Annals of Software Engineering, Vol.7, Oxford, 1999, pp. 283-296.

On Coping with Real-Time Software Dynamic Inconsistency by Built-in Tests

Yingxu Wang [1] **Graham King** [2] **Dilip Patel** [1] **Shushma Patel** [1] **Alec Dorling** [3]

[1] School of Computing, Information Systems and Mathematics, South Bank University, London
103 Borough Road, London SE1 0AA, UK
(yingxu.wang@acm.org, dilip@sbu.ac.uk, shushma@sbu.ac.uk)

[2] Research Centre for Systems Engineering, Southampton Institute
Southampton SO14 0YN, UK
(graham.king@solent.ac.uk)

[3] IVF Centre for Software Engineering
Argongatan 30, S-431 53, Molndal, Sweden
(alec.dorling@ivf.se)

Abstract

In real-time systems, dynamic inconsistencies of software are hardly detected, diagnosed and handled. A built-in test (BIT) method is developed to cope with software dynamic inconsistency. BIT is defined as a new kind of software testing which is explicitly described in object-oriented source code as member functions. BITs can be activated at any designed moment at run-time to detect, diagnose and handle software dynamic inconsistencies.

This paper develops a new approach to cope with software dynamic inconsistencies at run-time by BITs. In this paper, the concept of BITs is introduced. The standard structures which incorporate BITs into conventional object-oriented software are analysed. Reuse methodologies for BITs in OO software are developed at object and system levels. A case study is provided for showing how to create BIT and how to inherit and reuse BITs in OO programming. Methods for incorporating BITs into OO software at object, class and system levels are provided. Approach to dynamic inconsistency control by BITs is developed.

Keywords: Software engineering, real-time software, software inconsistency, dynamic inconsistency, built-in test, run-time testing, test reuse, OOP

1. Introduction

Software inconsistency can occur in any phases of software development and in run-time application. Avoidance of static inconsistencies in software design, implementation and maintenance, as well as protection of dynamic inconsistencies of software at run-time, are important topics in real-time software engineering research and in the software industry.

1.1 Classification of software inconsistency

Software inconsistency can be clarified by its causes [Wang 1988]. The authors have identified two categories of software inconsistency which are both static and dynamic inconsistencies. The former consists of design, implementation and maintenance inconsistencies. The latter contains run-time, decay, random, environment, configuration inconsistencies. As preparation for accurate discussion in this paper, eight types of software inconsistency are formally defined below:

Definition 1. Design inconsistency is defined as differences between user requirements and software specification.

Design inconsistency is caused by design errors.

Definition 2. Implementation inconsistency is defined as differences between software specification and implementation.

Implementation inconsistency is caused by implementation errors.

Definition 3. Maintenance inconsistency is defined as differences between system requirements and the changed implementation in corrective, adaptive, perfective, preventive and reengineering maintenance.

Maintenance inconsistency is caused by the above five-type maintenance activities. The first four causes were classified in [McDermid 1991]; and the reengineering maintenance cause was identified in [Wang *et al* 1999].

Definition 4. Run-time inconsistency is defined as permanent differences between specified dynamic behaviours and real run-time behaviours.

Run-time inconsistency is caused by run-time errors, such as hardware faults, interference, and bugs.

Definition 5. Decay inconsistency is defined as differences between original code and current code in system memory or storage.

Decay inconsistency is caused by code corruption.

Definition 6. Random inconsistency is defined as temporary differences between specified dynamic behaviours and real run-time behaviours.

Random inconsistency is caused by random errors, run-time resources and external interference.

Definition 7. Environment inconsistency is defined as differences of system behaviours under specified and real operation environments.

Environment inconsistency is caused by environment incompatibility such as those of target machine, operating platform, networking and run-time resources.

Definition 8. Configuration inconsistency is defined as differences of system behaviours under specified and real setups.

Configuration inconsistency is caused by configuration errors or incompletion.

Definitions 1 – 8 describe a list of independent software inconsistencies according to their causes. This classification provides a foundation for seeking countermeasures for coping with these inconsistencies.

Based on the above classification and definition, a model of software static and dynamic inconsistencies is developed in Table 1. In Table 1, detailed causes of inconsistencies are analysed and summarized.

Table 1. A model of software inconsistency.

No.	Category of inconsistency	Type of inconsistency	Causes
	Static		
1		Design	Design error: in requirement elicitation, system specification, software specification
2		Implementation	Implementation error: in detailed design, coding, integration, testing, configuration, reuse
3		Maintenance	Maintenance error: in correction, adaptation, perfection, prevention, reengineering
	Dynamic		
4		Run-time	Run-time errors: eg. CPU, memory, stack, system bus, I/O, peripheral device, page control, system deadlock, virus infection, interference, bugs, etc.
5		Decay	Code corruption: eg. Virus, interference, CPU fault, RAM fault, storage fault, etc.
6		Random	Random error: eg. Bugs, non-testable path, dynamic memory management, communication, networking, exceptions, virus, interference, etc.
7		Environment	Environment incompatible: eg. CPU, OS, system platform, run-time resources, network gate, protocols, version, updating, etc.
8		Configuration	Configuration error: eg. set-up options, code, library, interface, version, system platform, run-time resources, etc.

1.2 Problems in dynamic inconsistency handling

A summary of major measures for protecting static and dynamic inconsistencies is listed in Table 2. As shown in Table 2, the conventional techniques for static inconsistency control are mainly formal methods, prototyping and testing; and for dynamic inconsistencies are by inactive exception handling or system reset. The method of built-in test (BIT) has been recently developed by the authors [Wang *et al* 1998, 1997a, b, 1996, 1995]. BIT is defined as a new kind of software testing which is explicitly described in software source code as member functions.

Table 2. Inconsistency protection measures.

No.	Type of inconsistency	Category of inconsistency	Protection measures
	Static		
1		Design	Requirement engineering, formal methods, prototyping, review
2		Implementation	Testing, formal methods, field test, acceptance test, BIT
3		Maintenance	Testing, BIT
	Dynamic		
4		Run-time	BIT, exception handling
5		Decay	BIT, exception handling
6		Random	BIT, exception handling
7		Environment	BIT, adaptive testing, exception handling
8		Configure	BIT, post set-up testing, exception handling

Observing Tables 1 and 2, a large proportion of software inconsistency is identified as dynamic inconsistency, such as the run-time, decay, random, environment and configuration inconsistencies. The static inconsistencies have been relatively thoroughly studied in mathematical logics [Hatcher 1982], formal methods [Harry 1996; Cohen 1986; McDermid 1991], database systems [Ullman 1987], and etc. However the dynamic inconsistencies are hardly to be detected, allocated and handled with difficulty, because they can only be realized at run-time.

This paper develops a new approach to cope with the dynamic inconsistencies at run-time by BITs. Methods for incorporating BITs into OO software at object, class and system levels are provided in Section 2. Methods for dynamic inconsistency control by BITs are developed in Section 3.

2. Built-in test in software

Conventional testing of software is generally application-specific and rarely reusable, especially for a purchased software module or package. Even within a software development organization, software and tests are developed by different teams and are described in separate documents. This convention makes test reuse particularly difficult. The testing of conventional object-oriented software focuses on the generation of tests for existing objects and systems [Freedman 1991; Jorgensen *et al* 1994; Murphy *et al* 1994; Voas *et al* 1995]. The BIT method draws attention to building tests into objects and systems during design and coding, so that software testing can be self-contained. The most interesting feature of the BIT method is that tests can be inherited and reused in the same way as that of code in conventional OO software.

2.1 Built-in test at object level

A structural prototype of an object in OOP is described in Fig.1. An object consists of two structural parts: the interface and the implementation. The interface of an object is the only means of external access to the member functions contained in the object. The implementation of an object is the description of codes for all member functions.

```
Class class-name {
        // interface
        data declaration;
        constructor declaration;
        destructor declaration;
        function declarations;

        // implementation
        constructor;
        destructor;
        functions;
      } [object-name-list];
```

Figure 1. A structural prototype of an object.

An object is reusable because of its natural encapsulation and inheritability. For creating BITs, it is noteworthy that the standard constructor and destructor contained in an object are an interesting reusable structure. The authors have found that these standard structures can be extended further, to be able to contain reusable BITs in an object [Wang *et al* 1995].

A prototype of a built-in test object is developed as shown in Fig.2. Based on the conventional object structure described in Fig.1, the test declarations in the interface and the test cases in the implementation are embedded into the standard object. In this way, the BITs may be inherited and reused in the same way as that of standard and application-specific member functions within the object.

It is proposed that BITs should be a standard component in built-in test object structures. The BITs have the same syntactical functions as that of the standard constructor and destructor in an object. Thus BITs may well fit into an object via C++ or any other OO language compilers.

```
Class class-name {
        // interface
          Data declaration;
          Constructor declaration;
          Destructor declaration;
```

```
   Function declarations;
   Tests declaration;        // Built-in test declarations

 // implementation
   Constructor;
   Destructor;
   Functions;
   TestCases;                // Built-in test cases
} BITObject;
```

Figure 2. An object with built-in tests.

The BIT object has the same behaviour as that of the conventional objects when normal functions are called. But if the tests are called as that of member functions in the object, eg:

```
BITObject :: TestCase1;
BITObject :: TestCase2;
......
BITObject :: TestCaseI;
......
BITObject :: TestCaseN;
```

the object can be automatically tested and corresponding results are reported.

2.2 Built-in test at system level

The same built-in test method describe above can be extended to system level of object-oriented software. An OO software with BIT subsystem and classes is shown in Fig.3. Where modules 1.n, 3.k and 2.m are the BIT classes for the fully reusable, partially reusable and application-specific (non-reusable) subsystems respectively. The BITs at this subsystem level are designed to cope with interclass testing between class clusters. Subsystem 4 is a special BIT subsystem for the entire object-oriented system. As for the individual class level, such as within ASF_1, ASF_2, PRF_1 and etc, the BIT mechanisms have been described in section 2.1.

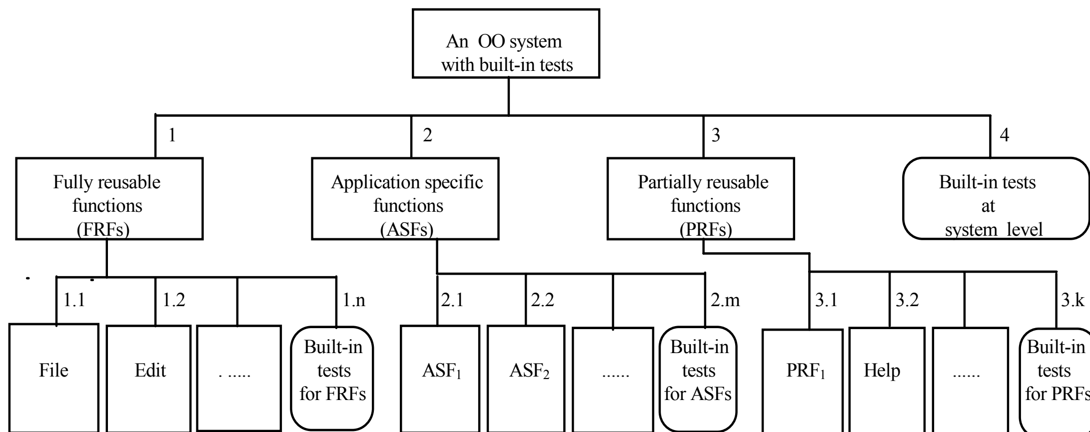

Figure 3. A prototype of built-in test OO system.

Reuse of BITs at object and object-oriented system levels can extend and maximise the benefits of inheritability enabled by OO programming. In this way, an ideal OO software which is testable, test inheritable and reusable is implemented at object, class, and system levels. The testable OO software also possesses a feature of easy maintenance because of its self-containment of code, and structure, as well as tests within a single source file [Wang *et al* 1999]. Thus the maintenance team and the end users of the OO software system are no longer required to redesign and reanalyse tests as well as the code and class structure for a BIT software system.

2.3 Implementation of BITs in OO software

A case study of application of the BIT method is provided in Fig.4. A typical example, the binary search, is given to show how to use the BIT method to develop built-in test objects and software, and to demonstrate how BITs are inherited and reused in the same way as those of the conventional code. A test case for the binary search algorithm is adopted from [Sommerville 1992].

Fig.4 shows a conventional program of binary search with the BITs as special member functions in the object. In Fig.4, the BIT object is implemented in two parts: the conventional functions and the BIT functions.

```
Class BITsBinarySearch {

//////////////////////////////////////////////////////////////////////////
//  Interface
//////////////////////////////////////////////////////////////////////////

// Member functions
BITsBinarySearch();                          // The constructor
~BITsBinarySearch();                         // The destructor
int BinarySearch (int Key;  int DataSet[10]);   // The conventional object
void BIT1();                                 // The built-in-test

////////////////////////////////////////////////////////////////////////
//  Implementation
////////////////////////////////////////////////////////////////////////

// ================================
//  Part 1: The conventional function code
// ================================
int BinarySearch (int Key,  int DataSet[10])
{
// The conventional object
// Assume: DataSet is ordered
//              LastElement -FirstElement >=0
//              and FirstElement >=0
// Input:      Key to be found in the DataSet
// Output:   TestElemIndex

Private:
        int bott, top, i;
        int found;
```

```
found = false;
Bott = 1;
Top = ArraySize (DataSet);   // The last element in DataSet
while (bott <= top) && (not found)
        {
         i = floor ((bott + top)/2));
         if DataSet[i] == Key
            Found = true;
            else   if   DataSet[i] < Key
                        Bott = i +1
                        else  Top = i +1;
        }
if found == true
   return i;           // The index of the element
   else return 0;      // An indicator of not existence
}

// ===================================
//  Part 2: The BITs
// ===================================

// BIT case 1
// ----------------------------------------------------------
void BIT1()
{
// BIT case 1: Odd array size, key not in array
private:
   int DataSet[7] = {16,18,21,23,29,33,38};
   int Key =  25;
   int StdElemIndex = 0;
   int TestElemIndex;
   char TestResult1 [5];
// Test implementation
   TestElemIndex = BinarySearch (Key, DataSet);
// Test analysis
   cout  << "StdElemIndex1 =  "   <<  StdElemIndex    <<  "\n";
   cout  << "TestElemIndex1 =  "   <<  TestElemIndex   <<  "\n";
   if  TestElemIndex == StdElemIndex
      TestResult1 = "OK";
      else  TestResult1 = "FALSE";
   cout << "TestResult1: " <<  TestResult1  <<  "\n";
}
}
```

Figure 4.   A BIT object of binary search.

For the binary search function listed in part 1 of Fig.4, a set of test cases can be generated by using equivalency partitioning technique or others. In Fig.4 one of the test cases is built-in to show the method of BITs. It is significant that the BIT method can incorporate any test cases generated by the black-box (functional) and/or white-box (structural) testing methods as the BITs. This feature sets up the foundation of dynamic inconsistency controlling by BITs.

In the normal mode, the conventional functions described in Fig.4 can be executed by calling:

```
BITsBinarySearch::
BinarySearch(int Key,  int DataSet[10])                                   (1)
```

In the test mode, the embedded BITs in the class can be activated by calling the follows:

```
BITsBinarySearch::BIT1();                                                 (2)
```

The other advantage of the BIT method is that when a new object is developed based on the existing ones, the BITs and inconsistency protection mechanisms embedded in the existing objects and system, can be inherited and reused directly as that of the conventional member functions. Also new BITs can be incorporated into the new object as shown in part 3 of Fig. 5.

```
class DatabaseQuery: public BITsBinarySearch
{
/////////////////////////////////////////////////////////////////////////////
//  Part 1: The inherited conventional functions
/////////////////////////////////////////////////////////////////////////////

int DatabaseQueryBinarySearch (int Key,  int DataSet[10]) :
    BITsBinarySearch::BinarySearch(int Key;  int DataSet[10]);
……

/////////////////////////////////////////////////////////////////////////////
//  Part 2: The inherited BIT functions
/////////////////////////////////////////////////////////////////////////////
void BIT1() : BITsBinarySearch::BIT1();
……

/////////////////////////////////////////////////////////////////////////////
//  Part 3: The newly developed BITs
/////////////////////////////////////////////////////////////////////////////

void BIT2()
{
// BIT case 2
......
}

}
```

Figure 5. Inheritability and reusability of BITs.

In the BIT approach, what the developer, end user and maintainer inherit is instant and self testable, so that the software inconsistency can be controlled both in static as well as at run-time by BITs. Assuming that the existing software system can be reengineered using the BITs method, the future software production will benefit strongly from the reuse of the BITs in system development, testing, maintenance and application.

3. Real-time dynamic inconsistency control by BITs

By incorporating the related BITs into an OO software system during development, software dynamic inconsistencies can be detected, diagnosed and handled at run-time. A generic example is provided in this section for demonstrating that how to control dynamic inconsistency by BITs at run-time, based on real world applications.

This section will not discuss the design of specific BIT for a certain type of dynamic inconsistency of a specific system. However, it is noteworthy that the BIT method enables incorporation of any kind of testing cases that are specifically designed for a specific system into source code. For instance, a decay inconsistency may be tested by the check-sum technique. Therefore a BIT which implements a check-sum testing case can be adopted to detect software decay inconsistency.

3.1 Dynamic inconsistency detection by BITs

Based on the BIT method developed in Section 2, software inconsistency can be tested regressively at run-time. When BITs at object, class and system levels, such as those shown in Figs. 3 and 4, are executed, testing results for an OO software system can be automatically reported as follows:

Table 3. System dynamic inconsistency status report.

Function level	**Dynamic inconsistency**				
	Run-time	**Decay**	**Random**	**Environment**	**Configuration**
System	Y/N	Y/N	Y/N	Y/N	Y/N
Subsystem i	Y/N	Y/N	Y/N	Y/N	Y/N
Class i	Y/N	Y/N	Y/N	Y/N	Y/N
Object i	Y/N	Y/N	Y/N	Y/N	Y/N
Function i	Y/N	Y/N	Y/N	Y/N	Y/N

In Table 3, a 'Y/N' indicates if a specific dynamic inconsistency at a specific software level is detected or not. The BITs at different levels can be called in predesigned interval or situation by the software system itself. BITs can also be activated manually. In any cases, if an inconsistency is detected at run-time at a specific level, inconsistencies alarm will be generated at this level and reported to the higher levels.

3.2 Dynamic inconsistency diagnosis by BITs

In case a dynamic inconsistency is detected, source and type of the inconsistency can be diagnosed and allocated based on the dynamic inconsistency detection report shown in the form of Table 3. An example diagnosis report for tracing the dynamic inconsistency is shown in Table 4. By this approach, detailed dynamic inconsistency can be allocated by the corresponding BITs at run-time.

Table 4. System dynamic inconsistency diagnosis report.

System structure			**Dynamic inconsistency allocation**				
			Run-time	**Decay**	**Random**	**Environment**	**Configuration**
System			Y	Y	Y	Y	Y

$Subsystem_1$			Y	Y	Y	Y	Y
	$Class_1$		Y		Y		
	$Object_{11}$			Y			
		$Function_{111}$					
		$Function_{112}$		Y			
		......					
		$Function_{11p}$					
	$Object_{12}$		Y		Y	Y	
		$Function_{121}$				Y	
		$Function_{122}$	Y				
		......					
		$Function_{12r}$	Y		Y		
		......					
	$Object_{1n}$		Y		Y		Y
		$Function_{1n1}$	Y		Y		
		$Function_{1n2}$	Y				Y
		......					
		$Function_{1ns}$			Y		
		......					
	Class 2		Y		Y		
	$Object_{21}$						
	$Object_{22}$		Y				
	......						
	$Object_{2m}$				Y		
	......						
$Subsystem_2$							
......							
$Subsystem_k$							

The diagnosis result shows the accurate sources and reasons of system dynamic inconsistency. Observing Table 4 it can be found that one subsystem, two classes, five objects and seven functions have been allocated for a specific or hybrid dynamic inconsistency.

3.3 Dynamic inconsistency handling by BITs

After the detection and allocation of any dynamic inconsistencies at run-time, a BIT at system level can invoke an appropriate dynamic inconsistency handling subroutine according to predesigned dynamic inconsistency processing strategy. Typical dynamic inconsistency handling strategies can be the measures such as to alarm, report, reset system, reload objects, reload data, reconfiguration, switch to stand-by system, replace hardware, etc. A decision table of dynamic inconsistency handling for a real-time software system, for instance, is given in Table 5.

Table 5. Dynamic inconsistency handling decision table.

Dynamic inconsistency diagnosis result	**Dynamic inconsistency handling strategy**						
	Alarm and	**System Reset**	**Reload objects**	**Reload Data**	**Re-configuration**	**Switch to stand-by**	**Replace hardware**

	report					system	
System	x	x	X	x	x	x	x
Subsystem_1	x	x	X	x	x	x	x
Class_1	x	x	X	x	x		x
Object_{11}		x					x
Function_{112}		x					x
Object_{12}	x		X				
Function_{121}	x		X				
Function_{122}	x		X				
Function_{12r}	x		X			x	
Object_{1n}	x			x	x		
Function_{1n1}	x			x			
Function_{1n2}	x					x	
Function_{1ms}	x				x	x	
Class_2	x		X				
Object_{22}	x		X				
Object_{2m}	x		X				

In Table 5, a decision table for detected dynamic inconsistency is generated automatically according to a system's predesigned dynamic inconsistency handling strategy. Based on the decision table, automatic or manual system fix can be carried out for a real-time system.

This section shows that the BIT method enables real-time detection, diagnosis and handling of software dynamic inconsistencies. BIT provides a systematic approach to improve real-time software system reliability and maintainability. In addition, the BIT method is also suitable for controlling static inconsistencies, such as the implementation and maintenance inconsistencies.

4. Conclusions

This paper has developed a new approach to detection, diagnosis and handling software dynamic inconsistency at run-time by BITs. Eight types of static and dynamic inconsistency in software development and run-time have been identified and modelled. Problems and measures in coping with dynamic inconsistency at run-time have been analysed.

The BIT method has extends not only inheritability and reusability of conventional OO software from code to embedded tests; but also software inconsistency protection capability from static to dynamic and from development to real-time application. The BIT method has found wide range of applications in controlling software dynamic inconsistency at run-time. Practical applications show that by adopting the BIT approach, dynamic software inconsistency can be detected, diagnosed and handled at run-time.

Acknowledgements

This work has been partially supported by EC SPIRE and Swedish PILOT projects. The BIT method has been applied in software development organisations for improving software testability, test reusability and reliability. The authors would like to acknowledge the

funding organisations for their support and the software organisations for their participation in the pilot projects.

References

[1] Cohen, B., Harwood, W. T. and Jackson, M. I. (1986), *The Specification of Complex Systems,* Addison-Wesley Publishing Co., pp. 15-22.
[2] Freedman, R.S. (1991), "Testability of Software Components", *IEEE Transactions on Software Engineering,* Vol.17, No.6, June, pp.553-564.
[3] Harry, A. (1996), *Formal Methods - Fact File - VDM and Z,* John Wiley and Sons, England, pp. 57-88.
[4] Hatcher, W. S. (1982), *The logic Foundations of Mathematics,* Pergamon Press, Oxford, pp. 32-39.
[5] Jorgensen, P.C. and Erickson, C. (1994), "Object-Oriented Integration Testing", *Communications of the ACM,* Vol. 37, No. 9, Sept., pp. 30-38.
[6] McDermid, J. A. ed. (1991), *Software Engineer's Reference Book,* Butterworth Heinemann Ltd., Oxford, pp.24.1-24.12.
[7] Murphy, G.C., P. Townsend and P. S. Wong (1994), "Experiences with Cluster and Class Testing", *Communications of the ACM,* Vol. 37, No. 9, Sept., pp. 39-47.
[8] Sommerville, I. (1992), *Software Engineering (4th ed.),* Addison-Wesley Publishing Company Inc., pp.427-437.
[9] Ullman, J. D. (1987), *Principles of Database Systems (2nd ed.),* Pitman Publishing Ltd., London, pp. 211-267.
[10] Voas, J.M. and Miller, K.M. (1995), "Software Testability: The New Verification", *IEEE Software,* Vol.12, No.3, May, pp.17-28.
[11] Wang, Y. (1988), "Testability Theory and Its Application in the Design of Digital Switching Systems", *Journal of Telecommunication Switching,* Vol.17, pp.30-36.
[12] Wang, Y. (1995), "On Testable Software and Test Reuse in OOP", Technical Report of Oxford University Computing Laboratory, OUCL-WANG-95002, pp.1-28.
[13] Wang Y., King, G., Court, I., Ross, M. and Staples, G. (1997), "On Testable Object-Oriented Programming", *ACM Software Engineering Notes (ACM SEN),* July, Vol. 22, No.4, pp.84-90.
[14] Wang Y., King, G., Patel, D., Court, I., Staples, G., Ross, M. and Patel, S. (1998), "On Built-in Test and Reuse in Object-Oriented Programming", *ACM Software Engineering Notes (ACM SEN),* Vol. 23, No.4, pp.60-64.
[15] Wang, Y., Staples, G., Ross, M., King, G. and Court, I. (1996), On a Method to Develop Testable Software, Proc. of IEEE European Testing Workshop (IEEE ETW'96), Montpellier, France, June, pp. 176-180.
[16] Wang, Y., Trujillo J. and Palomar M. (1997), "On a Metric of Software Testability", *Journal of the Spanish Computer Society (Novatica),* Vol.125, Jan/Feb Issue, pp.10-13.
[17] Wang, Y., King, G. and Wickburg H. (1999), "A Method for Built-in Tests in Component-based Software Maintenance", Proc. of 3rd European Conference on Software Maintenance and Reengineering, IEEE Press, Amsterdam, March, pp. S2.3.1-4.