A Grammar-Based Reverse Engineering Framework for Behavior Verification

Chunying Zhao Kang Zhang
The University of Texas at Dallas
{cxz051000, kzhang}@utdallas.edu

Abstract

A high assurance system requires both functional and nonfunctional correctness before the system is put into operation. To examine whether a system's actual performance complies with the requirement, an effective reasoning and verification mechanism is needed. This paper presents a graph grammar based reverse engineering framework for the behavior verification of high assurance systems. It casts the program verification problem to a visual language parsing problem, i.e. parsing the graphical representation of program behavior with user-specified constraints and properties expressed as a graph grammar. The approach allows developers to check the acceptable sequence of method calls corresponding to the specifications for some requirements.

1. Introduction

With the rapid development of information technology, software has been increasingly larger and more complex. Analyzing the correctness of software becomes challenging. Researchers have successfully developed many formal methods and verification techniques in order to eliminate errors as early in the development cycle as possible. Model checking [5] is a commonly used automatic verification approach to checking the requirement correctness and logical consistency of a design. Given a sound and detailed system design, however, it is possible that the actual program execution does not faithfully fulfill the design representing system requirements due to the misunderstanding of design documentations or miscommunication between designers and programmers.

To address the problem, execution information needs to be analyzed and verified with respect to its design, i.e. checking the behaviors of the program and verify if the implementation fulfills the specification as expected. There have been many instrumentation and analyzing techniques for reverse engineering program behaviors focusing on retrieving high-level scenarios, e.g. a UML sequence diagram, from execution traces. Instead of recovering scenario models from the runtime behavior, we aim at verifying rule-based specifications or constraints against program execution. Compared with text-based approaches, visual language techniques take advantage of the graphical representation of program behaviors, since graphs have been extensively used for program representations, such as UML diagrams, flowcharts and call graphs, etc. Moreover, it is more expressive to visually specify program properties as a graph grammar and parse the given graph.

This paper presents a semi-automatic graph grammar based reverse engineering framework. It allows developers to specify constraints or properties as a context-sensitive graph grammar. Then the program behavior represented as a graph is automatically parsed by the specified grammar. A parsing result indicates whether the investigated behavior satisfies the requirements or not. In this way, the behavior verification problem is cast to the problem of graph parsing. The framework is supported by a visual language environment called VEGGIE [1], an integrated graph grammar induction and parsing system.

2. Overview of the Framework

Fig.1 depicts an overview of the framework, which includes two stages.

- Trace collection: an execution trace is collected using AspectJ; a scenario represented as a call graph is constructed and abstracted from the trace.
- Behavior verification: the automatic verification system takes the abstracted scenario and the user' specification as inputs, and generates a verification result.

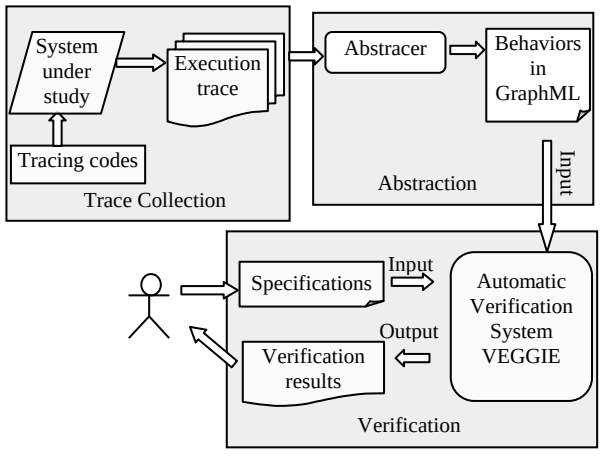


Fig.1 Overview of the framework

3. Program Behavior Verification

To obtain the trace, we have implemented an aspect-oriented instrumentation approach, which is less intrusive than those putting extra tracing codes into the original system. Without losing the necessary details, the following information is recorded in a log file: object/class identifiers, method/thread names, parameters of method invocations.

Scenarios can be derived from an execution trace based on the *causal* relationship. Causality characterizes the interactions between events, and could be derived based on the enter-exits of method invocations, i.e. the happening ordering of methods. Before the verification of behaviors, a preliminary information-lossless abstraction on a scenario is necessary to improve the parsing performance. To achieve this, a user can choose to abstract two types of information: (1) loops; (2) intra-object interactions. Loops are the major source of information redundancy, which can be represented by one instance and the number of repetitions. An intra-object method invocation shows the details of local interactions within the same object. It could be collapsed if the user only observes the activities between different objects, and also could be expanded to the original trace if needed. The lossless abstraction ensures the abstracted scenarios to represent the system behaviors in a reversible way, and allows users to focus on the activities they are interested in.

We can perform two types of behavior verification: (1) verifying the acceptable call sequences in the scenario; (2) detecting illegal behaviors or security related activities. Suppose in a application, object *A* does not have the authority to access methods on object *C*, thus object *A* can only indirectly invoke object *C* via some method in object *B*. Fig.2 describes such a scenario, in which the solid lines depict the correct scenario while the dotted line illustrates an illegal scenario. Both the correct and the illegal behaviors can be verified using predefined constraints. Likewise, other types of behaviors such as a missing connection in a causal link and a cycled causal link can also be identified.

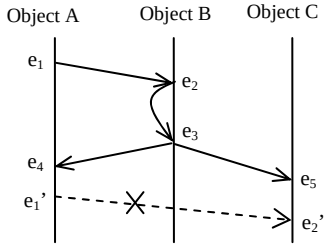


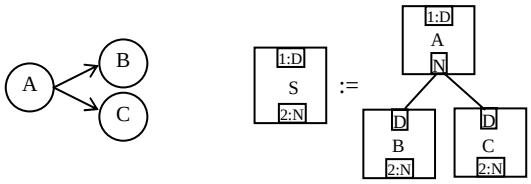
Fig.2 An example scenario

For each scenario we build a call graph represented in GraphML format as the input to the VEGGIE's parsing system.

4. Graph Grammar Parsing

We encode program specifications using a context-sensitive grammar formalism, the Spatial Graph Grammar (SGG) [7]. The program verification process is supported by the SGG parsing subsystem of VEGGIE. The visual editors of VEGGIE allow developers to specify the syntax and semantics of the behavioral properties using graphical elements. The parsing subsystem can parse the given graph representing the program behavior, and generate a parse tree for a valid parsing.

The rectangle labeled "*A*" in Fig.3 (b) is a typical node in SGG, which has two embedded vertices *D* and *N*. Following this format, developers can draw both terminal and non-terminal symbols. Attributes such as name and type can be annotated in a node. In general, nodes can represent modules of any granularity in a program. In this paper a node denotes an object's method, and an edge denotes a method invocation. Each grammar consists of a set of grammar rules called *productions*. A production has a *left graph* and a *right graph*. The context-sensitivity allows the left graph of each production to have more than one node. We use productions to represent the sequence of method calls corresponding to a behavioral specification. For instance, Fig.3 (b) is a graph grammar production. Its left graph is a new non-terminal node, and its right graph represents the call graph in Fig.3 (a). The terminal nodes represent the invoked methods in the given program.



(a) Method Invocations (b) A Production

Fig.3 Production representation

The VEGGIE grammar editor (illustrated in Fig.7 (a)) provides a visual interface for the user to define grammars. A graph grammar is used to specify all the acceptable method invocations in the given program behavior. Prohibited method invocations can also be specified with negative productions. Each production is associated with the predefined semantics using *action code* that is a piece of Java code executed when the right graph of the production is applied. A parsing process can be considered as a *subgraph matching and replacement*. When applied to behavior verification, the parsing process can check both the syntax and semantics of the given program. The integer annotated within a vertex servers as a marker to preserve the context, i.e. the connections with the surrounding elements in the parsed graph during the subgraph replacement. An advantage of using context-sensitive grammars over finite state machines is their

capability of specifying nested call structures such as recursion [6].

Formally, the context-sensitive grammar representing program behavioral properties is defined as a tuple $G = <T, N, E, P>$:

- P is a finite set of productions specifying the behavior properties, e.g. the acceptable sequence of method invocations satisfying a certain constraint.
- T is a finite set of terminal nodes in P, representing the events occurring in the scenario.
- N is a finite set of non-terminal nodes in P.
- E is a finite set of edges in P, connecting the caller and callee events.

The architecture of the verification process via graph grammar parsing is shown in Fig.4. The runtime behavior is represented as a call graph to be parsed in the grammar system. The specifications and properties for the program behavior are represented as productions and semantic actions.

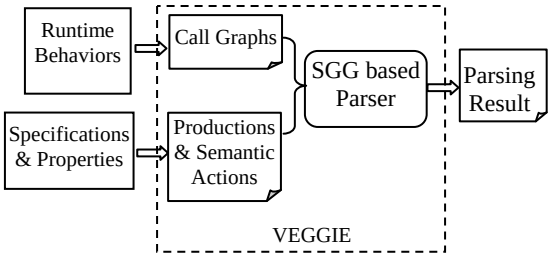
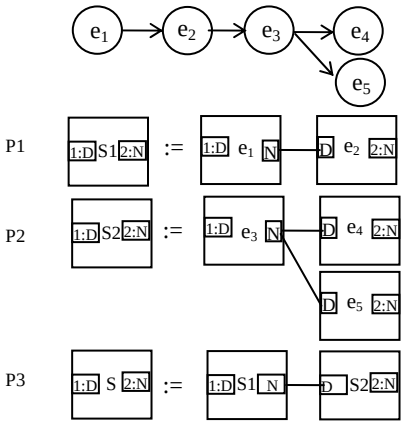


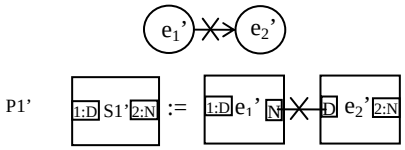
Fig.4 Architecture of the grammar system

Users can define productions in the grammar editor following the node-edge format of SGG. To define a production, users do not have to draw each node. Instead, they can select nodes from the node list previously stored in VEGGIE's type editor, and then define the left graph and the right graph by connecting the selected nodes via ports. The nodes in the node list can either be drawn in the type editor or imported directly from a GraphML data file listing all the terminal nodes. Similarly, the call graph representing the program behavior is also stored in a GraphML file, and can be imported into VEGGIE's graph editor (illustrated in Fig.7 (b)). Both of the GraphML files are generated during the scenario extraction and abstraction.

Consider Fig.2 which includes a legal scenario depicted in solid lines and an illegal scenario described in dotted lines. The legal scenario serves as the acceptable calling sequences we intend to verify, and the illegal scenario is the call sequence not allowed in the program behavior. Fig.5 (a) shows the corresponding productions (P1, P2 and P3) for the legal behavior. Users can use this set of productions as a specification to automatically parse the given program's call graph. Fig.5 (b) shows the corresponding productions (P1') for the illegal behavior. A valid parsing result for such productions indicates that the program violates certain constraints, i.e. illegal/prohibited calls in the example.



(a) A legal call sequence with corresponding productions



(b) An illegal call with a negative production

Fig.5 Example productions

5. Experiments

We experimented with JHotDraw (Version 6.0 Beta, available at http://www.jhotdraw.org/). There are several specifications that JHotDraw shall meet. Although we do not have the original design specifications for JHotDraw, we can derive some general rules about its behaviors. JHotDraw allows users to draw different shapes and move them in a drawing view, and then to save the view into a user-selected directory. One of these rules is that a "new" action must precede the "save as" action. This rule holds naturally since a drawing view cannot be saved before it is created. We identified such an error actually existing in Version 6.0 Beta. The verification process is as follows:

- Instrumented JHotDraw using AspectJ with the following pointcuts, and then executed it.
  *before(): execution(* * . *(..)) && !within (org. lib. drawApplication. *)*
  *after():execution(* * . * (..)) && !within(org. lib. drawApplication. *)*
- Constructed an abstracted call graph from the execution trace.
- Imported the call graph to VEGGIE and automatically parsed it using the graph grammar representing the program specification.

We translated the specifications (a *new* event should precede a *save* event) into a graph grammar. That means that we defined a graph grammar based on SGG notations describing the calling sequence for creating (by "new")

and saving (by "save") a view. The specification is illustrated in the dotted rectangles in Fig.6, which includes the expected call sequence for the two events. It can be customized by pruning branch nodes deeper than a certain threshold in the call tree to increase the verification efficiency.

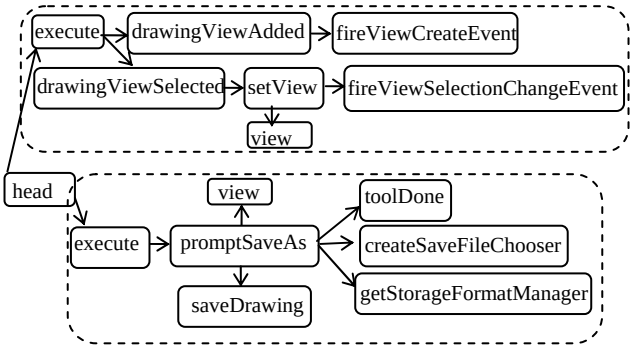
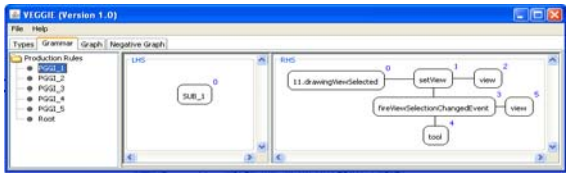
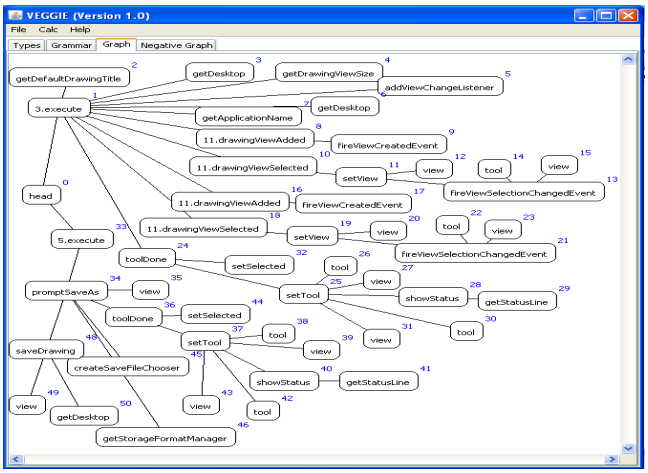


Fig.6 Expected calling sequences as the specification

The VEGGIE system essentially consists of two parts: visual editors and a parser. The visual editors include a type editor, a grammar editor and a graph editor. Fig.7 (a) is the grammar editor displaying the corresponding graph grammar of the specification in Fig.6. We executed a scenario by creating a new view and saving it. Fig.7 (b) shows the graph editor that imported the abstracted graph constructed from the execution trace of the two events. We successfully parsed the graph with the specified graph grammar, which means that the call sequence represented by the graph is correct.



(a) Productions



(b) An Abstract Graph

Fig.7 An abstracted call graph and its productions

6. Related Work

Use of natural language techniques for program verification has been studied in the literature. Cook *et al.* [2] and Dupont *et al.* [4] discovered program behaviors by inferring finite state machines. Costagliola *et al.* [3] applied visual language parsing techniques to the discovery of design patterns from source codes. Based on the fact that finite state machines essentially cannot specify nested call structures such as recursion, Hughes *et al.* [6] proposed an interface grammar for modular software model checking, which allows developers to specify nested call sequences using grammars. Different from this work [6], we define grammars based on actual method invocations, and parse a program's actual execution. In summary, to our knowledge, there have been no graph parsing approaches supported by a visual environment for the verification of program behaviors.

7. Conclusion and Future Work

This paper has presented a grammar-based reverse engineering framework for verifying program behavioral properties using visual languages and parsing techniques. The acceptable or prohibited unsafe call sequences are represented by a graph grammar, which is used to parse and verify the given program behaviors represented in call graphs. Our future work will focus on combining the developer's domain knowledge with the graph grammar syntax so that productions can be automatically induced by the induction subsystem of VEGGIE.

8. Reference

[1] K. Ates and K. Zhang, "Constructing VEGGIE: Machine Learning for Context-Sensitive Graph Grammars", In *ICTAI*, pp. 456-463, 2007.

[2] J.E. Cook and A.L. Wolf, "Discovering Models of Software Process from Event-Based Data", *ACM TOSEM*, Vol. 7(3), pp. 215-249, 1996.

[3] G. Costagliola, A.D. Lucia, V. Deufemia, C. Gravino, and M. Risi, "Design Pattern Recovery by Visual Language Parsing", In *ECSMR*, pp. 102-111, 2005.

[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(1&2), pp. 77-115, 2008.

[5] G. Holzmann, "The Spin model checker", *IEEE TOSE*, 23(5):279–295, 1997.

[6] G. Hughes and T. Bultan, "Interface Grammars for Modular Software Model Checking", In *ISSTA*, pp. 39-49, 2007.

[7] J. Kong, K. Zhang, and X.Q. Zeng, "Spatial Graph Grammars for Graphical User Interfaces", *ACM TOCHI*, 13(2): 268-307, 2006.