ISSN:0975-9646

Detection and Verification of Program Behavior : A Graph Grammar Approach

M.Sivalakshmi, T.M.Devi, K.AyyappaRaja
Syed Ammal Engineering College, Ramanathapuram

***Abstract* – In this work, a semi-automatic graph grammar approach is developed to retrieving the hierarchical structure of the program behavior. The hierarchical structure is built on recurring substructures in a bottom-up fashion. Formulate the behavior discovery and verification problem as a graph grammar induction and parsing problem, i.e., automatically iteratively mining qualified patterns and then constructing graph rewriting rules. The syntax of the grammar represents the behavioral properties of that program. Source code realizing certain functionality could be reused for different programs. Therefore, programs may have similar behavioral properties when they execute similar functionalities. Furthermore, using the induced grammar to parse the behavioral structure of a new program could verify if the program has the same behavioral properties specified by the grammar.**

I*ndex Terms*— **Visual language, graph grammar induction, program comprehension, reengineering.**

1. INTRODUCTION

With the wide deployment of software systems, software maintenance has become a challenging and costly task due to increasing software size and complexity, incomplete and incorrect documentation, Software maintainers usually need to understand a system before making changes. The program behavior mining and verification problem develop a visual language perspective using graph grammar induction and parsing techniques. We exploit the power of a graph grammar in specifying information visually with a precise meaning. More especially, method calls in the execution trace of a program can be naturally represented as a graph in which nodes represent methods and dges indicate method calls. Given such a graph, searching for frequent calling patterns assists in discovering the behavior of a program. When the call graph is considered as a visual sentence, the discovery process is essentially a grammar induction. Integrating graph grammar with grammar induction can support an automatic analysis of a program behavior. The result with a visual presentation can improve user comprehension with a precise meaning.

A valid parsing result means that the new program satisfies the same behavioral properties as the old program. Consequently, two types of behavior verification can be performed. verify acceptable call sequences in a scenario and detect illegal behaviors or security related activities.

A graph grammar induction algorithm iteratively finds common substructures from the given graph and organizes the hierarchical substructures in a grammatical form. When a common frequent substructure is found, a production will be created. The substructure consisting of terminal and nonterminal symbols identified from the graph is represented as the right graph of the production, and new nonterminal symbols will be created as the left graph. Then, the new production will be applied to the current data set.

2. LITERATURE SURVEY

2.1 *Constructing VEGGIE: Machine learning for context Sensitive Graph Grammars*

Context-sensitive graph grammar construction tools have been used to develop and study interesting languages. However, the high dimensionality of graph grammars results in costly effort for their construction and maintenance. Additionally, they are often error prone. These costs limit the research potential for studying the growing graph based data in many fields. Context-sensitive induction is extended from the context-free induction of overlapping single node recursive production rules. Overlap between instances of common substructures provides a connection context between those instances. Mapping the substructure defined by the overlap to the instances provides the means for extracting embedding information used in a production rule to create a context-sensitive production rule; the induction process must identify substructures within the host graph that cannot be simply reduced to a single non-terminal as is done for context-free grammars.

These substructures must contain some property that challenges the induction of a context-free production rule. To ensure the halting condition, an inferred context-sensitive production rule must reduce two or more RHS nodes to a single LHS non-terminal node. This ensures that the LHS size is less than the RHS size. A requirement of the halting condition .Non-terminal nodes are generated during the induction process. As there are not predefined node types, they are added by the induction system. These non-terminals are free of multiple port definitions as a context-free induction system only relies on the default ports.

2.2 Graph Grammar Induction on Structural Data for Visual Programming

This work proposed a methodology and instrumentation infrastructure toward the reverse engineering of UML (Unified Modeling Language) sequence diagrams from

dynamic analysis. However, such reverse-engineered dynamic models can also be used for quality assurance purposes. Another advantage is that our metamodels and rules can then naturally be used as specifications to develop tool prototypes. A more sophisticated analysis of the source code to identify the different objects involved in interactions is presented.

In particular, the technique is able to recognize when one call site always refers to the same runtime object, or when two different call sites do so, in which case, only one diagram object is displayed, thus resulting in an accurate diagram. The initial class structure of our prototype was indeed a direct reflection of our metamodels. It is also important to note that if our reverse engineering process were to be adapted to a different distribution middleware, the metamodels and rules would remain unchanged, except that timestamps might be measured in a different way.

2.3 Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software

This work introduces an algorithm, which is applied to trees with labels on nodes and directed unlabeled edges. The trees were generated from the structural representation of a computer program and XML files. The graph grammar inference algorithm was used to infer grammars from these trees. The partial program example demonstrated the application of the induction algorithm to rapid grammar development for VPLs. A graph is defined with labeled nodes and edges. Every edge of the graph can be directed or undirected. The definition of a graph grammar is described as the class of grammars that can be inferred by the induction method, which is currently limited to Context-free grammars.

The main characteristic of the inferred grammar productions is that they are recursive productions. Recursive productions are inferred such that, for a given production, a nonterminal node label on the left side appears one or more times in the node labels of the graph on the right side. The method can also infer non-recursive Productions which are frequent, non-overlapping sub graphs of an input graph. Here, the author extends our grammar to context-sensitive graph grammars (CSGGs) and to study the induction algorithm's performance in rapidly developing VPLs.

3. PROGRAM INDUCTION AND VERIFICATION

3.1 Execution trace

Execution traces of method invocations are collected during program execution. We construct call graphs from the execution traces using Abstracer and then abstract the call graph based on abstraction criteria. Abstracer removes execution traces that play less significant roles in representing the program behavior and then represents the abstracted call graph in GraphML. The abstracted call graph is the input to the grammar induction engine of VEGGIE. VEGGIE can automatically derive a set of graph rewriting rules from a given graph using a compression-based substructure mining algorithm.

In the current implementation, the following information are recorded for each method invocation:

1. Names of classes, objects, and methods
2. Method invocation: enter-exit of every static or nonstatic method.
3.

The abstraction ensures equivalent behavior semantics between the original call graph and the abstracted one, and allows users to focus on the activities. The abstraction on loops and low-level methods satisfies the safe property, meaning that methods in an abstracted scenario comply with the causality properties of the original call graph. In other words, the causal relationship between any two methods in an abstracted scenario S' remains if there exists a causal relationship between the two methods in the uncompressed scenario S. Therefore, our behavior pattern mining and verification can be performed on abstracted call graphs.

3.2 Call graph Construction

A behavioral pattern describes activities that happen in an order. To reflect this, we assign nodes in a call graph with temporal attributes. Without a temporal order, the inferred common patterns may not be correct even if they are graphically isomorphic. We use logical time stamps to keep track of methods' order. According to the method calling order the call graph is constructed

4. BEHAVIORAL STRUCTURE DISCOVERY AND VERIFICATION

Graph grammar induction uses graph-based substructure mining algorithms instead of text mining techniques. A substructure is defined as a representation of recurring sub graphs. An instance is one occurrence of such a substructure in the graph data set. The substructures recognized by the grammar induction procedure reveal hidden recurrent patterns within the graph data set. The hierarchical relations within the grammar can aid the developers in understanding and analyzing the composition of large and complex legacy systems. Those grammars can also be used to build graphs to simulate the execution of a system. VEGGIE emphasizes on the compressing of graph data sets rather than purely searching for the frequent subgraphs. The compression ratio for each substructure is calculated based on a minimum description length (MDL), and the substructure with the highest compression ratio among the competing substructures is selected. Therefore, the substructure found in each iteration may not be the most frequent one, but it can achieve the best compression ratio for the given graph, i.e., the ratio between the original and resulting graphs after the subgraphs is replaced with a nonterminal node.

5 RESULTS

Fig 1 shows an abstracted graph after removing the loop on method addOne. In order to eliminate low-level details, our approach allows a user to specify a threshold. If a method has a call depth that is greater than the user specified threshold in a call chain, this method is pruned.

Fig 2 shows an conflict between methods in abstract graph. Each method have some rules to find the conflict between them.

Fig 3 shows grammar for order topping to find any conflict in their rules and find dependency about method in abstract graph using graph grammar approach.

Fig 4 shows the complete parse result of the program and their conflict,depencency and rules for their match

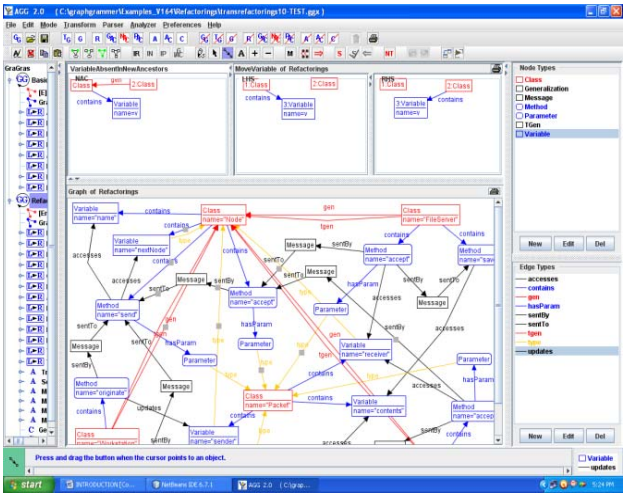


Fig. 1 Hierarchical behavior of the program

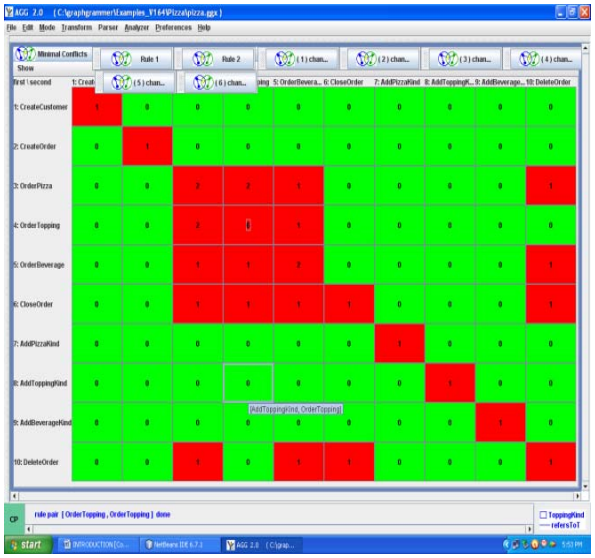


Fig 2 shows the rules of conflict

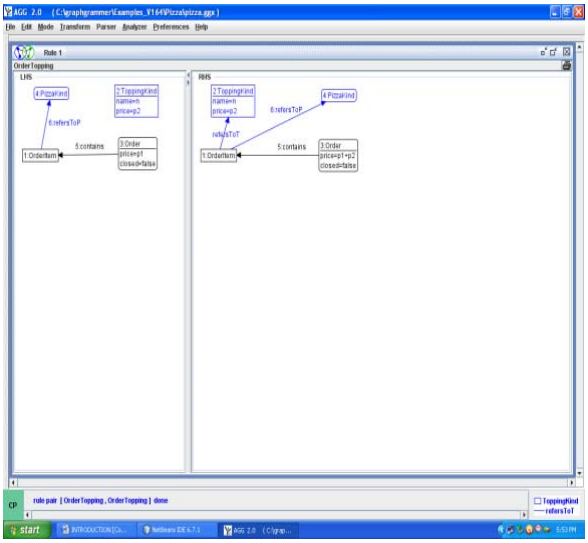
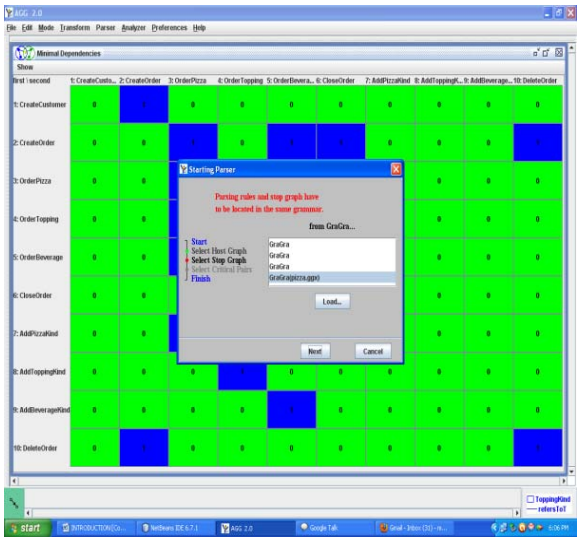


Fig 3 shows the rule1 of order topping conflict



. Fig 4 shows the parsing result for the graph

CONCLUSIONS

This proposed scheme develops the configuration code of a program and a call graph construction. The proposed algorithm develops a call graph using a abstractor tool. The future work is to identify the call graph and verify the program behavior using a graph grammar. We represent program behavior as a call graph, and apply the Spatial Graph Grammar formalism to discover and analyze the behavior pattern within the call graph. An inferred

graph grammar and a syntactic parse tree visually represent the hidden structures of the program behavior at different abstraction levels. The substructures found by a grammar induction algorithm are reusable software components.

REFERENCES

[1] K. Ates, J.P. Kukluk, L.B. Holder, D.J. Cook, and K. Zhang, "Graph Grammar Induction on Structural Data for Visual Programming," Proc. 18th IEEE Int'l Conf. Tools with Artificial Intelligence, pp. 232- 242, Nov. 2006.

[2] K. Ates and K. Zhang, "Constructing VEGGIE: Machine Learning for Context-Sensitive Graph Grammars," Proc. 19th IEEE Int'l Conf. Tools with Artificial Intelligence, pp. 456-463, Oct. 2007.

[3] L. Baresi, R. Heckel, S. Tho¨ne, and D. Varro´ , "Modeling and Validation of Service-Oriented Architectures: Application vs. Style," Proc. 11th European Software Eng. Conf. held jointly with Ninth ACM SIGSOFT Int'l Symp. Foundations of Software Eng., pp. 68-77, Sept. 2003.

[4] L. Baresi and R. Heckel, "Tutorial Introduction to Graph Transformation: A Software Engineering Perspective," Proc.First Int'l Conf. Graph Transformation, 2002.

[5] H.A. Basit and S. Jarzabek, "Detecting Higher-Level Similarity Patterns in Programs," Proc. 10th European Software Eng. Conf. held jointly with 13th ACM SIGSOFT Int'l Symp. Foundations of Software Eng., pp. 156-165, Sept. 2005.

[6] L.C. Briand, Y. Labiche, and J. Leduc, "Toward the ReverseEngineering of UML Sequence Diagrams for Distributed Java Software," IEEE Trans. Software Eng., vol. 32, no. 9, pp. 642-663,Sept. 2006.

[7] G. Casella, G. Costagliola, F. Ferrucci, G. Polese, and G. Scanniello, "Visual Languages for Defining Adaptive and Collaborative e-Learning Activities," Proc. IADIS Int'l Conf.: e-Soc. '04, vol. 1, pp. 243-250, July 2004.

[8] E.J. Chikofsky and J.H. Cross, II, "Reverse Engineering and Design Recovery: A Taxonomy," IEEE Software, vol. 7, no. 1, pp. 13-17, Jan. 1990.

[9] M. Christodorescu, S. Jha, and C. Kruegel, "Mining Specifications of Malicious Behavior," Proc. Sixth Joint Meeting of the European Software Eng. Conf. and ACM SIGSOFT Int'l Symp. Foundations of Software Eng., pp. 5-14, Sept. 2007.

[10] C. Csallner, Y. Smaragdakis, and T. Xie, "DSD-Crasher: A Hybrid Analysis Tool for Bug Finding," ACM Trans. Software Eng. And Methodology, vol. 17, no. 2, pp. 345-371, July 2008.

[11] G. Costagliola, V. Deufemia, and G. Polese, "A Framework for Modeling and Implementing Visual Notations with Applications to Software Engineering," ACM Trans. Software Eng. and Methodology, vol. 13, no. 4, pp. 431-487, Oct. 2004.

[12] G. Costagliola, A.D. Lucia, V. Deufemia, C. Gravino, and M. Risi, Design Pattern Recovery by Visual Language Parsing," Proc. Ninth European Conf. Software Maintenance and Reeng., pp. 102-111, Mar. 2005.

[13] G. Costagliola, V. Deufemia, F. Ferrucci, and C. Gravino, "Constructing Meta-CASE Workbenches by Exploiting Visual Language Generators," IEEE Trans. Software Eng., vol. 32, no. 3, pp. 156-175, Mar. 2006.

[14] G. Costagliola, V. Deufemia, and M. Risi, "Using Grammar- Based Recognizers for Symbol Completion in Diagrammatic Sketches," Proc. Ninth Int'l Conf. Document Analysis and Recognition,pp. 1078-1082, Sept. 2007.

[15] D.J. Cook and L.B. Holder, "Substructure Discovery UsingMinimum Description Length and Background Knowledge,"J. Artificial Intelligence Research, vol. 1, pp. 231-255, Feb. 1994.