Int. J. Appl. Math. Comput. Sci., 2010, Vol. 20, No. 4, 739–749
DOI: 10.2478/v10006-010-0056-9

CONTROL FLOW GRAPHS AND CODE COVERAGE

ROBERT GOLD

Faculty of Electrical Engineering and Computer Science
Ingolstadt University of Applied Sciences, Esplanade 10, D-85049 Ingolstadt, Germany
e-mail: robert.gold@haw-ingolstadt.de

The control flow of programs can be represented by directed graphs. In this paper we provide a uniform and detailed formal basis for control flow graphs combining known definitions and results with new aspects. Two graph reductions are defined using only syntactical information about the graphs, but no semantical information about the represented programs. We prove some properties of reduced graphs and also about the paths in reduced graphs. Based on graphs, we define statement coverage and branch coverage such that coverage notions correspond to node coverage, and edge coverage, respectively.

Keywords: directed graph, control flow graph, graph reduction, software testing, statement coverage, branch coverage.

1. Introduction

Control flow graphs or program graphs that represent the control flow of programs are widely used in the analysis of software and have been studied for many years (Jalote, 2005; Kosaraju, 1973; McCabe, 1976; Paige, 1977; Rapps and Weyuker, 1982; Tan, 2006; White, 1981; Zhu *et al*., 1997). The nodes of a control flow graph are statements of the program and the edges represent the control flow between the statements. The approaches differ with respect to the handling of branching and the merging of branches, and the representation of segments of statements that are always executed together.

The aim of this paper is to discuss control flow graphs and their application to software testing detailed. We define control flow graphs with one node for each statement and segment graphs where we replace segments by single nodes. We use the definition given by Rapps and Weyuker (1982) as well as Jalote (2005), and show structural properties and results. Our aim is to prove that for both types of control flow graphs node coverage corresponds to the notion of statement coverage used in software testing (Jalote, 2005; Sommerville, 2004; Zhu *et al*., 1997). For graph reduction we will use only (syntactical) information about the graphs, but no (semantical) information about the represented programs. Therefore, our reduction can be applied to all directed graphs, not only to control flow graphs. Furthermore, we define control flow graphs not only for single functions, but also for C-like programs consisting of sets of functions.

By a further reduction, using the definition of Paige (1977), we introduce decision graphs with one edge between decision nodes, i.e., one edge for each branch in a program, and therefore edge coverage corresponds to branch coverage used in software testing. As above, the reduction of graphs to decision graphs can be applied to all directed graphs, not only to control flow graphs.

The rest of the paper is organized as follows. Basic definitions regarding graphs and the definition of control flow graphs follow in Section 2. Segment graphs and statement coverage are described in Section 3. Decision graphs and branch coverage are introduced in Section 4. Section 5 discusses other works and contains conclusions.

2. Basic definitions

We start with necessary definitions about graphs.

Definition 1. A *directed graph with multiple edges* is a pair $G = (N, E)$ consisting of a finite set N of *nodes* and a finite set E of *edges* with $N \cap E = \emptyset$, together with functions start and $\mathrm{end} : E \to N$ that associate a *start node* and an *end node*, respectively, with each edge. With $|(n, n')|$ for two nodes $n, n' \in N$, we denote the number of edges with start node n and end node n'. For a node $n \in N$, the sets $\mathrm{pre}(n) = \{n' \mid \exists\, e \in E : \mathrm{start}(e) = n', \mathrm{end}(e) = n\}$ and $\mathrm{post}(n) = \{n' \mid \exists\, e \in E : \mathrm{start}(e) = n, \mathrm{end}(e) = n'\}$ are called *preset* and *postset* of n, respectively. Nodes with an empty preset or an empty postset are called *entry nodes* and *exit nodes* of the graph, respectively. An edge e with $\mathrm{start}(e) = \mathrm{end}(e)$ is called a

```
void Search(int arr[], int key,
            int *found, int *index)
{
  int i = 0;
  int b;

  *found = 0;

  while (i < N)
  {
    if (b = isabsequal(arr[i], key))
    {
      *found = b;
      *index = i;
      return;
    }

    i++;
  }
}
```

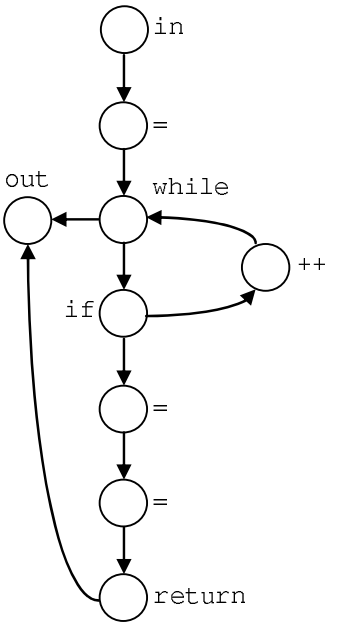

Fig. 1. Function `Search` and its control flow graph.

cycle. Two graphs $G_1 = (N_1, E_1)$, $G_2 = (N_2, E_2)$ are called *disjoint* if $N_1 \cap N_2 = \emptyset$ and $E_1 \cap E_2 = \emptyset$. A *path* d is a finite sequence of edges $e_1\, e_2\, \ldots e_k$ such that $\mathrm{end}(e_i) = \mathrm{start}(e_{i+1})$ for $i = 1, \ldots, k-1$. The start node of the first edge e_1 is called the *start node* of d, the end node of the last edge e_k—the *end node* of d. A *prefix* of a path $d = e_1\, e_2\, \ldots e_k$ is an initial part $d' = e_1\, e_2\, \ldots e_i$ of the path for $i \in \{0, \ldots, k\}$. We denote by $d' \leq d$ a prefix. For a path d, the set $\mathrm{prefix}(d) = \{d' \mid d' \leq d\}$, and for a set D of paths, the set $\mathrm{prefix}(D) = \bigcup_{d \in D} \mathrm{prefix}(d)$ are the sets of prefixes of d and D, respectively. A path starting with an entry node is called an *S-path*. An S-path that ends in an exit node is called a *complete path*. By $N(d)$ and $E(d)$ we denote the set of nodes and edges, respectively, that are contained in the path d.

In the remainder of Section 2 and in Section 3 we will not allow more than one edge with equal start nodes and equal end nodes (i.e., $\forall\, n, n' \in N : \ |(n, n')| \leq 1$), and we will call a graph with this property simply a *directed graph*. In this case, an edge e can be written as $(\mathrm{start}(e), \mathrm{end}(e)) \subseteq N \times N$. A path is then a sequence $(n_1, n_2)(n_2, n_3) \ldots (n_{k-1}, n_k)$. We can briefly represent a path as a sequence of nodes $n_1\, n_2\, \ldots n_k$. In Section 4 we will need directed graphs with multiple edges.

Each statement in a function written in a programming language will be a node in the control flow graph, the edges representing the control flow between statements. An entry and an exit node will be added as unique entry and exit points of the function. Statements in C are function calls, assignments and other expressions with a semicolon, `return-`, `break-`, `continue-`, `goto-`, `if-`, `switch-`, `do-while-`, `for-`, `while-`statements and the null statement.

Definition 2. The *control flow graph* $G_f = (N, E)$ of a function f has one node $n_a \in N$ for each statement a in f and two additional nodes n_{in}, n_{out}. We add an edge $(n_a, n_{a'})$ if the statement a' is executed immediately after the statement a. For the first statement a_1 in the function, we introduce an edge $(n_{\mathrm{in}}, n_{a_1})$. Furthermore, we add edges $(n_{a'}, n_{\mathrm{out}})$ for each node $n_{a'}$ that is associated to a statement a', after which the control flow leaves the function because of a return-statement or the right brace that terminates the function. The control flow graph of an empty function, i.e., a function without any statements consists of $N = \{n_{\mathrm{in}}, n_{\mathrm{out}}\}$ and $E = \{(n_{\mathrm{in}}, n_{\mathrm{out}})\}$.

The node n_{in} is the only entry node and the node n_{out} the only exit node of the control flow graph. Note that the control flow graph G_f is a graph where each node (except n_{in} and n_{out}) corresponds to one statement in the function f. Figure 1 shows an example where we labelled the nodes with the types of the statements or with "in", "out" for better readability. The called function `isabsequal` checks whether its parameters have equal absolute values. `N` is a defined constant denoting the length of the array.

3. Segment graphs

Segments are sets of consecutive nodes that can be replaced by single nodes.

Definition 3. A non-empty set $S \subseteq N$ of nodes of a directed graph $G = (N, E)$ is called a *segment* (Fig. 2)

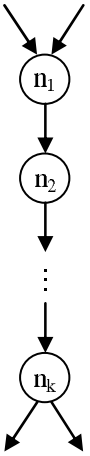

Fig. 2. Segment of a directed graph.

if and only if there exists an order $\langle n_1, n_2, \ldots, n_k \rangle$ of the elements of S such that

$$\mathrm{post}(n_i) = \{n_{i+1}\},\ \mathrm{pre}(n_{i+1}) = \{n_i\}$$

for $i = 1, \ldots, k-1$. A segment S is called *maximal* if none of supersets $S' \supset S$ are segments.

Before we show that the set of the maximal segments of a graph is a partition of the nodes, we need a lemma that provides some properties of segments.

Lemma 1. *Let $G = (N, E)$ be a directed graph and let S, S' be non-empty subsets of N.*

1. *$|S| = 1 \Rightarrow$ S is a segment.*
2. *Let S be a segment with $S' \subseteq S$ and let $S = \langle n_1, n_2, \ldots, n_k \rangle$ be an order of the nodes with $\mathrm{post}(n_i) = \{n_{i+1}\}$, $\mathrm{pre}(n_{i+1}) = \{n_i\}$ for $i = 1, \ldots, k-1$. Then*

   S' is a segment $\Leftrightarrow$
   there exist indices j, l with $1 \le j \le l \le k$
   such that
   $S' = \langle n_j, n_{j+1}, \ldots, n_l \rangle$ or
   $S' = \langle n_l, n_{l+1}, \ldots, n_k, n_1, n_2, \ldots, n_j \rangle$
   with $\mathrm{post}(n_k) = \{n_1\}$, $\mathrm{pre}(n_1) = \{n_k\}$.

3. *Let S, S' be segments and let S be maximal. Then $S' \subseteq S$ or $S \cap S' = \emptyset$.*

Proof. We give only an outline of the proof. The first part is obvious. The direction "$\Leftarrow$" of the second part follows directly from the definition. The proof of direction "$\Rightarrow$" of the second part can be provided by induction over $|S'|$ and has two cases: the first one where the segment S has unique start and end nodes n_1, n_k and therefore the subsegment S' is a simple subsequence (Fig. 3), and the second one where the segment is a loop of nodes with one-elementary pre- and postsets and therefore the subsegment S' can start anywhere in S and even span from the end node n_k to the start node n_1 (Fig. 4). For the proof of the third part we also use induction over $|S'|$. The case $|S'| = 1$ is obvious. Now assume that $\tilde{S} \subseteq S$ or $S \cap \tilde{S} = \emptyset$ holds for all segments $\tilde{S}$ with $|\tilde{S}| = m - 1$, especially for the segment $\tilde{S}$ that we get by clipping the last element n'_m of S'. If $\tilde{S} \subseteq S$, it follows from the second part that $\tilde{S}$ is a subsequence of S. If the clipped node n'_m is not in S, then n'_{m-1} has a postset with two elements and must be the last node in S. But then S can be lengthened by n'_m and S is not maximal. Therefore, $n'_m \in S$ and $S' \subseteq S$. Now let $S \cap \tilde{S} = \emptyset$ but $n'_m \in S$. If n'_m is the first node in S, both segments can be concatenated and S is not maximal. Otherwise the preset of n'_m has two elements and S is not a segment. Therefore $n'_m \notin S$ and $S \cap S' = \emptyset$. ■

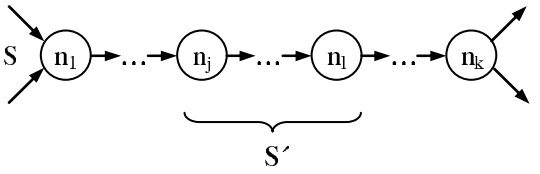

Fig. 3. Segment S in the case $S' = \langle n_j, n_{j+1}, \ldots, n_l \rangle$.

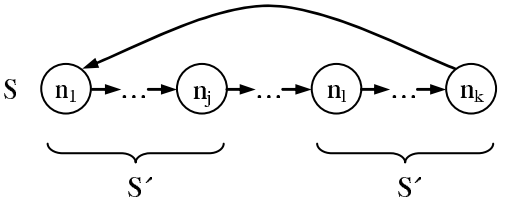

Fig. 4. Segment S in the case $S' = \langle n_l, n_{l+1}, \ldots, n_k, n_1, n_2, \ldots, n_j \rangle$.

A segment of the type shown in Fig. 4 and used in Lemma 1 is an isolated loop, thus unreachable from entry nodes, and can occur in control flow graphs only in programs with an unreachable code like in the graph for a function with body `x = 0; return; label: x++; goto label;`. The detection of an unreachable code plays an important role in software testing since such a code indicates a programming error and complete statement coverage cannot be achieved if an unreachable code exists.

Theorem 1. *Let $G = (N, E)$ be a directed graph. The set of the maximal segments of G is a partition of N.*

Proof. Let S and S' be two maximal segments. From Part 3 of Lemma 1 it follows that $S' \subseteq S$ or $S \cap S' = \emptyset$. If $S' \subset S$, then S' is not maximal. Therefore, $S' = S$

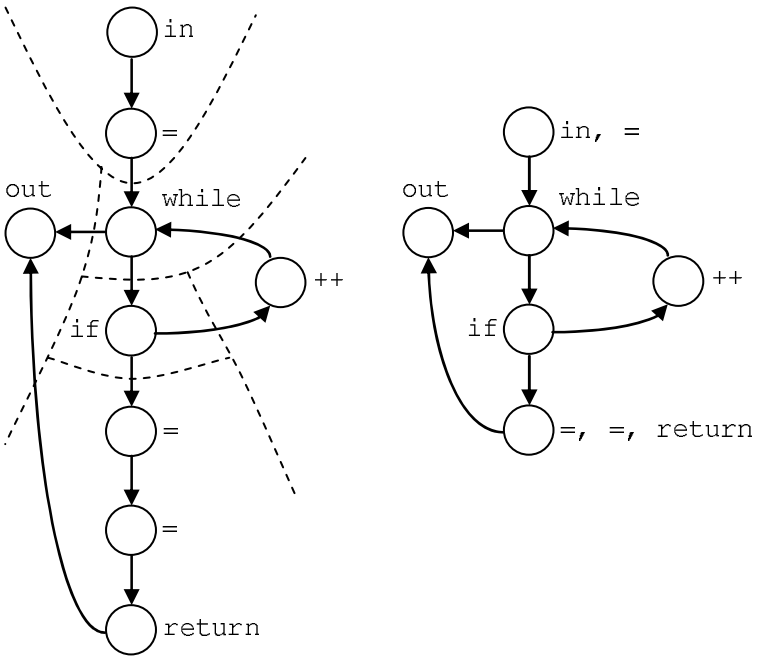

Fig. 5. Control flow graph and segment graph of function `Search`.

or $S \cap S' = \emptyset$. This means that two different maximal segments are disjoint. Let $n \in N$. The set $S_1 = \{n\}$, is according to Part 1 of Lemma 1, a segment. If S_1 is not maximal, a segment S_2 with $S_1 \subset S_2$ exists. If S_2 is not maximal, a third segment S_3 with $S_2 \subset S_3$ exists. Since N is finite, there is a maximal segment S_i with $n \in S_1 \subset S_2 \subset \ldots \subset S_i$. This means that each node is an element of a maximal segment. ■

By substitution of maximal segments with single nodes, we reduce graphs to segment graphs.

Definition 4. Let $G = (N, E)$ be a directed graph. The *segment graph* $G_r = (N_r, E_r)$ consists of the set of nodes

$$N_r = \{n_S \mid S \text{ is a maximal segment in } G\}$$

and the set of edges E_r with

$$\begin{aligned}(n_S, n_{S'}) \in E_r \quad \Leftrightarrow \quad & S \neq S' \text{ and there exist} \\ & n \in S, n' \in S' \\ & \text{with } (n, n') \in E \text{ or} \\ & S = S' \text{ and there exists a path} \\ & n_1\, n_2\, \ldots n_k\, n_1 \text{ in } G \text{ with} \\ & \{n_1, n_2 \ldots, n_k\} \subseteq S.\end{aligned}$$

The second part of the definition of the set of edges makes sure that loops in segments get cycles in the segment graphs. Figure 5 shows the segment graph of the function `Search`. Figure 6 gives an example of a segment graph with a cycle.

Not only graphs but also paths can be reduced.

Definition 5. Let $G = (N, E)$ be a directed graph and d a path in G. Let $d = d_1\, d_2 \ldots d_l$ be partitioned into subpaths such that in each subpath d_i only nodes of one maximal segment S_i and in two consecutive subpaths only nodes of two different maximal segments occur. Let each subpath d_i be partitioned once more in paths $d_{i1}\, d_{i2} \ldots d_{im_i}$ such that the first nodes in all paths d_{i1}, d_{i2}, $\ldots$, d_{im_i} are equal and occur only at the first position in these paths. The *segment path* d_r starts with $n_{S_1}^{m_1}\, n_{S_2}^{m_2} \ldots n_{S_{l-1}}^{m_{l-1}}$, where n_S is the node that is associated to a maximal segment S in the segment graph according to Definition 4. We append $n_{S_l}^{m_l}$ if and only if d_l contains all nodes of S_l.

Note that, in the case when we have $l = 1$ and d_l does not contain all nodes of S_l, the segment path is empty. Let the nodes of the control flow graph and the segment graph of the function `Pot` (Fig. 6) be $n_1, n_2, \ldots, n_6$ and n'_1, n'_2, n'_3 (from top to bottom), and let $d = n_1\, n_2\, n_3\, n_4\, n_2\, n_3\, n_4\, n_2\, n_3\ n_4\, n_5$ be a path in the control flow graph. The decomposition is then $d = d_{11}\, d_{21}\, d_{22}\, d_{23}\, d_{31}$ with $d_{11} = n_1$, $d_{21} = d_{22} = d_{23} = n_2\, n_3\, n_4$, $d_{31} = n_5$. Therefore, the segment path d_r is $n'_1\, n'^3_2$ since the last subpath d_3 does not run through all nodes of the third segment. But the path $d\, n_6$ has the segment path $n'_1\, n'^3_2\, n'_3$.

Theorem 2. *Let $G = (N, E)$ be a directed graph and d a path in G. The segment path d_r is then a path in the segment graph G_r.*

Proof. For there to be a path there must be cycles $(n_{S_i}, n_{S_i}) \in E_r$ for such i with $m_i > 1$. This follows

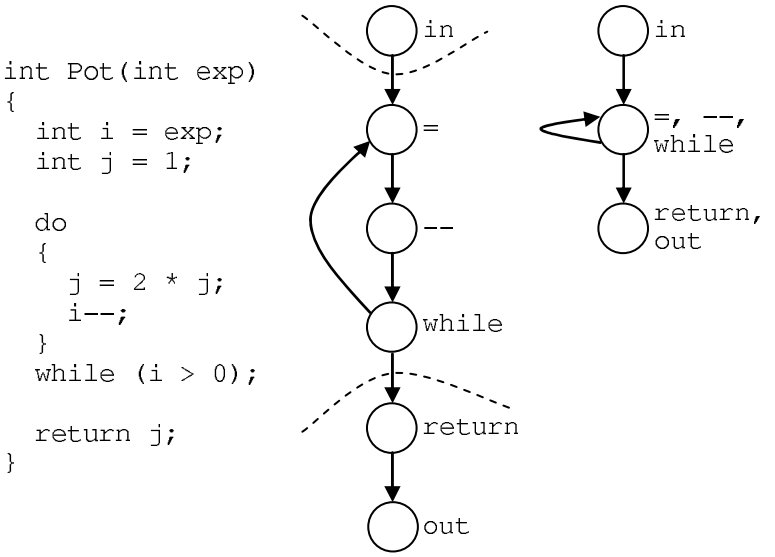

Fig. 6. Function `Pot`, control flow graph and segment graph.

from Definition 4 since $d_{i1}\, d_{i2}$ is a path in G that consists only of nodes from the maximal segment S_i and in which the first node of d_{i1} occurs twice. Furthermore, there must be edges $(n_{S_i}, n_{S_{i+1}}) \in E_r$ for $i = 1, \ldots, l-1$. This follows also from Definition 4 since S_i, S_{i+1} are different segments and there must exist an edge from a node in S_i to a node S_{i+1} in E because $d_i\, d_{i+1}$ is a path in G. ■

Segments (Definition 3), graph reduction (Definition 4), path reduction (Definition 5) and the results (Theorems 1 and 2) can be applied to all directed graphs, not only to control flow graphs. Now we return to control flow graphs and software testing. The execution of a test case for a function runs through the control flow graph and thus induces a path in the graph. Since the execution starts always with the first executable statement, i.e., with an entry node of the control flow graph, an S-path is induced, but not always a complete path, because the execution could encounter, for example, an infinite loop. Such loops induce infinite paths and cannot be tested in finite time. Therefore, in practical applications we have to stop the execution of such a test case after some time or, better, after a maximal length of the induced path because the latter is machine independent und reproducible. We replace (finite and) infinite paths induced by test cases by the sets of their finite prefixes. Thus we avoid infinite paths and can use as a test criterion some upper bound for the length of the paths that should be tested, i.e., in practical applications we can use a defined subset of all induced paths for testing.

Definition 6. Let t be a test case of a function f. Then we denote by $D(f,t)$ the prefix closed set of the S-paths in the control flow graph of the function that are induced by the execution of the test case t and by $D_r(f,t) = \{d_r \mid d \in D(f,t)\}$ the set of the segment paths. For a set T of test cases we write $D(f,T) = \bigcup_{t\in T} D(f,t)$ and $D_r(f,T) = \bigcup_{t\in T} D_r(f,t)$.

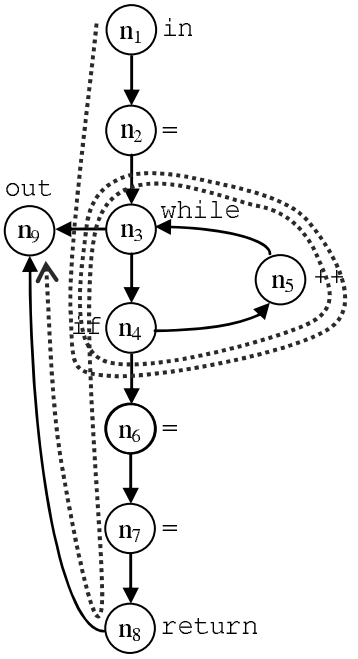

Fig. 7. Complete path in the control flow graph of the function `Search`.

In Fig. 7, a complete path $d = n_1\, n_2\, n_3\, n_4\, n_5\, n_3\, n_4\, n_5\, n_3\, n_4\, n_6\, n_7\, n_8\, n_9$ in the control flow graph of the function `Search` is shown. It is executed, e.g., for a test case t with input `arr == {1, 2, 3, 4}`, `key == -3`. Therefore, $D(\text{Search}, t) = \text{prefix}(d)$.

```
int isabsequal(int x, int y)
{
  if (x == y)
    return 1;
  else if (x == -y)
    return -1;
  else
    return 0;
}
```

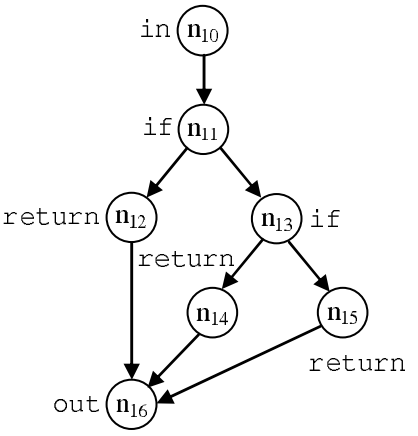

Fig. 8. Function `isabsequal` and its control flow graph.

When we execute a program that consists of a set of functions, the control flow graphs of these functions are run through. Each time a function is called, a prefix closed set of S-paths in the control flow graph of the called function is induced.

Definition 7. A *program* p is a set of functions. We always assume that the functions in a program have pairwise disjoint control flow graphs and also pairwise disjoint segment graphs (and in Section 4 also pairwise disjoint decision graphs). Let t be a test case of a program p. We denote by $D(p,t)$ the prefix closed set of the S-paths in the control flow graphs of the functions of the program p that are induced by the execution of the test case t, and by $D_r(p,t) = \{d_r \mid d \in D(p,t)\}$ the set of the segment paths. Accordingly, we define $D(p,T)$ and $D_r(p,T)$.

Consider the program p that consists of the functions `Search` and `isabsequal` (Fig. 8), and again the test case t as above and a second test case t' with `arr == {1, 2, 3, 4}`, `key == 3`; then the paths $D(p,\{t,t'\}) = \text{prefix}(\{d, d_1, d_2, d_3\})$ with d as above and $d_1 = n_{10}\, n_{11}\, n_{13}\, n_{15}\, n_{16}$, $d_2 = n_{10}\, n_{11}\, n_{13}\, n_{14}\, n_{16}$, $d_3 = n_{10}\, n_{11}\, n_{12}\, n_{16}$ are executed.

Statement coverage means that in a test of a program the test cases execute all statements in the program. For the control flow graphs of the functions in the program, this means that all nodes are covered by the paths that are induced by the test cases. Therefore, this coverage criterion is also called all-nodes criterion (Zhu *et al*., 1997).

Definition 8. Let p be a program. A set T of test cases satisfies *statement coverage* if and only if

$$\forall f \in p\ \forall n \in N_f\ \exists d \in D(p,T) : n \in N(d),$$

where N_f is the set of nodes of the control flow graph of the function f.

The set of test cases $T = \{t, t'\}$ with t, t' as above satisfies for the program p that consists of the functions `Search` and `isabsequal` as above statement coverage since each node in the control flow graphs of the functions is covered by at least one path in $D(p,T)$.

We define the same notion for segment graphs (use the set N_f^r of nodes of the segment graph instead of N_f and $D_r(p,T)$ instead of $D(p,T)$) and call it *segment coverage*.

Due to infinite loops we cannot guarantee that, if a test case enters a segment, the segment is executed completely. The program in Fig. 9 consists of the functions `f` und `g`. For the set $T = \{t_1, t_2\}$ with the test cases t_1: `i == -1`, `j == 1` and t_2: `i == 1`, `j == -1`, the only node that is not covered in G_f and G_g is the node representing the statement `i = j;` and the only uncovered segment in G_f^r and G_g^r is the segment that contains the node `i = j;` due to the infinite loop in the second call of `g` for the second test case. According to Definition 5, a node in the segment graph is only covered if all nodes in the segment are covered.

This leads to what follows.

Theorem 3. *Let p be a program and T a set of test cases. Then T satisfies statement coverage if and only if T satisfies segment coverage.*

Proof. "⇒": Let n' be a node in the segment graph of a function $f \in p$. Then there exists a maximal segment S in the control flow graph with $n' = n_S$. Let $S = \langle n_1, n_2, \ldots, n_k\rangle$ be an order of the nodes with $\text{post}(n_i) = \{n_{i+1}\}$, $\text{pre}(n_{i+1}) = \{n_i\}$ for $i = 1, \ldots, k-1$. Then there is a path $d \in D(p,T)$ with $n_k \in N(d)$ since T satisfies statement coverage and from Definition 7 it follows that $d_r \in D_r(p,T)$. If we decompose d according to Definition 5 into $d = d_1\, d_2 \ldots d_l$ and if n_k occurs in $d_1\, d_2 \ldots d_{l-1}$, we get that $n_S \in N(d_r)$. If n_k occurs only in d_l, all other nodes $n_1, n_2, \ldots, n_{k-1}$ of the segment must also occur in d_l, since n_1 is the only entry point in S and we cannot start d within a segment because d is

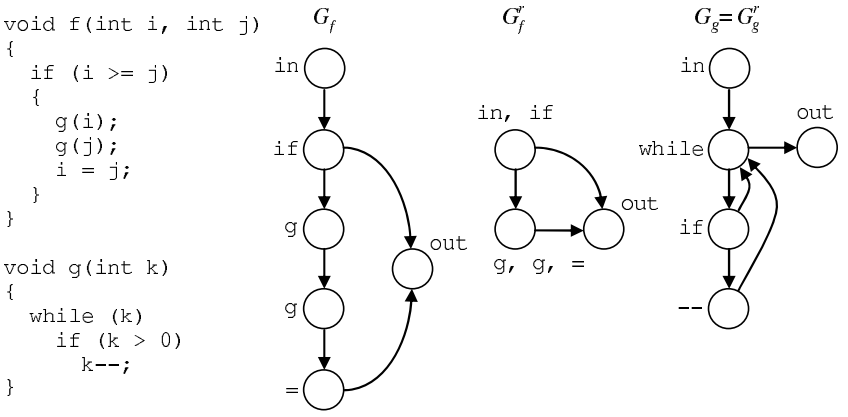

Fig. 9. Functions f and g, control flow graphs G_f, G_g and segment graphs G_f^r, G_g^r.

induced by a test case and therefore it is an S-path. It follows also that $n_S \in N(d_r)$.

"$\Leftarrow$": Let n be a node in the control flow graph of a function $f \in p$ and let S be the maximal segment that contains n. Then there exists a path $d' \in D_r(p,T)$ with $n_S \in N(d')$ since T satisfies segment coverage and it follows with Definition 7 that there is a path $d \in D(p,T)$ with $d' = d_r$. With the construction of Definition 5, we get $d_r = n_{S_1}^{m_1} n_{S_2}^{m_2} \ldots n_{S_{l-1}}^{m_{l-1}}$, possibly followed by $n_{S_l}^{m_l}$. If $n_S = n_{S_l}$, we know that all nodes of S are contained in d and $n \in N(d)$. Otherwise, $n_S = n_{S_h}$ for one $h \in \{1, \ldots, l-1\}$ and a node $n_j \in S$ must be contained in d_h (S as above). Since n_1 is the only entry point and n_k is the only exit point in S and we cannot start d within a segment, the node $n \in S$ must also be contained in d_h and therefore $n \in N(d)$. ■

4. Decision graphs

Directed graphs can be reduced further if we keep only entry and exit nodes and such nodes that represent decisions, i.e., with postsets that have two or more elements. Paige (1977) calls them D-nodes.

Definition 9. Let $G = (N, E)$ be a directed graph. A node $n \in N$ is called a *D-node* if it is an entry node or an exit node or if $|\mathrm{post}(n)| \geq 2$. A *DD-path* is a path $n_1\, n_2 \ldots n_k$ where the start and end nodes n_1, n_k are D-nodes and the other nodes $n_2, \ldots, n_{k-1}$ are no D-nodes.

To represent the branching structure of functions we will replace DD-paths, with single edges. The following lemma shows properties of DD-paths that are necessary for the definition of decision graphs.

Lemma 2. *Let $G = (N, E)$ be a directed graph.*

1. *Let n, $n' \in N$ be two D-nodes. Then there are at most $|\mathrm{post}(n)|$ different DD-paths that start in n and end in n'.*
2. *Let $d = n_1\, n_2 \ldots n_k$ be a DD-path. Each inner node $n_2, \ldots, n_{k-1}$ occurs only once in d.*

Proof.
1. Let $b = |\mathrm{post}(n)|$ and assume that there are $b+1$ different DD-paths that start in n and end in n'. Then two DD-paths d_1, d_2 have the same first edge since there are only b possibilities to start a path from n. Let $d_1 = n_{11}\, n_{12} \ldots n_{1k}$ and $d_2 = n_{21}\, n_{22} \ldots n_{2h}$. Then $n_{11} = n_{21} = n$, $n_{12} = n_{22}$ and $n_{1k} = n_{2h} = n'$. Since $n_{12}, \ldots, n_{1\,k-1}$ and $n_{22}, \ldots, n_{1\,h-1}$ are no D-nodes, they have postsets with only one element. It follows that $n_{13} = n_{23}$, …, $n_{1\,k-1} = n_{1\,h-1}$ and $k-1 = h-1$. This means that both paths are equal.

2. Assume that an inner node occurs twice in d as n_i and n_j. Then the path $n_{i+1} \ldots n_j$ can be inserted $v \geq 0$ times in d after n_j. The resulting path $n_1 \ldots n_i \ldots n_j\, (n_{i+1} \ldots n_j)^v\, n_{j+1} \ldots n_k$ is also a DD-path. This means that we have an infinite number of DD-paths that start in n_1 and end in n_k. ■

Since there may be two or more DD-paths between two D-nodes, we need graphs with multiple edges to define decision graphs if we want one edge for each branch in the function when we apply decision graphs to functions.

Definition 10. Let $G = (N, E)$ be a directed graph. The *decision graph* $G_c = (N_c, E_c)$ is a directed graph with multiple edges that consists of the set of nodes

$$N_c = \{n \in N \mid n \text{ is a D-node in } G\}$$

and the set of edges E_c that contains b edges with a start node n and an end node n' for all nodes $n, n' \in N$, where b is the number of different non-empty DD-paths in G that start in n and end in n'.

From Lemma 2 it follows that $b \leq |\mathrm{post}(n)|$ and therefore is b finite. Figure 10 shows the decision graph

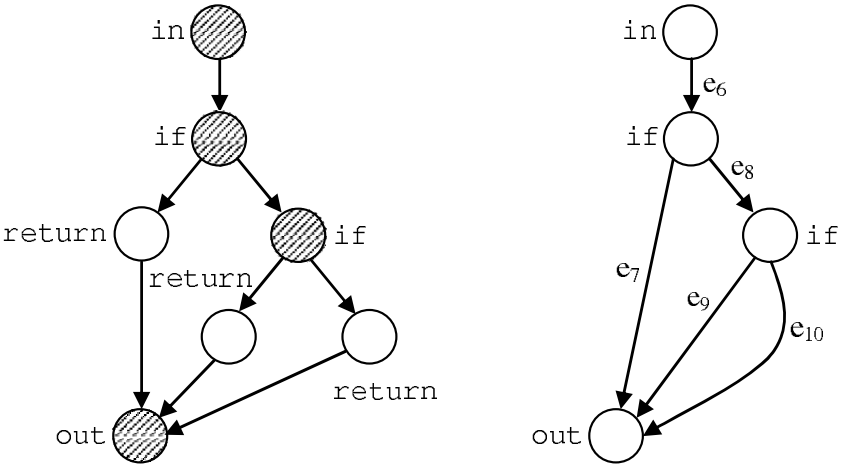

Fig. 11. Control flow graph and decision graph of function `isabsequal`.

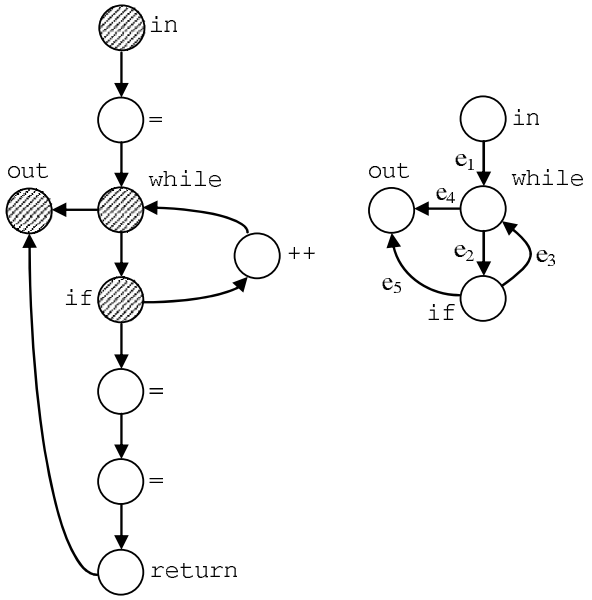

Fig. 10. Control flow graph and decision graph of function `Search`.

of the control flow graph of the function `Search`. The decision graph of the control flow graph of the function `isabsequal` (Fig. 11) contains multiple edges.

As in Section 3, we can also reduce paths, this time to decision paths.

Definition 11. Let $G = (N, E)$ be a directed graph and $d = n_1\, n_2 \ldots n_k$ a path in G that starts with a D-node. Let n_{i_1}, n_{i_2}, ..., n_{i_m} be the D-nodes in d with $i_1 < i_2 < \ldots < i_m$. Then $d_j = n_{i_j}\, n_{i_j+1} \ldots n_{i_{j+1}}$ are DD-paths for $j = 1, \ldots, m-1$. From Definition 10 it follows that there are edges e_j with $\mathrm{start}(e_j) = n_{i_j}$ and $\mathrm{end}(e_j) = n_{i_{j+1}}$ in the decision graph G_c associated to the DD-paths d_j. The *decision path* d_c is defined as $e_1\, e_2 \ldots e_{m-1}$. The decision path d_c for a path d in G without D-node is defined as the empty path.

Note that the nodes $n_{i_{m+1}} \ldots n_k$ following the last D-node n_{i_m} are clipped. When the path contains only one D-node, i.e., $m = 1$, the decision path is empty. From $\mathrm{end}(e_j) = n_{i_{j+1}} = \mathrm{start}(e_{j+1})$ for $j = 1, \ldots, m-2$ we obtain the following:

Theorem 4. *Let $G = (N, E)$ be a directed graph and d a path in G. The decision path d_c is then a path in the decision graph G_c.*

So far we have not used any semantical information and therefore the definitions and results about decision graphs can be applied to all directed graphs. Decision graphs used for functions in programming languages have one edge for each branch in the function plus one edge from the entry node to the D-node that represents the first decision statement. Branch coverage is usually defined as all-edges criterion for control flow graphs (Zhu *et al.*, 1997). But since in decision graphs edges correspond to branches, it is possible to define branch coverage based on decision graphs.

Definition 12. With the same notation as in Definition 7 we define $D_c(p, t) = \{d_c \mid d \in D(p, t)\}$ and $D_c(p, T) = \bigcup_{t \in T} D_c(p, t)$.

The program p that consists of the functions `Search` and `isabsequal` with the set of test cases $T = \{t, t'\}$ as in Section 3 has the set $D_c(p, T) = \mathrm{prefix}(\{e_1\, e_2\, e_3\, e_2\, e_3\, e_2\, e_5,\ e_6\, e_8\, e_{10},\ e_6\, e_8\, e_9, e_6\, e_7\})$ of decision paths.

Definition 13. Let p be a program. A set T of test cases satisfies *branch coverage* if and only if

$$\forall f \in p\ \forall e \in E_f\ \exists d \in D_c(p, T) : e \in E(d),$$

where E_f is the set of edges of the decision graph of the function f.

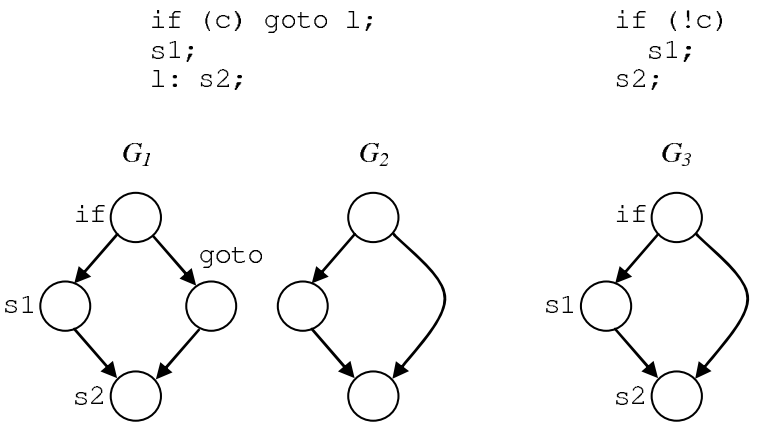

Fig. 12. Example with `goto` and its segment graph (G_1) and program graph (Rapps and Weyuker, 1982) (G_2); the same example without `goto` and its segment graph (G_3).

The set T of test cases does not satisfy branch coverage since the edge e_4 is not covered. If we add another test case t'' with `arr` $== \{$`1, 2, 3, 4`$\}$, `key == 0`, a new path $e_1\,e_2\,e_3\,e_2\,e_3\,e_2\,e_3\,e_2\,e_3\,e_4$ is executed and all edges of the decision graphs are covered.

This example also shows that node coverage of decision graphs is not equal to statement coverage, i.e., node coverage of control flow graphs the since test case t covers all nodes of the decision graphs of both functions but the node n_{12} in the control flow graph of `isabsequal` is not covered.

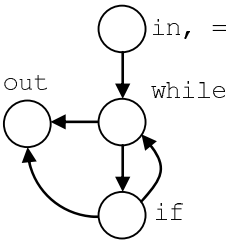

Fig. 13. Program graph reduced to segments (Paige, 1977) for the function `Search`.

5. Conclusion

We defined two graph reductions that can be applied to directed graphs and especially to control flow graphs. The definitions of segments and segment graphs are based on that of blocks and program graphs, respectively, given by Rapps and Weyuker (1982) as well as Jalote (2005) with two minor differences. Firstly, the latter definitions are directly on programs using semantical information whereas our definitions are based only on syntactical information about the graphs. The second difference is the treatment of constructs `if (condition) goto label;` since they are seen as one statement where we understand them as two statements and therefore the resulting graphs differ. But compared with the form of this construct without `goto`, the approaches are equal, as the example in Fig. 12 shows.

In the work of Rapps and Weyuker (1982), the definition of node and edge coverage is based on complete paths, whereas we use S-paths to capture infinite loops which can occur in practical applications, for example, in embedded control systems.

Paige (1977) defines a similar graph reduction called program graphs reduced to segments. There, a segment "is a simple path between two S-nodes such that no S-node appears within the segment". Nodes that have a preset or a postset with more than one node are S-nodes, and the entry and the exit node are also S-nodes. In a simple path, no edge is allowed to appear more than once and no node more than twice. A segment is reduced to the contained S-nodes and an edge between the S-nodes. With this definition, the S-nodes in the function `Search` are "in", "while", "if", "out" and the control flow graph is reduced to that shown in Fig. 13. According to our definition, a segment that contains no S-node will not occur in the program graph reduced to segments. In this case, program graphs do not distinguish between branches with statements and empty branches, as the example in Fig. 14 shows. But this distinction is essential for code coverage. Due to this property, a program graph reduced to segments can be smaller than the segment graph (e.g., the program graph of the function `Search` in Fig. 13 compared with its segment graph in Fig. 5). But since a segment can also contain two S-nodes (the first and the last node of the segment), the segment graph can have fewer nodes than the program graph, as the example `Pot` in Fig. 6 shows (the control flow graph contains four S-nodes but only three segments).

The definitions of D-nodes and DD-paths are based on the definitions by Paige (1977). The only difference is

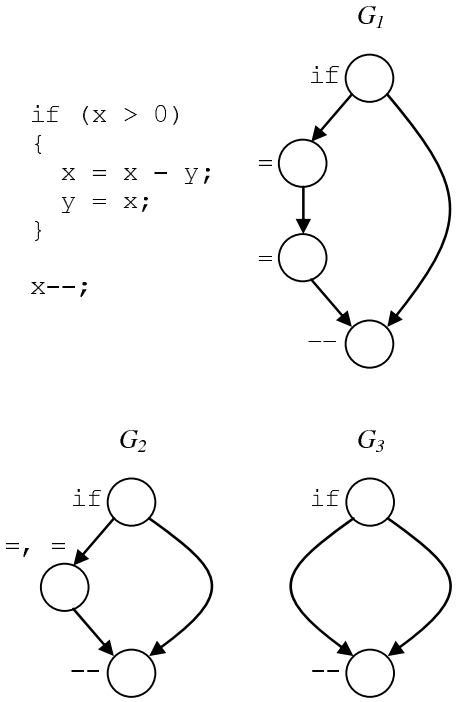

Fig. 14. Example with empty branch and its control flow graph (G_1), segment graph (G_2) and program graph reduced to segments (Paige, 1977) as a graph with multiple edges (G_3).

that there a DD-path is a simple path. But from Lemma 2 it follows that a DD-path, according to Definition 9, is always a simple path. Decision graphs correspond to program graphs reduced to DD-paths (Paige, 1977) with the difference that we explicitly use multiple edges if there is more than one DD-path between two D-nodes.

Segment and decision graphs can be seen as abstractions from statements to blocks of statements and to decisions, respectively. Furthermore, they differ in the treatment of entry nodes. A segment can start with an entry node of the graph and end with an exit node or with another D-node (but, of course, not with an entry node). Such a segment contains two D-nodes. A node in a segment graph can therefore represent two D-nodes (an entry node and another D-node), but a node in a decision graph is a single D-node by definition. Clearly, this difference could be solved by changing the definition of segments but, on the other hand, it is easy to see that if n is an entry node in a graph and S is the maximal segment that contains n, then the node n_S in the segment graph that corresponds to the segment S is also an entry node, and if n_S is an entry node in a segment graph, then the segment S in the graph contains an entry node. Therefore, the information about entry nodes is not lost during the construction of segment graphs.

The reductions to segment graphs and to decision graphs can be applied to all directed graphs, not only to control flow graphs of programs, and are useful when they should be abstracted from sequential actions or when the focus is on decisions only. Other fields of application include business or manufacturing process modelling, program compilation, static analysis, e.g., for software metrics or worst case execution times and digital system design, e.g., in VHDL or other hardware description languages.

Table 1. Properties of control flow graphs.

	control flow graphs	segment graphs	decision graphs
multiple edges	no	no	yes
nodes correspond to	single statements	sequences of statements (segments)	decision statements (D-nodes)
edges correspond to	control flow between single statements (Definition 2)	control flow between segments (Definition 4)	control flow between decision statements (DD-paths), i.e., branches (Definition 10)
defines coverage	statement coverage (Definition 8)	segment coverage	branch coverage (Definition 13)
node coverage equal to statement coverage	by definition	yes (Theorem 3)	no (example)
edge coverage equal to branch coverage	yes (not shown in this paper)	yes (not shown in this paper)	by definition

When we apply graphs to the control flow of programs and to software testing, the three graph types, (control flow graphs, segment graphs and decision graphs), have the properties as shown in Table 1. Nodes in control flow graphs correspond to statements and therefore node coverage defines statement coverage. Analogously, edges in decision graphs represent branches and therefore edge coverage correspond to branch coverage. The defined graph types are thus three different abstractions from programs. This paper establishes a uniform and formal basis for these graph types for the use in manual and tool-supported control flow oriented test case generation and graphical representation of programs.

Acknowledgment

The author wishes to thank the anonymous reviewers for their careful reading and helpful suggestions.

References

Jalote, P. (2005). *An Integrated Approach to Software Engineering*, Springer, New York, NY.

Kosaraju, S. (1973). Analysis of structured programs, *Proceedings of the 5th Annual ACM Symposium on Theory of Computing, Austin, TX, USA*, pp. 240–252.

McCabe, T. (1976). A complexity measure, *IEEE Transactions on Software Engineering* **SE-2**(4): 308–320.

Paige, M. (1977). On partitioning program graphs, *IEEE Transactions on Software Engineering* **SE-3**(6): 386–393.

Rapps, S. and Weyuker, E. (1982). Data flow analysis techniques for test data selection, *Proceedings of the 6th International Conference on Software Engineering, Tokyo, Japan*, pp. 272–278.

Sommerville, I. (2004). *Software Engineering*, 7th Edn., Pearson Education Limited, Boston, MA.

Tan, L. (2006). The Worst Case Execution Time Tool Challenge 2006: The External Test, *Technical report*, `http://www.absint.com/ait/wcet_tool_challenge_2006_final_report.pdf`.

White, L. (1981). Basic mathematical definitions and results in testing, *in* B. Chandrasekaran and S. Radicchi (Eds.), *Computer Program Testing*, North-Holland, New York, NY.

Zhu, H., Hall, P. and May, J. (1997). Software unit test coverage and adequacy, *ACM Computing Surveys* **29**(4): 366–427.

Robert Gold received his diploma and Ph.D. degrees in computer science from the Technical University of Munich. Since 1998 he has been a professor of software engineering and programming languages at the Ingolstadt University of Applied Sciences. His work and research interest include automotive software engineering, software test and usability.

Received: 14 January 2010
Revised: 16 August 2010