GNIREENIGNE: Reverse Engineering or Backwards Engineering?

Srinivas Palthepu

Supervisors: Prof. Gordon McCalla and Prof. Jim Greer

Abstract

Reverse engineering involves extracting high-level information from existing code. Reverse engineering is a crucial part of software maintenance since a maintainer needs to understand the code before attempting any modification. Many efforts have been made to automatically extract higher-level design information from the code to facilitate various maintenance activities. But many of them are limited to extracting and providing only static information like call graphs, and normally involve techniques like parsing. Some of them also try to "understand" the program in a limited toy domain so that it can generate a high-level description of the system in terms of the common programming concepts. Also, most of the systems do not have any knowledge of the domain. Only recently people have realized the limitations of current approaches and have begun advocating systems that will provide descriptions in terms of human-oriented domain concepts. This paper proposes a new approach to reverse engineering large software systems using granularity-based program recognition as the main recognition mechanism. The approach presented here is robust, flexible and scalable. The granularity formalism can represent human oriented strategic knowledge better than other formalisms. The central message of the paper is that we need to take the human into account when designing any software reverse engineering tools. Some of the implications of the proposed approach for future research in the field of reverse engineering are discussed. Also included is a discussion of how software reverse engineering provides an excellent domain where one can test different ideas about granularity research.

1 Introduction

Reverse Engineering is the process of reconstructing high-level design information from program code [6]. Reverse engineering in traditional engineering disciplines is an extraneous activity but in the field of software engineering it is an integral part of software maintenance. A maintainer needs to understand the code as to which part of the code is implementing which of 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.

Any reverse engineering effort typically consists of understanding an existing software system. It starts with existing information such as original design documents, user manuals, and the code. Although many sources of information are available, the actual code is the ultimate description of the current state of the system. Hence, at the crux of the reverse engineering process is the problem of program understanding.

Almost all of the previous attempts at program understanding have taken one of the two broad approaches [23]. 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 programming concepts variously known as cliches [24], programming plans [11], or abstract concepts in a conceptual model [5, 13, 12]. The system then finds out which of the plans in the library, combined in a particular way, will achieve the goals of the program under consideration. There might not be a single plan capable of achieving the desired set of goals, in which case there might be some techniques for modifying the plans and/or combining more than one plan to achieve the objectives. Once such plans are identified, they are connected to the actual program code fragments. The second approach, known as the bottom-up approach [15, 26, 27], 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 both these approaches, we need to have a large set of programming plans in some form or other.

1.1 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 does not communicate the intent of the programmer well to other human beings. But in many situations the program has to be understood in the absence of the person who has created it. The code can communicate the computational intent to some extent but it cannot communicate the conceptual intent and how that conceptual intent is related to the system's objectives in a domain.

This is one of the fundamental reasons why understanding a program is so difficult. Other reasons why reverse engineering is difficult include large size and enormous complexity of real-world systems, and inadequate, non-existing or out-of-date documentation.

1.2 Towards human-oriented software engineering tools

There are many tools and techniques developed so far for supporting reverse engineering activities. They involve techniques developed in many fields such as Programming Languages, Artificial Intelligence (AI), Intelligent Tutoring Systems (ITS), Graph Theory, and Graphical Visualisation. Many tools are syntactic in nature and extract information such as call graphs, interfaces, and dependency graphs. Some tools like Rigi [18, 19] provide a graphical environment in which an engineer can view/organise/group different parts of the system to understand them and record her/his understanding of them. But none of them takes the perspective that a software system is an artifact which is primarily a human creation. They consider a software system as an object of formal study and try to deploy traditional methods like measurement, etc, in the realm of software engineering. Only recently are people recognising the need to use human-oriented domain concepts to guide program understanding[2, 20]. Some of the program understanding techniques developed in other fields like AI [27] and ITS [11] work only in small toy domains. Similarly, many reverse engineering efforts are either restricted to small programs and might not scale up to real world applications, or too impoverished and leave much of the work to be done to the user without providing any real help.

Since programming and program understanding is primarily a cognitive activity, we need to develop tools centred around the central role of the human. Understanding a software artifact requires reconstructing, at least partially, many of the cognitive connections between various parts of the software system along with many engineering decisions made in the process. Software engineering is a unique field where there is always a tension between formal models inspired from traditional engineering disciplines and cognitive models reflecting the individuals' expression of ideas in computing. We need to develop tools that can try to resolve this tension by using both of these aspects in a way that compliments each other.

2 Motivations and goals

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

- Software reverse engineering in particular and software engineering in general are human cognitive activities. Systems need to provide support for humans to understand a human created artifact.
- Technology has been limited and brittle when it comes to cognitive domains.

Therefore we propose an approach for attacking the reverse engineering problem that:

- Views reverse engineering in human terms:
  - to describe software systems in terms of human cognitive concepts and goals,
  - to take into account human cognitive limitations to allow information about the software to be appropriately chunked,
  - to keep the human "in the loop" at each stage to overcome the brittleness of the system supporting reverse engineering,
  - to provide natural support for the reverse engineer by allowing her/him to guide, validate, and modify the decisions and conclusions about the system being reverse engineered;
- Views the process of reverse engineering as one requiring the description of the system in terms of domain objectives;
- Views a software system as an evolving organism as opposed to a static object.

With these ideas as central motivating factors, we are currently working on the adaptation of techniques originally developed in the cognitive domain of supporting human learning to the task of reverse engineering. In particular, we propose to use:

- granularity-based representation and reasoning:
  - for recognising human programming plans in the code at various levels of approximation and abstraction,
  - for representing multiple viewpoints/mental models of the program[1],
  - to record the users' evolving understanding of the software system being reverse engineered.
- heuristic knowledge to segment the code to facilitate recognition of large real-world systems.
- user modelling and human-computer interaction methods for effectively controlling the interaction with the user.

In addition, there are a few pertinent technical issues that are very crucial to the success of any reverse engineering technique. Some of the key issues can be listed as follows:

- Flexibility: Many techniques have been developed in isolation and have a rigid control for plan recognition and reverse engineering. The goal of this approach is to propose mechanisms that can be well integrated with many other techniques.
- Generality: The techniques should be general enough to work in many situations such as different languages, different domains, and different objectives of the reverse engineering activity.
- Scalability: Many techniques have been developed that work in a small domain but will not scale-up to large systems. The techniques should be able to scale up to large real-world systems to be useful and successful.
- Human in the loop: Techniques should allow user guidance and intervention during the process. This is very useful as it is unlikely that any technique will be able to fully automate the process of reverse engineering. In fact this can be considered as a central aspect that can make any tool/technique succeed or fail.

The approach we present in this paper directly addresses these issues. Section 2.1 describes an example situation that requires reverse engineering of an existing real-world software system. The purpose of the example is to illustrate some of the issues in a proper context. It also discusses a possible scenario of interaction with a hypothetical reverse engineering system. Section 3 describes how this can be implemented using a granularity-based frame work.

2.1 A working example – Mosaic

Let us consider a real world example to illustrate the kind of facilities one needs while trying to reverse engineer a software system. Mosaic is a program that allows one to browse through the World Wide Web (WWW, or simply the Web). It offers many facilities for a user browsing through the Web. One of the features Mosaic supports is keeping a user history. Mosaic keeps track of all the Web sites a user has visited during a session. When a user terminates a session with Mosaic, the list of the sites visited is saved in a file. This history file is loaded back when the user starts another session with Mosaic.

Let us assume that a hypothetical user (e.g. a software maintainer) wanted to enhance the features of the Mosaic's history mechanism. This is a very good example of a classical re-engineering task. The first step in the process of re-engineering is reverse engineering. Our hypothetical user needs to understand how the history mechanism is currently implemented in Mosaic. The reverse engineering problem can be stated as follows:

General Goals:

- (i) to understand Mosaic's history mechanism and,
- (ii) to modify this history recording mechanism.

From her/his knowledge of usage of the system, let us assume our user has drawn certain conclusions about Mosaic, listed below. This knowledge is only user-level, and does not depend upon how Mosaic is implemented.

- Mosaic stores the history as a sequence of Web site addresses and time stamps.
- The user does not know how the order of the entries in the file is determined. They are neither in chronological order nor in alphabetical order.
- Mosaic modifies the time stamp of all the entries in the history each time Mosaic is used, irrespective of whether that address is accessed or not during the session.

Subgoal (i), above, can thus be converted into a series of questions. In particular, the user needs to answer the following questions:

1. How does Mosaic store history in memory?
2. How does Mosaic load the previous history upon invocation?
3. How does Mosaic store the user history in the file?

Similarly the subgoal (ii) can be further refined into the following specific goals:

(i) To make Mosaic record the access frequency for each address:

(ii) To make Mosaic modify the time stamps only when they are accessed.

The first task in the reverse engineering process is to isolate the parts of the system that deals with the user history mechanism. One possible way to do this is to do a detailed analysis of the whole code and then try to find out which part of the system deals with the history mechanism. This approach has two problems: first, it requires detailed knowledge of the domain functions and how they are implemented, and second, it requires so much search that a complete recognition is not possible. Another alternative is to apply a coarse grained search mechanism to isolate possible starting points. For this, the system uses a domain model that will help in decomposing the software system into major constituent subsystems. For example if the domain is an operating system, the components might be a file system, a process management system, a memory management system, and a network management system. In our example, the user can specify, in the form of a query, what kind of function names and/or variable names could plausibly be used for operations related to user history. Once the reverse engineering system finds a possible list of function/variables along with the files in which they are declared/defined, then the user can override these decisions (select/reject).

Once this top-level decomposition is done, the user can then perform a detailed analysis of the isolated subpart of the code for recognition. There are many options for the user at this point. She/He can let the system do an exhaustive analysis of the isolated part of the system and find out all the plans in the code. Alternatively, the user can limit the search to only few specific plans according to her/his objective. If the user directs the system by specifying only some plans to be located, the system should only search for these plans and do a detailed recognition. Otherwise, the system should try to analyse the code for all possible plans. In the Mosaic example, the user needs to understand the mechanism by which the user history is stored. This should include a type of aggregate data structure plan. The user might suspect that there might be some ordering on these entries, so a sorting plan is included in the set of possible plans.

Once the user has described all the possible plans of interest, the recogniser will be run on the code to recognise all instances of such plans in the code. For the code shown in Figure 1, the system will come up with a description such as the one shown in Figure 2. Figure 3 shows an English description of the same design structure shown in Figure 2.

Once this analysis is done, the user will have starting points and some understanding of the code fragments. She/He can then proceed for further analysis to find out the effects of modifying these code parts. For example, she/he might want to find out what are the other modules that depend on this data structure and routines or she/he might want to figure out what will be the effect of the required changes on the rest of the system.

3 Granularity-based recogniser

3.1 Overall Scenario

This section describes a general framework for program recognition in the context of reverse engineering of large software systems and how granularity-based recognition is used in this framework. The next section gives a brief description of granularity-based recognition. Section 3.3 describes how the granularity mechanism is applied to recognise program patterns.

In the beginning, the code to be reverse engineered is converted into an abstract syntax tree (AST). The actual recognition is done on the AST. The following are the general steps a user goes through while doing reverse engineering using the granularity-based approach.

(i) The user starts with models of standard programming strategies and domain specific models encoded in granularity hierarchies, either retrieved from a library of such hierarchies or tailored for this application;

(ii) The system will recognise and generate explanations for the code in terms of the concepts and plans represented in these granularity hierarchies;

(iii) The user can augment the recognised plans ("instantiated hierarchy") by adding-in domain-specific strategies. The user and the system iterate through the recognition-augmentation cycle until the she/he is satisfied with the description.

(iv) The user will then use these descriptions and instantiated hierarchies produced by the system to construct a domain-specific model of the code.

The recognition step (ii) above is actually a two stage process. In the first stage, the system tries to locate the places in the code where there is evidence for the presence of any of the relevant plans. For this, the top-level observers in the granularity hierarchies of the plans in the library need approximate matching mechanisms. We plan to use pattern-based search mechanism such as the one in [22] for this purpose. Each observer can encode the patterns which correspond to evidence for the plans. Even if there are some false positives at this stage, it will not matter much since the next stage will reject

```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;
  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 1: A part of the C program in Mosaic implementing the user history mechanism

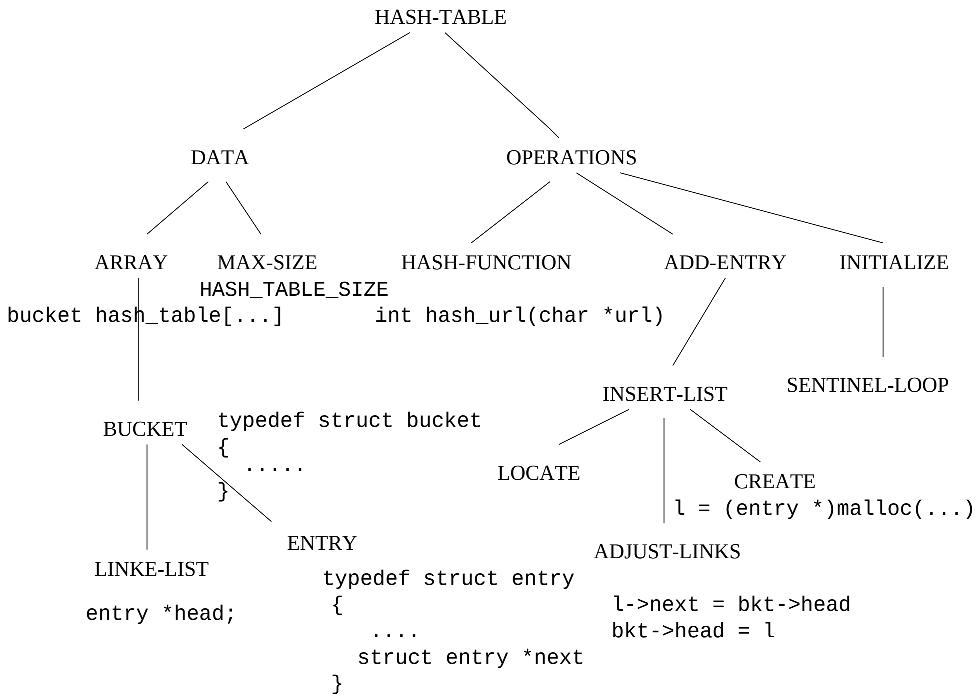

Figure 2: The design tree produced after the recognition process

```
HISTORY is stored in a HASH-TABLE, which is implemented as
an  ARRAY of BUCKETS.

BUCKET is a structure which contains a LINKED-LIST of ENTRYs

ENTRY is a structure containing <*url> and a pointer to next ENTRY

HASH-FUNCTIONS is <int hash_url(char *url)>

The function <dump_bucket_count()> uses a COMPLETE-LOOP over the
HASH-TABLE. It OUTPUTs <count> of each BUCKET

The function <add_url_to_bucket(int buck, char *url)>
does an INSERT-FRONT of the LINKED-LIST

the function <mo_read_gloabl_history_(char *filename)> creates
initial HASH-TABLE.
   Uses SENTINEL-LOOP to create the HASH-TABLE
. . .

etc
```

Figure 3: A description of user-history in Mosaic generated by the recognition system

any such false recognitions. In the second stage, the user can direct the system as to which of these plans are most relevant. The user can specify her/his intention by using a Granularity-based Query Language (GQL). GQL is a language in which a user can specify and direct the recognition tool to look for objects of interest belonging to specific parts of the plan library. GQL must be rich enough so that the user can specify various plans and various combinations of them. An instance of a granularity hierarchy is generated from GQL that will provide a context in which the detailed recognition will take place.

3.2 What is granularity?

The idea of formally studying granularity was first proposed by Hobbs [10] 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 [7, 16]. They developed a knowledge representation formalism called the *granularity hierarchy* and used it for representing knowledge of strategies used by a student in an intelligent tutoring system [8]. The granularity formalism has been used successfully in an ITS called the SCENT advisor [17, 9] to recognise strategies in students' LISP programs.

Granularity hierarchies are directed graphs where nodes represent strategies. Strategies are connected to each other via two distinct types of links. One type of link is the *abstraction* link and the other type of link is the *aggregation* link. The abstraction link provides for ISA and approximation links [1] between strategies whereas the aggregation link provides for a part-whole relationship among strategy nodes.

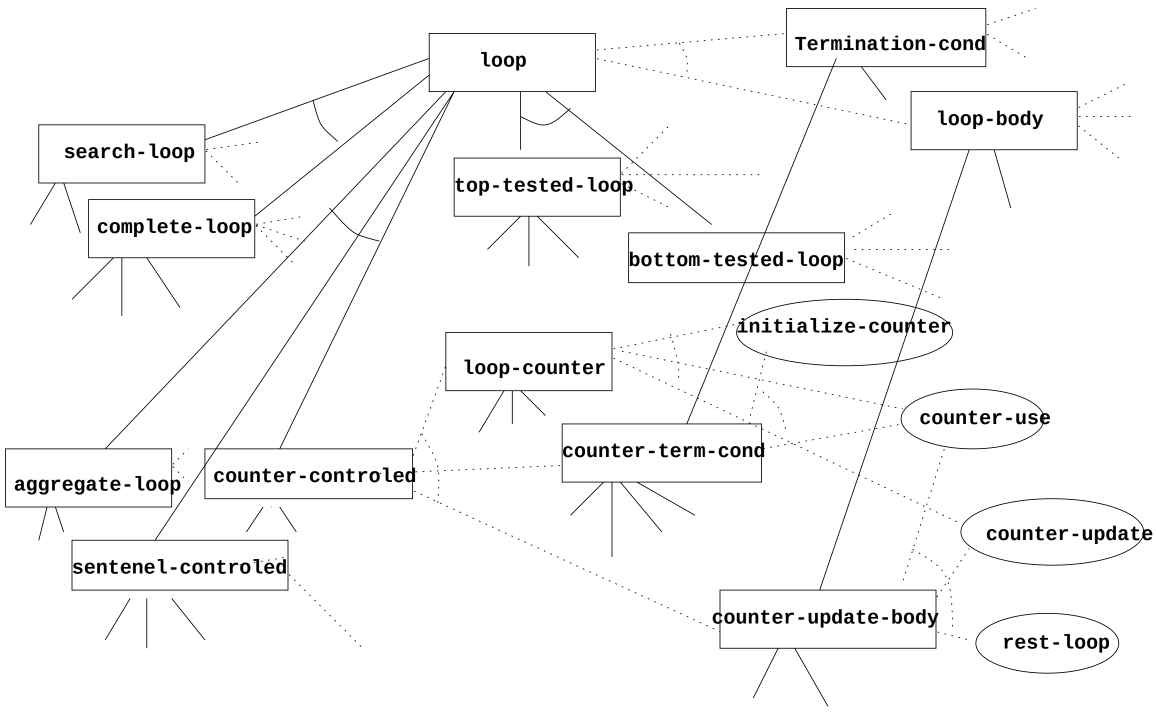

Figure 4: A Granularity Representation for Loops in Programming

Figure 4 shows an example of a common programming concept, `LOOP`, represented using a gran-

[1] ISA relationship between objects in semantic networks [14] has the meaning that if A is related to B by ISA relationship, it means that an object of type A is a special type of object of type B, in other words, set of instances of A are subset of set of instances of B

ularity hierarchy. In Figure 4, 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 each object. Similarly, along the abstraction dimension, an object can be classified into specialisations in many ways using *L-clusters.* These concepts are illustrated in Figure 4 using an example. The concept of a loop (`LOOP`) in a program consists of a termination condition (`TERMINATION-COND`) and a body (`BODY`) represented as a K-cluster. This type 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* in the real-world. In Figure 4, strategy objects are shown as rectangles and observer objects are shown as ovals. Also, a loop can be refined along the abstraction dimension in many ways. In the example in Figure 4, the concept loop can be specialised in three different ways, giving rise to three different L-clusters: (i) `TOP-TESTED-LOOP` and `BOTTOM-TESTED-LOOP`, (ii) `SEARCH-LOOP` and `COMPLETE-LOOP`, (iii) `AGGREGATE-LOOP`, `SENTINEL-CONTROLLED-LOOP` and `COUNTER-CONTROLLED-LOOP`. Once concepts are represented in the granularity formalism, they can be used to recognise plans in a program. Each strategy object can have various types of constraints and controls for specifying how the parts are aggregated and how the articulations (and its opposite, simplifications) are done. More details on the granularity formalism, and how it is used in recognition, can be found in [16, 9, 7].

3.3 Granularity for reverse engineering

The granularity formalism has many attractive features that make it useful for reverse engineering:

- Granularity can represent human-oriented concepts and strategies very well. That gives the system the ability to recognise and explain the code in the user's terms.

- Different dimensions of granularity allow representation of concepts/strategies at various levels of abstraction (abstraction dimension) and various levels of detail (aggregation dimension). This helps in recognising the strategies in a program at various levels. One can cut off the refinement at any level depending upon the user's objective.

- The same code can be viewed differently depending upon the task/goal and/or individual's conceptualisation. The L-clusters in granularity allow one to encode these different *viewpoints* on the code. For example, a `LOOP` can be viewed as either `COUNTER-CONTROLLED` or `SENTINEL-CONTROLLED` (see Figure 4). At the same time it can be viewed as one of `TOP-TESTED` (`DO-WHILE`) or `BOTTOM-TESTED` (`DO-UNTIL`). Thus, the L-cluster mechanism of granularity allows us to provide multiple perspectives on the same code.

- The granularity formalism is very general and allows encoding of many kinds of concepts including domain related concepts, computation related concepts, strategic concepts, and code level concepts.

- Observers can encode various kinds of recognition strategies. This gives a high degree of flexibility that allows the system to combine many different types of techniques that help in speeding up the process of recognising programming concepts. For example, some top-level observers can have simple pattern-based search engines such as the one proposed by Paul and Prakash [22]. The observers at a fine-grained level can encode detailed program units.

- Various levels of granularity allow a partial or an approximate recognition of plans in the presence of unfamiliar (un-recognisable) code. This ability is essential when trying to recognise/understand large programs as many programs are not entirely written out of predetermined plans. Instead, plans are used as templates around which a program is woven [25].

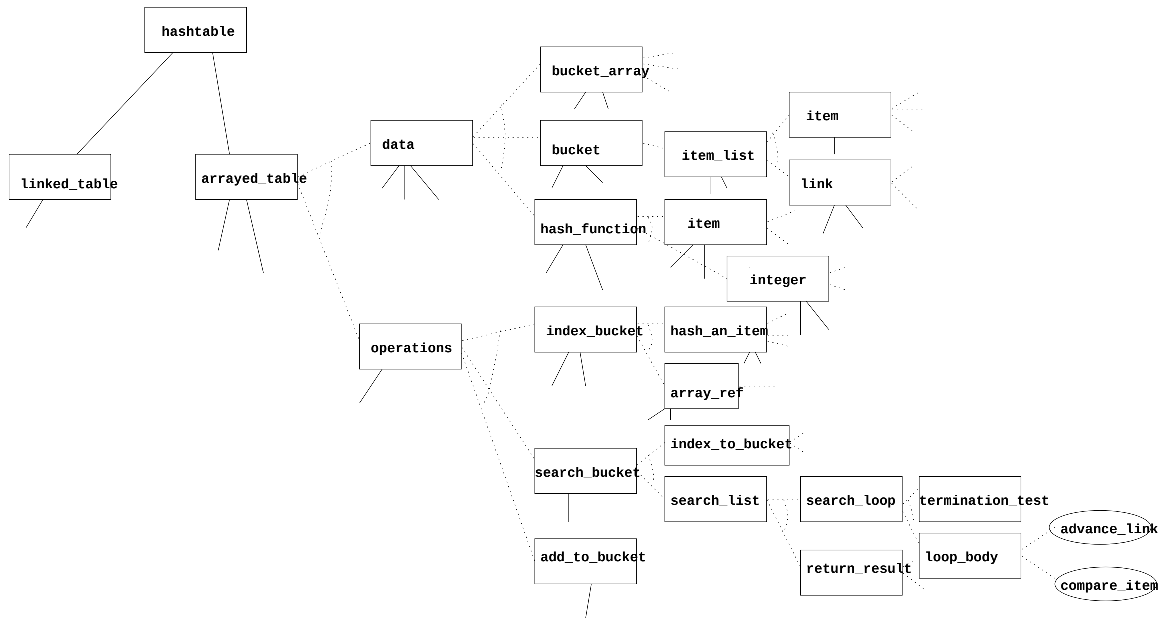

Figure 5: A portion of the granularity hierarchy for HASH-TABLE

In addition, our approach views the whole reverse engineering process as an iterative refinement process of model-construction similar to the process of knowledge-engineering/knowledge-acquisition. Granularity provides natural means to create a representation of the user's understanding at any stage and can form a basis for further analysis at the next stage.

The reverse engineering support system proposed here will have a large library of programming plans represented as granularity hierarchies. Figures 5 and 6 show a portion of the hierarchies for the programming patterns `HASH-TABLE` and `LINKED-LIST` respectively. Programming plans are of two types:

(i) Plans that represent an independent programming concepts meaningful in its own right. For example, the plan for `SORT` is independently meaningful and is useful to be recognised.

(ii) Plans that can be recognised as stereotypical patterns, but in and of themselves are not interesting. `ADVANCE-LINK` in a linked list traversal is one such example. Although `ADVANCE-LINK` may not by itself be of great use, it will be very crucial for recognising higher-level plans like the use of a `LINKED-LIST`.

The above distinction of plans into two categories is very useful since this provides a way of specifying where to start the search process. The hierarchies in the library of plans are always rooted at plans of type (i). Plans of type (ii) can be used to index other plans in the library. For example, the presence of the plan `ADVANCE-LINK` might be a very good indication of having a `LINKED-LIST` plan in the code. Hence, this can be used to trigger the search for a `LINKED-LIST`. A plan of type (i) can be indexed by more than one plan of type (ii), and a plan of type (ii) can also index more than one plan of type (i).

Once the system comes up with this granularity-based description, it can be used for constructing a domain model. The domain model serves three purposes:

(i) It serves as a record-keeping mechanisms where the user can record/modify her/his understanding of the system;

(ii) It serves as the evolving description (an active documentation) of the system that can be carried over to the next session and/or to the next person. It can be updated along with the software

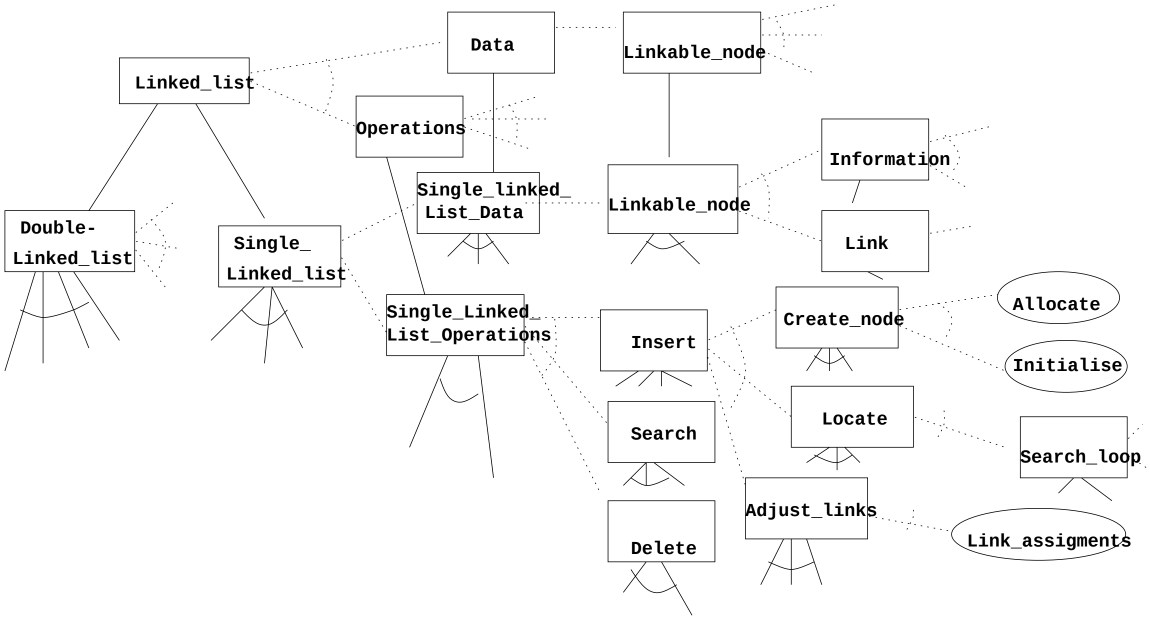

Figure 6: A portion of the granularity hierarchy for LINKED-LIST

so that the ongoing process of maintenance can be controlled in a structured way;

(iii) It provides a starting point for the next reverse engineering step which involves segmenting the code for recognition.

3.4 Example - revisited

Let us now reconsider the reverse engineering activity in the Mosaic software example. In the first stage, the user specifies that she/he is interested in the history mechanism of Mosaic. The user can use the granularity formalism to specify the domain at a coarse grained level. The system will come up with a set of all possible files which will have code dealing with aspects of user history. In our example the system can come-up with the following list of files:

```
globalhist.c
history.c
mosaic.h
```

The user then selects and/or rejects some of the items from this list of files. In the second stage, the system, using an approximate matching mechanism, would try to locate the presence of any of the relevant plans in these files. In the example above, the system would come up with the list of possible plans: `HASHTABLE`, `SORT`, `AGGREGATE-LOOP`, ...etc. In the next stage, the user can direct the system as to which of these plans are most relevant. The user can specify her/his intention using a Granularity-based Query Language (GQL). The user and the system iterates through the above steps until the user is satisfied with her/his understanding of the system. Figures 2 and 3 show the result of recognition on the code shown in Figure 1. Using this information, the user will refine the model of the system described in the granularity formalism. The process can be repeated again with this refined model as the basis for segmenting the code.

The granularity-based recognition tool will be a part of a general reverse engineering environment. Once the user constructs a granularity-based model of the software, it can be used in further steps for achieving other goals of reverse engineering. For example, once the parts of the code that are relevant to the task are identified and recognised, one can use techniques such as dependency analysis to see what other parts of the system will be affected by any changes to the recognised part.

4 Comparison With Other Approaches

There have been many efforts in reverse engineering and program understanding [21].

Biggerstaff's `DESIRE` system [3, 4] uses a domain model-based connectionist engine called `DM-TAO` to accomplish what he calls the *concept assignment* task. The concept assignment problem is basically determining which part of the program represents which domain-level concept. `DM-TAO` represents all the domain concepts as a network in which each concept is represented as a node and relationships between them as links. Each node contains information such as typical features that characterise the concept, its relationships to other concepts in the domain and informal knowledge such as the terminology the user might use to refer to this concept in the program. Biggerstaff's approach to assigning concepts to program fragments using a fuzzy matching mechanism has shown promising results for small programs (about 500 to 600 lines of code). One problem with this approach is that there is no simple way for a software engineer to guide the recognition process. It is very difficult to fully automate the problem of concept recognition for the large real-word systems.

Wills [27, 25] 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 cliches, 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 map to the same representation in flow graphs. The graph grammars are used as a means to recognise cliches in the program represented in plan calculus. Hence, the problem of program recognition becomes a graph parsing problem. Her system produces a description of a given program in terms of cliches in the library and the actual information from the program such as names of variables and procedures. It uses the concept of *overlays* [2] which map a higher-level specification of a cliche in terms of its implementation using lower-level cliches. Wills' system, called Recogniser, works correctly even if there is an overlapping implementation of two or more cliches. Overlapping implementation occurs when a piece of code is shared by two or more cliches. Since Recogniser does not work directly on the program text, it can recognise a cliche even if the code corresponding to a cliche is not contiguous at one location but spread across the whole program. Recogniser will skip any parts which cannot be recognised as part of any cliche.

Wills' method, though very comprehensive, has many limitations. The graph parsing approach will not handle global data structures very well. Flow graphs cannot represent recursive programs well. Also, there is no way a user can guide the recognition process.

Johnson's system, PROUST [11], 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 is only looking for semantic bugs. For each problem, there is a hierarchy of potential programming plans including buggy plans which will provide expectations as to what kind of bugs are possible in that problem. PROUST's approach, though useful for a tutoring system working in domains where the size of the program is very small (less than 100 lines of code), will be difficult to scale-up to any real-world software systems.

Quilici [23] describes a system that extracts design knowledge from C programs that is useful in translating them 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. 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.

[2] An overlay is composed of two plans and a set of correspondences between their parts.

Based on these observations, Quilici developed a mechanism to recognise plans in a program. He represents the plans in the library as a structure consisting of a set of *attributes* defining it, and a *recognition rule* that has a list of constraints and components. Each plan can index other plans in the library. An index consists of a set of test predicates and a pointer to a partial instantiation of the other plan. When the predicates are satisfied, the plan is instantiated. This kind of indexing mechanism allows testing for possible related plans that should be examined. This reduces the search space considerably. Plans are linked to one another through specialisation constraints. The plans in the library also contain a list of *implied* plans. Recognition of one plan implies some other plan(s) having been recognised.

The granularity-based representation proposed in this paper for use in reverse engineering captures all the specialisation constraints very well through the abstraction mechanism. It is more comprehensive and neatly tied to the other dimension of granularity, i.e., the aggregation dimension. Also, granularity-based recognition is more robust and works better in the presence of unrecognisable code. An index mechanism can be easily added to the existing granularity based recognition algorithm, by incorporating indices into the strategy objects. Also, the observers in granularity are very general and flexible to implement many other types of recognitions such as pattern-search based methods.

5 Implications and further work

We have proposed a new technique for program recognition and reverse engineering and a new method for developing reverse engineering tools based on the assumption that software is essentially a product of cognitive activity. There are many implications that can be drawn from this proposition. These implications can be classified into two categories, (i) implications for reverse engineering research, (ii) implications for granularity-based reasoning. They are discussed below.

5.1 Implications for reverse engineering research

Many problems in the process of program understanding and reverse engineering can be recast in terms of the new perspective of reverse engineering as a human cognitive activity.

Supporting human activity

There needs to be more emphasis on supporting the cognitive process of the human doing reverse engineering by developing methods and techniques for modelling this activity. We need to reorient the current methods that are preoccupied by an urge to compute formal properties of the programs (e.g., various metrics) towards the goal of supporting human understanding the software systems.

New tools for reverse engineering

There have been many tools produced to help reverse engineer and/or re-engineer software systems, but many of them are very limited. The principal reason why they are not more successful is that they do not deal with human oriented domain concepts. The new perspective will help produce tools that are more useful and usable by allowing the user to be “in the loop” in the process of model-construction. In addition, tools need to understand users’ needs and limitations.

5.2 Implications for granularity research

The field of AI has always been enriched by the domains in which the techniques have been tried. 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 problems will be generated by using techniques in new situations. Some of the implications for research in granularity are now presented:

Scalability of AI techniques

Most of the ideas and techniques in AI have been developed for toy domains. They work pretty well in a very small, narrow domains but many of them will fail when applied to any real world problem. Software reverse engineering provides an excellent field where many of the AI techniques can be stretched to the limit.

Three dimensional granularity

The original formulation of granularity by Hobbs [10] proposed only a single dimension of granularity. When these ideas are applied to the recognition problem, two different dimensions of abstraction and aggregation, have been discovered [7]. But in fact there seems to be three different dimensions, namely, (a) abstraction, (b) aggregation, and (c) approximation. The abstraction dimension described in [7] is really a combination of two different dimensions, abstraction and approximation. It will be interesting to investigate the ideas of granularity from the perspective of these three dimensions.

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 will come into play in the new setting. Also, it will become imperative to model the reverse engineering process so that effective cognitive support systems can be developed.

New issues

There are many potential new issues that arise out of the new perspective on 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. 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 crucial for human conceptualisation?

6 Conclusions

Reverse engineering is a crucial aspect of software maintenance. Recognising a program in terms of known programming patterns is essential to understand the program. Many efforts have been made in automatic program recognition, but most of them work in very small domains. This paper proposed a technique for program understanding using granularity based recognition. The approach assumes program understanding is primarily a cognitive activity and proposes a human support system for facilitating such an activity. The mechanism is very general and has the potential of scalability. It is also flexibly able to incorporate many of the techniques developed earlier by other researchers.

References

[1] Bhuiyan, S., Greer, J., and McCalla, G. Characterizing, rationalizing, and reifying mental models of recursion. In *Proceedings of 13th Meeting of the Cognitive Science Society* (1991), pp. 120–125.

[2] Biggerstaff, T., Mitbander, B. G., and Webster, D. Concept assignment problem in program understanding. In *Proceedings of Working Conference on Reverse Engineering* (Los Alamitos, California, 1994), R. C. Waters and E. J. Chikofsky, Eds., IEEE Computer Society Press, pp. 27–43.

[3] Biggerstaff, T. J. Design recovery for maintenance and reuse. *IEEE Computer 22*, 7 (1989), 36–49.

[4] Biggerstaff, T. J., Mitbander, B. G., and Webster, D. E. Program understanding and the concept assignment problem. *Communications of ACM 37*, 5 (May 1994), 72–83.

[5] Browne, J. C., and Johnson, D. B. FAST: A second generation program analysis system. In *Proceedings of 2nd International Conference on Software Engineering* (1977), pp. 142–148.

[6] Chikofsky, E. J., and J. H. Cross, I. Reverse engineering and design recovery: A taxonomy. *IEEE Software 7* (1990), 13–17.

[7] Greer, J. E., and McCalla, G. I. Formalising granularity for use in recognition. *Applied Mathematics Letters 1*, 4 (1988), 347–350.

[8] Greer, J. E., and McCalla, G. I. A computational framework for granularity and its application to educational diagnosis. In *Proceedings of IJCAI* (1989), pp. 477–482.

[9] Greer, J. E., McCalla, G. I., and Mark, M. A. Incorporating granularity-based recognition into SCENT. In *Proc. of the Fourth Intl. Conf. on Artificial Intelligence and Education* (Amsterdam, June 1989), pp. 107–115.

[10] Hobbs, J. R. Granularity. In *Proceedings of IJCAI* (Los Angeles, 1985), pp. 432–435.

[11] Johnson, L. W., and Soloway, E. PROUST:knowledge-based program understanding. *IEEE Transactions on Software Engineering 11* (1985), 267–275.

[12] Kozaczynski, W., Ning, J., and Engberts, A. Program concept recognition and transformation. *IEEE Transactions on Software Engineering 18*, 12 (December 1992), 1065–1075.

[13] Kozaczynski, W., Ning, J., and Sarver, T. Program concept recognition. In *Proceedings of 7th Conference on Knowledge Engineering and Software Engineering* (September 1992).

[14] Lehmann, F., Ed. *Semantic Networks*. Pergamon Press, 1992.

[15] Letovsky, S. *Plan Analysis of Programs*. PhD thesis, Department of Computer Science,Yale University, New Haven, CT, 1988.

[16] McCalla, G. I., Greer, J. E., Barrie, B., and Pospisil, P. Granularity hierarchies. *Computer Math. Applic,* Special Issue on Semantic Networks *23*, 2-5 (1992), 363–375. Also appears in *Semantic Networks*, F. Lehmann (Ed.), Pergamon Press, 1992.

[17] McCalla, G. I., Greer, J. E., and the SCENT Research Team. Intelligent advising in problem solving domains: The SCENT-3 architecture. In *Proceedings of ITS-88* (1988).

[18] Muller, H. A., and Klashinksy, K. Rigi: A system for programming-in-the-large. In *Proceedings of 10th ICSE* (1988), IEEE, pp. 80–86.

[19] Muller, H. A., Tilley, S. R., Organ, M. A., Corrie, B. D., and Madhavji, N. H. A reverse engineering environment based on spatial and visual software interconnection models. *Software Engineering Notes 17*, 5 (December 1992), 88–98.

[20] Ning, J. Q. *A Knowledge-Based Approach to Automatic Program Analysis*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Campaign, October 1989.

[21] Palthepu, S. Tools and techniques for reverse engineering and software maintenance: A survey. CMPT856 Term Paper, Department of Computer Science, University of Sasktatchewan, 1995.

[22] Paul, S., and Prakash, A. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering 20*, 6 (June 1994), 463–469.

[23] Quilici, A. A memory-based approach to recognising program plans. *Communications of ACM 37*, 5 (May 1994), 84–93.

[24] Rich, C., and Waters, R. C. *The Programmer's Apprentice.* Addison Wesley, 1990.

[25] Rich, C., and Wills, L. M. Recognising a program's design: A graph parsing approach. *IEEE Software* (January 1990), 82–89.

[26] Wills, L. *Automated Program Recognition by Graph Parsing.* AI Memo no. 634, Ph.D. Thesis, Artificial Intelligence Laboratory, MIT, The Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139, 1992.

[27] Wills, L. M. Automated program recognition: A feasibility demonstration. *Artificial Intelligence 45* (1990), 113–171.