World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:8, No:3, 2014

Reductions of Control Flow Graphs

Robert Gold

Open Science Index, Computer and Information Engineering Vol:8, No:3, 2014 publications.waset.org/9997636/pdf

Abstract—Control flow graphs are a well-known representation of the sequential control flow structure of programs with a multitude of applications. Not only single functions but also sets of functions or complete programs can be modeled by control flow graphs. In this case the size of the graphs can grow considerably and thus makes it difficult for software engineers to analyze the control flow. Graph reductions are helpful in this situation. In this paper we define reductions to subsets of nodes. Since executions of programs are represented by paths through the control flow graphs, paths should be preserved. Furthermore, the composition of reductions makes a stepwise analysis approach possible.

Keywords—Control flow graph, graph reduction.

I. Introduction

CONTROL flow graphs are widely used since many years to represent the sequential control structure of software and for its graphical representation. Each statement is represented by a node in the graph, the edges reflect the control flow between them. Among the manifold applications of control flow graphs are white box testing, test coverage notions [11], [12], [21], [23], [28], [35], [37] and generation of test cases [4], [16], [20]. Furthermore, control flow graphs are useful in the control flow analysis in compiler construction and optimization [2], [3], [10] and in the definition and evaluation of source code metrics [13], [19], [21], [31], such as, cyclomatic complexity [24].

Control flow graphs can not only be applied to single functions in a program, but there are also approaches for interprocedural frameworks [17], [18], [22], [29] that define control flow graphs for sets of functions or for complete programs. Each node that represents a function call is split in a call node and a return node and linked to the control flow graph of the called function by newly introduced edges. The problem with interprocedural control flow graphs is their large size. Especially, when software engineers use control flow graphs in program analysis the size of the graphs makes it difficult to understand the control flow structure. Therefore, the user should be enabled to apply reductions to the graphs that preserve the control flow and other important properties. One reduction that can be found very often in the related literature is to merge each maximal block of consecutive nodes with a single entry and a single exit, e.g., [2], [7], [11], [21]. Sequential statements are then represented by a single node. The size of the graphs is decreased, but in most cases only slightly. One property of this reduction is that the paths in the original control flow graph can be reconstructed when in the paths in the reduced graph the blocks are expanded again. In previous papers [12], [13], [14], [15] we adopted the approach of Paige [27] and defined a reduction where only entry nodes and exit nodes of the control flow graph and such nodes are kept that represent decisions, for example, if-statements. The resulting graphs are called decision graphs. Each edge in the decision graph corresponds to a branch in the program [14]. We reduce branches to single edges and get smaller graphs with less nodes. Since the branching structure of the programs is preserved, decision graphs can be used in branch testing and in the analysis of cyclomatic complexity [13]. Decision graphs can also be employed in the design of algorithms and functions to help the designer to focus on the decision structure.

In this paper we will generalize this approach of graph reductions with special focus to interprocedural control flow graphs. We will define a general reduction method to arbitrary subsets of nodes and give examples such as the reduction to the interface nodes and the reduction to the D-nodes that leads to decision graphs. Paths in the control flow graph represent executions of the program. Therefore, the control flow should be preserved by the reduction in the sense that the paths in the control flow graph should correspond to the paths in the reduced graph. This enables the software engineer to analyze the executions also in the reduced graph.

The contribution of this paper is to provide a general framework for graph reductions that preserve paths. The reductions can be applied to control flow graphs but are not restricted to them. This opens a wide field of other applications of graphs for graph reductions. We show that reductions can be applied to already reduced graphs and define the composition of reductions. That makes it possible to "zoom out" of a graph in several steps until the desired level of information details is reached, which supports the analysis of programs.

The remainder of this paper is organized as follows. We start with the necessary definitions about directed graphs and control flow graphs. In Section III we define the graph reduction and investigate its properties. The following chapter extends the reduction to interprocedural control flow graphs of sets of functions or of complete programs and shows the compositionality of reductions. Section V concludes the paper.

II. Basic Definitions

In this section definitions about graphs necessary for the following are summarized. The definitions are partly taken from previous papers [14], [15], a detailed introduction to graphs can be found, e.g., in [5] or [8].

Definition 1. A *directed graph (with multiple edges)* is a pair $G = (N, E)$ consisting of a finite set N of *nodes* and a (possibly infinite) set E of *edges* with $N \cap E = \emptyset$, together with functions $\mathrm{start} : E \to N$ and $\mathrm{end} : E \to N$ that associate a *start node* and an *end node*, respectively, with each edge.

R. Gold is with the Faculty of Electrical Engineering and Computer Science, Technische Hochschule Ingolstadt, Esplanade 10, 85049 Ingolstadt, Germany, (e-mail: robert.gold@thi.de).

Open Science Index, Computer and Information Engineering Vol:8, No:3, 2014 publications.waset.org/9997636/pdf

The *indegree* and *outdegree* of a node n is the number of edges that end and start in n, respectively, i.e., $\mathrm{indegree}(n) = |\{e \in E \mid \mathrm{end}(e) = n\}|$, $\mathrm{outdegree}(n) = |\{e \in E \mid \mathrm{start}(e) = n\}|$, or ω if there are infinitely many such edges. A node n with $\mathrm{indegree}(n) = 0$ is called *entry node* and a node with $\mathrm{outdegree}(n) = 0$ *exit node* of the graph.

A *path* d in a graph is a non-empty, 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 nodes $\mathrm{start}(e_1)\,\mathrm{start}(e_2) \ldots \mathrm{start}(e_k)\,\mathrm{end}(e_k)$ in the path d form a sequence denoted by $\mathrm{nodes}(d)$. 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, denoted by $\mathrm{start}(d)$ and $\mathrm{end}(d)$. The nodes $\mathrm{start}(e_2), \ldots, \mathrm{start}(e_k)$ are called *inner nodes* of d. The *concatenation* of two paths $d = e_1\, e_2\, \ldots e_k$ and $d' = e'_1\, e'_2\, \ldots e'_m$ is defined by $d\, d' = e_1\, e_2\, \ldots e_k\, e'_1\, e'_2\, \ldots e'_m$ if $\mathrm{end}(e_k) = \mathrm{start}(e'_1)$.

We represent multiple edges between two nodes graphically as one arc with the number of edges between these two nodes as a label at the arc. Infinitely many edges are denoted by ω. If there is only one edge between two nodes the number 1 is omitted.

As mentioned in the introduction, each statement in a function is represented by a node in the control flow graph. The control flow between statements is modeled by the edges. An additional entry node n_{in} and an additional exit node n_{out} is added that identify the entry when called and the exit when the function returns. In this chapter and Chapter III we will examine only control flow graphs of single functions. Therefore the exit from and the entry to the graph when a function is called within the function are not represented as exit and entry nodes. The examples in this paper are mostly written in the programming language C, but of course control flow graphs are not restricted to this language. Which constructs are represented by nodes is not always consistent in the literature. We consider such constructs which are syntactically statements in the programming language C [14].

Definition 2. The *control flow graph* $G_f = (N, E)$ of a function f is a directed graph that consists of

- a node n_a for each statement a in f,
- one additional entry node n_{in} and one additional exit node n_{out},
- an edge $(n_a, n_{a'})$ if the statement a' is executed immediately after the statement a,
- an edge $(n_{\mathrm{in}}, n_{a_1})$ for the first statement a_1 in f or an edge $(n_{\mathrm{in}}, n_{\mathrm{out}})$ if f contains no statements at all,
- edges $(n_{a'}, n_{\mathrm{out}})$ for each statement a' after which the control flow leaves the function. This could be because of a return-statement or when the right brace that terminates the function is reached.

The definition ensures that in the control flow graph G_f of a function f each node—with the exception of n_{in} and n_{out}—corresponds to a unique statement in the function. Between two nodes there is at most one edge. Therefore control flow graphs don't have multiple edges and their sets of edges are finite.

As an example, the following function `find` which looks in a linked list for the last name and the first name of a person is examined:

```
list find(list namelist,
          string last, string first)
{
  printf("start ");

  while (namelist != NULL)
  {
    if (namelist->lastname == last)
    {
      printf("lastname found ");

      if (namelist->firstname == first)
      {
        printf("name found ");
        return namelist;
      }
    }

    printf("next name ");
    namelist = namelist->next;
  }

  printf("name not found ");
  return NULL;
}
```

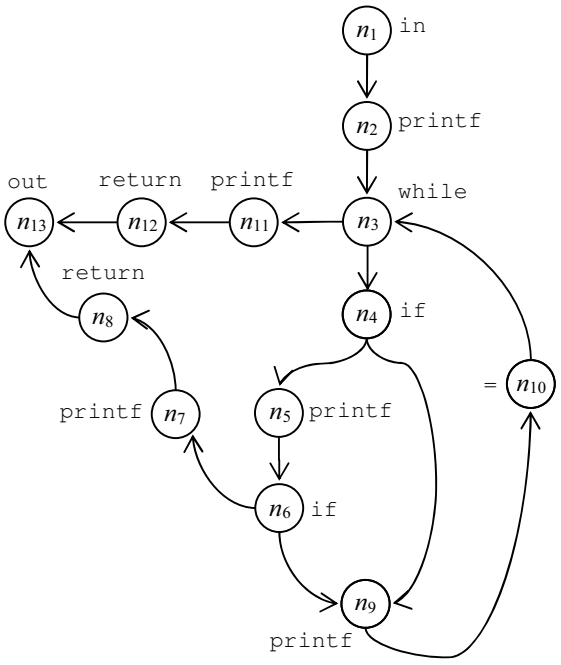


Fig. 1. The control flow graph of the function `find`

Fig. 1 shows the control flow graph of this function. For better readability the nodes in the control flow graph can be labeled. The labels "in" and "out" are assigned to the entry and exit nodes. The other nodes are labeled by "=", "while" etc. to show which kind of statements is represented.

III. Graph Reductions

In a graph reduction we select a subset of nodes and thus decrease the number of nodes. But paths should be preserved.

Open Science Index, Computer and Information Engineering Vol:8, No:3, 2014 publications.waset.org/9997636/pdf

Therefore, for each path between two nodes in the subset an edge is introduced in the reduced graph. Before we define the graph reduction we need the following notation about sets of paths in directed graphs.

Definition 3. Let $G = (N, E)$ be a directed graph and let $N' \subseteq N$ be a subset of its nodes. We will examine the following sets of paths in G:

- D: set of all paths in G,
- $D_{N'NN'}$: set of all paths in G, whose start and end nodes are in N',
- $D_{N'\bar{N}'N'}$: set of all paths in G, whose start and end nodes are in N' with inner nodes in $N \setminus N'$. Such paths are called $N'N'$-*paths*.

The difference between the sets $D_{N'NN'}$ and $D_{N'\bar{N}'N'}$ is that the inner nodes in the paths in the first set are not restricted to $N \setminus N'$ and can be any nodes from N.

When we reduce a graph to a set $N' \subseteq N$ of nodes, each $N'N'$-path between two nodes in N' will be replaced by an edge. This defines a bijection between $N'N'$-paths in G and edges in the reduced graph which keeps the start and end nodes.

Definition 4. Let $G = (N, E)$ be a directed graph and let $N' \subseteq N$ be a subset of its nodes. A directed graph $G' = (N', E')$ is a *reduction* of G to N' if there a bijective function

$$\delta : D_{N'\bar{N}'N'} \to E'$$

from the set of the $N'N'$-paths in G to the set of edges in G' such that

$$\begin{aligned} \mathrm{start}(d) &= \mathrm{start}(\delta(d)) \\ \mathrm{end}(d) &= \mathrm{end}(\delta(d)) \end{aligned}$$

for all $N'N'$-paths $d \in D_{N'\bar{N}'N'}$.

All reductions of G to the same subset of nodes N' are equal up to isomorphism. Therefore, we speak of *the* reduction of G to N'.

Since there may be more than one or infinitely many $N'N'$-paths in G between two nodes $n, m \in N'$—even if G has no multiple edges—, multiple edges are necessary to represent them in the reduced graph. The number of edges in the reduced graph G' between two nodes n and m is equal to the number of different $N'N'$-paths in G that start in n and end in m.

Clearly, the graph $G = (N, E)$ reduced to N is equal to G.

In the control flow graph of the function `find` there are infinitely many paths between the entry node and the exit node described by the regular expression

$$\begin{aligned} &n_1\, n_2 (n_3\, n_4 (n_5\, n_6\, n_9 \mid n_9) n_{10})^*\, n_{11}\, n_{12}\, n_{13} \mid \\ &n_1\, n_2 (n_3\, n_4 (n_5\, n_6\, n_9 \mid n_9) n_{10})^*\, n_3\, n_4\, n_5\, n_6\, n_7\, n_8\, n_{13} \end{aligned}$$

but, of course no path between the exit node and the entry node. The graph reduced to the entry node and the exit node is shown in Fig. 2.

In [14] and [15] we defined the reduction to the D-nodes, that are entry nodes, exit nodes and such nodes that represent decisions, that are, nodes with outdegree ≥ 2. This approach

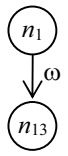


Fig. 2. The control flow graph of the function `find` reduced to the entry node and the exit node

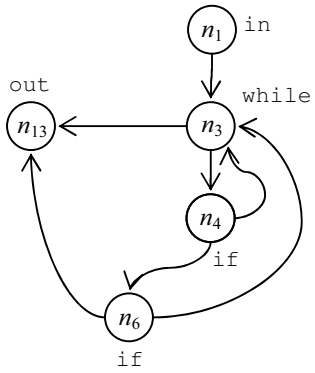


Fig. 3. Decision graph of the control flow graph of the function `find`

was introduced by Paige [27] in order to support the analysis of programs by partitioning the control flow graph.

Definition 5. 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 $\mathrm{outdegree}(n) \geq 2$. The reduction of G to its D-nodes is called *decision graph* of G.

A node that is not a D-node has indegree ≥ 1 and outdegree exactly 1.

In [14] the $N'N'$-paths where N' is the set of D-nodes are called *DD-paths*. Since there is no branching possible after leaving a D-node n until reaching another D-node, all inner nodes in a DD-path are different [12] and there are at most $\mathrm{outdegree}(n)$ different DD-paths that start in n. The decision graph of a control flow graph has therefore only finitely many edges.

The D-nodes in the control flow graph of the function `find` (Fig. 1) are the entry and exit nodes n_1, n_{13} and the while- and if-nodes n_3, n_4, n_6. The paths

$$\begin{aligned} &(n_1, n_2)(n_2, n_3) \\ &(n_3, n_{11})(n_{11}, n_{12})(n_{12}, n_{13}) \\ &(n_3, n_4) \\ &(n_4, n_5)(n_5, n_6) \\ &(n_4, n_9)(n_9, n_{10})(n_{10}, n_3) \\ &(n_6, n_9)(n_9, n_{10})(n_{10}, n_3) \\ &(n_6, n_7)(n_7, n_8)(n_8, n_{13}) \end{aligned}$$

between D-nodes are replaced by edges in the decision graph (when we write an edge as pair of nodes which is possible in graphs without multiple edges). In Fig. 3 the decision graph of the control flow graph of the function `find` is depicted.

To analyze the structure of the loops functions can be reduced to entry and exit nodes and nodes that represent loops. Fig. 4 shows the reduction of the control flow graph of the example `find`. This graph gives a simplified picture of the

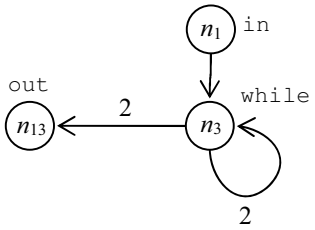


Fig. 4. The control flow graph of the function `find` reduced to entry, exit and loop nodes

control flow, which is in many cases sufficient for the analysis: The function starts with a while-loop. For each execution of the loop two possibilities arise (last name found but not the first name or neither last nor first name found). After the while-loop there are two different reasons for return (last and first name found or not found).

Each path $d \in D_{N'NN'}$ that starts and ends with a node in N' can be uniquely written as concatenation of $N'N'$-paths by splitting it at nodes in N': $d = d_1\, d_2 \ldots d_m$ with $m \geq 1$. The function δ can be straightforwardly extended to such paths by $\delta(d) = \delta(d_1)\, \delta(d_2) \ldots \delta(d_m)$.

For example, the path

$$d = (n_1, n_2)(n_2, n_3)(n_3, n_4)(n_4, n_5)(n_5, n_6)(n_6, n_9)\\ (n_9, n_{10})(n_{10}, n_3)$$

in the control flow graph of the function `find` can be split at the D-nodes n_3, n_4, n_6 into the DD-paths

$$d_1 = (n_1, n_2)(n_2, n_3)\\ d_2 = (n_3, n_4)\\ d_3 = (n_4, n_5)(n_5, n_6)\\ d_4 = (n_6, n_9)(n_9, n_{10})(n_{10}, n_3).$$

Furthermore, we have

$$\delta(d_1) = (n_1, n_3)\\ \delta(d_2) = (n_3, n_4)\\ \delta(d_3) = (n_4, n_6)\\ \delta(d_4) = (n_6, n_3)$$

and

$$\delta(d) = (n_1, n_3)(n_3, n_4)(n_4, n_6)(n_6, n_3).$$

From

$$\mathrm{end}(\delta(d_j)) = \mathrm{end}(d_j) = \mathrm{start}(d_{j+1}) = \mathrm{start}(\delta(d_{j+1}))$$

for $j = 1, \ldots, m-1$ it follows that $\delta(d)$ is a path in the reduced graph. Additionally, when we consider the sequence of nodes in d and project it to N' we get

$$\mathrm{start}(d_1)\, \mathrm{start}(d_2)\, \ldots\, \mathrm{start}(d_m)\, \mathrm{end}(d_m)$$

which is equal to the sequence of nodes in $\delta(d)$.

In the example, this sequence is

$$n_1\, n_3\, n_4\, n_6\, n_3.$$

The extended function $\delta : D_{N'NN'} \to D'$ from the set of paths in G that start and end with a node in N' to the set of paths in G' is a bijective function. This leads to the following theorem.

Theorem 1. Let $G = (N, E)$ be a directed graph, let $N' \subseteq N$ be a subset of its nodes and let $G' = (N', E')$ be the reduction of G to N' with the bijection $\delta : D_{N'\bar{N}'N'} \to E'$. The extension

$$\delta : D_{N'NN'} \to D'$$

of δ is also bijective. For all paths $d \in D_{N'NN'}$ it holds that

$$\mathrm{nodes}(d)\,|_{N'} = \mathrm{nodes}(\delta(d))$$

where $|_{N'}$ is the projection to N' and in particular

$$\mathrm{start}(d) = \mathrm{start}(\delta(d))\\ \mathrm{end}(d) = \mathrm{end}(\delta(d))$$

Proof. It remains to show that δ is bijective. Let d and $\overline{d}$ be two paths in G that start and end with a node in N' where $\delta(d) = \delta(\overline{d}) = e_1\, e_2 \ldots e_m$. This means that both d and $\overline{d}$ are the concatenation of m $N'N'$-paths $d_1\, d_2 \ldots d_m$, $\overline{d}_1\, \overline{d}_2 \ldots \overline{d}_m$, respectively. From $\delta(d_j) = e_j = \delta(\overline{d}_j)$ it follows that $d_j = \overline{d}_j$ (δ is bijective according to Definition 4) and $d = \overline{d}$. Let $e_1\, e_2 \ldots e_m$ be a path in G'. Then $d = \delta^{-1}(e_1)\, \delta^{-1}(e_2) \ldots \delta^{-1}(e_m)$ is a path in G that starts and ends with a node in N' with $\delta(d) = e_1\, e_2 \ldots e_m$. ■

This theorem shows that all paths in G that start and end in a node in N' are represented in the reduced graph. In a previous paper [14] we showed that there might be paths that start in a node in N' but do not end in N' and furthermore even can not be prolonged to a node in N'. For the reduction to D-nodes such paths have been called unconditional loops. One example for this situation is

```
if (x) label: goto label;
```

In the control flow graph there is a path starting in the D-node representing the if-statement leading to the goto-node which cannot be prolonged to a DD-path.

In a control flow graph an execution of the function that terminates with a return-statement or at the finishing brace induces a path that starts in the entry node n_{in} and ends in the exit node n_{out}. When the control flow graph is reduced to a subset of its nodes that includes n_{in} and n_{out}, the path and thus the execution is—according to Theorem 1—also represented in the reduced graph. For this reason, a reduction of a control flow graph should always include the entry node n_{in} and the exit node n_{out}.

So far we defined the

- the reduction to the entry and exit nodes,
- the reduction to the entry, exit and loop nodes and
- the reduction to D-nodes.

Another interesting reduction is

- the reduction to the interface nodes, that are entry nodes, exit nodes and nodes that represent function calls.

Fig. 5 shows the control flow graph of another example `calcSolution` and its reduction to the interface nodes. This function calculates the solution of a system of `N` linear equations with `N` variables using Cramer's rule. It gets as parameters the coefficient matrix `A`, its determinant and the vector of constants. The first if-statement checks the determinant of the coefficient matrix. If it is zero the function returns.

Open Science Index, Computer and Information Engineering Vol:8, No:3, 2014 publications.waset.org/9997636/pdf

Open Science Index, Computer and Information Engineering Vol:8, No:3, 2014 publications.waset.org/9997636/pdf

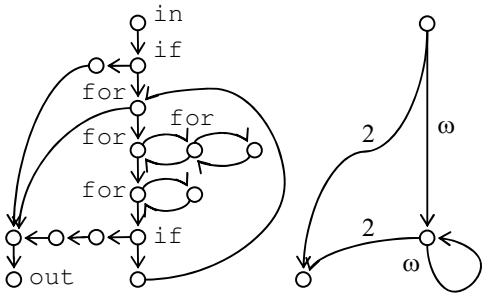


Fig. 5. Control flow graph of `calcSolution` and its reduction to the interface nodes

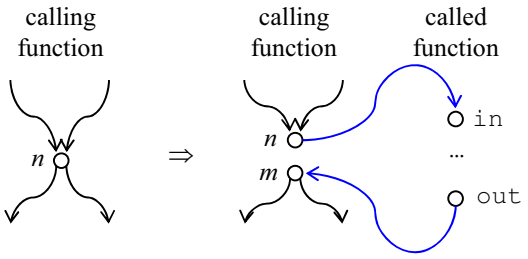


Fig. 6. Node splitting

The following for-statement loops through all columns of the matrix, copies the matrix to `B` and replaces the column by the vector of constants with three more for-loops. Then a function `calcDeterminant` is called with parameter `B` within an if-statement. If it returns with an error, the for-loop it terminated by a break-statement and the function returns. Otherwise, the determinant of `B` is divided by the determinant of `A`.

IV. Interprocedural Control Flow Graphs

So far, we examined only single functions and their control flow graphs. If we want to represent groups of functions or whole programs graphically, we need interprocedural control flow graphs. A well-known approach, e.g., proposed by Reps, Horwitz and Sagiv [29] or by Harrold and Rothermel [17], [18], is to split each node that represents a function call into two nodes—a call node and a return node. We add an edge from the call node to the entry node of control flow graph of the called function and from the corresponding exit node to the return node. All edges that start in the original node start then in the return node (Fig. 6).

Definition 6. Let F be a set of functions with control flow graphs (N_f, E_f) for $f \in F$ where we assume that the sets N_f and also the sets E_f are pairwise disjoint. The *interprocedural control flow graph* of F is defined as follows. In the graph $(\bigcup_{f\in F} N_f, \bigcup_{f\in F} E_f)$, we add for each node n that represents a function call (*call node*)

- a *return node* m,
- an edge (m, n') for each edge (n, n') that starts in n, the edges (n, n') are then deleted,
- an edge that starts in n and ends in the entry node of the control flow graph of the called function,
- an edge that starts in the exit node of the control flow graph of the called function and ends in m.

In interprocedural control flow graphs we label the arcs between a call node and the entry node of the control flow graph of the called function with the function name for better understanding.

The complete example program that calculates the solution of a system of linear equations mentioned above consists of seven functions and eight function calls, one of them a recursive call. The interprocedural control flow graph has 79 nodes. Fourteen of them are entry or exit nodes, 16 are call or return nodes. The reduction of this graph to its interface nodes, that are, entry, exit, call and return nodes, has only 30 nodes and is shown in Fig. 7 where edges between call nodes and entry nodes and edges between exit nodes and return nodes are drawn in blue color.

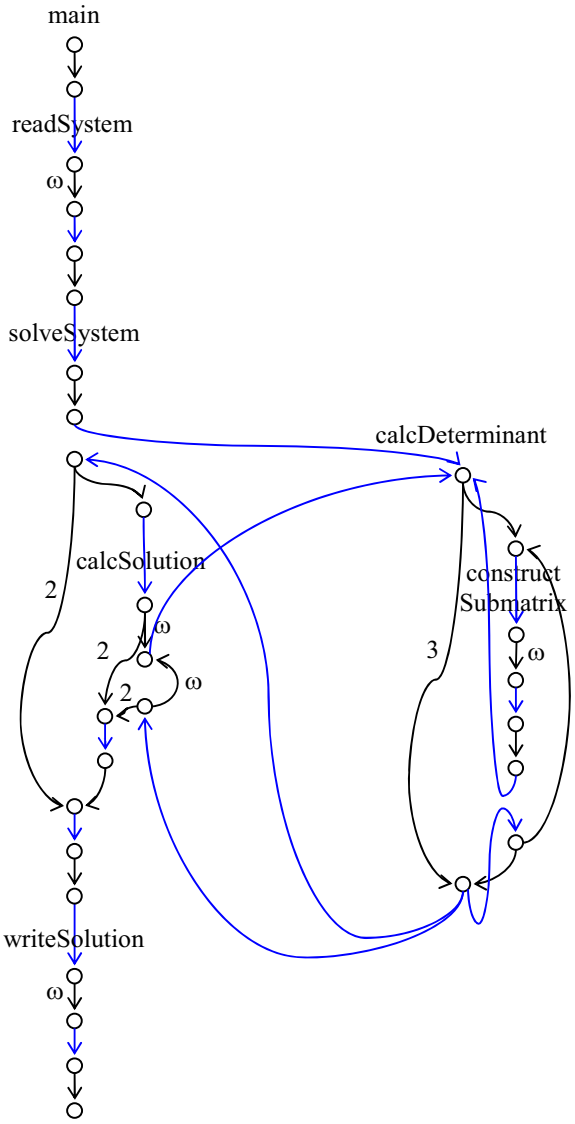


Fig. 7. Reduction of the example program to the interface nodes.

From a practical point of view, it would be helpful if a software engineer could apply reductions step by step to decrease the size of the graph and to see different levels of details. For example, the reduction to the interface nodes could

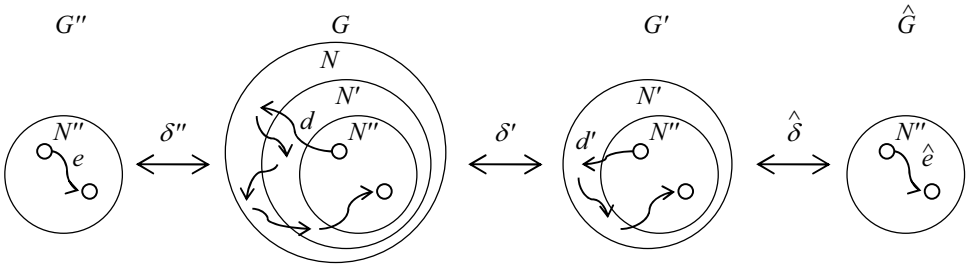


Fig. 8. Construction of the bijective function δ

be followed by eliminating the inner nodes of the function `calcDeterminant` in a second reduction step. Of course, the result should be the same if we do both reductions in one step.

Theorem 2. Let $G = (N, E)$ be a directed graph and let $N'' \subseteq N' \subseteq N$ be two subsets of its nodes. Then it holds that

$$\mathrm{red}(G, N'') = \mathrm{red}(\mathrm{red}(G, N'), N'')$$

where red is the reduction of a graph to a subset of its nodes.

Proof. Let us introduce the following notation:

- $G'' = (N'', E'')$ is the reduction of G to N'' with the bijection $\delta'' : D_{N''\bar{N}''N''} \to E''$,
- $G' = (N', E')$ is the reduction of G to N' with the bijection $\delta' : D_{N'\bar{N}'N'} \to E'$,
- $\widehat{G} = (N'', \widehat{E})$ is the reduction of G' to N'' with the bijection $\widehat{\delta} : D'_{N''\bar{N}''N''} \to \widehat{E}$.

We have to show that $G'' = \widehat{G}$. Both graphs G'' and $\widehat{G}$ have the same set N'' of nodes. Now we construct a bijection $\delta : E'' \to \widehat{E}$ that preserves the start and the end nodes.

Let $e \in E''$ be an edge in the reduced graph G'' (Fig. 8). Then $d = \delta''^{-1}(e) \in D_{N''\bar{N}''N''}$ is a path in G.

As shown in Theorem 1, we can extend δ' from $D_{N'\bar{N}'N'} \to E'$ to $D_{N'NN'} \to D'$. This extension can be restricted to $D_{N''\bar{N}''N''}$. A path $d \in D_{N''\bar{N}''N''}$ is then mapped to a path $d' = \delta(d) \in D'$. From Theorem 1 we know that $\mathrm{nodes}(d) \mid_{N'} = \mathrm{nodes}(d')$ and $d' \in D'_{N''\bar{N}''N''}$. This function $\delta' : D_{N''\bar{N}''N''} \to D'_{N''\bar{N}''N''}$ is injective since $\delta' : D_{N'NN'} \to D'$ is injective. Let d' be a path in $D'_{N''\bar{N}''N''}$. We know that it exists $d \in D_{N'NN'}$ with $d' = \delta'(d)$. From $\mathrm{nodes}(d) \mid_{N'} = \mathrm{nodes}(d')$ follows that d is in $D_{N''\bar{N}''N''}$. Together this means that $\delta' : D_{N''\bar{N}''N''} \to D'_{N''\bar{N}''N''}$ is also bijective.

If we apply $\widehat{\delta}$, we get an edge $\widehat{e} = \widehat{\delta}(d')$ in $\widehat{G}$. Thus we defined a bijective function $\delta = \delta''^{-1} \circ \delta' \circ \widehat{\delta} : E'' \to \widehat{E}$. ■

In Fig. 9 the reduction to the interface nodes is followed by a second reduction. This time we eliminate—with the exception of entry nodes and exit nodes—all nodes with only one incoming and one outgoing arc (regardless of the multiplicity of edges) and get a more compact graph. The entry and exit nodes are kept to identify the entry and exit points of the functions. The reduced graph consists of only 19 nodes. This graph is also sufficient to understand the control flow in the program. The function `main` calls `readSystem`. In this function infinitely many paths lead from the entry to the exit because the size of the system of linear equations is not limited. Then `solveSystem` is called. Firstly, it calls `calcDeterminant`. After the return `solveSystem` might return with one of two possible causes without calculating a solution. Otherwise, it calls `calcSolution`. For this function there are two causes of exceptions that lead to its return immediately after its call. If no exception occurs, it calls in a loop `calcDeterminant`. The reason for infinitely many paths on the loop is the same as above. At last `main` calls `writeSolution`. In the function `calcDeterminant` three different exceptions may cause it to return after being called. It calls `constructSubmatrix` and itself recursively in a loop.

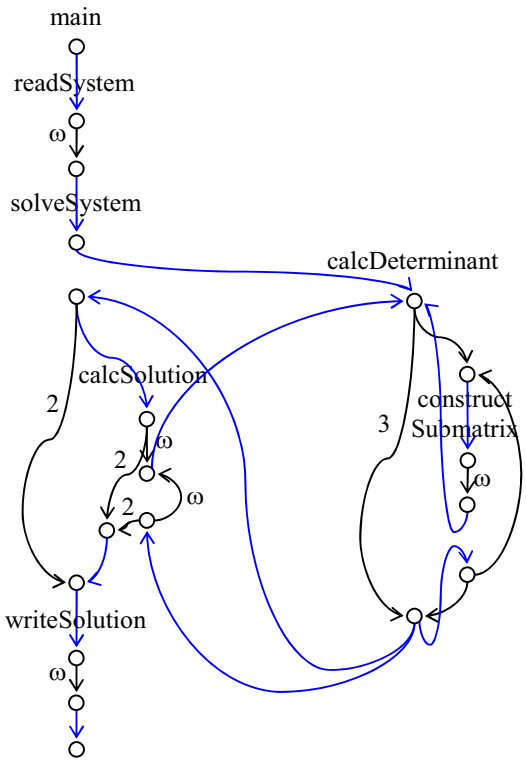


Fig. 9. Second reduction of the example program

Often library functions should not be analyzed and therefore should not be shown in the control flow graph. To model this, Definition 6 can be changed such that not all call nodes have to be split but only those that the user chooses. The control flow graphs of the corresponding called functions are excluded from

Open Science Index, Computer and Information Engineering Vol:8, No:3, 2014 publications.waset.org/9997636/pdf

Open Science Index, Computer and Information Engineering Vol:8, No:3, 2014 publications.waset.org/9997636/pdf

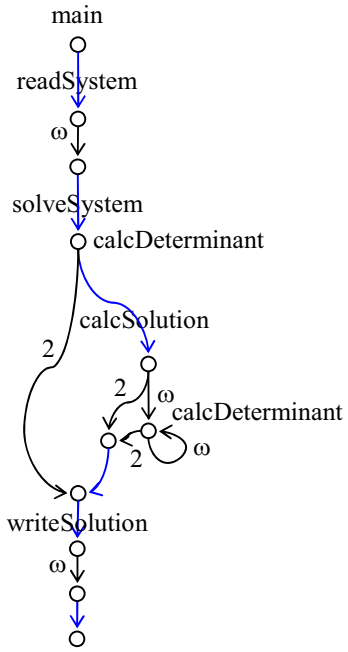


Fig. 10. Third reduction of the example program without library functions

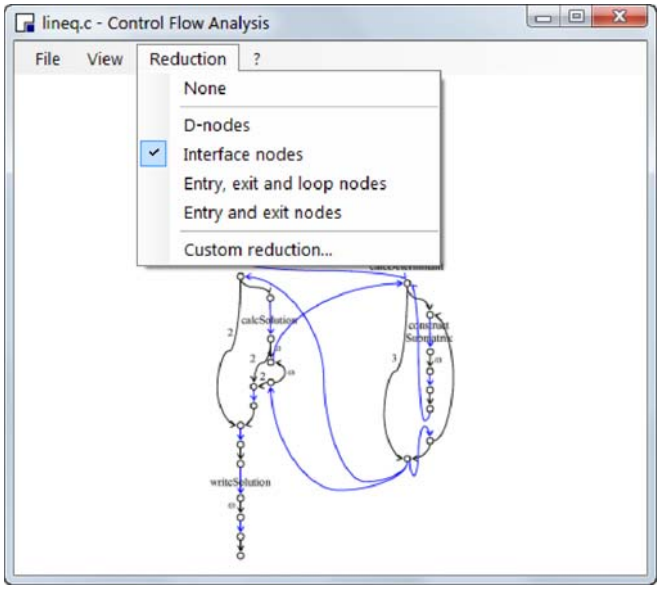


Fig. 11. Prototype of a control flow analysis tool

the interprocedural control flow graph. In the example, we exclude the library function `calcDeterminant` and get the reduced control flow graph as shown in Fig. 10. The call nodes of `calcDeterminant` are considered as interface nodes in this example.

V. Conclusion

In this paper we studied reductions of directed graphs to subsets of nodes. Paths between these nodes are replaced by edges. The reductions can be applied to control flow graphs of single functions or to interprocedural control flow graphs of sets of functions but are not restricted to control flow graphs. Executions are preserved and reductions can be done stepwise. This can help software engineers when they analyze their programs and the control flow structure.

There are, besides the applications mentioned in the introduction, also other fields where control flow graphs and reductions can be used, for example, the detection of malware [7], the analysis of the control flow to find deviations caused by attacks on the program [1], [33] or the extraction of the control flow structure from executables [32], [34]. And it seems promising to apply graph reductions to other domains, e.g., VHDL [36], workflow modeling [9], [30] or the testing of graphical user interfaces [6], [25], [26], and exploit their advantages.

In future work a prototype tool could be implemented to find a set of reductions that are most helpful to software engineers in practical applications (Fig. 11).

Acknowledgment

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

References

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

[2] A.V. Aho, M.S. Lam, R. Sethi and J.D. Ullman. 2007. *Compilers: Principles, techniques, and tools*. Amsterdam: Addison-Wesley Longman, 2. ed.

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

[4] A. Bertolino and M. Marré. 1994. 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.

[5] J.A. Bondy and U.S.R. Murty. 2008. *Graph theory*. London: Springer.

[6] P.A. Brooks and A.M. Memon. 2007. Automated GUI testing guided by usage profiles. *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering* (ASE '07), Atlanta, USA, 05. – 09.11.2007, pp. 333–342.

[7] D. Bruschi, L. Martignoni and M. Monga. 2006. Detecting self-mutating malware using control-flow graph matching. In: R. Büschkes and P. Laskov (eds.), *Proceedings of the Third International Conference on Detection of Intrusions and Malware & Vulnerability Assessment* (DIMVA '06), Berlin, Germany, 13. – 14.07.2006, Lecture Notes in Computer Science 4064, Berlin, Heidelberg: Springer, pp. 129–143.

[8] R. Diestel. 2010. *Graph theory*. New York, Berlin, Heidelberg: Springer, 4. ed.

[9] D. Draheim. 2010. *Business process technology: A unified view on business processes, workflows and enterprise applications*. Berlin, Heidelberg: Springer.

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

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

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

Open Science Index, Computer and Information Engineering Vol:8, No:3, 2014 publications.waset.org/9997636/pdf

[13] R. Gold. 2012. On cyclomatic complexity and decision graphs. *Proceedings of the 10th International Conference of Numerical Analysis and Applied Mathematics* (ICNAAM '12), Kos, Greece, 19. – 25.09.2012, AIP Conf. Proc. 1479, pp. 2170–2173.

[14] R. Gold. 2013. Decision graphs and their application to software testing. *ISRN Software Engineering*, vol. 2013.

[15] R. Gold. 2013. The decision structure of programs. *Proceedings of the International Research Conference on Information Technology and Computer Sciences*, IRCITCS 2013, Kuala Lumpur, Malaysia, 28. - 30.09.2013.

[16] N. Gupta, A.P. Mathur and M.L. Soffa. 2000. Generating test data for branch coverage. *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering*, Grenoble, France, 11. – 15.09.2000, pp. 219–227.

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

[18] M.J. Harrold and G. Rothermel. 1996. A coherent family of analyzable graphical representations for object-oriented software. *Technical Report OSU-CISRC-11/96-TR60*, Department of Computer and Information Science, The Ohio State University.

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

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

[21] P. Jalote. 2005. *An integrated approach to software engineering*. New York: Springer, 3. ed.

[22] G.M. Kapfhammer. 2004. Software testing. In: A.B. Tucker (ed.), *Computer Science Handbook*, Chapman & Hall/CRC, 2. ed.

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

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

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

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

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

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

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

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

[31] I. Sommerville. 2004. *Software engineering*. Boston: Pearson Education Limited, 7. ed.

[32] H. Theiling. 2000. Extracting safe and precise control flow from binaries. *Proceedings of the Seventh International Conference on Real-Time Computing Systems and Applications*, Cheju Island, South Korea, 12. – 14.12.2000, pp. 23–30.

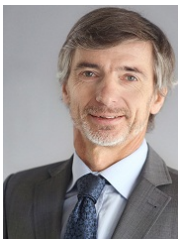
[33] Z. Wang and X. Jiang. 2010. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. *Proceedings of the IEEE Symposium on Security and Privacy* (SP), Oakland, USA, 16. – 19.05.2010.

[34] Y. Wenjian, J. Liehui, Y. Qing, Z. Lina and L. Jizhong. 2009. A control flow graph reconstruction method from binaries based on XML. *Proceedings of the International Forum on Computer Science-Technology and Applications*, IFCSTA '09, Chongqing, 25. – 27.12.2009, pp. 226–229.

[35] L.J. White. 1981. Basic mathematical definitions and results in testing. In: B. Chandrasekaran and S. Radicchi (eds.), *Computer Program Testing*, New York: North-Holland.

[36] Q. Zhang and I.G. Harris. 2000. A data flow fault coverage metric for validation of behavioral HDL descriptions. *International Conference on Computer-Aided Design* (ICCAD '00), San Jose, USA, 05.11.2000.

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



Robert Gold received his Diploma and Ph.D. degrees in computer science from TUM (Technische Universität München), Munich, Germany. Since 1998 his is professor for software engineering and programming languages at THI (Technische Hochschule Ingolstadt). His work interest include automotive software engineering, software test and usability. Recent research was focused on control flow graphs and software testing.