TEST COVERAGE ANALYSIS FOR OBJECT ORIENTED PROGRAMS
Structural Testing through Aspect Oriented Instrumentation

Fabrizio Baldini, Giacomo Bucci, Leonardo Grassi and Enrico Vicario
Dept. Sistemi e Informatica, Università degli Studi di Firenze
Via S. Marta 3, 50139 Firenze, Italy

Keywords: Data Flow Analysis, Object Oriented Testing, Abstract Syntax Tree, Aspect Oriented Programming.

Abstract: The introduction of Object Oriented Technologies in test centered processes has emphasized the importance of finding new methods for software verification. Testing metrics and practices, developed for structured programs, have to be adapted in order to address the prerogatives of object oriented programming. In this work, we introduce a new approach to structural coverage evaluation in the testing of OO software. Data flow paradigm is adopted and reinterpreted through the definition of a new type of structure, used to record def/use information for test critical class member variables. In the final part of this paper, we present a testing tool that employs this structure for code based coverage analysis of Java and C^{++}programs.

1 INTRODUCTION

Testing is the process of verifying the correctness of a system through the exercise of its components and functionalities (Beizer, 1990). As a part of this process, coverage analysis is the activity of evaluating the degree of coverage attained by performed tests with respect to some abstraction of the system. This provides a measure of the quality of tests, and, indirectly, of the confidence about the absence of residual undetected faults. In structural approaches, these abstractions are derived from the implementation of the system according to different paradigms which may address control flow (Ntafos, 1988), data flow (Rapps and Weyuker, 1985), finite state behavior (Fujiwara et al., 1991) (Yannakakis and Lee, 1995).

In control flow analysis, coverage is evaluated with reference to the flow of control, measuring the number of covered statements, basic blocks, branches, conditions or paths. Data flow testing extends the approach by focusing the analysis on paths between definitions and uses of program variables (Rapps and Weyuker, 1985). This follows a basic rational for which "paths formed by definitions and uses of variables are a good place to look for errors in software" (Binder, 1994). In fact, if there's a fault in the program we have a good chance of detecting it by covering the statements where the faulty-written memory location is read. Data flow theory prescribes different coverage criteria. The most relevant is *All uses* which requires that the test suite exercises at least one def-use path between each variable definition and every subsequent use of the same variable.

While data flow theory has been developed and successfully practiced mainly on structured programming, its capabilities effectively answer to some of the specific complexities posed by OO programming. In fact, OO programming basically reduces the significance of control flow and, at the same time, augments coupling based on concurrent def/usage of attributes: control flow does not follow a regular hierarchical pattern as in structured programming and rather evolves in a graph-like often tangled manner; member variables have global visibility within the class so that class methods are coupled through the inner state of the object; the object state is maintained even when the flow of control is not located in the object methods; objects have global visibility within the program, and can be concurrently invoked by a variety of clients, thus extending coupling to the interclass level.

Several abstractions have been proposed to address these complexities in the testing process. Baudry, in (Baudry and Traon, 2005), proposes a structure called *Class Dependency Graph* (CDG), which is used to support design through a measure of testability and through the identification of some specific *anti-patterns* which are supposed to be most error-prone. The CDG is derived from a detailed UML class diagram, which should, at the same time, capture implementation details and designer's intent; the graph represents the dependency among classes deriving from inheritance/implementation/override and from delegation relations with no reference to def/use coupling on class attributes. (Hong et al., 1995) and (Gallagher et al., 2006) propose a special type of control flow graph, called *Class/Component Flow Graph*, which describes the interaction among methods of one or more classes, using the formalism of state transition systems. The approach emphasizes state behavior more than data flow dependency on variables accesses and requires that program behavior be modeled as an FSM in a so-called *Class State Machine*. However, this cannot be automatically derived from code inspection and does not effectively encompass programs with dynamic object creation. In (Harrold and Rothermel, 1994), the structure of a program is modeled as a *Class Control Flow Graph* which extends the concept of inter-procedural control flow graph (Pande and Landi, 1991) to the OO context by capturing intra-method, inter-method and intra-class control flow deriving from decision structures and function calls. The graph is proposed as a basis to select test cases in data flow style for the limited scope of each single class.

In this paper, we address the application of data flow criteria to test coverage analysis of object oriented programs. We propose a technique and a tool performing static analysis of C^{++}or Java code to identify def/use dependencies in the access to selected salient attributes which occur among methods in the intra-class and inter-class scope; these are reported in so-called *Def/Use Tables*, replacing conventional control flow graphs in the representation of cases that should be covered for reliable testing. We then describe how Aspect Oriented Programming (AOP) can be used to efficiently instrument the code to log method invocations and to provide a basis for the offline evaluation of the coverage attained with respect to def/use tables, in the reference of various data flow criteria.

The rest of the paper is organized in five sections. In Sect. 2 and 3 we introduce Def/Use Tables as a way of representing data flow information and we present the set of syntactic rules used for their automated retrieval through code inspection. Sect. 4 formally defines the coverage criteria developed on Def/Use tables and describes how these criteria are evaluated though the offline analysis of an execution log produced by the aspect oriented (Kiczales et al., 1997) instrumentation of the program. In Sect. 5, we provide a description of two practical tools developed to automate coverage analysis in the testing of Java and C^{++}programs. Finally, a validation of these tools, with respect to design patterns, is presented.

2 OBJECT ORIENTED DATA FLOW

Data flow testing employs an annotated version of control flow graph, called *definition-use graph* (Rapps and Weyuker, 1985), to record the structural information of the program under test. Basic blocks represent the building unit of the graph. Each node of the definition-use graph is labeled with the set of variable definitions (*def*) and uses (*c-use* or *p-use*) performed in the corresponding basic block.

In our approach, we identify basic blocks with method invocations without unfolding the control flow graph within each method, this facilitates code instrumentation for coverage evaluation and simplifies the analysis so as to allow to scale to the inter-class level; at the same time, in code developed according to a good programming practice, this simplification does not hide relevant information due to the high cohesion between statements contained within the body of each single method.

Moreover, we do not distinguish between *p-uses* and *c-uses*. This is not restrictive for *All Uses* coverage and avoid the needs to parse the statements that encompass the expressions in which an attribute is referred to.

2.1 Def/Use Tables

The definition-use graph provides a global representation of data accesses in the program. In our work, the graph structure has been replaced by an annotated structure, called *Def/Use table*, expressing data flow information with respect to single variables.

We define a Def/Use table as follows: let x be the member variable under observation. Let $T_{use}(x)$ be the set of methods (functions) performing at least a use on x and let $T_{def}(x)$ be the set of methods (functions) performing at least a definition on x. Let n and m be the size of $T_{use}(x)$ and $T_{def}(x)$, respectively. A Def/Use table for variable x is a n-row, m-column table where rows are labeled with the elements of $T_{use}(x)$ and columns are labeled with the elements of $T_{def}(x)$. Note that, if x is read and written within the body of one method m, then m will be an element of both $T_{use}(x)$ and $T_{def}(x)$ and will appear in both a row and a column of the *x* table. A Def/Use table representation for attribute `age` of the following class `Person` is reported in Table 1.

```
class Person {
   public:
      Person() {age = 0;}
      ˜Person(){}
      void birthday() {age = age +1;}
      int getYears()  {return age;}
      bool isAdult()  {return age>=18;}
   private:
      int age;
};
```

Table 1: Def/Use table for the attribute age of class Person.

class: `Person` var: `age`		DEF	
		Person()	birthday()
USE	isAdult()		
	getYears()		
	birthday()		

Each cell of a Def/Use table can be regarded as a path between a *def* node and a *use* node of the definition-use graph. The set formed by such *definition clear paths* is thus given by the Cartesian product between the columns and the rows of the table. The number of the resulting (*DEF_method*, *USE_method*) sequences is anyway to be considered as a theoretical upper bound of the number of feasible paths. The identification of infeasible paths is a complex data flow issue (Holley and Rosen, 1981) and has not been part of this research. Indeed, during coverage evaluation, we accepted the structural or semantical infeasibility of some specific *def-use* pairs formed by methods not sequentially executable by the program.

2.2 Table Levels

Def/Use tables context of analysis can be extended to include those methods which indirectly cause an access to the variable under observation by invoking another method that defines or uses the variable.

Let x be the variable under observation. Let $T(x,i)$ be the set of functions contained in table of level i for variable x. Let $T_{def}(x,i)$ and $T_{use}(x,i)$ be the subsets of $T(x,i)$ identifying the functions that perform a definition or a use on x. As a result, $T(x,i) = T_{def}(x,i) \cup T_{use}(x,i)$ and, generally, $T_{def}(x,i) \cap T_{use}(x,i) \neq 0$. Table of level $i+1$ can be built from table of level i as follows:

- $T_{def}(x,i+1)$ is formed by all methods (functions) that directly invoke at least one method (function) contained in $T_{def}(x,i)$
- $T_{use}(x,i+1)$ is formed by all methods (functions) that directly invoke at least one method (function) contained in $T_{use}(x,i)$

As an example, consider the implementation of class `Parent`:

```
class Parent: public Person{
   public:
      Parent(){...};
      void childBirthday(){
         if(child.isAdult()){
         ...
         }
         celebrate(child);
      }
   protected:
      int celebrate(Person& p){
        p.birthday();
        return p.getYears();
      }
   private:
      Person child;
};
```

`Parent::celebrate(Person&)` invokes directly `Person::birthday()` and `Person::getYears()` which are in the table of level 0 for the attribute `age`; according to this, `Parent::celebrate(Person&)` will be recorded in level 1 table. The final form of level 1 table for `Person::age` will then be:

Table 2: Def/Use table of Level 1 for variable age.

class: `Person` var: `age` level: 1		DEF
		celebrate(Person&)
USE	celebrate(Person&)	
	childBirthday()	

Higher order tables provide an essential means for the completeness of the representation of the definition-use graph since they enable the evaluation of method interactions at both the inter-method and inter-class level (Harrold and Rothermel, 1994).

3 SOURCE CODE MODEL

The set of definitions and uses reported in a Def/Use table is given by the union of methods performing a direct or indirect access on the member variable under observation. Direct accesses are given by explicit references to the attribute in infix, prefix, postfix and assignment expressions contained in the method body. An indirect access happens when the examined method invokes another method by passing the variable under test as one of its parameter and/or when the current method changes the state of the attribute[1] by invoking a method on it (i.e. the variable is of an abstract data type).

In order to identify each expression contained in a method definition and addressing the variable the table refers to, we require an analyzable abstraction of the program under test. This model is provided by the Abstract Syntax Tree (AST) (Kuhn and Thomann, 2006). Code parsing is performed by exloring the nodes of the AST of each method, according to the following rules: let *x* be the table attribute.

1. Every assignment, prefix or postfix expression, that contains *x*, is examined as a node in the method AST to detect a possible direct *def* or *use* on *x*.
2. Every method invocation node of the AST is visited to detect a possible indirect *def* or *use* on *x*.
3. If a *use* and a *def* for that method have already been registered, the analysis for that method stops.

While the retrieving of direct accesses is a relatively simple task to accomplish, the activity of identifying indirect accesses strongly depends on the semantic of the adopted programming language. The only general rule that can be derived consists in the migration of the scope of analysis from the calling method to the called one.

As an example of indirect attribute access, we will refer to class `Parent`. Method `childBirthday()` invokes `isAdult()` on the attribute `child`. This method has no side effects for the invoking object so only a *use* is registered for `childBirthday()`. Moreover, `childBirthday()` passes a reference to `child` as a parameter to `Parent::celebrate(Person& p)`. The status of `p` is changed through the invocation of `p.birthday()`, consequently, `childBirthday()` is considered to perform a definition on `child`. The resulting Def/Use table for `child` is presented in Table 3.

Table 3: Def/Use table for attribute child.

class: `Parent` var: `child` level: 0		**DEF**
		childBirthday()
USE	childBirthday()	

[1]The state of an object is a predicate on the values of its member variables.

4 COVERAGE ANALYSIS

4.1 Logging Aspect

Aspect Oriented Programming (AOP) was used to instrument the code of the program under test and to produce a log file containing the trace of tested behaviors.

An AOP language is a language containing expressions for the encapsulation of crosscutting concerns into single units called aspects. An aspect is a modular unit of crosscutting implementation that can alter the behavior of multiple classes belonging to the non-aspect part of the program.

Logging is a type of crosscutting concern that AOP languages are able to address; several works (Chen et al., 2004), (Rajan and Sullivan, 2005), have employed this technique for structural and functional analysis of object oriented programs.

Our AOP model is created according to the program structure and to the scope of analysis determined by the tester. An aspect module is the result of an aspect generator that, on the basis of the Def/Use tables, automatically states which method calls must be instrumented. The resulting aspect is formed by a pointcut whose joinpoints are the elements of the tables. Two different advices are defined on the pointcut to detect when the flow of execution enters the body of a method and when the execution leaves it. The log file records these advices in XML format indicating, for each entry, the ID[2] of the class instance the logged method refers to, the identity of the process that generated the method execution and a *message* field used by the program to trace the execution sequence.

4.2 Coverage Criteria

Def/Use tables serve as a tracking utility to record couples of methods sequentially executed. Every cell of a table corresponds to a *def-use* pair between methods and functions of the same level, and can be marked with the *definition clear* path effectively exercised by test cases. To retrieve the paths covered during test execution, three rules have been defined to individually evaluate each method of the logged sequence:

- *Def method*: the method is labeled as the last one that performed an assignment to the variable.
- *Use method*: if there is a method labeled as *last def* then a *definition clear* path has been covered;

[2]In Java, the value of the ID field is given by an hashcode representation of the object. In C^{++}, the ID is given by the object's memory location.

the information is registered in the Def/Use table and the *last def* label is removed.

- *Def-use method*: the method performs both a reading and a writing on the variable. Conventionally, the *use* access is notified before the *def* access. Before reassigning the *last def* label, a check is performed to detect a possible *def-use* chain.

Classical data flow metrics have been adapted to the object oriented context, deriving three different coverage criteria.

- *All nodes*: this criterion represents the weakest of the three. Full coverage for this criterion is reached when test execution exercises every element contained in the Def/Use table.
- *All defs*: full coverage for this criterion requires the test set to exercise at least one *use* element for every *def* element contained in the table.
- *All uses*: full coverage for this criterion requires the test set to exercise every *use* element for every *def* element contained in the table. This criterion implies that every ordered couple of methods, belonging to the cartesian product between the rows and the column of the Def/Use table, is executed during the testing phase.

In our approach, if coverage analysis is performed on a set of member variables, the total coverage measure is derived from the partial measures, individually evaluated on each Def/Use table.

Instance control is a required means to guarantee that each *def-use* pair, reported in a table, is actually related to the same variable. Every time a nominal *definition clear* path is encountered, a check is performed to verify the correspondence between the identities of the object that invoked the *last defining* method and the object invoking the *use* method.

5 TOOL SUPPORT

To support the proposed technique we implemented two testing tools named *CppTest* and *JavaTest*. Both programs have been developed as plug-ins for the Eclipse platform, and use the CDT and JDT tools for structural analysis of source code. AspectC^{++}and AspectJ were also used to perform code instrumentation for the purposes of test logging. *CppTest* and *JavaTest* perform structural analysis of source code, create Def/Use tables for the user-defined variables, instrument the code and evaluate the coverage results of a test set execution.

To validate both the technique and the tools we performed experiments on a workbench comprised of a set of design patterns (Gamma et al., 1995) exhibiting a variety of interactions and structural dependencies which are common in the good practice of OO programming.

As an example, Table 4 represents the Def/Use table extracted from the implementation of an Observer pattern (Gamma et al., 1995). In the pattern, the attribute `observers`, represents the list of references to the observers previously attached to the subject. The table indicates which are the possible def/use dependencies among methods, and it marks with a token which of them have been actually exercised in a testing suite including the following testing sequence:

```
1 7021720 - Subject::Subject()
2 7019640 - Subject::Subject()
3 7019640 - void Subject::attach(Observer*)
4 7021720 - void Subject::attach(Observer*)
5 7021720 - void Subject::detach(Observer*)
6 7021720 - Subject::~Subject()
7 7021720 -   void Subject::detach(Observer*)
8 7019640 - void Subject::notify()
```

The path `attach()-detach()`, i.e. the registration and removal of one observer, is provided by lines 4 and 5. The destruction of one Subject instance is given by lines 6 and 7 which cover the def-use pair `detach()-∼Subject()`. Lines 3 and 8 cover the `attach()-notify()` path by updating the status of a registered observer.

6 CONCLUSIONS

We adapted significant concepts of data flow theory to the context of OO programming, and we proposed a technique of coverage analysis supported by tools for C^{++}and Java programs.

Def/Use relations appear to catch semantics relevant for class interaction, thus establishing a correspondence between the completeness of a requirement-based test case selection activity and the measure of structural data-flow coverage achieved by those tests. Besides, the adoption of Def/Use tables in place of a labeled graph provides a compact representation of method interactions focused on critical attributes that capture the intent of the composition of classes. This enables abstraction which is essential to scale up the scope of analysis in the integration testing.

The proposed technique and tools can be effectively cast into a methodology that spans over development and verification activities of an UP-like software lifecycle. In the development process, designers

Table 4: Def/Use table for the Observer pattern. Bullet items indicate test covered def-use pairs.

class: `Subject` var: `observers` level: 0		**DEF**		
		attach(Observer*)	detach(Observer*)	Subject()
USE	∼Subject()		●	
	attach(Observer*)			●
	detach(Observer*)	●		
	notify()	●		

are supposed to document the relevant attributes of salient classes which are most critical for inter-class interaction (Cockburn, 1996). This can be done as an annotation of UML class diagrams comprising the specification or the implementation model (Fowler, 2003). In the testing stage, Def/Use tables are automatically derived through source code analysis, and code is automatically instrumented using Aspect Oriented Programming. The application is then tested on a suite of cases selected according to any specific approach (e.g. derived from use cases (Heumann, 2001)), and their execution is logged to a file containing the trace of the methods effectively exercised. Coverage analysis is finally performed through offline check of the def-use paths reported in the log file and contained in the tables.

REFERENCES

Baudry, B. and Traon, Y. L. (2005). Measuring design testability of a uml class diagram. *Information and Software Technology*, 1(47).

Beizer, B. (1990). *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA.

Binder, R. V. (1994). Design for testability in object-oriented systems. *Commun. ACM*, 37(9):87–101.

Chen, Y., Qiu, W., Zhou, B., and Peng, C. (2004). An automatic test coverage analysis for systemc description using aspect-oriented programming. In *Computer Supported Cooperative Work in Design. Proceedings. The 8th International Conference on, Vol. 2*, pages 632 – 636.

Cockburn, A. (1996). The interaction of social issues and software architecture. *Commun. ACM*, 39(10):40–46.

Fowler, M. (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Fujiwara, S., von Bochmann, G., Khendek, F., Amalou, M., and Ghedamsi, A. (1991). Test selection based on finite state models. *IEEE Trans. Softw. Eng.*, 17(6):591–603.

Gallagher, L., Offutt, J., and Cincotta, A. (2006). Integration testing of object-oriented components using finite state machines: Research articles. *Softw. Test. Verif. Reliab.*, 16(4):215–266.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Harrold, M. J. and Rothermel, G. (1994). Performing data flow testing on classes. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 154–163. ACM Press.

Heumann, J. (2001). Generating test cases from use cases. *The Rational Edge*. Retrieved January 20, 2007, from www.therationaledge.com.

Holley, L. H. and Rosen, B. K. (1981). Qualified data flow problems. *IEEE Trans. Softw. Eng.*, 7(1):60–78.

Hong, H. S., Kwon, Y. R., and Cha, S. D. (1995). Testing of object-oriented programs based on finite state machines. In *APSEC '95: Proceedings of the Second Asia Pacific Software Engineering Conference*, page 234. IEEE Computer Society.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242.

Kuhn, T. and Thomann, O. (2006). Abstract syntax tree. Eclipse Corner Articles. Retrieved October 23, 2006, from http://www.eclipse.org/articles/index.php.

Ntafos, S. C. (1988). A comparison of some structural testing strategies. *IEEE Trans. Softw. Eng.*, 14(6):868–874.

Pande, H. D. and Landi, W. (1991). Interprocedural def-use associations in c programs. In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pages 139–153, New York, NY, USA. ACM Press.

Rajan, H. and Sullivan, K. (2005). Aspect language features for concern coverage profiling. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 181–191. ACM Press.

Rapps, S. and Weyuker, E. J. (1985). Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11(4):367–375.

Yannakakis, M. and Lee, D. (1995). Testing finite state machines: fault detection. In *Selected papers of the 23rd annual ACM symposium on Theory of computing*, pages 209–227. Academic Press, Inc.