Localization of Design Concepts in Legacy Systems

K. Kontogiannis R. DeMori M. Bernstein E. Merlo

McGill University
3480 University St., Room 318, Montréal, Canada H3A 2A7

Abstract

Complete automation of design recovery of large systems is a desirable but impractical goal due to complexity and size issues, so current research efforts focus on redocumentation and partial design recovery.

Pattern matching lies at the center of any design recovery system. In the context of a larger project to develop an integrated reverse engineering environment, we are developing a framework for performing clone detection, code localization, and plan recognition. This paper discusses a plan localization and selection strategy based on a dynamic programming function that records the matching process and identifies parts of the plan and code fragment that are most "similar". Program features used for matching are currently based on data flow, control flow, and structural properties. The matching model uses a transition network and allows for the detection of insertions and deletions, and it is targeted for legacy C-based systems.

1 Introduction

As its name implies, reverse engineering encompasses the set of activities which move from a lower, implementation-oriented level of abstraction to a higher, design-oriented level. When a successful software system is maintained and evolved for an extended period of time, original design documents become obsolete and design rationales are lost, so reverse engineering activities to reconstruct such information become critical for the continued viability of the software.

Reverse engineering techniques are used to achieve a variety of objectives from simple identification of system structure to the recovery of reusable components and design concepts. Defect detection is an important related activity.

Large software systems (1 MLOC) impose severe constraints on the complexity of analysis algorithms, so real-time, interactive design recovery is not a feasible objective. Therefore, alternative solutions must be devised.

One less ambitious objective is redocumentation, which is defined in [5] as "the creation or revision of representations of a subject system in order to improve the comprehensibility of the overall system. System representations are created from the source code alone and usually involve data and control flow properties".

In contrast, design recovery deals with the identification of higher level abstractions by attaching "meaning" to program segments [2]. These higher level abstractions may be instances of predefined plans, programming concepts, or abstract data types and operations.

Our research aims to develop *a)* redocumentation tools to focus design recovery efforts on particular subsystems, *b)* localization and recognition algorithms based on pattern matching techniques, and *c)* leverage obtained through integration with other tools [22] using a software repository [16].

We consider redocumentation to be a macroscopic activity or *reverse engineering in the large*. The objective is to obtain in useful insights about the entire system in a reasonable amount of time. For this type of analysis, research efforts focus on program data flow features that can be used to identify clusters of related components (modules). Currently, program features based on data bindings, common data resources, and software structure are under examination. We argue that *de facto* modules can be identified based on the number and type of shared data resources.

Information about modular structure obtained from our system can be exported to other tools for visualization, storage, or user refinement. In particular, user-defined system views [22] provide design information using the insights of the developer, based

*This work is funded by IBM Canada Ltd. Laboratory - Center for Advanced Studies (Toronto), National Research Council of Canada, IRIS Network of Centers of Excellence, and McGill University.

on a first-cut representation produced by our tool.

At the other end of the scale, we consider design concept localization, or plan recognition, to be a microscopic activity (*Reverse Engineering in the small*). More detailed insights about the organization and behavior of a system can be obtained on a smaller area of interest. In particular, we are interested in *(a)* recognizing specific models or plans in the code, *(b)* recognizing general design concepts, and *(c)* detecting highly similar code fragments (suggesting instances of code cloning). Programs are represented using annotated abstract syntax trees (ASTs) in an object-oriented environment.

Macroscopic and microscopic techniques work together to yield an effective synergy. Using Macroscopic techniques, "interesting" components are identified as candidates for further study. Once focus is narrowed to a small portion of the system, microscopic tools aid detailed design recovery.

The system currently runs in a distributed environment and performs clustering [22] and code localization [11], [15] on small to medium-sized programs (50-400 KLOC). Our ultimate objective is to work with large (1 MLOC) and very large (10 MLOC) systems.

Using these ideas, we are developing a Reverse Engineering environment in conjunction with the University of Toronto, the University of Victoria, and the Centre for Advanced Studies at IBM Canada Ltd. The environment incorporates several different tools which communicate through a software repository. Analysis results may be displayed, stored in the repository, or communicated to other tools for further analysis.

2 The Analysis Framework

In [5], four levels of abstraction for Reverse Engineering have been identified: *implementation, structural, functional, and domain.*

The implementation-level view examines individual programming constructs. For analysis at the implementation level, a program is typically represented as an abstract syntax tree (AST) [13], symbol tables, or plain source text [10].

The structural-level view examines structural relationships among the program constructs. Dependencies among program components are explicitly represented.

The functional-level view examines the relationship between program structures and their behavior ("function"). The rationale behind program constructs is investigated.

The domain-level view examines concepts specific to the application domain.

In this project, we have identified three analysis strategies that can be applied at each level of abstraction: *1)* structural analysis, *2)* behavioral analysis, and *3)* conceptual analysis. A structural analysis technique examines the structural elements of the program, without considering execution-time behavior. A behavioral analysis technique attempts to understand the semantics of the program by executing or simulating execution of the target system. A conceptual analysis technique uses subjective or heuristic information to discover concepts and relationships in the software.

Our efforts focus on performing redocumentation and design recovery for improving quality and enhancing performance by eliminating error-prone code, bad or unusual programming practices, and unnecessary redundancy. Examples of potential or actual defects include uninitialized variables, dead code, memory leaks, code clones.

3 Macroscopic Reverse Engineering

The first step in analyzing a large system is redocumentation, which allows immediately useful conclusions to be drawn by examining the source code itself. One approach to redocumentation is to present in a meaningful way control and data flow properties and information about structure. Such an approach can be executed in a reasonably short period of time even for large software systems and is a feasible starting point for performing semi-automatic user-assisted analysis. We have investigated redocumentation based on data flow properties and structural properties and we export results to a redocumentation tool [22] for further analysis and visualization.

3.1 Data Flow Properties

Our data-flow based redocumentation efforts have been focused on developing tools to perform system clustering using data bindings and common data references. A data binding [21] is a triplet *(p, q, x)* where p is a function that *updates* variable x and q is a function that *uses* x. A common reference is a triplet *(p, q, x)* where functions p and q both *use* or *update* variable x (not necessarily the *same* x), where the variable(s) x have the same name and data type. A common reference is a more relaxed relation than data binding because common references are based only on name and data type. On the contrary data bindings are computed based on global variables or parameters passed by reference.

A cluster is a set of functions which share a specified range of data bindings or common references. Such clusters can be stored in the repository and made available to other analysis tools. Figure 1 shows analysis results from a 50 KLOC C-program based on data bindings (upper part) and a listing of common references (lower part). Data bindings and common references provide valuable insights about system decomposition and allow the user to focus his analysis objectives on reduced portions of the system [21].

3.2 Structure Properties

A number of research teams are working on design recovery and redocumentation using the program's structural properties. In [19] a pattern language and a Non Deterministic Automaton mechanism is proposed to match expressions with code fragments. In [4] a rule-based system is used for identification of error prone structures (e.g. non void functions with no return statement). In [10] a text alignment program performs code clone detection based on textual information.

Structure-based redocumentation efforts focus on code localization using structural criteria such as language constructs used, keywords, and grammatical patterns. Code localization with these criteria can be used for identifying instances of code cloning and for performing system clustering. As with data-binding and data-reference clusters, structural clusters can be stored in the repository and made available to other analysis tools.

Clone detection is of particular interest as a key activity for both plan localization and defect filtering [4].

4 Microscopic Reverse Engineering

Our key objective for microscopic reverse engineering is to achieve user-assisted partial design recovery. The focus is on devising efficient program representation schemes and implementing fast algorithms for plan localization that can be applied to large software systems. Common plan localization algorithms used in existing applications include top-down goal agendas, depth-first search, best-first search, repeated traversals, and exhaustive search.

Plan localization algorithms must cope with problems due to syntactic variations, interleaved plans, implementation variations, overlapping implementations, and unrecognizable code [20].

Drawbacks in many existing applications include *(a)* complexity issues, and *(b)* failure to produce any results if perfect localization cannot be achieved. Plan localization is a problem which in several aspects relates to the problem of recognizing concepts in text. Segment localization based on features of a specific type (e.g., structure, keywords, data flow properties, control flow properties) becomes a key activity towards such a plan localization system.

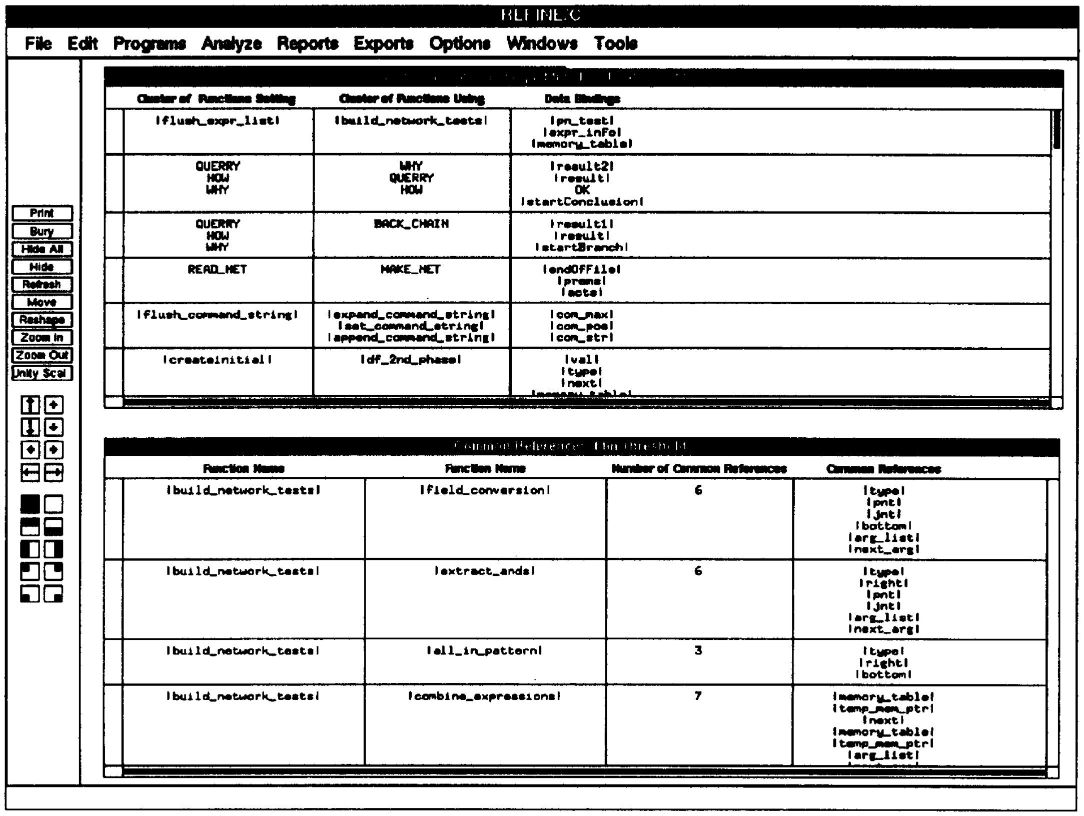
4.1 Program and Plan Representation

If reverse engineering technologies are to be ultimately successful, one of the central issues that must be addressed is the definition of a program representation formalism that encompasses design information at a higher level of abstraction than source code [17], [6], [9], [20], [18], [7].

Many researchers consider graph theory as the most useful mathematical formalism for representing the structure of a computer program. Some groups use directed graphs which represent low-level operations (loops, conditionals etc.) and define a context sensitive grammar for analyzing and interpreting these graphs [20, 3]. Others use directed graphs to represent more abstract properties of the program, such as condensed code descriptions or design specifications [1].

Most of these techniques use well established theories and formalisms borrowed from Formal Languages, Compilers and Parsers. Overall, the focus of Program Representation is on the development of mathematical formalisms and techniques which can facilitate:

1. representation of program functions in a more abstract way than source code,
2. the representation of the behavior of a program,
3. search techniques (borrowed from A.I, or graph theory),
4. ways to reflect information on the problem and the application domain,
5. user friendliness, in terms of how program design is presented to the programmer,
6. adaptability, in terms of how easily one representation can be transformed into another more abstract one (in case of reverse engineering) or less abstract (in case of forward engineering), and



Cluster of Functions Setting	Cluster of Functions Using	Data Bindings
|flush_expr_list|	|build_network_tests|	|pn_test| |expr_info| |memory_table|
QUERRY HOW WHY	WHY QUERRY HOW	|result2| |result| OK |startConclusion|
QUERRY HOW WHY	BACK_CHAIN	|result1| |result| |startBranch|
READ_NET	MAKE_NET	|endOfFile| |prems| |acts|
|flush_command_string|	|expand_command_string| |set_command_string| |append_command_string|	|com_max| |com_pos| |com_str|
|createinitial|	|df_2nd_phase|	|val| |type| |next|

Function Name	Function Name	Number of Common References	Common References
|build_network_tests|	|field_conversion|	6	|type| |pnt| |jnt| |bottom| |arg_list| |next_arg|
|build_network_tests|	|extract_ands|	6	|type| |right| |pnt| |jnt| |arg_list| |next_arg|
|build_network_tests|	|all_in_pattern|	3	|type| |right| |bottom|
|build_network_tests|	|combine_expressions|	7	|memory_table| |temp_mem_ptr| |next| |memory_table| |temp_mem_ptr| |arg_list|

Figure 1: A Cluster based on data bindings between functions (upper part) and a list of common resources between pairs of functions (lower part).

7. portability to a computer environment (be able to define data structures for encoding the formalism).

4.1.1 Program Representation

The foundation for our program representation scheme is an object-oriented annotated abstract syntax tree. The development of the Annotated Abstract Syntax Tree is a two-step process. First, the Refine[1] environment is used for the definition of a grammar and a domain model for the language of the subject system. The grammar is used for parsing, while the domain model defines the object hierarchies for the Abstract Syntax Tree nodes. In such a way an *If-Statement* and a *While-Statement* are defined as subclasses of the *Statement* class.

Second, the Tree is annotated with information on system structure (call graphs), data flow (data flow graphs), the results of previous analysis, and links to informal information. Annotations are produced either by analysis programs which are applied on the Abstract Syntax Tree level or by the parser.

The tool allows us to rapidly define new attributes for each class of AST node as our system evolves. Currently annotations produced by analysis programs include information such as, local variables, global variables, variables used, variables updated, functions called, points-to analysis and number of I/O operations (e.g. files opened, read/write statements). Annotations produced by the parser include information on object hierarchies between nodes of the AST, line numbers of code location, file names, declarations of variables and their corresponding references, include files, and data types. All annotations become part of AST tree and are available for any subsequent analysis algorithm to use.

[1]Refine is a Trademark of Reasoning Systems Corp.

In the current implementation, nodes of the AST are represented as objects in a LISP-based environment. Abstract Syntax Tree arcs and Abstract Syntax Tree annotations are represented as functional mappings from one node to another.

4.1.2 Abstract Data Types

Basic information on data types is defined as Abstract Syntax Tree annotations produced at parse time. These annotations provide information such as if a data entity is an Array-Type, Function-Type, Pointer-Type, and Floating-Type. At analysis level we must be able to provide the analysis algorithms with abstractions on the data type information provided by the parser and define/identify the basic operations on these abstract data types.

Currently we focus on the development of a domain model for a set of basic data types used in C. This domain model specifies an object hierarchy of Abstract Data Types and their corresponding operations. In such a framework, a *List-Data-Type* is a superclass of an *Array-Type* and a *Linked-List-Type.* A *Linked-List-Type* becomes a superclass of a *Double-Linked-List-Type* and of a *Single-Linked-List-Type.* The corresponding operations such as *Search-For-Element, Traverse, Empty-List?* become attributes which point either to Abstract definitions of programming plans or to specific nodes in the AST that implement the operation.

4.1.3 Plan and Concept Representation

A concept is a relevant abstraction of a set of ideas. For example, the concept of "linked list" abstracts ideas suggested by key words such as "first, "last", "next", "previous", "insert", and "append" [14]. Concepts can be from the programming domain, from the application domain, or universal. For example the traversal of a linked list is a concept which has associated with it keywords (e.g., "next", "new", "old", "previous"), a structure (e.g., a looping structure which is a superclass of a while, a for loop, or a repeat - until loop), an input abstract data type and an output abstract data type, (e.g., a listing structure which is a superclass of linked list structure, and of array structure, etc.). Actual data structures then become instances of the particular subclasses.

We are tentatively defining plans as user-defined portions of the annotated AST. The pattern matching and localization algorithm is used to match all code fragments that are similar to the model. The plan (model) can be stored for subsequent use and together with all the similar to it code fragments located can form a "similarity" class. This extends the power of the pattern matcher, as it allows more models to participate in the matching process.

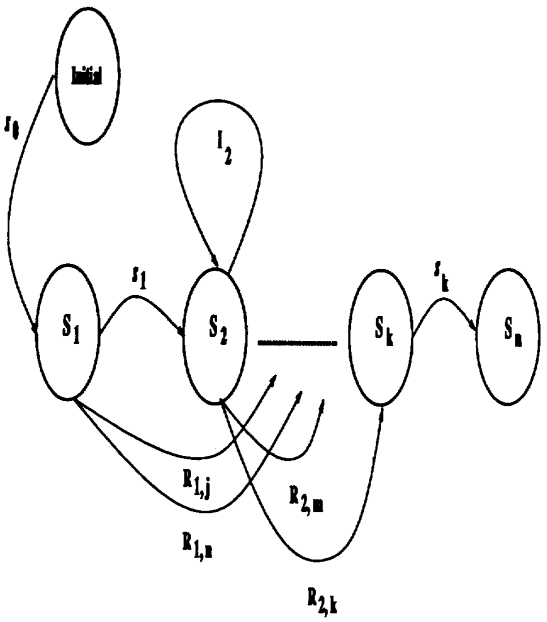


Figure 2: The Dynamic Programming model.

4.2 The Model Description

Comparison methods are used to perform simple plan instance recognition by computing a measure of distance or similarity between two simple programming plans. A simple programming plan is a plan that has no other plans interleaved. In our approach such simple plan instance recognition is performed by applying pattern matching on the annotated AST using program structure, keywords, and identifier names to represent meaningful software entities in specific contexts.

The model we consider is a sequential process in which plans (template) and programs (data) can be matched using dynamic programming. This process takes into account insertions and deletions but not substitutions or permutations (of data independent sequences of statements). A statement is the fundamental unit of the model.

In such a formalism the following notations are used:

- *A simple statement i :* S_i

- *A sequence of statements :* $S_1 S_2 ... S_n$
- *The composition* $\circ$ *of statements i, j :* $S_{i \circ j}$
- *A block statement taken from a sequence of statements* $S_1;S_2;..S_n$ *:* $S_{1 \to n}$
- *A vector of feature values for a block statement starting at position i and ending at position j* $(S_{i \to j})$: $\mathcal{E}(i,j)$

The sequential model is shown in Figure 2 where nodes represent states that correspond to statement sequences considered so far and arcs represent transitions between states. A state S_j contains vectors of feature values $\mathcal{E}(i,j)$, i $\in \{1,2,..j\}$. A vector $\mathcal{E}(k,j)$ contains feature values extracted in the program segment $S_{k \to j}$. A loop I_j on state S_j represents the insertion of arbitrary many g program elements at this state. A transition $R_{j,i}$ represents the fact that segment $S_j S_i$ was generated at state S_j, and then the process continues at state S_i.

4.3 The Comparison Process

The matching process is based on composition of *distances*. In this formalism, $D(\mathcal{E}(1,p), \mathcal{E}(1,j))$ is the distance between the first p elements of the program under analysis and the first j elements of the model. This distance is computed using *dynamic programming.*

The computation of such a distance is illustrated in Figure 3. The *x-axis* represents the input program in states and the *y-axis* the model or the programming plan.

Distance computation may take a number of forms according to what properties we want the distance function to have. For the sequential model we have adopted the following dynamic programming function:

$$D(\mathcal{E}(1,p),\mathcal{E}(1,j)) = Min \begin{cases} \Delta(\mathcal{E}(p-1,p),\mathcal{E}(j-1,j)) + \\ D(\mathcal{E}(1,p-1),\mathcal{E}(1,j-1)) \\ \\ I(\mathcal{E}(p-1,p),\mathcal{E}(j-1,j)) + \\ D(\mathcal{E}(1,p-1),\mathcal{E}(1,j-1)) \\ \\ C(\mathcal{E}(p-1,p),\mathcal{E}(j-1,j)) + \\ D(\mathcal{E}(1,p-1),\mathcal{E}(1,j-1)) \end{cases}$$

where, $D(\mathcal{E}(i,j),\mathcal{E}(k,l))$ is the the distance between the code fragment represented by statement positions i to j and the model or code fragment represented by statement positions k to l. Similarly $\Delta(x,y)$ represents the deletion cost for code feature vectors represented by x and y , I(x, y) the insertion cost, and C(x, y) the comparison cost.

The above Dynamic Programming function does not specify the program features used for computing the costs C, Δ, and I.

As the quality of the process depends on the program features selected for the comparison and not on the Dynamic Programming algorithm, we achieve different levels of accuracy as the features chosen change.

Currently we are experimenting with

(a) data and control flow properties (such as number of global and local variables, number of files opened, type of statements used, and number of functions called). Under this criterion two statements match if they have similar control and data flow properties, and

(b) operations on sets of used/updated variables combined with unification and substitution algorithms. In this category the matching criteria include *(a)* the *variables-used* criterion under which two statements match if they use the same variables (after unification and substitution), *(b)* the *variables-updated* criterion under which two statements match if they update the same variables, and *(c)* the *resources* criterion under which two statements match if one updates and uses a superset of variables used and updated in the other statement. Cardinality thresholds can be used in order to define similarity distances between statements.

Moreover, as program statements can be either simple or composite we introduce the idea of *nested* Dynamic Programming. In the case of simple statements the comparison process is based on program features for calculating model/program similarity vectors (e.g., number, names and data types of common variables used/set). The situation, though, is more complex when composite statements are involved. In such a case, the algorithm proceeds by starting in a recursive way a new matching session for the subcomponents of the original composite statements.

Different programming structures start different recursive matching sessions. For example, an *IF-THEN-ELSE* statement is represented as three statements corresponding to *(a)* the condition, *(b)* the *Then* part, and *(c)* the *Else* part. The *Else* part, in turn, may be a *While* statement. In this case the matching process will trigger a new Dynamic Programming (D.P) matching in which the *While* statement is represented as a *Condition* part and a *While-Body* part.

4.4 Partial Matching

The matching is based *(a)* on a sequential model allowing for insertions and deletions between a plan and a code fragment and *(b)* on the hierarchical definition of programming plans.

At model position q and code fragment position p the dynamic programming function D($\mathcal{E}$(1, p), $\mathcal{E}$(1, q)) gives the minimum distance corresponding to the generation by the first q positions of the model of a segment in the input program ending at position p.

As the domain model of the language allows for an hierarchical description of these programming structures, the nested D.P matching can take advantage of predefined representations for all these basic language constructs (e.g., blocks, iterative statements, conditional statements, expression statements) to guide the matching process, and create the appropriate expansions needed for nested D.P matching.

The object oriented paradigm used for program representation (e.g., the domain model and the corresponding Annotated AST) has been extended to include an hierarchical description of programming plans.

As an example of the use of the abstract data type hierarchy, consider the TRAVERSAL plan that may be applied to a variety of relevant data types such as arrays or linked lists.

The hierarchical definition of plans allows for imperfect or partial matching between different object classes. This has been achieved by extending the domain model for the language constructs, adding new object classes and object hierarchies for data types and common programming plans. Moreover, new object classes can be defined at run time in order to assist the user to specify his own plans. For example, application specific concepts and their corresponding code fragments can be represented as user-defined object classes in the domain model.

Plan recognition in terms of user-defined concepts is only one aspect of microscopic reverse engineering. In this framework we would like not only to recognize a plan against a code fragment but also use a code fragment as the *model* and find other code fragments that match it. The gain is that *(a)* we create similarity clusters of instances of a plan (even if this plan is not yet defined) and *(b)* we facilitate system modularization and code reuse.

For a more concrete example, consider the following instances of the TRAVERSAL plan for walking through a linked list:

```
P1:  fact_ptr = get_next_fact(((void *)0));
P2:  while
P2.1  (fact_ptr != ((void *)0))
        {
P2.2:       show_fact("wdisplay",fact_ptr);
P2.3:       cl_print("wdisplay","\n");
P2.4:       fact_ptr = get_next_fact(fact_ptr);
        }
```

and

```
Q1:  test_ptr = (new_fctn_args->arg_list);
Q2:  while
Q2.1: (test_ptr != ((void *)0))
        {
Q2.2:       total *= numget(test_ptr,"*");
Q2.3:       test_ptr = (test_ptr->next_arg);
        }
```

The nested dynamic programming matching for testing similarity between the code fragments P (used as model) and Q (used as input) is illustrated in Figure 3. Statements P_1 and Q_1 match under the ***variables-updated*** criterion and create the first binding {fact_ptr / test_ptr}. The second statement P_2 is a composite statement and triggers a new D.P matching, with the composite statement Q_2. The condition parts of statements P_2 and Q_2, $P_{2.1}$ and $Q_{2.1}$ match under the ***variables-used*** similarity criterion and by applying the previously defined binding {fact_ptr / test_ptr}. Similarly the while-bodies $P_{2.2}$ to $P_{2.4}$ and $Q_{2.2}$ to $Q_{2.3}$ match under ***variables-updated*** criterion at positions $P_{2.4}$, $Q_{2.3}$, after applying the binding {fact_ptr / test_ptr}. The nested D.P matching is illustrated in the lower part of Figure 3.

4.5 State of Practice

Within the framework presented above, we are working towards

- the selection of program features that allow for optimal performance of the pattern matcher
- the definition of methodologies to handle implementation variations between different instances of programming plans

Our objectives apply to three main directions

- Plan localization and recognition
- Code cloning detection

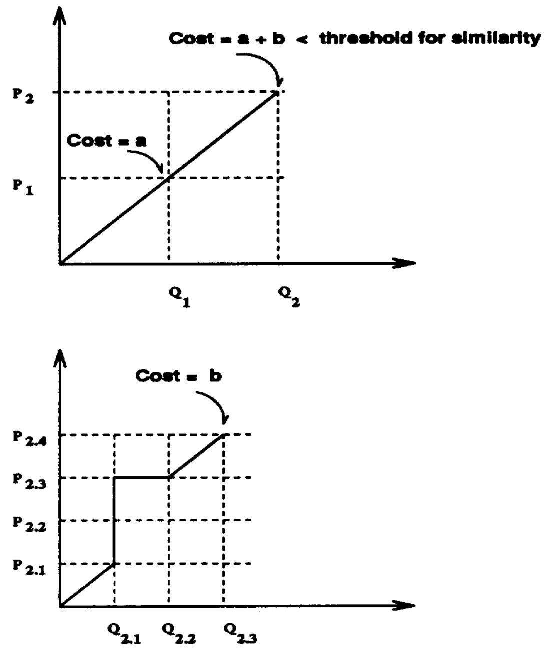


Figure 3: The matching process between two code fragments instances of the TRAVERSAL plan. The upper part of the figure illustrates the overall D.P matching while the lower part illustrates the nested D.P matching.

- Reuse and modularization

So far we have performed experiments using program features based on *(a)* measurements on control and data flow program properties (such as fanout, complexity measures, number of I/O operations etc.) and *(b)* operations on the sets of variables set/used per statement (e.g. subset / superset relations, cardinality of set intersections etc.).

Two examples include the application of the system on 50 KLOC and 40 KLOC C programs where we were able to detect a number of instances of code cloning. Applying simple pattern matching techniques as the ones defined above to the first program composed of approximately 750 functions distributed in 40 files, we obtained 68 clusters containing potential clones with approximately average size 3.6 functions per cluster. The second program analyzed consists of 233 functions distributed in 57 files. We obtained 9 clusters with the average size 2.5 functions per cluster. This system has been incorporated in a larger reverse engineering environment aiming for clustering and redocumentation of sizable C programs (400 KLOC) [11], [12].

Moreover, we are working on devising methodologies allowing for pattern matching between code fragments with implementation variations. We focus our efforts on two areas: *(a)* implementation variations due to different data types, and *(b)* implementation variations due to control and data flow. For the first category of variations we use an hierarchical description of basic data types used (e.g., double linked lists, linked lists, arrays, etc.). In such a framework a *linked list* and an *array* are subclasses of a *list* structure. The hierarchy of the data types is defined as an extension of the domain model of the language constructs. The second category is more complex and includes problems such as order of execution of statements and data dependencies between program components. In such a case the model described above fails because it handles only insertions and deletions of program statements between the model and the input. Currently, we are experimenting with dynamic programming models that maintain a set of statements per state instead of just a statement per state. A state then is defined by a collection of program statements that are accessible at this program point based on data and control dependencies. Once a match has been achieved at state i of the input then the corresponding matching statement is deleted from the set at state i as well as from all the other sets that contain it. For example (assuming normal conditions), statements $i = 1; j = 2$ can be interchanged without affecting program behaviour. The sequential model will not recognize such a variation but the extended model will succeed. The key point though is to recognize statements which do not have data dependencies and to update the states of the model in order to contain as accurately as possible the correct sets of control and data flow independent statements. For this reason we are working on incorporating compiler technology and flow analysis techniques [8].

5 Integration

The integration of different environments and tools in a distributed environment can be achieved by allowing a local workspace for each individual tool in which specific results and artifacts are stored. A translation program generates appropriate images of objects from each local workspace to the central repository, and vice versa. The central repository is responsible for *normalizing* these representations, making them available to other tools, and linking them appropriately with the other relevant artifacts already stored

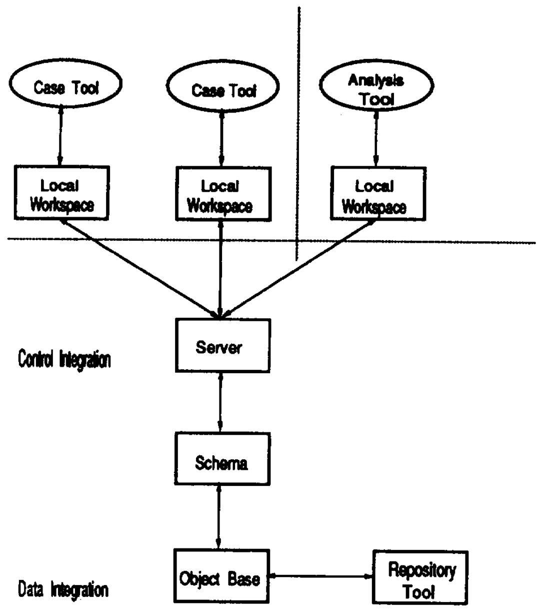


Figure 4: The architecture of a distributed reverse engineering environment. Dashed lines distinguish different machines, or computing environments.

in the repository (e.g. the corresponding nodes in the AST).

A system architecture for such an integrated reverse engineering environment which is currently investigated by the participants of the project[2] is shown in Fig. 4.

6 Conclusion

Within the framework presented in this paper, we are working towards the development of a practical reverse engineering environment that supports not only the identification of system components and dependencies but also the extraction of higher-level system abstractions, including design and requirements information.

Insights about low-level abstractions can be gathered about the overall system (*reverse engineering in the large*), while detection of higher-level concepts requires a more narrow range of focus (*reverse engineering in the small*). The tools we are developing are designed to work under the direction of the maintenance programmer, who can use the results of analyses performed to direct further queries.

[2]University of Toronto, Dept. of Computer Science - University of Victoria, Dept. of Computer Science - McGill University, School of Computer Science, and IBM Canada Ltd.

Comparison between plans and code fragments is achieved by using dynamic programming and distances between program features. Comparison criteria for this level of plan recognition include structural similarity, control flow similarity, and relations defined on uses and definitions of variables (e.g., the type of variables used, updated, or passed as parameters, preconditions, data bindings, and common references).

We are currently using the techniques described in this paper to perform *a)* system clustering and *b)* clone detection on small to medium sized programs (50-400 KLOC).

Our further research objectives include introduction of compiler technology (alias analysis, static data flow analysis etc.), refinement of the code features used to identify plan instances, refinement of the plan representation formalism, and experimentation on larger-scale commercial systems in conjunction with IBM Canada Ltd. Center for Advanced Studies.

References

[1] Arango, "Maintenance and Porting of Software by Design Recovery," *IEEE Conf. on Software Maintenance*, 1985, pp. 42.

[2] Biggerstaff, T. J., "Design Recovery for Maintenance and Reuse," *IEEE Computer*, July 1989, pp. 36.

[3] Bush, "The Automatic Restructuring of COBOL," *IEEE Conf. on Software Maintenance*, 1985, pp. 35.

[4] Buss, E., Henshaw, J., "Experiences in Program Understanding," *In Proceedings of CASCON '92*, IBM Centre for Advanced Studies, November 9 - 12, Toronto, Vol.2, pp. 157 - 189.

[5] Chikofsky, E.J. and Cross, J.H. II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, Jan. 1990, pp. 13 - 17.

[6] Engberts, A., Kozaczynski, W., Ning, J., "Automating Software Maintenance by Concept Recognition Based Program Transformation," *In CSM'91 : Proceedings of the 1991 Conference on Software Maintenance*, October 1991.

[7] Fickas, S., Helm, R., B., "Knowledge Representation and Reasoning in the Design of Composite

Systems", *IEEE Transactions on Software Engineering*, Vol. 18, No.6, June 1992, pp. 470-482.

[8] Hendren, L., Donawa, C., Emami M.,, Gao, G., Justiani, Sridharan, B. "Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations" *LCPC5, pp. 261 - 275*

[9] J. Hartman, "Understanding Natural Programs Using Proper Decomposition" *13th International Conference on Software Engineering, 1991, Austin, Texas, pp. 62-73.*

[10] Johnson, H., "Identifying Redundancy in Source Code Using Fingerprints" *In Proceedings of CASCON '93*, IBM Centre for Advanced Studies, October 24 - 28, Toronto, Vol.1, pp. 171 - 183.

[11] Kontogiannis, K., Bernstein, M., Merlo, E., DeMori, R., "The Development of a Partial Design Recovery Environment for Legacy Systems" *In Proceedings of CASCON '93*, IBM Centre for Advanced Studies, October 24 - 28, Toronto, Vol.1, pp. 206 - 216.

[12] Kontogiannis, K., Tilley, S., DeMori, R., M uller, H., " A Reverse Engineering Environment for Legacy Systems" *ICSE 16, Workshop on Software Engineering and Artificial Intelligence*, Sorrento, Italy, May 16 - May 22 1994.

[13] Kotik, G.B. and Markosian, L.Z., *Automating Software Analysis and Testing Using a Program Transformation System*, Reasoning Systems Inc., 1989.

[14] Merlo, E., McAdam, I., De Mori, R., "Source code informal information analysis using connectionist models," *In Proceedings of 13th. International Joint Conference on Artificial Intelligence*, Vol. 2 pp. 1339 - 1344.

[15] Merlo, E., De Mori, R., Kontogiannis, K., "A Process Algebra Based Program and System Representation for Reverse Engineering," *In Proceedings of Second Workshop on Program Comprehension*, July 8 - 9, Capri, Italy, pp. 17 - 25.

[16] Mylopoulos, J., "Telos : A Language for Representing Knowledge About Information Systems," *University of Toronto, Dept. of Computer Science Technical Report KRR-TR-89-1*, August 1990, Toronto.

[17] J. Q. Ning, M. T. Harandi, "Knowledge-Based Program Analysis," *IEEE Software, January 1990, pp74-81*

[18] Parnas, D., L., "Predicate Logic for Software Engineering", *IEEE Transactions on Software Engineering*, Vol. 19, No.9, September 1993, pp. 856-862

[19] Paul, S., Prakash, A., "Generating Programming Language-based Pattern Matchers" *In Proceedings of CASCON '93*, IBM Centre for Advanced Studies, October 24 - 28, Toronto, Vol.1, pp. 227 - 243.

[20] Rich, C. and Wills, L.M., "Recognizing a Program's Design: A Graph-Parsing Approach," *IEEE Software*, Jan 1990, pp. 82 - 89.

[21] Selby, R., Basili, V., "Analyzing Error Prone System Structure" *IEEE Transactions on Software Engineering*, vol 17, No. 2, February, 1991, pp. 141 - 152.

[22] Tilley, S., Muller, H., Whitney, M., Wong, K., "Domain-retargetable Reverse Engineering", *In CSM'93 : Proceedings of the 1993 Conference on Software Maintenance*, September 1993, pp. 142-151.