Using Attributed Flow Graph Parsing to Recognize Clichés in Programs

Linda Mary Wills

School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332-0250
linda@ee.gatech.edu
http://www.ee.gatech.edu/users/linda/

Abstract. This paper presents a graph parsing approach to recognizing common, stereotypical computational structures, called clichés, in computer programs. Recognition is a powerful technique for efficiently reconstructing useful design information from existing software. We use a flow graph formalism, which is closely related to hypergraph formalisms, to represent programs and clichés and we use attributed flow graph parsing to automate recognition. The formalism includes mechanisms for tolerating variations in programs due to structure sharing (a common optimization in which a structural component is used to play more than one functional role). The formalism has also been designed to capture aggregation relationships on graph edges, which is used to encode aggregate data structure clichés and the abstract operations on them. A chart parsing algorithm is used to solve the problem of determining which clichés in a given cliché library are in a given program.

1 Program Recognition

An experienced programmer can often reconstruct much of the hierarchy of a program's design by recognizing commonly used data structures and algorithms in it and reasoning about how they typically implement higher-level abstractions. We call these commonly used computational structures *clichés* [21]. 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. Since clichés have well-known properties and behaviors, the process of recognizing clichés, which we refer to as *program recognition*, provides an efficient means of reconstructing and understanding a program's design. It bypasses complex reasoning about how behaviors and properties arise from certain combinations of language primitives.

Several researchers have shown the feasibility and usefulness of automating recognition, most recently [9, 10, 12, 13, 20, 25, 26]. A primary motivation for automating recognition is to facilitate tasks requiring program understanding, such as maintaining, debugging, and reusing software.

We have developed an experimental recognition system, called GRASPR ("GRAph-based System for Program Recognition") [26], to automate program recognition.

Given a program and a library of clichés, GRASPR finds all instances of the clichés in a program. It can generate multiple views of a program as well as near-miss recognitions of clichés. It can also recognize clichés in programs even if they are surrounded by or interleaved with unfamiliar code.

The concept of programming clichés is a pre-theoretic notion with no precise formalization. The cliché library is a collection of *standard, frequently used* algorithms and data structures, encoded in terms of their data and control flow constraints. Finding all instances of a cliché in a program means finding program structures that satisfy these constraints; it does not mean recognizing any arbitrary program structure that has the same functionality as the cliché. For example, GRASPR will recognize common implementations of hash tables that are captured in the given library, not all possible program structures that can be viewed as hash tables.

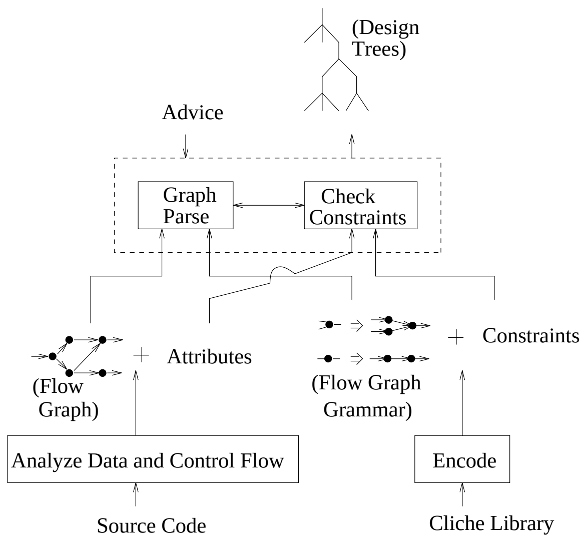

Fig. 1. GRASPR's architecture.

GRASPR uses a graph parsing approach to automating program recognition, shown in Figure 1. It uses data and control flow analysis to represent a program as a restricted form of directed acyclic graph, called a *flow graph* [1, 26], which is annotated with attributes. Nodes in the flow graph represent functions, ports on nodes represent inputs and outputs of the functions, edges connect ports and denote dataflow, and attributes capture additional information, such as recursion, control flow and data aggregation. The cliché library is encoded as an attributed graph grammar, whose rules impose constraints on the attributes of flow graphs matching the rules' right-hand sides. An example constraint is to require that a node have the same "control environment" attribute as another node (i.e., the nodes "co-occur"), which means that the two nodes represent operations that are performed under the same control conditions (e.g., within the same branch of a conditional).

The grammar rules capture implementation relationships between the clichés. Recognition is achieved by parsing the flow graph representing the program in accordance with the grammar. Attribute constraint checking and evaluation are interleaved with the flow graph parsing process. The control flow attributes are themselves represented and manipulated as graphs, since they encode a type of structural information; this allows well-defined graph operations to be used in attribute evaluation and constraint checking.

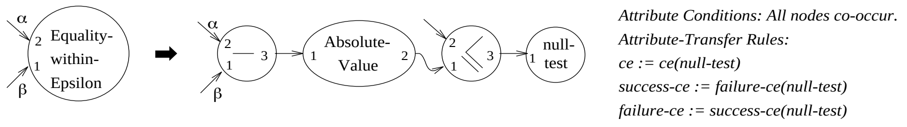
α
β
Equality-within-Epsilon
Absolute-Value
null-test
Attribute Conditions: All nodes co-occur.
Attribute-Transfer Rules:
ce := ce(null-test)
success-ce := failure-ce(null-test)
failure-ce := success-ce(null-test)

Fig. 2. A flow graph grammar rule encoding the Equality-within-Epsilon cliché.

Figure 2 shows an example of a flow 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. 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. In this example, the `null-test` is a terminal representing the primitive conditional test of a boolean value. In addition to a control environment (*ce*), it has a *success-ce* (resp. *failure-ce*) attribute representing the control environment of operations that are performed when the conditional test succeeds (resp. fails).

The control environment attribute of a node indicates under which conditions the operation represented by the node is executed, relative to when other operations in the program are executed. Control environments form a partial order imposed by the control structure of the program (i.e., its branches, iterations, and recursive calls). Other typical constraints on nodes are that they are within the control environment of the success branch of a conditional test, or that they are in the same control environment as a recursive function call. Similar constraints are also imposed on edges, restricting the control environments in which they carry dataflow. (Attribute constraints and transfer rules are stated informally in Fig. 2; [26] gives a formal description of the attribute language.)

Parsing yields a hierarchical description of a plausible design of the program in the form of derivation trees specifying the clichés found and their relationships to each other. In general, `GRASPR` generates a forest of design trees for a given program. These provide views of the program on multiple levels of abstraction.

Automating program recognition is difficult due to the wide range of possible variations among programs. An instance of a cliché may appear in a wide variety of forms in the text of a program, depending on the constructs or programming language chosen to express their data and control flow. The effort to encode these variations in a cliché library or to perform canonicalizing transformations to the *program text* tends to limit the variability and complexity of the structures that can be recognized. Also, delocalized clichés pose a serious problem.

Our graph-based approach overcomes these problems by shifting the representation of programs and clichés from *text* to a *flow graph*. `GRASPR` is able to overcome many of the difficulties of syntactic variation and noncontiguousness which hinder recognition. The flow graph representation abstracts away the syntactic features of the code, exposing the program's algorithmic structure. It concisely captures the data and control flow of programs, while suppressing details of the language in which they are expressed. Also, many clichés that are delocalized in the program text are much more localized in the flow graph representation. The flow graph formalism also provides a firm, mathematical basis for a well-defined, algorithmic recognition process that is more robust and easier to analyze and evaluate than previous, more ad hoc approaches.

The next two sections of this paper describe flow graphs and flow graph grammars more formally. This is followed by a formulation of the program recognition problem as flow graph parsing. It focuses on how variations in programs and clichés due to structure sharing are handled. It also describes how aggregation relationships are encoded and how variations in the way aggregate data structures are organized and nested is handled. The chart flow graph parser is then briefly described, followed by an evaluation of its efficiency and of the expressiveness of the flow graph formalism for representing programs and clichés.

2 Flow Graphs

We formulate the program recognition problem in terms of solving a parsing problem for flow graphs. To do this, we are building upon the flow graph formalisms of Brotsky [1] and Lutz [15]. A *flow graph* is an attributed, directed, acyclic graph, whose nodes have *ports* – entry and exit points for edges. Flow graphs have the following properties and restrictions:

1. Each node has a *type* which is taken from a vocabulary of node types.
2. Each node has two disjoint tuples of ports, called its *inputs* and *outputs*. Each port has a *type*, taken from a vocabulary of port types. All nodes of the same type have the same number and type of ports in their input and output port tuples. The size of the input (resp. output) port tuple of a node is called the *input* (resp. *output*) *arity* of the node.
3. A node's inputs (or outputs) may be empty, in which case the node is called a *source* (or *sink*, respectively).
4. Edges do not merely adjoin nodes, but rather edges adjoin ports on nodes. (This is important in representing programs, where input and output ports represent distinct inputs and outputs of functions.) All edges run from an output port on one node to an input port on another node. The ports connected by an edge must have the same port type.
5. More than one edge may adjoin the same port. Edges entering the same input (resp. output) port are called *fan-in* (resp. *fan-out*) *edges*.

6. Ports need not have edges adjoining them. Any input (or output) port in a flow graph that does not have an edge running into (or out of) it is called an *input* (or *output*) of that graph.
7. Each flow graph has a vocabulary of attributes, which is partitioned into two disjoint sets of node attributes and edge attributes. Each attribute has a (possibly infinite) set of possible values.

Notions of flow graphs and flow diagrams have appeared frequently in the literature for more than 20 years. However, our specific type of flow graph was first defined by Brotsky [1], drawing upon the earlier work on *web grammars* [4, 16, 17, 19, 23]. Wills [22, 25] extended Brotsky's definition so that flow graphs can include sinks and sources, fan-in and fan-out edges, and attributes.

Our flow graph formalism is related to that of Lutz, which in turn is equivalent to *hypergraph* formalisms with *hyperedge replacement* [8]. More specifically, Lutz's "flowgraphs" are a special type of our flow graph. (They derived from research on *plex* languages [7].) In addition to nodes, ports, and edges, they contain *tie-points*, which are intermediate points through which ports are connected to each other. Since each port is connected to exactly one tie-point, fan-in and fan-out are not captured to the same level of granularity as is captured by our flow graphs. For example, they cannot express the situation in which an output port p_1 fans out to input ports p_3 and p_4, while output port p_2 is only connected to p_4. Hypergraphs are analogous to Lutz's flowgraphs: nodes in a hypergraph correspond to tie-points and hyperedges correspond to flowgraph nodes.

3 Flow Graph Grammars

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.)

In addition, the flow graph grammar may be attributed: Each rule specifies how to compute attribute values from the attributes of nodes and edges in the rule. Each rule also imposes constraints on the attributes of the rule's nodes. Every flow graph in the language of an attributed grammar has attribute values that satisfy the constraints of the rules generating the flow graph.

More precisely, a flow graph grammar G has four parts: two disjoint sets N and T of node types, called non-terminals and terminals, respectively, a set P of *productions*, and a set S of distinguished non-terminal types, called the *start types* of G.

Each production in P consists of the following parts:

1. A flow graph L, called the *left-hand side*, containing a single node having a non-terminal type.
2. A flow graph R, called the *right-hand side*, containing nodes of non-terminal or terminal types.

3. A binary *embedding relation* C that specifies the correspondence between the ports of L and R.
4. A set of *attribute conditions* that impose constraints (in the form of relations) on the attribute values of nodes and edges in R.
5. A set of *attribute transfer rules*, each of which specifies the value of an attribute of L's node in terms of the attributes of the nodes and edges in R.

3.1 Embedding Relation

A binary embedding relation C relates each left-hand side port to a *tuple* of right-hand side and left-hand side port *sets*, where the position in the tuple is significant and the size of the tuple is ≥ 1.

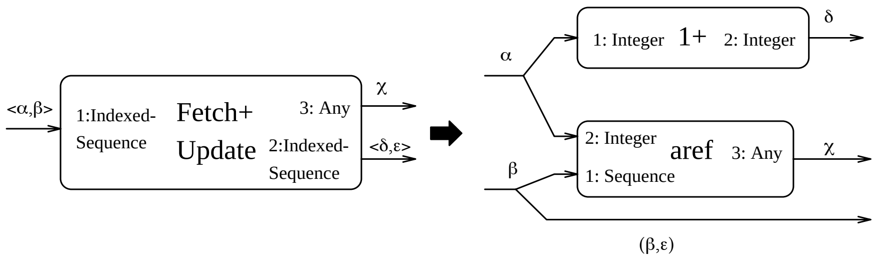

Fig. 3. A grammar rule encoding a clichéd operation on an aggregate data structure. (Attribute conditions and transfer rules are not shown.)

For example, Figure 3 illustrates a graph grammar rule that encodes the cliché Fetch+Update. This is a common implementation of the cliché Stack-Pop in which the Stack is implemented as an Indexed-Sequence (which has two parts: a base sequence and an integer index pointing to the "top" element). Fetch+Update accesses the base sequence and increments the index. This is encoded in the right-hand side of the rule for Fetch+Update. The embedding relation specifies how the inputs and outputs of the right-hand side are aggregated to form the Indexed-Sequences that are the input and an output of the Fetch+Update operation. The embedding relation is shown pictorially in the rule by corresponding Greek letters, where each Greek letter denotes a set of right- or left-hand side ports (e.g., α denotes the set of ports $\{1+_1, \text{aref}_2\}$).

A left-hand side port l_i and a right-hand side port or another left-hand side port p_j are said to "correspond" if $(l_i, t_j) \in C$ and p_j is a member of a set in the tuple t_j. Intuitively, the nonsingleton sets denote fan-in or fan-out of the right-hand side inputs and outputs; the tuples represent aggregation relationships. When a left-hand side port l_1 corresponds with another left-hand side port l_2, the rule is said to contain a *straight-through*. These are used in representing clichéd operations in which some of the input data is not acted upon, but passes directly to the output, as is the case with the base sequence part of the Indexed-Sequence input and output of Fetch+Update. Further details of how the embedding relation is restricted are given in [26].

4 Partial Program Recognition as Flow Graph Parsing

The *subgraph parsing problem* for flow graphs is: Given a flow graph F and a context-free flow graph grammar G, find all 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 (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.

To solve the subgraph parsing problem, `GRASPR` uses a graph parser which has evolved from Earley's string parsing algorithm [5] and string chart parsing. It incorporates four key improvements:

1. generalization of string parsing to flow graph parsing (Brotsky [1], Lutz [15]);
2. generalization of the control strategy to allow flexibility in the rule-invocation and search strategies used (Kay [11], Thompson [24], Lutz [15], Wills [27]);
3. extension of the grammar formalism to handle variation in graphs due to *structure-sharing* (Lutz [15], Wills [25, 26]), which is useful in dealing with variation due to common function-sharing optimizations; and
4. extension of the grammar formalism to capture *aggregation* relationships (Wills [26]) between single inputs or outputs of a left-hand side node and a tuple of inputs or outputs of a right-hand side sub-flow graph.

The fourth improvement 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. This is used, for example, in encoding the Fetch+Update cliché shown in Figure 3.

Other parsers have been developed which are related to our first two extensions described above. Bunke and Haller [2] and Peng, et al. [18] have both developed a parser for plex grammars which are generalizations of Earley's algorithm similar to Brotsky's. Wittenburg, et al. [28] give a unification-based, bottom-up chart parser which is similar to Lutz's and our chart parser. Grammar rules place a strict (total) ordering on the nodes in their right-hand sides. This specifies the order in which right-hand side nodes are matched (e.g., match the most salient node types first). This creates fewer partial analyses, which is advantageous in terms of efficiency, but is a drawback in terms of generating partial results when the graph contains unrecognizable sections.

Efficient parsers for *control* flow graphs have also been developed and applied to problems in global dataflow analysis [6] and program restructuring [14]. These parsers are able to take advantage of the fact that their reduction rules have the finite Church-Rosser property, which gives rise to a linear-time parsing process. These parsers also aim for a single full parse of the entire control flow graph. `GRASPR`, on the other hand, finds all recognizable subgraphs in order to deal with programs that are not constructed completely of clichés. This partial recognition process is inherently more expensive, but more flexible in dealing with code that contains novel or buggy structures.

4.1 Share-Equivalence

Following Lutz [15], we expand the language of a flow graph grammar to include all flow graphs derivable not only from a start type of the flow graph grammar, but also from flow graphs that are "share-equivalent" to a sentential form[1] of the grammar. The notion of share-equivalence is used to deal with variation due to *structure-sharing*: in a structure-sharing flow graph, a node plays the role of more than one node of the same type by generating output that fans out or by receiving input that fans in.

Share-equivalence is defined in terms of a binary relation *collapses* (denoted $\lhd$) on flow graphs. Flow graph F_1 collapses flow graph F_2 if and only if there are two nodes n_1 and n_2 of the same node type t in F_2, having input arity I and output arity O, such that all of these conditions hold:

1. Either one or both of the following are true:
   (a) $\forall i = 1 ... I$, the i^{th} input port of n_1 is connected to the same set of output ports as the i^{th} input port of n_2.
   (b) $\forall j = 1 ... O$, the j^{th} output port of n_1 is connected to the same set of input ports as the j^{th} output port of n_2.
2. F_1 can be created from F_2 by replacing n_1 and n_2 with a new node n_3 of type t with the i^{th} input (resp., output) of n_3 connected to the union of the ports connected to the i^{th} inputs (resp., outputs) of n_1 and n_2. We call this operation "zipping up" $F2$, and its inverse "unzipping."
3. The attribute values of n_1 and n_2 can be "combined." This is done by applying an *attribute combination function*, which is defined for each attribute, to the attribute values of n_1 and n_2. The attribute combination functions may be partial functions. If the function is not defined for n_1 and n_2's attributes, then the attribute values cannot be combined (and F_1 does not collapse F_2).

The reflexive, symmetric, transitive closure of collapses, $\lhd^*$, defines the equivalence relation *share-equivalent*. The *directly derives* relation ($\Rightarrow$) between flow graphs is redefined as follows. A flow graph F_1 directly derives another flow

[1] A *sentential form* of a graph grammar is any flow graph that is derivable from a start type of the grammar by the application of zero or more productions of the grammar.

graph F_2 if and only if either F_2 can be produced by applying a grammar rule to F_1, $F_1 \lhd F_2$, or $F_2 \lhd F_1$.

As in string grammars, the reflexive, transitive closure of $\Rightarrow$, is the *derives* relation ($\Rightarrow^*$). The language of a flow graph grammar G (denoted $L(G)$) is the set of all flow graphs, whose nodes are of terminal type and which can be derived from a start type of G. Thus, the notion of a language of a flow graph grammar G has been extended to include flow graphs that are generated by a series of not only production rule applications but also zip-up and unzipping transformations. Since a zip-up or unzipping step can happen anywhere in the derivation sequence, the language of a graph grammar G in this extended formalism is a superset of the set of flow graphs share-equivalent to flow graphs in the "core" language of G in the unextended formalism.

Both generators and parsers for the language of a flow graph grammar can interleave zipping and unzipping transformation steps with their usual expansion and reduction steps. The parser used by the program recognition system reported here simulates the introduction of these transformations into its reduction sequence, as is described in Section 5.1.

4.2 Aggregation-Equivalence

Grammar rules in our flow graph formalism specify how a non-terminal node can be rewritten as a particular grouping of terminal and non-terminal nodes (in the form of a flow graph). We now extend it to also specify how a single input or output of a non-terminal node can correspond to an *aggregation* of the inputs or outputs of a flow graph to which the non-terminal node is rewritten. We define an additional, *aggregation-equivalence* relation to relate flow graphs that differ only in how they aggregate port types. The language of the flow graph grammar now includes all flow graphs aggregation-equivalent to flow graphs derivable from a start type of the grammar.

A simple way to capture the aggregation of port types into fewer, more abstract port types is to use special *Make* and *Spread* nodes. A Make node represents an aggregate type constructor: the tuple of its input port types compose its single output port type. A Spread node represents aggregate type selectors: its input port type is decomposed into its tuple of output port types.

Aggregation-equivalence captures the equivalence between flow graphs that differ solely in the way port types are aggregated within the graphs, i.e., in the order and nesting of aggregation (aggregation is commutative and associative) and whether there is any aggregation at all.

We define the reflexive, symmetric, transitive relation *aggregation-equivalent* as follows. A flow graph F_1 is aggregation-equivalent to another F_2 (denoted $F_1 \equiv_A F_2$) if and only if there exists a flow graph F_3, such that F_1 and F_2 can each be transformed to a flow graph isomorphic to F_3, using a (possibly empty) sequence of the following transformations: the permutation of part port tuples in Spreads and Makes, the flattening of compositions of Spread (resp. Make) nodes into a single Spread (resp. Make) node, the replacement of any composition of corresponding Spread and Make pairs (in either order) with the equivalent edges, and the removal of any Spread node whose input is an input of the flow graph or any Make node whose output is an output of the flow graph. We call the first type of transformation the *permutation* transformation. The rest of the transformations are *aggregation-removal* transformations and their inverses are called *aggregation-introduction* transformations.

A generator or parser for the language of a flow graph grammar may perform the permutation, aggregation-introduction and aggregation-removal transformations as steps in their derivation or reduction sequence. Because there are many possible orderings in which to apply the transformations and because doing this efficiently involves an extension to the embedding relation of the graph grammar formalism, it is important to discuss how such a recognizer is constructed.

One way a recognizer for the language can work, given an input flow graph F, is in two stages. The first would apply some sequence of the permutation, aggregation-removal and aggregation-introduction transformations to F to produce a flow graph F', while the second would apply a recognizer for the core language to F'. A flow graph F would be recognized if a sequence of transformations is found which yields a new flow graph F' that is accepted by a recognizer for the core language. Unfortunately, the first stage could involve a great deal of search to find the appropriate transformation sequence.

A more promising approach is to divide up the stages differently so that no choices need to be made. In the first stage, only aggregation-removal transformations that work "downward" by creating less-aggregated flow graphs are applied until a minimally-aggregated flow graph is obtained. (Note that in the minimally-aggregated flow graph, there are residual Make and Spread nodes only if there are terminal nodes that have aggregate port types.) Then in the second stage, the aggregation-introduction and permutation transformations are interleaved with the reduction actions of the recognizer for the core language. The idea is that the grammar rules can provide guidance as to what to aggregate and how to organize the aggregation so that the flow graph will be recognizable as a member of the core language. This approach is taken by our parser.

5 Chart Parsing Algorithm

We have developed a chart parsing algorithm for solving the subgraph parsing problem for flow graphs. Chart parsers maintain a database, called a *chart*, of partial and complete analyses of the input. The elements in the chart are called *items*. (In string chart parsing, they are called "edges." Lutz [15] calls them "patches.") An item might be either complete or partial. Complete items represent the recognition of some terminal or non-terminal in the grammar. Partial items represent a partial recognition of a non-terminal.

The basic operation of a chart parser is to create new items by combining a partial item with a complete one. This is called the *fundamental event*. If there is a partial item that needs a non-terminal A at a particular location and if there is a complete item for non-terminal A at that location, then the partial item can be *extended* with the complete item. During extension, a copy of the partial item is created and augmented. This results in a new item which is added to the chart. Items are never removed from and duplicate items are never added to the chart. This avoids redoing work and guarantees termination.

The parser continually generates items, conceptually in parallel, but to implement the algorithm on a sequential machine, we use an agenda to queue up the items to be added to the chart. As an item is pulled from the agenda and added to the chart, 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.

The agenda makes it easy to control which things are added to the chart and when they are added. This explicit control can be used to enforce a particular rule invocation strategy or search strategy. In fact, the parser has several "control knobs" that can be set to achieve a desired control strategy or to implement various focusing heuristics [27]. These include parameters like bottom-up or top-down rule invocation, the criterion used to determine whether one item can extend another, the ordering in which right-hand side nodes are matched, and the ordering of attribute condition checking in general.

The chart parser also has chart monitors that trigger on opportunities to create additional, new views of selected sub-flow graphs. These alternative views can be used, for example, to canonicalize certain sub-flow graphs or fix and resume unsuccessful matches. They can also be used as question-triggering patterns to elicit advice from external agents. An important use of additional monitors is in performing "zip-up" steps in parsing share-equivalent flow graphs.

5.1 Recognizing Share-Equivalent Flow Graphs

A parser for a structure-sharing flow graph grammar must interleave zipping and unzipping transformation steps with the usual reduction steps. Our chart parser *simulates* this introduction in two ways. First, the grammar is made to maximally share by canonicalizing all right-hand sides and by allowing sub-derivations to be shared. Second, the input graph is made to maximally share by using a "zip-up" monitor. All items that represent nodes that are collapsible are merged into a new item representing the result of the zip up.

5.2 Recognizing Aggregation-Equivalent Flow Graphs

Recall from Section 4.2 that a recognizer for the flow graph grammar's language must interleave permutation, aggregation-introduction, and aggregation-removal steps into the reduction sequence. During recognition, Spread and Make nodes must be "inserted" whenever an isomorphic occurrence of a right-hand side is reduced to a left-hand side non-terminal with aggregate ports. The Spread and Make nodes serve to bundle up the edges surrounding the non-terminal node. The recognition process must also "simplify" any composition of Makes and Spreads that results from aggregation-introduction steps. These actions are simulated by our flow graph chart parser.

In particular, items keep track of where the right-hand side is found, using a set of *location pointers*, which indicate which edges correspond to the inputs and outputs of the right-hand side of the item's rule. To represent the addition of a Make or Spread, the location pointers are placed in tuples, which are nested in tree structures. The nested tuples reflect the organization of the aggregation of the edges to which they refer. An element of the tuple can be either another tuple or a set of location pointers. (A set of more than one location pointer represents fan-in or fan-out.) When items are combined, their location pointers are compared to see if they represent a Make/Spread composition that simplifies correctly. The corresponding parts of the tuples are compared. If both parts are tuples, they are compared recursively. If both are sets, the sets must have a non-empty intersection for the comparison to succeed. If one is a set and the other a tuple, the comparison fails.

This is a brief description; a fuller account of how the parser deals with structure-sharing and aggregation is given in [26], including how residual Spreads and Makes are handled and how straight-throughs are matched.

6 Efficiency

We are studying the graph parsing approach by experimenting with two real-world simulator programs, written in Common Lisp by parallel processing researchers [3]. These programs are in the 500 to 1000 line range. The largest program recognized by any other existing recognition system is a 300-line database program recognized by CPU[13]. All other systems work with "toy programs" on the order of tens of lines.

We empirically and analytically studied the computational cost of `GRASPR`'s parsing algorithm with respect to the simulator programs. `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 right-hand side node 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 [26], so it is unlikely that there is a subgraph parsing algorithm that is not worst-case exponential.

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 and are highly varied, reducing ambiguity; 2) edge connection constraints, which represent dataflow dependencies and tend to be sparse, so fewer pairs of incorrect matches between nodes satisfy these constraints; and 3) co-occurrence constraints, which are a class of control flow constraints that are especially powerful in reducing ambiguity in recognizing clichéd operations on aggregate data structures.

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. Fortunately, the complexity of near-miss recognition can be controlled by using grammar indexing and flow graph partitioning advice and controlling the application order of constraints [26, 27].

7 Expressiveness

The flow graph representation is able to suppress many common forms of program variation which hinder recognition. In particular, the flow graph formalism and graph parser enable GRASPR to be robust under many types of variation, including *syntactic* (e.g., differences in binding or control constructs chosen and in statement ordering), *organizational* (e.g., differences in procedural modularization and data structure nesting), and *implementational* variation (e.g., differences in algorithms chosen to achieve common abstract operations). It is also robust under variation due to delocalization, unfamiliar code, and common function-sharing optimizations.

We have used flow graph grammars to concisely encode algorithmic and data aggregation clichés whose constraints are primarily based on data and control flow. 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 were acquired by manually extracting and generalizing commonly occurring patterns in a corpus of example programs and by speaking with the designers of the simulator programs to codify their experience with typical algorithms and data structures in the simulation domain. We also gathered clichés from textbooks in general computer programming as well as the area of simulation. Clichés in our library include algorithmic computations, such as list enumeration, binary search, instruction-fetch, decode, and execute and event-driven simulation, as well as common data structures, such as priority queue and hash table. These are encoded in approximately 200 graph grammar rules. 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.

GRASPR 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 variable assignments. With the exception of CPU [13], existing recognition systems cannot handle aggregate data structure clichés and a majority do not handle recursion. Side effects to mutable data structures still present an open problem in program recognition, but see [26] for future directions in interleaving dataflow analysis with the recognition of stereotypical aliasing patterns.

8 Future Directions

One reason we developed a parsing algorithm with flexible control is that we wanted to complement our purely code-driven recognition with other design recovery techniques based on information from other sources, such as comments, identifier names, documentation, specifications, and testing suites. The parser's flexibility allows GRASPR to accept advice and heuristic guidance from external agents. In the future, such a hybrid system will allow us to explore the interactive potential of the chart parser and extensions for incremental analysis to which the chart parser lends itself. Another interesting future direction is the application of GRASPR to multiple tasks that require program understanding. This will help us determine which constraints various tasks place on the recognition process and representational formalism and what new types of control strategies are needed. Finally, our empirical studies have been limited to a few programs with a cliché library that co-evolved with GRASPR. More empirical studies are needed to expand and refine the cliché library, identify more classes of variation that can be tolerated, refine our understanding of the parser's performance in the context of practical reverse engineering problems, and evaluate the ability of the existing system to recognize clichés in new programs.

Acknowledgments: I am grateful to Chuck Rich, Richard Waters, David McAllester, and Peter Szolovits for guidance and insights. The comments from anonymous reviewers were also valuable. The research described here was conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's 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. D. Brotsky. An algorithm for parsing flow graphs. Technical Report 704, MIT Artificial Intelligence Lab., March 1984. Master's thesis.
2. H. Bunke and B. Haller. A parser for context free plex grammars. In M. Nagl, editor, *15th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, pages 136–150. Springer-Verlag, June 1989. Lecture Notes in Computer Science Series, Vol. 411.
3. W. Dally, A. Chien, S. Fiske, W. Horwat, J. Keene, M. Larivee, R. Lethin, P. Nuth, S. Wills, P. Carrick, and G. Fyler. The J-Machine: A fine-grain concurrent computer. In *Int. Fed. of Info. Processing Societies*, 1989.
4. P. Della-Vigna and C. Ghezzi. Context-free graph grammars. *Information and Control*, 37(2):207–233, 1978.
5. J. Earley. An efficient context-free parsing algorithm. *Comm. of the ACM*, 13(2):94–102, 1970.
6. R. Farrow, K. Kennedy, and L. Zucconi. Graph grammars and global program data flow analysis. In *Proc. 17th Annual IEEE Symposium on Foundations of Computer Science*, pages 42–56, Houston, Texas, 1976.
7. J. Feder. Plex languages. *Information Sciences Journal*, 3:225–241, 1971.
8. A. Habel. *Hyperedge Replacement: Grammars and Languages.* Springer-Verlag, New York, 1992. Lecture Notes in Computer Science Series, Vol. 643.

9. J. Hartman. Automatic control understanding for natural programs. Technical Report AI 91-161, University of Texas at Austin, 1991. PhD thesis.
10. W. L. Johnson. *Intention-Based Diagnosis of Novice Programming Errors.* Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.
11. 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.
12. V. Kozaczynski and J.Q. Ning. Automated program understanding by concept recognition. *Automated Software Engineering*, 1(1):61–78, March 1994.
13. S. Letovsky. Plan analysis of programs. Research Report 662, Yale University, December 1988. PhD Thesis.
14. U Lichtblau. Decompilation of control structures by means of graph transformation. In H. Ehrig, editor, *International Joint Conference on Theory and Practice of Software Development*, pages 284–297. Springer-Verlag, 1985. Lecture Notes In Computer Science Series, Vol. 185.
15. R. Lutz. Chart parsing of flowgraphs. In *Proc. 11th Int. Joint Conf. Artificial Intelligence*, pages 116–121, Detroit, Michigan, 1989.
16. U. G. Montanari. Separable graphs, planar graphs, and web grammars. *Information and Control*, 16(3):243–267, March 1970.
17. T. Pavlidis. Linear and context-free graph grammars. *Journal of the ACM*, 19(1):11–23, January 1972.
18. K. Peng, T. Yamamoto, and Y. Aoki. A new parsing algorithm for plex grammars. *Pattern Recognition*, 23(3-4):393–402, 1990.
19. J. L. Pfaltz and A. Rosenfeld. Web grammars. In *Proc. 1st Int. Joint Conf. Artificial Intelligence*, pages 609–619, Washington, D.C., September 1969.
20. A. Quilici. Memory-based approach to recognizing programming plans. *Comm. of the ACM*, 37(5):84–93, May 1994.
21. C. Rich and R. C. Waters. *The Programmer's Apprentice.* Addison-Wesley, Reading, MA and ACM Press, Baltimore, MD, 1990.
22. C. Rich and L. M. Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, 7(1):82–89, January 1990.
23. A. Rosenfeld and D. Milgram. Web automata and web grammars. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 307–324. John Wiley and Sons, New York, 1972.
24. H. Thompson. Chart parsing and rule schemata in GPSG. In *Proc. 19th Annual Meeting of the ACL*, Stanford, CA, 1981.
25. L. Wills. Automated program recognition: A feasibility demonstration. *Artificial Intelligence*, 45(1-2):113–172, 1990.
26. L. Wills. Automated program recognition by graph parsing. Technical Report 1358, MIT Artificial Intelligence Lab., July 1992. PhD Thesis.
27. L. Wills. Flexible control for program recognition. In *Proc. 1st Working Conference on Reverse Engineering*, Baltimore, MD, May 1993.
28. K. Wittenburg, L. Weitzman, and J. Talley. Unification-based grammars and tabular parsing for graphical languages. Technical Report ACT-OODS-208-91, MCC, June 1991.

This article was processed using the LaTeX macro package with LLNCS style