Hindawi Publishing Corporation
ISRN Software Engineering
Volume 2013, Article ID 432021, 12 pages
http://dx.doi.org/10.1155/2013/432021

Research Article

Decision Graphs and Their Application to Software Testing

Robert Gold

Faculty of Electrical Engineering and Computer Science, Ingolstadt University of Applied Sciences, Esplanade 10, 85049 Ingolstadt, Germany

Correspondence should be addressed to Robert Gold; robert.gold@haw-ingolstadt.de

Received 20 December 2012; Accepted 17 February 2013

Academic Editors: C. Rolland and S. Sutton

Copyright © 2013 Robert Gold. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Control flow graphs are a well-known graphical representation of programs that capture the control flow but abstract from program details. In this paper, we derive decision graphs that reduce control flow graphs but preserve the branching structure of programs. As an application to software engineering, we use decision graphs to compare and clarify different definitions of branch covering in software testing.

1. Introduction

Graphs that represent the control flow of programs have been studied since many years and are known under the names of control flow graphs or program graphs. There are mainly two types of such graphs: one that associates one node with each statement in programs, see, for example, [1], where control flow graphs are applied to optimization or [2] for the application in software engineering; and the other that replaces maximal sets of consecutive nodes with a single entry and a single exit called blocks or segments, by single nodes, for example, [3, 4]. Blocks can be derived from the control flow graphs of the first type or constructed directly from the programs. Both types capture the control flow by abstraction from the program details.

Since the control flow through programs is determined by the decisions, for example, the if-then-else-constructs, based on the data and the conditions in such constructs, it is promising to keep in graphical representations of programs only the decisions and the control flow between them and thus defining a reduction of control flow graphs that preserves the branching structure.

In this paper, we will study control flow graphs (of the first type) and derive decision graphs [5, 6] that represent the branching structure of programs based on the definition of program graphs reduced to DD-paths by Paige [7].

Statement coverage and branch coverage are widely used in software testing. The first property can be checked with control flow graphs since each node represents a statement (or block of statements). When regarding branch coverage, analogously, the question arises, in which in graph type, each edge represents a branch.

More general, decision graphs can be derived not only from control flow graphs but also from arbitrary directed graphs and thus represent the branching structure of the graphs. We show that the branches in a graph correspond to the edges in the derived decision graph.

As an application, we compare different definitions of branch covering in software testing that already existed but were specified in different ways in order to find where they differ from each other and thus get new results on branch coverage that clarify the definitions found in the related literature.

The contribution of this paper is to solve the problem of finding a graph type that has one edge for each branch, analogously to control flow graphs that have one node for each statement. Furthermore, we apply decision graphs to software engineering and clarify the different notions of branch covering in software testing, one of them based on decision graphs, in order to avoid confusion when using them in practice. We propose decision graphs, independently, from the application to software testing, as means to abstract from details and focus on the decision structure.

The remainder of this paper is organized as follows. We start with the necessary definitions and results about directed graphs, control flow graphs, and decision graphs. In Section 3, we show the correspondence of branches in a graph and the edges in the derived decision graph. As an application, we define three different notions of branch coverage and compare them in Section 4. Related work and other applications of control flow graphs are discussed in Section 5. Section 6 concludes the paper.

2. Basic Definitions and Results

This section presents definitions and results—partially taken from [5, 6]—necessary for the following.

2.1. Directed 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 $\text{start}: E \to N$ and $\text{end}: E \to N$ that associate a start node and an end node, respectively, with each edge.

For a node $n \in N$, the sets $\text{pre}(n) = \{n' \mid \exists\, e \in E : \text{start}(e) = n', \text{end}(e) = n\}$ and $\text{post}(n) = \{n' \mid \exists\, e \in E : \text{start}(e) = n, \text{end}(e) = n'\}$ are called *preset* and *postset* of n, respectively. An edge that ends in a node $n \in N$ is called *incoming edge* of n; an edge that starts in $n \in N$ is called *outgoing edge* of n. The *indegree* and *outdegree* of n is the number of the incoming and of the outgoing edges of n, respectively, that is, $\text{indegree}(n) = |\{e \in E \mid \text{end}(e) = n\}|$; $\text{outdegree}(n) = |\{e \in E \mid \text{start}(e) = n\}|$. Nodes with indegree 0 are called *entry nodes* and those with outdegree 0 *exit nodes* of the graph.

A *path* d is a nonempty, finite sequence of edges $e_1 e_2 \cdots e_k$ such that $\text{end}(e_i) = \text{start}(e_{i+1})$ 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. The *length* of a path d is the number k of its edges. The nodes $\text{start}(e_2), \ldots, \text{start}(e_k)$ are called *inner nodes* of d. A path d which contains one node twice and all other nodes only once is called *loop*. A loop d is called *unconditional loop*, if the inner nodes and the end node have outdegree 1. A special case of an unconditional loop is an *isolated loop*, that is, a loop that has only nodes with indegree 1 and outdegree 1. A path d' is *contained* in a path $d = e_1 e_2 \cdots e_k$ if $d' = e_i e_{i+1} \cdots e_j$ for indices $1 \le i \le j \le k$. A *prefix* of a path $d = e_1 e_2 \cdots e_k$ is an initial part $d' = e_1 e_2 \cdots e_j$ of the path for $1 \le j \le k$ denoted by $d' \le d$. If $j < k$, we write $d' < d$. For a path d, the set $\text{prefix}(d) = \{d' \mid d' \le d\}$ and, for a set D of paths, the set $\text{prefix}(D) = \bigcup_{d \in D} \text{prefix}(d)$ are the sets of prefixes of d and D, respectively. A path starting with an entry node is called S-*path*. An S-path that ends in an exit node is called *complete path*. By $N(d)$ and $E(d)$, we denote the set of nodes and edges, respectively, that are contained in the path d. A node n' is *reachable* from a node n if $n = n'$ or if a path d with start node n and end node n' exists.

If it is not clear from the context which graph is meant, we add a subscript to the functions start, end, pre, post, indegree, outdegree, and prefix, for example, $\text{pre}_G(n)$.

A detailed introduction to graphs can be found, for example, in [8, 9].

In many cases, we do not need multiple edges between nodes. If for all $e_1, e_2 \in E$: $\text{start}(e_1) = \text{start}(e_2) \wedge \text{end}(e_1) = \text{end}(e_2) \Rightarrow e_1 = e_2$, the graph is called simply *directed graph*. In such a graph the notation can be simplified and an edge e can be written as a pair $(\text{start}(e), \text{end}(e)) \subseteq N \times N$ of nodes. A path can then briefly be denoted by the sequence of the contained nodes $n_1 n_2 \cdots n_{k+1}$, where $k \ge 1$. Since there are no multiple edges in such a graph, the numbers of incoming and outgoing edges of a node n are equal to the numbers of elements of the preset, postset of n, respectively, that is, $\text{indegree}(n) = |\text{pre}(n)|$, $\text{outdegree}(n) = |\text{post}(n)|$.

A simple fact that will be used later is that in a directed graph with multiple edges, the number of edges is equal to the sum of the numbers of incoming edges of all nodes and also equal to the sum of the numbers of outgoing edges of all nodes: $|E| = \sum_{n \in N} \text{indegree}(n) = \sum_{n \in N} \text{outdegree}(n)$. The same holds for an unconnected component of a graph. An isolated loop forms a component of the graph, consisting of sequential nodes, that is unconnected to the rest of the graph. Therefore, the above equation is also valid for isolated loops.

2.2. Control Flow Graphs.

The control flow in a function written in a programming language can be modeled by a directed graph called control flow graph, which contains one node for each statement in the function and edges that represent the control flow between statements. We add an entry node and an exit node as unique entry and exit points of the function. When a function is called within a function, the control flow also leaves the function and enters it again after the execution of the called function. But since we discuss only single functions, we do not interpret function calls as exit and entry points of the function. Statements in the programming language C are function calls, assignments, and other expressions with semicolon, return-, break-, continue-, goto-, if-, switch-, do-while-, for-, and while-statements and the null statement. Syntactically, a block is also a statement, but since it consists of statements, only the statements in the block are nodes in the control flow graph, not the block itself. Declarations and definitions are not statements and are also not included in the control flow graph.

Definition 2. The *control flow graph* $G_f = (N, E)$ of a function f is a directed graph that consists of the set of nodes $N = \{n_a \mid a \text{ is a statement in } f\} \cup \{n_{\text{in}}, n_{\text{out}}\}$ and the set E of edges. This set contains 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_{in}, n_{a_1}). Furthermore, we add edges $(n_{a'}, n_{\text{out}})$ 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, that is, a function without any statements, consists of $N = \{n_{\text{in}}, n_{\text{out}}\}$ and $E = \{(n_{\text{in}}, n_{\text{out}})\}$.

From the definition, it follows that in the control flow graph G_f, each node—with the exception of n_{in} and n_{out}—corresponds to a unique statement in the function f.

Figure 1 shows as an example the control flow graph of the following function `Search`:

```
void Search(int values [], int key,
            int *found, int *index)
{
  int i = 0;
  *found = 0;
  while (i < N)
  {
    if (values [i] == key)
    {
      printf("found positive.");
      *found = 1;
    }
    else
    {
      printf("non-positive.");
      if (values [i] == -key)
      {
        printf("found negative.");
        *found = -1;
      }
    }
    if (*found)
    {
      printf("found at index = %i.", i);
      *index = i;
      return;
    }
    i++;
  }
}
```

This function checks whether the array parameter `values` contains the integer parameter `key` or `-key`. In these cases, it sets the output parameter `found` to 1 or to −1 and sets the parameter `index` to the found index. The length of the array is given by the constant `N`.

The nodes of the control flow graph are labeled with "=", "while", and so forth to show which kind of statements is represented or with "in", "out" for better readability. The preset of the exit node consists of two nodes, one representing the return-statement and the other the while-statement after which the function reaches the terminating brace.

2.3. Decision Graphs. As in [5, 6], we reduce directed graphs and keep in decision graphs only entry and exit nodes and such nodes that represent decisions, that is, nodes with postsets that have two or more elements, called D-nodes [7]. The following definitions and results are independent from the modeling of software by control flow graphs and can be applied to all directed graphs.

Definition 3. Let $G = (N, E)$ be a directed graph. A node $n \in N$ is called *D-node* if it is an entry node or an exit node or if outdegree$(n) \geq 2$. A *DD-path* is a path $n_1 n_2 \cdots n_{k+1}$ where the start and end nodes n_1, n_{k+1} are D-nodes and the other nodes $n_2, \ldots, n_k$ are not D-nodes.

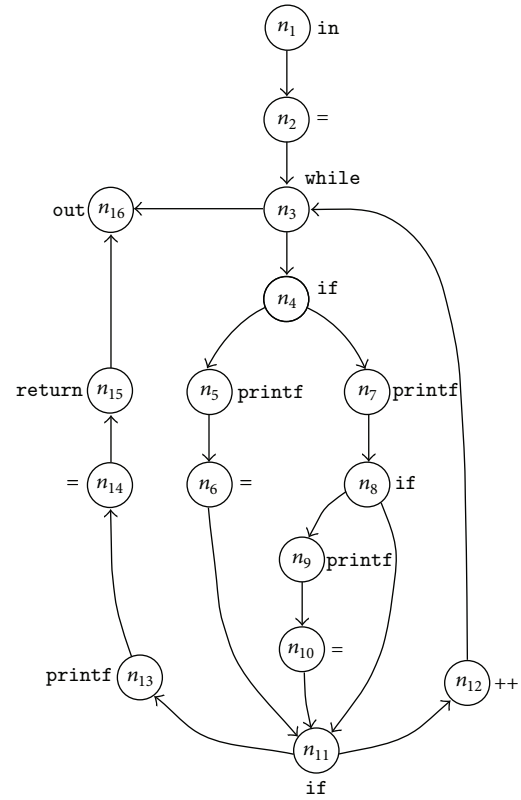


FIGURE 1: The control flow graph of the function `Search`.

A node is not a D-node if its indegree is at least 1 and its outdegree is exactly 1. The D-nodes of the control flow graph of the function `Search` (Figure 1) are the entry and exit nodes n_1, n_{16} and the while- and if-nodes n_3, n_4, n_8, n_{11}.

If an inner node $n_2, \ldots, n_k$ would occur twice in a DD-path, no D-node could be reachable from it. Therefore, all inner nodes are different [5]. Furthermore, there are at most outdegree(n) different DD-paths that start in a D-node n, since there is no branching possible after leaving n.

In order to reduce graphs but to preserve the branching structure we follow the idea of Paige [7] and replace DD-paths, with single edges. There may be more than one DD-path between two D-nodes, for example, between the second and the third if-node in the control flow graph of `Search` (Figure 1), and therefore multiple edges are necessary to avoid the merging of branches in the decision graph.

Definition 4. Let $G = (N, E)$ be a directed graph. The decision graph $\widehat{G} = (\widehat{N}, \widehat{E})$ of G is a directed graph with multiple edges that consists of the set of nodes $\widehat{N} = \{n \in N \mid n \text{ is a D-node in } G\}$ and the set of edges $\widehat{E}$ that contains an edge with start node n and end node n' for each DD-path in G that starts in n and ends in n'.

The number of edges in $\widehat{G}$ between two nodes n and n' is equal to the number of different DD-paths in G that start in n and end in n' and outdegree$_{\widehat{G}}(n)$ is equal to the (finite) number of different DD-paths in G that start in n.

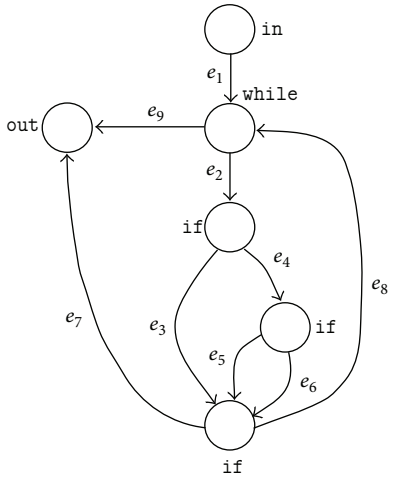


FIGURE 2: Decision graph derived from the control flow graph of the function `Search`.

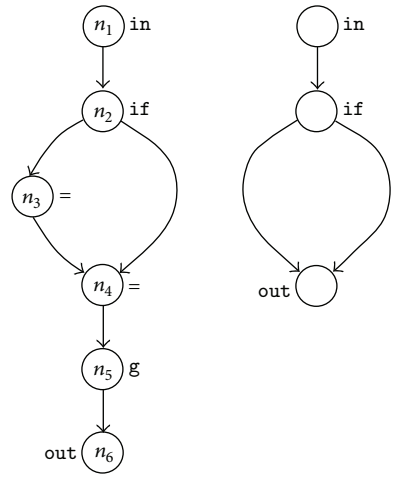


FIGURE 3: Control flow graph of function `f1` and the derived decision graph.

The assignment of edges in $\widehat{G}$ to DD-paths in G is a bijective function that allows us to identify the edge in the decision graph that corresponds to a given DD-path in the graph and vice versa. This will be necessary to distinguish multiple edges when we compare different coverage notions. In Figure 2, the decision graph of the control flow graph of the function `Search` is depicted.

Figure 3 shows the decision graph of the following function `f1`:

```
void f1(int x)
{
  int y;
  if (x > 0)
    x = 1;
  y = x;
  g(y);
}
```

This example shows that DD-paths need not to be disjoint ($n_2n_3n_4n_5n_6$ and $n_2n_4n_5n_6$).

Not only graphs can be reduced to decision graphs, but also paths can be reduced to to decision paths.

Definition 5. Let $G = (N, E)$ be a directed graph and $d = n_1n_2 \dots n_{k+1}$ a path in G that starts with a D-node and contains at least a second D-node. Let $n_{i_1}, n_{i_2}, \dots, n_{i_m}$ be the D-nodes in d with $i_1 < i_2 < \cdots < i_m$. Then $d_j = n_{i_j}n_{i_j+1} \cdots n_{i_{j+1}}$ are DD-paths for $j = 1, \dots, m-1$. From Definition 4, follows that there are edges e_j with start node n_{i_j} and end node $n_{i_{j+1}}$ in the decision graph $\widehat{G}$ associated to the DD-paths d_j. The *decision path* $\widehat{d}$ of d is defined as $e_1e_2 \cdots e_{m-1}$.

The nodes $n_{i_m+1} \cdots n_{k+1}$ following the last D-node n_{i_m} are not a complete DD-path and are therefore clipped. From $\text{end}(e_j) = n_{i_{j+1}} = \text{start}(e_{j+1})$ for $j = 1, \dots, m-1$, we obtain that the decision path $\widehat{d}$ is a path in the decision graph [5]. The path $\underline{n}_1n_2\underline{n}_3\underline{n}_4n_7\underline{n}_8\underline{n}_{11}n_{12}\underline{n}_3\underline{n}_4n_7\underline{n}_8n_9n_{10}\underline{n}_{11}n_{13}n_{14}$ in the control flow graph of the function `Search` contains nine ($m = 9$) D-nodes (underlined), can be split into eight DD-paths and therefore induces the decision path $e_1e_2e_4e_6e_8e_2e_4e_5$ with eight edges. The last two nodes $n_{13}n_{14}$ are not represented in the decision path.

3. Branches in Directed Graphs

In this section, we will discuss branches in directed graphs and their relationship to edges in the derived decision graphs. Let us examine the graph shown in Figure 4. This graph can be a control flow graph, for example, of the following function `f2`:

```
void f2(int x)
{
  if (x)
    label: goto label;
}
```

Since in the decision graph the goto-node will not appear, this example shows that unconditional loops are not represented in decision graphs. Therefore, the number of DD-paths that start in a D-node n, where an unconditional loop branches off is lower than outdegree(n). In the example, the if-node has two outgoing edges and a postset with two elements, but only one DD-path that starts in the if-node.

In programming languages like C, the only possibility to create unconditional or isolated loops is using goto-statements.

Lemma 6. *Let $G = (N, E)$ be directed graph. Then it holds:*

(1) *each node $n \in N$ that is not contained in an isolated loop is reachable from a D-node,*

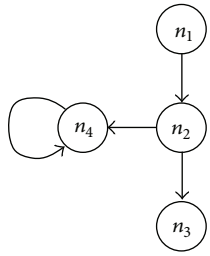


Figure 4: Control flow graph with unconditional loop.

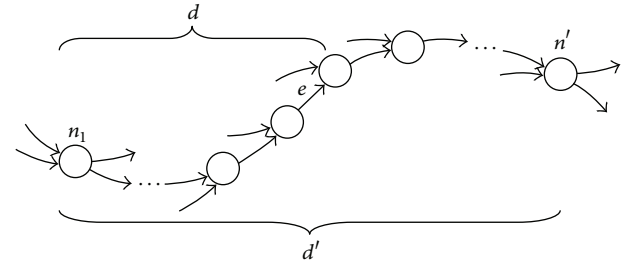


Figure 5: Path d and DD-path d' that contain the edge e.

(2) *each edge $e \in E$ that is not contained in an isolated loop is contained (as last edge) in a path that starts with a D-node and whose inner nodes are not D-nodes,*

(3) *each edge $e \in E$ that is not contained in an unconditional loop is contained in a DD-path [6].*

Proof. (1) Let N' be the set of nodes from which n is reachable and assume that N' does not contain a D-node. Let e be an edge that starts in $N' \setminus \{n\}$. If end(e) $\notin N'$, the node n cannot be reached from end(e). But n can be reached from start(e). Therefore, there must be a second edge that starts in start(e), and start(e) would be a D-node which contradicts to the assumption. This means that end(e) $\in N'$. Let e be an edge that ends in N'. Then n is also reachable from start(e) and start(e) $\in N'$. Together this means that, if there is a connection of N' with $N \setminus N'$ by an edge, this connection can only be the outgoing edge of n (which exists since n is not a D-node).

In the case that this connection does not exist and the end node of the outgoing edge of n is in N', the nodes N' form an unconnected component and $\sum_{n' \in N'} \text{indegree}(n') = \sum_{n' \in N'} \text{outdegree}(n') = |N'|$ (because of the assumption that N' does not contain D-nodes, outdegree(n') = 1 for all $n' \in N'$). Therefore, all nodes in N' must have indegree 1. Let us denote the successor of n by n_2, the successor of n_2 by n_3, and so on. Finally, we reach a node n_i (which must be n), a second time. This means that we found an isolated loop that contains n which is forbidden.

In the case that the end node of the outgoing edge of n is in not N', it follows that $\sum_{n' \in N'} \text{indegree}(n') = \sum_{n' \in N'} \text{outdegree}(n') - 1 = |N'| - 1$ and at least one node in N' must have indegree 0 and is a D-node which contradicts to the assumption.

(2) If the node start(e) is a D-node, e is contained in the path $d = e$. If start(e) is not a D-node, it follows from part 1 of this lemma that there is a path that starts in a D-node and ends in start(e). We prolong this path by the node end(e) and get the path d that contains e as last edge. If an inner node in the path d is a D-node, we shorten the path from the start node to the last inner D-node (let us denote it by n_1) in the path. The resulting path, denoted also by d, starts with the D-node n_1, which could be start(e), and ends with the edge e. The inner nodes of d are not D-nodes (Figure 5).

Part (1) of this lemma can be applied since e would be contained in the same isolated loop, if start(e) is contained in an isolated loop.

(3) If the last node in the path d (this is end(e)) according to part (2) of this lemma is not a D-node, we follow the edges until we find a D-node n' (an exit node is a D-node) or detect after at most $|N - 1|$ steps an unconditional loop that contains e which is forbidden by the assumption. Thus, we constructed a DD-path d' that contains the edge e. □

If we exclude unconditional loops, we can show that a decision graph has exactly one edge for each branch in the directed graph and thus abstracts from branch details. We identify branches by their first edges—that are outgoing edges of D-nodes. Such an edge leads from an entry node into the graph or selects a branch in a node with a postset of two or more.

Theorem 7. *Let G be a directed graph without unconditional loops. Then, the branches in G, that is, outgoing edges of D-nodes, correspond bijectively to the edges in the decision graph of G.*

Proof. Let $G = (N, E)$ be a directed graph. The set of outgoing edges of D-nodes will be denoted by $\overline{E}$. Let $e \in \overline{E}$. With part (3) of Lemma 6, follows that there exists a DD-path d that contains e. Since start(e) is a D-node, e must be the first edge in d. It is not possible that two different DD-paths start with the same first edge, and therefore the association $\beta : e \mapsto d$ is well-defined. Furthermore, $\beta(e_1) \neq \beta(e_2)$ for two different edges since e_1, e_2 are the first edges in $\beta(e_1)$ and $\beta(e_2)$, respectively. Of course, every DD-path starts with an edge in $\overline{E}$. Together, that means that β is a bijective association from $\overline{E}$ to the set of all DD-paths in G. In Definition 4, a bijective association was defined (let us call it δ) that associates an edge in the decision graph $\widehat{G} = (\widehat{N}, \widehat{E})$ of G with each DD-path in G. This means that $\delta \circ \beta : \overline{E} \to \widehat{E}$ is bijective. □

Note that this result holds for arbitrary directed graphs, not only for control flow graphs, and thus is independent from the modeling of software by graphs.

4. Branch Coverage

In this section, we apply decision graphs to software testing and compare different definitions of branch coverage.

4.1. Test Cases. When a test case for a function is executed, it runs through the function and also induces a path in the

control flow graph. This path always starts with the edge (n_{in}, n_{a_1}), where a_1 is the first executable statement or with $(n_{\text{in}}, n_{\text{out}})$, if the function does not contain any statements, and therefore is an S-path. In most cases, the execution reaches the end of the function, and the induced path is complete. But there are also cases where the exit node is not reached, for example, when a function is called that does not terminate or a division by 0 is encountered. In both cases, we observe a finite but not a complete path in the control flow graph of the function. The execution could also encounter an infinite loop in the function. Then, we observe in theory an infinite path. In practice, we have to stop the execution of the test case after some time and also get a finite path. This means that we always observe a finite path while executing a test case for a finite observation time. Mostly, we can distinguish these cases in practice while debugging the application. Clearly, this is impossible in general.

Definition 8. Let t be a test case of a function f. If τ is the observation time, we denote by $\mathrm{d}^{\tau}(f,t)$ the observed finite S-path in the control flow graph of the function that is induced by the execution of the test case t for time τ. The set

$$\begin{aligned}&\mathrm{D}(f,t)\\&\quad=\{d \mid \text{there exists an observation with } d=\mathrm{d}^{\tau}(f,t)\}\end{aligned} \tag{1}$$

is the set of all observed paths. For a set T of test cases, we write

$$\mathrm{D}(f,T)=\bigcup_{t\in T}\mathrm{D}(f,t). \tag{2}$$

If a path d is in $\mathrm{D}(f,T)$, any prefix d' can also be observed with shorter observation time. This means that $\mathrm{D}(f,T)$ is prefix closed.

The execution of the test case t of the function `Search` with the array 1, 2, 3, 4 for the parameter `values` and −2 for the second parameter `key` induces the complete path $d = n_1n_2n_3n_4n_7n_8n_{11}n_{12}n_3n_4n_7n_8n_9n_{10}n_{11}n_{13}n_{14}n_{15}n_{16}$ in the control flow graph. Therefore, $\mathrm{D}(\texttt{Search},t) = \{n_1n_2, n_1n_2n_3, n_1n_2n_3n_4, n_1n_2n_3n_4n_7, \ldots, d\} = \text{prefix}(d)$. The test case t_2 of the function `f2` (Figure 4) with parameter 1 results in an infinite execution that is represented by the infinite set of paths $\mathrm{D}(\texttt{f2},t_2) = \{n_1n_2, n_1n_2n_4, n_1n_2n_4n_4, n_1n_2n_4n_4n_4, \ldots\}$.

4.2. Coverage. The basic coverage notion in software testing is a statement coverage which is obtained if in a test of a function, the test cases in a set T execute all statements in the function. For the control flow graph $G_f = (N,E)$ of the function, it follows that all nodes are covered by the paths that are induced by the test cases, that is, $\forall n \in N\ \exists\, d \in \mathrm{D}(f,T) : n \in N(d)$. For this reason, this coverage criterion is also called all-nodes criterion [4, 10]. The set of test cases $T = \{t, t'\}$ with t as above and t' with parameters `values` = 1, 2, 3, 4 and `key` = 2, which induces the complete path $d' = n_1n_2n_3n_4n_7n_8n_{11}n_{12}n_3n_4n_5n_6n_{11}n_{13}n_{14}n_{15}n_{16}$ satisfies statement coverage for the function `Search` since each node in the control flow graph of the function is covered by the paths in $\mathrm{D}(\texttt{Search},T)$. A test case of a function induces paths in the control flow graph and thus also paths in the decision graph of the control flow graph if it runs through at least a second D-node.

Definition 9. Let t be a test case of a function f. We define

$$\begin{aligned}\widehat{\mathrm{D}}(f,t)&=\left\{\hat{d} \mid d\in \mathrm{D}(f,t) \text{ and } d \text{ contains}\right.\\&\qquad\left.\text{at least two D-nodes}\right\},\\\widehat{\mathrm{D}}(f,T)&=\bigcup_{t\in T}\widehat{\mathrm{D}}(f,t).\end{aligned} \tag{3}$$

The set $\widehat{\mathrm{D}}(f,t)$ is also prefix closed: Let $\hat{d} = e_1e_2\ldots e_{m-1} \in \widehat{\mathrm{D}}(f,t)$ (where $m \geq 2$) and $\hat{d}' = e_1e_2\ldots e_l$ a prefix with $1 \leq l \leq m-1$. Then, a path $d = n_1n_2\cdots n_{k+1} \in \mathrm{D}(f,t)$ exists such that $\hat{d}$ is the decision path of d. With Definition 5, it follows that the edges e_j, $1 \leq j \leq m-1$ correspond to the DD-paths $n_{i_j}n_{i_j+1}\cdots n_{i_{j+1}}$. The path $d' = n_1n_2\cdots n_{i_{l+1}} \leq d$ induces then $\hat{d}'$. Since $\mathrm{D}(f,t)$ is prefix closed, $d' \in \mathrm{D}(f,t)$ and $\hat{d}' \in \widehat{\mathrm{D}}(f,t)$.

The set of test cases $T = \{t, t'\}$ for the function `Search` as above induces two complete paths d and d' from which two decision paths $\hat{d} = e_1e_2e_4e_6e_8e_2e_4e_5e_7$, $\hat{d}' = e_1e_2e_4e_6e_8e_2e_3e_7$ and the set of decision paths $\widehat{\mathrm{D}}(\texttt{Search},T) = \text{prefix}(\{\hat{d},\hat{d}'\})$ can be derived.

In software testing, the notion of branch coverage where the test cases should cover all branches of the software is very popular because it is stronger than statement coverage but easier to obtain than more sophisticated coverage definitions like those that consider not only decisions but also the Boolean conditions that occur in the decisions or like data-flow-oriented coverage notions, for example, [4, 10], which can give better results [11]. In the following, we will give three definitions of branch coverage and investigate the relationship and differences between them. The first one captures the notion that in decisions like if-statements the conditions should be at least once true and once false during testing and thus all branches should be taken, the second one is edge covering of the control flow graph, and the last one is edge covering of the decision graph.

Definition 10. Let f be a function, $G_f = (N,E)$ the control flow graph of f and $\widehat{G} = (\widehat{N},\widehat{E})$ the decision graph of the control flow graph.

(i) A set T of test cases satisfies decision coverage if and only if

$$\begin{aligned}&\forall e \in E \text{ such that start}(e) \text{ is a D-node}\\&\exists d \in \mathrm{D}(f,T) : e \in E(d).\end{aligned} \tag{4}$$

(ii) A set T of test cases satisfies *edge coverage* (of the control flow graph) if and only if

$$\forall e \in E \exists d \in \mathrm{D}(f,T) : e \in E(d). \tag{5}$$

(iii) A set T of test cases satisfies branch coverage [5] if and only if

$$\forall e \in \widehat{E} \exists d \in \widehat{\mathrm{D}}(f,T) : e \in E(d). \tag{6}$$

Edge covering (of the control flow graph) is often called all-edges criterion or branch coverage by many authors, for example, [4, 10, 12], whereas our notion of branch coverage is defined as edge covering of the decision graph. Hierons et al. [13] define branch coverage based on outgoing edges of D-nodes similar to our definition (i). Frankl and Weyuker [11] do not distinguish between branch testing and decision coverage based on the Boolean conditions. Further definitions of branch coverage arise when other graph reductions are used. For example, Bertolino and Marré [14] define branches that start/end in D-nodes, junction nodes (with indegree ≥2), the entry node, or the exit node. In the reduced graphs, called ddgraphs, branches are replaced by edges. We do not consider these definitions of branches because we concentrate on the reduction to decision graphs.

Note that in the rest of the paper we mean by branch coverage the edge coverage of the decision graph, as in Definition 10(iii), unless otherwise stated.

The set T of test cases for the function `Search` does not satisfy any of these coverage notions since the edge between the while-node and the exit node is neither covered in the control flow graph nor in the decision graph. We need a third test case t'' with parameters `values` = 1, 2, 3, 4 and `key` = 0 to cover all edges in both graphs.

It is obvious that a set of test cases that satisfies edge coverage for a given function also satisfies decision coverage for that function. Such a relation between coverage criteria is often called *subsume*: a coverage C_1 *subsumes* a coverage C_2, if for all functions f (or for all programs p) and all specifications s all sets T of test cases that satisfy C_1 also satisfy C_2 [4, 11, 15]. The specification of the program is not used in our coverage definitions and therefore left out in this paper.

4.3. Comparison of Coverage Definitions. The example `f2` (Figure 4) shows that in the case of unconditional loops all DD-paths and thus all edges in the decision graph can be covered (e.g., by the test case with parameter 0) but not all edges in the control flow graph are executed. This leads to the following lemma.

Lemma 11. *Let f be a function such that the control flow graph $G_f = (N, E)$ of f does not contain unconditional loops, and let T be a set of test cases that satisfies branch coverage. Then, T also satisfies edge coverage and decision coverage.*

Proof. Let $e \in E$ be an edge in the control flow graph. From part (3) of Lemma 6, follows that there exists a DD-path d' that contains e. This DD-path induces an edge $\hat{e} \in \hat{E}$ in the decision graph $\hat{G} = (\hat{N}, \hat{E})$ of G (Definition 4). Since the set T of test cases satisfies branch coverage, a path $\hat{d} \in \hat{\mathrm{D}}(f, T)$ with $\hat{e} \in E(\hat{d})$ exists (Definition 10), and furthermore there is a path $d \in \mathrm{D}(f, T)$ such that $\hat{d}$ is the decision path of d (Definition 9). Since the edge $\hat{e}$ occurs in the decision path $\hat{d}$, we know from Definition 5 that d' is part of the path d and thus $e \in E(d)$. □

Unconditional loops can be allowed in cases where all test case sets that satisfy branch coverage also run through all unconditional loops. Figure 6 shows the control flow graph of the following function `f3`:

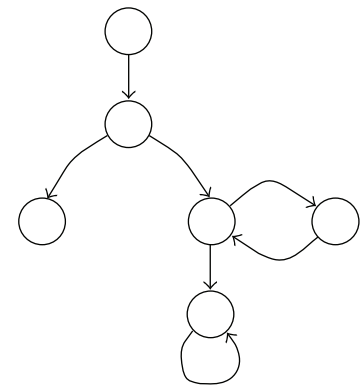


Figure 6: Another control flow graph with unconditional loop.

```
void f3(int x)
{
  if (x > 0)
  {
    while (x > 0)
      x = x - 1;
    label: goto label;
  }
}
```

In this example, we need at least two test cases, one with a positive and one with a nonpositive parameter, in order to satisfy branch coverage. The test case with positive parameter executes also the unconditional loop.

Nonterminating function calls result in incomplete test case paths. For example, when we assume that the function `g` called in `f1` (Figure 3) does not terminate if it is called with the parameter 0, the path induced when the function `f1` is called with the parameter 0 does not lead to the exit node. The consequence is that all edges are covered when the function `f1` is called with parameters 0 and 1 but the DD-path $n_2n_4n_5n_6$ and thus the corresponding edge in the decision graph is not executed.

Definition 12. Let $G = (N, E)$ be a directed graph. A set D of paths is called *complete*, if for each path $d' \in D$ that does not end in an exit node, there exists a path $d \in D$ with $d' < d$.

Infinite sequences $d_1 < d_2 < d_3 < \cdots$ of paths (which are induced in the control flow graph of a function by infinite loops) are allowed in complete sets of path, but not paths that stop before reaching an exit node. The set $\mathrm{D}(\texttt{f1}, t)$ where t is the test case with parameter 0 is not a complete set since $n_1n_2n_4n_5 \in \mathrm{D}(\texttt{f1}, t)$ but $n_1n_2n_4n_5n_6 \notin \mathrm{D}(\texttt{f1}, t)$.

Lemma 13. *Let $G = (N, E)$ be a directed graph, D a complete set of paths, and $d' = n_1 \ldots n_{k+1}$ a path with inner nodes that are not D-nodes such that there exists a path $d'' \in D$ with*

$(n_1, n_2) \in E(d'')$. *Then, there is a path* $d \in D$ *that contains* d'.

Proof. If $k = 1$, we have $d' = n_1 n_2$ and d' is contained in d'' since $(n_1, n_2) \in E(d'')$. Assume that the proposition holds for all paths d' with inner nodes that are not D-nodes with length k. Let $d' = n_1 \cdots n_{k+2}$ be a path with length $k+1$ where $n_2 \cdots n_{k+1}$ are not D-nodes such that a path $d'' \in D$ with $(n_1, n_2) \in E(d'')$ exists. From the assumption, it follows that there is a path $d \in D$ that contains $n_1 \cdots n_{k+1}$. If n_{k+1} is the last node in d, we can prolong d uniquely by n_{k+2} since n_{k+1} is not a D-node. This new path contains d' and is in D since D is complete and n_{k+1} is not an exit node (Definition 12). If n_{k+1} is not the last node in d, the node following n_{k+1} in d must be n_{k+2}, since n_{k+1} is not a D-node, and therefore d contains d'. □

When we apply this lemma to DD-paths d', we can prove the following proposition.

Lemma 14. *Let* f *be a function and* T *a set of test cases that satisfies edge coverage or decision coverage such that* $\mathrm{D}(f, T)$ *is complete. Then,* T *also satisfies branch coverage.*

Proof. Let $\hat{e}$ be an edge in the decision graph $\widehat{G} = (\widehat{N}, \widehat{E})$ of the control flow graph G of f. Then, there exists a DD-path $d' = n_1 \cdots n_{k+1}$ in G such that $\hat{e}$ is the associated edge in $\widehat{E}$ (Definition 4). Since the set T of test cases satisfies edge coverage of G or decision coverage, a path $d'' \in \mathrm{D}(f, T)$ with $(n_1, n_2) \in E(d'')$ exists (Definition 10). The set $\mathrm{D}(f, T)$ is complete, and therefore there is a path $d \in \mathrm{D}(f, T)$ that contains d' (Lemma 13). It follows from Definition 5 that the decision path $\hat{d}$ of d contains the edge $\hat{e}$. This means that for the path $\hat{d} \in \widehat{\mathrm{D}}(f, T)$ holds $\hat{e} \in E(\hat{d})$. □

Since branch coverage does not cover branches with unconditional loops, we could weaken the condition for complete sets and allow that the executions of unconditional loops are not fully contained in the path sets, that is, a path $d' \in D$ is allowed to end in a node that is contained in an unconditional loop without the existence of a path $d \in D$ with $d' < d$.

In the case that a control flow graph has an isolated loop, it is impossible to get edge coverage but decision coverage possibly can be achieved. Of course, such a loop can only appear in functions with unreachable code. This is not the only case in which edge coverage does not follow from decision coverage. Another case arises if the set of paths induced by the test cases is not complete. When, for example, the function `f1` in Figure 3 is called with 0 and 1, we get decision coverage, but not edge coverage, if we assume that g never terminates.

Lemma 15. *Let* f *be a function such that the control flow graph* $G_f = (N, E)$ *of* f *does not contain isolated loops and let* T *be a set of test cases that satisfies decision coverage such that* $\mathrm{D}(f, T)$ *is complete. Then,* T *also satisfies edge coverage.*

Proof. Let e be an edge in the control flow graph. Let $d' = n_1 n_2 \cdots n_{k+1}$ be the path according to part (2) of Lemma 6 that contains e as last edge. Since the set T of test cases satisfies decision coverage, a path $d'' \in \mathrm{D}(f, T)$ with $(n_1, n_2) \in E(d'')$ exists (Definition 10). From Lemma 13, follows that there is a path $d \in \mathrm{D}(f, T)$ that contains d'. Therefore, $e \in E(d)$. □

If we exclude unconditional and thus isolated loops, for example, by not allowing gotos, which is a simple syntactical criterion, we can summarize the results as follows.

Theorem 16. *In the set of all functions with control flow graphs without unconditional loops,*

> *branch coverage subsumes edge coverage and edge coverage subsumes decision coverage.*

For functions f *(with control flow graphs without unconditional loops) and test case sets* T *that induce complete sets* $\mathrm{D}(f, T)$ *of paths the reverse directions also hold, that is, from decision coverage follows edge coverage and from edge coverage follows branch coverage.*

5. Related Work and Applications of Control Flow Graphs

Control flow graphs can be used for white box testing to support test data selection and coverage notions as shown in [5] for statement, segment, and branch coverage or as discussed by Laski and Korel [16] and Rapps and Weyuker [12] for data flow oriented testing. In the latter papers, only complete paths, that is, paths that start in the entry node and end in the exit node, are considered, whereas in the first paper also, paths that do not end in the exit node are allowed in order to capture infinite loops which can occur in practical applications that run until switched off, for example, in embedded control systems. Jalote [10] and Zhu et al. [4] base the definitions of statement and branch coverage and of data flow coverage notions on control flow graphs, where the nodes represent blocks of statements. Different coverage definitions are compared in [4, 11]. In the first of these papers, the authors argue that a coverage that subsumes another coverage does not necessarily give better results with respect to the detection of faults and introduce a relation called "properly covers" with which they prove that decision coverage is weaker than condition based and data flow oriented coverages. White [17] models the structure of programs with control flow graphs in order to discuss different aspects of testing. Program transformation techniques also use control flow graphs to represent the program structure, for example, as shown by Hierons et al. [13] with the aim to apply automated test data generation to transformed unstructured programs. An approach to generate test data that uses control flow graphs to describe all paths that lead from the entry node to the branch which should be tested is shown in [18]. Bertolino and Marré [14] propose an algorithm to generate path covers for branch testing which is based on ddgraphs that reduce graphs to D-nodes and junction nodes and the paths between them. The

difference between ddgraphs and our decision graphs is the inclusion of the junction nodes in ddgraphs.

Another principal usage of control flow graphs is control flow analysis in compiler construction and optimization [19]. Aho et al. [3] use control graphs to represent intermediate code in the form of three address statements for code generation during the compilation of programs. These statements have the form `x = y op z` or are unconditional goto-statements `goto label` or conditional goto-statements `if (condition) goto label`. A conditional goto is treated as one statement. Nodes represent basic blocks of sequential statements, which can be entered only by the first statement in the block and left by the last statement. Entry and exit nodes are separate nodes and not part of blocks. Ferrante et al. [1] derive program dependence graphs from control flow graphs that describe the data and control dependences in the program and use them for transformation and optimization of programs.

Kosaraju [20] defines flow charts recursively using different types of basic constructs and compares them to study the computational power of the underlying constructs. Analysis of programs by partitioning using segments, DD-paths, and other approaches is discussed by Paige [7].

A further application is to support the definition and evaluation of source-code-based metrics. For example, cyclomatic complexity can be, based on the cyclomatic number in graph theory [21], defined by counting the linearly independent circuits in the graphs [10, 22]. Sommerville [2] combines cyclomatic complexity and independent paths to design test cases in the white box test. Cyclomatic complexity for sets of functions can be defined in several ways. In the original paper, McCabe [22] defines the complexity of p components by $v = e - n + 2p$, where e is the number of edges and n the number of nodes in the components. Henderson-Sellers and Tegarden [23] argue that if the components represent calling and called functions the control flow graphs of the called functions can be expanded in the control flow graph of the caller. Function call nodes are split into two nodes, a call node and a return-from-node with additional edges from the call node to the entry node of the called function and from the exit node of the called function to the return-from-node. Thus, $2(p-1)$ edges and $p-1$ nodes are added, if the p components consist of one calling and $p-1$ called functions. The expanded graph consists of one component and defines an alternative cyclomatic complexity of a set of functions: $v_{\mathrm{LI}} = (e + 2(p - 1)) - (n + p - 1) + 2 = e - n + p + 1$. In [6], we compare the cyclomatic complexity of a control flow graph and that of its decision graph and prove that $v(G_f) = v(\widehat{G}_f) + 1$.

In order to do interprocedural analysis, Reps et al. [24] define a framework that consists of the set of control flow graphs for all functions in a program using the technique of node-splitting and expansion as described above. For data-flow analysis, especially, the interprocedural approach gives much better results than intraprocedural analysis [25]. Kapfhammer [15] defines test coverage notions based on interprocedural control flow graphs. When classes are considered, interprocedural control flow graphs can be restricted to the methods of single classes. With this approach, Harrold and Rothermel [26] give a framework for data flow oriented testing of classes. One difference between procedural and object-oriented programming languages is polymorphism. In another paper, Harrold and Rothermel [27] solve this by the introduction of polymorphic call and return nodes.

So far, all mentioned approaches modeled the control flow on the level of higher programming languages or intermediate level. But it is also possible to analyze the control flow of machine-level programs.

One application of interprocedural control flow graphs on the lower level is the detection of self-mutating malware. Bruschi et al. [28] try to find the control flow graph of the searched malicious code as subgraph in the control flow graphs of the program and thus identify malware.

Abadi et al. [29] use the control flow graph of machine-level programs to detect deviations from the control flow caused by attacks on the program. Computed jumps especially, have to be secured against destination addresses forged by attackers.

Usually, the control flow graphs are known when software is analyzed. If only the executable code is available, the control flow graphs have to be extracted before the analysis can take place. Various problems make the construction of the graphs imprecise, for example, when jump tables with data dependent target addresses are used. Theiling [30] describes a software framework that extracts the control flow graphs such that they can be used in a safe analysis of the worst-case execution time (WCET). An approach to construct the control flow graphs based on XML representations of the executable code in assembly form is proposed by Wenjian et al. [31].

Several tools exist that visualize control flow graphs. Figure 7 shows the control flow graph of the function `Search` generated by Crystal FLOW from SGV Software Automation Research Corporation (http://www.sgvsarc.com/). Such tools usually show more information in the graphs in order to support the understanding of the code or to be used in code reviews or documentation. A tool that uses control flow graphs to show the results of program analysis, in this case the WCET, is aiT by AbsInt (http://www.absint.com/ait/). With the visualization tool aiSee, the user can explore the graphs and thus inspect the WCET analysis results [32].

Control flow graphs can also be applied to the testing of hardware descriptions in VHDL. Zhang and Harris [33] introduce timing nodes to represent the timing information in VHDL descriptions and define data flow oriented du pairs coverage for hardware descriptions. Flow graphs are also useful in business process modeling. Sadiq and Orlowska [34] define workflow graphs where nodes represent tasks and edges represent the workflow between tasks. In this paper, workflow graphs are defined as acyclic graphs. A special iteration task is used to express the repetition of tasks. Workflow graphs are checked for deadlocks and lack of synchronization by graph reduction. Workflow charts that model human-computer interactions that support business processes and allow loops are studied in [35]. There, computer screens, forms and links are modeled by nodes in the graphs. One difference between these graphs and control flow graphs is that in order to model the workflow concurrency is necessary.

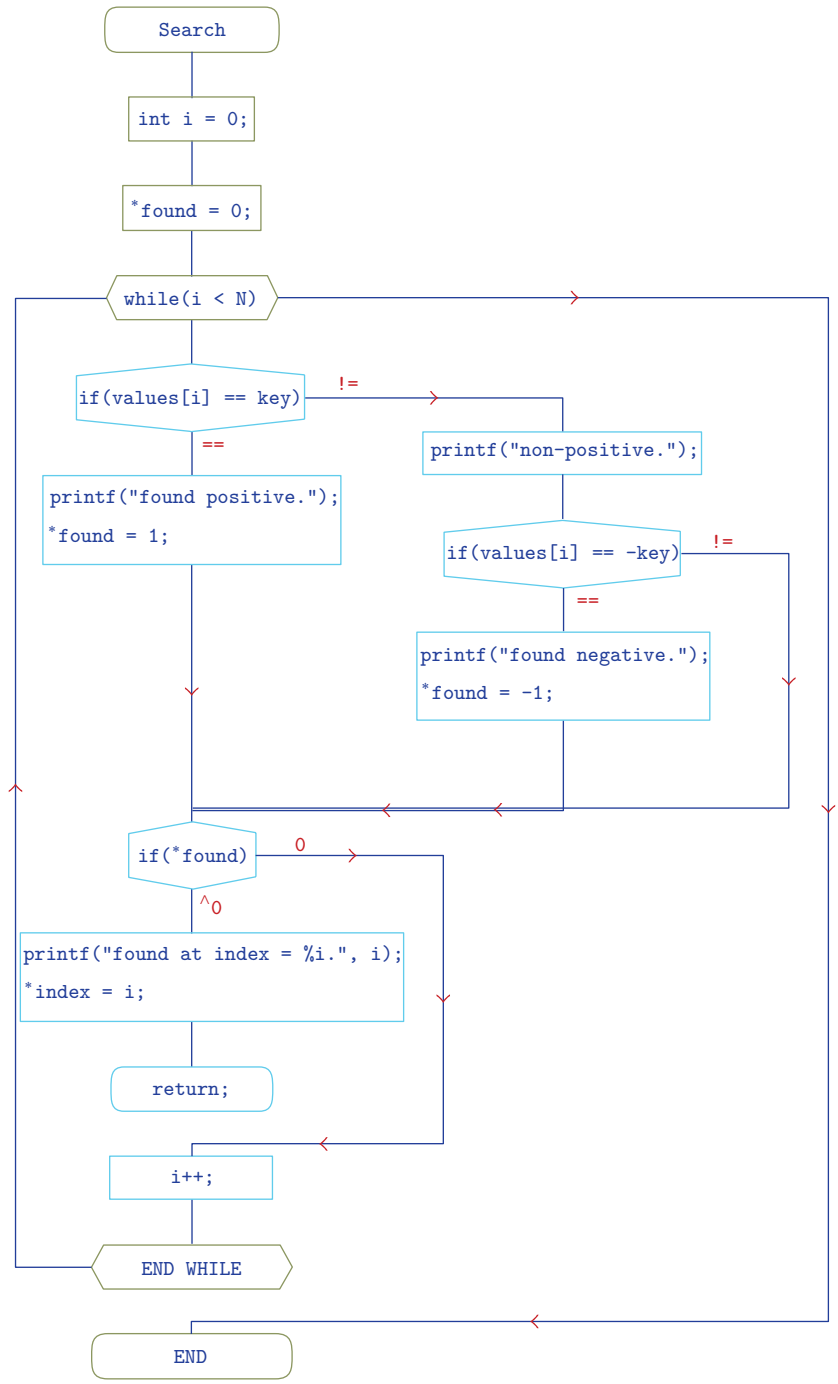


Figure 7: The control flow graph of the function `Search` generated by Crystal FLOW.

In the testing of graphical user interfaces, event flow graphs model the events that occur as reaction to the interaction of the user with the interface [36, 37]. Coverage criteria are based on paths in the graphs modeling event sequences. The analogue to edge coverage is called by Memon et al. [38] event selection coverage because an edge (e, e') models the selection of an event e' after the event e occurred. Like program decomposition into functions, graphical user interfaces are build up of components and intercomponent criteria can be defined.

Decision trees, introduced by Raiffa and Schlaifer [39], are well known in probability theory. A decision tree consists of decision nodes where decisions are taken and of chance nodes where unknown states are modeled by different successors with assigned probabilities. The leaves of decision trees are utility nodes that specify the outcome of the decisions. Each path from the root to a leaf thus models a sequence of decisions and state assumptions that lead to the outcome under assigned probabilities. The drawback that decision trees grow exponentially with the number of decisions can be

solved by more general structures such as influence diagrams [40]. The main difference between these graph types and our decision graphs is that decision trees and influence diagrams are acyclic, and thus the length of decision sequences have always a fixed upper bound. Oliver [41] defines decision graphs as generalizations of decision trees where duplicated subtrees are joined and applies them to construct decision procedures from sets of examples. A practical application of decision trees is shown for example in [42] where decision trees for the prediction of the diagnosis and the outcome of Dengue illness are constructed from simple clinical and haematological data of 1200 patients using a decision tree classifier software tool. The authors of this study state as a conclusion that their algorithms are expected to help disease management.

6. Conclusion

In this paper, we derived decision graphs from directed graphs such that the branching structure is preserved. It can be shown that the branches in a graph without unconditional loops correspond to the edges in the decision graph. One useful application is the modeling of programs with control flow graphs. Decision graphs form an abstraction from control flow graphs that display only the decisions, for example, if-then-else-constructs and the paths between decisions to the programmer. With this approach, we compared different definitions of branch covering in software testing that already existed and showed the differences. When we exclude unconditional loops, branch coverage based on the edges in decision graphs subsumes edge coverage of the control flow graph and decision coverage. Control flow graphs are not only popular in software modeling but also popular in different other fields. Therefore, it seems promising to apply decision graphs to other domains and exploit their advantages.

Acknowledgment

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

References

[1] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.

[2] I. Sommerville, *Software Engineering*, Pearson Education Limited, Boston, Mass, USA, 7th edition, 2004.

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Longman, Amsterdam, The Netherlands, 2nd edition, 2007.

[4] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, 1997.

[5] R. Gold, "Control flow graphs and code coverage," *International Journal of Applied Mathematics and Computer Science*, vol. 20, no. 4, pp. 739–749, 2010.

[6] R. Gold, "On cyclomatic complexity and decision graphs," in *Proceedings of the 10th International Conference of Numerical Analysis and Applied Mathematics (ICNAAM '12)*, vol. 1479 of *AIP Conference Proceedings*, pp. 2170–2173, Kos, Greece, September 2012.

[7] M. R. Paige, "On partitioning program graphs," *IEEE Transactions on Software Engineering*, vol. 3, no. 6, pp. 386–393, 1977.

[8] J. A. Bondy and U. S. R. Murty, *Graph Theory*, Springer, London, UK, 2008.

[9] R. Diestel, *Graph Theory*, Springer, New York, NY, USA, 4th edition, 2010.

[10] P. Jalote, *An Integrated Approach to Software Engineering*, Springer, New York, NY, USA, 3rd edition, 2005.

[11] P. G. Frankl and E. J. Weyuker, "Provable improvements on branch testing," *IEEE Transactions on Software Engineering*, vol. 19, no. 10, pp. 962–975, 1993.

[12] S. Rapps and E. J. Weyuker, "Data flow analysis techniques for test data selection," in *Proceedings of the 6th International Conference on Software Engineering (ICSE '82)*, pp. 272–278, Tokyo, Japan, 1982.

[13] R. M. Hierons, M. Harman, and C. J. Fox, "Branch-coverage testability transformation for unstructured programs," *The Computer Journal*, vol. 48, no. 4, pp. 421–436, 2005.

[14] A. Bertolino and M. Marré, "Automatic generation of path covers based on the control flow analysis of computer programs," *IEEE Transactions on Software Engineering*, vol. 20, no. 12, pp. 885–899, 1994.

[15] G. M. Kapfhammer, "Software testing," in *Computer Science Handbook*, A. B. Tucker, Ed., Chapman & Hall/CRC, 2nd edition, 2004.

[16] J. W. Laski and B. Korel, "A data flow oriented program testing strategy," *IEEE Transactions on Software Engineering*, vol. 9, no. 3, pp. 347–354, 1983.

[17] L. J. White, "Basic mathematical definitions and results in testing," in *Computer Program Testing*, B. Chandrasekaran and S. Radicchi, Eds., North-Holland, New York, NY, USA, 1981.

[18] N. Gupta, A. P. Mathur, and M. L. Soffa, "Generating test data for branch coverage," in *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, pp. 219–227, Grenoble, France, September 2000.

[19] F. E. Allen and J. Cocke, "Graph-theoretic constructs for program control flow analysis," Research Report RC 3923, IBM Research, 1972.

[20] S. R. Kosaraju, "Analysis of structured programs," in *Proceedings of the 5th Annual ACM Symposium on Theory of Computing*, pp. 240–252, Austin, Tex, USA, 1973.

[21] C. Berge, *The Theory of Graphs*, Dover, New York, NY, USA, 2001.

[22] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, 1976.

[23] B. Henderson-Sellers and D. Tegarden, "The theoretical extension of two versions of cyclomatic complexity to multiple entrylexit modules," *Software Quality Journal*, vol. 3, no. 4, pp. 253–269, 1994.

[24] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 49–61, January 1995.

[25] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," *Electronic Notes in Theoretical Computer Science*, vol. 217, pp. 5–21, 2008.

[26] M. J. Harrold and G. Rothermel, "Performing data flow testing on classes," in *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '94)*, pp. 154–163, December 1994.

[27] M. J. Harrold and G. Rothermel, "A coherent family of analyzable graphical representations for object-oriented software," Tech. Rep. OSUCISRC-11/96-TR60, Department of Computer and Information Science, The Ohio State University, 1996.

[28] D. Bruschi, L. Martignoni, and M. Monga, "Detecting self-mutating malware using control-flow graph matching," in *Proceedings of the 3rd International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA '06)*, R. Büschkes and P. Laskov, Eds., vol. 4064 of *Lecture Notes in Computer Science*, pp. 129–143, Springer, Berlin, Germany, July 2006.

[29] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security*, vol. 13, no. 1, article 4, 40 pages, 2009.

[30] H. Theiling, "Extracting safe and precise control flow from binaries," in *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications*, pp. 23–30, Cheju Island, South Korea, December 2000.

[31] Y. Wenjian, J. Liehui, Y. Qing et al., "A control flow graph reconstruction method from binaries based on XML," in *Proceedings of the International Forum on Computer Science-Technology and Applications (IFCSTA '09)*, pp. 226–229, Chongqing, China, December 2009.

[32] L. Tan, "The worst case execution time tool challenge 2006," Technical Report for the External Test STTT, TR 045, ICB/Computer Science, University of Duisburg-Essen, 2007.

[33] Q. Zhang and I. G. Harris, "A data flow fault coverage metric for validation of behavioral HDL descriptions," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD '00)*, San Jose, Calif, USA, November 2000.

[34] W. Sadiq and M. E. Orlowska, "Analyzing process models using graph reduction techniques," *Information Systems*, vol. 25, no. 2, pp. 117–134, 2000.

[35] D. Draheim, *Business Process Technology: A Unified View on Business Processes, Workflows and Enterprise Applications*, Springer, Berlin, Germany, 2010.

[36] P. A. Brooks and A. M. Memon, "Automated GUI testing guided by usage profiles," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*, pp. 333–342, Atlanta, Ga, USA, November 2007.

[37] A. M. Memon, "An event-flow model of GUI-based applications for testing," *Software Testing Verification and Reliability*, vol. 17, no. 3, pp. 137–157, 2007.

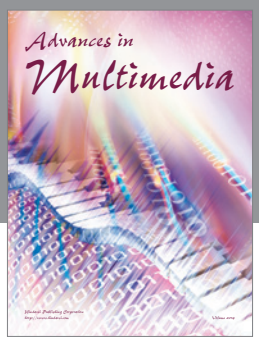
[38] A. M. Memon, M. L. Soffa, and M. E. Pollack, "Coverage criteria for GUI testing," in *Proceedings of the 8th European Software Engineering Conference with the 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pp. 256–267, Vienna, Austria, September 2001.

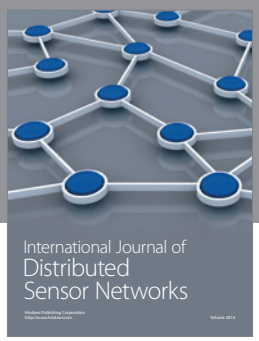
[39] H. Raiffa and R. Schlaifer, *Applied Statistical Decision Theory*, Division of Research, Graduate School of Business Administration, Harvard University, 3rd edition, 1964.

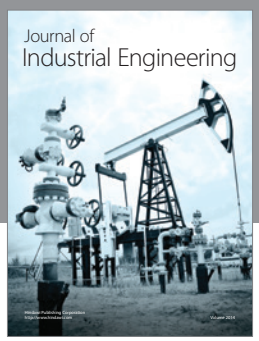
[40] F. V. Jensen and T. D. Nielsen, *Bayesian Networks and Decision Graphs*, Springer, New York, NY, USA, 2nd edition, 2007.

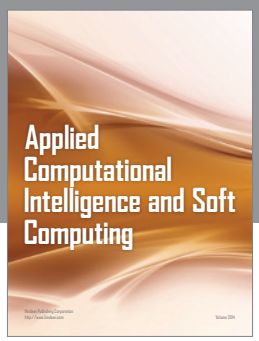
[41] J. J. Oliver, "Decision graphs—an extension of decision trees," Tech. Rep. CS 92/173, Department of Computer Science, Monash University, Melbourne, Australia, 1992.

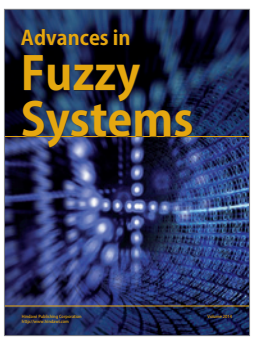
[42] L. Tanner, M. Schreiber, J. G. H. Low et al., "Decision tree algorithms predict the diagnosis and outcome of dengue fever in the early phase of illness," *PLoS Neglected Tropical Diseases*, vol. 2, no. 3, article e196, 2008.











Hindawi

Submit your manuscripts at
http://www.hindawi.com