TEST COVERAGE ANALYSIS FOR OBJECT ORIENTED PROGRAMS

Structural Testing through Aspect Oriented Instrumentation

Fabrizio Baldini, Giacomo Bucci, Leonardo Grassi, Enrico Vicario
Dept. Sistemi e Informatica, Università degli Studi di Firenze
Via S. Marta 3, 50100 Firenze, Italy
Email: {fbaldini, bucci, grassi, vicario}@dsi.unifi.it

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 (Cornett, 2005). 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. According to this, data flow starts from the selection of test critical variables and examines acyclic paths, called *def-use paths*, joining the location where a variable is assigned a value (called *def*) to the first locations where that same variable is used either to perform a computation (called *c-use*) or to decide a condition (called *p-use*). 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 mainly developed and successfully practiced on structured programming code, the growing relevance of OO development poses a number of new specific challenges. This seems to reduce the significance of basic control flow and rather give relevance to data flow approaches. Data flow analysis allows to overcome a variety of issues related to the substantial inconsistency of the flow of control in OO programs. Some of these issues are implicitly due to the fluidity and semantic richness of OO languages, others are ascribable to the architectural complexity of OO applications and to some features related to visibility of elements in OO programs: member variables have global

visibility within the class; as a result, class methods are coupled through the inner state of the object which is persistent even when the flow of control is not located in the object's methods. Symmetrically, objects have global visibility within the system and this determines a state of concurrency on the part of client methods that access system functions.

Several works have addressed the application of data flow testing to OO programs. In (Harrold and Rothermel, 1994), a data flow approach is adopted in the intra-method, inter-method and intra-class context of class testing. The approach extends the inter procedural control flow since no explicit correlation is presented between class member functions and variables. In particular all attributes are treated uniformly and no indication is provided on how to choose critical attributes within the program. (Hong et al., 1995) and (Gallagher et al., 2006) performed data flow analysis for single-class and integration testing, respectively, on a special type of control flow graph, called *Class/Component Flow Graph*, a structure derived by a FSM model of class behavior, called *Class State Machine*. The main drawback of this technique resides in the fact that a full set of specifications has to be *a priori* determined in order to obtain a state representation of the program. Baudry in (Baudry and Traon, 2005) proposes an approach aimed at reducing the testing effort in the earliest stages of software development. A measure of OO program testability is defined through the identification, on a structure called *Class Dependency Graph* (CDG), of some specific *anti-pattern* configurations of the software under test. The creation of a CDG requires a detailed UML class diagram representation of the program. This seems to be the main weakness of the approach since it relies on a type of information which, in the design phase, is rarely provided by the developer and not entirely representative of his/her intent.

In this paper we propose a data flow and code based methodology for structural coverage analysis of object oriented programs. Our approach to class testing is focused on selecting a number of critical attributes identified according to functional requirements and/or to the designer's intent and, successively, on identifying the full set of methods referencing these attributes with or without side effects (i.e. in *def* or *use* mode) . This data flow information is then reported into a table structure representing every *def-use* path, with respect to each variable under observation, that can be extracted from source code investigation. Coverage analysis is then performed through the evaluation of the sequence of paths effectively traversed during the execution of test cases.

Section II introduces Def/Use tables as a new way of representing data flow information. This serves as a basis for the definition of the coverage measure adopted in this work.

Section III presents the set of syntactic rules and language specific assumptions employed to retrieve data flow interactions from code.

Section IV concentrates on how a coverage measure is determined through the analysis of an execution log and how this log is obtained through aspect oriented instrumentation of the program (Kiczales et al., 1997).

In Section V, 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 pattern subsystems 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. In the definition-use graph each node 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, taking into account that the encapsulation of information within classes and the "one concern per class" principle greatly augment the semantic cohesion between statements contained in a single method's body. Moreover, we do not distinguish between *p-uses* and *c-uses*. The distinction is not relevant for *all uses* coverage and also addresses some detection difficulties which indicate a kind of definition flaw. In fact, from a syntactic view, the distinction between computational and predicate uses means that code analysis has to be performed at the statement level of the language. This remarkably increases the complexity of the review, since the analysis is charged with the task of individuating the modality through which a variable is addressed. Further, the massive amount of *p-uses* that would originate from vector indexation, from the use of pointers variables, from while and do-while statements would cause an unbalanced relationship between *p-uses* and *c-uses* and, consequently, between the *all p-uses* and *all c-uses* coverage criteria.

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();
      ~Person();
      void birthday();
      int getYears();
      bool isAdult();

   private:
      int age;
};

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

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, i.e. a sequence formed by a method modifying the value of the variable under observation and a method reading that variable. 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 such 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

Table 1 is a table of level 0, meaning that its rows and columns are indexed by methods that, in their body, directly reference the attribute `age`. Indeed, there are a number of methods which indirectly cause an access to the variable under observation by invoking another method that defines or uses the variable. To extend Def/Use tables to this context of analysis we introduce the concept of *table levels*.

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 $T(x,i)$ respectively 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();

   protected:
      int celebrate(Person& p);

   private:
      Person child;
};
```

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

`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. In turn `Parent::childBirthday()` invokes `Parent::celebrate(Person&)` which belongs to $T_{def}(age,1)$ and $T_{use}(age,1)$; therefore `Parent::childBirthday()` belongs both to $T_{def}(age,2)$ and $T_{use}(age,2)$. The final form of level 1 and level 2 tables 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()	

Table 3: Def/Use table of Level 2 for variable age.

class: `Person` var: `age` level: 2		**DEF**
		childBirthday()
USE	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). As an example, consider the case where a class attribute `int v` is accessed only through the methods `int get()` and `void set(int value)`. In this situation, common in the practice of OO programming, every reading or writing on the variable (included those performed by methods belonging to the same class of the attribute) is done by invoking the `get()` and `set(int)` methods. As a consequence, the table of level 0 contains only the method `set(int)` in the *def* column and only the method `get()` in the *use* row. A trivial 100% *all uses* coverage can then be achieved through the sequential invocation of `set(int)` and `get()`, thus masking all other accesses to `v`. Multilevel analysis, through tables of level $i > 0$, reveals the real behavior of class methods by examining the paths between all methods that call `get()` and `set(int)` and actually reference `v`.

3 CODE ANALYSIS

Def/Use tables can be extracted automatically through lexical and syntactic analysis of the source code.

3.1 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 body of the method. 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). An AST is a finite, directed, labeled tree that represents the operations that the compiler has to fulfill in order to translate the source code. In our approach, we use the AST model to explore the structure of the different compilation units down to the expression level, searching for nodes containing references to the attribute. Once the model of the source code is available, the context of analysis is determined through the definition of two parameters:

- *Search scope*: local[2], global[3], local with internal classes.
- *Depth of the hierarchy tree*: the maximum number of subtype levels that are explored to analyze possible method redefinition.

Let x be the table attribute. Code parsing is performed with the exploration of the nodes of every method's AST, according to the following rules:

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

[2]Concerning only methods contained in the class that defines the attribute.

[3]Concerning the entire project.

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*, according to the criteria defined in sec. 3.2.
3. If a *use* and a *def* for that method have already been registered, the analysis for that method stops.

The Eclipse Platform provides a set of API for the analysis and manipulation of program resources, in particular Java Development Tools (JDT) and C/C^{++}Development Tools (CDT) support the creation and the exploration of ASTs for Java and C^{++}code. We employed both JDT and CDT in the development of the two tools presented in this paper.

3.2 Indirect Attribute Access

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 (see Section 3.1). In this section we describe the set of rules that guided indirect access retrieving for C^{++}. Let *x* be the variable under observation. If *x* is passed as a parameter from a method to another the analysis proceeds iteratively, examining every statement contained in the body of the current method and then moving its scope to the invoked one. Let *caller()* be the currently examined method and let *callee()* be the invoked method. If the parsing of the expressions in *caller()* ends with both a *def* and a *use* on *x* then *callee()* is not examined, otherwise four cases are possible:

- *x* is passed to *callee()* by copy. If *callee()* reads the argument corresponding to *x* then a *use* is recorded for *caller()* method. No assignment on the parameter performed by *callee()* is reported as a *def* on *x*.
- *x* is passed to *callee()* by pointer. A *def* and/or a *use* on *x* is reported only if *callee()* reads and/or writes the variable through an adequate number of dereferencing operators applied to the pointer.
- *x* is passed to *callee()* by reference. Every *def*/*use* performed by *callee()* on the parameter is reported as a *def*/*use* on *x* performed by *caller()*.
- The definition of *callee()* is not available. If the passing is done by copy a *use* on variable *x* is reported. If the passing is done by pointer or by reference both a *def* and a *use* on *x* is registered for *caller()*.

Otherwise, if the indirect access is given by a change in the state of variable *x* due to a method invocation on it, three different situations are possible: let *A* be the statically known type of *x* in *caller()* method and let *callee()* be the method invoked on *x*.

- *callee()* performs an assignment on any of the member variables of *A*: *caller()* is considered to perform both a *use* and a *def* on *x*.
- *callee()* does not change any attribute of *A* then only a *use* is recorded for *caller()*[4].
- If the source code is not available then a *def* and a *use* are conventionally registered for *caller()* except for the case where *callee()* has a *const* declaration.

For every virtual method examined in the parsing of indirect accesses, a check is done to see if the context of invocation allows the substitution between types belonging to the same class hierarchy. If polymorphism is possible then every redefinition of the invoked method in its subtypes is added to the analysis.

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 4.

Table 4: Def/Use table for attribute child.

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

4 COVERAGE ANALYSIS

4.1 Logging Aspect

We used Aspect Oriented Programming (AOP) to instrument the code of the program under test and to

[4]In this case *x* variable is considered to be read anyway by *caller()* method.

subsequently produce a log file containing the trace of the test case execution.

A variety of works have employed this technique for structural and functional analysis of object oriented programs. In (Chen et al., 2004), AOP is applied to automate functional coverage analysis of SystemC descriptions for a hardware/software system. In (Rajan and Sullivan, 2005) white box coverage is performed through Aspect Oriented Programming by prototyping an approach to structural verification called *concern coverage*. Rajan introduces *concern coverage* as an extension of the aspect concept, developed to refine the granularity of AOP advices[5]. A language, Eos-T, is adopted to express different testing and coverage criteria as pointcuts[6] of the aspect model.

Historically, AOP has been considered too "coarse grained" for the retrieving of classical data flow information since aspects cannot isolate basic blocks. We could use AOP for the creation of a logging tool since our work is focused on methods listed in Def/Use tables and not on single statements. Our pointcut 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 as follows.

```
<record>
   <date>2006-09-09T14:29:20</date>
   <millis>1157804960899</millis>
   <sequence>0</sequence>
   <logger>it.dsi.javatest</logger>
   <level>FINER</level>
   <class>1102920</class>
   <method>Person()</method>
   <thread>10</thread>
   <message>ENTRY</message>
</record>
```

`Class` field identifies the ID[7] of the class instance the method refers to, `thread` field indicates the identity of the process that generated the method execution and the `message` field is used by the program to trace the execution sequence. Possible values for this field are:

- `ENTRY`, indicating that program execution is entering the `<method>` body.
- `RETURN`, indicating that `<method>` has been executed successfully[8].
- `CFLOW`, indicating that the `<method>` belongs to the flow of control of the current method in the table.

The `CFLOW` label is applied to every method of level 0 table when the table under test is of level $i > 0$. `CFLOW` methods are reported into the log file to allow an effective instance control on the elements of table *i*. In higher order table analysis the attribute that is relevant for an effective instance control is the variable on which level 0 methods perform an access. *Def-use* pairs are reported on a table of level $i > 0$ if the corresponding level 0 methods refer to the same object.

4.2 Test Sequences

Coverage analysis is performed through the identification of the different sequences of methods and functions traversed by the execution of test cases. In this phase, 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. The trace of execution is read from a log file (see Section. 4.1) and then translated into the corresponding table, in the form of *def-use* pairs. 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; 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.

The sequential form of the log file implicitly defines the ordering among the functions executed at runtime. As an example we refer to the following method sequence and, in table 5, we report covered *def-use* pairs for variable `age`.

[5] In AOP, the point where aspect code and program code are weaved together is called *joinpoint*. An *advice* is the portion of code that is executed when the program execution reaches the joinpoint.

[6] A *pointcut* is a description of the set of joinpoints.

[7] Hashcode representation of the object for Java and its memory location for C^{++}.

[8] i.e. Without throwing exceptions.

```
...
Person person;
person.birthday();
person.birthday();
person.isAdult();
...
```

Table 5: Def/Use table with covered def-use pairs.

class: `Person` var: `age` level: 0		**DEF**	
		Person()	birthday()
USE	isAdult()		•
	getYears()		
	birthday()	•	•

Instance control is a required means to guarantee that each *def-use* pair reported in a table is actually related to the same variable. During coverage analysis, every time a nominal *definition clear* path is encountered, a check is performed (see Section 4.1) to verify the correspondence between the identities of the object that invoked the *last defining* method and the object invoking the *use* method. The presence of public attributes or C^{++}friend functions limits the effectiveness of instance control method described above. In practice, if an attribute is accessed by methods belonging to an object of a different type, level 0 table will register a *def* or a *use* method belonging to an external class. Some of the *def-use* paths will then result infeasible since there will be no correspondence between the ID of the object externally accessing the attribute and the class ID of member functions. Such limit can be overcome through the use of a specific coding standard that imposes the respect of information hiding in the target program.

4.3 Coverage Criteria

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 the execution of the test set 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 represents the strictest of the three implying, in our domain, 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 these criteria are applied to the Def/Use table structure, meaning that 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.

5 TOOL SUPPORT

CppTest and *JavaTest* are the testing tools implemented in this project. Both programs have been developed as plug-ins for the Eclipse platform in order to exploit the services provided by the CDT and JDT tools for structural analysis of source code and for a better integration with the aspect oriented extensions of C^{++}and Java, AspectC^{++}and AspectJ, respectively. *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, according to the guidelines exposed in Section 4.

5.1 Design Patterns

CppTest was employed to perform coverage analysis of a set of test cases run against an example program. A collection of different subsystems, containing a number of pattern configurations, have been individually reviewed in order to validate our approach and assumptions about the criticality of specific member variables. Design patterns provided a significant workbench for our testing tool since the different interactions and structural dependencies among objects in a pattern can be retained as exemplary of the prerogatives of object oriented design. The presence of polymorphic virtual calls, shadow invocations and dynamic binding granted the use and validation of every analysis rule defined in Section 3. In order to create a table we had to select a specific member variable, critical to the functions of the pattern. When the classes under test did not provide one of their own, the analysis required the injection of a specific attribute to observe the sequence of method invocations on it. The observation of critical attributes demonstrated that coverage on a Def/Use tables was strictly related with the behavior of objects interactions in the pattern scheme.

6 CONCLUSION

We adapted some of most significant concepts of data flow theory to the context of OO classes. The methodology that followed such adaptation has proved to properly fit the critical features of object oriented programs. The adoption of Def/Use tables in place of a labeled graph has provided us a compact representation of method interactions focused on single attributes. The definition of table levels allowed to evaluate these interactions while incrementally scaling the scope of analysis. Aspect Oriented Programming was employed to achieve a dynamic and context independent logging device without having to manually instrument the code under observation. These concepts were implemented in two different tools devoted to test coverage analysis of Java and C^{++}software that have been validated on a number of design patterns configurations. These tools used Def/Use tables either as a structure to maintain the information needed for code instrumentation and as a coverage analysis device.

The applicability of our approach relies on the conformity of the program under test to the practices of OOP. Instance control technique, discussed un Section 4.2, finds its limitations in the check of member variables that can be accessed by external entities. This issue is inherently correlated to a design violation of the information hiding principle of the target program and can be therefore resolved outside the verification phase, with the aid of the developer that we assumed as an active subject of the testing method.

Further research is possible in order to apply this approach, initially conceived for coverage analysis, to test case selection activity.

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.

Cornett, S. (2005). Code coverage analysis. Bullseye Testing Technology. Retrieved July 20, 2006, from http://www.bullseye.com/coverage.html.

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.

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.

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.

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.