>CM Off</td>
<td>CM On</td>
<td>CM Off</td>
<td>CM On</td>
<td>CM Off</td>
</tr>
<tr>
<td>1320</td>
<td>174</td>
<td>2290</td>
<td>174</td>
<td>2290</td>
<td>121</td>
</tr>
<tr>
<td>5300</td>
<td>288</td>
<td>48782</td>
<td>182</td>
<td>5480</td>
<td>268</td>
</tr>
<tr>
<td>10909</td>
<td>9551</td>
<td>210212</td>
<td>1127</td>
<td>173107</td>
<td>9202</td>
</tr>
</tbody>
</table>

Table 5.1: Effect of various human interventions for using KARE on the linked list cliché.
Table 5.1, shows the number of instance combinations (computed using the above formula) generated by KARE for the three different code sizes. The next section describes these results in detail.

5.2.1.1 No Intervention

The first column of Table 5.1 (marked with "A") shows the result of running KARE without any human interventions. With CM disabled, this is effectively equivalent to a brute force search. As can be seen from the table, the number of combinations increases dramatically as the code size gets larger. For the code size 1320, the combinations are 2290. For the code size of 10909 (for a tenfold increase), the combinations become 210212 (a 100 fold increase). With context modifiers enabled, the number of combinations decreases considerably, from 210212 to 9551 for the code size of 10909. This illustrates the capabilities of CMs as an effective mechanism to reduce the search space.

5.2.1.2 File Scoping

Column 2 of Table 5.1 (marked with "B") shows results of running KARE using file-level scoping (FS) enabled. For the code size 1320, there was only one file in the source code. Hence, there is no difference in the combinations with or without FS, which are 174 when context modifiers (CM) are enabled, and 2290 when CMs are disabled, respectively. For code size 5300, the FS was used to specify only 3 files out of 5 available in the source code. The number of combinations are reduced from 288 to 182 when CMs are enabled. But the reduction is an order of magnitude higher when the CMs are disabled, namely from 48782 to 5480. For code size of 10909, the reduction is more when CMs are enabled. But when CMs are disabled, the reduction is much less compared to the case with code size 5300. The main reason for this is the nature of file scoping performed in this case. In this case, there were 10 files in all but only 3 files out of 10 were pruned, so the remaining 7 files were considered. This pruning is done based on the file names. When the code size increases, it is hard to prune any file without having a specific goal in mind. If
there is a specific reverse engineering goal at hand, then a knowledgeable software reverse engineer can use his/her goals to identify the files that are not needed for the problem at hand.

![Graph showing the effect of file scoping with CM.](image)

**Figure 5.1: Effect of file scoping with CM.**

Figures 5.1 and 5.2 shows the plot of the data taken from column 2 of Table 5.1. In these graphs, (and also all other graphs in this chapter) the horizontal axis shows the size of the code in which the recognition is done, and the vertical axis shows the number of combinations (candidate instance hierarchies) to be tried for recognition, unless specified otherwise.

### 5.2.1.3 Thresholding

Column 3 of Table 5.1 (marked “C”) shows the results of the experiment when thresholding is enabled. Figures 5.3 and 5.4 show the plot of the data taken from the table. It can be observed from the table that the effect of thresholding is very marginal when applied in isolation. The effect seems to be more noticeable (210212
without thresholding and 182911 with thresholding) in the absence of CMs. When CMs are enabled, the effect of thresholding is marginal (from 9551 to 9209).

The following are some of the heuristics used in pruning the extraneous partially recognised objects, using *thresholding* intervention. It may be noted that much of the pruning is highly situation dependent. It depends not only on the object whose instances are being pruned, but also on the task at hand. So, the following are not really guidelines for any general pruning rules, but are examples of the kind of pruning an expert software reverse engineer can do in this particular situation. However, they do serve to illustrate the potential effectiveness of pruning as a heuristic.

Here are the pruning heuristics:

- If the declaration of a variable is not in the specified file scope, then prune it.
- In recognising the code that tests for *empty linked list*, prune the tests that are comparing integers, not null pointer values.
Figure 5.3: Effect of thresholding with CM

Figure 5.4: Effect of thresholding without CM
• In recognising the head of a linked list, prune a candidate pointer declaration for a linked list head whose name is current_node. It is highly likely that this variable is used for keeping track of the "current" node (according to some notion of current). It is unlikely that this pointer serves as a head of a linked list, even though it is of the right type (i.e., a pointer to a linkable node).

• The earlier the better: a principle one can follow to prune effectively is the earlier the better. That is, the gain is more when one prunes unwanted/unneeded parts as early as possible in the recognition process, thus preventing combinatorial explosion before it grows out of hand.

On the whole, pruning effectively without removing some of the useful instances requires experience in using KARE and also the knowledge of the application domain and its functionality. Also, pruning can be done more effectively when there is a specific reverse-engineering goal in mind.

These heuristics illustrate how a software reverse engineer can bring his/her own intuitions about programming to bear upon the search for clichés. These intuitions would be very hard to capture as constraints in the hierarchy so that KARE can use them automatically. But with the human-in-the-loop, they can be brought naturally into play.

5.2.1.4 Reordering

As explained in earlier sections, the reordering mechanism allows the user to change the order in which the items on the agenda are recognised. By default there is a pre-specified local recognition order in each cluster of an object. But the exact order in which items on the agenda are recognised depends on many factors which are dynamic and can change during the recognition process. Hence, KARE does not compute a total linear order for recognition in advance; rather it selects each type of item from the agenda based on some heuristics. If the user observes that the system is trying to recognise an object of a cluster which he/she thinks is unnecessary for the objective, or the system is trying to propagate some recognition to other objects
which are not needed for the task, he/she can affect this order by forcing KARE to select a specific item for the next recognition cycle. If the user specifies an item to be worked on next, it will override the system’s choice for the next item to be recognised.

![Figure 5.5: Effect of reordering with CM](image)

Column 4 of Table 5.1 (marked "D") shows the results of using some reordering interventions. Figures 5.5 and 5.6 show the plot of graphs with the data from the table. It may be noted that not all types of reorderings can result in a reduction of instance combinations. Sometimes reordering might cause the number of combinations to increase as well. So the software engineer should have a good knowledge of the topology of the granularity hierarchy and potential occurrences of instances to be able to use this technique usefully. From the data in the table we can see that the decrease in the number of combinations is marginal for the first two code sizes, and is noticeable for the third code size.

The following are some of the heuristics for reordering the agenda:
- Declaration-related parts should be attempted before operation-related parts. This will help narrow the context for the operations-related parts.

- Instances that result in fewer combinations should be attempted first. This will help in reducing the combinatorial explosion problem before it starts.

- Propagating along abstractions may be attempted first, before propagating along aggregations, unless the specific task at hand suggests otherwise.

It should be noted that there is no "best" order that can be specified for a cliché. However, it is conceivable that a knowledge engineer could test the cliché on various typical applications and typical tasks and come up with a predetermined recognition order, or more specific guidelines for each kind of application or task. A software engineer performing a reverse engineering task would then use these results as a starting point for his/her own specific task.

Reordering not only affect the recognition order, but also avoids some of the
branches of the granularity hierarchy that are not needed. Even if the instance combinations are not affected by some reordering, there may be an indirect effect in that instances of parts of a cliché may be found faster by avoiding the recognition of other clichés that are not related to the current objective.

5.2.1.5 All Interventions

We also conducted an experiment in combining all types of user interventions described above. As shown in column 5 of Table 5.1 (marked "E") (graphed in Figure 5.7) combining these interventions can affect performance to a great extent. Even in the case of no CMs, the number of combinations is greatly reduced for all three code sizes. The best case scenario can be achieved by using all possible user interventions and also CMs. For this case, the number of combinations for the code size 10909 is only 1096. This is a good sign and suggests evidence that various features of granularity along with the human user's intervention can help in the scalability of KARE to large software systems.

From Table 5.1 and Figure 5.7, it can be observed that there is some interaction among different types of human interventions. For example, the effect of all three interventions used together is much more than any of them used individually.

5.2.1.6 General Observations and Implications

From data presented in the tables in the previous sections we can state the following general observations about system performance:

- Without any user interventions and context modifier constraints, the number of combinations increases exponentially with the code size.

- Context Modifiers have a large impact on performance. This is understandable as CMs directly reduce the search space by modifying the domain of possible places to search.

- User intervention is most effective when combined with context modifiers.
Figure 5.7: Effect of all interventions on LINKED-LIST cliché

- Without CMs, file scoping and pruning have more or less the same impact on the combinations when used independently.

- Reordering used alone is not effective in the absence of context modifiers. For example, for the code size 10909, the number of combinations is 174929 with reordering enabled. If the reordering is not done, the number of combinations is a little more at 210212, but it is not significantly higher.

- Human intervention works. By using all types of human interventions, we can decrease the number of combinations for the code size 10909 from 210212 to 23740.

- Human intervention with context modifiers works even better. By using all types of interventions with context modifiers, we can effectively drastically reduce the combinations from 210212 to 1096 for the code size 10909.
It appears that the granularity-based approach, with appropriate context modifiers and human intervention can tame the combinatorial explosion or at least keep the growth manageable in the range of code sizes that would actually be used by a software engineer.

5.2.2 Experiment 2: Transferability

The second experiment was conducted on the same source code (i.e., NCSA Mosaic) as the first experiment, using the Searching cliché. Searching cliché can be found in Appendix A.2. Specific goals for this experiment are:

- to see whether the scalability results of the first experiment would be similar for another cliché,

- to demonstrate the ability of the KARE’s recognition to work on various clichés.

As in the previous experiment, this experiment was also conducted on three different code sizes. For each code size, three recognition styles were tried, each with a different amount of intervention. They are:

- no human intervention and no context modification,

- no human intervention but with context modification, and

- all human interventions and with context modification.

<table>
<thead>
<tr>
<th>Code Size</th>
<th>Human in the loop with CM</th>
<th>No human in the loop with CM</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>1320</td>
<td>1843</td>
<td>6745</td>
<td>10754</td>
</tr>
<tr>
<td>5300</td>
<td>2849</td>
<td>18905</td>
<td>40586</td>
</tr>
<tr>
<td>10909</td>
<td>4516</td>
<td>255849</td>
<td>1178512</td>
</tr>
</tbody>
</table>

Table 5.2: Effect of human intervention and CMs for the searching cliché
Figure 5.8: Effect of human intervention and CMs for the Searching Cliché.

Table 5.2 (graphed in Figure 5.8) shows the results of the experiment. The results shown in this graph confirm that the human interventions can help reduce the combinations that can help KARE to work on a larger size of code than KARE working alone. Similar growth curves were obtained as in the first experiment for each recognition style, thus confirming the conclusion that KARE with appropriate context modifier and human interventions can constrain the combinatorial explosion.

One important observation made from this experiment is that, if a cliché has more structure to it, then it will take less effort to find it. For example, in recognising the Searching cliché, it took more time for finding instances of the array search cliché than instances of the hash-table search cliché. This is because in C any pointer variable is a potential array since C allows any pointer to any object to be used as an array for normal array access with indexing, whereas a typical implementation of a hash-table needs to have structures that can act as buckets that can hold overflow chains.
5.2.3 Experiment 3: Versatility and Generality

The first two experiments were conducted on a cliché a data-structure related. For the last experiment we chose an entirely a different type of cliché, the cliché of date and time computation. The cliché is chosen:

- to demonstrate the ability of KARE to work on various types of clichés,
- to demonstrate the generality of KARE to emulate other, simpler approaches for recognising program patterns, and
- to see how KARE works on a different target system, namely the UNIX mail program PINE.

Appendix A.3 shows the Date and Time cliché. The main differences between the this cliché and the clichés from experiment 1 and 2 are summarised below.

- Nature of the cliché: As explained earlier, the first two clichés are based on typical data structures and related algorithms used by programmers. But the Date and Time cliché is more conceptual in nature without a lot of structural information.

- Amount of tuning: Clichés for the first two experiments were engineered rather comprehensively which took a lot of time and effort. The Linked List cliché was designed over a few months and iterated through several cycles of modifications and redesign of various constraints as KARE was being built. The Searching cliché took three weeks for designing and programming the constraints. In contrast, the Date and Time cliché was deliberately designed to be simpler and the design took two days. This has been done to demonstrate that even a not-so-perfect design of a cliché can still be useful and KARE can use it in a fruitful way.

- Size of the cliché: The first two clichés are bigger with about 50 S-objects in each. The Date and Time cliché is small and has only 10 objects in all.
• Nature of constraints: The Date and Time cliché does not have much structural information (aggregation or abstraction). Observers and modifiers are based on simple regular expressions that contain words like “day”, “month”, “year”, and “time”.

• Target system: The first two experiments have been conducted on the Mosaic source code whereas this experiment has been conducted using the source code of one of the popular user mail programs called PINE. PINE is a program that allows a user to read and process his/her electronic mail. It is 77000 lines of code divided into 39 different files. We have chosen a subset of five files from these 39 files. The subset consists of 8669 lines of code.

<table>
<thead>
<tr>
<th>Code Size</th>
<th>Intervention and CM</th>
<th>No Intervention or CM</th>
</tr>
</thead>
<tbody>
<tr>
<td>8669</td>
<td>1288</td>
<td>7838</td>
</tr>
</tbody>
</table>

Table 5.3: Date and time cliché: All interventions

We ran KARE on the part of PINE described above. At first, a total of 81 instances of date operations were found (without any constraints). This can be thought of as similar to a simple regular expression search tool like grep available on UNIX\(^1\). But with one simple constraint (that the day, month, and year of an instance should all be in the same function or in the same block), the number was reduced to five instances. Only one of the five operations on date was “spurious” and not interesting (the initialisation of the date structure with a value of -1).

One particular interesting operation was found on date, the kind of “surprise” that knowledge-based reverse engineering can help to discover. One of the date operations was recognised as date parsing, i.e., converting string to numerical values for each of the date components such as day, month, year. This operation has an interesting future problem that can be called as “mid-century” problem. This is

\(^{1}\)UNIX is a trademark of AT&T Bell laboratories
in contrast to many software systems that have a potential problem of transition to year 2000. While converting a string representation to integer, PINE makes an interesting decision. When it encounters a two digit year such as 96 or 25, PINE converts that into four digit by the following rules:

- If the two digit year is less than 50 such as 25, or 49, then PINE assumes that it is the 21st century and adds 2000 to the year. So the year 25 will become 2025, and the year 49 will become 2049.

- If the two digit year is more than 50 such as 63, or 98, then PINE assumes that it is the 20th century and adds 1900 to the year. So the year 63 becomes 1963 and the year 98 becomes 1998.

The above heuristics is called windowing. It is a typical "fix" for the year 2000 problem (Y2K) that has another bug in it. The above heuristics may work without any problems until the year 2049. But when it crosses into 2050, any two digit year will be assumed to be in 20th century.

5.2.4 Instance Combinations and Computation Time

In the previous sections we described experiments using the metric number of instance combinations that are tried by KARE as a measure of the amount of computation performed by KARE. It would be interesting and useful to see how the instance combinations relate to the actual time of computation. We collected actual time taken by KARE for trying out various instance combinations.

Table 5.4 (graphed in Figure 5.4) maps instance combinations to time in minutes taken by KARE. From the graph it can be observed that the number of instance combinations correlates with the computation time more or less linearly.

It was difficult to measure actual CPU time as the system spent various amounts of time for garbage collection. The amount of garbage collection increases as the instance combinations increase. Despite these limitations, the above data serves
<table>
<thead>
<tr>
<th>Instance combinations</th>
<th>Time in minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1843</td>
<td>2</td>
</tr>
<tr>
<td>6745</td>
<td>4.3</td>
</tr>
<tr>
<td>10745</td>
<td>9.2</td>
</tr>
<tr>
<td>16326</td>
<td>15.25</td>
</tr>
</tbody>
</table>

Table 5.4: Computation time for various instance combination.

Figure 5.9: Instance combinations vs time.
the purpose of providing some general idea of how instance combinations relate to computation time.

We can draw the following conclusions about KARE in light of the data about the computation time as well as the data from the experiments:

- KARE can attempt approximately 1000 combinations per minute not including swap time and garbage collection time. When the combinations are few the user-level time more or less coincides with the system time. But as the combinations increase, the user-level time is an order of magnitude more than the system time, since a lot more time is spent in garbage collection and swapping. In our experience, at some point, when the combinations go beyond 100000, more computer time is spent in swapping and garbage collection than in actual computation. This could be rectified by having much more main memory.

- From the data of experiment 1, it can be observed that with CMs and human intervention, the number of combinations is small and KARE can work in more or less real time. But even without human intervention with CMs enabled, KARE can still be used. Both of these options have a tradeoff between the amount of time spent in knowledge engineering of context modifiers and the amount of time spent by a software engineer in terms of intervention. If CMs are encoded properly then KARE can be used without human intervention. If there is not enough time for the knowledge engineer to create the CMs a software engineer can compensate for this by using his/her own knowledge of the cliché under consideration and by intervening in the recognition process. However, this is not always the case as the next point illustrates.

- From the data of experiment 2, it can be observed that even for the code size of 10909 lines, we need both CMs and user intervention to be able to use KARE in practice. CMs alone can reduce the number of combinations from almost one million to a quarter million. But even this amount of computation
is impractical as it can take many hours of user time. By using human intervention, the combinations can further be reduced to a few thousands, thus making KARE practical. The advantage of human intervention becomes more apparent as the number of potential instances of clichés in the code increases. Also, the combinatorial explosion becomes severe as the code size increases and we will definitely need both CMs and human intervention.

- With all the human interventions and CMs enabled, a typical software engineer can take less than an hour to find instances of a typical cliché.

5.3 Typical Situations for Using KARE

In Section 4.3, we listed some of the typical software maintenance situations where KARE can be used. In this section we elaborate on some of these situations. This section is organised around questions that might arise naturally in software maintenance. For each question, we describe how KARE's recognition tool can be used to achieve the software engineering objectives. The idea is to motivate how KARE is able to be deployed naturally and effectively by an expert software engineer seeking answers to how a piece of source code carries out particular functionalities of importance.

5.4 User's Time vs System's Time

One of the major features of the KARE approach is that human user assists the system in finding clichés. A note on the relative amount work of the system and the user is in order. There is always a tradeoff between the amount of work done by the user and the amount of work done by the system. Since the user in turn can be either a knowledge engineer or a software reverse engineer, there is also a tradeoff between the amount of work done by each of these users. If clichés are engineered well and refined to incorporate various types of heuristics for ordering and pruning, then there is a less need for the actual user performing the reverse engineering task
to intervene. On the other hand, if clichés are not engineered well, then there will be more work needed by the reverse engineer while using these clichés. In the extreme case where the clichés are not very well engineered and contains only observer level objects without much aggregation and abstraction in them, KARE essentially is equivalent of simple pattern search tools like UNIX grep as demonstrated by the experiment 3 described earlier in Section 5.2.3. Thus, the amount of intervention needed by the software engineer depends on how well the knowledge engineering was done, how much time he/she is prepared to wait for results, and how much work is required in interpreting the results.

- How does Mosaic handle the history of urls (web site addresses) visited by the user?

In order to find an answer to this question, a software engineer can select a group of files that might potentially have the code for handling user history (identified, perhaps, through external documentation). He/she can then select a list of typical storage representations for strings such as urls that can be accessed efficiently. Some potential storage representations are arrays, linked lists, and hash-tables. Arrays and linked lists require sequential search. Hash-tables might require a good combination of fast indexing into a bucket and sequential searching in an array. So the software engineer can select these potential clichés (from KARE's cliché library) and deploy the recognition engine. KARE will then find the instances of these storage structures and routines that operate on these structures. The user can browse through the recognised instances. The software engineer's knowledge of these storage clichés, plus his/her access to Mosaic documentation, can enable him/her to intervene/direct the recognition process using the various types of interventions discussed earlier.

- Where are all the linked lists implemented in this system?

This question can be answered by selecting the Linked List cliché and deploying KARE's recognition on all the files in a module of interest. KARE will display
all the instances of linked lists, which can then be examined by the software engineer.

- Are there any linked lists in this file?
  This question is essentially a special case of the above question where the scope of searching can be limited to one file. So the file scoping facility provided by KARE can be used to specify the context for the cliché to be searched, i.e., the Linked List cliché.

- Where in this file is the code that implements searching operations?
  This question can be answered by placing the Searching cliché on the agenda, providing a context with one file to the cliché, and then deploying KARE.

- Is there any recursive searching algorithm in this group of files?
  This question can be answered by placing the Recursive Search cliché (a specialisation along the abstraction dimension of the general Search cliché) on the agenda and running the recognition engine on this cliché with the group of files as its initial context.

- What interesting clichés are there in this file?
  This question can be answered by asking KARE to initialise the agenda with all the clichés of interest and running the recognition engine on each of them. File scoping is limited to one file to reduce the search space.

- Are there any date manipulation clichés in this module?
  This question can be answered by placing Date and Time cliché on the agenda and attaching a context with the relevant files in the module to the cliché.

- Where are all the comparison, and update operations on date and time performed in this file?
  This question can be answered by placing the Date and Time operations on the agenda and attaching the specified file as the context.
5.5 Comparison With Other Approaches

We have presented our KARE system for program recognition and some experimental evidence that demonstrates the scalability of KARE. In this section, we will look at some of the previous attempts at program understanding in comparison to the granularity-based program recognition used in KARE.

5.5.1 KARE vs. Johnson’s PROUST

PROUST [34, 33] uses three levels of “abstraction” to represent programming knowledge. KARE does not have any goal-level encoding. The notion of task in KARE plays a somewhat similar role. KARE’s task can encode various types of information about the task including any heuristics to gather initial agenda items, in case the user does not provide any specific goals. In the absence of any other information, KARE’s task consists of a global agenda containing some of the granularity objects representing the clichés the software engineer is trying to recognise. The goals here do not correspond to the goals of the programmer. Instead, they are the goals of the software engineer looking for the existence of any such goals in the code.

Like PROUST, KARE’s granularity-based representation has “multiple-levels”, but they do not necessarily correspond to particular levels involved while implementing the code. Instead, multiple levels of granularity capture the various natural grain sizes of clichés (along the two dimensions of aggregation and abstraction).

PROUST’s intention-based program understanding follows a top-down approach as it starts with the goals and subgoals generated from the problem-specification. KARE’s agenda-based approach is more flexible in the sense that it can do top-down as well as bottom-up recognition. A user can initialise the task agenda to provide top-level goals the system should look for. On the other hand, the user can let the system choose the plausible goals and try to find them in the code submitted by the user. The user can also delimit the scope of search to only a small portion of the granularity-hierarchy, or let the system choose all the possible objects from the hierarchy.
PROUST generates hypothesised plans or plausible plans that might be present in the code. This is done by selecting goals generated from the specification including any inferred goals. There are some heuristics that suggest a most promising order in which goals should be tried. This type of ordering is similar to the local recognition order control encoded in a cluster of granularity-based recognition in KARE. The plans that are hypothesised in PROUST are evaluated against the code. If there is more than one plan that matches the code, then the closest matching plan is selected. Any partial mismatches are attempted to be resolved by applying bug rules and transformation rules.

PROUST works very well on small size student programs in the context of ITS. But it is difficult to extend this approach for reverse engineering task where the system needs to handle large amounts of source code.

5.5.2 KARE vs. Wills’ Recognizer

Wills’ Recognizer [87] views program understanding as a parsing problem. Wills represents the plans as graph grammar rules and the code as flow-graphs and performs an equivalent of chart-parsing on these graphs (See Section 2.3.1.2 and [87]). In contrast, KARE uses source code directly to work on, instead of idealising the program as flow graphs. Of course, KARE does not work on the static linear text of the program as input. Instead, it transforms the linear text into a syntax tree using a parser. This step does not remove any information from the code. On the contrary, it provides useful additional links that help collect information such as call-graphs by traversing the syntax tree. A syntax tree allows easy traversal and search for a given node that satisfies a given syntactic condition.

The agenda-based recognition algorithm used by KARE is not exactly the same as the parsing algorithm in Recognizer, but we can draw some parallels between them. The concept of agenda is already present in chart-parsing, roughly corresponding to the global agenda in KARE. Also, the notion of chart in chart-parsing algorithms roughly corresponds to the local agenda in KARE. But, the agendas in
KARE are more flexible and facilitate users' intervention and guidance.

Another important difference between KARE and Recognizer is that KARE is designed with an explicit goal of keeping a human in the loop by providing natural hooks that allow the user to provide guidance to the recognition engine. In the future work section of Will's thesis [88], she listed possible ways a human could advise the recognition system. In many ways our work is a natural followup of her work in this regard. As described in Section 2.4 (and other places in this thesis), we believe that a human using the system is central to the success of any AI technique, and we have taken this idea seriously by designing KARE as a human-centred tool that provides mechanisms for the user to naturally intervene in the recognition process.

Neither KARE nor Recognizer work directly on the program text, and hence they can recognise a cliché even if the code corresponding to a cliché is not contiguous at one location and spreads across the whole program. Recogniser will skip any parts which cannot be recognised as part of any cliché. In contrast, KARE uses a granularity representation for representing plans as opposed to plan calculus and flow graphs. Granularity allows representation of various levels of detail in the same concept which is not possible in the plan calculus. If an object cannot be recognised at a detailed level, KARE can generalise the concept to a higher-level and try to recognise an abstract version of the concept.

Different interpretations of the same input are handled in Wills' method by using overlays. In KARE, multiple ways of looking at the same input are handled in a more versatile and flexible manner by L-clusters. For example, A loop that iterates over a list of elements can be specialised in more than one way. It can be specialised as a search loop which can terminate without processing all the elements, or a process loop that iterates over all the elements in a list. At the same time, a loop can also be specialised as a top-tested loop, where the termination test is performed before entering into the loop, or a bottom-tested loop that performs its termination test at the end of the loop body. L-clusters in KARE can represent these specialisations well.
5.5.3 KARE vs. SCENT

SCENT and KARE both use granularity hierarchies for representing plans and in many ways share the same properties. Granularity in SCENT allows it to recognise students programs in a robust way. It has been successful in analysing many real programs created by real students. There are some additional features in KARE that makes it more suited to the domain of analysing large software systems.

Granularity as used in SCENT does not have a good mechanism for sharing non-local knowledge. All the knowledge is strictly local. KARE has some mechanisms such as attributes and attribute translations that allow propagation of information from one place to another in the hierarchy. SCENT’s recognition is strictly top-down and somewhat rigid. KARE’s agenda-based recognition is more flexible and allows explicit user guidance in the course of the recognition process. This makes KARE’s technique scalable to large systems.

Another important difference between KARE and SCENT is the notion of context. Context in SCENT is a simple set of Lisp symbols. But, context in KARE is represented by means of a 4-tuple that allows extra information to be placed in the context to help in the process of recognition. Context in KARE is used as a means to narrow the search space once some parts of a hierarchy are recognised. Also, context in SCENT is imposed from the top. This is possible because the context for the top-level goal is known beforehand. However, in KARE, it is not possible to determine the context before recognition begins. Rather, in KARE, context is accumulated as a byproduct of the recognition process. Before and during the recognition process, KARE’s context acts as a place holder for information about the possible locations in the code to search for.

5.5.4 KARE vs DECODE

Quilici’s DECODE system [66] is one of the attempts to bring in a human user to augment the system’s capabilities. DECODE works by describing clichés as a set of constraints and trying to satisfy these constraints using the data from
the source code. DECODE recognition works in a bottom-up fashion. Once it finds the lower-level clichés, DECODE tries to combine these lower-level clichés into higher-level clichés. This approach is somewhat similar to KARE’s approach, but in KARE we use the powerful context mechanism to reduce the number of combinations needed while combining the lower-level clichés into higher-level ones. KARE uses granularity-based representation which allows representation at various grain sizes. This explicit representation facilitates the recognition algorithm to shift focus from one level to another as and when needed. DECODE allows the human user to assist the program understanding process by providing a structure editor for making connections. KARE takes this “human-assisted understanding” one step further by putting the human user at the centre of the whole process. DECODE’s automatic program understanding component, which is based on Ning’s Concept Recogniser [39], is somewhat similar to the observer level recognition in KARE. DECODE offers a design editor for the user to make necessary higher-level connections among these design elements recognised automatically. If we consider how the type of reverse engineering questions we discussed in Section 5.3 can be answered by DECODE, it will quickly lead to combinatorial explosion as the automatic recognition is attempted at higher and higher levels of concepts. In an attempt to to reduce this combinatorial explosion, DECODE uses an indexing mechanism to select which cliché to attempt next. DECODE’s indexing serves the same purpose as the aggregation links in the granularity in KARE. KARE’s aggregation links are more general and can provide for a way of using context modifier operators which helps in reducing the combinatorial explosion. KARE’s ability to use human intervention coupled with its powerful context mechanism will make it possible to answer these questions in a reasonable time.

5.5.5 KARE and Constraint Satisfaction

There has been a suggestion that program recognition can be accomplished by pure constraint satisfaction techniques [90]. There are many problems with this approach.
Constraint satisfaction does not avoid the problem of brittleness unless the knowledge of how to relax the constraints is specified. Representing the knowledge of constraint relaxation in such a way that it can be used by a CSP problem solver is difficult. So a traditional CSP is very brittle and is difficult to apply for large software systems. Also, traditional techniques in CSP do not have an explicit place for the user's intervention and guidance for helping the system. KARE's method allows one to recognise clichés incrementally with the assistance of the user's input into the recognition process. This helps to overcome the problem of scalability. Woods and Quilici [89] have performed experimental studies of constraint-based program-plan recognition techniques from the scalability point of view. Their experiments demonstrate that adding some of the well-known constraint techniques such as forward checking and variable sorting helps improve the scalability. Also, dynamic constraint ordering performs better on synthetic programs generated randomly than on real code occurring naturally. Our granularity-based method tries to address these issues by providing the unique approach of putting the human in the loop for overcoming the limitations of the system and representing plans at various levels of detail so that a coarse-level recognition can be done when fine-grained recognition is impossible. One interesting question is whether granularity-based recognition can be formulated in the constraint satisfaction framework. We suspect that it will be difficult, though not impossible, because the role of controls and the role of the human user cannot be easily incorporated into this framework.

5.5.6 KARE and Syntactic Methods

We described many software reverse engineering tools and techniques in Chapter 2. Some of the tools such as Rigi [54] and Murphy's source model extraction tool [55] provide efficient methods for computing various properties of source code of a software system. KARE, on the other hand, extracts conceptual patterns (or clichés) from the source code. Tools like Rigi cannot directly answer reverse engineering questions like the ones we discussed in Section 5.3. In other words, these tools pro-
vide capabilities that are orthogonal to what KARE provides. Even though there is no direct comparison between these tools and KARE, it should be emphasised that KARE can make use of such tools to complement their capabilities.

5.5.7 KARE and Simple Searching Tools

Many people use simple pattern searching tools such as UNIX grep and awk scripts for locating regular expression like patterns in files including source code files. It is natural to try to compare the power and usefulness of KARE's cliché recognition tool with such simple pattern-searching tools. Grep-like tools are very efficient in locating patterns that can be expressed as regular expressions. But expressing clichés that are large enough to be useful in understanding a software system is difficult to near-impossible. The power of these grep-like tools is somewhat comparable to the power of the observer objects used in KARE's recognition, though they are limited to only regular expressions. One can potentially use grep-like tools in an incremental fashion by repeatedly refining one's hypothesis about the program being understood. In this case, the human user has to form higher-level aggregations of these low-level results, including the cluster constraints. This imposes a very heavy cognitive burden on the user.

Imagine a case where two observers associated with a cluster are being recognised by grep like tools. By using grep, if the user finds 100 instances of one observer and 100 instances of the other, the user has to figure out which of these 100*100 = 10,000 pairs of these instances can form a valid parent object instance. If he/she uses KARE instead, these combinations are tested by the system and many of these combinations that do not satisfy the cluster constraints are rejected in the process. Thus, KARE offers a very valuable tool that can help in reducing the cognitive burden on the part of the user. In addition, grep like regular expression patterns can be easily incorporated into KARE's observer functions as demonstrated by the cliché used in the third experiment described earlier.
5.6 Concluding Remarks

In this chapter, we presented some of the results of experiments conducted using KARE for recognising clichés in a real-world software system. These results suggest that with an expert user (software engineer) using the system, it is possible to make knowledge-based program recognition practical.
Chapter 6

Conclusions and Future Directions

This thesis demonstrates the feasibility of a new technique for program recognition and reverse engineering based on the assumption that software is essentially a product of cognitive activity. This chapter discusses some of the contributions of our research to the field of software engineering and AI. We also describe some of the ideas that are worth pursuing further.

6.1 Contributions

The contributions of this research can grouped into two categories, (i) contributions to software engineering and reverse engineering research, and (ii) contributions to AI granularity-based reasoning.

6.1.1 Contributions to Software Engineering

Many problems in the process of program understanding and reverse engineering can be recast by viewing reverse engineering as a human cognitive activity. Many software reverse engineering tools concentrate on computing properties such as cross reference information and various metrics. The next big leap in the usability of software engineering tools can come only if the human aspects of software are factored into the design of these tools. Clichés represent conceptual knowledge of the programmers. Any system that can communicate with the user in terms of these concepts that are easily understood by expert software engineers has a better chance
of success. Tools such as KARE, which recognise clichés in source code, take a step towards making such an approach practical.

One of the main goals of KARE is to make cliché recognition tractable in large real-world software systems. In this section, we discuss various properties of KARE that make this goal possible. Figure 6.1 illustrates graphically some of the concepts discussed in the following sections.

### 6.1.1.1 Robustness

KARE represents clichés using granularity hierarchies. This allows the representation of clichés at various levels of abstraction. This makes KARE’s recognition robust in the wake of insufficient information. When KARE cannot recognise certain parts of a cliché, it can shift the focus to a more abstract level and try to recognise at a coarser grain size. For example, if a software engineer is trying to locate the occurrences of the cliché *doubly linked list*, it will be more helpful for him/her if the system comes back with some linked lists, which may or may not be recognised as doubly linked lists. The user can then figure out whether any of these linked lists are obscure variations of doubly linked lists. Explicit links among clichés of different grain sizes helps the system react naturally to specialise or generalise as and when needed. Since any knowledge-based approach eventually has to deal with situations that may not be covered by the existing knowledge base, the granularity-based approach used by KARE supports graceful degradation of the recognition activity [25].

### 6.1.1.2 Human-In-The-Loop

The software engineer who is attempting reverse engineering usually has a specific task in mind. Depending upon the task, she/he can provide a first stage segmentation of the code by selecting only relevant files for consideration. This step drastically reduces the size of the code on which recognition needs to be attempted from possibly millions of lines of code to perhaps a few thousands of lines of code. Moreover, the human user can select particular clichés to be recognised. KARE’s approach
also allows the human user to guide the recognition process by intervening via the agenda. This kind of control is extremely useful when the system cannot decide on its own which parts of the granularity hierarchy should be attempted next, or when the system is going down a path that is not very relevant for the problem at hand. Thus, in KARE, the user has a role to play at the following various stages:

- Before the recognition starts, the user can add elements to the recognition agenda and attach possible code sections/files to these agenda items. This will help the recognition algorithm to focus and to reduce the search space.

- During the recognition process the user can affect the recognition process by changing the agenda as the recognition proceeds. Some of the constraints that the system cannot satisfy can be satisfied by the user explicitly; and the system can then proceed from there.

- At any point during or after the recognition is done, the user can look through the recognition done so far by browsing the corresponding instantiated hierarchy and seeing the portions of the code fragments that correspond to each of the parts of any cliché. He/she can also find out the causes for any object not recognised or partially recognised. He/she can then modify/augment the automatic recognition and try the recognition again on the modified instantiated hierarchy.

Figure 6.1 gives a schematic diagram that illustrates how the human user relates to the other components of KARE. Granularity-based representation forms a natural basis for interacting with the user. The user and the system can communicate with each other in terms of the human-oriented strategies the clichés represent. This is one of the key factors that allows the human-in-the-loop to operate effectively.

6.1.1.3 Scalability

As mentioned in Chapter 5, there are two different aspects of scalability. They are illustrated graphically in Figure 6.1. Here we elaborate on these types of scalability and how KARE supports them.
Speed: "Scale-Up"

The SCENT system on which KARE's approach is derived could only work for programs "in the small" (typically fewer than 20-30 lines of code). The SCENT approach does not "scale-up" when used for recognising clichés in large software systems. SCENT assumes that the top-level context of the program directly corresponds to the root of the granularity hierarchy. But, the strategies in any real-world software system cannot be represented as a single hierarchy. We do not know in advance where in the code the roots of these hierarchies might occur. In fact, one of the goals of the recognition process is to locate such occurrences of these clichés in the code. Clusters in granularity hierarchies of SCENT are strictly local, i.e., there is no mechanism to allow the information to be shared across various clusters.

KARE's recognition techniques share many of the features of the SCENT's recognition techniques such as clusters, context, constraints and controls which help to reduce recognition time. Further, the structure of granularity allows principled constraint relaxation along abstraction and aggregation links. KARE adds additional features that make it possible to use cliché recognition in a large software system. The notion of attributes is used to expand the capability of granularity to allow the objects to access information about other objects in other clusters. The notion of
context in KARE is much more versatile than in SCENT and allows context information to be accumulated as recognition proceeds. Context relations allow complex constraints to be encoded into context modifier operators. The agenda in KARE facilitates the human user using the system to intervene and control the recognition process. The results presented in the previous chapter in Figure 5.7 and Figure 5.8 show that by using various constraints in combination with appropriate human intervention, the system can work effectively on large programs while keeping the number of instance combinations needed to be searched under control.

Expressiveness: "Scale-In and Scale-Out"

To be able to deal with real-world program code, any cliché representation mechanism should be expressive enough to capture wide variations in the code. If a cliché manifests in the code in various ways, one cliché description should capture many such variations. This type of coverage is called scale-out. KARE uses granularity for representing the knowledge of programming clichés. Granularity-based representation has many features that account for variations in the source code [20]. Some of these features are: clusters that capture different implementation strategies of the same cliché, observer functions that allow recognition of primitive clichés corresponding to the observer object, constraints that describe the necessary relationships among the component objects and their contexts, and controls that help encode control information (explicitly provided by the knowledge engineer) to direct the behaviour of the recognition algorithm. These observer functions and various constraints are encoded in such a way that they are not specific to each manifestation of the cliché in a real-world program, rather, they encode general properties of the cliché and cover various manifestations of the cliché in any program.

Granularity hierarchies represent concepts at various grain sizes. This allows a recognition engine to recognise concepts at various levels of detail depending upon the requirements of the task at hand and/or the availability of information. SCENT demonstrated that granularity is robust in recognising student programs, which very often are too perturbed to enable detailed recognition. KARE inherits the
robustness of SCENT to deal with insufficient information. For example the cliché
linked list can be recognised as a general linked list or a specific type of linked list
such as a singly linked list or a doubly linked list. If the user does not need to know
the details of the type of linked list (e.g., if the problem at hand does not need it)
the system can be directed to stop the recognition at that level. On the other hand,
if the user wants to know whether a linked list found in a system is a singly linked
list or doubly linked list, she/he can do so. This ability allows the cliché library to
cover various different situations.

However big may be the knowledge-base (i.e., cliché library, in this case), there
will be cases where requirements are not satisfied by the existing clichés. A natural
solution for this problem is to add new clichés to the library. There are two ways
one can add new clichés to the library: either a completely new cliché is created,
or an existing cliché is modified to suit the new requirements. KARE provides
a knowledge engineering tool that allows creation of new clichés. In addition, the
granularity structure allows one to take an existing cliché and specialise or scale-in to
a particular situation where a slightly different specialisation of the cliché is needed.
One can add the newly created cliché to the existing hierarchy as a specialised
version of the existing cliché by extending the abstraction dimension via an L-
cluster. This type of natural specialisation allows the newly created cliché to share
some of the framework of the existing cliché such as constraints and controls. This is
similar to the way the object-oriented programming paradigm allows a programmer
to specialise an existing class defined in a class library. Additionally, the knowledge
engineer can deploy the existing clichés to be part of a new cluster being defined.
This is another aspect of scalability that KARE's approach supports.

6.1.1.4 Supporting Human Activity

The cognitive process of the human doing reverse engineering must be supported
by developing methods and techniques for modelling this activity. We need to re-
orient the current methods that are preoccupied by an urge to compute formal
properties of software systems (e.g., various metrics) towards the goal of supporting
human understanding of software systems by interpreting the software in human terms. Our research attempts to support human activity by providing tools that can compute conceptual patterns, and communicating with the user in terms that are easily understood by human software engineer.

6.1.2 Contributions to AI and Granularity

The field of AI has always been enriched by attempting to apply techniques to new domains. Software engineering is no exception. In the process of trying to apply AI techniques like granularity-based reasoning, the limits of the technology are tested. In addition, new technical issues arise by using techniques in new situations. Some of the implications for research in granularity are presented below.

6.1.2.1 Practical AI Application Development

Most of the work in the field of AI attempts to automate some aspect of human activity. Most often this work does not take into account the fact that the human using the system has a lot to offer the system to achieve its objective. However, some AI-based work has attempted to use human intervention, such as pre-processing and/or post-processing of the input to simplify the system's task. There is also a growing trend in applied AI work to focus on providing intelligent tools for the human user. Our approach is very much in tune with these approaches to human-assisted AI. We go further, though, in suggesting that we should design techniques that put the human in the loop as an integral part of the problem-solving process.

6.1.2.2 New Techniques for Program Recognition

Granularity has been shown to be effective for program recognition in the small domain of LISP programming in SCENT. The approach presented in this thesis shows the practicality of using granularity for recognising clichés in large software systems. The agenda-based recognition algorithm, developed as a part of the research in this thesis is a flexible and powerful program recognition technique that
helps to solve some of the problems of brittleness and scalability faced by previous program understanding/ recognition techniques (See Section 5.5).

6.1.2.3 Scalability of AI Techniques

Many of the ideas and techniques in AI have been developed for toy domains. They work pretty well in small, narrow domains, but tend to fail when applied to a significant real-world problem. Software reverse engineering provides an excellent field where many AI (and other) techniques can be stretched to the limit. The research presented here shows we can use knowledge-based program recognition for practical purposes.

6.1.2.4 Old Issues Re-cast in New Setting

When reverse engineering and program understanding are viewed as human-centred cognitive activities, many of the old issues of user modelling, and human-computer interaction re-emerge in the new setting. Also, it becomes imperative to model the reverse engineering process so that effective cognitive support systems can be developed.

There are many potential issues that arise out of this new perspective of the software reverse engineering activity. The focus now shifts from characterising software artifacts using formal properties to modelling how humans doing reverse engineering understand software systems. A software system is no longer a static artifact or just an object of formal study. A software system is like a living, evolving organism that is intimately tied to the cognitive conceptualisations of the humans who designed the system. This brings up many new issues hitherto not dealt with in the reverse engineering community. For example, how can we represent and manipulate aspects of software systems that are helpful for human conceptualisation? The next section discusses some of the issues that need further research.
6.2 Future Directions

This thesis demonstrates an approach with the potential to make program recognition practical by placing the human software engineer "in the loop." However, there are many issues that need to be further investigated. In this section we discuss these issues.

6.2.1 Extensions to KARE

Future work that is of direct relevance include extensions to KARE itself. The following are some of these extensions:

- Database support for KARE: The current implementation of KARE (which is based on the AROMA granularity knowledge engineering shell) needs to load all the data in memory before it can perform recognition. When the data becomes larger, it takes a lot of resources to load the abstract syntax tree for the whole target system. It not only needs a lot more primary memory, it also takes more time to garbage collect. A better approach would be to store the abstract syntax tree in an object-oriented database and provide a client connection for KARE to the database server. In that case, only objects needed/accessed by KARE are stored in memory. In addition, many of the optimising techniques in a database such as caching and buffering could be exploited by KARE. There have been some efforts [68] that show evidence for the benefit of using database technology in AROMA. More work needs to be done in this area.

- Integrating REFINE and KARE: The current implementation of KARE runs in a process separate from REFINE. REFINE is used first to parse the raw C code into an AST. This AST is then converted into another format that is understood by KARE. This converted AST is saved as a flat file consisting of Common LISP S-expressions. These files when loaded into a running KARE process, will re-generate the AST in memory. This process is quite tedious and
time consuming for the computer as well as the user. There are two possible solutions to alleviate this problem:

- Implement KARE on top of REFINE,
- Implement a REFINE style parser in KARE.

Both of these approaches have advantages and disadvantages. The first approach has the advantage that many of the tools developed in REFINE such as tools for browsing the AST can be utilised in KARE. It has the disadvantage that KARE's recognition techniques have to be "ported" or re-implemented including all the granularity-based tools such as the AROMA knowledge-engineering tool. The second approach has the advantage that all the capabilities of KARE (including AROMA) need not be re-implemented. It has the disadvantage that the parser generator of REFINE needs to re-implemented, which is not a trivial task. There is a third possibility, although it requires further research. If there is a standard language that allows different applications to share the results of partial analysis of software, then it is possible to share information such as the AST between KARE and REFINE. But a lot of research needs to be done to develop such a "software mark-up language." ¹

- Knowledge engineering, testing and validation: This thesis has been mainly concerned with the development of techniques that attempt to satisfy the twin goals of scalability and the ability to use the human effectively. We also implemented a research prototype that is a proof of concept. More work is needed to augment and enhance the knowledge-base to include more clichés that are useful for reverse engineering. There is also a need to test the system including the cliché hierarchies on wide variety of target systems and tune the hierarchy and the constraints if necessary. The granularity representation mechanism that is used to encode clichés uses structural properties of the

¹The idea of a software mark-up language (SML) was suggested by Dr. Cordel Green at the 4th IEEE Working Conference on Reverse Engineering, held in October, 1997. I am not aware of any published research work yet on these ideas.
source code to define various constraints. The granularity as used in KARE does not have explicit way of representing dynamic properties of the program execution. If a cliché needs dynamic properties of the program to define the program, then it might be difficult to define such a cliché in KARE. One good example of such a cliché can be a cliché for error handling (or general event handling) in a program. Even though sometimes static code might provide some clues, in general it is difficult to determine which error/event is active at a particular point in the code. It would be useful to investigate which types of clichés are difficult or easy to encode using granularity.

- Better interface, visualisation tools: KARE currently has a simple code browsing mechanism where an object/objects of the AST are shown in an Emacs window with the relevant parts highlighted. There need to be better program visualisation tools that can show various parts of the code from various files collated together for viewing.

- Optimisation of the current implementation of KARE: There are many ways to optimise the current implementation of KARE. For example, the objects representing AST nodes are very inefficient in the sense that we used one CLOS (Common LISP Object System) class called AST-OBJECT to represent all types of nodes in the AST. AST nodes are differentiated from one another by a specific slot called AST-OBJECT-TYPE-NAME. Since each type of AST node has a different set of slots to represent attributes and links, the class AST-OBJECT has to have the union of the slots needed to represent all types of AST nodes, even though only a few of these slots are actually used. A better approach would be to define a separate class for each AST node type. This would add some additional classes to the system. Since we have only a limited number of AST node types, and there are more instances of each node type than the classes, the increase in the number of classes is more than compensated for by the reduction in the size of each instance. Another way KARE can be optimised is by profiling the system to find weak points and provide appropriate compiler
declarations to make it efficient.

- Formal study of granularity: Granularity-based representation and reasoning has been used for many applications ranging from diagnosing student’s activities in a tutoring system to information retrieval and filtering. There was also a preliminary formalisation of granularity [23]. But, there needs to be a more comprehensive study of various aspects of granularity (in light of experiences in using granularity in various applications) with respect to its capabilities and its limitations as a knowledge representation formalism.

- Granularity and constraint satisfaction: There is a clear connection between granularity-based representation and classical constraint satisfaction problem (CSP). In addition to explicit constraints encoded in a granularity hierarchy, the granularity structure itself provides implicit constraints on the kinds of data that can be successfully matched with the hierarchy. It would be useful to study the exact connection between granularity and CSP.

- Three dimensional granularity: The current formulation of granularity-based representation uses two dimensions of abstraction and aggregation. It seems that there are actually three dimensions viz., abstraction, aggregation, and approximation [79]. The abstraction dimension in the current formulation of granularity is a mixture of abstraction and approximation. Separating these dimensions might clarify some recognition issues but at the expense of possibly making the recognition process more complex. It would be illuminating to study these three dimensions more rigorously.

6.2.2 Program Recognition Based Reverse Engineering Environment

This thesis mainly concentrates on one tool for reverse engineering, i.e., the cliché recognition tool. Many supporting tools and techniques have been developed for a variety of reverse engineering activities. A full-fledged reverse engineering ap-
plication should include various techniques and tools to assist with the variety of activities a user might want to perform. Program understanding via cliché recognition is only one part of the overall set of tools required. Moreover, each tool should be able to work with many other tools to provide a usable support environment for the human software engineer doing reverse engineering.

Some of the other types of tools that could augment KARE would include:

* **Static analysis tools**: Tools like Rigi [53, 54] that perform analysis on the code using techniques like dependency analysis are very useful in conjunction with program recognition tools.

* **Program slicing tools**: A slice of a program is a collection of statements in a program that collectively achieve a common objective. A linear text of a program generally consists of many slices running in parallel through the same code. Program slicing [84, 38, 41] is a method of separating these slices to help understand the program. If we can successfully separate the slices in a program, we can use recognition tools on each slice separately. Hence, program slicing shows promise to reduce the search space in program recognition. Recent techniques for such as inter-procedural program slicing [28] could also be combined effectively with KARE for encoding observers and constraints.

* **Lightweight segmentation tools**: Tools such as the one developed by Murphy [55] provide efficient lightweight ways of extracting high-level source models of a system. This type of analysis helps in modularising a software system so that more heavyweight tools such as KARE can be used on individual modules. KARE can work with such tools naturally to complement each other's capabilities.

* **Re-documentation tools**: Dynamic document generation tools like the one proposed by Johnson [35] can work together with program recognition tools. If the system understands the program elements in terms of known program-
ming patterns, that information can be used by a document generation tool to produce a document that gives a more detailed description of the system.

* **Program transformation tools**: Program transformation techniques [39, 83] are useful in many ways. For example, one can use them to transform programs to semantically equivalent programs that have simpler structure. Alternatively, programs can be translated from older languages like FORTRAN and COBOL to more modern languages like Eiffel and C++. This will help in future maintenance, and in particular might allow the reverse engineer to transform a piece of code into a form more tractable to KARE.

* **Other methods**: Various techniques have been proposed recently for extracting information from programs that can be profitably used by KARE. Anquetil and Lethbridge [5] describe methods for extracting conceptual information from file names and O’Collahan and Jackson [57] proposed various type-inference techniques. It will be interesting to see how these techniques can be combined with KARE.

There are many ways our granularity-based recognition tool can work together with these other techniques. Each of these requires further research and development.

### 6.2.3 Formal Studies of Program Manipulation

There have been many attempts in various sub-disciplines of computer science which have tried to employ formal methods to analyse programs. Some of them have had the goal of characterising and understanding the nature of computation and programming. It would be a fruitful exercise to investigate some of these methods in the context of reverse engineering to see how they could be used with KARE.

- Wile [86] has proposed a formal calculus for manipulating abstract syntax trees. Many of the mechanisms used in KARE such as context modifier operators, and various constraints, are essentially operations on the AST representation of source code. The granularity formalism used in KARE does not
require a program to be converted to an AST, but we found that working with the AST representation is more efficient than flat source code text. It might be revealing to see if the operators used by KARE can be formulated in terms of a formal mechanism such as Wile's. That may enable a knowledge engineer to encode various constraints, observer functions, and context modifiers to be formulated in a more declarative style.

- Abd-El-Hafiz and Basili [2] have proposed a method of program understanding using a formal analysis of loop structures. This approach is aimed at producing semantically sound program documentation from code. It would be interesting to see how such techniques could be used in conjunction with KARE.

- Paul and Prakash [63] have proposed an algebraic query language called the source code algebra that can help formulate queries for requesting information from source code. It may be that this algebraic query language could be used to encode observer functions in KARE. If we could do that, these algebraic queries could be compiled into very efficient search algorithms for locating the observers.

### 6.3 Concluding Remarks

This thesis has presented a program cliché recognition technique based on the robust granularity representation formalism. The recognition technique allows explicit human intervention before, during, and after the recognition. Since granularity objects represent human strategic concepts, cliché recognition based on this representation allows effective communication between the system and the user in terms of the concepts the user can understand. Experiments with our prototype implementation of the recognition engine in KARE show evidence that the technique is scalable to recognise clichés in large real-world software systems with appropriate intervention by an expert software engineer.

Of course, there needs to be more research done on various aspects of this work,
both theoretical, in terms of granularity, and practical, in terms of implementation. However, our approach of using the human user's expertise to guide the system is in tune with the growing trend in applied AI research to shift from completely automating an activity to providing effective tools for a human user. In fact our ideas of the human-in-the-loop takes this a step further and places the human user at the centre of the problem solving-process.
References


143


147


Appendix A

Granularity Hierarchies

A.1 Granularity Objects for Linked List Cliché

The listing of clichés in this appendix is simplified considerably to make them understandable. Only S-objects and observers are shown. Some of the aggregation and abstraction links are also shown. Attributes and various types of constraints are not shown. Also omitted are context modifiers, and clusters.

LINKED-LIST (root)
  abs-children
    SINGLE-LINKED-LIST
    DOUBLE-LINKED-LIST
  agg-children
    LINKED-LIST-DATA
    LINKED-LIST-OPERATIONS

LINKED-LIST-DATA (agg-parent: LINKED-LIST)
  abs-children
    SINGLE-LINKED-LIST-DATA
    DOUBLE-LINKED-LIST-DATA
  agg-children
    LINKABLE-NODE
    LINKED-LIST-HEAD

LINKABLE-NODE (agg-parent: LINKED-LIST-DATA)
  abs-children
    SINGLE-LINKABLE-NODE
    DOUBLE-LINKABLE-NODE
  agg-children
    LINKABLE-NODE-OBSERVER

LINKED-LIST-HEAD (agg-parent: LINKED-LIST-DATA,
SINGLE-LINKED-LIST-DATA,
DOUBLE-LINKED-LIST-DATA)

agg-children
  LINKED-LIST-HEAD-OBSERVER

SINGLE-LINKABLE-NODE (agg-parent: SINGLE-LINKED-LIST,
  abs-parent: LINKABLE-NODE)
  agg-children
    SINGLE-LINKABLE-NODE-OBSERVER

SINGLE-LINKED-LIST (abs-parent: LINKED-LIST)
  agg-children
    SINGLE-LINKED-LIST-DATA
    SINGLE-LINKED-LIST-OPERATIONS

DOUBLE-LINKED-LIST (abs-parent: LINKED-LIST)
  agg-children
    DOUBLE-LINKED-LIST-DATA
    DOUBLE-LINKED-LIST-OPERATIONS

SINGLE-LINKED-LIST-DATA (agg-parent: SINGLE-LINKED-LIST)
  agg-children
    SINGLE-LINKABLE-NODE
    LINKED-LIST-HEAD

DOUBLE-LINKED-LIST-DATA (agg-parent: SINGLE-LINKED-LIST)
  agg-children
    DOUBLE-LINKABLE-NODE
    LINKED-LIST-HEAD

LINKED-LIST-OPERATIONS (agg-parent: LINKED-LIST)
  abs-children
    SINGLE-LINKED-LIST-OPERATIONS
    DOUBLE-LINKED-LIST-OPERATIONS

SINGLE-LINKED-LIST-OPERATIONS (abs-parent: LINKED-LIST-OPERATIONS,
  agg-parent: SINGLE-LINKED-LIST)
  abs-children
    SINGLE-LINKED-LIST-INSERT
    SINGLE-LINKED-LIST-DELETE
    SINGLE-LINKED-LIST-SEARCH
SINGLE-LINKED-LIST-INSERT (abs-parent: SINGLE-LINKED-LIST-OPERATIONS)
  agg-children
   ALLOCATE-LINK-NODE
   NULL-TEST
   SINGLE-LINKED-LIST-INSERT-NULL-LIST
   SINGLE-LINKED-LIST-INSERT-NON-NULL-LIST

ALLOCATE-LINK-NODE (agg-parent: SINGLE-LINKED-LIST-INSERT,
                      abs-parent: ALLOCATE-MEMORY)
  agg-children
   ALLOCATION-CALL
   ASSIGN-TO-NODE

ALLOCATION-CALL (agg-parent: ALLOCATE-LINK-NODE,
                  ALLOCATE-MEMORY,
                  abs-parent: FUNCTION-CALL)
  agg-children
   ALLOCATION-CALL-OBSERVER

ASSIGN-TO-NODE (agg-parent: ALLOCATE-LINK-NODE)
  agg-children
   ASSIGN-TO-NODE-OBSERVER

NULL-TEST (agg-parent: SINGLE-LINKED-LIST-INSERT)
  agg-children
   NULL-TEST-OBSERVER

SINGLE-LINKED-LIST-INSERT-NULL-LIST
  (agg-parent: SINGLE-LINKED-LIST-INSERT)
  agg-children
   ASSIGN-TO-HEAD

ASSIGN-TO-HEAD (agg-parent: ALLOCATE-LINK-NODE)
  agg-children
   ASSIGN-TO-HEAD-OBSERVER

SINGLE-LINKED-LIST-INSERT-NON-NULL-LIST
  (agg-parent: SINGLE-LINKED-LIST-INSERT)
  abs-children
   SINGLE-LINKED-LIST-INSERT-REAR
   SINGLE-LINKED-LIST-INSERT-FRONT
   SINGLE-LINKED-LIST-OTHER-INSERT
  agg-children
ASSIGN-TO-NEW-NODE-LINK
ASSIGN-TO-LIST-LINK

. . .

<< truncated >>

. . .
A.2 Granularity Objects for Searching Cliché

SEARCH-FOR-ITEM (root)
   abs-children
     ARRAY-SEARCH
     HASH-TABLE-SEARCH

HASH-TABLE-SEARCH (abs-parent: SEARCH-FOR-ITEM, HASH-TABLE-OPERATIONS)
   agg-children
     HASH-TABLE-DATA
     HASH-INDEXING
     HASH-FUNCTION-CALL
     SEARCH-IN-BUCKET

ARRAY-SEARCH (abs-parent: SEARCH-FOR-ITEM)
   agg-children
     ARRAY-DECLARATION
     ARRAY-SEARCH-INITIALIZE
     ARRAY-ADVANCE-INDEX
     ARRAY-SEARCH-TERMINATION

HASH-TABLE-DATA (agg-parent: HASH-TABLE-SEARCH)
   agg-children
     BUCKET-ARRAY-DECLARATION
     HASH-TABLE-SIZE

BUCKET-ARRAY-DECLARATION (agg-parent: HASH-TABLE-DATA)
   agg-children
     ARRAY-DECLARATION

ARRAY-DECLARATION (agg-parent: BUCKET-ARRAY-DECLARATION)
   agg-children
     ARRAY-DECLARATION-OBSERVER

HASH-INDEXING (agg-parent: HASH-TABLE-SEARCH)
   agg-children
     HASH-INDEXING-OBSERVER

HASH-FUNCTION-CALL (agg-parent: HASH-TABLE-SEARCH)
   agg-children
     HASH-FUNCTION-CALL-OBSERVER

SEARCH-IN-BUCKET (agg-parent: HASH-TABLE-SEARCH)
abs-children
SEARCH-IN-LIST-BUCKET

SEARCH-IN-LIST-BUCKET (abs-parent: SEARCH-IN-BUCKET)
agg-children
  INITIALIZE-HT-SEARCH
  ADVANCE-BUCKET-LINK
  BUCKET-SEARCH-TERMINATION

INITIALIZE-HT-SEARCH (agg-parent: SEARCH-IN-LIST-BUCKET)
agg-children
  INITIALIZE-HT-SEARCH-OBSERVER

ADVANCE-BUCKET-LINK (agg-parent: SEARCH-IN-LIST-BUCKET)
agg-children
  ADVANCE-BUCKET-LINK-OBSERVER

BUCKET-SEARCH-TERMINATION (agg-parent: SEARCH-IN-LIST-BUCKET)
agg-children
  BUCKET-FOUND-TEST
  BUCKET-END-TEST

BUCKET-FOUND-TEST (agg-parent: BUCKET-SEARCH-TERMINATION)
agg-children
  BUCKET-FOUND-TEST-OBSERVER

BUCKET-END-TEST (agg-parent: BUCKET-SEARCH-TERMINATION)
agg-children
  BUCKET-END-TEST-OBSERVER

ARRAY-SEARCH-TERMINATE (agg-parent: ARRAY-SEARCH)
agg-children
  ARRAY-SEARCH-TERMINATE-FOUND-TEST
  ARRAY-SEARCH-TERMINATE-END-TEST

ARRAY-SEARCH-TERMINATE-FOUND-TEST
  (agg-parent: ARRAY-SEARCH-TERMINATE)
agg-children
  ARRAY-SEARCH-TERMINATE-FOUND-TEST-OBSERVER
ARRAY-SEARCH-TERMINATE-END-TEST
    (agg-parent: ARRAY-SEARCH-TERMINATE)
    agg-children
    ARRAY-SEARCH-TERMINATE-END-TEST-OBSERVER

ARRAY-SEARCH-INITIALIZE (agg-parent: ARRAY-SEARCH)
    agg-children
    ARRAY-SEARCH-INITIALIZE-OBSERVER

ARRAY-SEARCH-ADVANCE-INDEX (agg-parent: ARRAY-SEARCH)
    agg-children
    ARRAY-SEARCH-ADVANCE-INDEX-OBSERVER

    ...

    << truncated >>

    ...

157
A.3 Granularity Objects for Date and Time Cliché

DATE-TIME (root)
    abs-children
        DAY-DATE-RELATED
        TIME-RELATED
    agg-children
        DATE-TIME-OBSERVER

DAY-DATE-RELATED (abs-parent: DATE-TIME)
    agg-children
        DAY-DATE-DECLARATION
        DAY-DATE-OPERATIONS

DAY-DATE-DECLARATION (agg-parent: DAY-DATE-RELATED)
    agg-children
        DAY-DATE-DECLARATION-OBSERVER

DAY-DATE-OPERATIONS (agg-parent: DAY-DATE-RELATED)
    agg-children
        DAY-DATE-OPERATIONS-OBSERVER
    abs-children
        DAY-DATE-CONVERSION
        DAY-DATE-COMPARISON
        DAY-DATE-ARITHMETIC

DAY-DATE-CONVERSION (abs-parent: DAY-DATE-OPERATIONS)
    agg-children
        DAY-DATE-CONVERSION-OBSERVER

DAY-DATE-COMPARISON (abs-parent: DAY-DATE-OPERATIONS)
    agg-children
        DAY-DATE-COMPARISON-OBSERVER

DAY-DATE-ARITHMETIC (abs-parent: DAY-DATE-OPERATIONS)
    agg-children
        DAY-DATE-ARITHMETIC-OBSERVER
TIME-RELATED (abs-parent: DATE-TIME)
agg-children
   HOUR-TIME-DECLARATION
   HOUR-TIME-OPERATIONS

HOUR-TIME-DECLARATION (agg-parent: TIME-RELATED)
agg-children
   HOUR-TIME-DECLARATION-OBSERVER

HOUR-TIME-OPERATIONS (agg-parent: TIME-RELATED)
agg-children
   HOUR-TIME-OPERATIONS-OBSERVER
Appendix B

Sample Abstract Syntax Trees Used in KARE

Figure B.1: AST representation of some of the declarations in Mosaic
Figure B.2: AST representation of some of the declarations in Mosaic (contd.)
Figure B.3: AST representation of some of the functions in Mosaic
Figure B.4: AST representation of some of the functions in Mosaic (contd.)
Figure B.5: AST representation some of the functions in Mosaic (contd.)
Figure B.6: AST representation some of the functions in Mosaic (contd.)
Figure B.7: AST representation some of the functions in Mosaic (contd.)