2008 20th IEEE International Conference on Tools with Artificial Intelligence

Discovering Program's Behavioral Patterns by Inferring Graph-Grammars from Execution Traces

Chunying Zhao[1], Keven Ates[1], Jun Kong[2], Kang Zhang[1]
[1]The University of Texas at Dallas
{cxz051000, atescomp, kzhang }@utdallas.edu
[2]North Dakota State University
jun.kong@ndsu.edu

Abstract

Frequent patterns in program executions represent recurring sequences of events. These patterns can be used to reveal the hidden structures of a program, and ease the comprehension of legacy systems. Existing grammar-induction approaches generally use sequential algorithms to infer formal models from program executions, in which program executions are represented as strings. Software developers, however, often use graphs to illustrate the process of program executions, such as UML diagrams, flowcharts and call graphs. Taking advantage of graphs' expressiveness and intuitiveness for human cognition, we present a graph-grammar induction approach to discovering program's behavioral patterns by analyzing execution traces represented in graphs. Moreover, to improve the efficiency, execution traces are abstracted to filter redundant or unrelated traces. A grammar induction environment called VEGGIE is adopted to facilitate the induction. Evaluation is conducted on an open source project JHotDraw. Experimental results show the applicability of the proposed approach.

1. Introduction

Mining frequent patterns plays an important role in data analysis by identifying recurring and meaningful item sets in a large dataset. It has many successful applications in various research areas, such as natural language processing, DNA sequence interpretation, and social network analysis. In particular, research has shown several successful applications in software engineering. The literature on software engineering, especially in reverse engineering and program analysis, reflects a clear trend towards combining machine learning techniques with domain knowledge of software engineering. Frequent pattern mining is one of the important reverse engineering topics [10][11].

So far, conventional techniques for pattern mining in software engineering generally focus on recovering similar structures scattered in the program. The organization of these substructures, i.e., interactions and connections among these patterns and other components in the system, however, is also important, especially in dynamic analysis of program behaviors. The inference learning approaches, specifically grammar induction, can address this problem, because grammar induction is an iterative process of building a parse tree from given sentences in the language under study. The sentences can be considered as positive samples in the language so that a set of grammars can be inferred from the samples. When analyzing software behaviors, one can interpret events as tokens, and event streams as sentences in the language, then a natural analog becomes evident [3]. The intuition behind grammars is that a parse tree forms a hierarchical lattice, where a child node represents a more detailed substructure while a parent node is the abstraction of its children contents. As such, a hierarchical lattice formed by a grammar can reveal hidden structures of program behaviors. Discovering the hidden structures of program behaviors can ease the tasks of maintenance and comprehension. The inferred grammars can also help to model formal state machines to simulate program behaviors.

Researchers have used grammar inference techniques to discover program behaviors. A foundation work by Cook *et al.* [3] proposed using event-data in the form of an event stream, collected from software's execution, to infer a formal behavioral model. They cast the behavior discovery problem to the discovery of a grammar for a regular language from given example sentences in that language [3]. A most recent work by Walkinshaw *et al.* [13] inferred the state machine representation of a software execution using an interactive grammar inference approach from execution traces. Their work is based on the grammar inference algorithm (QSM) of Dupon *et al*. [4], which takes an initial manually generated scenario represented in strings as an input, and uses it as a basis to interactively generate a state machine for the whole system.

Existing approaches demonstrate the feasibility of using grammar induction to infer software behaviors from execution traces. They mostly derive grammars from sequential textual datasets, and do not take advantage of the graphical representation of program behaviors. Graphs have been used extensively for program representations, such as UML diagrams, flowcharts and call graphs, etc. Visual representations in diagrams and graphs sometimes convey more information than texts to human's cognition. Hence inferring grammars from graphical program behaviors is desirable.

1082-3409/08 $25.00 © 2008 IEEE
DOI 10.1109/ICTAI.2008.68

Graph grammar systems have been well-established for decades in graphical reasoning and parsing techniques. They are expressive in describing program's behaviors in terms of diagrammatic grammar rules. An inferred grammar can expose the hidden structure of a given graph dataset. Supplemented by a parsing system, the inferred grammar can be validated and used to automatically parse other datasets to verify structural properties.

We adopt VEGGIE [1][2], a *Visual Environment for Graph Grammars: Induction and Engineering*, to infer graph grammars from program execution traces. VEGGIE essentially incorporates two subsystems: SubdueGL [6][7] and SGG [8][9]. The former is a context-free graph-grammar induction system, and the latter is a context-sensitive graph-grammar parsing system. The current implementation of the integrated visual environment VEGGIE facilitates the (semi)automatic discovery of program behaviors represented in context-free graph-grammars. We are current extending the induction engine to handle context-sensitive grammars. We adapt SubdueGL's substructure matching algorithm by annotating graphs with temporal attributes since data describing program executions naturally have temporal attributes. To improve the induction performance, we use an abstraction scheme to reduce redundant and unrelated traces before the grammar induction.

The contribution of our work includes:

- A graph-grammar induction approach to discovering program's behavioral patterns.
- An abstraction scheme for enhancing the efficiency of grammar induction.
- An adapted graph-grammar implementation for the proposed approach, and experiments on an open source software JHotDraw.

The remainder of this paper is organized as follows: Section 2 introduces the background of visual languages including the Spatial Graph Grammar (SGG) [9] and graph-grammar induction. Section 3 describes an overview of the approach. Section 4 presents the graph-grammar approach to discovering the patterns of program behaviors with a running example. Section 5 reports the preliminary results of an experiment performed on an open source software JHotDraw. Section 6 reviews related work and Section 7 concludes the paper.

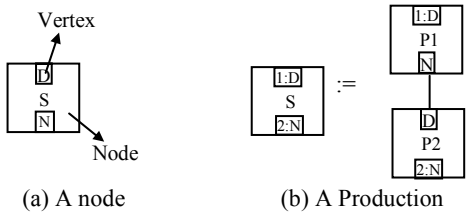
2. Background

2.1. Visual Languages and Graph Grammars

Visual programming languages allow developers to use graphical elements such as diagrams, boxes and arrows to represent program design and structures. Visual languages are advantageous over traditional text-based languages because of their expressiveness in visually representing structures and high-level patterns. The core concepts in visual languages include the construction and parsing of graph grammars. Graph grammars extend Chomsky's generative grammars into the domain of graphs. Different from string grammars expressing sentences in a sequence of characters, graph grammars specify syntactic structures in terms of diagrammatic rules. Each rule is called a *production* that consists of a *left graph* and a *right graph*.

A grammar can be context-free or context-sensitive. The difference between context-free and context-sensitive grammars is that the latter allows for more than one symbol in the left graph while the former allows for only one. The left graph of a context-sensitive grammar should be lexicographically smaller than the right graph to ensure the termination condition in parsing.

Each graph grammar has it own specifications. We use the Spatial Graph Grammar (SGG) [9] to illustrate related concepts. The SGG is a context-sensitive graph grammar formalism, capable of specifying various types of graphs with both logical and spatial types of relationships. The SGG formalism is expressed in a node-edge format as shown in Figure 1. Nodes are organized into a two-level hierarchy, where a large rectangle representing the node itself is the first level with embedded small rectangles as the second level called *vertices*. Figure 1(a) depicts a typical SGG node including two vertices. In a node, each vertex is uniquely labeled. A node can be viewed as a module, a procedure or a variable, depending on the design requirements and object granularities. A vertex functions as a port to connect other nodes by edges. Edges can denote any communications or relationships between nodes.



(a) A node (b) A Production

Figure 1 Spatial graph grammar representation

Figure 1(b) is a typical production with a left graph and a right graph. Applying a production to a given application graph can be called an *L-application* (i.e. replacing a sub-graph in the application graph that matches the left graph of the production by the right graph) or *R-application* (i.e. replacing a sub-graph in the application graph that matches the right graph of the production by the left graph). A visual language, defined by a graph grammar, can be derived by using *L-applications* from an initial graph, usually represented by a special symbol λ. On the other hand, *R-applications* are used to verify the membership of a graph, i.e. grammar parsing. If a given graph is eventually transformed into an initial graph, the parsing

process is successful and the graph is considered to represent the type of design with the structural properties specified by the graph grammar.

Due to the multi-dimensional nature of graphs, mechanisms are needed to address the embedding issue in subgraph replacements, i.e. establishing relationships between the surrounding of the replaced subgraph and its replacing subgraph in the given graph. The SGG addresses the embedding issue by a marking technique [9]. In a production, a vertex is marked by prefixing its label with a unique integer as shown in Figure 1(b). The SGG parser has a polynomial computational complexity [9].

2.2. Grammar Induction

Grammar induction, also known as *grammatical inference*, is a particular instance of inductive learning which can be formulated as the task of iteratively discovering common structures in examples [4]. In this case, a set of examples, also called *positive* samples, is usually a set of strings defined on a specific alphabet. A *negative* sample is a set of strings not belonging to the target language [4]. Informally, grammar induction is defined as [3]:

Given some sample sentences in a language, and perhaps some sentences specifically not in that language, infer a grammar that represents the language.

The development of grammar induction gain lots of algorithmic supports from machine learning techniques. Induction algorithm iteratively finds common substructures from a given set of data, and organizes the hidden hierarchical substructures in a grammatical way. When a common frequent substructure is found, a grammar production will be created. This newly created rewriting rule consists of two parts: a *left hand side* (LHS) and a *right hand side* (RHS). The substructure consisting of terminal symbols identified from the given data samples is represented as the right hand side of the production, and new non-terminal symbols will be created as the left hand side. Then the newly-created production will be applied to current dataset, i.e. a match of the *RHS* will be replaced by the *LHS*. The procedure of *pattern mining – production creation – substructure replacement* will be recursively performed on the original dataset until there are only non-terminal symbols, or a threshold, i.e. a stop criterion defined by the user, is reached. Different from conventional grammar inductions that primarily work on textual information like strings, graph grammar induction works on graphs, and produce diagrammatic rewriting rules (i.e. productions, where aforementioned LHS is the left graph and RHS is the right graph). In a graph grammar, a graph G can be denoted as a tuple $<N, E>$ where N is the set of nodes and $E \subset N \times N$ is the set of edges in the graph. A production rule r is in the format $S := P_1 \mid P_2 \ldots$. Both S and P_i are graphs. Graph grammar induction benefits from the graphical properties of some standard representations of program behaviors, e.g. call graphs. Figure 2 shows a call graph and its two inferred productions.

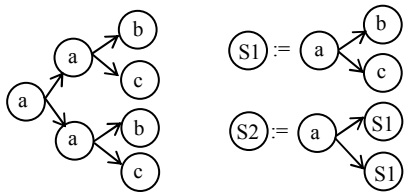


Figure 2 A graph with an inferred grammar

3. Approach Overview

Figure 3 is the overview of our approach. The overall process includes four steps: *trace collection*, *trace preprocessing*, *grammar induction*, and *grammar parsing*.

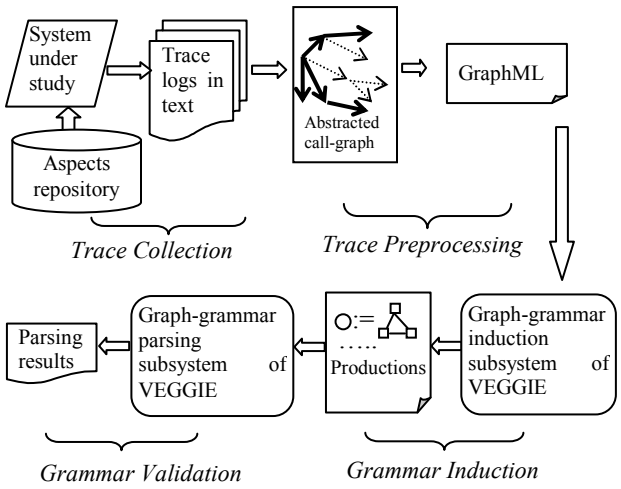


Figure 3 Overview of our approach

An aspect-oriented approach is used to collect program traces as it has less perturbation to the program under study than those putting extra tracing codes into the original program. We first build an aspect repository. Tracing aspects are then compiled together with the source code to generate execution traces, and saved in text files. After that, the traces are reconstructed into a call-graph represented in a linked-list. Objects and method invocations are encoded as attributes of the nodes and edges in the call-graph. To support scalability, a filter is used to preprocess the raw data by removing loops and pruning low-level branches in the call-graph, and produce an abstracted call-graph. The abstraction is tunable by users with adjusting parameters. To be compatible with the data format used in the induction system, the abstracted call graph is then converted to a GraphML format.

GraphML is an extension of XML, and is specialized in describing the structural properties of graphs. Strictly complying with GraphML specifications, the graph dataset is used as an input to the graph-grammar induction subsystem of VEGGIE. The induction subsystem implements a common substructure mining algorithm. In addition to the isomorphism for common substructures, we augment nodes in substructures with temporal attributes. Therefore, a set of graph-grammar rules could be inferred. To validate the grammar, the SGG in VEGGIE is used to parse the given traces represented in graphs based on the inferred grammars. A valid parsing result indicates the syntactic correctness of the inferred grammar against its corresponding program executions.

4. Methodology

This section explains how to apply the graph-grammar induction approach to discovering the structures of program behaviors using a running example.

4.1 Program Preprocessing

Data Acquisition: To define an instrumentation aspect using AspectJ, we declare (1) join points (i.e. the specific points in the execution of the program), (2) pointcuts (i.e. the collection of join points), and (3) advice (i.e. the piece of code that is executed when a pointcut is reached). The following information is recorded for each method invocation:

- The names of classes, objects, methods, and threads; arguments.
- The enter-exits of static and non-static methods.

Call graphs can be derived from the nested relationships of the enter-exits of method invocations.

Data Representation: Intuitively, data traced from a running program records the actual behaviors of objects in the program. Hence the caller-callee relationship among objects in terms of call graph can be used to illustrate program scenarios. Initially the call graph is saved in a linked list. Each caller maintains a pointer to each of its callees. For instance, Figure 4 is a call graph of a toy program.

To be compatible with the input data format in the VEGGIE system, the call graph is converted into the GraphML format. The information of objects and method invocation corresponds to the GraphML syntax, such as elements, attributes, nodes, and edges, etc. Essentially, a node representing an object in method invocation in GraphML has attributes on threads, objects and classes. Similar to the edges in call graphs, edges in GraphML connect two method invocations. Each edge is directed and explicitly connecting a starting node and an ending node.

Data Abstraction: In traditional analysis of software execution, developers would notice that there exists redundancy in execution traces, but they may be unaware of the impact of redundancy manifested themselves as noise in the mining process. To facilitate the induction process, we need to create a concise representation of the program by pruning unrelated information that does not contribute to the structural features. We use an abstraction mechanism for method invocations captured in program execution. It includes two abstraction criteria:

- continuous repetitions;
- low-level methods.

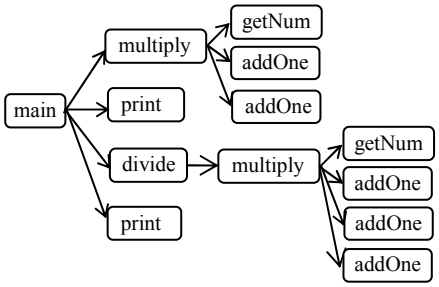
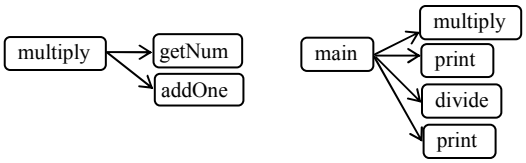


Figure 4 A call graph of a toy program

The first type is to reduce the possible redundant traces. The second one prunes unrelated sub-branches by hiding low-level details. For instance, *addOne* and *getNum* can be considered as details with respect to *multiply*. It allows developers to decide whether to reduce them or not.



(a) Reducing loops (b) Reducing low-level branches

Figure 5 Loop abstractions on the call graph

Eliminating the redundant or fine-grained structures can help the later induction to focus on the high-level behaviors. It can also avoid mining the behaviors of local components. Figure 5(a) shows an abstracted graph after removing the repetition on method *addOne*. Similarly, using the second abstraction criterion, methods with depths in a call chain greater than a user-specified threshold could be pruned. Figure 5(b) shows an abstracted call-graph after being pruned off the third-level branches. Users can choose not to prune sub-branches, so that the substructure *multiply-getNum-add* in this example could be a recurring structure.

4.2 The Grammar Induction System

Inferring meaningful grammars from graphs imposes great challenges. Several issues need to be addressed when inferring a graph-grammar, such as the selection and replacement of subgraphs. To address these problems, graph-grammar induction uses graph-based substructure mining algorithms instead of sequential based mining techniques. A *substructure* is defined as a representation of the recurring subgraphs. An *instance* is defined as one instance of such substructure in the graph dataset [2].

The substructures produced from grammar induction procedure reveal hidden recurrent patterns within the graph dataset. The hierarchical relations within the grammar can aid developers in understanding and analyzing the construction of large and complex legacy systems. Those grammars can also be used to create size-constrained graphs to simulate the growth of a system. Furthermore, researchers can compare the inferred grammar against predefined grammar rules, if exist, for the system to verify the designs.

The variety of substructure mining algorithms [10] results in several graph grammar induction systems [6][14][5]. For instance, Li *et al.* [10] used frequent subgraph mining to find substructures. Instead of using frequency, the VEGGIE's subsystem SubdueGL uses a compression-based frequent pattern discovery algorithm to identify substructures, and compresses the substructures having the highest compression ratio [6].

SubdueGL emphasizes on the compressing of graph datasets instead of purely searching for the frequent subgraphs. The compression value for each substructure is calculated based on a minimum description length (MDL) and the substructure with the highest compression value among the competing substructures are selected. The substructure found in each iteration may not be the most frequent substructure but it can produce the best compression ratio for the given graph, i.e. the ratio between the original and resulting graphs after the inferred subgraphs are compressed. By iteratively discovering substructures with the largest compression ratio, SubdueDL replaces subgraphs in the given graph. The iterations ultimately turn the given graph into one or more non-terminal nodes, or no qualified substructures exist. Users can also set the threshold for the number of interactions. The steps of compression constitute a structural hierarchical lattice that corresponds to a set of graph grammar productions. Details of SubdueGL algorithm can be found in Jonyer [6]. The most frequent substructure does not necessarily compress the graph best. For instance, the ratio of compressing substructures of size one with a frequency of five is less than compressing substructures of size ten with a frequency of two. Hence frequency is not always the best factor in graph compression. The beauty of this grammar induction system lies in that it has the most powerful compression capability, and needs the least amount of iterations to reduce a graph to the minimum.

4.3 Patterns with Temporal Properties

Behavioral patterns describe activities that happen in an order. To reflect this, we adapted the SubdueGL algorithm by augmenting patterns with temporal attributes. Without temporal ordering, the inferred common patterns may not be correct even if they are isomorphic.

We attach logical timestamps to a sequence of events to keep track of the events order. For instance, a sequence that *action A happened 5 seconds before action B* is considered the same as that a sequence *action A happened 2 seconds before B*.

Each substructure G is represented as a tuple $<N, E>$ where N is a set of nodes, and E is a set of edges connecting nodes in the substructure (i.e. subgraph). Each node in the subgraph has one additional attribute: timestamp represented by t_{ni} for node n_i. The timestamp is generated when the node is produced. In the GraphML representation, each node in the graph will have an integer timestamp. A node vector v_G represents an ordered sequence of nodes within the substructure G, i.e. $v_t = \{n_1, n_2, n_3, \ldots, n_n\}$ where $t_{ni} < t_{nj}$. That means n_i happens before n_j. Two substructures $G_1 = \{N_1, E_1\}$ and $G_2 = \{N_2, E_2\}$ are common structures if and only if they are isomorphic and have the same node vector.

The time attribute is considered when grammar induction performs subgraph matching. For instance, the two graphs in Figure 6 where integer figures represent logical timestamps are not common structures since their node vectors (*Multiply, getNum, addOne*) and (*Multiple, addOne, getNum*) are not equivalent.

Figure 6 Two graphs with temporal attributes

4.4 Grammar Validation

Grammar validation is a process of checking the syntactic correctness of the grammar by parsing examples in the language. We employ the SGG to parse the given traces represented in graphs based on the inferred grammars. A valid parsing result ensures the syntactic correctness of the inferred grammar against its corresponding program execution.

The SGG initially developed independently for visual languages and spatial parsing reasoning [9], has been integrated in VEGGIE. The parsing system shares the

same visual interface with the induction system, but works independently. Therefore, SGG can be used to check the correctness of the inferred grammars. Moreover, the context-sensitivity makes the SGG powerful enough to parse any context-free grammars inferred by the induction process.

The SGG can also be used to check the structure of other programs. For instance, if developers want to check if a new program satisfies the constraints specified by the inferred grammar, they can use the inferred productions to parse the new program. A valid parsing result means that the program satisfies these properties.

4.5 A Supporting Environment

A data preprocessor called *Abstracer* is built for data collection and preprocessing, including tracing the system under study, producing logs, reducing redundant and noise traces with tunable parameters, and generating GraphML files. Following the terminology of GraphML specification, we represent the caller-callee relationships in the form of schemas where objects and method invocations are denoted as nodes and edges, respectively. There are only starting and ending points for each edge without any other attributes. Therefore the method names are included in the corresponding nodes' attributes. Each node has a unique integer id, a type for its name, and a position in the graph editor.

For instance, a GraphML example for the structure "main→ multiply" is shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- SGG Graph Data -->
<graphml xmlns = http://graphml.graphdrawing.org/xmlns >
    <graph edgedefault = "directed" xmlns =
    http://viscomp.utdallas.edu/VEGGIE >
        <node id="1" type="main" pos="995 945">
            <port id="{main}"/>
            <data key="attrib">
                <attrib id="Terminal" type="2" bool="true"/>
            </data>
        </node>
        <node id="2" type="multiply" pos="878 252">
            <port id="{multiply}"/>
            <data key="attrib">
                <attrib id="Terminal" type="2" bool="true"/>
            </data>
        </node>
        <edgetype="E" directed="false" source="1" target="2"
        sourceport="{main}" targetport="{multiply}"/>
    </graph>
</graphml>
```

The VEGGIE grammar system has a user-friendly interface which includes three parts as shown in Figures 7, 8 and 9: the type editor, the graph editor, and the grammar editor. These editors are closely related and seamlessly working together.

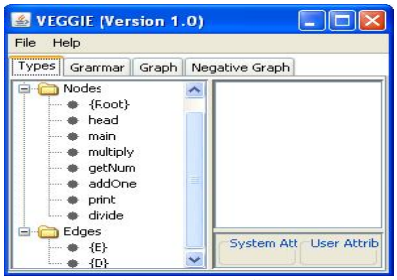


Figure 7 Type editor

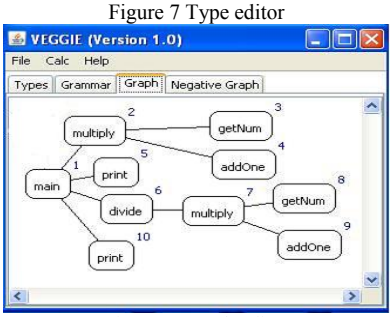


Figure 8 Graph editor

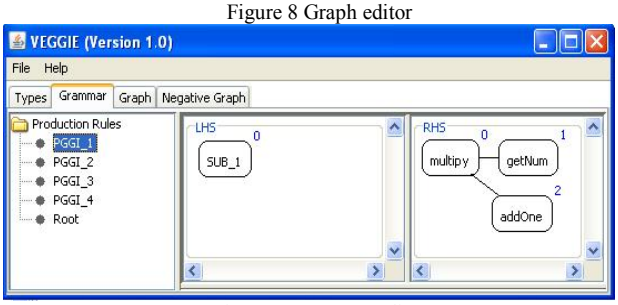


Figure 9 Grammar editor

The type editor as show in Figure 7 lists properties such as the types, attributes and ports for all the nodes and edges in the given input graph. The left panel includes information about nodes and edges. The head node is used as a root without any special meaning. The graph editor can import and display graphs generated by Abstracer in the GraphML format. Figure 8 shows the graphical representation of the program in the display panel. In this directed graph, each edges directs from a node with a smaller integer to a node with a larger integer. The integers associated with nodes specify the temporal order. The *Calc* button on the interface provides two actions for end-users to perform either grammar induction or grammar parsing on the given graph.

If the user issues an induction command, the graph grammar will be displayed in the grammar editor. As shown in Figure 9, the leftmost panel lists the production rules inferred from the toy program. By clicking on one of the productions, the corresponding details will be displayed in the middle and right panels, representing the left graph and right graph of the inferred production. Figure 9 shows the first graph production inferred by VEGGIE, and the right graph of which is the first compressed substructure. By analyzing these productions,

developers can get a hierarchical structure of the program behavior. Besides displaying the productions, the inferred grammar rules can be exported, and saved in the GraphML format.

VEGGIE not only assists grammar induction from graphs, but also supports parsing existing grammars using the SGG parser as described earlier. One can verify the syntactic correctness of any given graph by parsing the graph using the inferred grammar. To realize it, developers can use the *parse* command of *Calc* button in the graph editor in Figure 8. Then VEGGIE will popup a window and report the parsing result: valid or invalid.

The visual environment increases the expressiveness of visual languages with a friendly and easy-to-use interface. The technical details on trace processing, graph grammar induction and parsing are hidden from users.

5. Case Study

5.1 Experiments Design

To evaluate our approach in a real-world application, we experimented on an open-source project JHotDraw. JHotDraw is a GUI framework with structured drawing editors written in Java and was initially designed to illustrate the application of design patterns. We used Version 6.0 Beta that contains 136 classes, 1,380 methods, and 19 interfaces. The source codes have been used in many previous evaluations.

JHotDraw supports many drawing activities. Commonly used activities include: *Initiate the JHotDraw drawing environment; Create new display view; Draw graphs such as rectangle and triangle; Start and end animation; Close JhotDraw,* etc. Using AspectJ, we defined the instrumentation aspect by specify the pointjoint as follows: *execution (* *. * (..)) && ! within (org. lib. instrumentation) && within (org. jhotdraw. samples. * . *).* We designed four scenarios in the experiment.

Scenario 1: *draw a rectangle*. No abstraction was made on the raw trace. This intends to evaluate the grammar induction ability for identifying structures without abstractions.

Scenario 2: *draw one triangle four time*s. We apply the first criterion in abstraction process, i.e. continuous redundant traces were abstracted away. We intend to evaluate if the induction can identify the repeating behaviors "*drawing*" as productions.

Scenario 3: *draw one triangle four times*. We apply two criteria of abstraction process, i.e. both loops and method invocations with call depths larger than three in the call chain were removed automatically. We intend to compare with Scenario 2, and evaluate the influence of the abstraction on induction. We used the same raw trace as in Scenario 2.

Scenario 4: *draw a triangle and an eclipse, and start and end animation twice*. Continuous redundant traces were abstracted away. This intends to evaluate the patterns inferred from various activities.

5.2 Discussions

Preliminary results are shown in Table 1. We evaluate the related metrics of the approach, such as the size of trace, the number of reductions, and execution time.

Table 1 Preliminary results of four scenarios

Scenario	Lines of trace	Lines of abstracted trace	# of events	# of productions	Exec. time (sec)
1	200	n/a	99	25	86.996
2	348	100	50	12	0.911
3	348	90	50	10	0.521
4	1774	208	90	7	64.092

Based on the information in Table 1, we notice that reduction on loops and pruned traces can substantially increase the efficiency of induction. Compared with Scenario 1, Scenario 4 has much larger traces; its execution time, however, is less than Scenario 1 due to the abstraction. Similarly, Scenario 3 spent less time than Scenario 2 because its lower-level branches were pruned. We also notice that the number of inferred productions has no direct correspondence to the number of events in the system. Scenarios 1 and 4 provide the evidence. Moreover, the abstraction ratio is the same for all the scenarios. It may depend on the topology of the trace structure.

Since the induction algorithm is based on a compression-based subgraph mining, a substructure found during each round of iterations may not have a concrete meaning. Thus the grammar may not be necessarily the best in representing semantically relevant program events, since a grammar describes the syntax, rather than the semantics, of the given graph.

6. Related Work

Related work includes software pattern mining, dynamic program behavior discovery, and application of grammar induction in software engineering.

Cook *et al.* [3] discovered formal models of software behavior from event-based data using grammar inference. They evaluated the strengths and weakness of Ktail, Markov, and neural-network-based discovery methods. They used textual information in these methods instead of

graphs. A most recent and related work by Walkinshaw *et al.*[13] applied the QSM algorithms of Dupont *et al.* [4] to reverse engineer finite state machine of program behaviors from execution traces by interactive grammar inference. They mapped methods in traces to six predefined functions to reduce the traces. This means that there are only six symbols in that language. The QSM algorithm was used to select and merge the symbols, and generate a state machine. Our approach abstracts original methods, and mines behavioral patterns represented in graphs. There may be graphical symbols denoting method invocations specified by the inferred grammars.

Sartipi *et al.* [12] combined sequential pattern mining and concept analysis to recovery software structures from loop-free execution traces. Patterns were mined and then used to build a concept lattice. In our work, common patterns are subgraphs representing the method invocations between objects, while the sequential patterns cannot represent and display objects' interactions directly. Furthermore, we built hierarchical lattice naturally during the construction of a grammar, which is more efficient. Our lattice can express the construction of program behaviors for one scenario while their work can help to identify the distribution of functions in the lattice for the same scenario.

7. Conclusion and Future Work

This paper has presented a graph-grammar induction approach to the discovery of program's actual behaviors using a semi-automatic visual environment. We investigated the graph representation of program behaviors, and applied well-established graph-grammar formalisms. Inferred graph-grammars can be used to understand the hidden structures of program behavior. They also provide clues to the construction of complex or legacy systems. The common substructures found through induction are possible reusable software components. Based on our preliminary study, we believe that this approach could be extended to model formal state machines using the inferred graph grammars.

As the future work, we will conduct more experiments on real-world systems and investigate issues like scalability and efficiency. Software semantic constraints can be included in the subgraph mining algorithm, while currently we only include the temporal constraint. Systematic evaluations are planed as well. Empirical evaluation will be performed to experts and novices to test the usefulness of the inferred grammars.

8. References

[1] K. Ates, J.P. Kukluk, L.B. Holder, D.J. Cook, K. Zhang, "Graph Grammar Induction on Structural Data for Visual Programming", In *Proc. IEEE International Conference on Tools with Artificial Intelligence*, 2006, pp. 232-242.

[2] K. Ates and K. Zhang, "Constructing VEGGIE: Machine Learning for Context-Sensitive Graph Grammars", In *Proc. IEEE International Conference on Tools with Artificial Intelligence*, 2007, pp. 456-463.

[3] J.E. Cook and A.L. Wolf, "Discovering Models of Software Process from Event-Based Data", *ACM Transactions on Software Engineering and Methodology*, Vol. 7, Issue 3, 1996, pp. 215-249.

[4] P. Dupont, B. Lambeau, C. Damas, and A. V.Lamsweerde, "The QSM Algorithm and its Application to Software Behavior Model Induction", *Applied Artificial Intelligence*, Vol. 22, Issue 1 & 2, 2008, pp. 77-115.

[5] R. Jin, C. Wang, D. Polshakov, S. Parthasarathy, and G. Agrawal, "Discovering Frequent Topological Structures from Graph Datasets", In *Proc. 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, 2005, pp. 606-611.

[6] I. Jonyer, "Context-Free Graph Grammar Induction Based on the Minimum Description Length Principle", *Ph.D. Dissertation*, The University of Texas at Arlington, 2003.

[7] J. Kukluk, L. Holder, and D. Cook, "Inference of Node Replacement Recursive Graph Grammar", In *Proc. Sixth SIAM International Conference on Data Mining*, 2006, pp. 544-548.

[8] J. Kong, "Visual Programming Languages and Applications", *Ph.D. Dissertation*, The University of Texas at Dallas, 2006.

[9] J. Kong, K. Zhang, and X. Q. Zeng, "Spatial Graph Grammars for Graphical User Interfaces", *ACM Transactions on Computer-Human Interaction*, Vol.13, No.2, 2006, pp. 268-307.

[10] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System", In *Proc. 6th Symposium on Operating System Design and Implementation*, 2004, pp. 289-302.

[11] H. Safyallah and K. Sartipi, "Dynamic Analysis of Software Systems using Execution Pattern Mining", In *Proc. 14th IEEE International Conference on Program Comprehension*, 2006, pp. 84-88.

[12] K. Sartipi and H. Safyallah, "Application of Execution Pattern Mining and Concept Lattice Analysis on Software Structure Evaluation", In *Proc. International Conference on Software Engineering and Knowledge Engineering*, 2006, pp. 302-308.

[13] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. SalaHuddin, "Reverse Engineering State Machines by Interactive Grammar Inference", In *Proc. 14th Working Conference on Reverse Engineering*, 2007, pp. 209-218.

[14] X. Yan and J. Han, "gSpan: Graph-Based Substructure Pattern Mining", In *Proc. International Conference on Data Mining*, 2002, pp. 721.