Drawing Execution Graphs by Parsing

by

G.A.M. de Bruyn and O.S. van Roosmalen

95/07

ISSN 0926-4515

All rights reserved
editors: prof.dr. J.C.M. Baeten
         prof.dr. M. Rem
Drawing Execution Graphs by Parsing

G.A.M. de Bruyn and O.S. van Roosmalen
Department of Computing Science
Eindhoven University of Technology
The Netherlands

Abstract

In this paper we present an algorithm for drawing execution graphs. Such graphs represent the control flow in a program. The fact that a program is constructed according to a grammar is reflected in the corresponding execution graph. Therefore, we introduce graph production rules that are based on generally used programming language constructs. Each rule is applied to a certain class of topologies of the graph. By parsing an execution graph according to these rules a visually appealing layout of the graph can be generated. We take into account that the nodes in a graph can have variable sizes.

1 Introduction

In the field of computer science graphs are widely used. For the automatic generation of a graph drawing often rather complex algorithms must be used. However, graphs that are derived from a certain type of grammar rules can be drawn using very efficient algorithms.

Execution graphs are such a kind of graphs. They express all possible execution paths through a computer program, e.g., a process. Programming language constructs are reflected in the structure of the graph. An execution graph viewer [2] has been developed at Eindhoven University of Technology as part of the DEpendable Distributed Operating System (DEDOS) project [3]. This operating system is being developed as a platform for the execution of distributed real-time applications. As a support for off-line scheduling of hard real-time processes, the execution graph viewer helps the programmer to find bottle necks that prohibit a feasible schedule. The viewer is intended for the interactive manipulation of execution flow information, e.g., it helps to formulate scheduler directives. We describe the algorithm that was developed to generate proper drawings of execution graphs. It is fast enough for interactive use.

2 Execution graphs

For the purpose of scheduling, a computer program is divided into parts that we call beads. A bead is a consecutively executed group of statements or statement parts. An execution graph is a directed graph where the nodes represent a bead or a function call and the edges represent their precedence relations. We make a distinction between bead and call nodes because a call node may be replaced by the execution graph corresponding to its body. Interactively, the user may zoom in on selected function calls to investigate details of the control flow at a lower implementation level.

The topology of edges (representing precedence relations) of an execution graph is determined by the language constructs that are used in the program in question. The precedence relations restrict the order of execution of the nodes. If a node is executed then the next node to be executed is one of its successors. A node with more than one successor initiates a selection. The corresponding bead may contain an if statement or another statement causing the control flow to split into several branches. Those branches come together farther on in the graph. In case of a repetition, a sequence of nodes is executed more than one time. The corresponding program part contains, e.g., a while statement. In an execution graph, repetitions are characterized by cycles.

Figure 1 shows an example of an execution graph. It could be a representation of the following lines of a C++ program. This program part is (arbitrarily) divided into beads that contain one or two statements. Node 6 is a call node corresponding to the function call printer.print(b).

```cpp
0: if (a==1)
1:   b=a;
2: if (a==c)
3:   b=c;
else
4:   a = c;
    a++;
}
5: while (b>3)
{
6:   printer.print(b);
7:   b = b-a;
    b++;
}
```

Execution graphs are extracted from a program as follows. A graph generator accepts a program and stores in a graph-info file node numbers, the type of each node (bead or call node) and a list of predecessors. This graph information is parsed and the position of nodes and edges is calculated. Then the graph is drawn. The graph generator is the only component
that is programming language dependent. It may be integrated with the language compiler. To allow re-grouping of various nodes into a single node to influence the fine-grainedness of the scheduling problem we have decided to separately store the programs graph information instead of using the parsed program directly to obtain an execution graph.

3 Drawing execution graphs
Various algorithms have been developed for the automatic generation of a "good-looking" layout for graphs. To formalize "good-looking" aesthetic rules are used. Examples of generally used aesthetic rules are the minimization of crossings of edges and the minimization of the area occupied by the drawing. To help convey the intended meaning of a particular drawing, constraints can be used, e.g. shapes and relative size of certain icons may be fixed.

Generally, four kinds of graphs are distinguished. These are trees, planar graphs, hierarchical graphs and general undirected graphs. See [6] for a more detailed survey. Because our execution graphs are hierarchical graphs with a distinct tree-like structure, we will focus on trees and hierarchical graphs.

3.1 Trees
Trees are usually drawn following the straight-line standard. That means that levels are assigned to nodes and edges are drawn as straight-lines. Additional aesthetic rules are used such as centering a node horizontally above its children. In the algorithm for binary trees presented by Wetherell and Shannon [7] nodes are pushed as much to the left as possible, provided they are correctly placed with respect to their fathers and children. Reingold and Tilford [4] added that isomorphic subtrees have got mirror image drawings. Generalizations for n-ary trees are made by Walker [8] and Bloesch (allowing also variable-sized nodes) [1]. In all these algorithms two traversals through the graph are needed to produce a drawing.

3.2 Hierarchical Graphs
Electronic circuit layout graphs and visualization of data structures are examples of hierarchical graphs. These graphs have the common property to be directed and acyclic. Algorithms to compute a nice drawing are complex. Most of them are heuristic and derived from the algorithm of Sugiyama [5]. They basically consist of the following three phases: First, levels are assigned to nodes. The levels relate to a vertical displacement. For each long edge (i.e. an edge that spans several levels) a so-called dummy node is introduced for each level that is crossed. Second, nodes are sorted at each level by taking the average position of the predecessors, with the goal of minimizing crossings. This is done several times, upward and downward, till a minimum in edge crossings has been found or a user defined threshold has been reached. In the last phase, the nodes get their final coordinates. The number of bends in edges is minimized, nodes are centered over their predecessors and successors as much as possible.

3.3 Our approach
Because of their hierarchical structure, our execution graphs are of type hierarchical graphs. This means that a Sugiyama-like algorithm could be used. However, an execution graph is constructed according to a grammar. This is reflected in the drawing. This grammar is based on the grammar of generally used programming language constructs. Therefore, we introduce production rules to separate parts of an execution graph, so that nodes can be treated independently of the rest of the graph.

We don't assign levels to nodes. Assigning levels to nodes makes it hard to vary the node sizes: if nodes may not span more than one level, the distance between two levels will be determined by the longest node in the graph. This can cause much unnecessary empty space in the drawing.

As in most tree drawing algorithms, our algorithm needs only two passes to calculate the positions of the nodes. In the first pass preliminary coordinates of nodes are calculated as well as the width and height of the subgraphs. This is done recursively by taking synthesized attributes from subgraphs. In the second pass the coordinates get their final values.

4 Specification of graph drawings
In this section we specify our graph positioning algorithm. By presenting a graphical grammar in a kind of Backus-Naur form derived from generally used language constructs. We show what kind of graphs we accept and how an execution graph should be drawn. We will also introduce the semantic constraints and aesthetic rules.
4.1 Grammatical description

We introduce a graph grammar that is context sensitive in a kind of a Backus-Naur form (see Figure 2). The graphs \(G, K\) are non-terminals. Terminals are (call and bead) nodes (indicated by boxes containing \(n\)) and edges (indicated by arrows). We sometimes label an edge with a multiplicity, say \(p\). This means that in fact \(p\) edges are present with multiplicity one.

The graph production rules are related to the language constructs that we take into account. We allow for three types of program statements: simple sequences of statements, repetitions and selections. It is only necessary to distinguish instances of each of these types for as far they differ with respect to the induced precedence relations. Typically, the various repetition- and selection statements as found in, e.g., in C++ are covered by the productions, as well as selections due to the occurrence of dynamic binding to object member functions in object-oriented languages.

4.2 The node hierarchy

The node hierarchy in an execution graph is based on the the precedence relations between the nodes in the graph. These precedence relations are defined by the set of predecessors and successors of a node. To make the parsing of an execution graph easier, we expect that the nodes are numbered. This numbering reflects the precedence relations between the nodes. The start node (with number 0) of the graph is considered the node at the top of the hierarchy, a successor is considered as being lower in the hierarchy. The precedence relation in the graph, represented by a directed edge, reflects the execution order of the nodes: node \(m\) is a predecessor of \(n\) only if there exists a run of the program where \(n\) is the first node that is executed after \(m\) has terminated. Repetitions in a program imply cycles in the corresponding execution graph. These cycles "disturb" the hierarchy. Therefore we define the strong precedence relation:

**Strong predecessor**: Node \(m\) is a strong predecessor of \(n\) only if \(m\) is a predecessor of \(n\) and there is a path through the graph from the start node to \(m\) that does not pass through \(n\).
We base the hierarchy on the strong precedence relations between nodes and introduce a numbering accordingly. Let $N$ be the number of nodes of an execution graph:

1. Each node has a unique number $t$ where $0 \leq t < N$
2. If a node $m$ is a strong predecessor of node $n$ then $m$ has a number less than the number of $n$.
3. Each node is reachable by a path, starting in the start node that is increasing in the node numbers.
4. If a node $n$ is not reachable by an increasing path from a node $m$ nor $m$ is reachable by an increasing path from $n$ then $n$ and $m$ are part of different alternatives of a selection. (If $n$ has a number greater than $m$ then the alternative containing $m$ appears before the alternative containing $n$ in the program text.)

It can be shown that a numbering as described exists and can be generated in linear time if the start node is known [2].

### 4.3 Graphical representation

In this paragraph we describe how we want nodes and edges to be placed in order to express the language constructs on which the graph is based. We choose to take the upper-left corner of the screen as the origin. This implies that "lower" means "having a greater $y$ coordinate".

Our execution graph $GN$ is a set of $N$ nodes. We represent the nodes as boxes with their own width $dx$ and height $dy$. The upper-left corner is the position of the node. An edge is drawn between two nodes but does not have to be one straight line. At points where an edge can bend we introduce dummy nodes. To make a formal description of semantic constraints easier we represent an edge as a set of dummy nodes. Here we give a list of types and functions that we use in the remainder.

**Types:**

```plaintext
NODE =
[ ar : Nat, "Node number"
  x : real, "Position"
  y : real, "Height"
  pred : P(NODE), "Predecessors"
]

DUMMYNODE =
[ from : NODE, "Edge from from to to"
  x : real, "Position"
  y : real, "Height"
  dtype : {Normal, Loop}, "Type of the edge"
]
```

**Constants:**

- $N$ $\in$ $\mathbb{N}$ "Number of nodes"
- $GN$ $\in$ P(NODE) "The execution graph"

$DN$ $\in$ P(DUMMYNODE) "Set of dummy nodes"

**Functions:**

- $GN() : (nr, nr) \rightarrow P(NODE)$
  
  $GN(u, v) = \{n \in GN \mid u \leq n.nr < v\}$

  "Subset of $GN$"

- $GN() : nr \rightarrow NODE$

  $GN[u] = n$

  "Node in $GN$ with number $u$"

- $WD() : NODE \cup DUMMYNODE \rightarrow Nat.$

  if $n \in NODE$ : $WD(n) = n.dx$
  
  if $n \in DUMMYNODE$ : $WD(n) = 0$

  "The width of a node"

- $succ() : NODE \rightarrow P(NODE)$

  $succ(n) = \{s \in GN \mid n \in s \cdot pred\}$

  "The successors of $n$"

- $strp() : NODE \rightarrow P(NODE)$

  $strp(n) = \{p \in n \cdot pred \mid p.nr < n.nr\}$

  "The strong predecessors of $n$"

- $strs() : NODE \rightarrow P(NODE)$

  $strs(n) = \{s \in succ(n) \mid s.nr > n.nr\}$

  "The strong successors of $n$"

We use $strs'(n)$ to denote the transitive closure of the strong successor relation: nodes reachable by an increasing path from $n$.

**Concatenation.** To express the execution flow, we draw the graph from top to bottom. This means that every node is placed below its strong predecessors. We take a minimum distance $ay$ in the $y$ dimension. Now we can introduce the following semantic constraint:

$S1$ $(\forall p, n \in GN : p \in strp(n) : p.y + p.dy + ay \leq n.y)$

Each node has a $y$ coordinate greater than that of all its strong predecessors.

**Repetition.** A repetition causes sequence(s) of nodes to be executed a number of times. We want to show this by drawing a so-called "loop" edge from the last node of the repetition to its first node. Because we want to put nodes in the same sequence on a vertical line, the first and last node of a repetition are placed at the same $x$-coordinate. The rest of the nodes of the repetition are placed between the first and the last node. To identify the begin node and the end node of a repetition we introduce the following functions:

- $BREP() : NODE \rightarrow \{True, False\}$

  $BREP(n) = (strp(n) \neq n \cdot pred)$

  "$n$ is beginning of a repetition"

- $loopend() : NODE \rightarrow NODE$

  $loopend(n) \in GN \land loopend(n).nr = (\MAXp : p \in n \cdot pred \land n.nr < p.nr : p.nr)$

  "If $n$ is beginning of a repetition then $loopend(n)$ is the last node of the repetition"
We can formulate the following semantic constraint concerning repetitions:

\[ S_2 \ (v_b, e \in GN : BREP(b) \land e = \text{loopend}(b) : \ (\forall n : b.nr < n.nr < e.nr : b.y < n.y < e.y) \land b.x = e.x) \]

"Let b be the first node in a repetition and e the last node of the same repetition. Then all nodes with a number between b.nr and e.nr have a y coordinate between b.y and e.y. Also b has the same x coordinate as e."

Selection. A selection gives a number of alternative execution paths. For example, a C switch statement concerning repetitions:

Selection. A selection gives a number of alternative paths. For example, a C `switch` statement concerning repetitions:

\[ \text{selnext}(n) : NODE \rightarrow \{ \text{True, False} \} \]

"n is beginning of a selection"

\[ \text{MAXSC}(n) : NODE \rightarrow \text{MAXSC}(n) \]

\[ \text{MAXSC}(n).nr = (\max_{e \in GN : n \rightarrow e} : \text{succ}(n) : s.nr) \]

selnext\((n) : NODE \rightarrow NODE\)

\[ \text{selnext}(n) \rightarrow \text{BSEL}(n) \]

\[ \Rightarrow \text{selnext}(n).nr = (\min_{e \in GN : e \rightarrow n} : \text{MAXSC}(n).nr \land (\exists p : p \neq e \text{pred} : n.nr < p.nr < \text{MAXSC}(n).nr : e.nr) \]

If \( n \) is the beginning of a selection then \( \text{selnext}(n) \) is the node where the branches of this selection come together.

Now we can introduce the following semantic constraints for a selection:

\[ S_3 \ (\forall m, n \in GN : m \neq \text{strs}(n) \land n \neq \text{strs}(m) : m.nr > n.nr \Rightarrow m.x + d.x > n.x) \]

If nodes \( m \) and \( n \) are in different alternatives of a selection then the node with the greatest number is put to the right of the other node.

\[ S_4 \ (v_b, e \in GN : BSEL(b) \land e = \text{selnext}(b) : b.x = e.x \land (\forall n : b.nr < n.nr < e.nr : b.y < n.y < e.y)) \]

If \( b \) is the beginning of a selection and \( e \) is the next node following this selection then \( b \) and \( e \) are put on the same vertical line and all other nodes in the selection are put between \( b \) and \( e \).

\[ S_5 \ (v_m, n, d_n : m, n \in GN \land d_n \in DN \land d_n.from = m \land d_n.to = n) \]

\[ \text{dn.dtype} = \text{Normal} : (\forall v : m.nr < t.nr < n.nr \land m.y < t.y < n.y : d_n.x > t.x)) \]

A "normal" edge between two nodes \( m \) and \( n \) is drawn to the right of all the other nodes between \( m \) and \( n \).

Other language constructs. Apart from the basic language constructs we have discussed here, there are also "non-structural" constructs such as `goto` and `return` (in C). They are discussed in more detail in [2].

4.4 Aesthetic Rules

The aesthetic rules (in order of priority) are:

\[ \text{AR1} \text{ Center a node over its predecessors and successors.} \]

\[ \text{AR2} \text{ Put a node as close as possible below its strong predecessors.} \]

\[ \text{AR3} \text{ Minimize the number of bends of an edge.} \]

\[ \text{AR4} \text{ Minimize the area occupied by the graph.} \]

Besides a list of aesthetic rules, we also give an aesthetic constraint, i.e. a rule that is followed strictly.

\[ \text{AC1} \text{ Between any two (dummy) nodes there is a distance of at least } \alpha \text{ in } y \text{ dimension or a distance of at least } \alpha \text{ in } x \text{ dimension.} \]

4.5 Sub Execution Graphs (SEGs)

In the beginning of this section we have given a grammar describing execution graph topologies. One can observe that most of the subgraphs are execution graph themselves. Therefore, we introduce the concept of Sub Execution Graphs (SEGs). A SEG as part of a bigger graph can be treated independently of the rest. This implies that isomorphic SEGs are drawn the same. This can be useful for recognizing similar patterns (e.g. of the same function body) in the graph. For each SEG the relations between the positions of their nodes can be calculated as if it were an execution graph itself. Later on the SEGs are put together in a new SEG taking the semantic constraints and aesthetic rules into account, till we get the whole execution graph. Based on the properties of an execution graph we can give the following definition of a SEG:

\[ \text{definition: A subgraph } GN[u,v] \ (0 \leq u \leq v \leq N) \text{ is a SEG if and only if:} \]

1. \( GN[u,v] \) is empty or each node in \( GN[u,v] \) is reachable by an increasing path starting in \( GN[u] \).

2. And each node except \( GN[u] \) (unless \( GN[u,v] \) is empty) contains only predecessors and successors with a number of at least \( u \).
5 Derivation of the positions of nodes and edges

In the previous chapter we have already specified how we can draw an execution graph. We give production rules by which a graph can be split into a set of subgraphs, nodes and dummy nodes. For each production rule we derive, according to the semantic rules and aesthetic constraints, the relations between the positions of these subgraphs, nodes and dummy nodes. At the end we will describe how from these relations the final positions can be calculated in an efficient way.

5.1 Drawing of edges.

A normal edge (i.e., not a loop edge) from node m to node n is drawn by a line (containing bends) starting in \((m.x + m.dx/2, m.y + m.dy)\) and ending in \((n.x + n.dx/2, n.y)\). According to aesthetic rule AR3 we want to reduce the number of bends of an edge. In fact, every edge can be drawn containing at most two bends in each edge.

The second bending point can be derived from Figure 3: Edge from m to n is drawn by a line (containing bends) starting in \((m.x, m.y + m.dy + ay)\) (with \(dn.x\) to be determined still). This is the first bending point. The second bending point can be derived from \(dn\) and \(n\) and is \((dn.x, n.y - ay)\).

![Diagram]

Figure 3: Edge from m to n with two bends (left) and a direct edge (right). In the area surrounded by a dashed line there should be no interference with other nodes.

In a situation as depicted in the right hand side of Figure 3 we don't require a dummy node at all: if an edge from node m to n can be drawn vertically as a straight line from m to n upto a distance of ay before n, from where it bends to n.

To assure that we can draw edges as depicted in Figure 3 we introduce the following invariant:

\[
\begin{align*}
\text{invariant DIRECT-EDGE} \\
(\forall m, n, dn : m, n \in GN \land dn \in DN \land m = dn.from.nr \land n = dn.to.nr) : \\
(\forall p, x, y : \\
p \in GN \cup DN \land p.x \leq z \leq p.x + p.dx \\
\land p.y \leq y \leq p.y + p.dy \\
\land (dn.x - az < z < m.x + m.dx/2) \\
\land m.y + m.dy < y < dn.y) \\
\land \\
((dn.x - az < z < dn.x + az) \\
\land m.y + m.dy + ay < y < n.y - ay) \\
\land \\
((n.x + n.dx/2 < z < n.x + az) \\
\land dn.x - az < z < n.x + n.dx/2) \\
\land n.y - ay < y < n.y)
\end{align*}
\]

The invariant states that if there is a dummy node \(dn\) for the edge from node m to n, there are three strips in which no nodes are situated as can be seen in the left hand side of Figure 3. If there is no dummy node for the edge from m to n, then we get two strips as depicted in the right hand side.

If we want to draw a loop edge as in the example in our introduction then DIRECT-EDGE is not sufficient to avoid overlap of nodes and edges. We have to guarantee that also the two boxes right from m en n as depicted in Figure 4 are free from other boxes or dummy nodes. Therefore, we introduce an additional invariant:
We use (as in the production rules for the graph grammar \(AGG = \{NT, T, P, G\}\) here, \(NT = \{G, K\}\) is the set of (parameterized) non-terminals. We use (as in the production rules for the graph) \(G\) for SEGs and \(K\) for branches in selections. \(T = GN \cup DN\) is the set of terminals and \(G\) is the axiom (the start graph).

For a non-terminal \(O \in NT\), \(O_{u,v}\) corresponds with the subgraph \(GN[u,v]\) and the dummy nodes \(\{dn \in DN | u \leq \text{dn.from.nr} < v \land u \leq \text{dn.to.nr} < v \lor \text{BSEL(dn.from)}\}\), where \(0 \leq u \leq v \leq N\). \(P\) is the set of productions. For \(p \in P\):

\[
p = (O_{u,v}, [NO, DU, GR], C)
\]

where \(O \in NT\) and

\[
\begin{align*}
NO &\subseteq \{n \in GN | u \leq n.nr < v\} \\
GR &\subseteq \{O_{i,j} | O \in NT \land u \leq i < j \leq v\} \\
DU &\subseteq \{d \in DN | u \leq \text{d.from.nr} < v \lor u \leq \text{d.to.nr} < v\}
\end{align*}
\]

This means that \(O_{u,v}\) is split into a set of subgraphs \(GR\), nodes \(DN\) and dummy nodes \(DU\). The condition \(C\) determines the class of topologies to which the with \(O_{u,v}\) corresponding graph \(GN[u,v]\) must adhere. Sometimes the conditions of two production rules with the same non-terminal can overlap. We do this to avoid (unnecessary) long boolean expressions. In case of non-determinism the production rule introduced last has priority. In the following we will introduce the productions and at the same time we give the relations between the positions.

Concatenation. Let \(G_{u,v}\) be a SEG, where \(0 \leq u \leq v \leq N\). In case \(G_{u,v}\) is empty (i.e. \(u = v\)) the graph can't be split and there are no (dummy) nodes produced. This means that we get the following production rule:

\[
g_1 : G_{u,v} ::= \emptyset \\
\text{if } u = v
\]

Because in case of \(g_1\) \(G_{u,v}\) is empty, there is no relation to be defined between the position of the nodes. In case \(u < v\) we can split \(G_{u,v}\). If \(GN[u]\) is not the beginning of a selection or repetition then \(G_{u,v}\) can be split into the SEGs \(G_{u,u+1}\) and \(G_{u+1,v}\). If \(u+1 = v\) then trivially \(G_{u+1,v}\) is a SEG. If \(u+1 < v\) it is also not difficult to prove that \(G_{u+1,v}\) is a SEG.

Thus we can introduce the following production rule:

\[
g_2 : G_{u,v} ::= n G_{u+1,v} \\
\text{if } u < v \\
\text{where } n = GN[u]
\]

We now discuss the position of \(n\). If \(G_{u+1,v}\) is empty then no equation has to be introduced for \(n\).

If \(u+1 = v\) then let \(ng = GN[u+1]\). According to \(S1\) we must choose \(n.y\) and \(n.dy\) such that \(n.y \leq ng.y + n.dy + ay\) and because \(ng.y\) is a strong successor of \(n\). Moreover, we know that \(n\) is the only strong predecessor of \(ng\), because of the property of a SEG: \(ng.y\) has no predecessors or successors \(i\) with \(i.nr < u\).

Now no constraint is left to influence the relation between \(n.y\) and \(ng.y\). As far as the aesthetic rules \(AR1, AR2\) and \(AR4\) are concerned we put \(ng\) as close as possible below \(n\). So we get:

\[
n.y + n.dy + ay = ng.y
\]

and \(AC1\) is satisfied. As far as \(n.x\) and \(ng.x\) are concerned there are no constraints left that influence their relation. According to \(AR1, AR2\) and \(AR4\) we choose:

\[
n.x = ng.x
\]

We can depict this as in Figure 5.

An edge can be drawn straight from \(n\) to \(ng\). We see that if invariant \(\text{DIRECT-EDGE}\) holds for \(G_{u+1,v}\), it also holds for \(G_{u,v}\).  

![Figure 4: m is the beginning of a repetition](image-url)

![Diagram](image-url)
Figure 5: n is ng's only strong predecessor and n is not the beginning of a repetition or selection

Repetition. Let $G_{u,v}$ be a SEG, where $0 \leq u < v \leq N$. So $G_{u,v}$ is not empty. Let $m = GN[u]$ and BREP$(m)$. So $G_{u,v}$ starts with a repetition. Let $n = loopend(m)$. We assume that $n \in G_{u,v}$.

Now we can introduce the following production rule:

$$g_3: G_{u,v} ::= \begin{cases} m & G_{u+1,w} \ \text{if } u < v \wedge \text{BREP}(m) \wedge m = GN[u] \\ n & G_{u+1,v} \ \text{where } m = GN[u] \wedge w = n.nr \wedge n = loopend(m) \end{cases}$$

A graphical representation of this rule is depicted in Figure 6. Because $m$ has no predecessors $s$ with $u+1 < s.nr < w$ and $G_{u,v}$ is a SEG, we can conclude that $G_{u+1,w}$ is also a SEG. Similarly we can conclude that $G_{u+1,v}$ is a SEG.

About the positions of the nodes: If $G_{u+1,w}$ is empty then $n$ is the successor of $m$.

If $G_{u+1,w}$ is not empty then $mg = GN[u+1]$. Let $ng$ be $GN[w+1]$ if $G_{u+1,v}$ is not empty. Because of S1 and AR2 we choose:

$$m.y + m.dy + ay = mg.y$$
$$n.y + n.dy + ay = ng.y$$

This does not fix $n.y$ sufficiently. We must also take care that $n$ is placed below all its strong predecessors. This means that, taking S1 and AR2 into account:

$$n.y = (\text{MAX}_p: p \in G_{u+1,w} \cup \{m\}) : p.y + p.dy + ay$$

Finally we must add a dummy node $dn$ that guides the (loop) edge from $n$ to $m$. So $dn.from = n$, $dn.to = m$ and $dn.dtype = \text{Loop}$. According to S5 we must choose $dn.x$ such that $dn$ is placed left or right from all the nodes in $G_{u+1,v}$. To minimize the area occupied by the graph, (AR4) we must put $dn$ as close as possible to $G_{u+1,v}$. For reasons of uniformity we place $dn$ always right from all the nodes on $G_{u+1,v}$.

Now we get by also taking AC1 into account:

$$dn.x = (\text{MAX}_p: p \in (G_{u+1,w} \cup \{m\}) : p.x + p.dx + ax)$$

$dn.y$ is more simple to choose. In fact, as we know, $dn$ represents all the dummy nodes that should represent the edge from $n$ to $m$. There is no aesthetic rule on which we can base our choice for $dn.y$, as long as $m.y \leq dn.y \leq n.y$. We take:

$$dn.y = m.y + m.dy/2$$

The above considerations are illustrated in Figure 6.

Figure 6: m is the beginning of a repetition

Sometimes the graph generator can generate an execution graph which exhibits in the first node of the repetition the evaluation of a guard of a selection. We will not discuss this case in this paper (see [2]).
Selection. Finally we can introduce the last production rule for a SEG. We will see that a SEG that begins with a selection can not be simply split into two different SEGs as with a repetition. Therefore, we introduce a subgraph \( K \) consisting of the SEGs that represent the "branches" in a selection.

Let \( G_{u,v} \) be a SEG, where \( 0 \leq u < v \leq N \). Let \( n = GN[u] \) and BSEL(n) and ~BREP(n). So \( G_{u,v} \) starts with a selection. Now we can introduce the following production rule:

\[
\begin{align*}
g4 : G_{u,v} ::= & \; n \; K_{u+1,w} \; G_{w,v} \\
& \text{if } u < v \land \text{BSEL}(n) \land \text{~BREP}(n) \\
& \text{where } n = GN[u] \\
& \text{mg = selectionend}(n) \land \\
& \text{if } \text{mg exists: } w = \text{mg}.nr \\
& \text{otherwise } \; w = v
\end{align*}
\]

We write \( K_{u+1,w} \) because this part of the graph is not an SEG. In fact, \( K_{u+1,w} \) is a set of SEGs, where each SEG starts with a successor of \( n \). An example is depicted in Figure 7.

\[
\text{According to S4 we have to choose } mg.y \text{ such that } mg \text{ is placed below all nodes in } K_{u+1,w}. \text{ Because of AR2 we choose:}
\]

\[
mg.y = (\text{MAX} : p \in K_{u+1,w} : p.y + p.dy + ay)
\]

The following choice is in accordance with S4:

\[
n.x = mg.x
\]

However, there is not only a relation between \( mg.x \) and \( n.x \). According to AR1 we try to center \( n \) over its successors in \( K_{u+1,w} \). But over which successors do we exactly have to center? In stead of taking the average of the x coordinates of all the immediate successors we choose to take only the left and right most immediate successors: then it seems more like \( n \) is centered over the drawing of \( K_{u+1,w} \). So now we get:

\[
n.x = (\text{MIN} : s \in K_{u+1,w} \land \{n\} = s.pred : s.x) + \\
(\text{MIN} : s \in K_{u+1,w} \land \{n\} = s.pred : s.x)/2
\]

It can be that \( mg \) is a successor of \( n \). Then there is a long edge from \( n \) to \( mg \). Therefore, we introduce a dummy node \( dn \), where \( dn.from = n \), \( dn.to = mg \) and \( dn.dtype = \text{Normal} \). According to S7 we have to place \( dn \) below all nodes in \( K_{u+1,w} \). Taking also AC1 and AR4 into account we get:

\[
dn.x = (\text{MAX} : p \in K_{u+1,w} : p.x + \text{WD}(p)) + ax
\]

if \( \text{mg} \in \text{succ}(n) \)

Now there is still \( K_{u+1,w} \) left to divide into SEGs. Therefore we introduce the following production rules:

\[
k1 : K_{u,v} ::= \emptyset \\
\text{if } u = v
\]

\[
k2 : K_{u,v} ::= K_{u,w} \; G_{w,v} \\
\text{if } u < v \\
\text{where } \{n\} = GN[u].pred \land \\
w = \text{mg}.nr
\]

About the positioning of the nodes in \( K_{u,v} \): There are only two constraints we have to take into account: S3 and AC1. To minimize the area occupied by the drawing (AR4) we have to "push" \( G_{w,v} \) as close as possible to \( K_{u,w} \). So now we get:

\[
(\text{MAX} : m \in K_{u,w} : m.x + \text{WD}(m)) + ax
\]

\[
(\text{MIN} : m \in G_{w,v} : m.x)
\]

We can depict this in Figure 8. Note that this does not disturb the invariant DIRECT-EDGE. Because \( K_{u,w} \) and \( G_{w,v} \) are placed next to each other, and if we assume this invariant is satisfied in \( K_{u,w} \) and \( G_{w,v} \), then we can conclude by means of induction that it is also satisfied in \( K_{u,v} \).
5.3 Calculation of the positions of nodes

From the relations between the positions of the nodes, their final positions can be calculated. This is done in two passes through the execution graph. In the first pass, the relative positions of nodes and the sizes of the subgraphs are calculated as synthesized attributes. In the second pass, the frames in which the subgraphs reside are calculated. At the same time the sizes of the subgraphs are calculated as synthesized attributes. In the second pass, the frames in which the subgraphs reside are calculated. At the same time the final positions of the nodes are calculated from their relative positions and the frame in which they reside.

First pass. We introduce the following functions:

\[

definitions: \begin{align*}
\text{firstx}(u, v) &: (nr, nr) \rightarrow \text{Real} \\
\text{firsty}(u, v) &: (nr, nr) \rightarrow \text{Real} \\
\text{height}(u, v) &: (nr, nr) \rightarrow \text{Real} \\
\text{width}(u, v) &: (nr, nr) \rightarrow \text{Real}
\end{align*}
\]

We calculate for each SEG \( G_{u,v} \) the positions of the nodes as if \((0,0)\) is the upper-left corner of the drawing of \( G_{u,v} \). The \text{firstx} coordinate of the first node is stored in \text{firstx}(u, v). To calculate, e.g., in \( g4 \) the difference between \( n \)'s leftmost and rightmost placed successor we need for the subgraph \( K_{u+1,v} \) the following functions: \text{firstx} for the leftmost successor of \( n \) and \text{firstx}2 for the rightmost successor of \( n \). Then \text{firstx2}(u + 1, v) - \text{firstx}(u + 1, v) \) is the difference we needed. To calculate \text{firstx}2 in \( k2 \) we also need the \text{width} of the subgraphs. At the same time the \text{height} of the subgraphs can already be calculated.

Second pass. For this pass the following functions are introduced:

\[

definitions: \begin{align*}
\text{beginx}(u, v) &: (nr, nr) \rightarrow \text{Real} \\
\text{beginy}(u, v) &: (nr, nr) \rightarrow \text{Real}
\end{align*}
\]

We draw each subgraph \( G_{u,v} \) or \( K_{u,v} \) in a frame with upper-left corner \((\text{beginx}(u, v), \text{beginy}(u, v))\). Starting from a user-defined upper-left corner \((lx, ly)\) for \( G_{0,N} \) (i.e., \( lx = \text{beginx}(u, v) \) and \( ly = \text{beginy}(u, v) \)), we can calculate for each subgraph its final upper-left corner. If for a node \( n \) its preliminary position is known then \((x, y) = \text{beginx}(u, v) + n.x, \text{beginy}(u, v) + n.y) \) is its final position.

When the final positions of all the (dummy) nodes are known, the following must yield for a non-empty subgraph \( SG \):

\[
\begin{align*}
\text{beginx} &= (\text{MIN}_{p \in \text{SG}} px) \\
\text{beginy} &= (\text{MIN}_{p \in \text{SG}} py)
\end{align*}
\]

6 Conclusions

We have derived an algorithm to generate drawings of execution graphs in which general language constructs such as repetition and selections are taken into account.

The size of different nodes has no effect on the complexity of the algorithm and therefore execution graphs with different node sizes can be handled.

The algorithm to calculate the positions of the nodes is \( O(N) \), where \( N \) is the number of nodes. More precise: Only two passes through the graph are needed. A few addition operations per node are required to calculate its position. We can conclude that the algorithm is fast enough for interactive use.

The presented algorithm for drawing graphs is based on the grammar describing the graph topology. Although we have applied the method to execution graphs, considering a limited number of programming language constructs, the method is more generally applicable. Not only additional language constructs leading to more variety in the graph topology can be included [2], but also the drawing of other types of graphs based on a grammar can be tackled in this fashion.

References


* e-mail: wsinonno@win.tue.nl
<table>
<thead>
<tr>
<th>Year</th>
<th>Author(s)</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>93/01</td>
<td>R. van Geldrop</td>
<td>Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.</td>
</tr>
<tr>
<td>93/02</td>
<td>T. Verhoeff</td>
<td>A continuous version of the Prisoner’s Dilemma, p. 17</td>
</tr>
<tr>
<td>93/03</td>
<td>T. Verhoeff</td>
<td>Quicksort for linked lists, p. 8.</td>
</tr>
<tr>
<td>93/04</td>
<td>E.H.L. Aarts, J.H.M. Korst, P.J. Zwietering</td>
<td>Deterministic and randomized local search, p. 78.</td>
</tr>
<tr>
<td>93/05</td>
<td>J.C.M. Baeten, C. Verhoef</td>
<td>A congruence theorem for structured operational semantics with predicates, p. 18.</td>
</tr>
<tr>
<td>93/06</td>
<td>J.P. Veltkamp</td>
<td>On the unavoidability of metastable behaviour, p. 29</td>
</tr>
<tr>
<td>93/07</td>
<td>P.D. Moerland</td>
<td>Exercises in Multiprogramming, p. 97</td>
</tr>
<tr>
<td>93/08</td>
<td>J. Verhoosel</td>
<td>A Formal Deterministic Scheduling Model for Hard Real-Time Executions in DEDOS, p. 32.</td>
</tr>
<tr>
<td>93/10</td>
<td>K.M. van Hee</td>
<td>Systems Engineering: a Formal Approach Part II: Frameworks, p. 44.</td>
</tr>
<tr>
<td>93/16</td>
<td>H. Schepers, J. Hooman</td>
<td>A Trace-Based Compositional Proof Theory for Fault Tolerant Distributed Systems, p. 27</td>
</tr>
<tr>
<td>93/17</td>
<td>D. Alstein, P. van der Stok</td>
<td>Hard Real-Time Reliable Multicast in the DEDOS system, p. 19.</td>
</tr>
<tr>
<td>93/18</td>
<td>C. Verhoef</td>
<td>A congruence theorem for structured operational semantics with predicates and negative premises, p. 22.</td>
</tr>
<tr>
<td>93/19</td>
<td>G-J. Houben</td>
<td>The Design of an Online Help Facility for ExSpect, p.21.</td>
</tr>
</tbody>
</table>

A Typechecker for Bijective Pure Type Systems, p. 28.

Relational Algebra and Equational Proofs, p. 23.

Pure Type Systems with Definitions, p. 38.


Multi-dimensional Petri nets, p. 25.

Finding all minimal separators of a graph, p. 11.

A Semantics for a fine λ-calculus with de Bruijn indices, p. 49.

GOLD, a Graph Oriented Language for Databases, p. 42.

On Vertex Ranking for Permutation and Other Graphs, p. 11.

Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.


ILIAS, a sequential language for parallel matrix computations, p. 20.

Real Time Process Algebra with Infinitesimals, p. 39.

Abstract Reduction and Topology, p. 28.

Non Interleaving Process Algebra, p. 17.

Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.

A general conservative extension theorem in process algebra, p. 17.

Job Shop Scheduling by Constraint Satisfaction, p. 22.
<table>
<thead>
<tr>
<th>Page</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>93/41</td>
<td>A. Bijlsma</td>
<td>Temporal operators viewed as predicate transformers, p. 11.</td>
</tr>
<tr>
<td>93/42</td>
<td>P.M.P. Rambags</td>
<td>Automatic Verification of Regular Protocols in P/T Nets, p. 23.</td>
</tr>
<tr>
<td>93/43</td>
<td>B.W. Watson</td>
<td>A taxonomy of finite automata construction algorithms, p. 87.</td>
</tr>
<tr>
<td>93/44</td>
<td>B.W. Watson</td>
<td>A taxonomy of finite automata minimization algorithms, p. 23.</td>
</tr>
<tr>
<td>93/48</td>
<td>R. Gerth</td>
<td>Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.</td>
</tr>
<tr>
<td>94/01</td>
<td>P. America, M. van der Kammen, R.P. Nederpelt, O.S. van Roosmalen, H.C.M. de Swart</td>
<td>The object-oriented paradigm, p. 28.</td>
</tr>
<tr>
<td>94/02</td>
<td>F. Kamareddine, R.P. Nederpelt</td>
<td>Canonical typing and Π-conversion, p. 51.</td>
</tr>
<tr>
<td>94/04</td>
<td>J.C.M. Baeten, J.A. Bergstra</td>
<td>Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.</td>
</tr>
<tr>
<td>94/06</td>
<td>T. Basten, T. Kunz, J. Black, M. Coffin, D. Taylor</td>
<td>Time and the Order of Abstract Events in Distributed Computations, p. 29.</td>
</tr>
<tr>
<td>Year</td>
<td>Authors</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>94/08</td>
<td>O.S. van Roosmalen</td>
<td>A Hierarchical Diagrammatic Representation of Class Structure, p. 22.</td>
</tr>
<tr>
<td>94/09</td>
<td>J.C.M. Baeten, J.A. Bergstra</td>
<td>Process Algebra with Partial Choice, p. 16.</td>
</tr>
<tr>
<td>94/10</td>
<td>T. Verhoeven</td>
<td>The testing Paradigm Applied to Network Structure. p. 31.</td>
</tr>
<tr>
<td>94/13</td>
<td>R. Seljée</td>
<td>A New Method for Integrity Constraint checking in Deductive Databases, p. 34.</td>
</tr>
<tr>
<td>94/14</td>
<td>W. Peremans</td>
<td>Ups and Downs of Type Theory, p. 9.</td>
</tr>
<tr>
<td>94/16</td>
<td>R.C. Backhouse, H. Doornbos</td>
<td>Mathematical Induction Made Calculational, p. 36.</td>
</tr>
<tr>
<td>94/18</td>
<td>F. Kamareddine, R. Nederpelt</td>
<td>Refining Reduction in the Lambda Calculus, p. 15.</td>
</tr>
<tr>
<td>94/19</td>
<td>B.W. Watson</td>
<td>The performance of single-keyword and multiple-keyword pattern matching algorithms, p. 46.</td>
</tr>
<tr>
<td>94/20</td>
<td>R. Bloo, F. Kamareddine, R. Nederpelt</td>
<td>Beyond β-Reduction in Church's λ→, p. 22.</td>
</tr>
<tr>
<td>94/22</td>
<td>B.W. Watson</td>
<td>The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.</td>
</tr>
<tr>
<td>94/23</td>
<td>S. Mauw and M.A. Reniers</td>
<td>An algebraic semantics of Message Sequence Charts, p. 43.</td>
</tr>
<tr>
<td>94/25</td>
<td>T. Kloks</td>
<td>K_{1,3}-free and W_4-free graphs, p. 10.</td>
</tr>
</tbody>
</table>


Correctness of Real Time Systems by Construction, p. 22.

Process Algebra with Feedback, p. 22.

A Boyer-Moore type algorithm for regular expression pattern matching, p. 22.


A formalization of the Ramified Type Theory, p. 40.

The Barendregt Cube with Definitions and Generalised Reduction, p. 37.

Delayed choice: an operator for joining Message Sequence Charts, p. 15.

Canonical typing and Π-conversion in the Barendregt Cube, p. 19.

Simulating and Analyzing Railway Interlockings in ExSpect, p. 30.

Point-free substitution, p. 10.

On the equivalence covering number of splitgraphs, p. 4.

Distributed Consensus and Hard Real-Time Systems, p. 34.

Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.

Concatenation of Graphs, p. 7.

Category Theory as Coherently Constructive Lattice Theory: An Illustration, p. 35.

Verifying Sequentially Consistent Memory, p. 160

Tutorial voor de ExSpect-bibliotheek voor "Administratieve Logistiek", p. 43.
<table>
<thead>
<tr>
<th>Year/ID</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>94/46</td>
<td>R. Bloo, F. Kamareddine, R. Nederpelt</td>
<td>The $\lambda$-cube with classes of terms modulo conversion, p. 16.</td>
</tr>
<tr>
<td>94/47</td>
<td>R. Bloo, F. Kamareddine, R. Nederpelt</td>
<td>On II-conversion in Type Theory, p. 12.</td>
</tr>
<tr>
<td>94/48</td>
<td>Mathematics of Program Construction Group</td>
<td>Fixed-Point Calculus, p. 11.</td>
</tr>
<tr>
<td>94/50</td>
<td>H. Geuvers</td>
<td>A short and flexible proof of Strong Normalization for the Calculus of Constructions, p. 27.</td>
</tr>
<tr>
<td>94/52</td>
<td>W. Penczek, R. Kuiper</td>
<td>Traces and Logic, p. 81</td>
</tr>
<tr>
<td>95/01</td>
<td>J.J. Lukkien</td>
<td>The Construction of a Small Communication Library, p. 16.</td>
</tr>
<tr>
<td>95/02</td>
<td>M. Bezem, R. Bol, J.F. Groote</td>
<td>Formalizing Process Algebraic Verifications in the Calculus of Constructions, p. 49.</td>
</tr>
<tr>
<td>95/03</td>
<td>J.C.M. Baeten, C. Verhoeven</td>
<td>Concrete process algebra, p. 134.</td>
</tr>
<tr>
<td>95/05</td>
<td>P. Severi</td>
<td>A Type Inference Algorithm for Pure Type Systems, p. 20.</td>
</tr>
</tbody>
</table>