The Development of a Partial Design Recovery Environment for Legacy Systems *

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

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

Abstract

Computer Assisted Program Understanding systems take input primarily in the form of source code and produce output representing system concepts in some useful form. This paper discusses the development of a program design recovery environment based on structural and behavioral recognition of programming plans. Our research investigates program and plan representation methods and the issues related to the detection of code fragments using pattern matching techniques. In particular, we consider the integration of diverse tools allowing selection from a choice of strategies. Our investigation focuses on techniques to allow partial design recovery when complete recognition is not feasible. Finally, we discuss the underlying process paradigm, called "Goal-Question-Analysis-Action".

1 Introduction

Reverse engineering is concerned with the development of tools and techniques for understanding unfamiliar code to facilitate system maintenance [5]. Design recovery, a fundamental task for software maintenance and reverse engineering, refers to the process in which a program is analyzed in order to be understood as a whole with respect to

- functional specifications,
- input parameters,
- expected output,
- performance, and
- the software and hardware environment in which the system runs.

The objective is to discover features such as the organization of program structure (how procedures or submodules are organized), run-time functionality of the program (how and in what order modules are invoked), parameter passing, aliases, side effects, and meaningful system abstractions.

The primary input is source code which is represented usually as a flat text file. The file may contain a variety of forms of *informal* information such as comments, I/O commands (e.g., *print* statements), indentation, and meaningful variable names. Unfortunately, informal information provides insufficient data for analyzing complex program features. It is necessary to apply other techniques which operate on the level of abstract syntax trees (AST) or other program representation schemes.

In the past few years several research groups have focused their efforts on the development of tools and techniques for program understanding and program restructuring. Research issues include the development of formalisms for representing program structure, control and data flow analysis, and techniques for visualizing program execution. Design recovery research has been concentrated mostly on program representation and plan localization.

For real applications, however, additional issues must be addressed. Practical problems include the limited number of programming plans (templates), the difficulty of analyzing unstructured code, and the complexity of plan localization algorithms. Together, these problems make program understanding applications rigid and unable to cope with large-scale systems. We take an approach in which methods and tools for design recovery are conceived based on a set of strategic but specific objectives.

We introduce the idea of *Goal Directed Design Recovery* in which the representation method, the level

*This work is funded by IBM Canada Ltd. Laboratory - Center for Advanced Studies (Toronto). Parts of this paper appear at "Program Representation and Behavioural Matching For Localizing Similar Code Fragments " paper of CASCON '93 Proceedings.

of abstraction, the appropriate analysis tool, and the control strategy are dictated by the objectives and program attributes that the maintainer wishes to recover. Depending on the particular design recovery objective, specific techniques include

- program flow analysis,
- program equivalence relations (using formal methods), and
- knowledge-based techniques (heuristics for complexity reduction, domain knowledge).

At lower levels of abstraction, the analysis process examines features derived directly from the static code structure. Issues considered at this level of abstraction include problems such as syntactic variations, implementation variations, overlapping implementation, unrecognizable code (or partial recognition), and diffuse structures. In general, they are issues regarding the identification of relationship among code fragments.

At higher levels of abstraction, behavioral analysis is used to recognize abstract functionalities in program fragments. Approaches for understanding behavioral aspects of a program are derived primarily from language semantics. Denotational and operational semantics provide a formal description of program behavior and can be used as a basis to represent and recognize semantic cliches.

In this research project a number of different reverse engineering tools and methodologies must be integrated into a single reverse engineering environment. The environment must be able to

- perform clustering,
- represent the program's structural relationships,
- allow for the definition of useful metrics, and
- perform selection operations based on user-defined criteria.

This yields a number of research issues that must be addressed, including the development of program representation schemes, the development of comparison algorithms, the development of repositories for software components, and the integration of user-interface technology to browse, navigate, and search large collections of software artifacts.

Within this framework, we investigate techniques for performing partial program design recovery. Because of the complexity of the problem, partial design recovery is a more realistic objective than completely automated design recovery and can be very useful when used by an experienced maintainer. The techniques we investigate focus on three major issues:

1. program representation schemes using language semantics and information obtained from the program's abstract syntax tree;
2. comparison methods based on structural and behavioral program properties; and
3. interface programs for integrating different reverse engineering tools and environments in one operational system using a common software repository.

In the following sections we discuss the structure and functionality of such an integrated reverse engineering environment and we investigate the relevant research issues.

2 An Integrated Reverse Engineering Environment

Reverse engineering is a process that produces high-level descriptions of software from lower-level representations. Usually this means extracting design from a program's source code. Once design decisions are recognized, they are then organized into a form that is convenient for the developer. This process requires the integration of different technologies, environments, and analysis tools. In developing an integrated reverse engineering environment, it is necessary to address issues of

1. program representation,
2. definition of algorithms and methodologies to extract system abstractions,
3. design of software repository systems,
4. user-interface technology to browse, navigate, and search large collections of software artifacts, and
5. the definition of process models for reverse engineering.

Current reverse engineering research focuses on the development and integration of several different techniques and methodologies. There exist a few reverse engineering environments on the market and in research laboratories which support reverse engineering functions that are limited in scope to syntactic and lexical analysis of the source code. The objective in our research is to extend the range of available approaches for reconstructing design requirements. In particular, we are considering

- program abstraction schemes,
- approximate pattern matching (with respect to stored code descriptions),
- interactive sessions with programmers,
- graph theoretic properties [14], and
- repositories for storing software components and results of software analysis.

Techniques based on compiler technology, such as parsers, flow analysis methods, slicing, and dicing are used to build abstract syntax trees and for performing a number of analysis tasks (e.g., constant propagation analysis, value range analysis).

Database technology is used to develop software repositories that are used for storing code descriptions, documentations, test data, results of previous analysis, and other relevant information.

Reverse Engineering environments such as REFINE [1] are used to provide a prototyping framework and a programming environment for developing reverse engineering applications. These environments can be very flexible, incorporating a variety of programming paradigms (such as object oriented, logic, and functional programming) and facilitating the construction of appropriate user interfaces.

Finally, pattern matching technology is used to detect common programming plans in source code fragments.

Within this framework we are working towards the development of a reverse engineering environment that supports not only the identification of system's components and their dependencies but also the extraction of system abstractions, including design and requirements information.

Some of the issues that must be address when designing a reverse engineering environment include

- communication among the different tools,
- definition of recoverable design features and their approprate level(s) of abstraction,
- development of representation schemes for highlighting software attributes, and
- the development of appropriate comparison techniques.

Abstract Syntax Trees provide the basic structure for program representation because they serve as inputs of a number of flow analysis algorithms and can be annotated with higher-level semantic information.

[1]REFINE is a trademark of Reasoning Corp.

The process of discovering and identifying system abstractions involves the development of a mental model describing the function of the system at different views. In [8] four different levels of program views are identified:

1. Implementation, represented as an Abstract Syntax Tree (AST) and a symbol table of program tokens;
2. Structural, which gives an explicit representation of the dependencies among program components,
3. Functional, which relates parts of the program by their functions and shows logical relations among them; and
4. Domain, which replaces items in the functional view by concepts specific to the application domain.

The implementation-level program view in our system is achieved by the use of a powerful parser/generator [12] which creates an AST stored as a collection of objects in a software repository. The system includes a very high level language (VHLL) for querying and updating the object base and for manipulating the tree.

The structural view is implemented using tools [14] which:

1. parse the target program to extract relevant system components and dependencies, storing them in a repository,
2. allow the user to generate hierarchies of subsystems,
3. determine interfaces among the subsystems, and
4. evaluate subsystem structures using established software engineering principles.

The functional view can be achieved by using language semantics to annotate AST representations. Examples of contributions from language semantics include denotational semantics [19, 20] and operational semantics [10] [16]. Denotational and operational semantics have been quite effective in achieving mathematical understanding of the dynamic behaviour of programs.

The domain-level view can be achieved by adding specific annotations to the AST using project-specific semantics (such as programming standards) and domain-specific information such as integrity constraints.

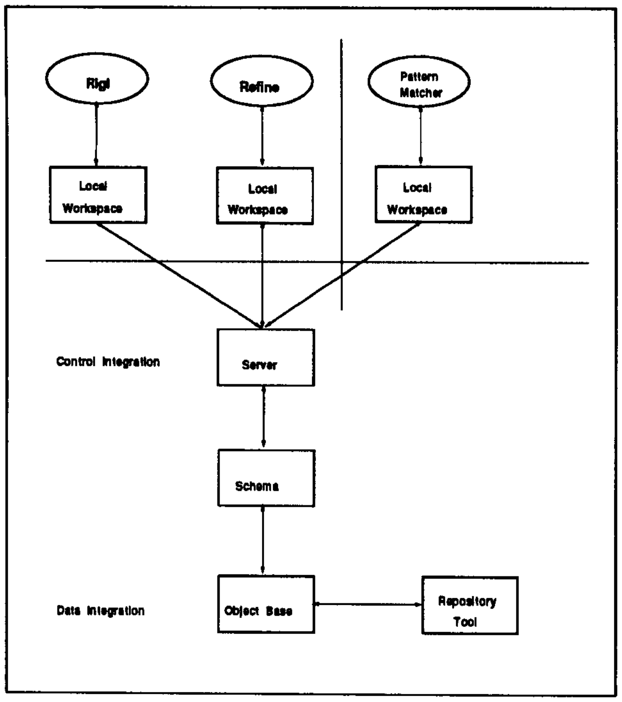

Figure 1: A possible architecture for a distributed reverse engineering environment. Dashed lines distinguish different machines, or computing environments.

At lower levels of abstraction, the focus is mainly on structural recognition and uses information derived from the code. Questions asked at this level include analysis of resource usage between modules (e.g., global variables, data structures, actual/formal parameter lists, files) and the identification of dependencies between program components (e.g., preconditions/postconditions, data dependencies). Structural recognition involves the analysis of control and data flow properties, system organization, and data dependencies.

At higher levels of abstraction, behavioral recognition is used to recognize abstract functionalities in program fragments. Code features for behavioral recognition include the investigation of the *effects* of a computation in terms of initial and final values induced, *how* a computation is performed in terms of transition systems, and the definition of meaningful relationships between different program parts (e.g., which computations affect which program parts). Behavioral recognition implies the analysis of the semantics of the exchanged resources between program modules (e.g., how exchanged resources are used, what effects they have), the operations which change the imported resources into their exported values, and, finally, the concepts, relationships, and actions that occur in a module.

From the preceding discussion, it is evident that a number of different tools and environments must work together order to produce the effect of a *Goal Directed Design Recovery Environment.* Moreover, we would like to have the whole environment distributed across different machines and available to multiple users. These requirements suggest a system architecture in which each tool has its own local workspace and all communicate through a central software repository that stores the results of analysis, design concepts, links to graph structures, and any other information needed for the different tools to operate. A possible system architecture for such an integrated reverse engineering environment is shown in Fig. 1 [2].

Finally, the nature and the dynamics of the process used for design recovery suggest a model that is based on a *Goal-Question-Analysis-Action* paradigm. This strategy focuses on the improvement of the software process by considering the specific project goals and environments. In such a way, project objectives and the domain specific information in the development environment guide the selection of appropriate models, methods, and tools in the software process.

In this model, the maintainer operates within a framework of top-level goals. These goals provide the framework and the justification for a series of questions that must be addressed to satisfy the top-level goals. Specific goals give rise to specific questions which require the use of specific analysis strategies and tools. Once the strategy and tool have been selected, an action is recommended to achieve the goal. The maintainer may require additional information and can set new goals, reapplying the whole model as shown in Fig. 2. Some relationships between *Goals, Questions, Analysis Tools,* and *Actions* are shown in Fig. 3.

3 Partial Design Recovery

In order to recover the design of a program, several issues must be addressed.

1. A code representation formalism must be chosen. This formalism should allow for representing both structural and behavioral code features and for the application of a different of analysis algorithms.

[2]This architecture was suggested by the participants of the *Program Understanding project* (University of Toronto, University of Victoria, McGill University, and IBM).

2. A comparison algorithm must be devised to match the source code representations with programming plans that represent commonly used algorithms and programming structures. Comparison algorithms depend heavily on the program and on the selected representation method and they do not always involve simple pattern matching.

3. A top-level control strategy must be defined. Top-level control methods focus on techniques to select program parts and programming plans for comparison in order to achieve plan instance localization.

Partial recognition deals with the problem of recognizing plan instances even when these plans are interleaved with other types of information in the code or are scattered throughout the program. Multiple, failed, or incomplete plan recognition must be taken into consideration. Multiple recognition occurs when a single programming plan matches more than one program part. Ambiguities can be resolved using needs, domain knowledge, or other external information. On the other hand, failed recognition should produce failure information explaining the cause of failure. However, the most common case occurs when no success or failure can be proven. In this case incomplete bindings should be produced for explanation and control.

3.1 Program Representation

One of the central issues to be addressed for design recovery is the definition of a program representation formalism that encompasses design information at a higher level than the source code. Most of the research approaches focus 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 behaviour 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

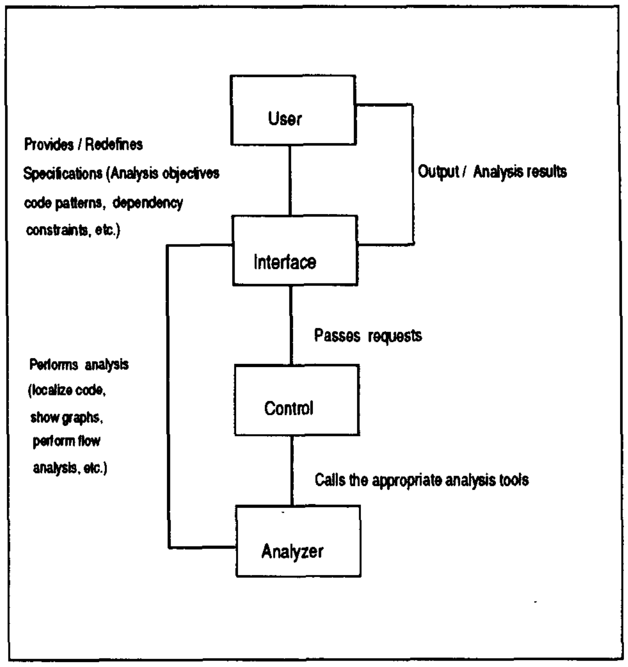
User
Provides / Redefines Specifications (Analysis objectives code patterns, dependency constraints, etc.)
Output / Analysis results
Interface
Passes requests
Performs analysis (localize code, show graphs, perform flow analysis, etc.)
Control
Calls the appropriate analysis tools
Analyzer

Figure 2: The overall functionality of a Reverse Engineering Environment using the "Goal-Question-Analysis-Action" paradigm.

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

Most approaches use internal representations that are based on structural and lexical properties, but it is also necessary to find representations based on behavioral code properties. A popular internal representation of code is the Abstract Syntax Tree (AST) which is *(a)* language independent *(b)* represents the functionality of the system as a whole, and *(c)* can be used in a a variety of software engineering activities.

Our approach distinguishes between structural and functional code recognition. For structural code recognition AST program representations are considered. For functional code recognition, representations based on language semantics are investigated. This paper focuses on structural code recognition in which ASTs are 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 defined as needed, and thus we maintain the capability to change or add annotations as the system evolves. Moreover, Annotated Text (AT) representations of the code can be created using such a schema. Annotated Text representations link source code fragments with meaningful concepts, comments, and abstractions so that design recovery can be assisted.

The main idea is to parse source code, creating an Abstract Syntax Tree that uses a domain model to show the structure and the hierarchies between the different language constructs. Once such an AST is created [4], analysis can be performed to annotate the nodes of the tree. Annotations take the form of mappings from AST nodes to other AST nodes or other entities. The REFINE environment is used to create such an Abstract Syntax Tree and its corresponding domain model. The resulting AST is stored in an object base and can be retrieved using a query language and a specialized programming language. Abstract Syntax Trees have been used extensively [1] for representing structural properties and serving as a basis for a several analysis methods.

Examples of reverse engineering environments using ASTs include the RECORDER system [3], the REFINE system [12], and the Programmer's Apprentice Project [18].

Once the representation of the basic components of a program by plans, clichés, or other formalisms has been studied, representation and comparison methods for controlling plan detection in complex applications must be considered.

3.2 Comparison Methods

Comparison methods are used to perform simple plan instance recognition by proving equivalence or showing a partial order relationship 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 will be performed by applying equivalence relations on some behavioral representation of the program.

Plans can be described at different levels of abstraction. The most abstract level is the one corresponding to the most concise description of a very complex system. With new representations, plan fragments at higher levels of abstraction could be detected and described. The most common plan-code comparison methods include:

- simple pattern matching,
- similarity metrics,
- graph matching, and
- body structure.

Some systems (e.g., PROUST) [11], [7] match syntax trees with syntax tree templates. A plan matches a program statement if its unified template matches the statement's syntax tree and its constraints and subgoals are satisfied. TALUS [15] compares student and reference functions by applying a heuristic similarity measure. In CPU [13], programs are represented as lambda calculus expressions and procedural plans. Comparisons in CPU are performed by applying a unification and matching algorithm on lambda calculus expressions. In UNPROG [9], program control flow graphs and data flow relations are compared with the programming plan's control flow graph and data flow relations. Quilici [17] matches frame schema representations of C code and if they structurally match then data flow graphs are compared too. GRASP [21, 22] uses attributed data flow sub-graphs to represent programs and programming plans. Comparisons are performed by matching subgraphs and by checking constraints involving control dependencies and other program attributes.

We consider two types of comparison methods: structural and behavioral. In structural comparison, the dominant program representation scheme is the Annotated AST. The objective is to locate code fragments based on structural/lexical properties, control and data flow properties, and communication properties.

Comparison criteria for this level of design recovery include structural similarity, control and data flow similarity, and relations defined on uses of variables (e.g., initial and final values, uninitialized data, value ranges).

In behavioral comparison, the dominant representation scheme is again the Annotated Abstract Syntax Tree emphasis placed on the annotations representing the *effects* of a computation and*how* a computation is performed. The objective is to locate code fragments based on equivalence and partial order relations defined over Annotated Abstract Syntax Trees and on logic relations that structural properties of language constructs with their corresponding semantics. Information obtained from Annotated Abstract Syntax Trees, language semantics (denotational, operational), the domain model, flow analysis results, and slicing techniques can help for perfoming behavioral comparison of simple programming plans and code fragmennts.

3.3 Top-Level Control

Program parts and programming plans represented at higher levels of abstraction are selected using a top-level control strategy and are used as input to a comparison module. The output of such comparisons are

Goal	*Question*	*Analysis*	*Action*
Code Correction	Data Type Mismatch	Type Inferencing	Update declarations
Code Performance	Detection of Inefficient Code	Refine programs	Normalize and Replace Code
Code Correction	Memory Allocation	Flow Analysis	Eliminate Memory Leaks
Code Performance	Equivalent Code	Language Semantics , Pattern matching	Perform Abstractions
Code Performance	Find Specific Algorithms	Language Semantics, Pattern matching	Perform Abstractions
Code Correction	Unitialized data	Flow Analysis	Initi alize Data
Code Correction	Change/Impact Analysis	Slices, Dependency Ana lysis	Localize Erroneous Code

Figure 3: Basic relationships for the Goal-Question-Analysis-Action paradigm

recognized concepts and program parts satisfying the specifications of a programming plan. Control can be guided by the needs of the particular application and the results of previous comparisons. Search algorithms are used to select from the program representation different programming parts for comparison. Bottom-up search strategies systematically select all program parts covering the program representation, while top-down search strategies seek single parts that can be used to satisfy a given expectation (subgoal). Programming plans and program parts are not always represented using the same formalism. Moreover, during the recognition process, comparisons must be performed between already recognized concepts and original program material. Hierarchical recognition control strategies focus on such multi-leveled representations and are used for compositional recognition where complex concepts are recognized in terms of their subcomponents.

Program decomposition can be used to guide the selection process. Performance is best when decomposition produces program parts that correspond to plans in the library. Program decomposition can be performed *a priori* before the selection process starts or in a *dynamic* way, based on previous recognition results and the current needs of the application as the selection process is performed.

For the control strategy, we focus on investigating an *island driven opportunistic search* in which search subgoals are set around some well-recognized point in the program. The idea is to use this positively identified point as an anchor and then try to satisfy subgoals around it.

Islands are obtained by starting with design decisions that have been positively (or with high evidence) identified and are of particular semantic relevance, and then proceeding outward, extending the analysis in both directions. Irrelevant and intermixed plans can be ruled out by allowing recognition gaps and partial recognition of plans.

As an example, consider intermixed plans or parts of plans which do not share data dependencies. Such plans can be distinguished irrelevant; the island-driven search will skip them and proceed towards what it considers the current goal to be.

Such an island-driven search starts by establishing a top-level goal which, in practice, is the satisfaction of some programming plan. The top-level goal is established by examination of the properties of the well-identified program part and the properties of the plans in the plan base. The search continues by trying to prove the existence in the code of other parts of the plan as these exist in the generic plan. Throughout the process, incomplete and partial satisfaction of the plan properties causes the search to jump in an opportunistic way in an attempt to satisfy whatever it can. Island-driven opportunistic search will not fail if the plan and the code representation do not match exactly. In this case the island- driven search will jump to another point with high likelihood of recognition and will start again from this point. The search will terminate when no further plan recognition can take place. Thus, even partially matched plans will be reported by such an algorithm. Failure occurs only when none of a plan's subparts can be positively recognized. The influence of constraints between nonadjacent fragments can be investigated in the case where other fragments are left as uninterpreted gaps. These concepts can be used to specify interactive tools that propose an interpretation of a fragment to the maintainer who may not be interested in every fragment or who may provide a personal interpretation of some uninterpreted fragments.

4 Research issues

Large legacy systems represent significant assets for the companies that use them. These systems have evolved over time and tend to require continuous maintenance. The definition of the objectives for a design recovery process is based both on a generic list of desired design attributes to be recovered (such as call graphs, data types, value ranges, control flow, data flow) and on domain-specific design attributes of a specific product or specific language.

Within the framework of this project, maintenance objectives were identified during a sequence of contextual interviews with the Program Understanding Group (PUG) at IBM's Centre for Advanced Studies. The top-level objectives of the Program Understanding Project (PUP) address problems related to

a) code correctness and;

b) performance enhancement.

The research issues which arise in such a framework and constitute the objectives of our work include:

- selection, incorporation, and use of an appropriate process model;
- development of code representation schemes and plan localization techniques that are appropriate for mechanical manipulation and allow for the analysis of structural and behavioral properties of the code;
- normalization and correction of erroneous code (e.g., removing dead code, simplifying or removing redundant expressions, localizing erroneous communication points); and
- integration and use of different technologies (flow analysis, language semantics, pattern matching, search and control strategies) in a reverse engineering environment.

Existing Program Understanding systems attempt to recognize plan instances by comparing program representations against programming plans. A common theme in these approaches is to use a *program representation* formalism, a *plan repository*, a *plan localization control strategy* , and a *comparison* algorithm.

The major problems associated with the program understanding systems built so far can be summarized as follows:

- *Repository completeness*: it is not possible to encode and store all possible plans occurring in an application, and usually the ones encoded are trivial cases (e.g., sorting algorithms, list traversals).
- *High complexity*: most understanders perform well only in small and medium-sized (approx. 5000 lines) programs. In large programs complexity makes design recovery an extremely difficult and time-consuming process. Difficulties are caused by interleaved and scattered plans and syntactic or implementation variations.
- *Structural and lexical matching*: in most applications, plan-program similarity or equivalence is determined by testing structural and lexical information from both the code and the plan. This causes a problem when the code fragment contains other plans, irrelevant statements, or when the plan is scattered among different parts of the application, because the structural and lexical-based representations of the code and the plan cannot be matched. Moreover, the behaviour of a program can not be encoded and represented.
- *Graph-based matching*: not all approaches use program representations that are based on structural and lexical information. Several program understanders use specific graph-based formalisms which incorporate data and control flow. Graph grammars are used to perform abstractions and to perform plan instance recognition. The problem in these applications is that graph transformations are usually very expensive to compute and graph pattern matching algorithms can have high complexity. This imposes a serious problem when large programs are analyzed because their corresponding graph representation can be very large and complex.

The approach taken in this project focuses on minimizing the drawbacks of other existing approaches. Below, we examine the basic views that we have adopted and the points of difference with respect to existing approaches.

- *Use of a development strategy.* In our approach we use a strategy for performing a Reverse Engineering (Design Recovery) task. This strategy was inspired by the *"Goal-Question-Metrics"* (GQM) strategy described by Basili in [2]. Our strategy, the *"Goal-Question-Analysis-Action"* strategy, is task oriented, goal driven, and focuses on decomposing the task of design recovery into a number of interrelated but distinct subtasks. This can be justified in two ways. Firstly, it is a natural way to perform design recovery. In practice, human maintainers do not try to understand the whole program at once but, instead, they gather different pieces of information and then they relate them using their expertise and their programming skills. Secondly, maintainers gather this information in a goal-driven way. At the beginning, they establish an objective (e.g., "find what this procedure returns and how it computes its returned values", "Is it a sorting algorithm?"), and then they gather information (e.g., "find where this variable is updated", "when does this loop terminate?"), which they believe is relevant for meeting the specific objective. Information gathering is not a random process but is guided by the experience and the programming skills of the maintainer. We believe that this experience is valuable and that a Reverse Engineering environment should give the programmer the flexibility to specify his own queries and provide him with the tools to gather the information *he* thinks is important and relevant. Thus, the question the maintainer asks is the key to selecting the right analysis tool and the required actions to be performed.

- *Use of behavioral program representation model.* This approach uses a program representation method which reveals the behavioral aspects of the represented program. Specifically, instead of using a representation method based only on structural and lexical properties of the program, we use information which is based on language semantics and focuses on *how* a computation is performed, on what is the *effect* of a computation after its termination. The advantage of this approach is twofold. Firstly, it is based on a well-established formalism which is supported by a solid mathematical theory (language semantics). Secondly, the mathematical theory supporting the model serves as a foundation for defining equivalence and partial order relations which can then be used to relate plans and programs. Most program understanding applications use program representations that incorporate control flow, data flow, and structural information. The drawbacks of these representation schemes are that plan-program equivalence has no mathematical foundation and that they are based on simplistic pattern-matching algorithms. The complexity of these formalisms is high, which makes the design recovery of realistically large programs difficult. Our approach, which uses language semantics, has not been investigated yet on complexity issues, but it definitely gives an advantage on plan-program matching, which can be achieved based not only on structural and syntactic properties, but also on semantic and behavioral properties of programs.

- *Use of an efficient and flexible plan localization strategy.* Plan localization is a key issue in most program understanders. Plan localization algorithms must cope with problems due to syntactic variations, interleaved plans, implementation variations, overlapping implementations, and unrecognizable code. Some of the most common plan localization algorithms used in existing applications are top-down goal agendas, depth-first search, best-first search, repeated traversals, and exhaustive search. The drawbacks in most applications are *(a)* the complexity issues and *(b)* failure to produce any results if perfect localization can not be achieved. Our approach has the complexity of an island-driven parser [6] and has the advantage that if the plan localization algorithm fails to recognize a plan completely it still recognizes parts of it, so that partial plan-program localization can be performed. In most cases, then, the maintainer can use his experience to recognize the rest of the plan himself.

- *Use of an integrating environment.* As indicated above, the view adopted in this project favours programmer-assisted program understanding over *automatic* program understanding. Our approach considers plan localization only as a subgoal of design recovery. Plans should be instances of principles and not simply instances of abstracted code fragments. Principles could be high-level descriptions of concepts occurring in the code (something that we believe it is not feasible with the current technology) or practical properties of the code itself (e.g., data dependencies, control dependencies, preconditions, and postconditions) with which programmers are familiar. Within this framework, a programmer may set high level-objectives with abstracted concepts. In order to address these objectives, he examines practical properties of the program such as control and data flow, dependencies, partial correctness properties, communication points, and informal information. A Reverse Engineering environment should provide to the maintainers tools for addressing these practical questions and not rely exclusively on automatic plan recognition using a static plan library. This approach fits well with the *Goal-Question-Analysis-Action"* process model, and has not been incorporated in any of the existing program understanders.

5 Conclusion

In this paper we gave an overview of a research project whose goal is the development of a reverse engineering environment. Reverse engineering and in particular design recovery are complex tasks and require the application of a number of diverse methodologies and techniques ranging from compiler technology to artificial intelligence and language semantics. Our research focuses on *(a)* the definition of a strategy for specifying the design recovery process, *(b)* the identification of a repertoire of design recovery objectives useful to a typical system maintainer, *(c)* the definition of program representation and plan representation schemes, *(d)* the development of a control strategy for localizing programming plans, *(e)* the development of a comparison technique for relating plans and code representations, and *(f)* the integration of analysis results from a diverse tools such as graph editors, parsers, and repositories.

Within this framework we have discussed a strategy for describing the design recovery process, the "Goal-Question-Analysis-Action" model. In this model top-level objectives give rise to specific questions which in turn require the application of specific analysis tools and actions. We have looked at a list of top-level objectives identified from contextual interviews with IBM researchers and software engineers. Objectives include: *(a)* data type mismatch in expressions (PL/AS related data type mismatch problems not handled by the compiler) *(b)* appropriateness of data structures used (e.g., all fields of a structure are used); *(c)* localization of equivalent or similar code fragments (code reduction); *(d)* localization of specific algorithms in the code (plan localization); *(e)* detection of inefficient and error prone code (high complexity, high and complex module interaction); *(f)* Code flow analysis and unitialized data (control/data flow, value ranges, constant propagation), and, finally; *(g)* change / impact analysis (slicing, dicing, module dependencies)

The program representation scheme chosen to meet these objectives is the Abstract Syntax Tree annotated with information accommodating links to concepts, results of previous analysis, references to control flow, data flow and structure graphs as well as representations of program behaviour and functionality by using formalisms borrowed from the area of language semantics. The advantage of this representation scheme is that it incorporates both a well-established model for representing software structure (AST) with formal annotations borrowed from the well-defined and formal area of programming language semantics for representing program functionality.

Comparison methods are used to relate plans and code representations. We distinguish between comparisons for structural and functional design recovery. Comparisons for structural design recovery are based on control and data flow properties, system organization properties, and data dependencies. On the other hand, comparisons for functional design recovery are based on the semantics of the exchanged resources between different program parts, the operations which change imported resources into their exported values, and the attributes, relations, and actions that manipulate program's components. Within this framework plan localization control strategies must be defined so that comparison can be performed even when interleaved plans, implementation variations, and scattered plans exist in the code. Island-driven search algorithms may contribute towards the partial design recovery objective.

Finally, 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 global repository for storing information and data used in all applications and tools, and, finally, a translation program for transforming tool specific software entities into a common and compatible in all environments entities stored in the global repository.

The target system is currently in its design phase and it will be applied for recovering the design of large complex legacy systems such as the SQL/DS package.

About the Authors

Kostas Kontogiannis is a Ph.D student at Computer Science Department, McGill University. He is working in the area of software design recovery, pattern matching, and artificial intelligence. He can be reached at kostas@cs.mcgill.ca.

Morris Bernstein is a research assistant at Computer Science Department, McGill University and he is working towards the development of a reverse engineering environment using data flow analysis techniques. he can be reached at zaphod@cs.mcgill.ca.

Renato DeMori is a professor and chairman of the Computer Science Department at McGill University. His interests are focused on the areas of pattern matching, speech recognition, and program design recovery. He can be reached at renato@cs.mcgill.ca

Ettore Merlo is a researcher at CRIM and an adjunct professor at McGill University. He is investigating program representation methods and data flow analysis techniques for design recovery. He can be reached at merlo@crim.ca.

References

[1] "AAAI92 Workshop Program on AI & Automated Program Understanding", *Workshop Notes AAAI92*, July 12-16, San Jose, 1992.

[2] Basili V. R., Rombach H. D., "Tailoring the Software Process to Project Goals and Environments" *9th International Conference on Software Engineering* ,1987, pp. 345-359.

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

[4] Buss, E. Henshaw, J. "Experiences in Program Understanding" *Proceedings CASCON'92*, Toronto, November 1992.

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

[6] Corazza, A., De Mori, R., Gretter, R. and Satta G., "Computation of Probabilities for an Island-Driven Parser," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Sept. 1991.

[7] Engberts, A., Kozaczynski, W., Ning, J. "Automating software maintenance by concept recognition-based program transformation," *IEEE Conference on Software Maintenance - 1991*, IEEE, IEEE Press, October 14-17 1991.

[8] Harandi, M.T. and Ning, J.Q., "Knowledge-Based Program Analysis," *IEEE Software*, Jan 1990, pp. 74 - 81.

[9] Hartman, J., "Understanding Natural Programs Using Proper Decomposition," *Proceedings of the 13th International Conference of Software Engineering*, May 1991.

[10] Hennessy M., "The Semantics of Programming Languages : An Elementary Introduction using Structural Operational Semantics", Wiley 1991.

[11] Johnson, W.L. and Soloway, E., "PROUST: Knowledge-Based Program Understanding," *IEEE Transactions on Software Engineering*, March 1985, pp. 267 - 275.

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

[13] Letovsky, S. "Plan Analysis of Programs," *Ph.D thesis, Yale University, Dept. of Computer Science, YALEU/CSD/RR662*, December 1988.

[14] Mueller, H.A., *Rigi as a Reverse Engineering Tool*, A report from the Dept. of Comp Sci, U of Victoria, March 1991.

[15] Ourston, D., "Program Recognition," *IEEE Expert*, Winter 1989, pp. 36.

[16] Plotkin G. D, "Structural Operational Semantics", *Lecture notes, DAIMI FN-19, Aarhus University, Denmark*, 1981.

[17] Quilici, A., Khan, J. "Extracting Objects and Operations from C Programs, " *Workshop Notes, AI and Automated Program Understanding, AAAI'92* 1992, pp. 93 - 97.

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

[19] Scott, D., "Data Types as Lattices," *SIAM Journal of Computing*, Vol. 5, No 3, 1976, pp. 522 - 587.

[20] Stoy, J.E., *Denotational Semantics*, MIT Press, 1977.

[21] Wills, L.M., "Automated Program Recognition: A Feasibility Demonstration," *Artificial Intelligence*, Vol. 45, No. 1-2, Sept. 1990.

[22] Wills, L.M., "Automated Program Recognition: Breaking out of the Toy Program Rut, " *Workshop Notes, AI and Automated Program Understanding, AAAI'92* 1992, pp. 129 - 133.