Automated support for state based testing

Ilona Bluemke
Institute of Computer Science
Warsaw University of Technology
Nowowiejska 15/17
00-665 Warszawa, Poland
e-mail: I.Bluemke@ii.pw.edu.pl

Abstract. In this paper the object-oriented software testing environment (TOSTER) is presented. This environment supports the state based testing of object programs. TOSTER extracts information from two commercial CASE tools – Paradigm Plus and Select Enterprise and supports test generation for C++ and Java classes. In TOSTER the state vector testing method is used. The state vector method and its implementation is briefly described. A simple example of testing in this environment is also given.

Keywords: object oriented testing, CASE

1. Introduction

Currently produced software is very complex and has to be of good quality. There are many software firms at the market so the software cost should be low and the time of its production should be as short as possible. Since the 1980s a wide range of tools supporting software development have been provided. CASE (Computer Aided Software Engineering) technology can be applied in a cost-effective way to reduce software costs and development time significantly, as well as to increase software quality. CASE workbenches are available to support most activities in the software development process. As there are two main approaches to the software development: functional (structural) and object-oriented, there are also two kinds of CASE tools: structural and object-oriented. In this paper we will concentrate on object-oriented testing CASE tools. Workbenches to support analysis and design are widely used in industry. They can support a specific design and analysis method or constitute more general diagram editing systems augmented with knowledge of the most common methods. Representative of this kind of workbenches is Paradigm Plus supporting many object-oriented analysis and design methods (UML [3], OMT, Booch, Coad-Yourdon, Shlaer-Mellor, Fussion) or Select Enterprice and Rational Rose supporting only UML.

For some kinds of software systems the effort to test software is half or even more of the total effort to develop software. Any activities decreasing the testing time give financial effects decreasing the cost and facilitate the software development process. Many automated testing tools has been developed, some of them are briefly described in section 2. The automated software testing tools, that are able to
work with the object-oriented CASE tool are rare. To our best knowledge only Rational Test Studio is able to use information prepared during analysis and design with CASE- Rational Rose.

In the Institute of Computer Science, Warsaw University of Technology, an environment supporting object-oriented testing has been designed and implemented. The object-oriented software testing environment (TOSTER) is able to extract information from two commercial CASE tools repositories – Paradigm and Select. The unit being tested in TOSTER is C++ or Java class. Currently in TOSTER two class testing approaches are implemented:

- sequence method proposed by S. Kirami and W.T. Tsai in [2, 4]
- state-vector method introduced by C.D. Turner and D.J. Robson in [2, 6].

However state based testing approach is well known, to our best knowledge, only TOSTER and Tootsie (Total Object Oriented Testing Support Environment) described in [1] support this approach in testing object programs. In section 3 the main features of TOSTER are described. The implementation of state vector method is described in section 4 and a simple example is presented in section 5.

2. Object oriented testing and tools

In testing software two main approaches can be identified [5]:

- functional testing – the unit under test is treated as “black box”, the unit’s functions are tested
- structural testing - the internal structure of the unit under test is known, unit is seen as a “white box”, and the tests are prepared to check some structure properties eg. statement coverage.

Structural testing is important in the testing process, but it should not be the primary basis for testing. Even if a code based test suite achieves high coverage it cannot prove the absence of faults, validate requirements, or test missing functions. An implementation-based test can only show that the code does what it does, but it is not necessarily what it is supposed to do (this could be checked by functional testing). Code-based test generation can be automated. Some testing tools are briefly described below. These tools are dedicated to object programming language C++ or Java. The well known and used commercial testing tools are:

  a) WhiteBox
  b) ParaSoft Jtest
  c) Panorama++
  d) Rational Suite TestStudio.

Ad a)

WhiteBox is a product of Reliable Software Technologies (www.rstcorp.com). This testing tool is integrated with MS Visual Studio and can support coverage testing of C++ and JAVA programs. Following types of code coverage can be tested:

- Branch (BC) – for each branch point checks if all branches can be reached
- Condition Decision (CD) - for each two branch points, like if and while, checks if all branches can be reached
- Multiple Condition (MCC) – used to test logical expressions
- Function (FC) – checking if all functions have been executed.

In WhiteBox some metrics, useful in the prediction of testing complexity can be calculated. For example: number of attributes (private, protected, public), number
of methods, depth of inheritance, number of inheritance hierarchies. The tested program has to be compiled with a special option, enabling the generation of a test report during the execution of program. The test report has to be viewed with WhiteBox viewer.

Ad b) Jtest can support the test of classes, applets or even application written in Java. In Jtest black-box and white-box test can be executed. A part of Jtest is CodeWizard, which can be used to statically analyse the code. CodeWizard contains some rules checking properties of the source code (e.g. for statement with empty body). User can add his own rules to be checked. In Jtest some metrics can also be calculated (similar to metrics in WhiteBox). Jtest for C++ programs is also available.

Ad c) Panorama++ (www.softwareautomation.com) supports testing of C++ programs. Automatic tests are not generated but some analysis during the execution of a program is made. Frequency of instruction’s execution can be given, cyclomatic complexity and other metrics are calculated and function dependency graphs can be seen. Control flow of the program can also be observed.

Ad d) Rational Suite TestStudio contains several programs supporting different testing domains eg. software planning, managing and monitoring, performing functional, efficiency and coverage test. TestStudio enables simulation of many users (Virtual Users component) and measuring response time of the program under different load (in GUI Users component). Purify, Quantify and PureCoverage programs enable some quality checks of the system under test. Rational Purify is used to find runtime errors and memory leaks. Program can be integrated with Visual C++ environment or can work independently. Rational Visual Quantify program can be used to find parts of the tested application that are executed slowly. PureCoverage program is developed to find different kinds of code coverage in the tested program.

An object oriented system can be viewed as a set of cooperating agents. Each agent is responsible for its state. System behaviour is the result of interacted individual behaviours. To develop and deliver trustworthy object-oriented systems a high level of confidence is needed. That means: each component will behave correctly, collective behaviour is correct and no incorrect collective behaviour will be produced. Although object-oriented design and programming support many kinds of fault prevention, testing is necessary. An effective approach for testing must enable effective tests of components and collections of components. Object-oriented testing is well suited to state based testing because:

1. In many object-oriented analysis and design methods eg. UML [3], the behaviour of a class is modelled by finite state machines. Methods of a class must be used sequentially. Some method’s sequences may be prohibited by specification or may cause the implementation to fail.

2. Model used for testing must help to find faults. State control is typically distributed over an entire system. Individual and collective behaviour faults are likely as a result of this complex implicit structure. State based testing provides a straightforward means to develop test suites that will reveal these faults.

State machines may be used to model the behaviour at any scope: class, cluster, component, subsystem or system. Details must decrease when state machines are developed for larger scope. The packaging of instance variables and methods into a class is fundamental to object-oriented programming. Although the number of
message sequences and instance variable value combinations is infinite a state machine can nevertheless provide a compact and predictable model of behaviour.

3. TOSTER

TOSTER, the object-oriented software testing environment, has been designed and implemented in the Institute of Computer Science, Warsaw University of Technology. Currently in TOSTER two class testing approaches are implemented:

1. sequence method proposed by S. Kirami and W.T. Tsai in [2, 4]
2. state vector method introduced by C.D. Turner and D.J. Robson in [2, 6].

State transition models are present in all object-oriented design methods. They enable us to design the dynamic behaviour of objects. A lot of information like class operations and state transitions, necessary for the operation of the Turner-Robson method, is collected during the design phase. This information is available on the class diagram and state transition diagrams and is kept in the CASE repository. TOSTER is able to extract information from two commercially available CASE tools repositories – Paradigm and Select. Paradigm as well as some other CASE tools (eg. Select), provide a language which enables access to almost all information kept in the repository and is similar to Visual Basics. In this language there are control structures enabling extracting information from diagrams, all or specific one, like a class diagram with a specified name. The script language provides mechanisms to get the class attribute, operation and state information as well. A script in the Paradigm script language, and a program in Visual Basics to extract information from Select repository, has been written, transforming information from the CASE repository into XML file. The XML file is read by TOSTER and prepares internal information for the testing process. The architecture of TOSTER is shown in figure1.

![TOSTER Architecture Diagram](image)

The unit being tested in TOSTER is C++ or Java class. In the testing process in TOSTER following phases can be indicated:

1. extraction of information from the CASE repository (automatically),
2. indicating and mapping source files of tested program (automatically, user)
3. adding information, specific for the testing method (user)
4. automated modifications in the source files performed by TOSTER
5. compilation of the program
6. preparation of test cases (user)
7. program executions (in TOSTER environment)
8. test results presentation and savings (TOSTER).

Model of software prepared in a CASE tool is automatically transferred into TOSTER environment through XML file. Sometimes this model should be refined.

Testing process based on state transition diagrams needs detailed information on state transitions diagrams describing the behaviour of an object. Some additional information, dependent on testing method, is also necessary. For state vector method, user should add attribute values and state vectors. User is also responsible for preparing the test cases. The executions of program are performed in TOSTER environment. TOSTER modifies program source files. The modifications depend on the testing method. For state vector method TOSTER modifications are following:

1. New method evaluating the state of an object is added,
2. New class, nested in the class under test, is added. In this class constructor and destructor a method evaluating the state of the tested object is called.
3. At the beginning of each method of the tested class, which exists on the transition in the state transition diagram, a local object of the new class (ad. 2) is added. This object is used to save the state of the object under test, before and after the method call.
4. The class code is extended with mechanism allowing for the object identification.

During the test execution the results are written to a file. This file is analysed by testing method. The results are compared with the test model kept in TOSTER. In the case of inconsistency an error message is generated.

The specified above actions are typical for a test supporting CASE tool. TOSTER is implemented in Java. The graphical user interface is based on SWING library. A testing method is implemented as a Java class.

### 4. State based testing method

The state based testing method (state vector method) was introduced by C.D. Turner and D.J. Robson in [6]. The main goal of this method is to check, if the behaviour of the implemented class is in accordance with its state transition model. State transition models are present in all object-oriented design methods and are used to design the dynamic behaviour of objects. The state transition diagram consists of nodes representing states and edges showing the possible state transition caused by stimuli (e.g., an operation call). Depending on the result of an operation call the following types of operation can be given:

1. changing object’s state to a new one – according to the transition on the diagram to a new state
2. remaining object’s state unchanged - according to the loop transition on the diagram (the same input and output state)
3. inappropriate for current state call – there is no transition from the current state mark with this operation
4. changing object’s state to an undefined one, erroneous – there are no nodes and edges on state transition diagram
5. changing object’s state to a defined one, but inappropriate - there is no such edge on state transition diagram
6. leaving object’s state unchanged when it is supposed to change.

The state-based testing method assigns each operation call to one of the above groups, so one of the following defects in operation calls could be found:

- inappropriate operation for current state,
- operation forcing an object to change its state to an undefined one,
- or to an inappropriate one.

The first defect is caused by inappropriate use of an object while the second and the third defect, could be the effect of errors in the operation implementation.

The object’s state is determined by values of its data members – attributes and the current point in time. Attributes are used to store data and control information (used to trigger events). The more control attributes are in a class, the more complicated is the interaction between operations. This affects the time required to design and generate state-based tests. Attribute values can be divided into three groups:

1. specific values – described in the design as elements of special significance (example: for integer attribute 5 is a specific value)
2. general values – a group of attribute values considered in the same manner, a set of values (example: for integer attribute values greater than 5 constitute general value)
3. dynamic values – an attribute value will be established during program execution.

The state of object is the combined value of all its attributes at the current point in time.

The example of a linked list written in C++ is from [6]:

```cpp
class list
{
  public:
    // class interface
  protected:
    struct list_element {
      list_element *pNext;
      TYPE tItem;
    };
    // pTop - pointer to the first element of list
    list_element *pTop;
    // pCur - pointer to the current list element
    list_element **pCur;
};
```

Attribute pTop points to the first element in the linked list or is equals NULL if there is no list. Attribute pCur points to a pointer to the current element in the list. For the above shown class the attributes are of following types:

A) pTop attribute
- 1. pTop_1 - specific value equals NULL
- 2. pTop_2 - is not equal to NULL (general value)

B) pCur attribute
- 1. pCur_1 - specific value equals NULL
- 2. pCur_2 - dynamic value *pCur == pTop
- 3. pCur_3 - dynamic value *pCur != pTop
For the class list two cases can be identified: empty list and list containing at least one element. The empty list is defined by vector of attribute’s values \(<p_{\text{Top}_1}, p_{\text{Cur}_2}\). The second case is defined by two attribute vectors: \(<p_{\text{Top}_2}, p_{\text{Cur}_2}\> \text{ and } <p_{\text{Top}_2}, p_{\text{Cur}_3}\>. The first vector defines the object of the class list with one element, the second with more list elements. If it is sufficient to define a case “list containing at least one element”, for \(p_{\text{Cur}}\) a general value \(p_{\text{Cur}_4}\) - not equal to NULL should be specified and this case is defined by vector \(<p_{\text{Top}_2}, p_{\text{Cur}_4}\>. Other combinations of attribute’s values can be considered as errors in the object’s state.

The next part of the state-based testing is to determine for each method:

- input state (state when the method is called) and
- output state (state when the method is finished).

These states are created on the basis of the class design not the class code (as there may be errors in the code). In TOSTER these states are created from state transition diagram. Exceptions are constructor and destructor. The constructor is called, when the object is in an undefined state, before it has been initialised. The destructor leaves the object in an undefined state and accepts all valid states as input. The states for constructor and destructor are not specified by vector of attribute values. In many programming languages (eg. C++) attributes values in these states can be undetermined.

To enable state-based testing some modifications in the class code are necessary. To the class code a controller is added. The controller watches state changes and determines an object’s current state. The method code is enlarged with some instructions allowing state determination before and after the method execution. It is also necessary to determine if the change in state is correct and to report information about tested situation.

**4.1 State-based testing process**

State-based testing process consists of following steps:

1. For each attribute define its values
2. For each state define vector of attributes values
3. For each method define initial and final states
4. Generate code causing the object to be in the initial state. Check if the state is correct.
5. Generate sequences of methods calls
6. Generate code causing the object to be in the final state. Check if the state is correct.
7. Generate code necessary to finish the test
8. Run the program
9. Analyse test results.

State-based testing can be used in testing interactions between different classes. This involves creation of states, not only for the main object under test, but also for the objects that will be passed as parameters. Integration testing of two objects A and B comprises following steps:

1. create object A, move it to state in which the interactions with B take place
2. create object B, move it to state in which the interactions with A take place
3. call the method of interaction (eg, the parameter of this method is B object)
4. validate resultant state of object B
5. validate resultant state of object A.
In the implementation of state-based testing method some modifications and constraints of the original Turner-Robson method were introduced. A new value type of attribute was added – the dynamic value. The arguments and results of methods are not analysed.

### 4.2 Method assessment

State-based testing is useful for more than simply detecting the change in state of an object, it can be used to detect the correct construction of a more complex dynamic data structure (for example a linked list). State vector testing has particular classes on which it is more effective than on others. If a class is designed simply as a repository of information or as non-dynamic data structure then this method will have limited effect. For dynamic data structures it is essential to determine when and which particular changes can occur to the structure. For dynamic classes state-based testing enables the detection of errors which are difficult to detect using other testing methods. The technique is not a substitute for functional testing, nor structural one. It should be considered as a complimentary technique for testing interactions with an object’s state. These interactions are essential in the object-oriented method. If a class is written with no order imposed upon the execution of operations, the less restrictive it is upon any use of it by a programmer. The state vector method can be adapted to test the class interactions as well.

### 5. Example

As an example of state based testing in TOSTER simple class Stack will be used. This class provides methods to push/ pop an integer value on/from the stack. During creation of a stack object, the initial size is given. In figure 1 the model of the class Stack is shown.

```
«business»
::Stack
  size
  top
  pStack
  Stack
  ~Stack
  Push
  Pop
  Resize
```

Fig. 2. Select model of class Stack

The declaration of the class Stack in C++ is given below:

```c++
#define STACK_SIZE_INCREASE 10
class Stack {
    protected:
    int size;
```
int top;
int top;
int * pStack;

public:
    Stack(int initial_size);
    ~Stack();
    void Push(const int value);
    int Pop();

protected:
    void Resize(const int new_size);
};

Attribute pStack points to an array of integer elements, the size of this array is in size attribute. Attribute top contains the index of the first free element in table pStack. The method Resize is changing the size of table containing stack elements. In figure 2, a hierarchical state transition diagram describing the behaviour of Stack class is shown.

![State transition diagram for class stack](image)

Fig. 3. State transition diagram for class stack

An object of the class Stack can be in one of the following states:
- **initial** - creating an object, constructor not called yet
- **initialised** - object is initialised. Initialisation is made by constructor. This state comprises states: **empty**, **full** and **not full**.
  - **empty** – stack with no elements on it
  - **full** – object has filled all elements on the stack. Pushing of a new element will be preceded by increasing, resizing the stack.
  - **not full** – there are free stack elements
- **final** – final state of object

According to the rules of state based testing types of attribute’s values and state vectors for each state must be determined. In table 1 types of values and in table 2 state vectors for class Stack are given.
<table>
<thead>
<tr>
<th>ATTRIBUTUE</th>
<th>Value name</th>
<th>Value Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>initialized</td>
<td>general</td>
<td>size&gt;0</td>
</tr>
<tr>
<td>top</td>
<td>correct</td>
<td>general</td>
<td>top&gt;=0</td>
</tr>
<tr>
<td></td>
<td>value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>empty stack</td>
<td>specific</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>full stack</td>
<td>dynamic</td>
<td></td>
<td>top==size</td>
</tr>
<tr>
<td>not full stack</td>
<td>dynamic</td>
<td></td>
<td>top&gt;0 &amp;&amp; top&lt;size</td>
</tr>
<tr>
<td>pStack</td>
<td>initialized</td>
<td>general</td>
<td>pStack!=NULL</td>
</tr>
</tbody>
</table>

Table 1. Attribute value types

For each state determined state vectors are shown in table 2.

<table>
<thead>
<tr>
<th>State</th>
<th>State vector (attribute [value name])</th>
</tr>
</thead>
</table>
| initial   | size[initialised]
| initialise| top[correct value]
| d         | pStack[initialised]
| empty     | size[initialised]
|           | top[empty stack]
|           | pStack[initialised]
| full      | size[initialised]
|           | top[full stack]
|           | pStack[initialised]
| not full  | size[initialised]
|           | top[not full stack]
|           | pStack[initialised]
| final     |                                                     |

Table 2. State vectors

States initial and final are specific and for these states the state vectors are not determined. Object is in initial state only at the moment the constructor is called. State final is the ending state for the destructor. After determination of state vectors the mapping between methods and source files is performed. Next the transitions on state transition diagram are linked with respective methods. At last the test cases are executed.

For the Stack class following test cases were generated:

a) Initial size 5. Method’s sequence: push, pop, push, push, push, push, push, push, push, push, pop, pop, pop, pop, pop, pop, pop, pop, pop, pop. This is correct sequence of methods and the object should reach state initialized/empty (state initialized and substate empty).

b) Method’s sequence: push, push, push, push, push, push, push, pop, pop, pop, pop, pop, pop, pop, pop, pop, pop, pop. This is an incorrect sequence because more times the method pop than push was called. The error should be detected during testing.
For the test case b) test report but in the form of sequence method’s calls written to a text file, is presented below. In brackets are given incorrect calls.

```
Stack ( const int initial_size )
void Push ( const int value )
void Push ( const int value )
void Push ( const int value )
void Push ( const int value )
void Push ( const int value )
void Push ( const int value )
void Push ( const int value )
void Resize ( const int new_size )
int Pop (  )
int Pop (  )
int Pop (  )
int Pop (  )
int Pop (  )
int Pop (  )
[ int Pop (  ) ]
[ int Pop (  ) ]
[ int Pop (  ) ]
[ int Pop (  ) ]
~Stack (  )
```

Incorrect method call means that no transition on state transition diagram was found. For an incorrect method call TOSTER generates also:

```
Method was called in incorrect state.
Before method call object was in states:
Stack/initialized
Stack/initialized/empty

Correct method call:
void Push ( const int value )
~Stack (  )
```

Above text reports generated by TOSTER were given and in appendix 1 screens are shown.

6. Conclusions

Due to space limitations the details of the TOSTER design and implementation are not described here, more are given in [7]. The new testing methods can be easily added by new refinements of the same base class. Currently, TOSTER testing environment extracts information from two object CASE tools - Paradigm Plus and Select Enterprise. An interface with any other CASE tool, providing a language to access all information kept in the repository as Rational Rose, can be added. Now TOSTER supports C++ and Java class testing but could be modified to support class integration testing as well.

State based testing methods are well known but only TOSTER and Tootsie (Total Object Oriented Testing Support Environment) described in [1] support theses methods in testing object programs. The automated software testing tools, able to work with the object-oriented CASE tools, are rare. To our best knowledge only
Rational Test Studio is able to use information prepared during analysis and design with CASE - Rational Rose.

References


Appendix 1

Some TOSTER screens appearing during testing a stack class (presented in section 5) are shown below.