<table>
<thead>
<tr>
<th><strong>Title</strong></th>
<th>ROCS: An object-oriented class-level testing system based on the Relevant Observable ContextS technique</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Author(s)</strong></td>
<td>Chen, HY; Tse, TH; Deng, YT</td>
</tr>
<tr>
<td><strong>Citation</strong></td>
<td>Information And Software Technology, 2000, v. 42 n. 10, p. 677-686</td>
</tr>
<tr>
<td><strong>Issued Date</strong></td>
<td>2000</td>
</tr>
<tr>
<td><strong>URL</strong></td>
<td><a href="http://hdl.handle.net/10722/55519">http://hdl.handle.net/10722/55519</a></td>
</tr>
<tr>
<td><strong>Rights</strong></td>
<td>This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.; Copyright © 2000 Elsevier Science B.V. All rights reserved.</td>
</tr>
</tbody>
</table>
ROCS: an object-oriented class-level testing system based on the Relevant Observable Contexts technique

Huo Yan Chen
Department of Computer Science, Jinan University, China

T. H. Tse
Department of Computer Science and Information Systems, The University of Hong Kong

Yue Tang Deng
Department of Computer Science, Polytechnic University, Brooklyn, NY

Abstract
Given an algebraic specification of a class of objects, we define a fundamental pair as two equivalent terms generated by substituting all the variables on both sides of an axiom by normal forms. For any implementation error in the class, if it can be revealed by two equivalent terms in general, it can also be revealed by a fundamental pair. Hence, we need only select test cases from the set of fundamental pairs instead of equivalent pairs in general. We highlight an algorithm for selecting a finite set of fundamental pairs as test cases. Further, by using the relevant observable contexts technique, we highlight another algorithm to determine whether the objects resulting from executing a fundamental pair are observationally equivalent. If not, it reveals an implementation error.

Using these algorithms, we have constructed a system to test object-oriented programs at class-level. We describe in detail the implementation of a prototype system, including the data structure of a Data member Relevant Graph (DRG) for the class, the procedures for the construction and path traversal of the DRG, the generation and execution of relevant observable contexts on the objects under test, and the reporting of implementation errors. The implementation illustrates an innovative idea of embedding testing algorithms into an interpreter to facilitate software testing.

Keywords: Equivalent terms; Fundamental pairs; Class-level testing; Object-oriented testing; Observational equivalence; Relevant observable contexts

1 INTRODUCTION
The object-oriented paradigm enhances the reliability, maintainability, and reusability of resulting software. It is known as a technique for improving the productivity, quality, and innovation in software development [1]. However, software testing becomes more complex and difficult than that for conventional programming. It contains four levels: algorithmic level, class level, cluster level, and system level [2]. In this paper, we will only discuss the most fundamental and yet very important level, namely the class level.

There are many possible combinations when methods in a class are invoked. Hence, test cases for object-oriented software at the class level involves not only individual operations but various sequences
of operations, which are known formally as “ground terms”. If two ground terms are equivalent according to the specification, but their implemented method sequences generate observationally non-equivalent objects, then there is an error in the implementation. Using this concept, pairs of equivalent ground terms should be selected as test cases for any given class. This recommendation cannot be directly applied in practice, however, because the set of all equivalent pairs is infinite in most circumstances.

Various authors have proposed black-box techniques for selecting class-level test cases from equivalent ground terms [3–8]. Others have proposed white-box techniques for test case selection [2, 9–14]. We have proved that it is impossible to determine whether two objects are observationally equivalent using a pure black-box technique [15]. On the other hand, when part of the specification is missing in an implementation, there is no way of revealing this problem using a pure white-box technique.

In order (a) to overcome the shortcomings of the black-box and white-box techniques and (b) to reduce the domain of selection of test cases while keeping the test coverage unchanged, we propose an improved methodology for class-level testing that integrates the black and white approaches. This method covers the selection of test cases from the set of fundamental (equivalent) pairs and the generation of a relevant finite subset of the set of observable contexts for determining the observational equivalence of the objects resulting from the execution of a test case.

We will first of all outline our black-and-white integrated approach, and then focus our discussions on the implementation of a prototype system based on the “Relevant Observable Contexts” (ROCS) technique.3 The implementation illustrates an innovative idea of embedding testing algorithms into an interpreter to facilitate software testing.

Section 2 introduces the basic concepts to be used in this paper. In Section 3, we outline our black and white integrated approach. In Section 4, we present the implementation and experiment of a prototype system based on the ROCS technique, including the representation, the construction and path traversal of a data member relevance graph for a given class, the generation and execution of relevant observable contexts on the objects under test, and the determination of implementation errors. Finally, Section 5 concludes the paper.

2 BASIC CONCEPTS
Among formal methods for the specification of object-oriented programs, algebraic specifications are one of the more popular approaches [16–19]. A syntax declaration and a semantic specification compose an algebraic specification for a class. The syntax declaration declares the operations involved, plus their domains and co-domains, corresponding to the input parameters and output of the operations. The semantic specification contains equational axioms that specify the behavioral properties of the operations.

Example 1
An algebraic specification for the class of integer stack.

```
module INTSTACK is
classes Int Bool IntStack
inherting INT
operations
  new: → IntStack
  _empty: IntStack → Bool
  _push (y: IntStack Int) → IntStack
  _pop: IntStack → IntStack
  _top: IntStack → Int ∪ {NIL}
variables
  S: IntStack
  N: Int
```

3 If we regard “Black and White” as a Scotch Whiskey, it will be nice to have it “on the ROCS”.

2
A term is a sequence of operations in an algebraic specification satisfying its syntactic requirements. For example, \texttt{new.push(1).push(2).pop} is a term in the class of integer stacks above. A term may be transformed into another using the equational axioms of the specification as progressive left-to-right rewriting rules. It is in normal form if and only if it cannot be further transformed by any axiom in the specification. For example, \texttt{new.push(1).push(2)} is in normal form but \texttt{new.push(1).push(2).pop} is not, since the latter can be transformed into \texttt{new.push(1)} using axiom \texttt{a4} as a rewriting rule.

A term without variables is referred to as a ground term. We only consider ground terms in this paper because actual test cases in dynamic testing involve ground terms only. An algebraic specification is said to be canonical if and only if every sequence of rewrites on the same ground term reaches a unique normal form in a finite number of steps. In other words, every ground term of a canonical specification has a unique normal form. Canonical specifications are terminating and free from confusion, and hence only such specifications will be discussed in this paper. Two ground terms \(u_1\) and \(u_2\) are said to be equivalent (denoted by \(u_1 \sim u_2\)), if and only if both of them can be transformed into the same normal form by some axioms in the algebraic specification as left-to-right rewriting rules.

Let \(C\) be a class in a given specification. An observer of \(C\) is an operation or method that returns attribute values of an object in \(C\) without affecting any observable attributes. A creator of \(C\) is an operation or method that returns initial objects of \(C\). The state of an object in \(C\) is the combination of all the attribute values of this object. A constructor or transformer of \(C\) is an operation or method that transforms the state of an object in \(C\). In other words, when a constructor or transformer acts on an object, it changes at least one attribute value of the object. The difference between a constructor and a transformer is that the former can appear in a normal form but the latter cannot. In Example 1, for instance, the operation \texttt{new} is a creator, \texttt{._push(N)} is a constructor, \texttt{._pop} is a transformer, and \texttt{._empty} and \texttt{._top} are observers.

An observable context on \(C\) is either (a) an observer in \(C\) or (b) a sequence of operations or methods satisfying the syntactic requirements in \(C\), that starts with a constructor or transformer but ends with an observer.

Given a canonical specification and its implementation in a class \(C\), two objects \(O_1\) and \(O_2\) in \(C\) are said to be observationally equivalent (denoted by \(O_1 \cong O_2\)) if and only if, for any observable context \(oc\) on \(C\), \(O_1.\text{oc}\) and \(O_2.\text{oc}\) produce either identical results or observationally equivalent objects in the output class of \(oc\) [15].

3 SUMMARY OF OUR APPROACH
There are two important theoretical aspects in our approach.

One is the concept of a fundamental pair, which is obtained by replacing all the variables on both sides of an axiom by normal forms. In example 1, for instance, the pair of equivalent ground terms \texttt{new.push(2).push(6).pop} \(\sim\) \texttt{new.push(2)} is a fundamental pair, since it can be formed by replacing the variables \(S\) and \(N\) in axiom \texttt{a4} by the normal forms \texttt{new.push(2)} and “6”, respectively. However, \texttt{new.push(8).pop.push(6).push(7).pop} \(\sim\) \texttt{new.push(6)} is not a fundamental pair, since it cannot be formed directly from any of the axioms.

The other theoretical aspect in our approach is the formulation of a theorem, which states that an implementation of a canonical specification is consistent with respect to all equivalent terms if and only if it is consistent with respect to all fundamental pairs. In this way, although the set of fundamental pairs is a proper subset of the set of general equivalent ground terms, the testing coverage of fundamental pairs remains identical to that of general equivalent ground terms, and hence we need only concentrate on the testing of fundamental pairs. For example, we need only select test cases such as the
fundamental pair $\text{new.push}(2).\text{push}(6).\text{pop} \sim \text{new.push}(2)$, and need not consider more general equivalent ground terms such as $\text{new.push}(8).\text{pop}.\text{push}(6).\text{push}(7).\text{pop} \sim \text{new.push}(6)$.

Unfortunately, an axiom may induce an infinite number of fundamental pairs by assigning different normal forms to its variables. We need some means of selecting a finite number of representative test cases from this infinite number of cases. Assuming the regularity hypothesis and uniformity hypothesis [3], we have proposed an algorithm GFT for Generating a Finite number of Test cases. It consists of the following main steps:

1. Replace each variable of a non-observable type in an axiom $ax$ by a number of normal form patterns with lengths no greater than a given positive integer $k$, thereby unfolding $ax$ into several new equations. Repeat the process until all the variables in the new equations are of observable types. The integer $k$ may be determined by a white-box technique, such as by referring to the maximum sizes of arrays or the boundary values of variables declared in the implemented code.

2. Partition the input domain of the operation in each derived equation in (1) into sub-domains using the conditions in the set of the axioms defining the operation.

3. Randomly select an element from each sub-domain obtained above. Use these elements to replace all occurrences of the corresponding variables in the new equation to obtain a group of fundamental pairs for the axiom $ax$.

On the other hand, in spite of the theorem above, an infinite set of observable contexts may be required to check the observational equivalence of objects resulting from the fundamental pairs. We have proved that this problem cannot be solved by any black-box technique. Based on white-box techniques, therefore, we have constructed a heuristic algorithm to select a finite subset of the set of relevant observable contexts. It is known as DOE for Determining Observational Equivalence. The basic idea is as follows:

Suppose we want to decide whether the objects $O_1$ and $O_2$ resulting from the execution of a test case are observationally equivalent. Suppose, further, that $O_1$ and $O_2$ have different values for the same data member $d_i$ of the implemented class. Such different values may or may not have an effect on the observable attributes of $O_1$ and $O_2$. If no observable attribute is affected, $d_i$ need not be considered. If some observable attribute is affected, $d_i$ must have affected the attribute through some series of methods in the implemented class. Such a series of methods is called a relevant observable context. We need only use the relevant observable contexts to decide whether $O_1$ and $O_2$ are observationally equivalent. We can ignore any other observable contexts for this decision. The relevant observable contexts are constructed from a Data member Relevance Graph (DRG), which is an abstraction of the given implementation of a given specification.

In the DRG of an implemented class, a bold rectangular node denotes a data member and a thin rectangular node represents some constant coming from the program under test. If a data member $d_2$ directly affects another data member $d_1$ in the method $m_1$ under a condition $p(...)$, we draw an arc from $d_2$ to $d_1$ and label it as $(p, m_1)$. We call $[d_2, (p, m_1), d_1]$ a segment of the DRG, $d_2$ a start node of the segment, $d_1$ an end node of the segment, $(p, m_1)$ an output arc of $d_2$, and $(p, m_1)$ an input arc of $d_1$. If $d_2$ is identical to $d_1$, the segment is said to be a cycle. Otherwise it is said to be acyclic. If $m_1$ is an observer, the segment is called an observable segment. Each DRG contains a special node called observed, which is the end node of an observable segment. An arc with observed as an end node is call an observer arc.

**Example 2**

Consider the specification in Example 1. Suppose the implementation of the specification is as follows.

```cpp
#include <iostream.h>
define SIZE 100
define NIL 0
enum bool { false, true };
```
class intStack
{
    /* intStack consists of 2 data members: */
    int array[SIZE];
    int height;
    public:
        void newStack( );
        bool empty( );
        void push(int i);
        void pop( );
        int top( );
};

void intStack :: newStack( )
{  
    height = 0;
    for (int j = 1; j <= 100; j++)
        array[j] = NIL;
}

bool intStack :: empty( )
{  
    if (height == 0)
        return true;
    else return false;
}

void intStack :: push(int i)
{  
    if (height == SIZE)
        cout << "Stack is full";
    else
        {  
            height = height + 1;
            array[height] = i;
        }
}

void intStack :: pop( )  
{  
    if (height > 6)   /* Error: The condition should be height > 0 */
        height = height − 1;
}

int intStack :: top( )  
{  
    if (height > 1)   /* Error: The condition should be height > 0 */
        return array[height];
    else  return NIL;
}

The DRG of the implementation is shown in Figure 1.
Suppose $d_i$ is a data member of an implemented class $C$, and $O_1$ and $O_2$ are two objects of $C$. If ($a$) $O_1.d_i \neq O_2.d_i$, ($b$) there is a path $P$ from the node $d_i$ to the node observed in the DRG of class $C$, and ($c$) the methods in the labels of the arcs in path $P$ are $op_1, op_2, \ldots, op_t$, and $obs$, then $op_1.op_2.\ldots.op_t.obs$ will be a relevant observable context induced from the path $P$ with respect to $O_1$ and $O_2$.

The benefits of our integrated approach are as follows.

($a$) We reduce the selection domain of test cases but the test coverage remains identical.

($b$) We skip the testing of many irrelevant observable contexts when deciding whether the objects $O_1$ and $O_2$ resulting from the execution of a test case are observationally equivalent.

($c$) We have overcome the problem of “missing paths” in pure white-box techniques.

($d$) When compared with the work of Doong and Frankl [6, 7], we need not require the specifier to add a special-purpose axiom $eqn$ to each class in order to define the operational semantics of the equivalence of objects. Nor do we require the designer and programmer to implement a special-purpose recursive method for the respective $eqn$ axiom in each class. Thus, we can avoid rejecting a correct implementation of the original class having a problematic $eqn$ axiom or implementation.\footnote{The situation is acceptable only if the $eqn$ function happens to be part of the original class under test.}

\footnotetext[4]{The situation is acceptable only if the $eqn$ function happens to be part of the original class under test.}
4 IMPLEMENTATION OF THE ROCS SYSTEM

The main objective of our previous paper [15] was to present the theoretical details of our approach. It only provided an outline of the algorithms involved. We are presenting in this paper the implementation details of the ROCS system so that readers may have a better insight on the working of the methodology.

A special feature of our implementation of the ROCS system is that the test algorithm DOE has been embedded into a C++ interpreter, which is an extension of the interpreter for Little C [20]. Thus, ROCS can be regarded as an interpreter that has been enhanced to include testing functions. It covers the construction and path traversal of DRG, the execution of relevant observable contexts, and the determination of implementation errors. In general, testing techniques should scan the program code under test, and hence it is natural and effective to consider embedding them into a compiler or interpreter. Our implementation of the ROCS prototype provides a successful experience in this aspect.

The prototype of ROCS has been implemented using Borland C++. It consists of five modules: parser.c, drg.c, pigeonC.h, subLib.c, and pigeonC.c. The module pigeonC.h defines the main data structure, and subLib.c defines the interfaces to internal library functions. The module parser.c consists of a lexical analyzer and a recursive descent parser. The lexical analyzer can also be called by drg.c and other modules. The module drg.c constructs the DRG, traverses executable paths by backtracking, and generates and executes the corresponding relevant observable contexts for any two given objects. Finally, pigeonC.c serves as the main module of the prototype. It reads the C++ program code for a given class under test, allocates memory for the program, pre-scans it, and calls and coordinates other modules to perform the tasks.

Let $O_1$ be an object of the implemented class $C$, and let $d_1, d_2, ..., d_n$ be the data members. A path $P$ in the DRG of $C$ is said to be executable for $O_1$ if $O_1.d_i$ satisfies all the conditions on the labels of the arcs in $P$ as initial data. Otherwise $P$ is said to be inexecutable for $O_1$.

The tasks of the ROCS system are as follows.

1. Read the code of the class $C$ under test.
2. Read a given fundamental pair as a test case generated by Algorithm GFT from the specification of $C$.
3. Scan the code of $C$ and draw all the arcs from the code.
4. Generate the segments from the arcs and obtain the data structure of the DRG.
5. Execute the method sequences corresponding to the given fundamental pair.
6. Let $O_1$ and $O_2$ be two objects resulting from the execution.
   
   For each data member $d_i$ such that $O_1.d_i \neq O_2.d_i$, traverse some executable paths from the node $d_i$ to the node observed in the DRG (with backtracking if necessary) and obtain some relevant observable contexts $oc_j$ induced from these paths.
7. Execute $O_1.oc_j$ and $O_2.oc_j$ in the program under test.
   
   If some execution result shows that $O_1.oc_j \neq O_2.oc_j$, then report an implementation error.

In order to implement these tasks, we should first of all consider the data structure of the DRG. The data structure will affect significantly the space efficiency of the system.

4.1 Data Structure of DRG

A DRG consists of three kinds of basic elements, namely nodes, arc labels, and segments. A segment denotes the connection between two nodes as well as the arc label involved. The module pigeonC.h defines three data structures to represent the arc labels, segments, and nodes as follows:
struct arcLabel
{
    char condition[STATEMENT_LEN];  /* Condition p */
    char method[ID_LEN];             /* Method m */
};

struct segment
{
    int arcLabelIndex;             /* The index of the arc label of the segment */
    int endNodeIndex;              /* The index of the end node of the segment */
    int iterationNumb;             /* For the case of cycles, the number of iterations required when backtracking */
};

Since a number of segments may have the same arc label and since the number of segments is much more than that of nodes, we use indices rather than real entities in the fields of the struct segment. In this way, the space for storing segments will be reduced. We will explain later why there is no need to include a field in struct segment to denote its start node.

struct node
{
    char dataMembName[ID_LEN];    /* Name of the data member */
    int dataMembType;            /* Type of the data member */
    int value;                   /* Value of the data member, if the data member is of a simple type */
    int arraySize;               /* Array size if this data member is an array; otherwise it is 0 */
    struct segment acycSegList[LIST_LEN];  /* The list of acyclic segments starting from the node */
    int acycSegsSize;            /* The size of acycSegList */
    int acycSegsIndex;           /* The current position of acycSegList */
    struct segment obsSegList[LIST_LEN];   /* The list of observable segments starting from the node */
    int obsSegsSize;             /* The size of obsSegList */
    int obsSegsIndex;            /* The current position of obsSegList */
    struct segment cycSegList[LIST_LEN];  /* The list of cycle segments starting from the node */
    int cycSegsSize;             /* The size of cycSegList */
    int cycSegsIndex;            /* The current position of cycSegList */
};

Here, the field acycSegList, obsSegList, or cycSegList in struct node is called a segmentList field. The data elements acycSegsIndex, obsSegsIndex, and cycSegsIndex are used to mark the first untraversed segment for backtracking in the traversal of the executable paths of a given object. Their initial values are 1.

A natural representation of a segment is \([d_2, (p, m_1), d_1]\), where \(d_2\) is the start node of the segment. We need not, however, define a field in struct segment to denote \(d_2\). It is because struct segment is used only in the segmentList fields of struct node, and hence the start node of struct segment is just the struct node self. By omitting the obvious start node, the space for storing struct segment can also be reduced.

The module pigeonC.h uses the following data structures for representing a DRG for a given class under test:

1) An array known as arcLabelList, containing all the arc labels of the DRG. It is declared by the statement

\[
\text{struct arcLabel arcLabelList[NUM_ARCS].}
\]
(2) A struct node for each data member of the class. All of these struct nodes make up an array known as nodeList, which is declared by the statement

\[
\text{struct nodeList[NUM_DATA_MEMBERS].}
\]

4.2 Construction of DRG
The process of constructing a DRG is as follows.

(1) The module pigeonC.c pre-scans the program code for the given class under test, finds the locations of all the functions, methods, and global variables in the program, and sets up corresponding tables for future use, such as functionsTable, methodsTable, and globalVarsTable.

(2) The module pigeonC.c calls another module parser.c to perform lexical analysis and recursive descent parsing. The module parser.c also conducts the initialization for drg.c and other modules, including the supply of information to the appropriate fields in the struct nodes in nodeList, such as dataMembName and dataMembType.

(3) Based on the tables and the information above, the function constructDRG( ) in the module drg.c creates the arcLabelList of the DRG for the implemented class under test. It also completes the remaining fields of the struct nodes in nodeList, such as struct segment acycSegList[ ] and int acycSegsSize, by means of a function scanBlock( ), which is a variant of the function interp_block( ) in an interpreter [20].

(4) Suppose \( d_1 \) and \( d_2 \) are two data members in the implemented class. Let \( p \) be a predicate and \( c \) be a constant. In order to construct the DRG, the function scanBlock( ) performs the following tasks for each method \( m_i \). The generated segments corresponding to the scanned statements in this step is listed in Table 1.

(a) Scan the code of \( m_i \).

(b) When a statement of the form “\( d_1 = c \)” or “\( d_1 = f(..., d_2, ...) \)” is found,

put the arcLabel (true, \( m_i \)) into the arcLabelList, and

put the segment [index of arcLabel (true, \( m_i \)), index of \( d_1 \)]

into a segmentList field of the node \( c \) or the node \( d_2 \).

(c) When a statement such as “if (\( p \)) {\ldots; d_1 = c; \ldots}” or “if (\( p \)) {\ldots; d_1 = f(\ldots, d_2, \ldots); \ldots}” is found,

put the arcLabel (\( p, m_i \)) into the arcLabelList,

put the segment [index of arcLabel (\( p, m_i \)), index of \( d_1 \)]

into a segmentList field of the node \( c \) or the node \( d_2 \),

if \( p = p(\ldots, d_3, \ldots), d_3 \) is a data member different from \( d_2 \),

put the segment [index of arcLabel (\( p, m_i \)), index of \( d_1 \)]

into a segmentList field of the node \( d_3 \).

(d) If the statement also contains “else {\ldots; \( d_4 = c_0; \ldots}” or “else {\ldots; \( d_4 = g(\ldots, d_5, \ldots); \ldots}”,

put the arcLabel (\( \neg p, m_i \)) into the arcLabelList, and

put the segment [index of arcLabel (\( \neg p, m_i \)), index of \( d_4 \)]

into a segmentList field of the node \( c_0 \) or the node \( d_5 \).

(e) Skip the other statements in the method \( m_i \).

The time complexity for constructing the DRG of the class has been analyzed in [15].
### Table 1. Correspondence between Scanned Statements and Generated Segments

<table>
<thead>
<tr>
<th>Scanned Statement</th>
<th>Corresponding Generated Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d_1 = c )</td>
<td>((true, m_i)) ( d \leftarrow d_1 )</td>
</tr>
<tr>
<td>( d_1 = f(\ldots, d_2, \ldots) )</td>
<td>((true, m_i)) ( d_2 \rightarrow d_1 )</td>
</tr>
<tr>
<td>if ( ( p ) { \ldots; d_1 = c; \ldots } )</td>
<td>((p, m_i)) ( c \rightarrow d_1 )</td>
</tr>
<tr>
<td>if ( ( p ) { \ldots; d_1 = f(\ldots, d_2, \ldots); \ldots } )</td>
<td>((p, m_i)) ( d_2 \rightarrow d_1 )</td>
</tr>
<tr>
<td>if ( ( p(\ldots, d_3, \ldots) ) ( { \ldots; d_1 = f(\ldots, d_2, \ldots); \ldots }, ) [ where ( d_3 \neq d_2 ])</td>
<td>((p, m_i)) ( d_3 \rightarrow d_1 )</td>
</tr>
<tr>
<td>else ( { \ldots; d_4 = c_0; \ldots } )</td>
<td>((-p, m_i)) ( c_0 \rightarrow d_4 )</td>
</tr>
<tr>
<td>else ( { \ldots; d_4 = g(\ldots, d_5, \ldots); \ldots } )</td>
<td>((-p, m_i)) ( d_5 \rightarrow d_4 )</td>
</tr>
</tbody>
</table>

A question arises here. If the condition \( p \) appears in a \textit{for} or \textit{while} statement, how do we deal with it? We note that the DRG technique is concerned with the “directly affects” relations only among the data members and constants coming from the given program, rather than among other local auxiliary variables. If we regard a \textit{for} loop or \textit{while} loop as a function, we can concentrate only on the effects of the input data members of the function to the output data members of the function, and ignore the effects from local auxiliary variables. Suppose, for instance, that the code for operation \textit{push(i)} in Example 1 is as follows:

```c
void intStack :: push(int i)
{
    for ( int j = 1; j <= 99; j++ )
        array[j] = array[j+1];
    array[100] = i;
}
```

Since the condition \( j \leq 99 \) in the \textit{for} statement is related only to a local auxiliary variable \( j \), it need not be considered in the construction of the DRG. We can regard this \textit{for} statement as a function through which the data member \( array[\ ] \) affects itself. Since \( i \) in the statement “\( array[100] = i \)” represents a constant rather than an auxiliary variable, the effect of \( i \) on the data member \( array[\ ] \) must also be considered. Thus, we obtain two segments \( [array[\ ], (true, push(i)), array[\ ]] \) and \( [i, (true, push(i)), array[\ ]] \) from the code.

#### 4.3 Traversal of Executable Paths in DRG

The construction of a DRG for the given class is independent of the given fundamental pair as a test case. After constructing a DRG, ROCS executes the method sequences corresponding to the given fundamental pair, and produces two objects \( O_1 \) and \( O_2 \). For each data member \( d_s \) such that \( O_1.d_s \neq O_2.d_s \), ROCS must traverse the executable paths from the node \( d_s \) to the node \( observed \) in the DRG (with backtracking if necessary) to obtain the corresponding relevant observable contexts \( oc_j \) and then perform \( oc_j \) on the current states of \( O_1 \) and \( O_2 \). These tasks are conducted by the module \( drg.c \). The following stack is used for the traversal and backtracking:
struct stack
{
    int topIndex; /* The index of the top of the stack */
    struct traversedSegment travdSegList[LST_LEN]; /* The list of traversed segments */
};

struct traversedSegment
{
    /* A traversed segment */
    int arcLabelIndex; /* The index of the arc label of the segment */
    int endNodeIndex; /* The index of the end node of the segment */
    char preObject1[OBJECT_SIZE]; /* The state of object O1 before running the method in the arc label */
    char preObject2[OBJECT_SIZE]; /* The state of object O2 before running the method in the arc label */
};

Suppose a segment Seg2 is contiguous to another segment Seg1 in the travdSegList[]. The state of object O after running the method in the arc label of segment Seg1 is the same as that of object O before running the method in the arc label of segment Seg2. Hence, there is no need to include a field in struct traversedSegment to denote the state of object O after running the method in the arc label of the segment. In this way, the space for storing the segments in travdSegList[] can also be reduced.

For each given data member di such that O1.di ≠ O2.di, the path traversal process starts from the node di and consists of the following steps. Readers may also refer to the flowchart in Figure 2 for a better understanding the procedure.

(1) Assign data member di to a working variable d, and empty the stack by setting stack.topIndex to 1.

(2) Select an untraversed segment and use it as the value of the working variable thisSeg, as follows:

(a) If d.obsSegsIndex ≤ d.obsSegsSize, then
    select the segment d.obsSegList[obsSegsIndex],
    copy it to thisSeg, and
    increase obsSegsIndex by 1.

(b) Otherwise if d.acycSegsIndex ≤ d.acycSegsSize, then
    select the segment d.acycSegList[acycSegsIndex],
    copy it to thisSeg, and
    increase acycSegsIndex by 1.

(c) Otherwise if (d.cycSegsIndex ≤ d.cycSegsSize)
    and (d.cycSegList[cycSegsIndex].iterationNumb ≤ T),
    then
    select the segment d.cycSegList[cycSegsIndex],
    copy it to thisSeg,
    increase d.cycSegList[cycSegsIndex].iterationNumb by 1, and
    if the updated d.cycSegList[cycSegsIndex].iterationNumb > T, then
    increase d.cycSegsIndex by 1.

(d) If (d.obsSegsIndex > d.obsSegsSize) and (d.acycSegsIndex > d.acycSegsSize)
    and (d.cycSegsIndex > d.cycSegsSize), then
    /* there is no untraversed segment in d.obsSegList[obsSegsIndex],
    d.acycSegList[acycSegsIndex], or d.cycSegList[cycSegsIndex] */
    perform backtracking as described below.

Here, T is a global ceiling allowed by the system for the number of iterations of any cycle segment.
(3) If the condition `thisSeg.arcLabelIndex↑.condition` is satisfied by the current states of \(O_1\) and \(O_2\), then

- copy the values of `arcLabelIndex` and `endNodeIndex` in `thisSeg`
  to the corresponding fields of `stack.travSegList[topIndex]`.
- copy the current state of \(O_1\) to the field `preObject1` of `stack.travSegList[topIndex]`,
- copy the current state of \(O_2\) to the field `preObject2` of `stack.travSegList[topIndex]`,
- execute the method `thisSeg.arcLabelIndex↑.method` on \(O_1\) and \(O_2\), \(^6\)
- increase `stack.topIndex` by 1, and
- conduct the following:

  (a) If `thisSeg.endNodeIndex↑` is the special node `observed`, then
      /* `thisSeg.arcLabelIndex↑.method` should in fact be an observer */
      if the values resulting from the executions of step (3) are not identical, then
        report that an implementation error has been found
        because the original \(O_1\) and \(O_2\) are not observationally equivalent, and
        exit from the process;
      otherwise perform backtracking as described below.

  (b) If `thisSeg.endNodeIndex↑` is not the special node `observed`, then
      update the working variable \(d\) to `thisSeg.endNodeIndex↑`, and
      go to step (2) above.

(4) If the current states of \(O_1\) and \(O_2\) do not satisfy the condition `thisSeg.arcLabelIndex↑.condition`, then backtrack to step (2) to select another untraversed segment.

When backtracking is required, the following will be conducted:

(i) If `stack.topIndex = 1`, that is, if `stack` is empty,
    report that the given data member \(d\) has successfully passed the check, and
    exit from the process.

(ii) If `stack.topIndex > 1`, decrease `stack.topIndex` by 1.

(iii) Restore \(O_1\) to `stack.travSegList[topIndex].preObject1[]` and
    \(O_2\) to `stack.travSegList[topIndex].preObject2[]`.

(iv) Update the working variable \(d\) to `stack.travSegList[topIndex − 1].endNodeIndex↑`.

(v) Go to step (2) to update `thisSeg` by selecting an untraversed segment from
    \(d.\text{obsSegList}[^\text{obsSegsIndex}]\), \(d.\text{acycSegList}[^\text{acycSegsIndex}]\), or \(d.\text{cycSegList}[^\text{cycSegsIndex}]\).

Note that the concept of executability of a given path for a given object defined in previous section is very different from the concept of feasibility of a path in other flow graph techniques [21]. An infeasible path is usually defined as a path whose conditions cannot be satisfied by any input value, and is well-known to be undecidable. However, since the executable and unexecutable paths defined in the previous section are related to a given object \(O_1\), their conditions can be determined from the known values \(O_1.d\) of the data members of the given object \(O_1\). Thus, unlike the concept of feasibility, the executability of a given path for a given object is decidable.

---

\(^6\) Note that the states of \(O_1\) and \(O_2\) may be changed after the execution of this step.
4.4 Execution of Relevant Observable Contexts
We see from the previous subsection that the execution of a relevant observable context on the given objects $O_1$ and $O_2$ should be synchronized with the traversal of the corresponding executable path in the DRG. In the graph traversal process, whenever a path is extended by a segment, the method contained in the arc label of the segment is executed on the current states of $O_1$ and $O_2$.

![Flowchart on Traversal of Executable Paths in DRG](image)

**Figure 2. Flowchart on Traversal of Executable Paths in DRG**

4.5 Determining Object Equivalence or Implementation Error
During the path traversal process for a given data member $d_x$ such that $O_1.d_x \neq O_2.d_x$, when we reach the special node observed, we have completed traversing the entire executable path and executing the corresponding relevant observable context on $O_1$ and $O_2$. If the results of the execution on $O_1$ and $O_2$ are not identical, ROCS concludes that the original $O_1$ and $O_2$ are not observationally equivalent, report an
implementation error, and then exit from the task. Otherwise it will trigger backtracking, with a view to traversing another untraversed path. If all the data members $d_e$ such that $O_1.d_e \neq O_2.d_e$ have successfully passed the check, then ROCS reports that $O_1 \approx O_2$ and exit from the system.

4.6 Experimentation and Analysis
We have implemented the system on a Pentium II and experimented it with Example 2. All the errors in the example can be exposed. The time for constructing the DRG is 0.038731 s. The respective number of observable contexts generated and the total run time required are shown in Table 2.

<table>
<thead>
<tr>
<th>Global ceiling supplied by the user for the number of iterations of any cycle</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of observable contexts generated by the prototype</td>
<td>5</td>
<td>8</td>
<td>19</td>
<td>25</td>
<td>37</td>
</tr>
<tr>
<td>Total run time for all observable contexts</td>
<td>0.083 686 s</td>
<td>0.184 558 s</td>
<td>0.305 763 s</td>
<td>0.576 521 s</td>
<td>0.734 295 s</td>
</tr>
<tr>
<td>Run time for the first observable context that reports the error</td>
<td>–</td>
<td>–</td>
<td>0.186 549 s</td>
<td>0.195 764 s</td>
<td>0.274 391 s</td>
</tr>
</tbody>
</table>

Table 2. Run Times for Example 2

We have also experimented with the prototype using other programs that contain various types of error. Some programs contain common typos, such as having a condition $height > 0$ coded as $height < 0$ or $height > 10$. Some refer to non-existent elements of arrays, such as $array[1000]$. Some contain faults caused by placing statements in erroneous positions. Others have errors in the general ideas behind the programs, rather than in individual statements. All types of errors have been exposed by the system. The run times are acceptable.

Let $s$ be the number of statements in the code for a given class $C$. Since the construction of the DRG for class $C$ is based on scanning and processing each statement in the code, and the time for processing each statement is bounded, the time for constructing the DRG for $C$ is $O(s)$.

Let $L$ be the maximum length of all acyclic paths from any node to the node observed. Given any objects $O_1$ and $O_2$, let $n$ be the maximum number of output arcs in any node such that the Boolean conditions in the arcs are true for the current values of $O_1.d_e$ and $O_2.d_e$. Here, $L$ is a constant and $n$ is a variable depending on different objects $O_1$ and $O_2$. Since the ceiling for the number of iterations of cycles is a constant, the complexity of traversing executable paths is $O(n^2)$ for the worst case.

5 CONCLUSION
We have proposed an integrated approach for selecting fundamental pairs of equivalent ground terms as class-level test cases for object-oriented programs and applying observable context technique to determine whether the objects resulting from the execution of a test case are observationally equivalent. After outlining the basic idea of the approach, this paper describes in detail the prototype system based on the relevant observable contexts (ROCS) technique, including the representation, construction, and path traversal of a data member relevant graph for a given class, the generation and execution of relevant observable contexts on the objects under test, and the determination of implementation errors. The production of a prototype of the ROCS system provides an innovative experience for embedding testing processes into the language interpreter. A white-box testing technique involves the scanning and parsing of program code, and hence its integration with interpreters or compilers would help to
expedite the process. Some experiments have been conducted via the prototype system, and the empirical results agree with the outcome predicted by our framework.

REFERENCES


