A Brief Survey on Automatic Integration Test Order Generation

Zhengshan Wang\textsuperscript{1,2,3}, Bixin Li\textsuperscript{1,2}, Lulu Wang\textsuperscript{1,2}, Qiao Li\textsuperscript{1,2}

\textsuperscript{1}School of Computer Science and Engineering, Southeast University, Nanjing, China
\textsuperscript{2}Key Lab of Computer Network \& Information Integration (Southeast University), Ministry of Education
\textsuperscript{3}Department of Computer Science and Technology, Chuzhou University, Chuzhou, China

Email: \{zhshwang, bx.li, wanglulu, qiao\}@seu.edu.cn

Abstract

A common problem in object-oriented software integration testing is to determine the order in which classes are integrated and tested. In this paper, we first overview some related work based on their objectives in current literature and then provide some analysis and evaluation.

Index Terms—object-oriented; integration test order; coupling measurement; graph-based algorithm; search-based algorithm

I. Introduction

During object-oriented software integration testing, an important problem is to determine the order in which classes are integrated and tested, which is called class integration test order (CITO) problem. When integrating and testing a class that depends on other classes that have not been developed or tested, some stubs must be developed to simulate these other classes. There are two types of stubs: specific stub and realistic stub [16] (or generic stub [9]). It is very error-prone and costly to construct stubs. All related papers tried to find a (optimal) test order using minimal stubbing cost which is usually modeled as the number of stubs to be constructed or modeled as the overall stub complexity.

Many approaches proposed to solve this problem can be divided into two categories based on their objectives: minimizing the number of stubs [2], [5], [7], [11], [12], [13], [14], [15], [16] and minimizing the overall stub complexity [1], [3], [4], [6], [17]. If the objective is to minimize the number of stubs, there are two key steps which are modeling and breaking cycles. A dependency model which can be created from UML diagrams or source code is used to represent classes and inter-class dependency relationships using nodes and edges, respectively. In current literature, there are two kinds of dependency models which are object relationship diagram (ORD) [12] and test dependency graph (TDG) [16]. If a dependency model has no cycles, a test order can be generated by using a simple reverse topological sorting algorithm based on inter-class dependency relationships, otherwise all existing approaches tried to design an effective and efficient algorithm to break cycles. In current literature, there are two types of typical algorithms for breaking cycles which are graph-based algorithm (GBA) and search-based algorithm (SBA). If the objective is to minimize the overall stub complexity, one extra step must be introduced to measure each stub complexity based on inter-class coupling information before breaking cycles.

The rest of this paper is organized as follows: Section II overviews some existing approaches for minimizing the number of stubs. Section III overviews some existing approaches for minimizing the overall stub complexity. Section IV provides some analysis and evaluation. Section V draws some conclusions.

II. Minimizing the Number of Stubs

A. Minimizing the Number of Specific Stubs

Kung et al. [12] used an ORD as a dependency model which was created from source code and had three types of dependency relationships: inheritance, aggregation and association. They used a GBA to break cycles by selecting and removing association edges. Their algorithm did not use any heuristic information to select association edges.
to break cycles. The time complexity of their algorithm is $O(n^4)$.

Tai et al. [15] used an ORD as a dependency model which was created from UML class diagrams and had three kinds of dependency relationships: inheritance, aggregation and association. They determined major level numbers of classes based on inheritance edges and aggregation edges and determined minor level numbers of classes based on association edges. If the classes with the same major level number contained cycles, they used a GBA to break cycles. Their algorithm preferred to select and remove an association edge $(u,v)$ with the highest weight to break cycles. The weight was defined as the sum of in-degree of $u$ and out-degree of $v$. The time complexity of their algorithm is $O(n^4)$.

Briand et al. [5] used an ORD as a dependency model which was created from UML class diagrams and had five kinds of dependency relationships: inheritance, aggregation, association, composition and usage. They used a GBA to break cycles. Their algorithm preferred to select and remove an association edge $(u,v)$ with the highest weight to break cycles. The weight was defined as the product of in-degree of $u$ and out-degree of $v$. The time complexity of their algorithm is $O(n^4)$.

Malloy et al. [13] and Kraft et al. [11] used an ORD as a dependency model which was created from source code and had six types of dependency edges: inheritance, composition, association, dependency, polymorphism, and ownedElement. They used trial-and-error method to assign different weights to different edges based on their types and then used a GBA to break cycles. Their algorithm preferred to select and remove an edge with the lowest weight to break cycles. The time complexity of their algorithm is $O(n^4)$.

Mao et al. [14] used a weighted ORD (WORD) as a dependency model which was created from UML class diagrams and had three types of dependency relationships: inheritance, aggregation and association. They used a triple (cycle weight, direction factors of edges, association intensity) to represent weight for each association edge. They defined the cycle whose length is two as a cycle pair. They first selected and removed association edges to break all cycle pairs, then used a GBA to break other types of cycles. Their algorithm preferred to select and remove an association edge with the highest cycle weight to break cycles. The time complexity of their algorithm is $O(n^4)$.

Hashim et al. [8] used an ORD as a dependency model which was created from source code and had five types of dependency edges: inheritance, composition, aggregation, association and dependency. They used coupling between objects as a new cost function instead of Malloy et al’s parameterized cost function to assign weights to all types of edges with a deterministic way. They used a GBA to break cycles. Their algorithm preferred to select and remove a node with the lowest weight to break cycles. The time complexity of their algorithm is $O(n^3)$.

Bansal et al. [2] used an ORD as a dependency model which was created from source code and had eight types of dependency edges: inheritance, composition, association, dependency, polymorphism, ownedElement, friend and exception. They used Malloy et al.’s idea [13] to assign weights to all types of edges and used Abdurazik et al.’s method [1] (discussed in Section III) to break cycles.

## B. Minimizing the Number of Generic Stubs

Traon et al. [16] used a TDG as a dependency model which was created from UML diagrams. They used a GBA to break cycles. Their algorithm preferred to select and remove a node with the highest weight to break cycles. The node weight was defined as the sum of the number of incoming back edges and the number of outgoing back edges. The time complexity of their algorithm is $O(n^3)$. The number of generic stubs depended on the starting node of deep first search (DFS).

Hanh et al. [7] used a TDG as a dependency model which was created from UML diagrams. They provided two different types of strategies to break cycles. The first called Triskell used a GBA to break cycles. They preferred to select and remove a node with the highest weight to break cycles. The weight was defined as the number of cycles the node appeared in. The second used a GA to break cycles. The time complexity of Triskell strategy is non-polynomial because it takes non-polynomial time to enumerate all cycles before assigning weights to nodes.

Hewett et al. used a TDG as a dependency model which was created from UML class diagrams. In [10], they used a fast algorithm to find a (optimal) test order with minimal generic stubs. In [9], they improved their previous algorithm by adding an additional heuristic information. Two papers all used GBAs to break cycles and their time complexity is $O(n^2)$.

## III. Minimizing the Overall Stub Complexity

Briand et al. [4] used an ORD as a dependency model which was created from UML class diagrams and had four types of dependency relationships: inheritance, aggregation, association and usage. They proposed a coupling measurement technique to estimate stub complexity based on inter-class coupling information and used a GA to break cycles. They disallowed to remove inheritance edges and composition edges to break cycles.

Abdurazik et al. [1] used a WORD as a dependency model which was created from source code and had nine
types of dependency relationships: inheritance, implementation, composition, aggregation, association, dependency, etc. They provided a coupling measurement technique to estimate stub complexity using more fine-grained information and presented three different algorithms to break cycles. Each of them used a GBA to break cycles, and preferred to select and remove an edge (node) with the highest cycle-weight ratio (CWR). Their coupling measurement technique only simply assigned fixed values to inheritance edges and composition edges. The time complexity of their algorithms is non-polynomial because their ideas are similar to Triskell’s proposed by Hanh et al.

Borner et al. [3] used an ORD as a dependency model which was created from source code and had three kinds of dependency relationships: inheritance, association and dependency. They believed that test focus should be considered when performing integration testing. They used a simulated annealing algorithm and a genetic algorithm to find a (optimal) test order that considered not only the simulation effort but also the test focus.

Our previous work [17] used an extended WORD (EWORD) as a dependency model which was created from source code and had six types of dependency relationships: inheritance, implementation, composition, aggregation, association and usage. We proposed a coupling measurement technique to estimate stub complexity for all types of edges and used a random iterative algorithm (RIA) to break cycles. Our RIA algorithm used some properties of minimal feedback arc set and the idea of simulated annealing, which made it more effective.

Cabral et al. [6] used an ORD as a dependency model which was created from source code and had four kinds of dependency relationships: inheritance, composition, association and dependency. They used a multi-objective ant colony algorithm (MOCA) to generate a set of Pareto optimal solutions that achieved a balance between attribute complexity and method complexity.

IV. Analysis And Evaluation

Table I is a summary table, where the time complexity of all SBAs is not provided because their time complexity is related to specific implementation information. From this table, we can obtain some analysis results as follows.

- Published year: Three papers are published from 1995 to 2000, six papers are published from 2001 to 2005 and eight papers are published from 2006 to 2010, which indicates that more and more papers give attention to CITO problem.

- Objective: Though only five papers try to minimize the overall stub complexity, four of them are published from 2009 to 2010, which indicates that finding a (optimal) test order with minimal overall stub complexity is current hot topic. The reason is that minimizing the overall stub complexity is more reasonable than minimizing the number of stubs.

- Model: Seventeen papers are listed. Ten of them create their dependency models from source code and six of them are published from 2006 to 2010, which indicates that CITO problem is given more attention in reverse engineering research field. The reason is that source code can provide more fine-grained information.

- Algorithm type: Though five papers use search-based algorithms to break cycles, three of them are published in 2009 and 2010, which indicates that based-search algorithms have more utilization potentiality than based-graph algorithms. The reason is that based-search algorithms can escape from a local optimal solution and can be used to perform multi-objective optimization.

- Time complexity: Two papers are published in 2008 and 2009 whose time complexity is $O(n^2)$, which indicates that designing a fast algorithm to break cycle is given more attention in recent years. The reason is that some software systems have a large number of cycles to be broken.

- Algorithm constraint condition: From 2006 to 2010, seven papers allow to remove inheritance edges and composition edges to break cycles and only one paper does not, which indicates that it is acceptable to remove inheritance edges and composition edges to break cycles in recent years.

Some challenging problems encountered are identified and explained as follows.

- The models proposed for representing inter-class dependencies lack precision. An ORD and a TDG can represent polymorphism dependency relationships, but applying class hierarchy structure to construct these dependency relationships is too simply, because some dynamic dependency edges may be not exist.

- The techniques proposed for measuring inter-class coupling information lack precision. Different stubs may need different test effort to be constructed, so it is more reasonable to find a (optimal) test order with minimal overall stub complexity than minimal number of stubs. The number of accessed attributes and the number of called methods are usually used as inter-class coupling information to calculate stub complexity, but it is not sufficient to use these information to calculate stub complexity.

- The algorithms proposed for breaking cycles lack effectiveness and efficiency. For a large-scale application case, it is very difficult to design an effective and efficient algorithm to break cycles. Graph-based algorithms are usually very fast but they have no chance to escape from a local optimal solution. On the contrary, search-based algorithms are usually very slow and require to adjust a lot of parameters to improve their performance, but they have chances to escape from a local optimal solution.
V. Conclusions

In this paper, we first review some related work based on their objectives and then provide some analysis and evaluation. By analysis, it is found to be a good strategy for automatic integration test order generation to construct a more precise dependency model from source code, to calculate stub complexity using a more precise coupling measure technique, and to design an effective and efficient algorithm for breaking cycles.

References