

Context-driven Testing of Object-oriented Systems

Amie L. Souter
Department of Computer Science
Drexel University
Philadelphia, PA 19104
souter@cs.drexel.edu

Abstract

Many different testing techniques have been proposed by researchers, but essentially only two main testing philosophies exist, black box and white box. There exists a number of different testing methods for structural testing of procedural languages, however, the features of object-oriented languages are not addressed by such techniques. This dissertation explores a new structural testing technique for object-oriented systems by developing a testing methodology based on object manipulations and driven by the context of the program under test.

1. Introduction

Object-oriented programming languages have emerged into the mainstream programming community. The underlying ideology of an object-oriented approach is to model concepts through the use of objects. Such an approach has been incorporated into software engineering analysis and design methodologies, which help in developing well designed programs. In addition, object-oriented features enable the programmer to develop software that is reusable and modular. Although there are many benefits to an object-oriented approach, testing is still a required aspect of the software engineering life cycle, and is required to ensure the reliability, robustness and usability of software. Unfortunately, testing, and in particular ad hoc testing, is labor and resource intensive, accounting for 50%-60% of the total cost of software development [4]. Therefore, it is imperative that techniques be developed that reduce the cost of testing and improve the quality of software, while also providing as much automation and ease of use as possible.

Many different testing techniques have been proposed by researchers, but essentially only two main testing philosophies exist, *black box* and *white box* testing. While they both have their tradeoffs and should be used in combination, white box, also called *program-based* or *structural testing*,

is the focus of this dissertation. Structural techniques are based on knowledge of the program. While there exists a number of different testing methods for structural testing of procedural languages, they all share the same underlying motivation—analyzing the program code to develop test cases and ensure that coverage of specific structural elements of the program is achieved by a given test suite. *Control flow-based* techniques are motivated by the intuition that covering different control flow paths would exercise a large proportion of program behaviors. *Data flow testing* is based on the premise that testing paths that read and write values stored into the same memory locations tests the behavior of a program in terms of its manipulation of data [5]. Even if these testing techniques can be extended to deal with the features of object-oriented languages, the usefulness of these techniques is unclear in the object-oriented programming domain where object-oriented design principles result in programs with a structure that differs significantly from the procedural programs originally targeted by these methods.

This dissertation explores a *new structural testing technique for object-oriented systems by developing a testing methodology based on object manipulations and driven by the context of the program under test* [6].

2. Overview of Research Problems and Results

Static def-use information has been shown to be useful not only for optimizing and parallelizing compilers, but also for debuggers, editors, program integration, and software maintenance. In procedural languages, a def-use association (or pair) for variable v is an ordered tuple (v, d, u) where d is a statement defining v and u is a statement that is reachable by at least one definition-clear path from d , and u uses v or a memory location bound to v .

In an object-oriented programming paradigm, not only is the computation of def-use associations complicated by polymorphism and inheritance, but object definitions and uses could be interpreted in a number of ways due to *ob-*

ject aggregation. Programmers create object aggregation relationships when they design a class that includes one or more objects of other classes. Because the object-oriented paradigm promotes significant use of aggregation, manipulations of objects can have far reaching effects on the object definitions and uses of other objects through the aggregation has-a relation.

This dissertation describes a novel formulation of individual object definitions and uses, which captures object aggregation relations in addition to addressing inheritance and polymorphism [8]. The resulting object def-use associations are *contextual* in that they provide context with the computed def-use associations in an object-oriented program, in contrast to traditional def-use associations, which are called *context-free* because they are reported free of any context.

This new formulation was inspired by observations made during experimental studies of Java program characteristics [12]. In particular, by extending the points-to escape graph representation developed by Whaley and Rinard [14], an algorithm for generating contextual def-use associations with several strategies has been developed with each strategy providing a different level of context, but all based on the program's object aggregate relationships.

An approach for constructing contextual def-use associations (cdus) is explored as part of this dissertation [7]. In order to construct cdus, a program representation capable of capturing object aggregation relations is necessary. The points-to escape graph developed by Whaley and Rinard [14] represents object relationships by modeling the manipulation of objects. By extending the points-to escape graph representation, contextual def-use associations can be generated for the different context levels.

The extended points-to escape graph representation, which is referred to as the *ape* (**A**nnotated **P**oints-to **E**scape) graph¹, contains adequate information to identify potential manipulations to the same object in the presence of polymorphism, aliasing, inheritance, and of course, aggregation. In addition, this program representation can be used to determine which object manipulations are captured within the software component being tested, which objects may be potentially manipulated from outside the component, and the different ways in which an object is accessible from outside the component. A component is defined as a set of methods that define a region of a program. For example, a set of user-defined methods in a class could represent a component. Therefore, interaction between components can be identified using the *ape* graph. This representation of object manipulations enables a straightforward identification of object creation sites and potential writes and reads to the same object.

¹The *ape* graph apes, or mimics, the object manipulations potentially performed at runtime.

While escape analysis was invented for compiler optimization, it can be exploited for software engineering tasks. Escape analysis provides information that determines whether the lifetime of data exceeds its static scope. In an object-oriented program, an object *O* is said to *escape* a method *m* of the program if the lifetime of *O* may exceed the lifetime of *m* [3]. This dissertation demonstrates how knowledge about the interaction of objects between different regions of an object-oriented program can enable novel testing techniques, as well as provide useful information for program understanding. By utilizing escape analysis information embedded in the points-to escape graph [14], testing can be targeted to arbitrary regions as the units for test coverage, and then integration testing on an object basis can be performed [11].

The formulation of contextual def-use associations has one fundamental problem; infeasible contextual def-use pairs can be generated. However, the infeasibility problem is not unique to *cdu* formulation; in fact, it is a problem that many static analyses encounter [2]. This dissertation explores how the conservative nature of the call graph often contributes to the infeasible results of static analyses, as well as how inherently polymorphic call sites may also cause infeasible results. Infeasible *cdus* in the context of call chains is addressed and a solution based on properties discovered in the call graph, for automatically detecting whether a particular call chain is *type infeasible* under certain conditions is presented [10].

Because object-oriented software is typically characterized by many methods, with a large number of them being quite small, sophisticated program analysis for compiler optimization, software testing, debugging, semantic program browsers and other software development tools perform *interprocedural* analysis. The key data structure associated with interprocedural analysis is the program call graph, which is traversed in order to compute desired program information, typically data flow information. While a sound call graph can be computed quickly, a more precise call graph leads to more useful information for the client software tool. However, there is a substantial cost in exhaustively computing a precise call graph.

As an application changes in composition with components being added, deleted, replaced, and updated during the software maintenance life cycle, the call graph for the application changes, which can impact the validity of the interprocedurally gathered information. In order to update contextual def-use associations based on program changes, by either constructing new *cdus* or determining obsolete *cdus*, the underlying call graph essential to *cdu* construction must also be updated based on the program changes. Therefore, given the expense of exhaustively computing a call graph with sufficient precision and the fact that the construction of contextual def-use associations relies on a pre-

cise call graph, this dissertation investigates developing an incremental call graph construction algorithm with acceptable cost for use with interactive software tools [9]. In addition to avoiding an expensive exhaustive call graph construction, an incremental call graph algorithm also easily indicates the potentially affected regions to begin incremental reanalysis of the gathered data flow information.

The incremental call graph construction problem is nontrivial for object-oriented software for the same reasons that complicate exhaustive call graph construction—dynamically dispatched message sends and the interdependent roles of type information and call graphs. Furthermore, in an object-oriented system, edits other than call site changes can affect the call graph structure because type information is changed, which impacts the information known at polymorphic call sites. This dissertation presents incremental algorithms to update a call graph that has been initially constructed using the Cartesian Product Algorithm developed by Agesen [1]. The exhaustive algorithm constructs a very precise call graph based on the concept of templates. The properties of templates make this approach to call graph construction an ideal candidate for efficient incremental call graph construction with the goal of retaining a highly precise call graph during software maintenance.

Finally, the results related to the effectiveness and cost of testing based on cdus are presented. Specifically, cdus are compared to previous structural techniques using a subsumption hierarchy, a qualitative analysis is presented illustrating the likely faults cdus uncover, a prototype tool is presented in order to show that such a testing environment is practical [13], and finally an evaluation of test cost is presented in terms of space for the program representation.

3. Contributions

The key contributions presented in this dissertation include the following:

- A program-based testing framework for object-oriented programs that provides coverage based on object manipulations, in particular with respect to aggregate object relationships. The framework includes both unit and integration testing based on object manipulations.
- The formulation of the notion of *contextual* object def-use pairs that encompasses object-oriented features of inheritance, polymorphism, and aggregation.
- Contextual def-use associations that are scalable due to multiple levels of context.
- Extensions to the points-to escape graph program representation that allow the construction of contextual def-use pairs.

- An algorithm for constructing contextual def-use pairs with different levels of context, which can be applied independent of the construction of the program representation.
- A demonstration of how to use escape analysis for software engineering tasks.
- Algorithms that use escape analysis for object-based and regional interaction coverage.
- Algorithms that apply escape analysis to regression testing activities specifically for additional coverage requirements, and the selection of obsolete cdus.
- The utilization of an arbitrary region of analysis for testing.
- An algorithm for incremental call graph construction based on the exhaustive CPA call graph algorithm.
- A formulation of the type infeasibility property for object-oriented programs, which helps determine call chains that are potentially infeasible.
- Two algorithms based on different approaches for automatic infeasible call chain detection.
- An implementation and design of a prototype testing tool that incorporates the object-based testing framework developed in this dissertation.
- Proofs that show the subsumption relationship between the cdu coverage criteria and the traditional coverage criteria.
- An empirical investigation of the cost of such a testing technique. The program representation was evaluated in terms of both the space of the graph and the space required for annotations.

4. Future Directions

Many questions were answered through the course of this research, yet many new questions were raised for further exploration.

The program representation utilized in this dissertation has many advantages, yet its space requirements do not scale well for very large programs. The investigation of how to utilize the properties that the points-to escape graph exhibits, without the cost of the space overhead is an important direction of future work. The investigation of a flow-insensitive approach and how the precision affects the results is key to the use of a more scalable program representation. Another avenue is to investigate other possible context levels for cdus. For example, library summaries is one possibility to explore.

The use of escape analysis for software engineering tasks was explored in this dissertation. Further research in the use of escape analysis for regression testing, priority-based testing, and program understanding would be fruitful.

Program understanding research investigates how to help a programmer understand complicated code. The use of escape analysis for such tasks could also be more thoroughly examined. Expanding *TATOO* to include query capabilities in which a programmer could specify a query, and a reply could be given both textually and through the use of code highlighting may prove to be useful to the software maintainer.

The algorithms for incremental call graph analysis were only a first step in developing useful software tools tailored towards interprocedural analyses of programs whose structure or composition frequently changes. More work is needed in developing algorithms that both scale better towards larger codes and are precise. The precision of the call graph is fundamental to precise interprocedural analyses.

Finally, as programming languages and environments continue to advance, testing will continue to prove to be critical. Incorporating testing techniques into the programming language and the programming environment will be a crucial aspect to ensure quality software in the future. Even with test cases being developed in conjunction with code, evaluation techniques that report code coverage are crucial in determining the quality of testing. Developing programming languages that incorporate testing techniques into the language as well as tools that automatically generate test cases is a key future direction for software testing research.

5. Acknowledgements

I would like to acknowledge the members of my Ph.D. committee, namely, Bob Caviness, Error Lloyd, Cindy Norris, and Lori Pollock. I am grateful for all the advise and encouragement my advisor, Lori Pollock, provided during my years at the University of Delaware. Her guidance still proves to be extremely valuable.

References

- [1] O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of the European Conference on Object-Oriented Programming*, 1995.
- [2] R. Bodik, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. In *Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT symposium on Software Engineering (ESEC/FSE 97)*, Sept. 1997.
- [3] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Nov. 1999.
- [4] W. Perry. *Effective Methods for Software Testing*. John Wiley Inc, 1995.
- [5] M. Roper. *Software Testing*. McGraw-Hill, 1994.
- [6] A. L. Souter. *Context-Driven Testing of Object-Oriented Systems*. PhD thesis, University of Delaware, 2002.
- [7] A. L. Souter and L. L. Pollock. OMEN: A strategy for testing object-oriented software. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, Aug. 2000.
- [8] A. L. Souter and L. L. Pollock. Contextual def-use associations for object aggregation. In *Proceedings of the ACM SIGSOFT-SIGPLAN Workshop on Program Analysis for Software Tools and Engineering*, 2001.
- [9] A. L. Souter and L. L. Pollock. Incremental call graph reanalysis for object-oriented software maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance*, 2001.
- [10] A. L. Souter and L. L. Pollock. Characterization and automatic identification of type infeasible call chains. *Information and Software Technology*, October 2002.
- [11] A. L. Souter and L. L. Pollock. Putting escape analysis to work for software testing. In *Proceedings of the IEEE International Conference on Software Maintenance*, 2002.
- [12] A. L. Souter, L. L. Pollock, and D. Hisley. Inter-class def-use analysis with partial class representations. In *Proceedings of the ACM SIGSOFT-SIGPLAN Workshop on Program Analysis For Software Tools and Engineering*, Sept. 1999.
- [13] A. L. Souter, T. M. Wong, S. A. Shindo, and L. L. Pollock. *TATOO: Testing and analysis tool for object-oriented software*. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Apr. 2001.
- [14] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Nov. 1999.