Acta Cybernetica **18** (2008) 481–501.

A New Concept of Effective Regression Test Generation in a C++ Specific Environment

Mihály Biczó*, Krisztián Pócza*, István Forgács†, and Zoltán Porkoláb*

Abstract

During regression testing test cases from an existing test suite are run against a modified version of a program in order to assure that the underlying modifications do not cause any side effects that would demolish the integrity and consistency of the system. Since the ultimate goal of a regression test set is to effectively test all modifications and reveal errors in the earliest possible stage, the maintenance of a relevant test set containing effective test cases is of utmost importance. In this paper we present an efficient, C++ specific framework to automatically manage the regression test suite. Our two main contributions are a new interpretation of reliable test cases and a dynamic forward impact analyzer method that eases the transformation of existing tests to meet the definition of reliability. Using this approach we complement the test set with test cases that pass through a modification and have an impact on at least one output. Our approach is designed to be applicable to large-scale applications.

Keywords: regression testing, dynamic impact analysis, software maintenance, C++

1 Introduction

Regression testing is an important tool of software engineers to successfully manage issues rising during the evolution of software systems. During the lifetime of large systems numerous modifications are performed over possibly many years, yet it is of vital importance that none of these modifications is allowed to remain untested, or cause unwanted and undiscovered side effects to other previously tested parts.

In order to achieve this goal, a regression test set that covers the whole system has to be maintained and adjusted according to the modifications performed. Therefore, it is desirable to find a test selection method that selects those and only

*Eötvös Loránd University, Fac. of Informatics, Dept. of Prog. Languages and Compilers, Pázmány Péter sétány 1/C. H-1117, Budapest, Hungary E-mail: mihaly.biczo@t-online.hu, kpocza@kpocza.net, gsd@elte.hu

†4D SOFT Ltd. Telepy u. 24. H-1212, Budapest, Hungary E-mail: forgacs@4dsoft.hu

those test cases that might reveal an error [6]. However, it is equally important to re-use and transform existing test cases so that the coverage of the modified system would not be affected.

An important subset of regression tests contains *modification revealing tests* for which the original and modified programs give different output. All modification revealing tests are *modification traversing*, they reach at least one modified statement. Consequently, the set of modification traversing tests also contains all error revealing test cases [16]. Unfortunately, the reverse case is not true: a modification traversing test is not necessarily modification revealing. Existing methods consider a test case successful if the outputs of the original and modified programs are identical. As it can be seen easily, using this approach it is not assured that the modification is really tested. In other words, the test case is not necessarily reliable.

In this paper we alter the existing definition of reliability: the definition of a reliable test pair will be established. According to this definition, we develop an approach that eases the generation of reliable test pairs in a C++ specific environment. We will not cover test data generation techniques, related work can be found in [2, 3, 9, 10]. Instead, we identify those input variables from the whole state space on which the new generation process can be started. For the generation process, the method described in [17] can be used initiated on a reduced input variable set.

Our main contribution is a simple forward dynamic impact analyzer algorithm which, if there is a given modification, will efficiently select the set of influencing input variables and help boost the performance of the test pair generation process. As opposed to existing methods [14], instead of directly comparing the output of the original and modified programs for a given test case, the modified program and the underlying test case are considered. The test suite will be extended with a reliable test pair that is derived from the original test. This test pair will assure that the modification is tested and that some output statements are affected in the modified version of the program. An additional benefit of our approach is that it is designed to work for real C++ based systems, since many C++ specific constructs are covered including pointers and function pointers, as well as object-oriented constructs and paradigms like classes, inheritance and polymorphism.

The structure of the paper is the following: Section 2 defines the problem we are going to solve and presents the general overview of the generator framework through simple examples. We also give an insight into test categorization methods we are going to employ.

In Section 3 we discuss some related work and research directions we are aware of. We will primarily focus on the motivating ideas behind existing techniques.

In Section 4 an overview of the used notations and necessary language specific instrumentation mechanism will be described in detail.

In Section 5 and Section 6 the two stages of the dynamic forward input analyzer algorithm that detects affecting input variables will be discussed. While the first stage of the algorithm categorizes test cases and identifies affected statements; the second stage selects the underlying input variables based on the results of the

first phase.

In Section 7 a full example of our approach will be presented in C++.

In Section 8 we summarize our results and discuss the limitations of the approach as well as some possible research directions.

2 Framework overview

2.1 The necessity of a concept change

As we have mentioned in the introductory section, traditional regression testing approaches categorize test cases based on the outcome of the test case run against the original and the modified programs. However, numerous anomalies might prevent this comparison from being a good filter of errors.

The fundamental issue is that if a modification traversing test gives identical output for the original and modified programs, this does not mean that any of the modifications have really been tested. This is the case when the given modification does not affect any output statements. The reverse case - when the outcome of the original and modified programs differs - can also be problematic, because the test might not be modification traversing for a given modification. As a consequence, if there are more than one modifications (which is typically the case), classical modification revealing tests might not be effective, and once again untested modifications might lurk in the source code. A further example for different output is when the mistakenly modified statement is a predicate, and the test takes another execution branch, although it should go along the original path.

Listing 1 Three versions of a simple program

```
int main()
{
  double a,b,c, d;

  cin >> a;
  b=2;
  c=3;

  d=a+c;

  if(a>0)
    cout << b << endl;
  else
    cout << c << endl;

  //Use d...

  exit(0);
}
```

```
int main()
{
  double a,b,c, d;

  cin >> a;
  b=2;
  c=3;

  //Mod. #1: d=a+c;
  d=a-c;

  if(a>0)
    cout << b << endl;
  else
    cout << c << endl;

  //Use d...

  exit(0);
}
```

```
int main()
{
  double a,b,c, d;

  cin >> a;
  //Mod. #1: b=2;
  b=3;
  c=3;

  d=a+c;

  if(a>0)
    cout << b << endl;
  else //Mod. #2
    cout << c+2 << endl;

  //Use d...

  exit(0);
}
```

Let's consider the three different versions of a simple program in Listing 1. If the input of the program (the test case) is a=1, then the outcome of the original program is that 2 is printed on the screen. The second version still prints 2 for a=1, which is a modification traversing test case, and is successful even though no

modifications have been tested. In the third version there are two modifications. Although for a=1 the outcome of the original and modified programs differs, the test case is still not modification traversing for Modification #2. Although these simple examples show only two possible anomalies, theoretically, there are four of them: if the test case does not traverse any modifications, the output cannot be affected (S0). The other three types of the same output symptom are *coincidental correctness* (S1); *predicate-only symptom*, e.g. the modification influences (either directly or indirectly) only a predicate (S2); or the *modified statement does not affect any output* (S3).

2.2 The changed concept

In order to overcome the above mentioned shortcomings, we have to introduce a new regression testing concept and criterion. Our goal is to test each modification in such a way that - if possible - after the test traverses the location of the modification at least one output statement would be affected. Of course the original test suite might not contain tests that meet this criterion, so it is desirable to establish a method that transforms all possibly usable regression test cases. This way, errors can be revealed with a much higher probability and in an earlier stage.

In order to detect a faulty modification, the underlying test case has to

1. reach the fault (it has to be modification traversing with respect to the faulty modification)

2. the inner state of the program has to be erroneous (the behavior of the program has to differ from the expected)

3. the fault has to reach an output statement resulting in a failure (after the traversal through the erroneous statement, an output statement should be reached)

Common methods consider a test case successful if the outputs of the original and modified programs are identical. The biggest concept change is that we fulfill these requirements using a pair of test cases derived from the original test case instead of just one test and these tests should affect output statements.

2.3 The test generator framework

We build our framework around the above set of criteria. We have had a strong cooperation with an industrial partner, and the framework we present in this paper is part of their project.

In order to fulfill the first requirement, modification traversing test cases have to be selected for a given (possibly erroneous) modification. Different techniques can be found in [5, 7, 8]. Identifying modification traversing test cases requires two steps:

1. The modification needs to be detected
2. Appropriate test cases have to be identified

Of course in the case of real systems extending over possibly millions of lines of code, it is far from being trivial to identify each modification in source code. Our industrial partner has a static analyzer solution that identifies modifications within due time for millions of lines of code. Although the algorithm and the implementation is part of a commercial application (which means that it is copyrighted and cannot be published), for publicly accessible implementation the Columbus framework [11] could be used.

In order to fulfill the second requirement, we establish the following definition of reliable test cases:

Definition (Reliable test pair). Let GI=$\{I_1, I_2, ... I_M\}$ the set of input variables, $I \subseteq \{1, ...M\}$, I=$\{i_1,i_2,...i_n\}$, $J \subseteq \{1, ...M\}$, J=$\{j_1,...j_k\}$ index sets. Consider the following test cases: t_1:=<i_{i1}, i_{i2}, ... i_{in}>, t_2:=<i_{j1},i_{j2},...i_{jk}>, where i_{i1}, i_{i2},... i_{in}, i_{j1}, i_{j2}, ... i_{jk} are the values of the corresponding input variables. A pair (t_1, t_2) of test cases is a reliable test pair with respect to statement s^q (where s^q represents the q^{th} execution of statement s) if t_1 and t_2 travels along the same execution path until s^q, the result of s^q differs for t_1 and t_2, and t_1 and t_2 are 'close' to each other (for numeric values the difference should be minimized according to some metric).

Informally, the above definition states that a test case is reliable with respect to a given modification if and only if the two test cases generate the same execution path as far as s^q, both of them have an influence on at least one output statement, and the result of the output statements differ for the two test cases. Besides this, their difference should be minimized according to the following rule: the number of common variables in set I and J should be minimized, and for the common variables, the difference between them should be minimized according to some metric. For the Eucledian metric, this would be

$$\sqrt{\sum_{\gamma \in I \cap J} i_{i_\gamma} - i_{j_\gamma}}$$

As for the third requirement, we will assume that all test cases in this case reach a modification. Some of them will have an influence on the output, some of them will not. However, in both cases it is highly desirable to transform them to a test pair that meets the definition of reliability.

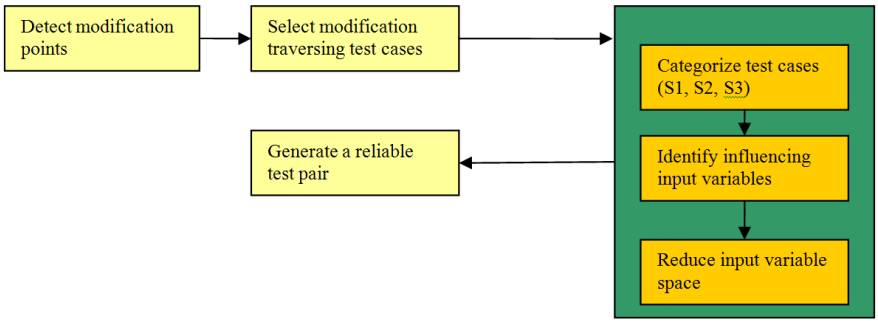
For the effective generation of test cases that meet the definition of our regression testing criterion, we need a reduced set of input variables. This is the main task we solve in this paper: according to the altered definition of reliable test cases (reliable test pair), we are to reduce the set of input variables to so-called influencing input variables. These are input variables that can be used to generate reliable test pairs based on a modification traversing test case.

Our suggested solution for finding influencing input variables is a two stage process. In the first stage the symptom (the anomaly, previously categorized as

S0-S3) is determined. In the second phase the set of influencing input variables are identified which can be used to turn the underlying test case to a reliable test pair. Both stages of the algorithm are based on *forward dynamic impact analysis*. Consequently, there is no need for large data structures in memory, and all results can be obtained on-the-fly. The high-level structure of the framework is the following:

1. Identify modifications
2. Select modification traversing test cases from the original test suite
3. For each selected test case
   a) Identify symptom (S1, S2, S3)
   b) Identify reduced set of influencing input variables
4. Generate a test pair in the reduced variable space using influencing input variables

Our main contributions are 3a and 3b. The schematic structure can be seen in the following figure (Our contributions are in the dark rectangle).



3 Related work, research directions

In this section we present related work that motivated our research. Paper [18] deals with the empirical comparison of test selection techniques. Besides the commonly used but rather desperate random and retest all techniques, minimization, dataflow and safe test selection families are also covered in that article. Our suggested approach has common properties with dataflow techniques that require that every definition-use pair that is deleted, changed, or inserted into the changed program should be tested. In [19] Harrold and Soffa select test cases that exercise the definition-use pairs affected by the modification. Our approach is quite similar with the important remark that we employ test pairs that are safer in case of predicates

and require not only the testing of the modification, but also the employment of at least one output statement.

The two most relevant papers that motivated our research are [15] and [16]. The first paper deals with slicing algorithms [4] that do not use traditional data structures, only dependence analysis to calculate program slices. The second paper categorizes regression test cases based on their effect on the program output. We compose and further simplify these approaches to reduce the set of input variables on which new reliable regression test case generation can be based.

3.1 Graph-less dynamic slicing and impact analysis

In [15] a new approach of producing dynamic program slices is proposed. The main idea of the work is to apply dependence analysis to dynamic slicing [1, 13, 12] instead of employing traditional techniques that usually require a graph-based representation and might seriously confine application possibilities due to memory consumption. The dynamic dependences that are tracked are the same as in the case of the graph representation, but instead of one huge graph, various smaller data structures are maintained.

Besides introducing alternative dependence-based methods, slicing scenarios are categorized [4] based on slicing direction, processing direction and global or demand-driven nature of the algorithm. Our impact analysis that will be presented in Section 5 relate closely to the forward, demand driven algorithm in [15]. The difference between the two approaches lies in the fact that we will not produce dynamic slices; therefore different data structures will be maintained. The reason why we do not apply dynamic slicing is that we need only a set of variables, and not a slice of the entire program.

3.2 Mutation-based regression testing

Paper [16] deals with regression test generation. The generation process has two stages: in the first stage existing test cases are categorized similarly to the previously mentioned (S0, S1, S2, S3) cases. Based on the outcome of the first stage, a new test case will be generated that effectively tests a modification. Our work is derived from that article; however, there are a few important improvements. First of all, we allow more than one modification to occur in the source code, and generate not only a test case, but a test pair. The test pair should match the changed definition of reliability.

4 Tools and notations

In order to perform dynamic impact analysis, the source code has to be carefully instrumented. During the execution of the instrumented code each traversal through a previously inserted sensor is registered. We will show that it is not necessary to maintain a log file (which again can grow huge) and log the registered traversal.

The relatively complex instrumentation that is required for real C++ code can be performed using various tools, like the Columbus framework [11]. In the following we briefly describe the used notations and information that instrumentation must provide. We are going to employ sequence-point level instrumentation, which means that sensors are inserted after each sequence point. This might imply that the trace can grow too large to handle, however, as we will see, it can be produced and processed on the fly.

For the identification of types, their fully qualified name is used

`(namespace1::namespace2::...::Class1::Class2..)`.

All typedefs have to be resolved so that their corresponding type that can be identified.

For the unique identification of variables, we use the following notation:

$\texttt{D(v, s, q, A}_v\texttt{, A}_p\texttt{)}$,

where v is the fully qualified name of the variable, s is the identification number of the statement which runs the q^{th} time, and v appears in the q^{th} run of s. Since C++ support pointer types, we have to distinguish between the memory location where the variable resides, and the memory location it points to in case it is a pointer. A_v represents the memory location of the variable, and A_p is the pointed memory location (for non-pointer typed variables, A_v and A_p are equivalent). A variable can be either global, static, local, or member variable. For the latter the

$\texttt{DD(D}_{object}\texttt{, D}_{member}\texttt{)}$

notation is used. Let's consider the example when there is a class named Foo and there is a Bar typed member variable called b. When we instantiate an object of Foo at a uniquely identified program location, both members of the DD pair can be filled in.

```
(s, q):  Foo f;
```

Let's suppose we would like to describe member b. Then the following entry will be generated:

```
DD(D(Foo::f, s, q, 0x13217ffa4, 0x13217ffa4),
      D(Bar::b, s, q, 0x1322a4c28, 0x1322a4c28))
```

Static, local or global variables can also be described this way with D_{object} being NULL in these cases.

For local variables the fully qualified name has to be integrated with the exact block number where the local variable is defined. Pointer and function pointer variables can be described similarly.

At each sequence point along the execution path we have to record the defined (DEF) and used (USE) variables, and each variable has to be identified with the above specified granularity. (Both of them contain variables that are identified using the DD notation above.)

At each function we store the exact signature along with the source code location in the call stack.

C++ rigorously defines destructors to be run deterministically when execution leaves scope, or when an explicit delete is requested. Destructors should be instrumented just like ordinary functions, but with virtual or estimated line or column number.

Another instrumentation requirement is in connection with the lazy evaluation strategy of C++. Only those variables should appear in the instrumentation log that are really used or defined. In the following we will refer to these variables as actually defined/actually used variables.

5 Forward symptom analyzer algorithm

In this section we present our forward dynamic impact analyzer algorithm.

As we have shown previously, it is possible to identify memory locations for each variable. Consequently, it is not necessary to start our algorithm from the beginning of the program, rather from the location of the first occurrence of the modified statement. However, this approach implicitly implies that the execution history of the test case is the same for the original and modified programs until the first occurrence of the modification. Unfortunately, the execution history can be too large to log and to keep in memory, and the solution would not have a significant advantage over dynamic slicing.

To overcome this difficulty, it is also possible to start the algorithm from the very beginning of the program, and employ an online algorithm that processes log entries on-the-fly. By online we mean that the instrumentation sensors write the log entries to a buffered stream, and the impact analyzer fetches them on-the-fly. Although this way a slight performance loss occurs, but we gain significant advantage in the field of storage and memory consumption, which are usually the critical factors.

Therefore, the input of the algorithm is not the execution history, rather a test case that has previously been selected. The algorithm will identify both the same-output symptom (S1/S2/S3) and the affected output statement or predicate (P).

Along the execution path, all variables have to be meticulously identified and tracked in order to easily maintain the DEF and USE sets at each sequence point. To achieve this goal, all kinds of assignment operations between variables need to be described in terms of the above notations. Originally, we treated simple (built-in) and user-defined types separately, but it turned out that it is not necessary to make distinction between the two categories. Since these two elements take the same form, we explain only the assignment of simple (local) variables. Different

cases are shown in Table 1. In the first column the possible types of underlying variables and the location of their definition is shown. The second column contains the assignment operations again with the location, while in the third column the instrumentation entry can be seen. The format of the entry is in the following form: L stands for assignment location, R for actually referenced variable, D for defined variable. Please note that the location means a sequence point.

Variable types and locations	**Assignment location/ operation**	**Instrumentation entry**
(Si,Qi) int i; (Sj,Qj) int j;	(S,Q) i= j;	*L: (S,Q)* *R: D(j, Sj, Qj, Avj, Avj)* D: *D(i, Si, Qi, Avi, Avi)*
(Si,Qi) int *i; (Sj,Qj) int *j;	(S,Q): i = j;	*L: (S,Q)* *R: D(j, Sj, Qj, Avj, Apj)* *D: D(i, Si, Qi, Avi, Apj)*
(Si,Qi) int *i; (Sj,Qj) int j;	(S,Q): i = &j;	*L: (S,Q)* *R: D(j, Sj, Qj, Avj, Avj)* *D: D(i, Si, Qi, Avi, Avj)*
(Si,Qi) int *i; (Sj,Qj) *j; int a;	(S,Q): i = j+2+a;	*L: (S,Q)* *R: D(j, Sj, Qj, Avj, Apj)* *D: D(i, Si, Qi, Avi, Apj+sizeof(*j)*(2+a))*
(Si,Qi) int *i; (Sj,Qj) int *j;	(S,Q): *i = *j;	*L: (S,Q)* *R: D(j, Sj, Qj, Avj, Apj)* *D: D(i, Si, Qi, Avi, Api)*
(Si,Qi) int *i; (Sj,Qj) int j;	(S,Q): *i = j;	*L: (S,Q)* *R: D(j, Sj, Qj, Avj, Avj)* *D: D(i, Si, Qi, Avi, Api)*
(Si,Qi) int *i;	(S, Q): i = new int;	*L: (S,Q)* *R: -* *D: D(i, Si, Qi, Avi, Api-New)*

Table 1: Assignment of primitive types

As we have previously mentioned, the assignment of primitive types can be applied to user-defined types as well, although there are some important extensions. C++ allows programmers to overload default operators, including the assignment operator. If there is no explicit user defined assignment operator in a class, then the default assignment operator (member-wise assignment) will be applied. On the other hand, if there is a custom assignment operator, its effect has to be preserved in the execution history.

At this point we have all of the necessary information to describe the intra-procedural version of the forward dynamic symptom analyzer algorithm. Later, it will be extended to its final inter-procedural form.

The input of the algorithm is the test case, the location of the modification,

and the set of variables that are defined at the modified statement. Because the set of defined variables at the modification is not known, and generating the whole execution trace is not acceptable, the values of the actually defined variables have to be calculated in a preprocessing step.

The preprocessing step requires the introduction of a set that stores variables of interest along the execution path. This set will be referred to as *Varstore*. *Varstore* is set to empty. At each variable assignment the defined variable calculated based on rules listed in is added to *Varstore*. When execution leaves the scope, all local variables will be removed. The same case holds when an explicit delete operation is requested. When we reach the first occurrence of the modified statement, the memory location of the actually defined variables can be calculated, and the set *Varstore* can be deleted. The introduction of *Varstore* is important because of the pointer typed variables of C++. Consider the example in Listing 2. All three variables (*n*, *ip*, *jp*) will be added to *Varstore*, and at the predicate we can detect that we refer to the same variable that was modified.

Listing 2 Pointer example

```
int n = 2;

int *ip = *jp = &n;

//modification:

*ip = 3; //original: *ip = 6;

...

if(*jp > 5)
```

After the initialization step we introduce a set called *Affect* for storing variables that are directly or indirectly affected by variables defined at the modified statement. The main steps of our algorithm without the preprocessing step are the following.

1. *Affect* is initialized with variables that are defined at the modified statement i_{mod} and refer to the same memory location (based on *Varstore*), starting point is set to i_{mod}.

2. From i_{mod} we traverse over statements (i_q) along the execution path according to test case T, and based on the type of this statement, we take one of the following actions:

   a) If i_q is an output statement (in other words it is not a predicate and does not define variables), and there is at least one used variable from *Affect*,

then the underlying symptom is trivially S1, and the affected statement is sq, in addition the algorithm can safely terminate.

b) If i_q is a predicate, then all actually used variables are considered. If the intersection of this set and *Affect* is not empty, then S2 is identified, and the affected statement is i_q. When predicate $i_q$is the modified statement then S2 is also identified at the modified statement.

c) If i_q is an assignment, then for each w variable used at i_q the presence of the variable in *Affect* is checked. If a w variable is in *Affect* then the defined variables in i_q are added to *Affect*. The statement defining w is marked as effective.
For each w variable in *Affect* it is checked if the w variable is defined at i_q and the last definition of w is not at i_q. If the previous condition is true then w is removed from *Affect*, moreover if the last definition is not effective the S3 is identified.

In order to successfully extend this algorithm to the inter-procedural case, we have to address parameter passing methods, and return values as well.

Method	**Modelling**
By value The same as (int i=j)	*D(j, Sj, Qj, Avj, Avj)* *D(i, Si, Qi, Avi, Avi)*
By address The same as (int *i=&j) OR (int *i=j) if j is a pointer	*D(j, Sj, Qj, Avj, Avj)* *D(i, Si, Qi, Avi, Avj)* OR *D(j, Sj, Qj, Avj, Apj)* *D(i, Si, Qi, Avi, Api)*
By reference	*D(j, Sj, Qj, Avj, Avj)* *D(i, Si, Qi, Avj, Avj)*

Table 2: Parameter passing methods and their representation

In C++ parameters can be passed via one of the following methods: by value, by address, and by reference.

Within a function any assignment to a parameter that has previously been passed by reference will not take effect outside the function, yet these assignments can alter the execution path through the return value of the function. A parameter passing by value can be modeled as an assignment to a local variable.

Parameter passing by address means the passing of pointer variables. Any assignment to a pointer variable within a function will not cause any side effect outside. Nevertheless, with the modification of the pointed memory location via dereference (*) operator side effects may occur. Passing by address can be thought of as introducing a new local pointer variable originally set to the same memory location as the pointed memory location of the actual parameter.

Since a reference can be regarded as the synonym of a memory location, any modification to a parameter passed by reference will take effect outside the function.

Algorithm 1 Symptom analyzer

Function SymptomAndLocation(T, S, P)

```
1:  Affect={i_mod.Def}
2:  S = Nothing
3:  for each statement i_q in ExecutionPath(T) from i_mod to i_last do
4:    if i_q is output and i_q.Use∩Affect ≠ ⊘ then
5:      S = S1;
6:      P = i_q;
7:      terminate;
8:    end if
9:    if i_q is predicate and (i_q.Use∩Affect ≠ ⊘ or i_q is the modified statement)
      then
10:     S = S2;
11:     P = i_q;
12:     terminate;
13:   end if
14:   if i_q is definition or function return then
15:     for each w ∈ i_q.Use do
16:       if w ∈ Affect then
17:         Affect=Affect ∪ i_q.Def
18:         mark d^w as effective
19:       end if
20:     end for
21:     for each w ∈ Affect do
22:       if w ∈ i_q.Def and d^w ≠ i_q then
23:         Affect=Affect\{w}
24:         if d^w is not effective then
25:           S=S3
26:           P=i_q
27:         end if
28:       end if
29:     end for
30:   end if
31:   if i_q is function call then
32:     AssignParams(i_q)
33:   end if
34: end for
```

The three cases are summarized in Table 2 using the previously introduced notation. Please note that j represents the actual parameter, while i is the formal parameter.

In order to complete our extension, we have to cover the handling of return values. The return statement can also be substituted by a virtual assignment.

Namely, return I; can be exchanged to the *actal_retvar=I* assignment operation. This way we can trace back the problem of return values to different parameter passing methods.

The pseudo-code of the inter-procedural algorithm (which is basically the same as the intra-procedural version) can be seen in Algorithm Listing 1.

6 Finding influencing input variables

In theory, we could either apply a backward or a forward algorithm to find those input variables that have an influence on the statement that causes ineffectiveness. However, since in the first step we developed and applied a forward method, it would be more comfortable to extend that and keep the key concept. As we will see, with very little adjustment the previous algorithm can be tuned to solve our second problem. Consequently, the two stages can share the same implementation.

The main steps of the algorithm include:

1. Finding variable definitions of input variables. An input variable can be a constant definition, data read from standard input/file, any parameter of main, or a default parameter of a function.
2. We perform the previously described impact analysis with the difference that we also keep track of effective input variables and omit those parts of the algorithm that identify symptoms. Since we are unaware of at which predicate should the execution path be altered, we monitor each predicate.

In the following we detail only the intra-procedural version of the algorithm, since the inter-procedural version remains unchanged.

- The set *Affect* will contain variables directly or indirectly affected by any input variables. We index the elements of Affect with the input variable that has an influence on that specific variable. This means that *Affect* might contain the same variable multiple times with different indices related to input variables.
- Traverse along the execution path from the first statement, and consider each statement $\mathrm{i}_{q.}$ Based on the type of $\mathrm{i}_{q.}$, the behavior of the algorithm differs.
- If $\mathrm{i}_{q.}$ is an input statement, and variable d will be assigned at $\mathrm{i}_{q.}$, then $d_{(d)} \in$ *Affect*
- If $\mathrm{i}_{q.}$ is a predicate, then only actually executed conditions should be evaluated. For each actually executed condition and for each actually used variable if *u(v)* $\in$ *Affect*, then v is added to effective input variable set of the predicate
- If $\mathrm{i}_{q.}$ is an assignment statement, then the defined variable is deleted from *Affect* with all indices. If some input variables are used, then the defined variable is added to *Affect* indexed with the input variable. If a j non-input variable is used, for which $\mathrm{j}_{(v)} \in$ *Affect*, then $\mathrm{d}_{(v)}$ is added to Affect.

- The influencing input variables can be calculated as the the union of effective input variable sets of the predicates.

Algorithm 2 Influencing inputs

Function FindInfluencingInputs(T, V)

```
1: Affect={}
2: for each statement i_q in ExecutionPath(T) from i_first to i_last do
3:   if i_q is input statment then
4:     Affect=Affect ∪ i_q.Def (iq.Def)
5:   end if
6:   if i_q is predicate then
7:     for each u ∈ i_q.ExecConditions.Use do
8:       if u_(v) ∈ Affect then
9:         V(i_q)=V(i_q) ∪ v
10:      end if
11:    end for
12:  end if
13:  if i_q is definition or function return then
14:    for each d ∈ i_q.Def do
15:      for each d_(v) ∈ Affect do
16:        Affect=Affect \ d_(v)
17:      end for
18:    end for
19:    for each j ∈ i_q.Use ∩ Affect.Indices do
20:      Affect=Affect ∪ {i_q.Def_(j)}
21:    end for
22:    for each w ∈ i_q.Use do
23:      for each w_(j) ∈ Affect do
24:        Affect=Affect ∪ {i_q.Def_(j)}
25:      end for
26:    end for
27:  end if
28:  if i_q is function call then
29:    AssignParams(i_q)
30:  end if
31: end for
```

7 Full example

In order to present the usability of the proposed solution, we show the two stages in work through a fully C++ compliant example.

The example deals with arithmetic operations. A general operation is represented as an abstract class, and all specific operations derive from this class. In this simplified source we use two operations: addition and multiplication.

```
1. #include <iostream>
2. #include <string>
3. #include <stdlib.h>

4. using namespace std;

5. namespace MathOperation
6. {
7.   class Operation //base class
8.   {
9.   public:
10.   //Pure virtual function
11.     virtual int DoOperation(int x,int y)=0;
12.     virtual string OpName()=0;
13.   };

14.   //Derived class 1
15.   class AddOperation: public Operation
16.   {
17.   public:
18.     int DoOperation(int x, int y)
                       {return x + y;}
19.     string OpName() {return "Add";}
20.   };

21.   //Derived class 2
22.   class MulOperation: public Operation
23.   {
24.   public:
25.     int DoOperation(int x, int y)
                       {return x * y;}
26.     string OpName() {return "Mul";}
27.   };
28. }

29. class CImpactAnal
30. {
31. public:
32.   void QueryMethod()
33.   {
34.     int oplocal;
35.     cin >> oplocal;
36.     this->opcode = oplocal;
37.   }
38.   void DoCalculation(int x, int y)
39.   {
40.     MathOperation::Operation *op = NULL;
41.     //original: int opcode2=opcode;
42.     int opcode2=opcode-1;
43.     if(opcode2 > 1)
          op = new MathOperation::AddOperation();
44.     else
          op = new MathOperation::MulOperation();
45.     lastOp = op;
46.     cout << " Result: " <<
          op->DoOperation(x,    y) << endl;
47.   }
48.   void LastOperation()
49.   {
        cout << "Last op: " <<
        lastOp->OpName() << endl;
        delete lastOp;
50.   }
```

```
51. private:
52.   int opcode;
53.   MathOperation::Operation *lastOp;
54. };
55. int main(int argc, char* argv[])
56. {
57.   CImpactAnal *ia = new CImpactAnal();
58.   int x = 2;
59.   int y = 3;
60.   ia->QueryMethod();
61.   ia->DoCalculation(x, y);
62.   ia->LastOperation();
63.   return 0;
64. }
```

The client code executes an arithmetic operation on two constants based on user input. According to our previous definition, the program has three input variables: *x*, *y*, and *oplocal*. Remember that an input variable is either a constant definition, data read from standard input/file, any parameter of main, or a default parameter of a function. *x* and *y* are constants (might be either parameters of the main function), *oplocal* is user input.

For the sake of clarity, we present an example with exactly one modification, which takes place at line no. 42. The modification affects an assignment statement, because *opcode2=opcode* was changed to *opcode2=opcode-1*. (In case of more than one modification, the same procedure applies until the first occurrence of the first modification.)

In the following part we review the stages of the previously introduced algorithm in order to identify test-case critical input variables.

The first section of the first stage is the preprocessing step. During this stage the aim is to identify those variables that possibly get a new value at the modified statement. In the current example the preprocessing step works as follows: The entry point of the algorithm is set to the first line of the main function (to the beginning of the program).

The *Varstore* set that stores variables of interest in the preprocessing step is initialized to empty. After traversing line no. 57, there would be two entries in the set *Varstore*. One of them represents the object pointer variable *ia*, and the other the member variable opcode of type int. Then variables x and y are added during the traverse over lines 58 and 59.

At line 60 there is a call to the *QueryMethod* member function of the *CImpactAnal* class. When we reach line no. 34, the local variable *oplocal* is also added to the *Varstore* set. At line 35 the value of *oplocal* is redefined, but its memory location is unaffected, therefore there is no need to update its entry in the *Varstore* set. At line 36 the value of class member variable *opcode* is set therefore it should be added to *Varstore*. Since *oplocal* introduced at line 34 is a local variable, and we leave the scope of this definition at line no. 37, at that point it is removed from *Varstore*. Then execution returns to line no. 61, where there is a call to *DoCalculation*. At line 40 variable op, at line 42 variable opcode2 is added to *Varstore*. At line 42 we reach the modified statement for the first time. The actually defined variables can be calculated (in this case it is only opcode2). If there are any pointer

variables in *Varstore*, we have to check whether these variables point to some used variables.

The next stage is the symptom analyzer algorithm. Variable *opcode2* will be added to the set *Affect*, and the algorithm is started from the modified statement. Since the modified statement is an assignment, the used variables should be removed from *Affect*, but *opcode* is not in Affect, so this step is not required. After that step variables that are defined at the underlying statement, but the last definition did not take place at the current statement, are removed from *Affect*. This step now does not execute because the previously mentioned conditions do not hold. Now the algorithm advances to the next statement that takes places at line 43, and is a predicate. Because the intersection of the used variables and the set *Affect* is not empty, the modification has an influence on this predicate, so S2 is identified, and the algorithm terminates.

The last step before automatic test generation is the identification of influencing input variables. This stage starts at the beginning of the program. At each input variable definition, the variable will be added to *Affect* indexed by itself. In our example that means that $\mathrm{x}_{(x)}$, $\mathrm{y}_{(y)}$, and $\mathrm{oplocal}_{(oplocal)}$ are added to *Affect* during execution. At line 35 variable *oplocal* is redefined, there is no need to change *Affect*. When we reach line 36, $\mathrm{opcode}_{(oplocal)}$ is also added to *Affect* After leaving method *QueryMethod* and entering *DoCalculation* we reach line 40 where the definition of *op* resides. There is no need to add it to *Affect* because it does not depend on any variables of *Affect*. As we previously mentioned there is an entry $\mathrm{opcode}_{(oplocal)}$ in *Affect* therefore when we reach line 42 defining *opcode2* based on *opcode*, the $\mathrm{opcode2}_{(oplocal)}$ is added to *Affect*. Now we reached the modified statement where the used variable is only *opcode2* indexed by *oplocal* therefore we can establish that the only influencing input variable is *oplocal*.

At this point we know that the identified symptom is S2, in other words the modification influences a predicate in line 43. We have also managed to identify the only input variable that influences that predicate.

The whole framework would then choose a regression test from the test suite that reaches the modification. From this test a pair of test cases will be created using a dataflow based generation algorithm. The variables that are allowed to modify are the ones that have an influence on the predicate, in our case 'oplocal'. The values of 'oplocal' should be close to each other, and close to that value where the predicate evaluates to different values (the test case should be sharp regarding the influenced predicate).

8 Conclusion

In this paper we introduced a changed concept of regression testing that was motivated by the shortcomings of existing techniques. We have shown that classical modification traversing and modification revealing regression tests are not necessarily error revealing. The new concept focuses on the reliability of test cases, it tries to assure that each test that reaches a modification has an influence on at

least one output statement.

In order to achieve this goal, existing test cases have to be filtered, and those that are usable, should be transformed. So the main task is automated test generation that is appropriate for the changed regression testing concept. Test generation can be thought of as a search in a space spanned by the input variables of the program. Unfortunately, the dimension of this search space can grow over the limits where a search can be comfortably managed. Consequently, our goal was to reduce the dimension of this search space, and find only input variables that have an influence on the modification for a given test case.

Instead of dynamic program slicing, we applied a custom dynamic impact analysis which is more appropriate for this problem. Besides operating in a forward manner, the algorithm is also superior to dynamic slicing in memory consumption, which can be a critical factor when dealing with large applications.

All of our methods have been developed to work in a C++ specific language environment. Therefore, many C++ constructs have been covered in detail including both procedural and object oriented constructs like pointers and function pointers, different parameter passing methods, classes, member and object variables and inheritance. Since C++ exposes a wider range of language constructs and gives more freedom to the programmer than most modern object oriented languages, we believe that the proposed approach can be successfully adjusted to work in other environments as well. The relative simplicity of the dynamic impact analyzer algorithm is due to a complex instrumentation mechanism. The instrumentation step is supported by the Columbus framework [11]. Since Columbus is currently not able to handle all requirements we described, first we should extend that tool. A further technical limitation is the handling of different C++ dialects. Although the ANSI standard is adopted by nearly all compilers that are used for production systems, many of them support additional features that do not comply with the standard. Consequently, the instrumentation step has to prepare for differences.

In order to fully cover the potential of C++, we also have to address issues related to the template mechanism. Technically, it is a must to be able to insert the instrumentation stage after the preprocessing step has been completed.

References

[1] A. Beszedes, T. Gergely, Zs. M. Szabo, J. Csirik, T. Gyimothy. Dynamic slicing method for maintenance of large C programs. CSMR 2001, pages 105–113.

[2] B. Korel, Ali M. Al-Yami. Automated Regression Test Generation. ISSTA 1998: 143–152

[3] B. Korel. Automated Test Data Generation for Programs with Procedures. ISSTA 1996: 209–215

[4] F. Tip. A survey of program slicing techniques. Journal of Programming Languages, 3(3):121–189, Sept. 1995.

[5] G. Rothermel and M.J. Harrold. A safe, efficient algorithm for regression test selection. Proceedings of the International Conference on Software Maintenance, pp. 358–367, September 1993.

[6] G. Rothermel and M.J. Harrold. Selecting tests and identifying test coverage requirements for modified software. Proceedings of the International Symposium on Software Testing and Analysis, pp. 169–184, August 1994.

[7] G. Rothermel and M. Harrold. Analyzing regression test selection techniques. IEEE Transactions on Software Engineering, 22(8):529-551, August 1996.

[8] G. Rothermel, M. J. Harrold, and J. Dedhia. Regression test selection for C++ software. Journal of Software Testing, Verification and Reliability, 10(2), June 2000.

[9] I. Forgacs and A Hajnal. An Applicable Test Data Generation Algorithm for Domain Errors. In Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, Clearwater Beach, Florida, March, 1998.

[10] Pargas, R. P., Harrold, M. J., and Peck, R. R. Test data generation using genetic algorithms. The Journal of Software Testing, Verification and Reliability 9 (1999), 263–282.

[11] R. Ferenc, A. Beszedes and T. Gyimothy. Extracting Facts with Columbus from C++ Code. In Tool Demonstrations of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004), Tampere, Finland, pages 4-8, March 24-26, 2004.

[12] R. Gupta, M. Harrold, M. Soffa. An approach to regression testing using slicing. Conference on Software Maintenance, 1992, pp. 299–308.

[13] T. Gyimothy, A. Beszedes, and I. Forgacs. An efficient relevant slicing method for debugging. In Proceedings of ESEC/FSE'99, number 1687 in Lecture Notes in Computer Science, pages 303-321. Springer-Verlag, Sept. 1999.

[14] W. Wong, J. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In Proceedings of the Eighth International Symposium on Software Reliability Engineering, pages 230–238, Nov. 1997. 10.

[15] A. Beszedes, T. Gergely and T. Gyimothy. Graph-Less Dynamic Dependence-Based Dynamic Slicing Algorithms. In Proceedings of the 6th IEEE Int'l Workshop on Source Code Analysis and Manipulation, pages 21–30. IEEE Computer Society, 2006.

[16] I. Forgacs, E. Takacs. Mutation-Based Regression Testing. Conference proceedings. Tenth International Software Qualiti Week 1997. San Francisco, 1997. Vol. 2. San Francisco, Software Res. Inst., 1997.

[17] N. Gupta, A. Mathur, M. Soffa. Automated Test Data Generation Using an Iterative Relaxation Method. Foundations of Software Engineering, pages 231–244, 1998.

[18] Graves, T. L., Harrold, M. J., Kim, J., Porter, A., and Rothermel, G. An empirical study of regression test selection techniques. ACM Transactions on Software Engineering and Methodology. 10, 2 pages 184–208, 2001.

[19] M. Harrold and M. Soffa. An incremental approach to unit testing during maintenance. In Proceedings of the Conference on Software Maintenance, pages 362–367, Oct. 1988.

Received 10th October 2006