Working Conference on Reverse Engineering. Baltimore, MD. pp. 134-143, May 1993.

Flexible Control for Program Recognition

Linda M. Wills
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

Abstract

Recognizing commonly used data structures and algorithms is a key activity in reverse engineering. Systems developed to automate this recognition process have been isolated, stand-alone systems, usually targeting a specific task. We are interested in applying recognition to multiple tasks requiring reverse engineering, such as inspecting, maintaining, and reusing software. This requires a flexible, adaptable recognition architecture, since the tasks vary in the amount and accuracy of knowledge available about the program, the requirements on recognition power, and the resources available. We have developed a recognition system based on graph parsing. It has a flexible, adaptable control structure that can accept advice from external agents. Its flexibility arises from using a chart parsing algorithm. We are studying this graph parsing approach to determine what types of advice can enhance its capabilities, performance, and scalability.

1 Introduction

An experienced programmer can often reconstruct much of the hierarchy of a program's design by recognizing commonly used data structures and algorithms and knowing how they typically implement higher-level abstractions. We call these commonly used computational structures *clichés* [14]. Examples of clichés are algorithmic computations, such as list enumeration, binary search, and event-driven simulation, and common data structures, such as priority queue and hash table. The recognition process, which we refer to as *program recognition*, provides a short-cut to understanding a program's design. It bypasses complex reasoning about how behaviors and properties arise from certain combinations of language primitives.

Our practical motivation for studying program recognition stems from an interest in building automated systems that assist software engineers with tasks requiring reverse engineering, such as inspecting, maintaining, and reusing software. An automated recognition system that is to be applicable to a wide range of tasks must have a flexible, adaptable control structure. There may be sources of knowledge about the program in addition to the source code which can be used to guide recognition, such as its specification, documentation, comments, execution traces, a model of the problem domain, and typical properties of the program's inputs and outputs. The availability, completeness, and accuracy of this knowledge varies greatly across these tasks. For example, in debugging or verification applications, a specification of the program is often available from which strong guidance can be generated, while this information is often lacking in maintaining old code.

Different tasks also require different degrees of recognition power. For example, if the recognition system is going to be applied to verification, it can use a strategy that finds *any* complete recognition of the program, consisting only of *exact* matches of the clichés. On the other hand, if it were applied to documentation generation, it should generate multiple views of the program, reporting all possible instances of clichés. For debugging, partial (near-miss) recognitions of clichés should be produced.

In addition, tasks vary in the resources they can allocate to recognition. For example, a real-time response may be required if a person is using it interactively as an assistant in maintaining code. In this situation, it may be more desirable to quickly recognize clichés that are more "obvious" rather than spending more time to uncover clichés that are obscured (e.g., by an optimization which must be undone for them to be revealed). It should be possible to prioritize the search for certain clichés, so that obvious ones are recognized early, while still providing a later, "try harder" phase in which the more hidden clichés can be found. This gains efficiency without permanently sacrificing completeness.

Furthermore, the knowledge about the program, the requirements on recognition power, and the resources available typically change as the tasks are being performed.

We have developed an experimental recognition system, called GRASPR [19], which when given a library of clichés, finds all instances of clichés in a program. It can generate multiple views of a program as well as near-miss recognitions of clichés. It has a flexible, adaptable control structure that can accept advice and guidance from external agents.

GRASPR is intended to be part of a future hybrid reverse engineering system that integrates not only purely code-driven recognition, but also more heuristic techniques that generate expectations from other information sources, such as documentation, to guide the search for clichés. GRASPR's flexibility is crucial for interaction with such techniques, particularly given the variation in knowledge and resources available for different application tasks.

Heuristics can be provided in a data-driven way, rather than being hard-coded into the system. Expectations about the clichés that are likely to be found and their locations can be used by GRASPR to focus its search. But these are not required as inputs. The trade-off between recognition power and computational expense can be explicitly controlled so that some clichés are recognized quickly, while other, more expensive recognitions are postponed.

GRASPR, which stands for "GRAph-based System for Program Recognition," uses a graph parsing approach to automating recognition. Its flexibility arises from using a chart parsing algorithm which makes control and search strategies explicit.

1.1 Previous Recognition Work

Several researchers have shown the feasibility of automating recognition and the usefulness of its results, most recently Bertels [1], Hartman [6], Johnson [8], Letovsky [10], Murray [12], Ning [13], and Wills [18]. (See [19] for a more detailed description of these systems and earlier research in this area.)

All existing recognition systems are isolated, standalone systems which are not expected to interact with people or with other reverse engineering techniques. They all are committed to a rigid control strategy, typically targeting a particular application, such as debugging, restructuring, or documentation.

Some [8, 12] have cost-cutting heuristics built in which are chosen on a trial-and-error basis. Some [8, 10, 12] search for a single best interpretation of the program, while permanently cutting off alternatives, so their power cannot be incrementally increased. They also cannot generate multiple views of the program when desired, nor provide partial information when only near-misses of clichés are present.

Some recognition techniques take as input information about the goals and purpose of the program (in the form of a specification [8] or model program [12]). While these techniques show the utility of these additional sources of information, they *rely* on this information being given as input, rather than accepting and responding to it if it is available for a given task.

Much of the early work in program recognition provides no evaluation of the representations or techniques used. More recent research includes some empirical analysis, typically studying the accuracy of recognition and the recognition rates over sets of programs (usually student programs in program tutoring applications [8, 12]). However, discussions of limitations (except Hartman's [6]) have focused mainly on implementational limitations, rather than on inherent limitations of the approach. They also do not describe how additional information or guidance from external agents can help.

Our current work moves beyond studying feasibility, by examining computational costs, GRASPR's tolerance to variation, and our graph formalism's expressiveness in capturing programming clichés. We do not expect GRASPR by itself to scale up to large programs and to recognize all forms of clichés. Our goal is to determine what forms of advice can focus its search and broaden its representational capabilities. We are interested in how GRASPR will interact with other reverse engineering techniques, including heuristic forms of reasoning based on expectations, to contribute to a wide range of software development and maintenance tasks.

The next section discusses GRASPR's graph parsing approach to recognition, the parsing algorithm it uses, and how it can be explicitly controlled. Then a summary of the results of our experimentation with GRASPR on real-world programs is given, pointing out some forms of advice that are useful.

2 GRASPR's Architecture

GRASPR employs a graph parsing approach to automating program recognition, shown in Figure 1. It represents a program as a restricted form of directed acyclic graph, called a *flow graph* [3, 19], which is annotated with attributes. Nodes in the flow graph represent functions, edges denote dataflow, and attributes capture control flow information. The cliché library is

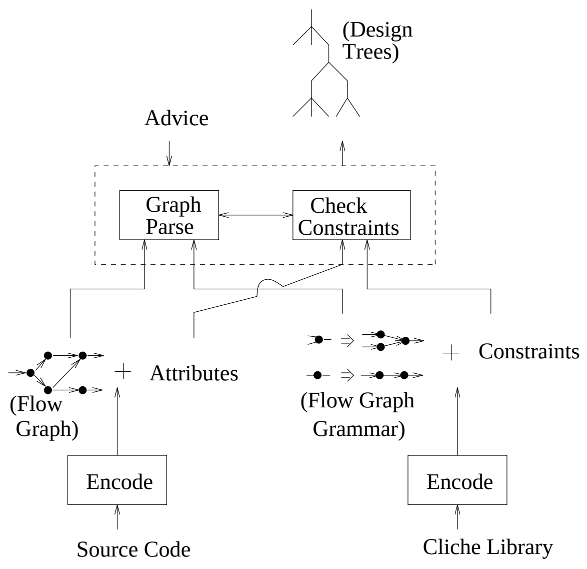
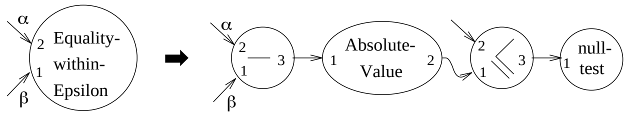


Figure 1: GRASPR's architecture.



Attribute Conditions: All nodes co-occur.
Attribute-Transfer Rules:
ce := ce(null-test)
success-ce := failure-ce(null-test)
failure-ce := success-ce(null-test)

Figure 2: An example flow graph grammar rule.

encoded as an attributed graph grammar, whose rules impose constraints on the attributes of flow graphs matching the rules' right-hand sides. Recognition is achieved by parsing the dataflow graph in accordance with the grammar. Attribute constraint checking and evaluation is interleaved with the parsing process.

Figure 2 shows an example of a graph grammar rule encoding a simple cliché: testing whether two numbers are within some "epsilon" of each other. The right-hand side is a typical flow graph, capturing primarily dataflow information. Control flow information is stored in the attributes of a flow graph representing a program. Each node has a *control environment* attribute whose value indicates under which conditions the function represented by the node is executed. Nodes in the same control environment represent functions that are all executed under the same conditions; they are said to *co-occur*. Sink nodes, representing conditional tests, carry two additional attributes, *success-ce* and *failure-ce*, specifying the control environments containing functions that are executed when the conditional test succeeds or fails, respectively. The rule for Equality-within-Epsilon constrains all the nodes that match its right-hand side to co-occur. The attribute-transfer rules specify how to synthesize the left-hand side node attributes from the attributes of the flow graph matching the right-hand side. (These are stated informally in Figure 2; see [19] for a formal description of the attribute language used in encoding clichés.)

The parsing technique yields a hierarchical description of a plausible design of the program in the form of derivation trees, which we call design trees. These specify the clichés found and their relationships to each other. In general, GRASPR generates a forest of design trees for a given program. These provide multiple views of parts of the program on multiple levels of abstraction.

For example, one of the programs GRASPR recognizes is a simulation program written in Common Lisp by members of a parallel-processing research group at MIT. It sequentially simulates a parallel message-passing system. GRASPR correctly recognizes the main algorithm of the program as being a clichéd synchronous simulation algorithm, which mimics the real parallel machine by simulating the actions of its processing nodes in "lock-step." A portion of the design tree produced by GRASPR describing the recognized cliché is shown in Figure 3. (The dashed lines at the tree's fringe are links to primitive operations in the source code, which indicate the location of a particular cliché in the code. Triangles denote subtrees that are not shown due to space limitations.)

The main advantages of using a graph grammar formalism for representing programs and clichés is that (1) it eliminates many common forms of variation that hinder recognition, such as, programming language chosen and syntactic constructs used, (2) it tends to localize clichés, and 3) it captures hierarchical relationships between clichés so that parsing uncovers implementational design decisions.

Our recognition system is able to recognize structured programs and clichés containing conditionals, loops with any number of exits, recursion, aggregate data structures, and simple side effects due to assignments. With the exception of CPU [10], existing recognition systems cannot handle aggregate data structure clichés and a majority do not handle recursion. We are working with programs that are in the 500 to 1000 line range. The largest program recognized by any existing recognition system is a 300-line database program recognized by CPU. All other systems work with programs on the order of tens of lines.

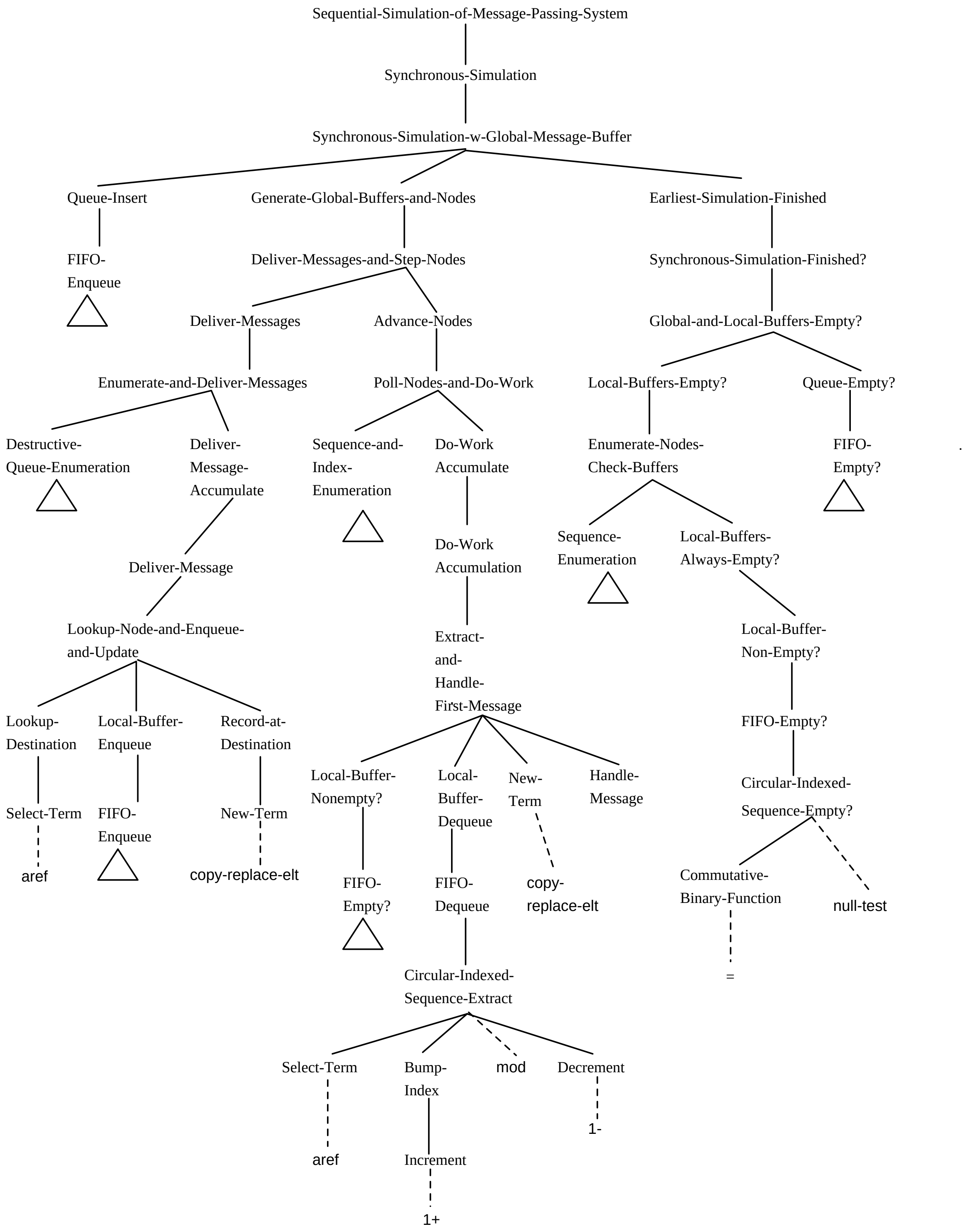

Sequential-Simulation-of-Message-Passing-System
Synchronous-Simulation
Synchronous-Simulation-w-Global-Message-Buffer
Queue-Insert
Generate-Global-Buffers-and-Nodes
Earliest-Simulation-Finished
FIFO-Enqueue
Deliver-Messages-and-Step-Nodes
Synchronous-Simulation-Finished?
Deliver-Messages
Advance-Nodes
Global-and-Local-Buffers-Empty?
Enumerate-and-Deliver-Messages
Poll-Nodes-and-Do-Work
Local-Buffers-Empty?
Queue-Empty?
Destructive-Queue-Enumeration
Deliver-Message-Accumulate
Sequence-and-Index-Enumeration
Do-Work Accumulate
Enumerate-Nodes-Check-Buffers
FIFO-Empty?
Deliver-Message
Do-Work Accumulation
Sequence-Enumeration
Local-Buffers-Always-Empty?
Lookup-Node-and-Enqueue-and-Update
Extract-and-Handle-First-Message
Local-Buffer-Non-Empty?
Lookup-Destination
Local-Buffer-Enqueue
Record-at-Destination
FIFO-Empty?
Local-Buffer-Nonempty?
Local-Buffer-Dequeue
New-Term
Handle-Message
Circular-Indexed-Sequence-Empty?
Select-Term
FIFO-Enqueue
New-Term
aref
copy-replace-elt
FIFO-Empty?
FIFO-Dequeue
copy-replace-elt
Commutative-Binary-Function
null-test
=
Circular-Indexed-Sequence-Extract
Select-Term
Bump-Index
mod
Decrement
aref
Increment
1-
1+

Figure 3: A portion of a design tree produced in recognizing a sequential simulator program.

Our motivation in working with these programs is to push the limits of the graph parsing recognition approach to determine what advice would be helpful in scaling up to larger programs. Because techniques for generating this advice have not yet been developed, whenever we identified an inherent limitation requiring advice, we performed transformations to the programs to avoid encountering the limitation. This allowed us to continue our study.

We also performed transformations to avoid limitations in the current recognition technology, which are not inherent to the graph parsing approach. The most notable of these is the problem of dealing with programs that perform side effects to mutable data structures. To get around this limitation, we manually translated our example programs to pure (functional) versions and recognized pure clichés in them. Fortunately, the translation was straightforward and we plan to semi-automate it in the future by interleaving dataflow analysis with the recognition of stereotypical aliasing patterns [19].

Our cliché library contains a core set of general-purpose, "utility" clichés, along with a set of clichés from the domain of sequential simulation. These are encoded in approximately 200 graph grammar rules. They were manually collected from introductory computer science textbooks, books on simulation and queueing systems, and by examining two example simulator programs and speaking to their programmers. The library's coverage is by no means absolute. However, it demonstrates the kinds of algorithms and data structures that can be expressed within our graph grammar formalism.

2.1 Recognition as Subgraph Parsing

We formulate the program recognition problem in terms of solving a parsing problem for flow graphs. A flow graph is an attributed, directed, acyclic graph, whose nodes have *ports* – entry and exit points for edges. A flow graph grammar is a set of rewriting rules (or productions), each specifying how a node in a flow graph can be replaced by a particular sub-flow graph. (A flow graph H is a *sub-flow graph* of a flow graph F if and only if H's nodes are a subset of F's nodes, and H's edges are the subset of F's edges that connect only those ports found on nodes of H.)

The *subgraph parsing* problem for flow graphs is: Given a flow graph F and a context-free flow graph grammar G, find all possible parses of all sub-flow graphs of F that are in the language of G.

The program recognition problem of determining which clichés in a given library are in a given program

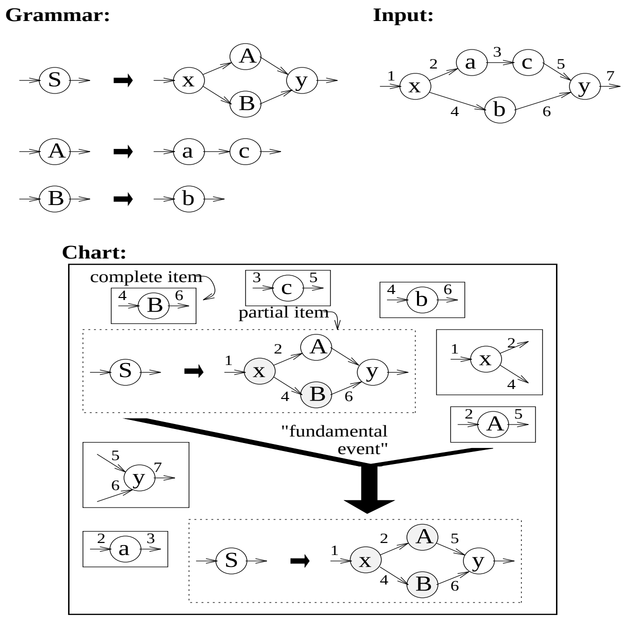


Figure 4: Flow graph chart parsing.

(and their locations) is formulated as a subgraph parsing problem: Given a flow graph F representing the program's dataflow and a cliché library encoded as a flow graph grammar G, solve the subgraph parsing problem on F and G. This formulates *partial* program recognition as well as recognition of the program as a whole. A cliché instance may be surrounded by or interleaved with unfamiliar code, but if it is localized in a sub-flow graph of the program's flow graph, it will be recognized by subgraph parsing.

2.2 Chart Parsing Flow Graphs

To solve the subgraph parsing problem, GRASPR uses a graph parser which has evolved from Earley's string parsing algorithm [4], incorporating three key improvements: (1) generalization of string parsing to flow graphs (Brotsky [3], Lutz [11]), (2) generalization of the control strategy to allow flexibility in the rule-invocation and search strategies employed (Kay [9], Thompson [16], Lutz [11]), and (3) extension of the grammar formalism to capture aggregation relationships (Wills [19]) between single inputs or outputs of a non-terminal left-hand side node and a tuple of inputs or outputs of a right-hand side sub-flow graph. (This is used to express the relationships between the inputs and outputs of an abstract operation on aggregate data structures and aggregates of the inputs and outputs of the lower-level operations that make up its concrete implementation.) The formalism was

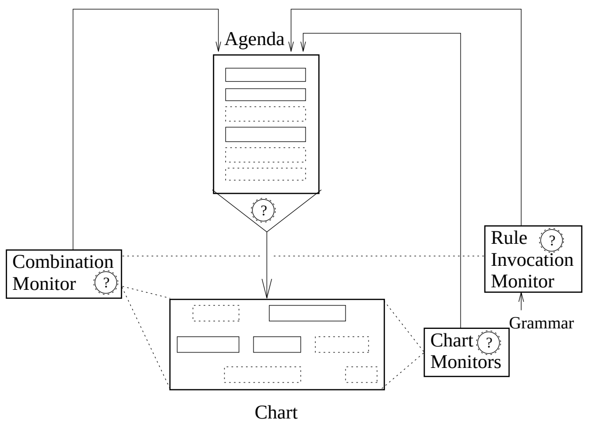


Figure 5: Agenda-based implementation of chart parser, with control knobs.

also extended to handle variation in graphs due to structure-sharing (Lutz [11], Wills [18, 19]) and aggregation organization (Wills [19]).

The flexible control strategy inherited from the chart parsers of Kay [9], Thompson [16], and Lutz [11] gives GRASPR its flexibility, adaptability, and ability to accept advice. This section conceptually describes this aspect of the parser and how it is used by GRASPR. (For more details, see [19].)

The parser maintains a database, called a *chart*, of partial and complete analyses of the input graph in the form of *items*. This is shown in Figure 4. A complete item represents the recognition of some sub-flow graph as some terminal or non-terminal in the grammar. A partial item represents a partial recognition of a non-terminal. The central action of the parser is to repeatedly create new items by *extending* partial items with complete items for nodes not yet matched. This is called the "fundamental event." In other words, the parser is searching for matches of right-hand side flow graphs of grammar rules to sub-flow graphs of the input graph. Items represent stages in the search.

The parser continually generates items and adds them to the chart. The process is conceptually parallel, but to implement it on a sequential machine, an agenda must be used to queue up the items to be added to the chart. Items are iteratively pulled off the agenda and placed in the chart, as illustrated in Figure 5. As an item is added, it is paired with other items with which it can be combined. If the item being added is a complete item, then it is paired with partial items that need it. On the other hand, if the item added is a partial item, then it is paired with any complete items for the non-terminals it needs.

This parser has a set of "control knobs" and parameters that can be set to achieve a desired control strategy or accept and respond to advice. (Examples of how these are used are given in the next section.)

Agenda Access: Specifying the strategy for pulling items from the agenda to be placed in the chart is one way of controlling the parser's search strategy. For example, certain partial items might be pulled from the agenda, based on which part of the input graph they have started to match or based on how much of their right-hand sides they have matched already.

Node Orderings: Associated with each rule in the grammar is an ordering of the nodes in its right-hand side specifying the order in which the nodes are to be matched. This can be used to control the order in which constraints (such as node-type and edge connection constraints) are enforced.

The ordering may be *strict*, in which case the nodes are related in a chain, having exactly one minimal node and exactly one maximal node. Otherwise, if nodes can be related to more than one node in the ordering, the ordering is *partial*. The choice of whether a node ordering is strict or partial affects the computational complexity and power of the parser. Strict node orderings are cheaper, since they generate fewer partial and duplicate items. However, partial node orderings provide more near-miss information, which is important in dealing with buggy programs and in eliciting advice.

Adding to Agenda: By controlling which items are added to the agenda and chart, the rule invocation strategy can be controlled. For example, to make the parser adopt a bottom-up parsing strategy, whenever a complete item is added to the chart, new empty items can be added to the agenda for each rule that needs the complete item to get started (i.e., the rule has a minimal node in its node ordering that is of the same type as the type derived by the complete item). Currently, GRASPR uses this bottom-up strategy, since it facilitates recognizing programs that contain clichés interleaved with unfamiliar code. However, if strong expectations are available about which clichés are likely to be found (and where), a top-down rule invocation strategy can be used.

Extendibility Criterion: The extendibility criterion constrains which pairs of items can be combined. As is discussed later, this can be used to impose partitioning constraints to enhance GRASPR's performance.

Chart Monitors: Special-purpose monitors can be defined to watch the chart for particular types of items to enter. They can look for opportunities to view part of the input graph in an alternative way in order to yield more parses. The graph is not explicitly changed to the alternative view. Instead, new items are created which represent the alternative views and these are added to the agenda. For example, monitors such as these are used to deal with variation due to redundant computations: when the result of some inexpensive computation is needed more than once, programs can vary in whether they recompute the value each time it is needed or cache the result in a temporary variable. A monitor watches for redundant computations and generates a canonical view in which the computations are performed once.

Monitors can also elicit advice by detecting question-triggering patterns (which are encoded in rules included in the grammar along with rules for clichés). For example, a pattern might indicate that a particular constraint is likely to hold. When such a pattern is found, the recognition system can ask whether the constraint is satisfied. This is useful if the constraint is difficult or costly for the parser to check. The question might be more easily answered by some other source (such as a person).

Monitors also watch for classes of near-miss recognitions to arise that can be fixed and resumed. Items that are failing certain constraints might be made to complete by explicitly making simplifying assumptions that allow these constraints to be satisfied.

The tasks set up by chart monitors can be prioritized so that those that are expensive or less likely to be effective can be postponed while quick, promising tasks are accomplished first.

3 Evaluation: What Advice is Useful?

GRASPR is intended to be integrated with techniques that can focus its search and complement its code-driven, formal parsing process with heuristic reasoning from expectations. To determine what forms of advice should be given to GRASPR, we are studying its strengths and weaknesses in the context of our example simulator programs. This study is facilitated by the formal graph grammar framework on which GRASPR is based. This section briefly summarizes our findings so far in three areas: (1) the variation GRASPR can tolerate, (2) the expressiveness of its grammar formalism for capturing clichés, and (3) the costs of its recognition tasks. In each case, examples of some sources of difficulties are presented to illustrate the forms of guidance that would be useful to GRASPR. (For more details and examples, see [19].)

3.1 Tolerating Variation

Program recognition is difficult because clichés may appear in programs in a wide variety of forms. The flow graph representation for programs and clichés has significant advantages over the text-based representations used by many other recognition systems. It makes GRASPR robust under syntactic variation (in the choice of syntactic constructs and programming language), implementational variation (in the choice of concrete algorithms and data structures for abstractions), and organizational variation (in the nesting of aggregate data structures and in subroutine decomposition). In addition, GRASPR is robust under variation due to delocalization of clichés, unfamiliar code, and function-sharing optimizations. (See [18, 19, 15] for details and examples of how these classes of variation are tolerated.)

Difficulties arise when a program's data and control flow are implicit or derived or cannot be determined statically. For example, when a program accepts functional inputs, some data and control flow information is statically unavailable. If portions of clichés are contained in these inputs, then GRASPR must explicitly ask whether the input functions satisfy the relevant constraints.

A common source of derived dataflow is the use of handles, in which some piece of data is stored in a pooling structure (e.g., a hash table) and an index to the data is passed around the program. To use the piece of data, the program must first look it up. This introduces intermediate computation that interrupts data flowing from one primitive operation to another.

To deal with derived dataflow, GRASPR needs advice pointing it out. One way GRASPR might elicit this advice is by looking for "question-triggering" patterns. For example, standard look-up and update functions suggest that a handle is being used. Recognizing patterns of data structure creation in which each new structure (e.g., "NODE") is collected in a larger structure (e.g., stored in the variable "*NODES*") suggests that the larger structure might be a pooling structure. These hypotheses can be presented to the user or some expectation-driven component for confirmation (e.g., based on mnemonic names and documentation). Once the derived dataflow is uncovered, GRASPR can generate an alternative view of the flow graph in which the derived dataflow links become explicit attributed edges.

Unpredictable variations are also introduced by special-case optimizations of clichés which simplify the general case in the context of a particular program. Not all special-case simplifications of clichés are themselves clichés, so we do not want to enumerate them all in the cliché library.

Non-clichéd optimizations often cause some, but not all of a cliché to be recognized. They often avoid computation by taking advantage of an opportune equality or an intentionally cached value. One way to elicit advice on whether some computation is a special-case optimization is to generate maximally-sized near-miss recognitions of the cliché and then generate a hypothesis that the value used is equal to the result of the computation in the part of the cliché not yet matched. This can then be confirmed by an external agent, such as a person, or by applying limited reasoning techniques to uncover dataflow equalities or conditional simplifications in simple cases [5, 10].

3.2 Grammar Expressiveness

Our graph formalism is expressive enough to capture general-purpose programming clichés, such as priority-queue insert, as well as clichés from the simulation domain. The formalism is able to concisely encode algorithmic and data aggregation clichés whose constraints are primarily based on data and control flow.

However, although the graph formalism allows us to encode clichés on a high level of abstraction, the level of abstraction is still limited by the amount of detail that must be specified about the clichés (e.g., function types and arity, and exact dataflow connections). This makes it difficult to capture some loosely constrained clichés. A simple example of one cliché that is difficult to capture is a common type of conditional dispatch which occurs in program interpreters (particularly for Lisp-like languages). The standard algorithm dispatches on the type of an expression, to code for handling that expression. Instances of this cliché vary with the types of expressions that can be interpreted. The number and type of test cases in the conditional dispatch vary, as do the actions to which they are dispatched. Also, the dataflow connection constraints are loose. This cliché is difficult to encode in the grammar formalism because it requires specifying which functions are involved in the cliché, their arity, and the exact dataflow between them.

GRASPR would benefit from collaboration with a complementary (yet to be developed) recognition system that recognizes collections of coarse control flow and dataflow constraints. The results of this recognition could be easily integrated into GRASPR by adding complete items representing them to GRASPR's agenda and chart.

3.3 Cost

GRASPR is performing a constrained search for matches of clichés – for each rule of the grammar, the parser is searching for a way to match each node of the rule's right-hand side to an instance of the node's type in the input graph. This is inherently exponential. (In fact, the subgraph parsing problem for flow graphs is NP-complete [19], so it is unlikely that there is a subgraph parsing algorithm that is not exponential in the worst case.)

However, in the practical application of graph parsing to recognizing *complete* instances of clichés, constraints are strong enough to prevent exponential behavior in practice. The three key constraints that come into play are: (1) constraints on node types, which correspond to function types, (2) edge connection constraints, which represent dataflow dependencies, (3) and co-occurrence constraints, which are a class of control flow constraints that require a set of functions to all be executed under the same control conditions. (See [19] for a detailed discussion and supporting statistics.)

As we increase the recognition power of GRASPR to make it generate more partial recognitions of clichés, we lose the advantage of strong constraint pruning. What is most expensive for GRASPR to do is the task of near-miss recognition of clichés – recognizing all possible *partial* (as well as complete) instances of clichés. This task is useful in robustly dealing with buggy programs, learning new clichés, and eliciting advice.

This section shows how we can explicitly make the trade-off between recognition power and cost, and control the application order of constraints. It also shows how GRASPR can use indexing and partitioning advice.

Recognition Power versus Cost

We can explicitly control how much partial information GRASPR is to generate by specifying the type of node ordering associated with each grammar rule. Strict node orderings specify only one order of matching a rule's right-hand side. If a node is missing or violates some constraint, all nodes following it in the ordering are prevented from being matched. Strict orderings are used to recognize only complete instances of clichés. Partial node orderings allow more than one order in which to match right-hand side nodes, since a partial item can be extended with more than one item if there is more than one next unmatched node in the ordering. If a portion is missing or violates a constraint, it does not necessarily prevent other parts of the right-hand side from being matched. Partial node orderings allow GRASPR to explore more of the search space, at the expense of space and time. The extreme (and most costly) partial node ordering in which no node is ordered with respect to any other node causes maximally-sized near-misses to be recognized.

Ordering Constraint Application

To achieve its best performance, GRASPR should apply the strongest constraints first. One particularly easy way to advise GRASPR to do this is through the choice of node orderings. Salient node types should be matched before more common node types. This imposes strong disambiguation constraints early. Nodes that are connected to lots of other nodes or are constrained to co-occur with other nodes should be matched early, since more binary constraints apply to them and their matches will better constrain the nodes that follow them in the ordering. Node orderings can be produced automatically based on the properties of a given input graph and grammar.

Indexing and Partitioning

Advice on which non-terminals to look for (indexing) and which sub-flow graphs to focus GRASPR's search on (partitioning) would be helpful, particularly in dealing with the expense of near-miss recognition. This advice can come from an external agent, that has access to more information about the program than is found in the source code. For example, people can often break up a program into pieces that "go together" in that they provide a particular functionality or belong to the same abstract domain-specific concept. They base this partitioning on design documentation and program comments or even simply names of subroutines and variables. The DESIRE system [2] attempts to automate this process. Based on a rich domain model, it recognizes patterns of organization and linguistic idioms in a program associated with concepts in the program's problem domain. This information can be used to quickly draw attention to sections of the program where there may be clichés related to a particular concept. Other, more conventional techniques for reverse engineering large programs also provide ways of extracting possible partitions (e.g., by clustering [7] and slicing [17]).

Partitioning and indexing advice can be used by GRASPR in two ways. One way is to use it to statically narrow down the grammar and input flow graph given to the parser beforehand. The other is to interleave indexing and partitioning techniques with recognition, allowing the focus to change as deeper knowledge of the program is acquired.

The danger of *static*, a priori partitioning is that a cliché might be missed if it is not contained within some partition boundary. This technique works best if there are standard partitionings of clichés and the clichés appear in programs in these same organizations. (For example, Hartman [6] has identified a restricted class of clichés, called *control concepts*, that have this property.)

GRASPR can use *dynamic* partitioning advice by incorporating it into the extendibility criterion so that items that are candidates for combination must represent recognitions of sub-flow graphs within the same partition. The extendibility criterion need not permanently prohibit pairs of items from being combined. Rather, pairs that fail it might be postponed from combination until a "try-harder" phase. This allows certain combinations to be preferred over others, while allowing less favorable combinations to be tried later, as the partitions are refined. The choice of which postponed combinations to try might be based on whether they are in areas where no cliché has yet been recognized or whether they involve partial items that are near-misses or have salient parts matched already. Thus, completeness need not be lost due to heuristic partitioning. Also, the partitioning constraint can be selectively applied (e.g., rule by rule or between selected pairs of nodes within a rule's right-hand side).

4 Conclusion: Looking to the Future

Recognition by graph parsing has significant advantages in tolerating variation and uncovering implementational design decisions. Parsing also provides a solid, formal framework for characterizing its strengths and weaknesses.

Although the scalability of this approach is restricted by its representational rigidity and by the expense of near-miss recognition, GRASPR is not intended to scale up on its own. It has been given a flexible control strategy, which enables it to accept advice from external agents to address its limitations. Depending on the information available about the program, heuristics can be easily added, changed, applied selectively, or overruled. GRASPR can also be tailored to the resources available and recognition power required for a particular task, making it applicable in multiple reverse engineering contexts.

In general, recognition is an inherently code-driven (bottom-up) technique, which focuses on what is familiar in the code. To round out a reverse engineering system that incorporates it, recognition must be complemented in two ways: (1) with an expectation-driven (top-down) technique and (2) with techniques to deal with the unfamiliar (novel or buggy) parts of programs. GRASPR's flexible control architecture provides a seed for a future hybrid reverse engineering system which integrates many different components for extracting design information from source code and any knowledge sources associated with it, such as its documentation, domain model, and execution traces.

More empirical study must be done to determine the role of recognition in this system. To do this, current recognition technology must be extended to deal with broader classes of programs and clichés. (Important extensions are the ability to deal with side effects to mutable objects and the ability to capture and recognize loosely constrained clichés.) Tools must be developed to facilitate cliché acquisition. More study is also needed of how recognition scales up to larger cliché libraries.

Acknowledgments

I am grateful to Chuck Rich and Richard Waters for guidance and insights on this research. I have also benefited from discussions with David McAllester, Spencer Rugaber, and Peter Szolovits.

The research described here was conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the following organizations: National Science Foundation under grants IRI-8616644 and CCR-898273, Advanced Research Projects Agency of the Department of Defense under Naval Research contract N00014-88-K-0487, IBM Corporation, NYNEX Corporation, and Siemens Corporation.

References

[1] K. Bertels. Qualitative reasoning in novice program analysis. Technical report, Universiteit Antwerpen, June 1991. PhD thesis.

[2] T. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, 22(7):36–49, July 1989. Also published as MCC Technical Report STP-378-88.

[3] D. Brotsky. An algorithm for parsing flow graphs. Technical Report 704, MIT Artificial Intelligence Lab., March 1984. Master's thesis.

[4] J. Earley. An efficient context-free parsing algorithm. *Comm. of the ACM*, 13(2):94–102, 1970.

[5] R. Hall. Program improvement by automatic redistribution of intermediate results. Technical Report 1251, MIT Artificial Intelligence Lab., February 1990. PhD thesis.

[6] J. Hartman. Automatic control understanding for natural programs. Technical Report AI 91-161, University of Texas at Austin, 1991. PhD thesis.

[7] D. Hutchens and V. Basili. System structure analysis: Clustering with data bindings. *IEEE Trans. on Software Engineering*, 11(8), August 1985.

[8] W. L. Johnson. *Intention-Based Diagnosis of Novice Programming Errors*. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

[9] M. Kay. Algorithm schemata and data structures in syntactic processing. In B. Grosz, K. Sparck-Jones, and B. Webber, editors, *Readings in Natural Language Processing*, pages 35–70. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

[10] S. Letovsky. Plan analysis of programs. Research Report 662, Yale University, December 1988. PhD Thesis.

[11] R. Lutz. Chart parsing of flowgraphs. In *Proc. 11th Int. Joint Conf. Artificial Intelligence*, pages 116–121, Detroit, Michigan, 1989.

[12] W. Murray. *Automatic Program Debugging for Intelligent Tutoring Systems*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1988.

[13] J.Q. Ning. A knowledge-based approach to automatic program analysis. Technical report, University of Illinois, Urbana-Champaign, 1989. PhD thesis.

[14] C. Rich and R. C. Waters. *The Programmer's Apprentice*. Addison-Wesley, Reading, MA and ACM Press, Baltimore, MD, 1990.

[15] C. Rich and L. M. Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, 7(1):82–89, January 1990.

[16] H. Thompson. Chart parsing and rule schemata in GPSG. In *Proc. 19th Annual Meeting of the ACL*, Stanford, CA, 1981.

[17] M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, 10:352–357, 1984.

[18] L. Wills. Automated program recognition: A feasibility demonstration. *Artificial Intelligence*, 45(1-2):113–172, 1990.

[19] L. Wills. Automated program recognition by graph parsing. Technical Report 1358, MIT Artificial Intelligence Lab., July 1992. PhD Thesis.