This article has been accepted for publication in a future issue of a journal/proceedings, but has not been fully edited. Content may change prior to final publication. Citation information: Gossens, S., Belli, F., Beydeda, S., Dal CIN, M., View Graphs for Analysis and Testing of Programs at Different Abstraction Levels, Proc. IEEE High-Assurance Systems Eng. Symp. (HASE), pp.121-130

View Graphs for Analysis and Testing of Programs at Different Abstraction Levels

Stefan Gossens[1], Fevzi Belli[2], Sami Beydeda[3], Mario Dal Cin[4]

[1]*Framatome ANP GmbH*
Freyeslebenstrasse 1,
91058 Erlangen,
Germany
gossens@cs.fau.de

[2]*University of Paderborn*
Angew. Datentechnik
33095 Paderborn,
Germany
belli@upb.de

[3]*Bundesamt für Finanzen*
Informationsverarbeitung
Friedhofstr. 1
53225 Bonn, Germany
sami.beydeda@bff.bund.de

[4]*University Erlangen-Nuremberg*
Lehrstuhl Informatik 3
Martenstr. 3,
91058 Erlangen, Germany
dalcin@cs.fau.de

Abstract

This paper introduces view graphs, which allow representation of source code for program analysis and testing at different levels of abstraction. At a low level of abstraction, view graphs can be used for white-box analysis and testing, and at a high level of abstraction, they can be used for black-box analysis and testing. View graphs are thus an approach to integrate black-box and white-box techniques.

1. Introduction and Related Work

When abstracting a subject such as a software system, one generally concentrates on a relevant type of feature and discards details that might hinder the view of the system from this relevant, special perspective. The result is usually a model that enables, for instance, analysis and test of the system, focusing on the required features that can be:

- *behavior* of the system, given by its external communication with the environment, e.g., user, or other systems,
- *interfaces* of the system's components that enable internal communication,
- *structural characteristics*, e.g., statements, branches, begin-end blocks, or special purpose structures as used for fault-tolerant operations.

This paper introduces *view graphs* as a means for abstracting structural and behavioral information of a program from its control flow. At the lowest level of abstraction, a view graph represents the *internal structure*, i.e., the control flow of the program under test (PUT). At a higher level of abstraction, a view graph might solely describe the *external behavior* of the PUT as observed at its interfaces.

This paper also handles aspects of test termination to determine the point in time when to stop testing, based on an *adequacy criterion.* An adequacy criterion provides a measure of how effective a given set of test cases (*test suite*) is in terms of its potential to reveal faults [36]. Most conventional adequacy criteria are *coverage-oriented*, i.e., they rate the portion of the system specification (behavioral aspects) or implementation (code) that is covered by the given test case set when it is applied to exercise the PUT. The ratio of the portion of the specification or code that is covered by the given test set in relation to the uncovered portion can then be used as a decisive factor in determining the point in time at which to stop testing, i.e., to release the PUT, or to revise the PUT and/or the test set and continue testing. The approach introduced in this paper uses the concepts of view graphs and coverage testing to give a possible solution to the test termination problem:

- At a low-level of abstraction, i.e., code-level, coverage of the control flow graph (CFG) is used.
- At a high-level of abstraction, i.e., behavioral-level, coverage of the underlying view graph is used.
- Coverage of the graphs at different levels of abstraction leads to different test suites with different *potentials to detect failures* based on the different *behavioral diversities* of these test suites,

The notions "failure detection potential" and "behavioral diversity" are of decisive importance for the effectiveness of the test process. The formalization and usage of these notions for analysis and testing is one of the objectives of this paper.

Models obtained by abstraction are often state-based or event-based. Nowadays, numerous techniques and tools are available for visual representation of a model, mostly by an appropriate digraph. State-based methods have been applied to testing of system behavior for long, e.g., for conformance and protocol testing [7, 1, 29], as well as to specification of software systems and their user interface [9, 24, 30, 26, 32, 35]. They are also used for generation and selection of test cases [10, 23].

Another state-oriented group of approaches to test case generation and coverage assessment is based on model checking, e.g., the *SCR (Software Cost Reduction)* method, as described in [11], or *black-box checking*, which combines black-box testing and model checking [25]. These approaches identify negative and positive scenarios to automatically generate test cases from formal requirements specifications. A different approach is introduced in [21], which deploys methods of knowledge engineering to generate test cases, test oracles, etc., and to handle the test termination problem. Most of these approaches use some heuristic methods to cope with the state explosion problem.

The present work is intended to extend and refine these approaches by taking not only white-box testing, but also black-box testing into account. Apart from the introduction and usage of the notions "failure detection potential" and "behavioral diversity", this could be seen as the most important contribution of this work, i.e., testing a system not only with test cases that have been generated and selected based on information gained through analysis of its implementation (code), but also based on information gained through analysis of its behavior. Thus, an *integrated* view emerges that considers the complete information that is available about the program under test (see also IEC 60300-3-1 and ISO/IEC 9126).

This integrated technique brings several benefits:

- The tester has only to be familiar with the concepts underlying one single technique and needs training only for one single tool. Consequently, less effort is required for training and tool administration.
- Testing is carried out more efficiently, since an integrated technique can generate test cases covering both the specification *and* the source code at the same time. The number of necessary test cases is generally less than the one required using an integrated technique than black-box testing and white-box testing individually.

As view graphs provide convenient means to extract properties of interest from control flows, they are useful to check many features of programs, e.g., fault tolerance. Fault tolerance features of an explicit fault-tolerant program correspond to characteristic patterns of its control flow. Hence, view graphs can be used to verify the fault-tolerant behavior of such programs [14]. For example, in a program that can, by itself, detect a single design fault, there should not only be a twofold calculation of all critical values, but also comparison of these values before use or output in every possible flow of control. Structural analysis of such features offers the basis for verifying software with the goal to determine or to introduce robustness and fault tolerance.

View graphs can also be used in completely different areas. Automated generation of an abstract model of a program can strongly facilitate documentation and comprehension of software systems, particularly those having a high level of inherent complexity or are long-living legacy systems. Documentation of legacy systems often not up-to-date and programmers involved in development and maintenance are usually not available for further maintenance. In such cases, rather than trying to gather the necessary documentation directly from the source code, which is often not feasible due to the size of such systems, view graphs at various levels of abstraction and for the required feature can be automatically generated to update and analyse the documentation at considerably lower costs.

A further potential application area of view graphs is given in reengineering and regression test of legacy systems. Both the original legacy system and the newly developed system can be compared with respect to their view graphs. This comparison can be conducted at varying levels of abstraction and can thus be as detailed as required. Hence, a consistent set of view graphs extracted from a given program can serve as a basis for extensive and profound structural and behavioral analysis of the PUT.

Based on well-known concept of control flow graphs, Section 2 introduces the notions "view" and "view graph" and discusses their primary application to integrate black-box testing and white-box testing. Along with simple examples, Section 2 summarizes also the theoretical background needed to describe the approach. The kernel notion, "Behavioral diversity", is introduced in Section 3 which also explains its usage and interpretation for testing at different abstraction levels. Section 4 validates the approach by a non-trivial application and deepens the discussion on behavioral diversity and failure detection potential of test suites. Section 5 concludes the paper and gives insight into prospective, future work.

2. Towards Integration of Black-Box Testing and White-Box Testing

2.1 Integrated Black- and White-Box Testing

Few works have investigated integration of black- box testing and white-box testing. In [8] an approach is presented based on algebraic specifications that are exploited to generate test cases. The code itself is required to enable a decision whether two objects are observationally equivalent, which provably cannot be decided solely based on the specification. Another approach to integrating black-box and white-box testing is introduced in [4]. This approach relies on a graphical representation combining both specification and implementation of the PUT. The underlying combination idea is to identify definition-use pairs [27] (*def-use pairs*) based on the specification and to represent these def-use pairs in a control flow graph of the implementation. Having generated this graphical representation of the PUT, structural testing technique can be used for test case generation. An idea similar to the one in [6] is employed in [4] with *Class Specification Implementation Graphs* (CSIGs). CSIGs have a similar underlying idea for combining black-box testing and white-box testing. The CSIG construction process can automatically be carried out and incorporates a test suite reduction strategy with the benefit of scalability of costs. The construction strategy takes into account that black-box testing and white-box testing can be conducted to some extent with the same test cases, which leads to a considerable reduction of test cases.

The core idea of this paper, i.e., the extraction of view graphs from the code is different from the above mentioned ones and rather related to the technique of *slicing*, which was initially proposed by Weiser [34]. Slicing aims at generating executable partial programs from a PUT, that calculate the value of a variable of interest at a given program point in the same way as the original program. *Dynamic* slicing evaluates run-time information as to variable values from operation scenarios, traces, etc. to analyze dependencies between the program elements when creating a slice.

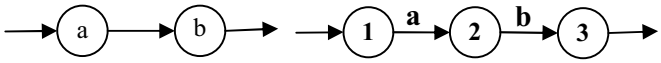
Slicing compiles the structure of certain operations (vertices of the CFG of the PUT) into a slice, taking into account a specific set of conditions concerning variables, data flow structures, run-time data, etc. These operations correspond to the generation of a view graph where the vertex set is determined upon the stated criteria. Generating view graphs, however, does not imply a specific selection criterion for the vertices to be compiled. Thus, view graph construction can be considered as a core operation of slicing, or, a weakening, or extension of the slicing concept.

2.2 Control Flow Graphs

CFGs are, similar to the flow charts and flow diagrams, being used for so long that they almost belong to folklore of the art and science of programming. Programming standards precisely describe their elements and construction rationales prescribe how to enable a universal understanding and usage. CFGs have also being used for measuring the complexity and for testing programs [20].

The nodes of a CFG are interpreted as statements of the given code. On a higher level, however, the nodes can be interpreted as states or events, whereby the edges between the nodes represent transitions from one event (or state) to another.

CFGs have an affinity with the *event sequence graphs (ESG)* which were introduced in [2]. Basically, an *event* is an externally observable phenomenon, such as an environment or user stimulus, or a system response, punctuating different stages of the system activity. It is clear that such a representation disregards the detailed internal behavior of the system and, hence, is a more abstract representation compared to, for example, a *state-transition diagram (STD),* or a *finite-state automaton* (*FSA*). A simple example of an ESG, and a related FSA, is shown in Fig. 1. Obviously, this implies that, given a need, the relevant FSA can be recovered from the ESG or, more accurately, the ESG can be refined into an "equivalent" FSA in an appropriate manner.



(a) An event sequence graph (ESG)

(b) State transition diagram of an equivalent ESG

Fig. 1: Relationship between ESG and FSA.

ESGs are comparable with the Myhill graphs [22] which are also adopted as *computation schemes* [16], or as *syntax diagrams*, e.g., as used in [18] to define the syntax of the programming language Pascal. The difference between the Myhill graphs and the ESG, and thus CFG, is that the symbols, which label the nodes of an ESG, are interpreted not merely as symbols and meta-symbols of a language, but as operations on an event set.

Similarly to ESGs, also CFGs can be used to represent different aspects of programs, from their syntax to semantics, even behavior, depending on the abstraction level of the interpretation. The arcs that connect the nodes are transitions which are to be interpreted in accordance with the interpretation of the nodes.

A given CFG can be transformed to a more abstract graph in order to model, i.e., to focus on, the relevant is-

sues by neglecting the irrelevant elements. This graph transformation is explained in the following section

2.3 View Graphs

The assumption is made that the sequences of visible system activities (inputs, outputs) as well as invisible ones (calculations, internal communication) are given by the program source code at the compile time, i.e., self-modifying systems are out of scope of the following considerations.

Different types and constellations of program constructs are specified by an appropriate CFG as depicted in Fig. 2 (code) and Fig. 3 (CFG) by a simple example.

```
read(a);
mult;
if( x )
{
   read(a);
   div;
   write(b);
}
else
{
   write(b);
   write(c);
}
add;
sub;
write(b);
```

Fig. 2: A simple program code.

The CFG in Fig. 3 represents one-to-one the structure of the program given in Fig. 2, including all of its elements that implement the functionality of the system. However, one has often to focus on certain elements of the control flow, e.g.., operations performing input and output actions. To enable this, based on the CFG, the concept of view graphs was introduced [12, 13].

Definition 1 (Control Flow Graph): Let $G=(V,E)$ be a digraph with a set of vertices V and a set of edges $E \subseteq V \times V$. G is a *control flow graph* of a given PUT iff every vertex of V corresponds to a statement of the PUT, whereby the edges represent the flow of the control between the statements.

Definition 2 (View Graph): Let $V_V \subseteq V$ be a set of vertices of special interest, which is called a *view*. Let $v_1 \rightarrow^+ v_2$ denote a path from v_1 to v_2 containing at least one edge. The *view graph* of G with respect to V_V, G_V, is defined as $G_V = (V_V, E_V)$ with

$E_V = \{(v_1,v_2) \in V_V \times V_V \mid$ There is a path $v_1 \rightarrow^+ v_2$ in G and $\forall v_3 \in V_V, v_3 \neq v_1, v_3 \neq v_2$: $v_1 \rightarrow^+ v_2 \neq v_1 \rightarrow^+ v_3 \rightarrow^+ v_2\}$.

A path in a view graph *abstracts* a path in the corresponding control flow graph by "omitting" all vertices, which do not belong to the relevant *view*.

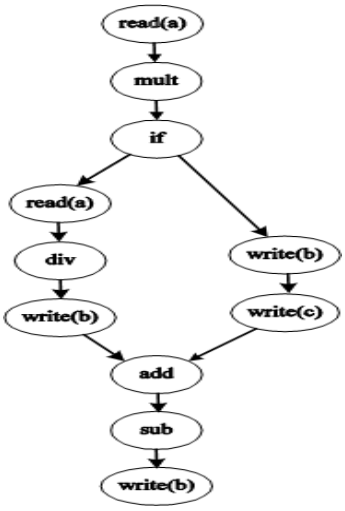


Fig. 3: The CFG of the code given in Fig. 2.

Definition 3 (Abstraction of Paths): Let $G=(V, E)$ be a control flow graph, $G_V=(V_V,E_V)$ a corresponding view graph and $p=v_1 \rightarrow \ldots \rightarrow v_n$ a path in G. Let $v_{s1},\ldots,v_{sm}$ be the vertices of p contained in V_V, ordered by indices with $1 \leq s_i < s_{i+1} \leq n$, $1 \leq i < m$. Then $v_{s1} \rightarrow \ldots \rightarrow v_{sm}$ is called the *path of G_V abstracting p*.

View graphs compactly specify all sequences of the vertices of a CFG of the relevant view.

Fig. 4 depicts the view graph of the example given in Fig. 3 with V_V representing the set of the vertices which are I/O-operations. This is the *behavioral view* as it is going to be defined in section 3.1.

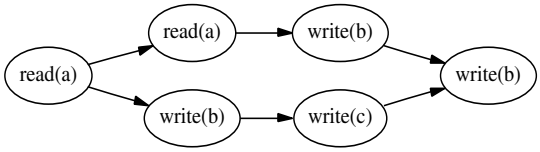


Fig. 4: View Graph.

2.3 Constructing View Graphs

View graphs can be derived from control flow graphs using a modified version of Warshall's reachability matrix algorithm [31].

Algorithm 1 (Construction of View Graphs): Let $G=(V,E)$ and $G_V=(V_V,E_V)$ be defined as above, $V=\{v_1,\ldots,v_{|V|}\}$. Let A be a $|V| \times |V|$-matrix. E_V is constructed following the scheme below.

```
A := Adjacency matrix of G
E_V := ∅
for "v_i,v_j ∈ V_V" do
 if "v_i→v_j ∈ E" then do
   A[i,j] := 0; E_V := E_V ∪ {v_i→v_j}
for i := 1 to |V| do A[i,i] := 1
for j := 1 to |V| do
 for i := 1 to |V| do
  if A[i,j] = 1
     then for k := 1 to |V| do
       if A[j,k] = 1   then   if "v_i,v_k ∈ V_V and j≠k"
                                then E_V := E_V ∪ {v_i→v_k}
                                else A[i,k] := 1
```

Algorithm 1: Construction of View Graphs

In Algorithm 1, the if-statement in the body of the loop (with the counter k) detects edges between two vertices of the view. At the same time, it is ensured that the trivial reachability of a vertex by itself (which is the "anchor" of the Warshall algorithm's calculation) does not lead to an edge in the view graph.

The algorithm has the worst case complexity $O(n^3)$ in time, with n edges in the control flow graph, but as the number of calculation steps to perform primarily depends on the density of the given CFG, view graphs can usually be calculated with less effort than this worst case.

3. View Graphs for Analysis and Testing

3.1. Behavioral Diversity

A special type of view graph, a *behavioral graph*, is constructed from a CFG when solely input/output actions are selected for V_V. These actions play an important role in black-box testing as they and only they have the potential to reveal faults in the behavior of the PUT.

The behavioral graphs reflect patterns of interaction between the PUT and its environment. However, a finite set of program executions usually covers only a small fraction of the potential paths and, thus, of the possible interaction patterns.

The *behavioral diversity* of a path set P specifies the ratio between the number of covered interaction patterns and the number of paths in P.

Definition 4 (Behavioral Diversity): Let G be a control flow graph and G_B the view graph abstracting G. Let further P_G be a set of paths in G and P_B the set of paths in G_B that abstracts the paths of P_G. Then v_{PG}, the *behavioral diversity* of P_G, is defined as $v_{PG} = |P_B|/|P_G|$.

The behavioral diversity represents a metric to determine the variety of interaction patterns covered by a set of paths.

3.2 A Quantitative Failure Detection Model

Intuitively, the potential of a series of test runs for revealing latent faults in a PUT increases with the likelihood of execution of statements of this PUT in different constellations. In other words, the greater the execution frequency and variety of the code of the PUT are, the greater is the chance for a latent fault (if any) to be detected by the test suite. Accordingly, the individual contribution of a statement to the potential to reveal a latent fault depends on its likelihood to be executed within the execution of the PUT during the test runs.

Definition 5 (Failure Detection Potential): Under the assumption that either the CFG of the PUT P has a finite set of paths or the number of paths is bounded by means of a specific assumption (about structure, length, etc. of the paths)**,** the *failure detection potential* f_a of a statement a in a program P is defined as

$$f_a = n_a/n_t$$

ESG_1

with n_a as the number of test runs that execute the statement a, and n_t as the total number of test runs.

The maximum of the failure detection probabilities of the vertices on a *path* p in the control flow, f_p^{lb}, characterizes a lower bound of the failure detection probability of the path.

Under the assumption of uniform distribution of test runs from a control flow path set $p=p_1,\ldots,p_n$, the lower bound of failure detection probability f_P^{lb} is given by

$$f_P^{lb} = 1/n \sum_{i=1}^{n} f_{pi}^{lb.}$$

Evidence was found in a case study with a large corpus of artificially generated programs in [13] that there is a nontrivial correlation between behavioral diversity and f_P^{lb}.

3.3 Testing at Different Levels of Abstraction

A behavioral graph as used for the analysis of behavioral diversity focuses on external behavior of the PUT. It considers the PUT as a black-box and abstracts from its internal states. Nevertheless, view graphs can be generated at a lower level of abstraction, too. At the lowest level, the nodes of a view graph represent the single statements of the PUT and thus the view graph encompasses information with the required granularity for, e.g., white-box testing.

Remember that view graphs are abstracted from CFGs of the PUT. The level of abstraction is determined by the selection of nodes to be focused on in the corresponding view graph. The nodes can represent a wide variety of semantic objects. Arbitrary abstraction levels may be used. V_v (Definition 2) may consist of vertices that model PUT's behavior in a high abstraction - which, e.g., may be used for black-box testing - down to block-level vertices or statement-level vertices that model the PUT's control flow, which can be used for white-box testing. Thus, view graphs are capable to assist both black-box testing and white-box testing. They can even be used for integrating both testing strategies by choosing an abstraction level in between the two ends of the possible variety.

Irrespective of the abstraction level assumed during construction, a view graph is formally, such as a ESG, a digraph. A large number of testing techniques are available to date for structural testing on the basis of graphs, e.g., [2]. Most of these testing techniques require a CFG model of the PUT and, since a view graph is syntactically an abstracted CFG, such testing techniques can also be used in the context of view graphs. One of the testing approaches in the context of CFGs is the well-known *statement coverage* criterion which requires that all statements, i.e., nodes, have to be covered by an appropriate test suite for adequate testing. The statement coverage can also be used in the context of view graphs. In this context, adequate testing requires again that all nodes be covered, irrespective of their semantic meaning. In the case of a view graph representing the PUT at a low level of abstraction, testing does not differ from that conducted using a respective CFG. However, in the case a view graph representing the PUT at a high level of abstraction, e.g., through a behavioral graph, the PUT is black-box tested. View graphs thereby allow black-box testing by structural, graph-based techniques.

4. Example

4.1. A Pocket Calculator

The assembler code given in Fig. 6 is a fragment of a pocket calculator program and implements the *keyboard scan* routine. This code is selected from a pool of a large number of programs having the same functionality developed by students during a computer architecture course at the University of Erlangen-Nuremberg (it has been adapted slightly for presentation). The students had the task to implement a simple pocket calculator. The experimental hardware environment included a microcontroller that is connected to a LCD to implement different units and functions of the pocket calculator (see Fig. 5).

Fig. 5: Pocket Calculator Hardware.

The keyboard is merely a collection of seven wires, which are arranged crosswise in three columns and four rows with contact points at the intersections. To detect any pressed key, voltage is attached to the column wires one after another which is done by setting bits in an processor I/O port. The row wires are attached to the same I/O port.[1] Thus, knowing which of the column is currently under voltage, it is possible to detect the pressed key. All I/Os in this process are performed by read or write access to I/O port *a* of the microcontroller. The lower four bits of the port mirror the keyboard rows. The output variable `key` holds the numerical code of the pressed key (the numeric key value, 10 for "-", 11 for "+") upon leaving the routine. If an invalid key combination (i.e., more than one key in a column) has been pressed, the column is expected to be returned in the upper four bits of `key`, and the number of keys pressed shall be contained in the lower four bits.[2]

The port access actions and write actions to `key` semantically constitute the I/O actions of the program. Their structure corresponds to the behavioral graph of Fig. 7, where "<a" means a read from port `a`, ">a" represents a write to port `a`, and ">k" means a write to the variable `key` that holds the return value (i.e., the code of the detected key). I/O code elements are commented with their corresponding vertex numbers.

```
LDX #%01000000
STX COLUMN
STX porta // 1
LDA porta // 2
AND #%00001111
```

[1] Ports in the used processor are memory mapped and unbuffered; a read does usually not return a value written to the port, but one provided by the external hardware at the time of reading.

[2] Check of multiple key presses in *different* columns is omitted here for reasons of simplicity.

```
                BEQ try_column2
try_column1:    LDX #10
                STX KEY // 3
                CMP #%00001000
                BEQ Key_END
                LDX #7
                STX KEY // 5
                CMP #%00000100
                BEQ Key_END
                LDX #4
                STX KEY // 6
                CMP #%00000010
                BEQ Key_END
                LDX #1
                STX KEY // 7
                CMP #%00000001
                BEQ Key_END
                JMP error
try_column2:    LDX #%00100000
                STX COLUMN
                STX porta // 4
                LDA porta // 8
                AND #%00001111
                BEQ try_column3
                LDX #0
                STX KEY // 9
                CMP #%00001000
                BEQ Key_END
                LDX #8
                STX KEY // 11
                CMP #%00000100
                BEQ Key_END
                LDX #5
                STX KEY // 12
                CMP #%00000010
                BEQ Key_END
                LDX #2
                STX KEY // 13
                CMP #%00000001
                BEQ Key_END
                JMP error
try_column3:    LDX #%00010000
                STX COLUMN
                STX porta // 10
                LDA porta // 14
                AND #%00001111
                BEQ Key_END
                LDX #11
                STX KEY // 15
                CMP #%00001000
                BEQ Key_END
                LDX #9
                STX KEY // 16
                CMP #%00000100
                BEQ Key_END
                LDX #6
                STX KEY // 17
                CMP #%00000010
                BEQ Key_END
                LDX #3
                STX KEY // 18
                CMP #%00000001
                BEQ Key_END
error:          CMP #%00000011
                BEQ two_keys
                CMP #%00000110
                BEQ two_keys
                CMP #%00000101
                BEQ two_keys
                … // compare to combinations
                  // of two keys
three_keys:     LDA #3
                JMP error_code
two_keys:       LDA #2
error_code:     ORA COLUMN
                STX KEY // 19
Key_END:        RTS
```

Fig. 6: Pocket Calculator Program.

One can clearly identify the three program regions for the three columns in the routine's behavioral graph. In these regions, the write instruction to port a sets the stage for the detection process by attaching voltage to a column, the subsequent read detects the pressed key(s). The obtained value is interpreted and one of the four alternative values is written to the output variable key (note that there may be no key pressed which is indicated by the lowermost read from port a being a possible terminal state). The lower part of the program calculates values of key signaling an error. There are a total of 12! = 479,001,600 possible combinations of pressing keys on the keyboard. However, from the program's point of view, there are 256^3 possible test cases, as the (maximum) three subsequent reads of the eight bit valued port *a* may render elements from that input space.

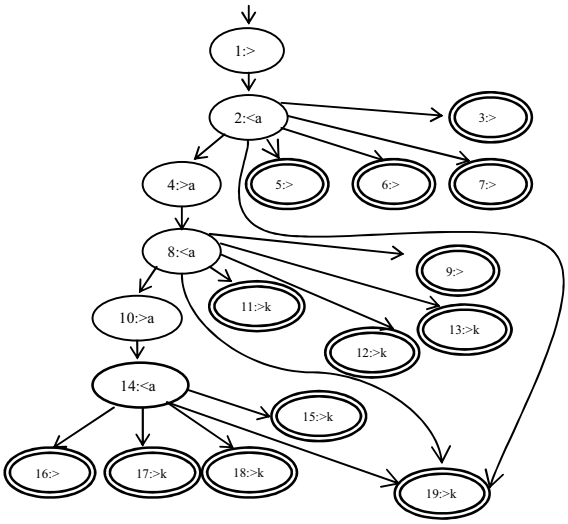


Fig. 7: Behavioral Graph of the Pocket Calculator.

4.2 Determining the Behavioral Diversity of the Example

For the code given in Fig. 5, the behavioral diversity can be determined using definition 4. It is assumed that the test case set

$$T := \{1, 147, 47, 52, 8, 85, 83, 9\}$$

is given whereas the elements of T represent a collection of simultaneously pressed keys. These test cases define eight alternative paths through the CFG and four alternative paths through the behavioral graph. Only the four inputs 1, 8, 83 and 9 produce non-erroneous outputs, which are 1, 8, and 9 (key combinations 8 and 83 correspond to the same path in the program and thus yield the same result[3]), the rest of the selected combinations result in error values. We have seven different paths through the control flow graph: Three for the different "legal" results, one for "147", one for "47", and one for "52" and "85" each. They map to five paths in the behavioral graph, as

[3] Multiple key presses in different columns are not detected (see above).

the different control flow paths “147” and “47” trigger an identical I/O action sequence. Thus, the resulting behavioral diversity is

$v = 5/7$.

4.3 Benefits of Behavioral Graphs for Testing

Testing is an activity that usually has to cope with limited resources, especially with limited time and thus a limited number of tests to be performed. Therefore, it is crucial to focus the activity to test cases in which a good detection of failures is expected. Conventionally, CFG-based coverage metrics, such as statement and branch coverage, do not offer much assistance in selecting the significant test cases in this regard. They rather serve as stopping criteria or monitors to justify the test progress regarding the *quantity* of tests. Achieving a high level, e.g., 99%, of branch coverage in testing is supposed to impose a high level of confidence in the testing process. However, the performed tests might still not cover a very low level, e.g., 1%, of the program code that contains, crucial, i.e., more relevant, interactions which might expectedly hide much of the potential failures. Judging test cases using coverage criteria applied to behavioral graphs, however, gives clues about the *quality*, *i.e.,* effectiveness of the selected test cases to detect faults – in accordance with the goal to cover as many interaction patterns of the program as possible. Empirical studies suggest that the more of the behavioral graph is covered (and thus, the higher the behavioral diversity and failure detection potential of the selected test suite are), the higher is the expected rate of failure detection [13, 15].

Black-box testing by means of structural, graphical model-based testing technique is seldom studied in the literature. Examples of other approaches to structural, graph-based black-box testing have been given above. Assume that the implementation of the pocket calculator from the last section is to be black-box tested using popular coverage criteria as a metric of quantity. The same methodology can be applied to view graphs. All vertices – or even better, paths – of the behavioral graph (instead of the CFG) then have to be covered by proper test cases. This coverage leads to a metric and a stopping criterion in terms of spread of alternative interactions. As an example, consider the following test suite for the pocket calculator

T :={-, +, 0, ..., 9, 14},

which can be interpreted as pressing each of the calculator’s keys individually plus an additional test case (pressing 1 and 4 simultaneously). This triggers a failure. Though this is obviously a simple example, applications with complex I/O patterns may not always enable this kind of insight without formal analysis - except an advanced expert level of knowledge of the user is available.

Applying coverage criteria to behavioral graphs – instead of to CFG – helps to better spread the limited resources of testing across the interaction patterns of a program in a controlled way. Even a careful construction of test cases can fail to include just the one “important” case (interaction pattern) that will systematically break the PUT. However, there are no guarantees to find existing errors, even if a test case is constructed for each possible pattern. While not being a general solution to guide test case selection, using evaluation techniques based on behavioral graphs provides a measure to assess how much of the interaction capabilities have already been tested instead of how much of the code.

5. Conclusions and Further Work

This paper introduced an approach to visualize program code at various levels of abstraction. For this purpose, the control flow graph (CFG) of the program is (i) automatically generated and (ii) transformed into a graph of a specific level of abstraction. This graph is called view graph. Whereas the CFG is exploited to generate white-box tests, the view graph is used to generate black-box tests.

The primary objective of this paper is to apply the view graphs to the integration problem of black-box testing and white-box testing. A strong correlation was found in [13, 15] between minimum failure detection capabilities of path sets and behavioral diversity. The results are promising and encourage selecting test data that correspond to control flow paths with a high value of behavioral diversity. However, further research is needed to strengthen this finding, which entails applying the approach to industrial size programs and constructing appropriate failure detection models. The approach also offers an open platform for evaluations of other metrics or criteria.

View graphs possess further characteristics which enable their application to testing of commercial-of-the-shelf (COTS) components. A problem in the context of COTS components is that the exchange of information between the *component provider* and the *component user* is strongly limited by the former to protect commercial interests [5]. Such a limited exchange of information can cause various problems and one approach of avoiding these conflicts is to augment the component with meta-information describing how to check the program, e.g., by self-testing of its behavior [3, 5]. However, meta-information might not properly describe the component due to inconsistencies in self-test generation. The generation process of view graphs, as introduced in this paper, guarantees the conformance of the meta-information with the code of the COTS, without revealing this code to the user.

A technique to compare view graphs to reference structures of safety architectures using model checking techniques was proposed in [14]. Similarly, comparison of view graphs of different versions can be applied to regression testing. Regression testing generally aims at verifying that modifications have not caused unintended effects and that the system still complies with its originally specified requirements. Following the idea in [28] the view graphs of the original and the modified version can be compared to detect those parts of the system which need to be tested. Again, this can be conducted at various levels of abstraction and for various features. At the lowest level of abstraction, such a comparison of view graphs corresponds to the selective regression testing technique [28].

References

1. Aho, A. V., Dahbura, A. T., Lee, D., Uyar, M.Ü., 1991. An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours, IEEE Trans. Comm. 39, pp. 1604-1615.
2. Belli, F., 2001. Finite-State Testing and Analysis of Graphical User Interfaces, Proc. 12th ISSRE, pp. 34-43
3. Beydeda, S., 2003, The Self-Testing COTS Components (STECC) Method, Dissertation, University of Leipzig.
4. Beydeda, S., Gruhn, V., 2000. Integrating white- and black-box techniques for class-level testing object-oriented prototypes. In SEA Software Engineering and Applications Conference, pp. 23-28. IASTED/ACTA Press.
5. Beydeda, S., Gruhn, V., 2003. State of the art in testing components. In International Conference on Quality Software (QSIC), pp. 146-153. IEEE Computer Society Press.
6. Beydeda, S., Gruhn, V., Stachorski, M., 2001. A graphical representation of classes for integrated black- and white-box testing. In International Conference on Software Maintenance (ICSM), pp. 706-715. IEEE Computer Society Press.
7. Bochmann, G. V., Petrenko, A.. 1994. Protocol Testing: Review of Methods and Relevance for Software Testing, Softw. Eng. Notes, ACM SIGSOFT, pp. 109-124.
8. Chen, H. Y., Tse, T. H., Chan,F. T. Chan, Chen, T. Y. 1998. In black and white: an integrated approach to class-level testing of object-oriented programs. ACM Transactions on Software Engineering and Methodology, 7(3), pp. 250-295.
9. Chow, T. S., 1978. Testing Software Designed Modeled by Finite-State Machines, IEEE Trans. Softw. Eng. 4, pp. 178-187.
10. Fujiwara, S., Bochmann, G.V., Khendek, F., Amalou, M., 1991. Test Selection Based on Finite State Models, IEEE Trans. on Softw. Eng.17/6, pp. 591-603.
11. Gargantini, A., Heitmeyer, C., 1999. Using Model Checking to Generate Tests from Requirements Specification, Proc. ESEC/FSE '99, ACM SIGSOFT, pp. 146-162.
12. Gossens, S., 2002, Enhancing System Validation with Behavioral Types, Proc. HASE, IEEE Comp. Press, pp. 201-208
13. Gossens, S., 2004. Sichtgraphen: Ein Konzept zur gezielten Untersuchung von Kontrollflussstrukturen, Dissertation, University of Erlangen-Nuremberg.
14. Gossens, S., Dal Cin M, 2004. Structural Analysis of explicit fault-tolerant programs. In Proc. High-Assurance Systems Engineering Symposium pages 89-96. IEEE Computer Society Press.
15. Gossens, S., Dal Cin M., 2004. A View-based Control Flow Metric. In Proc. COMPSAC Workshops and Fast Abstracts, pp. 26-28. IEEE Computer Society Press.
16. Ianow, J.I., 1958. Logic Schemes of Algorithms", Problems of Cybernetics I (in Russian), pp. 87-144.
17. IEEE Software Engineering, 1991. Standards Collection.
18. K. Jensen, N. Wirth, 1974. Pascal, User Manual and Report, Springer-Verlag, New York.
19. Kramkar, M., 1995. An Overview and Comparative Classification of Program Slicing Techniques, J. Systems Software, pp. 197-214.
20. McCabe, T. J. 1966. A Complexity Measure, IEEE TSE 2/4, pp. 308-320.
21. Memon, A. M., Pollack, M. E. and Soffa, M. L., 2000. Automated Test Oracles for GUIs, SIGSOFT 2000, pp. 30-39.
22. Myhill, J., 1957. Finite Automata and the Representation of Events, Wright Air Devel. Command, TR 57-624, pp. 112-137.
23. Offutt, J., Shaoying, L., Abdurazik, A., Ammann, P. 2003. Generating Test Data From State-Based Specifications, The Journal of Software Testing, Verification and Reliability, 13(1), pp. 25-53.
24. Parnas, D.L., 1969. On the Use of Transition Diagrams in the Design of User Interface for an Interactive Computer System, Proc. 24th ACM Nat'l. Conf., pp. 379-385.

25. Peled, D.A., 2001. Software Reliability Methods, Texts in Computer Science, Springer-Verlag, New York.

26. Raju, S.C.V., Shaw, A. 1994. A Prototyping Environment for Specifying, Executing and Checking Communicating Real-Time State Machines, Software - Practice and Experience 24/2, pp. 175-195.

27. Rapps, S., Weyuker, E. J., 1985. Selecting software test data using data flow information. IEEE Transactions on Software Engineering, 11(4), pp. 367-375.

28. Rothermel, G., Harrold; M. J., 1997. A safe, efficient regression test selection technique. ACM Transactions on Software Engineering and Methodology, 6 (2), pp. 173-210.

29. Sarikaya, B., 1989. Conformance Testing: Architectures and Test Sequences, Computer Networks and ISDN Systems 17, North-Holland, pp. 111-126.

30. Shaw, A.C., 1980. Software Specification Languages Based on Regular Expressions, in Software Development Tools, ed. W.E. Riddle, R.E. Fairley, Springer-Verlag, Berlin, pp. 148-176.

31. Sedgewick, R., 1991. Algorithms, Addison-Wesley.

32. Shehady, R. K. and Siewiorek, D. P., 1997. A Method to Automate User Interface Testing Using Finite State Machines, Proc. FTCS-27, pp. 80-88.

33. Tip, F., 1995. A Survey of Program Slicing Techniques, J. Programming Languages, pp. 121-189.

34. Weiser, M., 1982. Programmers Uses Slices When Debugging, Commun. ACM 25, pp. 446-452.

35. White, L. and Almezen, H., 2000. Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences, Proc. 11th ISSRE, pp. 110-119.

36. Zhu, H., Hall, P.A.V., May, J.H.R., 1997. Unit Test Coverage and Adequacy, ACM Comp. Surveys, pp. 366-427.