

Algorithms for Optimally Tracing Time Critical Programs

Las Vegas Nevada, USA, June 26-29, 2006

Dr. Sergej Alekseev
Siemens AG, Berlin, Germany
Communications Mobile Networks
Charging and Care
Berlin, 13629-Germany

Keywords: *Graphalgorithm, Tracing, Time Critical Object Oriented Software*

Abstract—The current work is the closing article of an ongoing series of papers that illustrate a high-performance solution to the tracing of time critical applications with minimal impact on the running system. The idea of the proposed solution is to instrument the application at significant points with event functions, log the events of these functions during execution and reconstruct the complete control flow on the basis of protocolled events. The finding of the minimum set of the monitoring points in the control flow graph, which is sufficient to reconstruct all possible control flows, is a \mathcal{NP} -complete problem (Feedback-Vertex-Set (FVS) [Gar79]).

In this paper we present the graph-theoretical algorithms for the selection of the monitoring points in the control flow graphs, which are based on decomposition of graphs into connected components. We present the solution, which combines the graph-theoretical algorithms with the Knuth-Stevenson Algorithm and provides in most cases the minimal set of monitoring points. But generally this problem is still \mathcal{NP} -complete.

This paper deals with results from a joint research project between the Siemens AG, COM MN CC ST, Berlin and the University of Oldenburg.

I. INTRODUCTION

Tracing large server applications is a challenging task as execution of these applications is multithreaded and time-critical. Normally large systems should be available around-the-clock and can not be stopped for a long time. The software updates and services are executed in the running system. Examples of such systems are database-applications, telecommunication servers, bank systems etc. If the system behaves abnormally (some applications are stopped or crashed), then it is necessary to localize and solve the problem as fast as possible. The abnormal behaviour may be caused by software-bugs, wrong administration of the application, hardware defects, a power blackout, network problems etc.

Last year we presented a technique for the monitoring and reconstruction of the control flows in Java-based applications [Ale05b]. This technique allows to reduce the number of events because the event functions will be optimally placed. Moreover this technique has only a low influence on performance of the monitored applications due to asynchronous

logging [Ale05a]. The result of this project is a working black box writer (like a black box in an aircraft) for application processes, which logs the events of the last 10-20 minutes of the execution. If the system has crashed, then it is possible to reconstruct all executed control flows of the application from the logged events.

In this article the graph-theoretical algorithms for reduction of the monitoring points are summarized and some new approaches will be presented, which allow to find in most cases the minimal set of the monitoring points in the control flow graph. The finding of the minimum set of the monitoring points in the control flow graph is the \mathcal{NP} -complete problem. Our approach doesn't solve the problem, but the graph-theoretical extension via decomposition into connected components provides the algorithms, which deliver the same or better results than the known Knuth-Stevenson Algorithm¹.

II. GRAPH BASICS

The analysis of control flow graphs is based on graph-theoretical decompositions of graphs into connected components. Directed graphs are decomposed in a natural way into weakly connected components. A *weakly connected component* is the subgraph generated from a maximal class of vertices connected by a-paths. An a-path is a path where an arc is passed through in forward or in backward direction. A *strongly connected component* is the subgraph generated from a maximal class of vertices mutually connected by f-paths, i.e. paths where arcs are passed through only in forward direction. Every strongly connected component is part of a weakly connected component. Every weakly connected component is decomposed in a natural way into strongly connected components and arcs which do not belong to any strongly connected component. The subgraph generated by these arcs is the *external dag* of the weakly connected component. It is a directed f-acyclic graph (dag). The vertices common to a strongly connected component and the external dag are called *weak attachment points*. The algorithms for finding

¹The proofs of the algorithms are not included in this article and can be found in [Ale04a]

the weakly connected and the strongly connected components as well as the external dag with *weak attachment points* are described in [Sti04], [Sti01b]. A second graph decomposition, namely the *standard a-decomposition*, is also used in the analysis of control flow graphs. It uses a-paths only, as if the Graph were undirected. The graph is decomposed into maximal 2-edge-connected subgraphs, the *subcomponents*, and subgraphs which are a-trees. The subcomponents in turn are subdivided into maximal 2-connected subgraphs, the *biblocks*. Several biblocks may have a vertex in common. Such vertices are *hinge points*. For details of the standard a-decomposition including algorithms see [Sti98].

III. FORMALIZATION OF MINIMAL CRITERIA

In mathematics we speak about a minimum concerning a characteristic A , if something is smallest possible value with respect to A . For the control flow monitoring a subset of nodes of a control flow graph is used, which is sufficient for the reconstruction of all possible control flows. Then there should be a subset of nodes, with minimum regarding the number of elements, which are needed to make the reconstruction of control flows possible. From these considerations the following definition can be derived.

Definition 3.1: The set of the monitoring vertices X of the control flow graph G is called minimal X_{min} , if there is no other set X' with $|X'| < |X_{min}|$, which is sufficient for reconstruction of all possible control flows.

The Definition 3.1 applies with reservation of the following note.

Remark 3.1: Each control flow in the control flow graph G is identified by a start and end vertex, Sometimes the start and end events can be ignored.

In practice the reconstruction program can be adapted for analysing the start and end events.

IV. SETEVENTS AND RECONSTRUCTFLOW ALGORITHMS

The placing of monitoring events in the program and the reconstruction of the control flow inside a method is realized by the two graph theoretical algorithms *SETEVENTS* and *RECONSTRUCTFLOW*. The algorithm *SETEVENTS* places events (i.e functions which trigger an event). The second algorithm *RECONSTRUCTFLOW* takes these events and reconstructs the control flow of the recorded sequence. The first version of these algorithms was published in the technical report [Ale04a] and presented at the science conference in Ilmenau [Ale04b]. The adapted version of these algorithms have been presented at *The 2005 International Conference on Software Engineering Research and Practice* [Ale05b]. Due to completeness of the algorithms the short description of these algorithms will be done here.

A. Algorithm for placing of events

The algorithm *SETEVENTS* (figure 1) uses as input a flow graph G . The flow graph G is a quadruple (V, A, S, E) , where V is a nonempty set, the set of *vertices* (or nodes). A is

SETEVENTS (G)

```

1   for (all start and end vertices  $v$  of  $G$ )
2     mark  $v$  as event vertex;
3   find weakly connected components of  $G$ ;
4   for (all weakly connected components  $W$ ) {
5     find strongly connected components
      and DAG  $D$  of  $W$ ;
6     for (all strongly connected components  $S$ ) {
7       SETSCOMPEVENTS( $S$ );
8     }
9     SETDAGEVENTS( $D$ );
10  }
```

Fig. 1. SETEVENTS Algorithm for placing of events

a (possibly empty) set of *arcs*. A is the finite set of sets (a, b) , where $a, b \in V$. $S \subset V$ is a nonempty set, the set of *start-vertices*. $E \subset V$ is a nonempty set, the set of *end-vertices*. After execution of the algorithm *SETEVENTS* the event vertices in the flow graph G are marked. So the result of the algorithm *SETEVENTS* is a nonempty set $X \subseteq V$. The set X is the set of all vertices, which have been marked as event vertices.

The algorithm *SETEVENTS* is executed in the following steps:

- 1) At first (line 1 and 2) all *start* and *end* vertices will be marked as *event vertices*.
- 2) In line 3 the flow graph G will be decomposed into weakly connected components. For finding the weakly connected components a simple depth first search algorithm can be used. One variant of this algorithm is described in [Sti01b].
- 3) Each weakly connected component will be decomposed into strongly connected components and the external DAG (line 5). For finding the strongly connected components, external DAG and weak attachment points, the algorithm from [Sti01b] is used.
- 4) For the placing of event vertices in strongly connected components (line 7) the procedure *SETSCOMPEVENTS* is called (figure 2).
- 5) For the placing of event vertices in the external DAG (line 9) the procedure *SETDAGEVENTS* is called (figure 3).

The input for the procedure *SETEVENTSSCOMP* is a subgraph S generated from each strongly connected component. The procedure *SETEVENTSSCOMP* starts a special depth first search (lines 1-5). For that the recursive subprocedure *SETSCOMPEVENTSPR* is used, that checks, if any outgoing arcs of the currently processed vertex v lead to a vertex, which lies on several f -cycles (*SETSCOMPEVENTSPR*, line 6). If this condition is affirmative, then the vertex v will be marked as event vertex.

In the procedure *SETSCOMPEVENTSPR* two help functions are used: *otherend* and *indegree*. The function *otherend* returns the neighbor vertex of the given arc and the function *indegree*

```

SETSCOMPEVENTS ( $S$ )
1  for (all vertices  $v$  of  $S$ ) {
2      if ( $v$  unmarked) {
3          SETSCOMPEVENTSPR( $v$ );
4      }
5  }
6  mark all weak attachment vertices of  $S$ 
   as event vertices;

```

```

SETSCOMPEVENTSPR ( $v$ )
1  if( $v$  processed)
2      return;
3  mark  $v$  as processed;
4  for (all outgoing arcs  $l$  of  $v$ ) {
5      SETSCOMPEVENTSPR ( $otherend(l, v)$ );
6      if ( $indegree(otherend(l, v)) > 1$ ) {
7          mark  $v$  as event vertex;
8      }
9  }

```

Fig. 2. Procedure *SETSCOMPEVENTS*

returns the number of incoming arcs of the given vertex.

The input of the procedure *SETDAGEVENTS* (figure 3) is a subgraph D generated from the *external DAG*. The graph D is considered as an undirected graph and

```

SETDAGEVENTS ( $D$ )
1  find biblocks of  $D$ 
2  for (all biblocks  $B$ ) {
3      for (all vertices  $v$  of  $B$  with  $indegree(v) = 0$ ) {
4          SETBIBLOCKEVENTS( $v$ );
5      }
6  }

```

```

SETBIBLOCKEVENTS ( $v$ )
1  if( $v$  processed  $\vee$   $outdegree(v) = 0$ )
2      return;
3  mark  $v$  as processed;
4  for (all outgoing arcs  $l$  of  $v$ ) {
5      SETBIBLOCKEVENTS ( $otherend(l, v)$ );
6      if ( $indegree(otherend(l, v)) > 1$ ) {
7          mark  $v$  as event vertex;
8      }
9  }

```

Fig. 3. Procedure *SETDAGEVENTS*

decomposed into peripheral trees and the stopfree kernel. The stopfree kernel is decomposed into subcomponents and internal trees. Subcomponents are decomposed into biblocks (line 1). The detailed description of the connectivity structure of undirected graphs can be found in [Sti96], [Sti98] and [Sti01a]. The algorithm for finding the connectivity

structure was published in the technical report [Sti97]. The modified version of this algorithm is presented in [Sti04]. All biblocks from this structure are processed as directed subgraphs (lines 2-6). The event vertices will be marked only in biblocks, because the flow in internal and peripheral trees can be reconstructed without additional events. For the placing of events in the biblocks the subprocedure *SETBIBLOCKEVENTS* is used (line 4). The procedure is started on the vertex, which doesn't have any incoming arcs (line 3). The subprocedure *SETBIBLOCKEVENTS* works in the same way as subprocedure *SETSCOMPEVENTSPR* for the strongly connected components. It will be checked, if any outgoing arcs of the currently processed vertex v leads to a vertex, which lies on several f -paths (*SETBIBLOCKEVENTS*, line 6). If this condition is affirmative, then the vertex v will be marked as event vertex. *The algorithm SETEVENTS finds in the given flow graph a set of event vertices, which is sufficient and in many cases minimal for reconstructing of all control flows.*

Accuracy of this statement was proven in the [Ale04a].

B. Algorithm for reconstruction of a flow sequence

For the reconstruction of the sequence flow the auxiliary arc structure will be generated via modified DFS Algorithm (procedure MARCARCS). MARCARCS starts in each event vertex and assigns to the reachable arcs the event label of this vertex. The recursion of the MARCARCS procedure is finished if the next event or end vertex is reached. The arc structure should be generated only one time and can be used for reconstruction of all protocol led flows in this control flow graph.

The algorithm *RECONSTRUCTFLOW* (figure 6) uses as input the control flow graph with marked arcs G' (the result of the MARCARCS procedure), and a stack of events Y , both has been saved during monitoring. In figure 4 an example

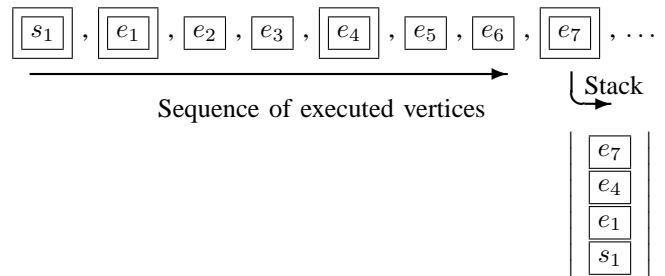


Fig. 4. Stack

of the execution sequence is demonstrated. The events of the execution sequence are mapped to the vertices of the control flow graph. The events in the double frame are mapped to the event vertices, which were marked by algorithm *SETEVENTS*. They will be logged and put into the stack Y . The result of the algorithm *RECONSTRUCTFLOW* is the control flow sequence (f -path) R reconstructed as queue structure. The

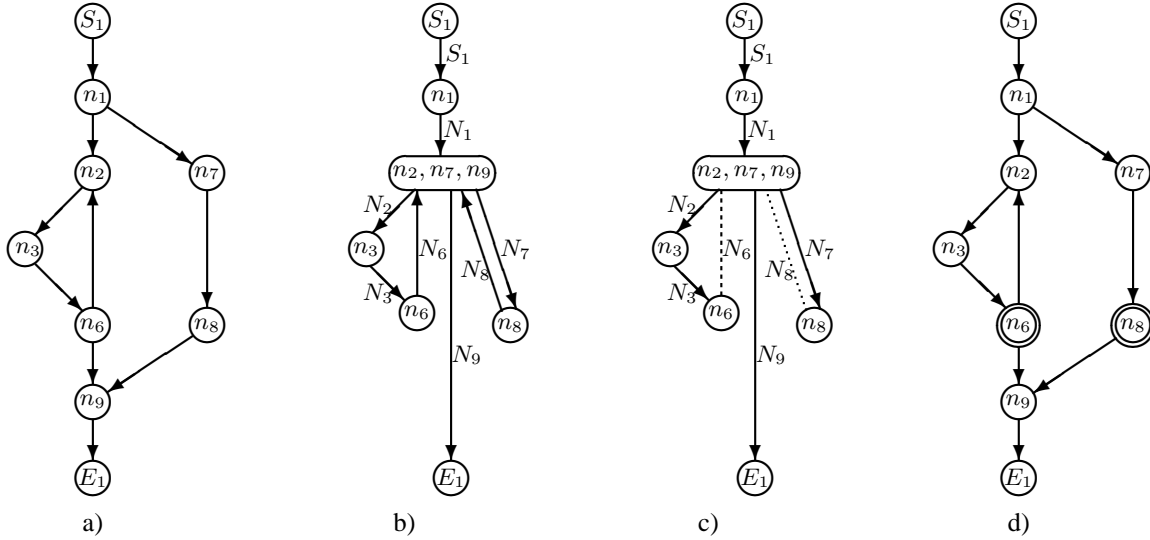


Fig. 7. Execution of the Knuth-Stevenson algorithm

and E edges, and there exists a vertex c and arcs $c \rightarrow a$ and $c \rightarrow b$, then vertex a is considered to be in the same equivalence class as vertex b ($a \equiv b$). The resultant transformed graph consists of nodes that correspond to the equivalence classes and edges that correspond to the nodes in the original graph (figure 7b). After applying the spanning tree algorithm the missing arcs can be mapped to the nodes in the original graph (figure 7c and 7d). The algorithm combined with the Knuth-Stevenson algorithm *SETEVENTS'* is presented in figure 8. The first steps of the algorithm *SETEVENTS'* are similar

strongly connected components are not marked by a modified depth search (Procedure *SETSCOMPEVENTS*), but by the Knuth-Stevenson-Algorithm (line 7).

Afterwards the weakly connected component W will be transformed into the weakly connected component W' as follows. From the weakly connected component W' all outgoing arcs (Arc set A_{out}) of the already found event vertices are deleted. Only the DAG remains, because the Knuth Stevenson algorithm interrupted all f -circles in the strongly connected components by finding the event vertices.

SETEVENTS' (G)

- 1 for (all start and end vertices v of G)
- 2 mark v as event vertex;
- 3 find weakly connected components of G ;
- 4 for (all weakly connected components W) {
- 5 find strongly connected components of W ;
- 6 for (all strongly connected components S) {
- 7 apply Knuth-Stevenson Algorithm to S ;
- 8 }
- 9 $W' = \text{BREAKFLOW}(W, A_{out})$;
- 10 find Biblock of W' ;
- 11 for (all Biblocks B of W') {
- 12 apply Knuth-Stevenson Algorithm to B ;
- 13 }
- 14 }

Fig. 8. Algorithm *SETEVENTS'*

to the steps of the algorithm *SETEVENTS*. The starting and end nodes of the flow graph G are marked as event nodes (lines 1-2). The graph G is decomposed into weakly connected components (line 3) and in each weakly connected component the strongly connected components are found (line 6). In contrast to the algorithm *SETEVENTS* the event vertices in

BREAKFLOW (W, A_{out})

- 1 $W' = W \setminus A_{out}$;
- 2 mark all reachable vertices from start vertices in W' ;
- 3 delete recursive unmarked vertices v from W'
(Stop if $v =$ entry or exit attachment vertex)
- 4 return W'

Fig. 9. Procedure *BREAKFLOW*

At the end the event vertices in the biblocks of the DAG (lines 10-12) are marked by the Knuth Stevenson algorithm. Thereby some vertices in the strongly connected components can be marked as event vertices.

VI. DECOMPOSITION OF STRONG CONNECTED COMPONENTS

The idea is to divide the strongly connected component recursively like a bowl. In the first step the outside f -circle of the strongly connected component is broken and the result is considered as an independent graph. This graph can have further strongly connected components and external DAG. For each strong connected component this step is repeated, until in the result graph no more further strongly connected components occur. Then the event vertices can be verified by the Knuth Stevenson algorithm.

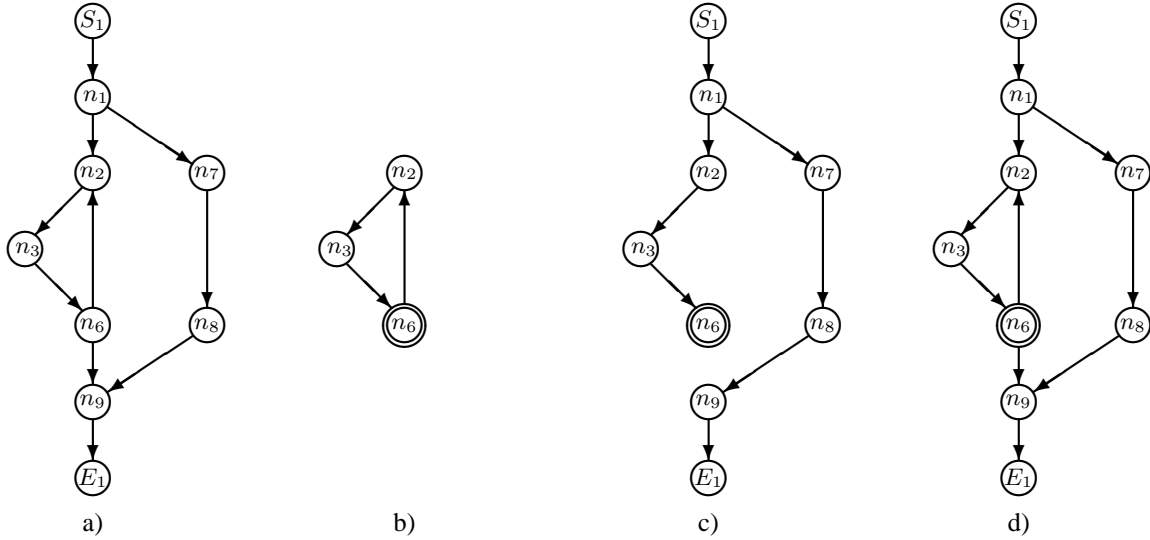


Fig. 10. Execution of the *SETEVENTS'* algorithm

In figure 11 the decomposition method of the strongly connected component is formalized in form of the algorithm *DECSCOMP*. The Input of the algorithm is a strong connected

connected component S' is executed. Afterwards the Knuth Stevenson algorithm is applied for finding the event vertices in the biblocks.

DECSCOMP (S)

- 1 break outer f-circle of S and generate graph G' ;
- 2 decompose G' in strongly connected components and external DAG ;
- 3 for(all strongly connected components s of G') {
- 4 *DECSCOMP*(s);
- 5 }
- 6 $S' = \text{BREAKFLOW}(S, A_{out})$;
- 7 find Biblock of S' ;
- 8 for (all Biblocks b of S') {
- 9 apply Knuth-Stevenson Algorithm to b ;
- 10 }

Fig. 11. Decomposition of strong connected components

component S . The outside loop of this connection component is broken. For that all incoming arcs of the entrance attachment vertex are redirected to a new vertex (*attachment vertex'*) (line 1). From the modified connected component S a graph G' is generated. The graph G' is composed in the next step into strongly connected components and the external DAG . For that the algorithm *STRONGCOMP* from [Sti04] can be used. For all found strongly connected components in the loop (lines 3-5) the algorithm *DECSCOMP* is recursively called. The recursion ends, if the graph G' does not contain further strongly connected components.

The next step (line 6) is the transformation of the strongly connected component S into a connection component S' by the procedure *BREAKFLOW*. From the strong connected component S all outgoing arcs (Arc set A_{out}) of the already found event vertices are deleted.

VII. HIERARCHY OF ALGORITHMS

In reference to the number of monitoring vertices, which are found by the algorithms, the algorithms can be set up as follows in a hierarchy (figure 12). With the expression $A_1 \geq A_2$ we define, that the number of monitoring vertices, which is verified by the algorithm A_1 , is greater or equal to the number of monitoring vertices verified by algorithm A_2 . Figure 13 presents the algorithm hierarchy in reference to complexity. The hierarchy described above is explained by an example (figure 14). Figure 14 shows the control flow graph G with one strongly connected component $\{n_1, n_2, n_3, n_5, n_6, n_7, n_8, n_{10}\}$.

The result of the algorithm *SETEVENTS* for the control flow graph G contains four event vertices (figure 14a). Two event vertices n_6 and n_7 are verified inside of the strongly connected component and two event vertices are weak attachment vertices n_1 and n_{10} . In the external DAG no event vertices are verified. The result of the *Knuth-Stevenson-Algorithm* is shown in figure 14b and contains four event vertices as well. The algorithm *SETEVENTS'* will verify three event vertices (figure 14c) and the algorithm *SETEVENTS''* only two event vertices (figure 14d).

In the choice of an algorithm attention must be paid to the fact, that the additional effort during the preparation remunerates as profit of tracing during execution of the application. The choice depends on which application has to be supervised. If the preparation effort does not have any restrictions, then it is suggested to use the algorithm *SETEVENTS''*. The algorithm with linear complexity (*Knuth-Stevenson* or *SETEVENTS'*) should be applied if the

$SETEVENTS \geq Knuth - Stevenson \geq SETEVENTS' \geq SETEVENTS'' \geq Unknown$

Fig. 12. Algorithms hierarchy

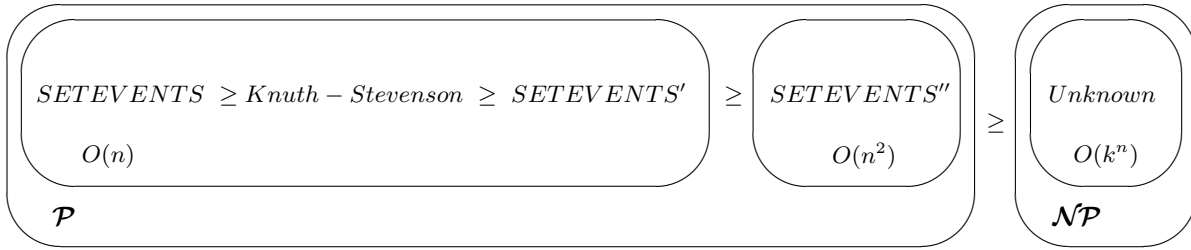


Fig. 13. complexity hierarchy of the algorithms

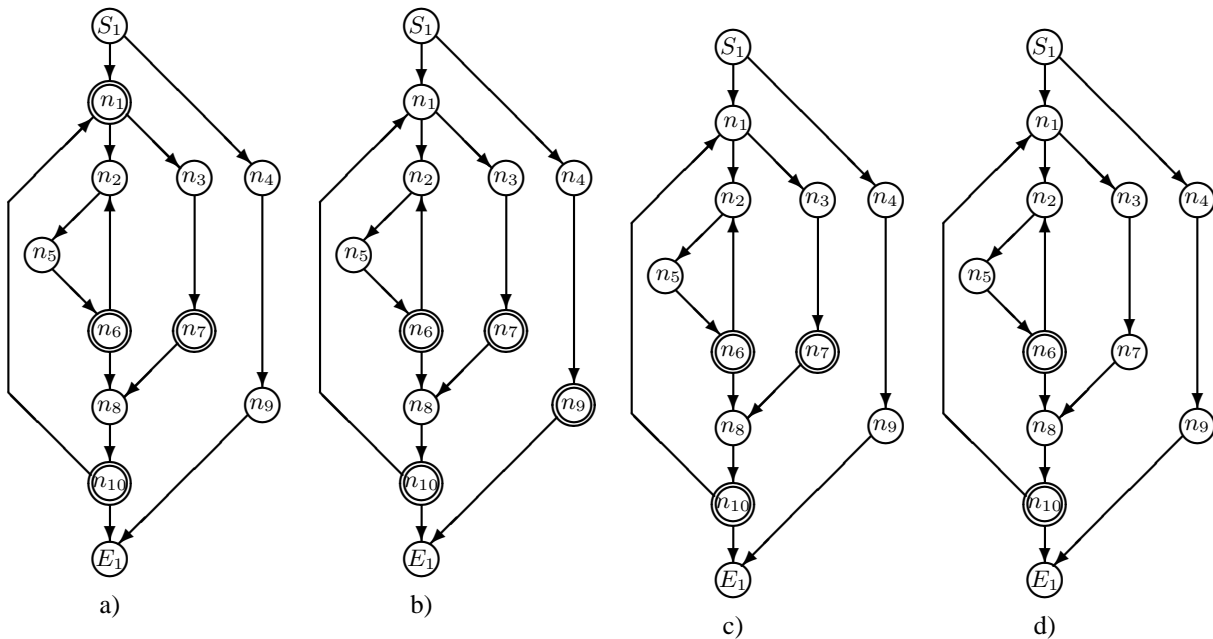


Fig. 14. a) SETEVENTS b) Knuth-Stevenson c) SETEVENTS' d) SETEVENTS''

preparation for monitoring must take place at run time.

REFERENCES

[Ale04a] Alekseev, Sergej. Dienste Intelligenter Netze – Graphentheoretische Methoden in der Kontrollflussanalyse. Technical report, Universität Oldenburg, 2004.

[Ale04b] Alekseev, Sergej; Stiege, Günther. Applying Graph Theory to The Components of An Intelligent Network. In *Proc. 49. Internationales Wissenschaftliches Kolloquium Ilmenau, 27.-30.09.2004*, volume 2, pages 319–324. Shaker Verlag, 2004.

[Ale05a] Alekseev, Sergej; Johannes Wust. Graph Theoretical Methods in the Control Flow Analysis of Object Oriented Real Time Software. Technical report, Universität Oldenburg, 2005.

[Ale05b] Alekseev, Sergej; Stiege, Günther. Graph Theory in the Control Flow Analysis of the large time critical Applications. In *The 2005 International Conference on Software Engineering Research and Practice Las Vegas, Nevada, USA, June 27-29, 2005*, volume 1, pages 253–259. CSREA Press, 2005.

[Gar79] Johnson D. S. Garey, M. R. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[Sti96] Stiege, Günther. Connectivity and Periodicity in Undirected Graphs. Technical report, Universität Hildesheim, 1996.

[Sti97] Stiege, Günther. An Algorithm for Finding the Connectivity Structure of Undirected Graphs. Technical report, Universität Hildesheim, 1997.

[Sti98] Stiege, Günther. Edge Partitions in Undirected Graphs. Technical report, Universität Oldenburg, 1998.

[Sti01a] Stiege, Günther. Higher Decompositions in Undirected Graphs. Technical report, Universität Oldenburg, 2001.

[Sti01b] Stiege, Günther. Standard Decomposition and Periodicity of Digraphs. Berichte aus dem Fachbereich Informatik 5/01, Universität Oldenburg, 2001.

[Sti04] Stiege, Günther. *Graphen und Graphalgorithmen*. Universität Oldenburg, 2004. Available by <http://www-bvs.informatik.uni-oldenburg.de/Literatur/Skripten/graphbook.html>.