A Framework for Testing Object-Oriented Components *

Ugo Buy†, Carlo Ghezzi‡, Alessandro Orso‡, Mauro Pezzè‡ and Matteo Valsasna‡

†EECS Dept. (M/C 154)
University of Illinois
851 South Morgan Street,Chicago, IL 60607
Phone: (312) 413-2296, Fax: (312) 413-0024
buy@eecs.uic.edu

‡Dip. di Elettronica e Informazione
Politecnico di Milano
P.za Leonardo da Vinci, 32, I-20133 Milano, Italy
Phone: +39-02-2399-3638, Fax: +39-02-2399-3411
[ghezzi|orso|pezze|valsasna]@elet.polimi.it

1 Introduction

The terms "component" and "component-based software engineering" are relatively recent. Although there is broad agreement on the meaning of these terms, different authors have sometimes used slightly different interpretations. Following Brown and Wellnau [4], here we view a component as "a replaceable software unit with a set of contractually-specified interfaces and explicit context dependencies only." A component is a program or a set of programs developed by one organization and deployed by one or more other organizations, possibly in different application domains.

To date, many component systems are designed and developed using object technology. The same is true of middleware architectures that support the deployment of components, such as Java Beans [3], CORBA [1], and DCOM [5]. Although there is consensus that the issues in component-based software engineering cannot be solved merely by object technology [4, 14], these technologies will clearly play a fundamental role in the production of software components [15]. It is quite likely that developers will use with increasing frequency object technologies in the design and implementation of components. For this reason, in the sequel we focus on testing of components that consist of an object or a set of cooperating objects. In addition, we consider messages sent to a component as method invocations on the objects contained in the component.

Programs developed with object technologies have unique features that often make traditional testing techniques inadequate. An important feature of these programs is that the behavior of a method may depend on the state of the method receiver. Since the state of an object at a given time reflects the sequence of messages received by the object up to that time, erroneous behaviors may be revealed by exercising specific sequences of messages. Component testing should identify sequences of messages whose execution is likely to show the existence of faults; however, traditional unit and integration testing techniques do not produce this kind of information [2]

Our method uses various kinds of structural analyses in order to produce sequences of method invocations for the component under test. A byproduct of these analyses is the definition of formal specifications in the form of preconditions and postconditions for the methods contained in the component under test. When performing integration testing, the method sequences resulting from the analysis of single components are then used jointly.

Developers and users of a component have different needs and expectations from testing activities. Our framework for component testing can benefit both parties involved. On the one hand, the method sequences we identify can be used as test cases to be applied during the development of the component under test. On the other hand, during integration testing component users can combine sequences generated for single components to validate groups of components and their interactions. In addition, the set of formal specifications produced by symbolic execution can be extremely valuable to prospective users of a component as they evaluate alternative implementations for a given task.

* ‡This work has been partially supported by ESPRIT Project TWO (Test Warning Office - EP n.28940) and by Italian Ministero dell'Università e della Ricerca Scientifica e Tecnologica (MURST) in the framework of the MOSAICO (Design Methodologies and Tools of High Performance Systems for Distributed Applications) Project.

Many other techniques have been defined that can produce method sequences for object or component testing (e.g., [9, 6, 11]). In contrast with some of those techniques, we do not require the existence of formal specifications in order to produce method sequences [9, 6]. In this regard, our approach is similar to the technique defined by Kung et al. [12]. An advantage of our approach is that we can produce a meaningful set of method sequences even when symbolic execution and automated deduction techniques cannot be fruitfully applied to the component code. If these sequences are not sufficient, the user can be asked to complement the information obtained by the automatic techniques, to produce a better set of invocation sequences. An additional advantage is that in our case the integration of method sequences obtained for different components is relatively straightforward.

The paper is organized as follows. Section 2 gives an overview of the proposed testing methodology. Section 3 introduces the vending machine example used to illustrate relevant portions of the methodology. Sections 4, 5, and 6 discuss the techniques for data flow analysis, symbolic execution and automated deduction. An example of the technique for integration testing is provided in Section 7. Finally, conclusions and future research directions are discussed in Section 8.

2 Overview of Testing Framework

The approach to the generation of message sequences for component testing is based on two kinds of analyses. The first phase considers each component in isolation. This phase leads to the definition of a set of method sequences for each component under test (CUT). The second phase considers a set of interacting components by combining the method sequences for each of the interacting components.

The first phase is based on the following paradigm. First data-flow analysis is applied to the methods contained in the CUT. Second, symbolic execution is applied to these methods to derive formal specifications for each method. Third, automated deduction is used to derive sequences of method invocations from the information produced in the previous two steps. The CUT is tested by executing all method sequences identified in this manner.

The second phase considers incremental integration of components according to the following strategy. First it identifies an order of integration that minimizes the need for scaffolding by analyzing dependencies among components. The algorithm for defining the integration order is discussed elsewhere [13]. Next it performs pairwise integration of components according to this integration order.

Consider, for instance, the integration of two components, a client and a server. This integration can follow many strategies for generating test cases. The tradeoff is between the number of test cases generated and the completeness of the test set. At one extreme the test cases for the component group can be the cross-product of the set of method sequences of the two components. Suppose that a method sequence s for the client contains calls to the server. The outcome of these calls is likely to depend on the state of the server. Thus, a sequence s should be repeatedly executed on different server states to detect potential defects. To accomplish this effect, we explicitly invoke each method sequence of the server, which can lead to a distinct server state, before we execute sequence s. In this way, we test all combinations of method sequences of the client and the server.

At the opposite extreme, we only exercise each method sequence s of the client with one method sequence for the server. This strategy is especially appropriate when the server has no other clients than the client under test. In this case, the server is first initialized (e.g., by executing a suitable constructor). Next, each method sequence s of the client is executed once.

3 The Automated Vending Machine Example

Our automated vending machine extends an example originally introduced by Kung et al. to illustrate their technique for generating test cases [12]. In their formulation, the example consists of a class *CoinBox*, which models an automatic vending machine. We believe that this class is quite appealing because it contains a fault that will manifest itself only when an instance of the class is in a particular state and a specific method is invoked. In our version of the example we use component *CoinBox* to illustrate our technique for unit testing. We also define a client component to discuss our technique for component integration.

The code for the *CoinBox* class is listed in Figure 1. Similar to Kung et al. [12], we assume that the vending machine requires at least two coins before a sale can occur. The class *CoinBox* has three

```
class CoinBox {
  unsigned totalQtrs;
  unsigned curQtrs;
  unsigned allowVend;
public:
  CoinBox() {
    totalQtrs = 0;
    allowVend = 0;
    curQtrs = 0;
  }
  void AddQtr() {
    curQtrs = curQtrs + 1;
    if (curQtrs > 1)
      allowVend = 1;
  }
  void ReturnQtrs() {
    curQtrs = 0;
  }
  unsigned isAllowedVend() {
    return allowVend;
  }
  void Vend() {
    if (isAllowedVend()) {
      totalQtrs = totalQtrs + curQtrs;
      curQtrs = 0;
      allowVend = 0;
    }
  }
};
```

Figure 1: Code for component CoinBox.

instance variables, namely *totalQtrs*, *curQtrs*, and *allowVend*. Variable *totalQtrs* keeps track of all coins collected for items sold by the machine. Variable *curQtrs* keeps track of coins entered since the last sale. Variable allowVend determines whether enough coins have been entered for a sale to take place.

This class contains a fault in method *returnQtrs()*, that does not reset variable *allowVend*. The sequence consisting of the invocations *CoinBox(), addQtr(), addQtr(), returnQtrs(),* and *vend()* will result in a successful sale when it should not.

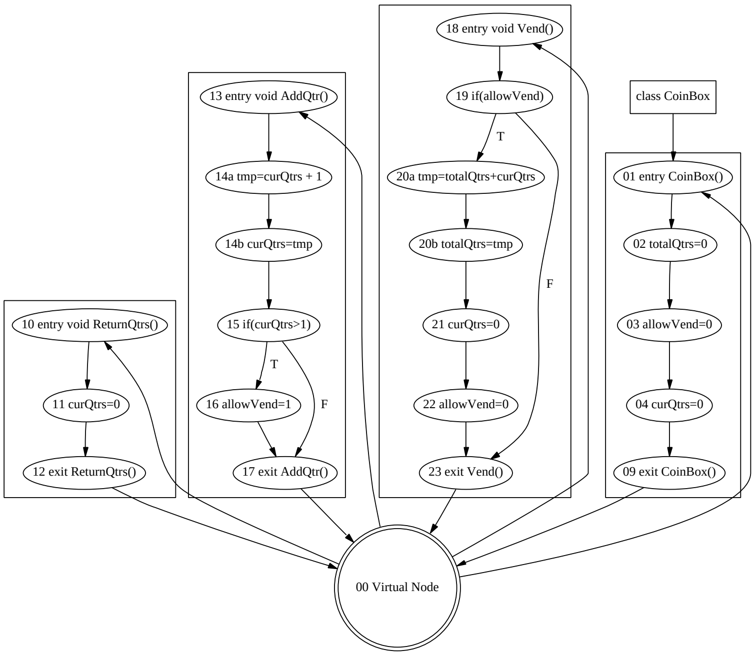

Figure 2: CCFG for class CoinBox.

4 Data-flow analysis

Our data flow analysis identifies a set of so-called du-pairs for the CUT. A du-pair for a variable *v* consists of two statements, s_d and s_u, subject to the following two conditions. First, s_d and s_u are both contained in a component *C*. Second, s_d modifies (i.e., writes) the value of *v* and s_u uses (i.e., reads) *v*'s value. In this case, we denote the methods containing s_d and s_u by m_d and m_u, and we write $\langle s_d, s_u, v\rangle$ to denote the du-pair.

Our data-flow analysis is characterized by two main features. First, we consider all methods in a component together while defining du-pairs. As a result, the two instructions forming a du-pair can belong to different methods. A similar view was taken by Harrold and Rothermel [10]. Second, we consider only instance variables, as opposed to method parameters, for our data-flow analysis. Instance variables retain their values between method executions, thus defining the state of each component instance.

We use a structure called a Class Control Flow Graph or CCFG for our intermethod data-flow analysis. Graph nodes represent single-entry, single-exit regions of executable code. Edges represent possible execution branches between code regions. The CCFG of a component consists of a set of Control Flow Graphs (CFGs), one graph for each method in the CUT, and two additional nodes. Each CFG has an *entry node* and an *exit node*, both labeled with the name of the corresponding method. One of the additional nodes is the entry node for the CUT. It is connected by an out arc with the entry nodes of all the constructors for the CUT. The second node captures the fact the methods of the CUT can be invoked in an arbitrary order by the clients. It has an in arc from the exit node of the CFG of every method and out arc to the entry node of the CFG of every method. The CCFG of the component *CoinBox* is shown in Figure 2.

The CCFG representation allows us to apply standard techniques for data-flow analysis. For the *CoinBox* component, we obtain the du-pairs shown in Table 1. As with traditional data-flow testing, for each du-pair $\langle s_d, s_u, v\rangle$ we try to identify a feasible, def-clear path from s_d to s_u. A def-clear path for du-pair $\langle s_d, s_u, v\rangle$ is a path that does not contain definitions of v after s_d.

#	**variable**	m_d **(node#)**	m_u **(node#)**
01	curQtrs	CoinBox (4)	AddQtr (14a)
02	curQtrs	CoinBox (4)	AddQtr (15)
03	allowVend	CoinBox (3)	Vend (19)
04	totalQtrs	CoinBox (2)	Vend (20a)
05	curQtrs	ReturnQtrs (11)	AddQtr (14a)
06	curQtrs	ReturnQtrs (11)	AddQtr (15)
07	curQtrs	ReturnQtrs (11)	Vend (20a)
08	curQtrs	AddQtr (14b)	AddQtr (14a)
09	curQtrs	AddQtr (14b)	AddQtr (15)
10	curQtrs	AddQtr (14b)	Vend (20a)
11	allowVend	AddQtr (16)	Vend (19)
12	totalQtrs	Vend (20b)	Vend (20a)
13	curQtrs	Vend (21)	AddQtr (14a)
14	curQtrs	Vend (21)	AddQtr (15)
15	curQtrs	Vend (21)	Vend (20a)
16	allowVend	Vend (22)	Vend (19)
17	curQtrs	CoinBox (4)	Vend (20a)

Table 1: du-pairs of *CoinBox* component.

5 Symbolic Execution

We use symbolic execution to capture three relevant aspects of method behavior: (1) the conditions associated with the execution of paths in the method's control flow, (2) the relationship between input and output values of a method, and (3) the set of variables defined along each path. Both kinds of specifications are expressed as a set of propositional formulas involving the values of method parameters and component attributes before and after a method is executed.

The information we extract for a component method consists of a set of formulas in the following form:

$$\begin{array}{lll} < precondition > & \Rightarrow & (< attribute >' = < symbolicexpression >)^* \\ & & < setofdefinedattributes > \end{array}$$

Each *precondition* is a predicate on attributes and method parameters that leads to the execution of a given path traversing the method. The *set of defined attributes* includes all the attributes that are defined along such path. The *symbolic expression* defines the new value of an attribute after a method is executed. This expression involves method parameters and the old values of the attributes.

Due to the inherent complexity of this kind of static analysis techniques, our algorithms for symbolic execution can take advantage of programmer provided information (e.g., loop invariants). Details on these algorithms are described in [8].

For our example, the specifications extracted for component *CoinBox* are shown in Table 2.

CoinBox		
$(true)$	$\Rightarrow$	$totalQtrs' = 0$
def={totalQtrs,curQtrs,allowVend}		$curQtrs' = 0$
		$allowVend' = 0$
AddQtrs		
$(curQtrs > 0)$	$\Rightarrow$	$totalQtrs' = totalQtrs$
def={curQtrs,allowVend}		$curQtrs' = curQtrs + 1$
		$allowVend' = 1$
$(curQtrs == 0)$	$\Rightarrow$	$totalQtrs' = totalQtrs$
def={curQtrs}		$curQtrs' = 1$
Vend		
$(allowVend \neq 0)$	$\Rightarrow$	$totalQtrs' = totalQtrs + curQtrs$
def={totalQtrs,curQtrs,allowVend}		$curQtrs' = 0$
		$allowVend' = 0$
$(allowVend == 0)$	$\Rightarrow$	$allowVend' = 0$
def={}		
ReturnQtrs		
$(true)$	$\Rightarrow$	$totalQtrs' = totalQtrs$
def={curQtrs}		$curQtrs' = 0$

Table 2: Information extracted for component *CoinBox*.

6 Sequence Generation Technique for Unit Testing

We use the specifications obtained through symbolic execution to derive sequences of method invocations that cover the du-pairs identified by data-flow analysis. Each test case exercises one or more du-pairs with a sequence of method invocations that starts with a constructor, and contains a du-path for each of the du-pairs exercised.

Our technique is based on backward-chained deductions, starting from method m_u and the path condition $P(s_u)$ associated with the execution of statement s_u in m_u. Next, we match $P(s_u)$ with the postconditions of all methods in the CUT, in an effort to find a method subsequence that makes $P(s_u)$ true. In general, we will find one or more methods that satisfy this condition. Therefore, we incrementally build a tree of method invocations.

The root of the tree is a node n_u that corresponds to method m_u and condition $P(s_u)$. In general, given a node n_i corresponding to a method m_i, the children of n_i represent methods whose execution does not bring the component in a state that is in contradiction with the path condition of n_i. Consider a child n_j of n_i. The postconditions of the method, m_j, corresponding to n_j must not contradict the path condition of n_i. The path condition of n_j is built from the path condition of n_i by removing the clauses satisfied by the postconditions of m_j and by adding the preconditions on the execution of m_j.

The process of tree construction can terminate in three possible ways:

1. No nodes can be added to the tree before a feasible du-path is found. In this case, the du-pair is deemed infeasible.

2. The depth of the tree reaches a given threshold, without finding a feasible du-path. In this case, our analysis is inconclusive and we report this fact to the user of our framework.

3. A feasible du-path is found. This happens when (1) a leaf node in the tree corresponds to a constructor, (2) the path from the root to the leaf contains a node n_d that corresponds to the execution of statement s_d by method m_d, and (3) the path contains no nodes in which variable *v* is defined between n_d and the root.

   In this case, the path from the root to the leaf represents the reverse of the sequence of method invocations that exercises the given du-pair.

A complete description of our algorithm can be found in [7]. Here we report on the tree construction activity for one of the du-pairs in the *CoinBox* component. We chose du-pair 7 from Table 1 because it is a non-trivial case, and it reveals the fault in the component.

variable:	$curQtrs$
definition:	$ReturnQtrs$, node 11
use:	$Vend$, node 20a
$P(s_d)$:	$true$
$P(s_u)$:	$allowVend \neq 0$

To generate an invocation sequence that covers this du-pair, we start from the root of the tree, represented by a node for method $Vend$. The path condition $P(n_u)$ of the root node is $(allowVend \neq 0)$ because in this case the execution of s_u is governed by an *if* statement with that condition. The only method that does not contradict the path condition and does not define $curQtrs$ is $ReturnQtrs()$, i.e. m_d. After this node has been added, methods that define $curQtrs$ can be added. By further applying our automated deductions, we obtain the tree shown in Figure 3. The resulting tree contains a leaf node satisfying the conditions on a feasible du-path. The resulting method sequence is:

$$\boxed{CoinBox(),\ AddQtr(),\ AddQtr(),\ ReturnQtrs(),\ Vend()}$$

Note that this sequence shows the erroneous behavior of component $CoinBox$ because an item is sold but no coins are collected by the machine.

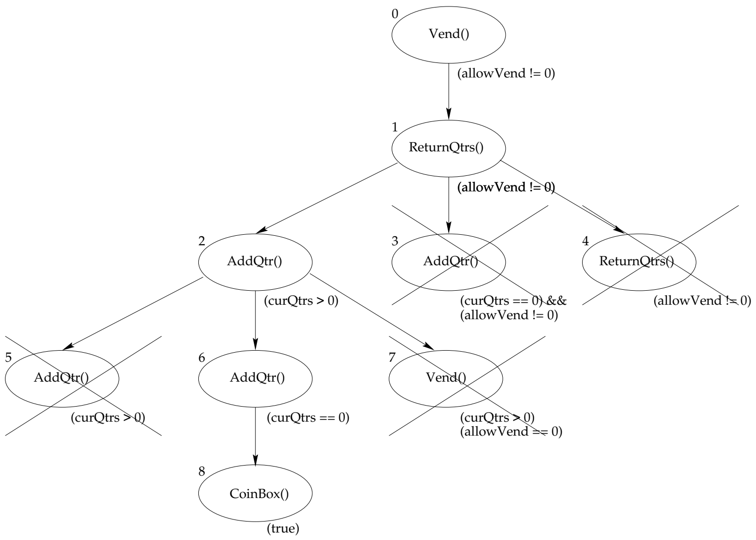

Figure 3: Tree for du-pair 7.

7 Sequence Generation for Integration Testing

To illustrate our technique for integration testing we enrich the system with an additional component $Drinker$, which uses services provided by the $CoinBox$ component. In this case, we play the role of

```
class Drinker {
 public:
  Drinker();
  void getThirsty();
  void earn(int coins);
  void drink();
}
```

Figure 4: Interface for component Drinker.

the user, who obtained the compiled component along with its specifications and the set of method sequences exercising it. Therefore, in Figure 4 we only provide the interface of component $Drinker$ in a class declaration fashion.

In this case, the identification of the order of integration is trivial. We only have two components in client-server relationship.

Table 3 shows a subset of the message sequences exercising the component. Messages between square brackets represents indirect method invocations, i.e., messages sent by the client component to the server component. Those invocations occur within methods of the client.

1	Drinker(), drink()
2	Drinker(), earn(3), drink(), [addQtr()], [addQtr()], [returnQtrs()], getThirsty(), drink(), [addQtr()], [addQtr()], [Vend()]
3	Drinker(), getThirsty(), earn(5), drink(), [addQtr()], [addQtr()], [Vend()], drink(), [addQtr()], [addQtr()], [Vend()], getThirsty()
4	...

Table 3: Subset of sequences for component $Drinker$.

We integrate the components by following both approaches presented in Section 2. In the rest of this section, we illustrate the sequences that are built during integration by suitably combining the sequence identified in Section 6 for component $CoinBox$ and the sequences shown in Table 3. Only a subset of sequences are considered, in order to provide the reader with the essentials of the technique.

The first approach requires integration testing to be performed by simply trying to reproduce the invocation sequences for component $Drinker$ involving component $CoinBox$ on the subsystem composed of the two components. We consider the subsystem as correct if

- indirect invocations take place as foreseen during unit testing,
- the outcomes of the invocation sequences are correct.

If the subsystem is behaving correctly, we can consider it as verified and proceed with the integration. In further steps of integration, the two components $CoinBox$ and $Drinker$ have to be considered as a single component, whose interface simply consists in the interface of component $Drinker$. The resulting invocation sequences are:

1	$CoinBox()$, $Drinker()$, $earn(3)$, $drink(), [addQtr()]$, $[addQtr()]$, $[returnQtrs()]$, $getThirsty()$, $drink(), [addQtr()], [addQtr()], [Vend()]$
2	$CoinBox()$, $Drinker()$, $getThirsty(), earn(5)$, $drink()$, $[addQtr()]$, $[addQtr()]$, $[Vend()]$, $drink(), [addQtr()], [addQtr()], [Vend()], getThirsty()$

The second approach requires the construction of invocation sequences by composition of the sets of sequences exercising the client and the server, in order to exercise the client sequences for several possible states of the server. Again, we consider the subsystem as correct if

- indirect invocations take place as foreseen during unit testing,
- the outcomes of the invocation sequences are correct.

If the subsystem is behaving correctly, we can consider it as verified and proceed with the integration. In further steps of integration, the two components $CoinBox$ and $Drinker$ have to be considered as a single component, whose interface consists in the union of the interfaces of both components. The resulting invocation sequences are:

1	$CoinBox()$, $Drinker()$, $earn(3)$, $drink()$, $[addQtr()]$, $[addQtr()]$, $[returnQtrs()]$, $getThirsty()$, $drink()$, $[addQtr()]$, $[addQtr()]$, $[Vend()]$
2	$CoinBox()$, $AddQtr()$, $Drinker()$, $earn(3)$, $drink()$, $[addQtr()]$, $[addQtr()]$, $[returnQtrs()]$, $getThirsty()$, $drink()$, $[addQtr()]$, $[addQtr()]$, $[Vend()]$
3	$CoinBox()$, $AddQtr()$, $AddQtr()$, $Drinker()$, $earn(3)$, $drink()$, $[addQtr()]$, $[addQtr()]$, $[returnQtrs()]$, $getThirsty()$, $drink()$, $[addQtr()]$, $[addQtr()]$, $[Vend()]$
4	$CoinBox()$, $AddQtr()$, $AddQtr()$, $ReturnQtrs()$, $Vend()$, $Drinker()$, $earn(3)$, $drink()$, $[addQtr()]$, $[addQtr()]$, $[returnQtrs()]$, $getThirsty()$, $drink()$, $[addQtr()]$, $[addQtr()]$, $[Vend()]$
5	$CoinBox()$, $Drinker()$, $getThirsty()$, $earn(5)$, $drink()$, $[addQtr()]$, $[addQtr()]$, $[Vend()]$, $drink()$, $[addQtr()]$, $[addQtr()]$, $[Vend()]$, $getThirsty()$
6	$CoinBox()$, $AddQtr()$, $Drinker()$, $getThirsty()$, $earn(5)$, $drink()$, $[addQtr()]$, $[addQtr()]$, $[Vend()]$, $drink()$, $[addQtr()]$, $[addQtr()]$, $[Vend()]$, $getThirsty()$
7	$CoinBox()$, $AddQtr()$, $AddQtr()$, $Drinker()$, $getThirsty()$, $earn(5)$, $drink()$, $[addQtr()]$, $[addQtr()]$, $[Vend()]$, $drink()$, $[addQtr()]$, $[addQtr()]$, $[Vend()]$, $getThirsty()$
8	$CoinBox()$, $AddQtr()$, $AddQtr()$, $ReturnQtrs()$, $Vend()$, $Drinker()$, $getThirsty()$, $earn(5)$, $drink()$, $[addQtr()]$, $[addQtr()]$, $[Vend()]$, $drink()$, $[addQtr()]$, $[addQtr()]$, $[Vend()]$, $getThirsty()$

A complete description of the technique, along with a fully developed example, can be found in [7].

8 Conclusions

This paper presents an approach for the automatic generation of test cases for unit and integration testing of software components. The preliminary results of our investigations are quite promising. First, our approach seems to be quite powerful in that the test cases we generate automatically can detect subtle errors such as the ones contained the automated vending machine example. In addition, our approach lends itself to the analysis of multiple integrated components. Our future investigation aims at strengthening the techniques for symbolic execution and automated deduction. An implementation of our approach is currently under way.

References

[1] *The Common Object Request Broker: Architecture and Specification*. Object Management Group, July 1996.

[2] S. Barbey, D. Buchs, and C. Péraire. A theory of specification-based testing for object-oriented software. In *Proceedings of EDCC2 (European Dependable Computing Conference), Taormina, Italy, October 1996*, LNCS (Lecture Notes in Computer Science) 1150, pages 303–320. Springer-Verlag, 1996.

[3] Javabeans documentation. http://java.sun.com/beans/docs/index.html.

[4] A. W. Brown and K. C. Wallnau. Enginnering of component-based systems. In A. W. Brown, editor, *Component-Based Software Engineering*, pages 7–15. IEEE Press, 1996.

[5] N. Brown and C. Kindel. *Distributed Component Object Model protocol: DCOM/1.0*. Jan. 1998.

[6] M. Büchi and W. Weck. A plea for grey-box components. techreport 122, Turku Centre for Computer Science, Turku, Finland, Sep 1997.

[7] U. Buy, A. Orso, and M. Valsasna. A framework for testing object-oriented components. Technical report, Politecnico di Milano, Milan, Italy, December 1998.

[8] A. Coen-Porisini, F. De Paoli, C. Ghezzi, and D. Mandrioli. Software specification via symbolic execution. *IEEE Trans. Software Engineering*, SE-17(9):884–899, Sept. 1991.

[9] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, Apr. 1994.

[10] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. In *2nd ACM-SIGSOFT Symposium on the foundations of software engineering*, pages 154–163. ACM-SIGSOFT, December 1994.

[11] P. Jorgensen and C. Erickson. Object-oriented integration testing. *Communications of the ACM*, 37(9):30–38, September 1994.

[12] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, Y.-S. Kim, and Y.-K. Song. Developing and oject-oriented software testing and maintenance environment. *Communications of the ACM*, 38(10):75–86, Oct. 1995.

[13] A. Orso. *Integration Testing of Object-Oriented Software*. PhD thesis, Politecnico di Milano, Milano, Italy, 1998.

[14] C. Pfister and C. Szyperski. Why objects are not enough. In *Proceedings, First International Component Users Conference (CUC'96)*, Munich, Germany, jul 1996.

[15] C. Szyperski. *Component Oriented Programming*. Addison-Wesley, first edition, 1997. The book is expected to be out in November 97.