Flow Diagram Decomposition Using Graph Transformations

Arend Rensink and Maria Zimakova

August 20, 2009

Content

Abstract

The key challenge of model transformations in model-driven development is in transforming higher-level abstract models into more concrete ones that can be used to generate implementation level models, including executable business process representations and program code. Many of the modelling languages (like UML Activity Diagrams or BPMN) use unstructured flow graphs to describe the operation sequence of a business process. If a structured language is chosen as the executable representation, it is difficult to compile the unstructured flows into structured statements. Even if a target language structure contains *goto*-like statements it is often simpler and more efficient to deal with programs that have structured control flow to make the executable representation more understandable.

In this paper, we take a first step towards an implementation of existing decomposition methods using graph transformations, and we evaluate their effectiveness with a view to readability and essential complexity measures.

Chapter 1.
Introduction

Over the last few years, a new option has evolved to define solutions in software industry: Model-Driven Development (MDD). The key challenge of model transformations in MDD is in transforming higher-level abstract models into more concrete ones that can be used to generate implementation level models, including executable business process representations and program code. With this trend, the decomposition of the models into structured elements is of increasing importance.

In the large, a number of motivations can be given to justify the implementation of this work:

- Imagine a dynamic behaviour of business process is described as an unstructured flow graph (which can represent, by-turn, a UML activity or BPMN diagrams). If a structured language is chosen as the target executable representation, it is difficult to transform the unstructured flows into structured statements. This problem is analyzed, for instance, in [EKR08] and attempts to compile UMLA to BPEL programs; the last issue is discussed, for instance, in [ZHB05]. The main task of our graph transformations is to translate the unstructured *goto*-like statements into well-structured statements in the target language.
- The second very important reason for the presented work is to improve software reliability and readability – making programs less error prone and easier to understand. Because understanding of behavior is an essential prerequisite to effective program development and modification, programmers are forced to devote substantial time to this task [CV06].

There exist today a number of variants on the idea of well-structured models. A lot of restructuring methods were done in the context of flow diagram decomposition. It is commonly agreed that a natural interpretation of flow diagrams is in terms of *graphs* – essentially, just nodes with connecting edges. Consequently, a most natural implementation of flow diagram decomposition methods is by *graph transformations*.

The aim of this work is to bridge the gap between formalism of the existing flow diagram decomposition methods and practical implementation in terms of graph transformations to use it for modern programming environments including executable business process languages.

The remainder of this paper is structured as follows: after providing the basic definitions to set the stage in Chapter 2, we discuss the flow graph decompositions and complexity measure problem in Chapter 3. We consider these to be the heart of our contribution. In Chapter 4 we implement those methods with graph transformations, employing the graph-transformation tool Groove [Ren04] for rule execution. Finally, in the conclusion (Chapter 5) we come back to the above considerations, evaluate our results and discuss plans for future work.

Chapter 2.

Basic Notions

Graphs and flow graphs. One of the core concepts of this paper is that of *graphs*. We assume a countable universe Λ of labels. We start by repeating the usual definition of a graph.

Definition 1 (labeled directed graph) A *labeled directed graph* is a tuple $G = (N, E, \lambda)$ where

- N is a finite nonempty set called a set of nodes;
- $E \subseteq N \times \Lambda \times N$ is a set of edges;
- λ is a *labeling function* $\lambda: N \cup E \to \Lambda$.

Each edge in a directed graph is a triple (v, a, w). We say the edge *leaves* v and *enters* w, v is a predecessor of w, and w is a successor of v, v is a *source* node and w is a *target* node of the edge. Given $e = (v, a, w) \in E$, we denote $src(e) = v$, $tgt(e) = w$ and $a = \lambda(e)$ for its source, target and label, respectively. An edge (v, a, v) is a *loop*. A labeled directed graph $G_1 = (N_1, E_1, \lambda_1)$ is a subgraph of a graph $G_2 = (N_2, E_2, \lambda_2)$ if $N_1 \subseteq N_2$, $E_1 \subseteq E_2$, $\Lambda_1 \subseteq \Lambda_2$ and $\lambda_1 \leq \lambda_2$ in the sense of following

$$\forall x \in N_2 \cup E_2 \quad \lambda_2(x) = \begin{cases} \lambda_1(x) \in \Lambda_1, x \in N_1 \cup E_1; \\ l \in \Lambda_2, otherwise. \end{cases}$$

Definition 2 (path) A *path* in a graph G is an alternating sequence of nodes and edges represented as $\{v_1, e_1, v_2, e_2, \ldots\}$ beginning and ending with nodes such that for each $i \geq 1$ we have $v_i \in N$, $e_i \in E$, $src(e_i) = v_i$ and $tgt(e_i) = v_{i+1}$.

There is a path of no edges from any node to itself. A node w is *reachable* from a node v if there is a path from v to w. The path length $|p|$ of $(v_1, \ldots, v_k)$ is $k - 1$. The concatenation of two paths p, q is denoted by pq, where we require p to be finite and end at the initial node of q [Har89].

Let G be a labeled directed graph as above with a labelling function $\lambda: N \cup E \to \Lambda$, then a path $p = \{v_1, e_1, v_2, e_2, \ldots, v_{k-1}, e_{k-1}, v_k\}$ in G can be represented by the *word* from the *alphabet* Λ as follows:

$$\lambda(p) = \lambda(v_1)\lambda(e_1)\lambda(v_2) \ldots \lambda(v_{k-1})\lambda(e_{k-1})\lambda(v_k).$$

We call this the *word representation* of p.

Let us imagine now that given directed graph G, there is the equivalent *undirected graph* G' such as $N_G = N_{G'}$ and $E_{G'}$ was obtained from E_G by replacing all of directed edges in G with undirected edges. In undirected graph G' two nodes are called *connected* if there is a path between them. An undirected graph G' is called *connected* if every pair of distinct nodes in the graph G' can be connected through some path. A *connected component* in an undirected graph G' is a maximal connected subgraph.

Definition 3 (connected graph) A directed graph G is called *weakly connected*, or just *connected*, if replacing all of its directed edges with undirected edges produces a connected undirected graph G'.

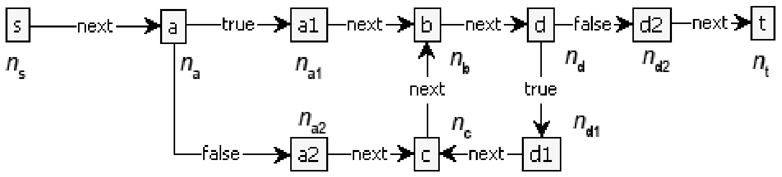


Figure 1: *Flow graph example*

An undirected *tree* is a connected graph with no cycles. A *directed tree* is a directed graph which would be a tree if the directions on the edges were ignored. A tree is called a *rooted tree* if one node has been designated the *root*, in which case the edges have a natural orientation, towards or away from the root.

A directed graph is called *strongly connected* if there is a path from each node in the graph to every other node.

Definition 4 (strongly connected component) The *strongly connected components* (SCC) of a directed graph are its maximal strongly connected subgraphs.

Now let us define the flow graph as follows.

Definition 5 (flow graph) A *flow graph* Φ is a triple (G, s, t), where

- $G = (N, E, \lambda)$ is a (weakly) connected labeled directed graph;
- Node $s \in N$ is the unique *start node* such that there are no incoming edges to s in G.
- Node $t \in N$ is the unique *terminal node* such that there are no outgoing edges to t in G.

A flow graph is *empty*, denoted ε, if $N = \{s, t\}$ and $E = \{e\}$ where $e = (s, l, t)$, $1 \in \Lambda$ is a distinguished label of e. A flow graph is an *elementary a-labelled* flow graph if $N = \{s, v, t\}$ and $E = \{e_1, e_2\}$ where $e_1 = (s, l, v)$, $e_2 = (v, l, t)$ and $\lambda(v) = a$.

Figure 1 shows the simple example of a flow graph graphical representation, which will be used throughout this paper, because it contains most of the features needed to explain the transformation algorithms. In this example

$$N = \{v_s, v_a, v_{a1}, v_{a2}, v_b, v_c, v_d, v_{d1}, v_{d2}, v_t\},$$
$$E = \{(v_s, next, v_a), (v_a, true, v_{a1}), (v_a, false, v_{a2}), \ldots, (v_{d2}, next, v_t)\},$$
$$\Lambda = \{s, a, a_1, a_2, b, c, d, d_1, d_2, t, next, true, false\},$$

v_s with label s is the starting node and v_t with label t is the terminal node.

Further, to ease identification, we will identify nodes by their labels when it is possible (when a bijection between nodes and node labels exists).

According to the above definitions the flow graph shown in Figure 1 is connected and has 7 strongly connected components: $\{s\}$, $\{a\}$, $\{a_1\}$, $\{a_2\}$, $\{b, c, d, d_1\}$, $\{d_2\}$, $\{t\}$.

There are two most common types of nodes in a flow graph:

- The *functional type* (*function*) which represent some operations (semantically described by label $\lambda(n)$) to be carried out on an object $v \in N$ (nodes a_1, a_2, b, c, d_1, d_2 in our example).
- The *predicative type* (*predicate*) which do not operate on an object but decide on the next operation to be carried out, according to whether or not a certain property of $v \in N$ holds (nodes a and d in our example).

There are two usual representations of node types in flow graphs: functions are represented by rectangular boxes and predicates are represented by diamond-shaped boxes. But because of our graph tool features we can use only rectangular boxes for our representations.

Therefore in this paper we distinguish functional and predicative node types by count of their leaving edges as follows:

- the functional box can has just only one leaving edge (with *next* label for our example in Figure 1) and
- the predicative box can has just only two leaving edges (with *true* and *false* labels for our example in Figure 1).

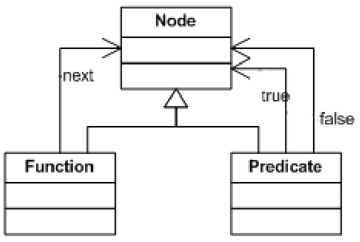


Figure 2: *The types in the flow diagram*

The different node types that are supported by flow graphs, together with their relationships, are shown in Figure 2, where we appeal to the reader's intuition about the meanings of this graph.

Let $\Phi = (G, s, t)$ be a flow graph and p be a some path in G from the start node s to the terminal node t. Then we will say that p is a *full path* in the flow graph Φ.

Definition 6 (execution sequence) Let p be a full path in a flow graph $\Phi = (G, s, t)$. Then an *execution sequence* $\lambda(p)$ is the word representation of p:

$$\lambda(p) = \lambda(s)\lambda(e_0)\lambda(v_1)\lambda(e_1)\lambda(v_2) \ldots \lambda(v_k)\lambda(e_k)\lambda(t).$$

For instance, sequences (s *next* a *true* a_1 *next* b *next* d *false* d_2 *next* t) and (s *next* a *false* a_2 *next* c *next* b *next* d *true* d_1 *next* c *next* b *next* d *false* d_2 *next* t) are two execution sequences for our example in Figure 1.

Let Path be a set (may be infinite) of all possible full paths in the flow graph $\Phi = (G, s, t)$. The word representation of Path thus regarded as a *formal language* (or just *language*) Lang = λ(Path) defined over the alphabet Λ by flow graph Φ.

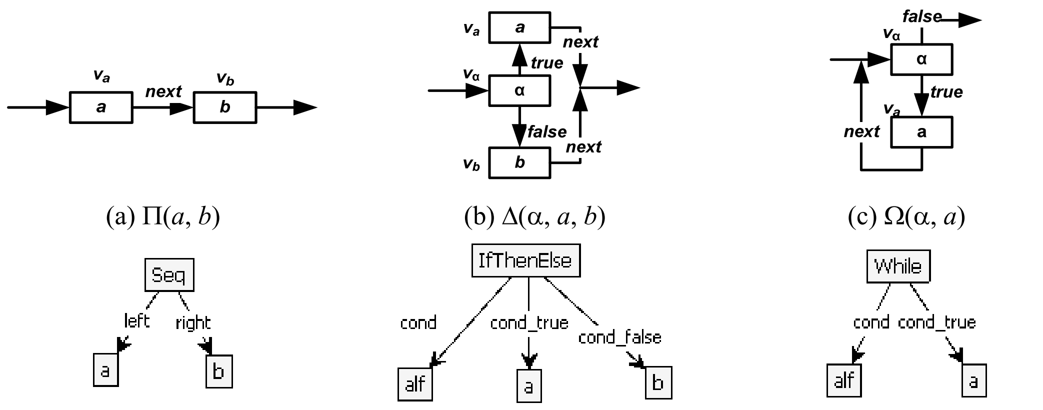
Definition 7 (equivalence of flow graphs) Two flow graphs Φ_1 and Φ_2 are *equivalent* (denote it as $\Phi_1 \sim \Phi_2$) if they define the same languages.

For instance, flow graphs Ξ and Ξ' shown in Figure 5 (a) and Figure 5 (b) respectively are equivalent. Indeed, they define the same languages Lang(Ξ) = Lang(Ξ') = $\{a$ *next* α (*false* a *next* α)* *true*$\}$ where "*" denotes matching of the preceding element zero or more times.

Algebra of flow diagrams. For our subsequent definitions we also need the formal concept of an algebra, which we likewise repeat here, for the sake of completeness and to introduce the notations [EEP06].

Definition 8 (signatures and algebras) An *algebraic signature*, or *signature* for short, is a tuple `Sig` = (*Sort*, *Oper*, *par*) where

- *Sort* is a set of sorts;
- *Oper* is a set of operation symbols;
- *par*: *Oper* $\rightarrow$ *Sort*$^+$ is a mapping that associates to every operation $op \in$ *Oper* a non-empty string of sorts $par(op) = \varphi_0 \ldots \varphi_n$ for $n \geq 0$, of which the elements $\varphi_0 \ldots \varphi_{n-1}$ constitute the parameter sorts of the operation, and φ_n constitutes the result



(a) $\Pi(a, b)$ (b) $\Delta(\alpha, a, b)$ (c) $\Omega(\alpha, a)$

(d) `Seq` construction (e) `IfThenElse` construction (f) `While` construction

Figure 3: *Diagrams of* $\Pi(a, b)$, $\Delta(\alpha, a, b)$, $\Omega(\alpha, a)$ *and respective syntax tree constructions*

sort.

A `Sig`-*algebra* is a tuple $A = (Data, Func)$ where

- *Data* is a set of disjoint *carrier sets*, one per sort $\varphi \in Sort$, denoted D_φ;
- *Func* is a set of *functions*, one per operation $op \in Oper$, denoted f_{op};
- The type of the functions in *Func* should be consistent with *par*, in the sense that $f_{op}: D_{\varphi_0} \times \ldots \times D_{\varphi_{n-1}} \to D_{\varphi_n}$ whenever $par(op) = \varphi_0 \ldots \varphi_n$.

Note that an operation *op* with no parameters represents a constant value, which in a `Sig`-algebra is given by $f_{op}()$.

A flow diagram is a graphical representation of the flow graph which is suitable for representing programs, Turing machines, etc. Diagrams are usually composed of boxes connected by directed lines.

Following [BJ66], we can distinguish three elementary types of flow diagrams Π, Δ and Ω which denote, respectively, the diagrams of Figure 3 (a)-(c) and the constructions '*sequence*', '*if-then-else*' and '*while*' in programming languages. Let us call these tree elementary types of flow diagrams $\Gamma = \{\Pi, \Delta, \Omega\}$ *base subdiagrams*.

Let us assume a universe Θ of arbitrary flow graphs, a set $\theta_{func} \subset \Lambda$ of all functional node labels and a set $\theta_{pred} \subset \Lambda$ of all predicative node labels.

Definition 9 (flow graph substitution) Let $\Phi = (G, s, t)$ be an arbitrary flow graph where $G = (N, E, \lambda)$ and $N' = N \setminus \{s, t\}$. A *flow graph substitution* is a mapping `Sub`: $N' \to \Theta$ that maps each node $v \in N'$ to a flow graph $\Phi_v = (G_v, s_v, t_v)$ where $G_v = (N_v, E_v, \lambda_v)$, and obeys the following rules:

- $\Phi[\Phi_v / v] = (G_{\text{Sub}}, s_{\text{Sub}}, t_{\text{Sub}})$ is a flow graph, $G_{\text{Sub}} = (N_{\text{Sub}}, E_{\text{Sub}}, \lambda_{\text{Sub}})$;
- $s_{\text{Sub}} = s$ and $t_{\text{Sub}} = t$;
- $N_{\text{Sub}} = (N \setminus v) \cup (N_v \setminus \{s_v, t_v\})$;
- $E_{\text{Sub}} = (E \setminus E^{\text{Del}}) \cup (E_v \setminus (E_v^{\text{Del}}) \cup (E_s^{\text{Ins}} \cup E_t^{\text{Ins}})$ where
    - $E^{\text{Del}} = \{e \in E: src(e) = v \text{ or } tgt(e) = v\}$,
    - $E_v^{\text{Del}} = \{e_v \in E_v: src(e_v) = s_v \text{ or } tgt(e_v) = t_v\}$,
    - $E_s^{\text{Ins}} = \{e_{\text{Sub}} \in E_{\text{Sub}} \mid \exists\, e_v \in E_v: src(e_v) = s_v, tgt(e_v) = tgt(e_{\text{Sub}}), \lambda(e_v) = \lambda(e_{\text{Sub}});$
      $\exists\, e \in E: src(e) = src(e_{\text{Sub}}), tgt(e) = v\}$,

- $E_t^{\mathrm{Ins}} = \{e_{\mathrm{Sub}} \in E_{\mathrm{Sub}} \mid \exists\, e_v \in E_v\colon src(e_v) = src(e_{\mathrm{Sub}}),\ tgt(e_v) = t_v,\ \lambda(e_v) = \lambda(e_{\mathrm{Sub}});$
  $\exists\, e \in E\colon src(e) = v,\ tgt(e) = tgt(e_{\mathrm{Sub}})\}$.

A substitution `Sub` can be extended to the whole flow graph as

$$\Phi[\mathtt{Sub}] = \Phi\ [\Phi_{v_1} / v_1]\ [\Phi_{v_2} / v_2]\ \ldots\ [\Phi_{v_n} / v_n].$$

Let us define the signature `Sig` = (*Sort*, *Oper*, *par*) for the flow graphs. We have a sort *fg*, representing the arbitrary flow graphs, a sort *pred*, representing the predicative nodes, and a sort *func*, representing the functional nodes. We also define a constant *empty* for the empty flow graph and operation symbols for the elementary flow graphs (for each functional node) and our base subdiagrams $\Gamma = \{\Pi, \Delta, \Omega\}$:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathtt{Sig} =} \\
\textit{Sort}: & fg, pred, func; \\
\textit{Oper}: & empty, elem, \Pi, \Delta, \Omega; \\
\textit{par}: & empty\colon \to fg, \\
 & elem\colon func \to fg, \\
 & \Pi\colon fg\ fg \to fg, \\
 & \Delta\colon pred\ fg\ fg \to fg, \\
 & \Omega\colon pred\ fg \to fg.
\end{array}$$

Then the implementation of the signature `Sig` for flow graphs is the following algebra `FlowGraph`:

$$\begin{array}{ll}
D_{fg} & = \Theta, \\
D_{func} & = \theta_{func}, \\
D_{pred} & = \theta_{pred}, \\
f_{empty} & = \varepsilon \in \Theta, \\
f_{elem} & : D_{func} \to D_{fg}, \\
 & \quad a \mapsto \{(N, E, \lambda) \mid N = \{s, v, t\}, E = \{(s, l, v), (v, l, t)\}, \lambda(v) = a\} \\
f_{\Pi} & : D_{fg} \times D_{fg} \to D_{fg}, \\
 & \quad (\Phi_a, \Phi_b) \mapsto \Pi[\Phi_a / v_a][\Phi_b / v_b] \\
f_{\Delta} & : D_{pred} \times D_{fg} \times D_{fg} \to D_{fg}, \\
 & \quad (\alpha, \Phi_a, \Phi_b) \mapsto \Delta[\Phi_a / v_a][\Phi_b / v_b] \\
f_{\Omega} & : D_{pred} \times D_{fg} \to D_{fg}, \\
 & \quad (\alpha, \Phi_a) \mapsto \Omega[\Phi_a / v_a].
\end{array}$$

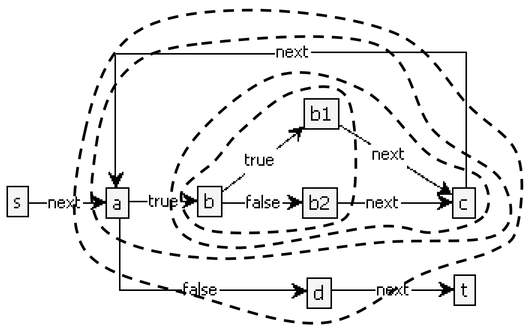
Definition 10 (strong decomposition or well-formedness) A flow diagram $\Phi = (G, s, t)$ where $G = (N, E, \lambda)$ is *strongly decomposable* (or *well-formed* in terms of [PKT73] and [EKR08]) if there exists an expression *exp* in the `Sig`-algebra `FlowGraph` such that `FlowGraph`$[\![exp]\!] \cong \Phi$.

An example of strong decomposition is shown in Figure 4 (a) where the base subdiagrams are isolated with dashed lines; it can be expressed as follows:

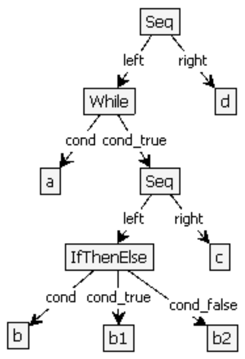
$$\Phi \cong f_{\Pi}(\,f_{\Omega}(a, f_{\Pi}(\,f_{\Delta}(b, f_{elem}(b_1), f_{elem}(b_2)), f_{elem}(c))), f_{elem}(d)).$$

The opposite example of a flow diagram which can not be strongly decomposed is our common example in Figure 1.

Together with a strong decomposition, [BJ66] considered another decomposition which is obtained by operating on an *equivalent* strongly decomposable flow graph.



(a) Strongly decomposable graph



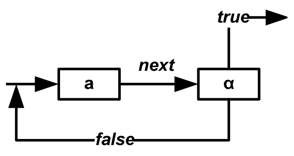
(b) Syntax tree

Figure 4: *Strongly decomposable graph* Φ *and respective syntax tree*

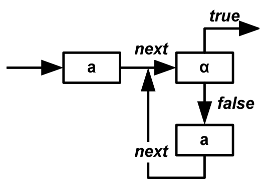
Formally, a flow graph Φ is *weakly decomposable* if $\Phi \sim \Phi'$ for some strongly decomposable flow graph Φ'.

As example of weak decomposition we introduce $\Xi(\alpha, a)$ denote construction '*repeat*' in programming languages and the diagram of Figure 5 (a) which cannot be strongly decomposed according to base subdiagrams. But we can define the equivalent strongly decomposable flow diagram $\Xi'(\alpha, a)$ denoted by the diagram of Figure 5 (b) as follows:

$$\Xi(\alpha, a) \sim \Xi'(\alpha, a) \cong f_{\Pi}(f_{elem}(a), f_{\Omega}(\alpha, f_{elem}(a))).$$



(a) $\Xi(\alpha, a)$



(b) $\Xi'(\alpha, a) \cong f_{\Pi}(f_{elem}(a), f_{\Omega}(\alpha, f_{elem}(a)))$

Figure 5: *Diagram of* $\Xi(\alpha, a)$ *and diagram equivalent to* $\Xi(\alpha, a)$

Algebra of syntax trees. The other data structure for representing programming language constructs by compilers, converters and transformation tools is a tree structure known as an *abstract syntax tree*. At large, an abstract syntax tree is a data structure consisting of types that represent language constructs connected by sequence and unit valued relationships to other types [OMG05].

In terms of graph theory, an abstract syntax tree is a tree, that is to say, an acyclic graph with a single root node, connecting nodes and leaf nodes. Then, similarly to the graph definition above, we can define a syntax tree as follows.

Definition 11 (syntax tree) An *abstract syntax tree*, or just *syntax tree*, is a tuple $T = (G_T, root)$ where

- $G_T = (N_T, E_T, \lambda_T)$ is an acyclic connected labeled directed graph;
- $root \in N_T$ is a single root node;
- $N_T = N_n \cup N_l$ such as $N_n \cap N_l = \varnothing$ where N_n is a set of *internal nodes* and N_l is a set of *leaf nodes*.

Each node of the syntax tree in our case should denote a construction occurring in the flow diagram. For instance, the base subdiagrams in Figure 3 (a)-(c) may be denoted by

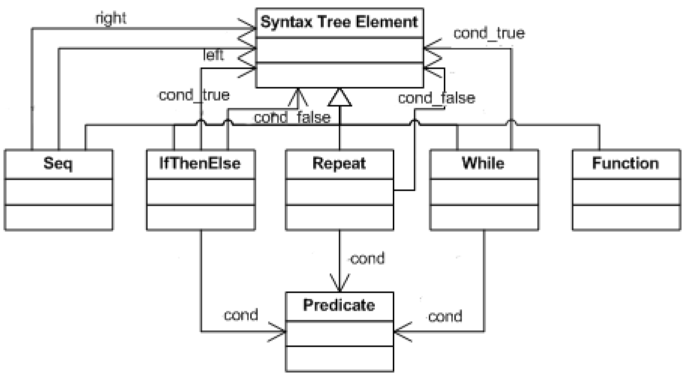


Figure 6: *The types in the syntax trees for our implementation*

constructions `Seq`, `IfThenElse` and `While` in Figure 3 (d)-(f), respectively. The different node types that are supported by syntax trees, together with their relationships, are shown in Figure 6.

Let us assume a universe Ψ of arbitrary syntax trees and a set ϑ of syntax tree constructions {`Seq`, `IfThenElse`, `While`}.

Definition 12 (syntax tree substitution) Let $T = (G_T, root)$ be an arbitrary syntax tree where $G_T = (N_T, E_T, \lambda_T)$ and $N_T = N_n \cup N_l$. A *syntax tree substitution* is a mapping Sub: $N_T \rightarrow \Psi$ that maps each leaf node $v \in N_l$, representing functional nodes in the flow graph, to a syntax tree $T_v = (G_{T_v}, root_{T_v})$ where $G_{T_v} = (N_{T_v}, E_{T_v}, \lambda_{T_v})$, and obeys the following rules:

- $T\,[T_v / v] = (G_{\texttt{Sub}}, root_{\texttt{Sub}})$ is a flow graph, $G_{\texttt{Sub}} = (N_{\texttt{Sub}}, E_{\texttt{Sub}}, \lambda_{\texttt{Sub}})$;
- $root_{\texttt{Sub}} = root$;
- $N_{\texttt{Sub}} = (N_T \setminus v) \cup N_{T_v}$;
- $E_{\texttt{Sub}} = (E_T \setminus E_T^{\text{Del}}) \cup E_{T_v} \cup E^{\text{Ins}}$ where
  - $E_T^{\text{Del}} = \{e \in E_T\colon tgt(e) = v\}$,
  - $E^{\text{Ins}} = \{e_{\texttt{Sub}} \in E_{\texttt{Sub}} \mid \exists\, e \in E_T^{\text{Del}}\colon src(e) = src(e_{\texttt{Sub}}),\ \lambda(e) = \lambda(e_{\texttt{Sub}});$ $tgt(e_{\texttt{Sub}}) = root_{T_v}\}$.

A substitution `Sub` can be extended to the whole syntax tree as

$$T\,[\texttt{Sub}] = T\,[T_{v_1} / v_1]\,[T_{v_2} / v_2] \ldots [T_{v_n} / v_n].$$

Note that a syntax tree is *empty* if $N = \varnothing$ and $E = \varnothing$. A syntax tree is an *elementary a-labelled* syntax tree if $N = \{v\}$, $root = v$, $\lambda(v) = a$ and $E = \varnothing$.

Since a signature `Sig` gives us only the syntax, we can implement it with a different algebra SyntaxTree, which describes operations on syntax trees as follows:

$$\begin{aligned}
D_{T_{fg}} &= \Psi, \\
D_{T_{func}} &= \theta_{func}, \\
D_{T_{pred}} &= \theta_{pred}, \\
g_{empty} &= \varepsilon \in \Psi, \\
g_{elem} &: D_{T_{func}} \rightarrow D_{T_{fg}}, \\
&\quad a \mapsto (N, E, \lambda) \mid N = \{v\},\ \lambda(v) = a,\ E = \varnothing \\
g_{\sqcap} &: D_{T_{fg}} \times D_{T_{fg}} \rightarrow D_{T_{fg}}, \\
&\quad (T_a, T_b) \mapsto \texttt{Seq}[T_a / v_a][T_b / v_b]
\end{aligned}$$

$$
\begin{aligned}
g_\Delta \quad &: D_{T_{pred}} \times D_{T_{fg}} \times D_{T_{fg}} \to D_{T_{fg}}, \\
&\quad (\alpha, T_a, T_b) \mapsto \texttt{IfThenElse}[T_a / v_a][T_b / v_b] \\
g_\Omega \quad &: D_{T_{pred}} \times D_{T_{fg}} \to D_{T_{fg}}, \\
&\quad (\alpha, T_a) \mapsto \texttt{While}[T_a / v_a].
\end{aligned}
$$

Let us consider now a representation of a flow graph Φ as a syntax tree T, called a *syntax tree decomposition*.

Definition 13 (syntax tree decomposition) A *syntax tree decomposition* of a weakly decomposable flow graph $\Phi = (G, s, t)$ is a following morphism:

$$STD: \Phi \mapsto \{\texttt{SyntaxTree}[\![exp]\!] \mid \texttt{FlowGraph}[\![exp]\!] \cong \Phi' \sim \Phi\}$$

where $\Phi' = (G', s, t)$ is an strongly decomposable flow graph equivalent to Φ.

An example of a syntax tree decomposing the flow graph Φ from Figure 4 (a) is shown in Figure 4 (b).

Chapter 3.
Flow Diagram Decomposition

There exist today many variants on the idea of a structured program. One of the first approaches to restructuring was given by Böhm and Jacopini [BJ66]. Their restructuring method was done in the context of flow diagram decomposition. Some researchers mentioned (see, for instance, [Amm92], [EH94]) that this result is mostly of historical and theoretical interest and does not give a complete algorithm. On the contrary, we believe that their method presents a sufficient set of pattern matching rules and transformations for implementation in terms of graph transformations. We discuss details of this approach in Section 3.1 and present some results of the implementation in Groove in Chapter 4.

There have been several other approaches to restructuring program flow diagrams. Peterson *et al.* present a proof that every flow diagram can be transformed into an equivalent strongly decomposable (well-formed) flow diagram [PKT73]. They present a graph algorithm to do such a transformation using a technique of node-splitting and they proved that this transformation is correct. We found this algorithm very useful to improve the quality of flowcharts, especially a process of eliminating multiple entries in the strongly connected components. Application of this algorithm as a part of Böhm-Jacopini approach is discussed in Section 3.2. The complexity measure to evaluate the advantage of the Peterson method is considered in Section 3.3 and some concrete implementation results are presented in Chapter 4.

A concise review of many of other results developed in this field has been prepared in [EH94]. We also come back to that discussion in the closing remarks about future work in Chapter 5.

3.1. Base subdiagram decomposition

The set of definitions introduced in the previous section is within the scope of the existing graph theory. In this section, we introduce a way to enrich the usual definitions, and so formalize the concepts of flow graph decomposition.

The preliminaries of Böhm-Jacopini method [BJ66] were presented in Chapter 2. In addition to three base subdiagrams Π, Ω and Δ, they introduced three new functions denoted by *T*, *F*, *K*, and a new predicate ω which define a behavior of auxiliary boolean variables set.

The effect of the first two functions *T* and *F* is to create a new boolean variable with value *true* or *false*, respectively, and the function *K* deletes the last boolean variable. The predicate ω is verified or not according to whether the last boolean variable value is *true* or *false*; the value of the predicate ω is *true* iff the last boolean variable value is *true*

Recall that if `Path` is a set of all possible full paths in the flow graph Φ, then the word representation of `Path` can be regarded as a language `Lang`(Φ) defined (in the extended case) over the alphabet $\Lambda \cup \{T, F, K, \omega\}$. Let the node types and their relationships be as it shown in Figure 2.

Then we can define a '*satisfiability*' function `Sat`: `Lang`(Φ) → `Lang*`(Φ), where `Lang*`(Φ) = `Lang`(Φ) ∪ {ε}, as following: for all words $w = (x_1\ x_2 \dots x_i \dots x_j \dots x_n) \in$ `Lang`(Φ) where $x_k \in \Lambda \cup \{T, F, K, \omega\}$, $k \in [1, n]$

$$\mathtt{Sat}(w) = \begin{cases} \varepsilon & \text{if } \exists\, i, j \in [2, n-2], i < j: \\ & \quad x_i \in \{T, F\}; x_j = \omega; \\ & \quad x_{j+1} \in \{true, false\} \setminus \{\tilde{x}_i\} \text{ and} \\ & \quad \forall k \in [i+1, j-1]: x_k \notin \{T, F, K, \omega\}; \\ w & \text{otherwise} \end{cases} \quad \text{where } \tilde{x} = \begin{cases} true & \text{if } x = T; \\ false & \text{if } x = F; \\ x & \text{otherwise} \end{cases}.$$

Therefore the language `Sat`(`Lang`(Φ)) denotes a set of all full path word representations in the flow graph Φ that satisfy our definitions of new functions *T*, *F*, *K* and predicate ω.

Let us denote a function

`Restrict`: `Lang*`(Φ) → `Lang*`(Φ) \ {*T*, *F*, *K*, ω}

as following: for all words $w = (x_1\ x_2 \dots x_{i-1}\ x_i\ x_{i+1}\ x_{i+2} \dots x_n) \in$ `Lang*`(Φ) where $x_j \in \Lambda$, $j = 1, 2, \dots i-1, i+1, \dots, n$ and $x_i \in \{T, F, K, \omega\}$

`Restrict`$(w) = (x_1\ x_2 \dots x_{i-1}\ x_{i+2} \dots x_n)$.

Then a language $\overline{\mathtt{Lang}}(\Phi)$ = `Restrict`(`Sat`(`Lang`(Φ))) is a *restricted language* of the flow graph Φ over the alphabet Λ.Then we can extend the definition of flow graph equivalence.

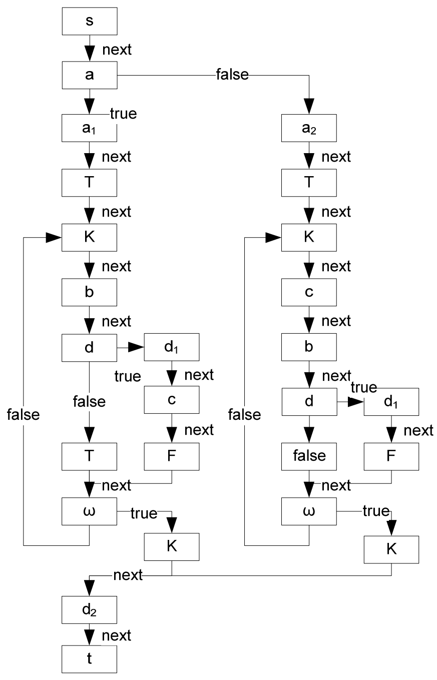
Then we can extend the definition of flow graph equivalence.

Definition 14 (equivalence of extended flow graphs) Two flow graphs Φ_1 and Φ_2 extended by functions *T*, *F*, *K* and predicate ω are *equivalent* if `Restrict`(`Lang`(Φ_1)) = `Restrict`(`Lang`(Φ_2)).

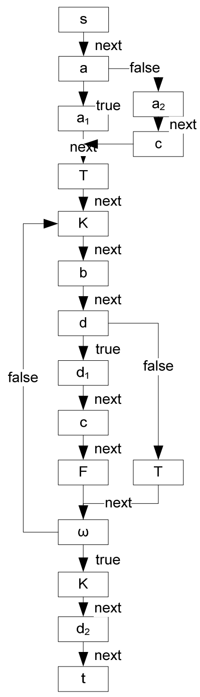
In the light of this discussion above the definition of weak decomposition can be extended as a decomposition which is obtained by operating on an equivalent strongly decomposable *extended* flow graph.

For instance, we can define tree words w_1, w_2, w_3 over the alphabet {*s*, *a*, a_1, a_2, *b*, *c*, *d*, d_1, d_2, *t*, *next*, *true*, *false*, *T*, *F*, *K*, ω} for the flow graph Φ in Figure 7 (a): w_1 = (*s next a true* a_1 *next T next K next b next d false T next* ω *true K next* d_2 *next t*), w_2 = (*s next a true* a_1 *next T next K next b next d true* d_1 *next c next F next* ω *true K next* d_2 *next t*) and w_3 = (*s next b next d false T next* ω *true* d_2 *next t*). Thereby we have two flow graph execution sequences $w_1, w_2 \in$ `Lang(Path)`, two words satisfied to our CF-grammar `Gram` $w_1, w_3 \in$ `Lang(Gram)` and only one word belongs to their intersection $w_1 \in$ `Lang`(Φ) = `Lang(Path)` ∩ `Lang(Gram)`. The reduction function application is follows: `Reduct`(w_1) = (*s next a true* a_1 *next b next d false* d_2 *next t*) that is the same as in example in Chapter 2. In the large, the flow graph in Figure 1 and two extended flow graphs in Figure 7 (a)-(b) are equivalent.

Theorem 1 (weak decomposition of flow graphs) *For any flow graph* Φ_1 *there is (at least) one equivalent strongly decomposable flow graph* Φ_2 *extended by the functions K, T, F and predicate* ω*; in other words, any flow graph is weakly decomposable.*



(a) Böhm-Jacopini decomposition



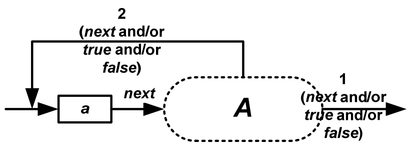
(b) Decomposition using SCC method

Figure 7: *Two strongly decomposable (well-formed) extended flow graphs equivalent to the flow graph in* Figure 1

Proof Sketch. The proof suggested in [BJ66] (and the decomposition algorithm proper) is based on the flow diagram classification represented in Figure 8 (a)-(c).

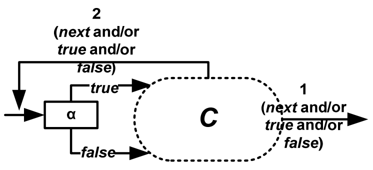
The main idea of this classification is to define a type of the flow diagram first element (functional or predicative) and a rest part of the diagram (inside the dashed lines in Figure 8 (a)-(c)) that is called A, B and C. The edges marked 1 and 2 denote an aggregated set of edges from nodes inside A, B and C structures to a first element or to a last element of the flow diagram, respectively. The edges 1 and 2 may not always both be present; nevertheless, from every A, B and C structures at least one edge 1 or 2 must start.

For instance, the flow graph $\Phi = \Pi(\Omega(a, \Pi(\Delta(b, b_1, b_2), c)), d)$ in Figure 4 (a) can be considered as a type II graph where a is a first predicative node, A is a subgraph of Φ with nodes $\{b, b_1, b_2, c\}$ and edges between them, $B = \{d\}$, a structure A has edge $(c, next, a)$ marked 2 and no edges marked 1, a structure B has edge $(d, next, t)$ marked 1 and no edges marked 2.



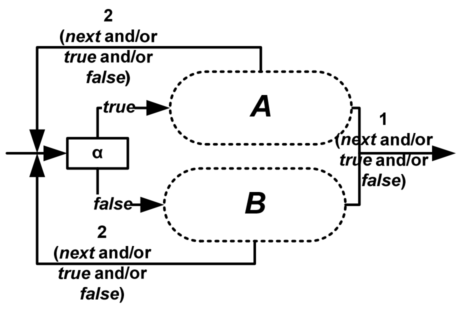
2
(*next* and/or *true* and/or *false*)
a
next
A
1
(*next* and/or *true* and/or *false*)

(a) Structure of a type I diagram



2
(*next* and/or *true* and/or *false*)
true
α
false
C
1
(*next* and/or *true* and/or *false*)

(c) Structure of a type III diagram



2
(*next* and/or *true* and/or *false*)
true
A
α
false
B
1
(*next* and/or *true* and/or *false*)
2
(*next* and/or *true* and/or *false*)

(b) Structure of a type II diagram

Figure 8: *Three types of flow diagrams*

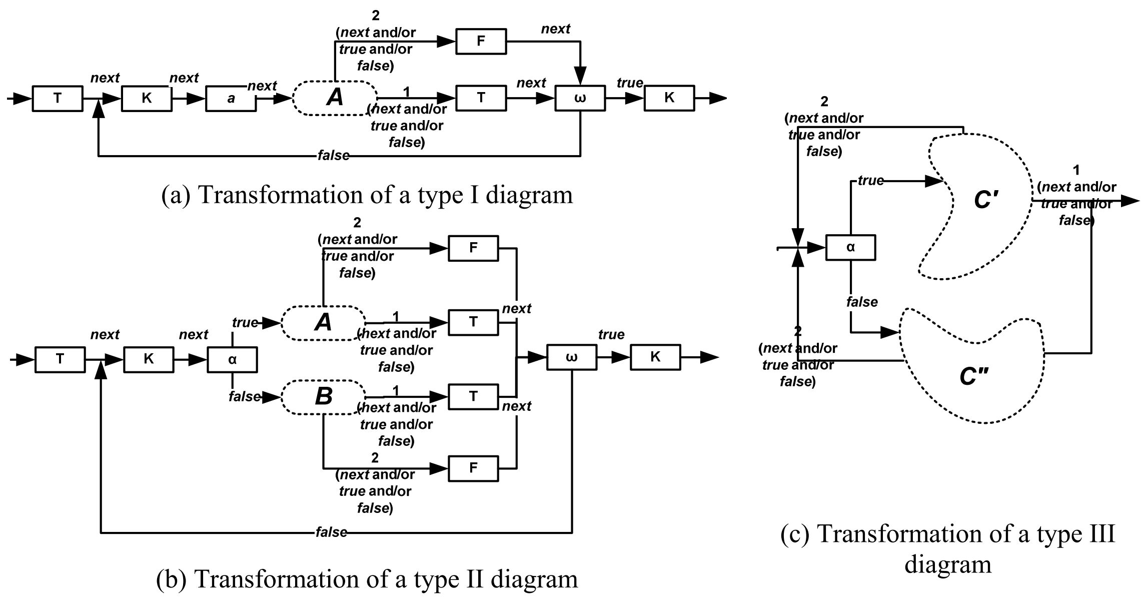

T
K
a
A
F
T
ω
K
next
true
false
1
2
(next and/or true and/or false)
α
B
C'
C"
(a) Transformation of a type I diagram
(b) Transformation of a type II diagram
(c) Transformation of a type III diagram

Figure 9: *Transformation of three types of flow diagrams to the equivalent extended strongly decomposable flow diagrams*

The equivalent strongly decomposed flow diagrams of type I and type II diagram are shown in Figure 9 (a)-(b). The case of the diagram of type III may be dealt with as case II by substituting Figure 9 (c), where C' indicates that subpart of C accessible from the upper entrance, and C'' that part accessible from the lower entrance.

Let Φ_1 be a flow graph of type I as it shown in Figure 8 (a). It is obvious that $\texttt{Reduct}(\texttt{Lang}(\Phi_1)) = \texttt{Lang}(\Phi_1) = \texttt{Lang}(\texttt{Path}_1) = \{(a\ next\ A\ (next|true|false)_2)^*\ a\ next\ A\ (next|true|false)_1\}$ where "*" denotes matching of the preceding element zero or more times and $(next|true|false)_k$, $k = 1{,}2$, denotes edges marked 1 or 2, respectively.

Let Φ_2 be a strongly decomposable flow graph extended by the functions K, T, F and predicate ω as it shown in Figure 9 (a). Then $\texttt{Lang}(\texttt{Path}_2) = \{T\ next\ K\ next\ a\ next\ A\ ((next|true|false)_1\ T\ next\ |\ (next|true|false)_2\ F\ next)\ \omega\ (false\ K\ next\ a\ next\ A\ ((next|true|false)_1\ T\ next\ |\ (next|true|false)_2\ F\ next))^*\ true\ K\ next\}$ and $\texttt{Lang}(\texttt{Gram}_2)$ is a CF-language generated by the CF-grammar $\texttt{Gram}$ over the alphabet $\{a, A, next, true, false, T, F, K, \omega\}$. Therefore $\texttt{Lang}(\Phi_2) = \texttt{Lang}(\texttt{Path}_2) \cap \texttt{Lang}(\texttt{Gram}_2) = \{T\ next\ (K\ next\ a\ next\ A\ (next|true|false)_2\ F\ next\ \omega\ false)^*\ K\ next\ a\ next\ A\ (next|true|false)_1\ T\ next\ \omega\ true\ K\ next\}$and $\texttt{Reduct}(\texttt{Lang}(\Phi_2)) = \{(a\ next\ A\ (next|true|false)_2)^*\ a\ next\ A\ (next|true|false)_1\} = \texttt{Reduct}(\texttt{Lang}(\Phi_1))$, so Φ_1 and Φ_2 are equivalent. Similarly, the same operations can be applied to a type II flow graph.

If it is assumed that A and B are, by inductive hypothesis, strongly decomposable, then the theorem is proved. □

Applying the Böhm-Jacopini decomposition to the example in Figure 1, the extended flow diagram shown in Figure 7 (a) is generated.

3.2. SCC Decomposition

Peterson *et al.* present the algorithm enabled to improve characteristics of Böhm-Jacopini method in case if flow graph consists of strongly connected components with multiple entry points [PKT73].

Theorem 2 (well-formed flow diagram) *Every flow diagram can be transformed into an equivalent strongly decomposable (well-formed) flow diagram by node duplication* (proof see [PKT73]).

In the proof of this theorem the authors presented the following algorithm that examines strongly connected components for multiple entry points and removes extra entry points by node duplication.

Suppose we have a flow diagram that contains a strongly connected component U with multiple entry nodes. Choose one, X, to become the unique entry node; the others, Y_1, Y_2, …, to be removed by node duplication. Next, introduce new nodes Y_1', Y_2', ... and remove any entry branches from Y_1, Y_2, … and connect them to Y_1', Y_2', ..., respectively. Now for each primed node Z', including the ones introduced in this step, if the original node Z connects to a node W outside U, place a branch representing the same processing from Z' to W. If Z connects to X, then connect Z' to X with a branch representing the same processing. If Z connects to any other node W of U, then make a new node W' if this hasn't already been done, and connect Z' to W' with a branch representing the same processing as the branch from Z to W.

It can be seen that the new flow diagram and the old one are equivalent, and the old flow diagram can be reconstructed if the corresponding primed and unprimed nodes are merged in the new one.

Let us come back to our main example in Figure 1 where nodes *b, c, d* and d_1 form a strongly connected component, and *b* and *c* are multiple entry nodes. If *b* is chosen as the entry node and *c* is duplicated, the well-structured flow diagram with the extended flow graph shown in Figure 7 (b) results. This turns out to be the better choice because this flow graph is intuitively 'better' than the flow graph in Figure 7 (a).

But if *c* is chosen as the entry node and *b* is duplicated, some more duplicating steps are necessary, and after four steps we can obtain the same flow graph as in Böhm-Jacopini method shown in Figure 7 (a), as well as three different flow graphs not shown here.

The fact that there are many variants of equivalent flow graphs, and some of them are 'better' than another, brings us to the issue of *complexity measuring* presented in the next section.

3.3. Complexity measuring

Maintenance typically required more resources than new software development. For years researchers have tried to understand how programmers comprehend programs. The literature provides two approaches to comprehension: cognitive models that emphasize cognition by what the program does (a functional approach) and a control-flow approach which emphasizes how the program works. A modern state of the art of this direction is reflected in the reviews [CV06], [WY96].

A well-known and often used measure was proposed by McCabe in [McC76].

Definition 15 (cyclomatic number) The *cyclomatic number* $v(\Phi)$ of a flow graph Φ with n nodes, e edges, and p connected components is

$$v(\Phi) = e - n + 2\,p.$$

McCabe suggested to measure the complexity of a program by computing the number of linearly independent paths $v(\Phi)$, control the "size" of programs by setting an upper limit to $v(\Phi)$ (instead of using just physical size), and use the cyclomatic complexity as the basis for a testing methodology.

In [Mil72] the following was proved: if the number of functions and predicates in a structured program is θ and π, respectively, and e is the number of edges, then $e = 1 + \theta + 3\pi$.

Since $n = \theta + 2\pi + 2$ and assuming $p = 1$, we get that the cyclomatic complexity of a structured program equals the number of predicates plus one:

$$v(\Phi) = \pi + 1.$$

In addition, McCabe proposed a method of measuring the "structuredness" of a program as follows.

Definition 16 (decomposition degree) Let Φ be a flow graph some subgraphs of which are strongly decomposed flow graphs $\Phi_1, \Phi_2, \ldots, \Phi_k$, and $\Phi_i = \Theta_i(\theta_{i\,1}, \theta_{i\,2}, \ldots, \theta_{i\,l})$ where (as in Definition 8) $\theta_{i\,j} = \theta_{i\,j}\,(x_1, \ldots, x_h, y, \theta_{i\,1}, \ldots \theta_{i\,l-1})$, $i = 1,2, \ldots, k, j = 1, 2, \ldots, l$; $x_q \in N$, $q = 1, 2, \ldots, h$ and $y \in \Gamma' = \{\Pi, \Delta, \Omega, \Xi\}$. Then a *decomposition degree* $m(\Phi)$ of a flow graph Φ is the number of $\theta_{i\,j}$ such that $y \in \Gamma' \setminus \{\Pi\}$.

Definition 17 (essential complexity) Let $m(\Phi)$ be a decomposition degree of a flow graph Φ. Then the following definition of *essential complexity* $v_e(\Phi)$ is used to reflect the lack of structure:

$$v_e(\Phi) = v(\Phi) - m(\Phi).$$

In the large, we propose to measure a full complexity of the flow diagram as follows:

Definition 18 (full complexity) Let $v(\Phi)$ be the cyclomatic number, $v_e(\Phi)$ - the essential complexity number and $v_d(\Phi)$ - the number of duplicated nodes in a flow graph Φ. Then the following defines the *full complexity* $V(\Phi)$:

$$V(\Phi) = [v(\Phi) + v_d(\Phi)] \times v_e(\Phi).$$

This formula stresses that the full complexity of a flow diagram is equal to the summation of its cyclomatic number and number of duplicates. The multiplication dictates that the full complexity and essential complexity of a flow diagram must be in the same order of magnitude.

Let us illustrate all of that complexity measuring by our main example shown in Figure 1. The initial flow diagram contains two predicates, therefore $v = 3$, $v_e = 3$, $v_d = 0$ and $V = (3 + 0) \times 3 = 9$. If we apply the straight Böhm-Jacopini method the final flow diagram shown in Figure 7 (a) has $v = 6$, $v_e = 1$, $v_d = 4$ and $V = (6 + 5) \times 1 = 11$. The 'best choice' of SCC method represented in Figure 7 (b) has $v = 4$, $v_e = 1$, $v_d = 1$ and $V = (4 + 1) \times 1 = 5$. Other four flow graphs obtained by SCC method have $V = 6$, $V = 11$, $V = 12$ and $V = 12$, respectively.

Hereby, the introduced full complexity measure V reflects an intuitive notion of readability and enables us to compare the final syntax trees and minimize their complexity.

Chapter 4.

Graph Transformations: Implementation within GROOVE

Graph transformation is a systematic, rule-based transformation technique. It has a solid research foundation [EEP06] and applications in many areas in computer science.

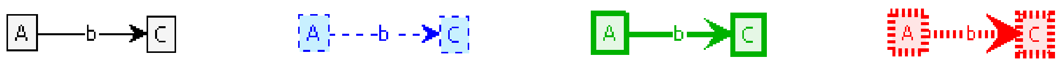
4.1. Graph transformations

A graph production system (GPS) is a set of *graph production rules*, each of which can transform a *source graph* into a new graph called the *target graph*. The rule specifies both the conditions under which it applies and the changes it makes to the source graph. Technically, a graph production rule consists of two partially overlapping graphs, a *left hand side* L and a *right hand side* R, and a set of *negative application conditions* N, which are also (connected) graphs partially overlapping with L. In order to apply the rule, the left hand side L is *matched* to (a part of) the source graph G, after which the image of L in G is replaced by a copy of R; but a matching is only valid if it cannot be extended to any of the graphs in N – in other words, the structure in the negative application conditions is forbidden in the source graph.

In our visual presentation of a rule used in this paper (which is taken from the Groove tools) we combine all these elements together into one graph, made up of four types of elements:

- *Readers*: elements present in both L and R. They have to be present in the source graph for L to match and are preserved in the target graph;
- *Erasers*: elements present in L but not in R. They are matched in the source graph but are not preserved in the target graph, i.e. they are removed.
- *Creators*: elements absent in L but present in R. They are introduced to the target graph.
- *Embargoes*: elements absent in L but present in one of the negative application conditions in N.

To distinguish these four types visually, each element has a distinct color and form, as shown in Figure 10: *readers* are black, *erasers* are dashed blue (darker gray in black-and-white presentations) *creators* are bold green (light gray in black-and-white presentations) and *embargoes* are bold, dashed red (dark gray in black-and-white presentations).



(a) Reader (b) Eraser (c) Creator (d) Embargo

Figure 10: *The graph production rule elements*

4.2. Implementation within GROOVE

We implemented techniques described in Chapter 3 within the Groove (see [Ren04], [EKR08], [KR08]) framework, a standard tool for graph transformations. This allowed a more thorough exploration of more examples and for a qualified judgment on practical scalability.

The flow diagram decomposition rules construct a syntax tree by contracting and transforming a flow diagram. In this transformation process, syntax tree elements are introduced to the flow diagram and flow diagram elements are contracted (iteratively) to one node. Our flow diagram decomposition approach consists of following issues.

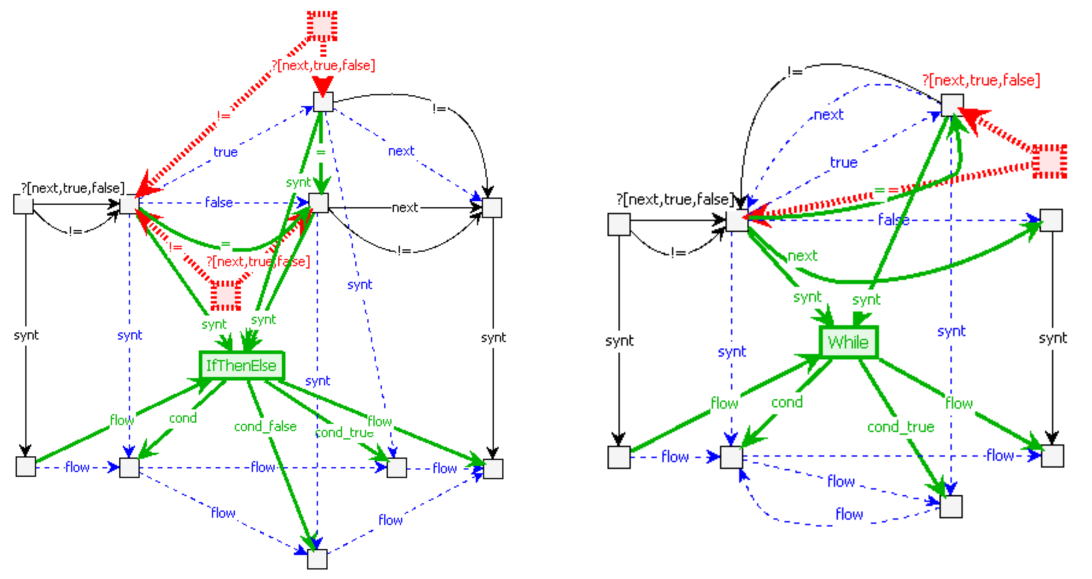
Flow diagram and syntax trees. On the first step of our transformations we copy the initial flow diagram Φ to create the same structure for the syntax tree T such that:

- every node $v \in \Phi$ has a node image $Im(v) \in T$ with the same node label;
- every edge $(v_1, v_2) \in \Phi$ has an edge image $Im(v_1, v_2) \in T$ in the syntax tree with the edge label *flow*;
- every node $v \in \Phi$ has to be connected with its image $Im(v) \in T$ by edge $(v, Im(v))$ with the edge label *synt*;
- for each rule we create and support a connection between flow diagram elements and syntax tree elements by edges with the label *synt*.

Contraction rules. For each type of elementary flow diagrams Π, Ω, Δ and Ξ, we design one flow diagram *contraction rule* that introduce the necessary syntax tree elements and contracts elementary flow diagram to one node. Two examples of flow diagram contraction rules for the *IfThenElse* (Δ) and *While* (Ω) statements are shown in Figure 11 (a)-(b) (compare to Figure 3 (b)-(c) and then Figure 3 (e)-(f), respectively).

The notation for contraction rules in Figure 11 is follows:

- the node types are used in compliance with the notation shown in Figure 10;



(a) Contraction rule for *IfThenElse* (Δ) statement (b) Contraction rule for *While* (Ω) statement

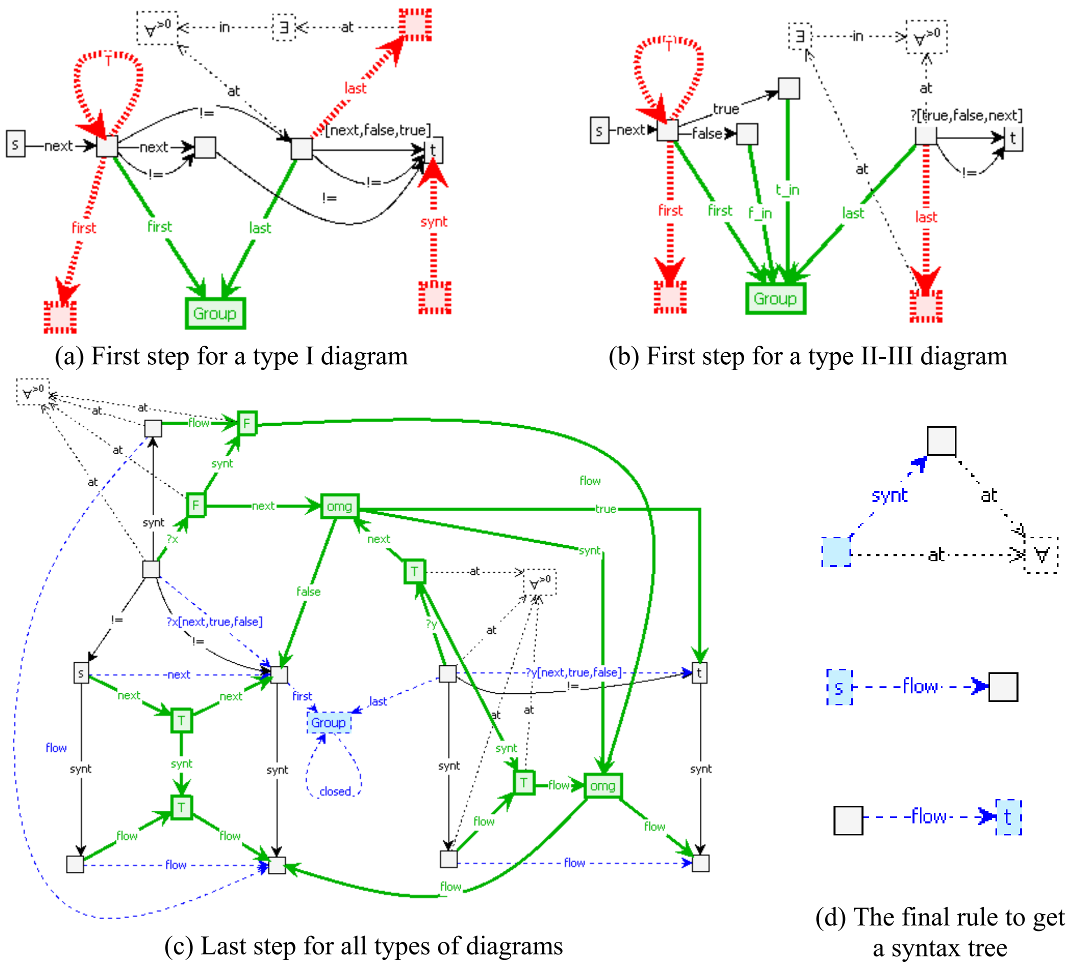
Figure 11: *The flow diagram contraction rules*

- to itemize all possible edge labels for this connection we are using separate labels or regular expressions such as "?[*next*, *true*, *false*]";
- to emphasize discrepancy of two nodes we are using an edge with the label "!=" between them;
- type diagram for the flow diagram is shown in Figure 2;
- type diagram for the syntax tree is shown in Figure 6.

Decomposition rules. The flow diagram *decomposition process* operates top-down, starting from the root-node of the flow diagram under construction and choosing an appropriate type of flow diagram (I, II or III) as was discussed in Section 3.1. Figure 12 (a)-(b) shows how a first step of a decomposition process for graph types I and II-III, respectively, is resolved (compare to Figure 8 (a)-(c)) and a general last step for all graph types is shown in Figure 12 (c) (compare to Figure 9 (a)-(c)).

The notation for decomposition rules in Figure 12 is follows:

- the node types are used in compliance with the notation shown in Figure 10;
- to itemize all possible edge labels for this connection we are using separate labels or regular expressions such as "?[*next*, *true*, *false*]";
- to emphasize discrepancy of two nodes we are using an edge with the label "!="



(a) First step for a type I diagram

(b) First step for a type II-III diagram

(c) Last step for all types of diagrams

(d) The final rule to get a syntax tree

Figure 12: *The flow diagram decomposition rules and the final rule to get a syntax tree*

between them;

- to define a logical behavior we are using the universal quantifier $\forall$ and existential quantifier $\exists$; the quantifier $\forall^{>0}$ restricts usability of universal quantifier to existing nodes;
- type diagram for the flow diagram is shown in Figure 2.

The flow diagram decomposition process shown in Figure 12 operates as following:

- on the first step of the decomposition process the new node `Group` is added and it should be connected with the first and last flow diagram elements by edges *first* and *last*, respectively (Figure 12 (a)); in the case of types II or III it also should be connected with two first elements in conditional branches *true* and *false* by edges *t_in* and *f_in*, respectively (Figure 12 (b));
- in the case of types II or III we have the iterative process to find all elements in conditional branches *true* and *false* and connect them with node `Group` by edges *t_in* and *f_in*, respectively;
- on the last step of the decomposition process the node duplication for nodes located in both conditional branches *true* and *false* and the extension of the flow diagram by special functions *T*, *F*, *K* and predicate *omg* is executing (Figure 12 (c)).

SCC rules. To improve readability of the flow diagrams, we also use *strongly connected component (SCC) decomposition rules* as it was discussed in Section 3.2.

Bottom-up and top-down decomposition. In general, the flow diagram contraction and decomposition process operates in both directions: while an extraction of elementary flow diagram is possible, we are applying one of contraction rules and have a *bottom-up* process; otherwise we are applying one of decomposition rules and have a *top-down* decomposition (the terms of top-down and bottom-up decomposition are used in compliance with formal language theory [AU73]).

Syntax trees. On the last step of our transformation we delete the contracted flow diagram elements and get a final syntax tree (see Figure 12 (d)). The type diagram for the syntax tree is shown in Figure 6.

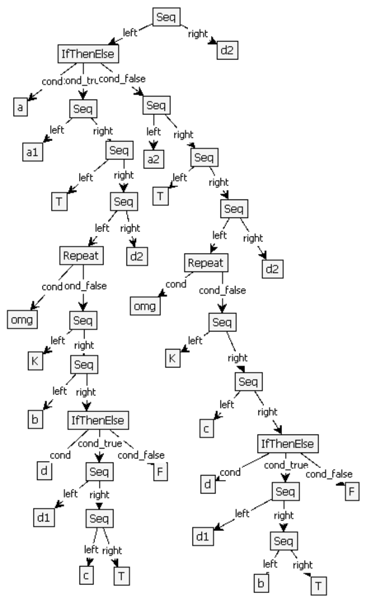
Unfortunately, we cannot explain the precise workings of the Groove implementation in the available space; however, the rules and some example cases are available in Appendix A for the reader to try out.

The example of final syntax tree for the straight Böhm-Jacopini method applying to the initial flow diagram in Figure 1 is shown in Figure 13 (a) and has $v = 6$, $v_e = 1$, $v_d = 4$ and $V = (6 + 5) \times 1 = 11$. The best of five final syntax trees corresponding to that initial diagram obtained by the nondeterministic SCC method (see Section 3.2) is shown in Figure 13 (b) and has $v = 4$, $v_e = 1$, $v_d = 1$ and $V = (4 + 1) \times 1 = 5$. The text code representation corresponding to the final syntax trees in Figure 13 (a)-(b) are presented in Figure 13 (c)-(d), respectively.

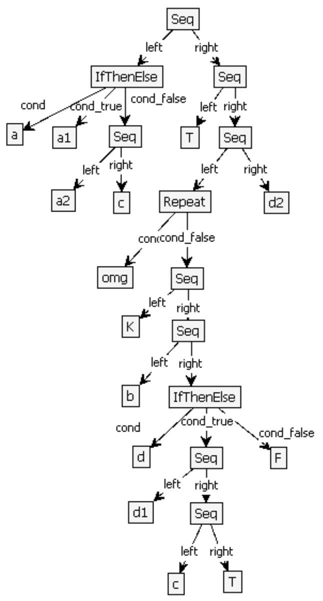
Some example results for the complexity measuring implementation are given in Table 1. From the table, we can observe that (as expected) the SCC method always yields results at least as good as, and in all larger cases better than, the Böhm-Jacopini method. The detailed description of examples is available in Appendix B.

Table 1. *Example cases for the complexity measuring implementation* (n is the number of nodes in the flow graph and V is the complexity measure proposed in the Section 3.3). The bold line (case #3) represents the example from Figure 1.

Case #	Initial flow graph		Böhm-Jacopini method (deterministic)		SCC method (non-deterministic)				
					Result count	Min V		Max V	
	n	V	n	V		n	V	n	V
1	8	3	8	3	1	8	3	8	3
2	9	9	12	4	1	12	4	12	4
3	**10**	**9**	**26**	**11**	**5**	**17**	**5**	**32**	**12**
4	14	36	38	18	12	25	11	63	29
5	50	156	82	64	52	71	32	82	64
6	100	276	237	154	72	112	84	289	312



(a) The final syntax tree with $V = 11$



(b) The final syntax tree with $V = 5$

```
begin
    if $a$ then begin
        $a_1$;
        $var\_bool$ := true;
        repeat
            $b$;
            if $d$ then begin
                $d_1$;  $c$;
                $var\_bool$ := false;
            end else
                $var\_bool$ := true;
        until $var\_bool$;
    end else begin
        $a_2$;
        $var\_bool$ := true;
        repeat
            $c$;  $b$;
            if $d$ then begin
                $d_1$;
                $var\_bool$ := false;
            end else
                var_bool := true;
        until $var\_bool$;
    end;
    $d_2$;
end.
```

(c) The text code representation of tree (a)

```
begin
    if $a$ then
        $a_1$;
    else begin
        $a_2$;
        $c$;
    end;
    $var\_bool$ := true;
    repeat
        $b$;
        if $d$ then begin
            $d_1$;
            $c$;
            var_bool := false;
        end else
            var_bool := true;
    until $var\_bool$;
    $d_2$;
end.
```

(d) The text code representation of tree (b)

Figure 13: *Two final syntax tree examples and text code representation*

Chapter 5.
Conclusions and Related Work

In this paper we take a first step towards an implementation of existing flow graph decomposition methods using graph transformations.

As stated in the introduction, well-structuredness was one of our main guidelines. We investigated several alternative and mutually complementary classical methods of flow diagram decomposition. We implemented the Böhm-Jacopini approach in terms of graph transformations employing the graph-transformation tool Groove. For the implementation we used an extended concept of equivalent flow graphs defined through the notion of context-free languages.

The Böhm-Jacopini decomposition method was enhanced and improved by using the Peterson *et al*. method that examines strongly connected components for multiple entry points and removes extra entry points by node duplicating.

In the introduction we stated that the well-structuredness of models is very important. Our full complexity measuring of a flow diagram reflects an intuitive notion of readability and enables us to compare the final syntax trees to evaluate different decomposition methods and different results of non-deterministic methods and minimize their complexity.

An important issue is to expand the set of implemented methods and apply them to improve software reliability and readability, for instance in model transformations from UMLA to Java programs. A concise review of many of other results developed in this field has been prepared in [EH94].

The described approach is still work in progress. The applying well-formed structures is just the first step in the general decomposition approach: the next step is to review the different cases of flow graphs with parallelism and loops and develop universal method similar simple flow graphs without parallelism.

In general, we intend to investigate the applicability of our framework to enhance a model transformation from UMLA to structured models and formally prove the correctness of this transformation. After enriching that model transformation, our long-term goal is to implement the same methods to transformations from UMLA to business process execution languages.

Acknowledgements: The research in this paper was carried out in the GRASLAND project, funded by the Dutch NWO (project number 612.063.408).

Bibliography

[AU73] Aho, A.V., and Ullman, J.D. The Theory of Parsing, Translation and Compiling. Prentice Hall, Englewood Cliffs, N.J., 1973.

[Aig97] Aigner, M. Combinatorial theory. Springer-Verlag Berlin Heidelberg, 1997.

[All70] Allen, F.E. Control Flow Analysis. *ACM Sigplan Notices*, 1970.

[Amm92] Ammarguellat, Z. A control-flow normalization algorithm and its complexity. *IEEE Transaction on software engineering*, Vol. 18, No. 3, pp. 237–251, 1992.

[AM71] Ashcroft, E., and Manna, Z. The translation of 'GOTO' programs to 'WHILE' programs. *Proceedings of IFIP Congress*, Ljubljana, Yugoslavia, pp. 250-255, 1971.

[Bak77] Baker, B. An algorithm for structuring flowgraphs. *Journal of the ACM*, Vol. 4, No. 1, pp. 98-120, 1977.

[Ber73] Berge, C. Graphs and Hypergraphs. Amsterdam, The Netherlands: North-Holland, 1973.

[BJ66] Bohm, C., and Jacopini, G. Flow diagrams, Turing machines and languages with only two formation rules. *Communications of ACM*, Vol. 9, No. 5, pp. 366-371, 1966.

[BS65] Busacker, R.G., and Saaty, T.L. Finite Graphs and Networks: An Introduction with Applications. *McGraw-Hill Book Co.*, New York, 1965.

[CV06] Collar, E., and Valerdi R. Role of Software Readability on Software Development Cost. *21st Forum on COCOMO and Software Cost Modeling*, Herndon, VA, 2006.

[EEP06] Ehrig, H., Ehrig, K., Prange, U., and Taentzer, G. Fundamentals of Algebraic Graph Transformation. Springer, 2006.

[EKR08] Engels, G., Kleppe, A.G., Rensink, A., *et. al.* From UML Activities to TAAL - Towards Behavior-Preserving Model Transformations. *Proceeding of the European Conference on Model Driven Architecture - Foundations and Applications* (*ECMDA-FA*). Lecture Notes in Computer Science 5095, Springer Verlag, Berlin, Germany, pp. 94-109, 2008.

[EH94] Erosa, A.M., and Hendren L.J. Taming control flow: A structured approach to eliminating goto statements. *Proceedings of the 1994 International Conference on Computer Languages*, Toulouse, France, pp 229–240, 1994.

[Har89] Harary, F. Graph Theory. Addison-Wesley, Canada, 1989.

[KR08] Kleppe, A.G., and Rensink, A. A Graph-Based Semantics for UML Class and Object Diagram. *Technical Report TR-CTIT-08-06 Centre for Telematics and Information Technology*, University of Twente, Enschede, 2008.

[LPW02] Linger, R., Pleszkoch, M., Walton, G., and Hevner, A. Flow-Service-Quality (FSQ) Engineering: Foundations for Network System Analysis and Development. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002.

[McC76] McCabe T. A Complexity Measure. *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, pp. 308-320, 1976.

[Mil72] Mills, H.D. Mathematical foundations for structured programming. *IBM Federal System Div.*, Gaithersburg, 1972.

[OMG05] Object Management Group. Abstract Syntax Tree Metamodel, *Request For*

Proposals (RFP). http://www.omg.org/cgi-bin/doc?admtf/05-02-02.pdf, 2005.

[PKT73] Peterson, W.W., Kasami, T., and Tokura, N. On the capabilities of while, repeat and exit statements. *Communications of ACM*, Vol. 16, No. 8, pp. 503-512, 1973.

[Ren04] Rensink, A.. The GROOVE Simulator: A Tool for State Space Generation. Pfaltz, J.L., Nagl, M., Bohlen, B. (eds.) *AGTIVE 2003*. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg, 2004.

[WY96] Woods, S., and Yang Q. The Program Understanding Problem: Analysis and a Heuristic Approach. *Proceedings of the 18th international conference on Software engineering* (*ICSE*), IEEE, Berlin, Germany, pp. 6 – 15, 1996.

[ZHB05] Zhao, W., Hauser, R., Bhattachaya, K., and Bryant B. Compiling Business Processes: Untangle Unstructured Loops in Irreducible Flow Graphs. Technical report UABCIS-TR-2005-0505-1, Department of Computer and Information Sciences, University of Alabama at Birmingham, 2005.

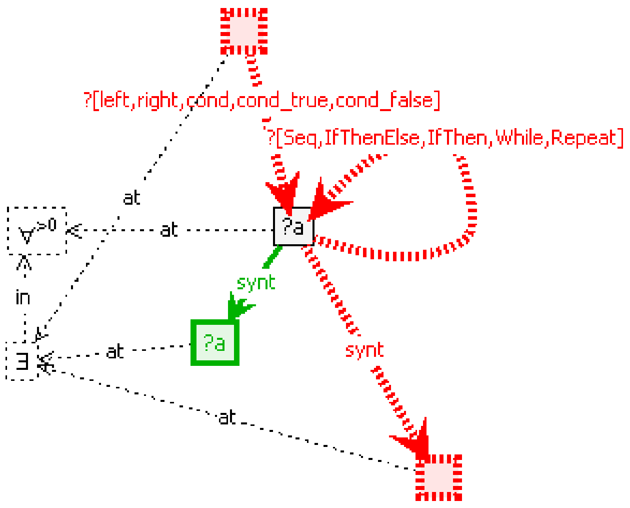
Appendix A.

GROOVE Rule Description

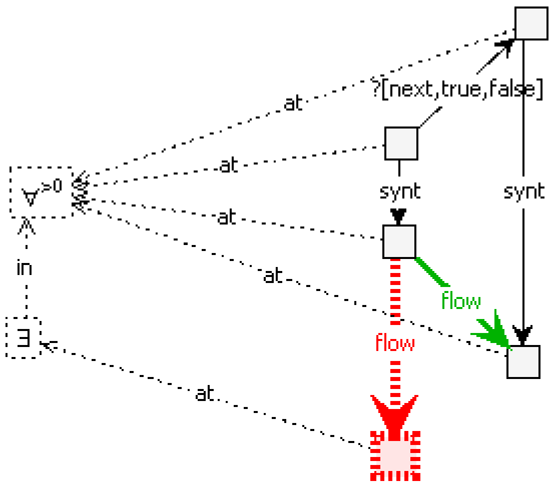
1. Flow diagram and syntax trees. On the first step of our transformations we copy the initial flow diagram Φ to create the same structure for the syntax tree T such that:

- every node $v \in \Phi$ has a node image $Im(v) \in T$ with the same node label;
- every edge $(v_1, v_2) \in \Phi$ has an edge image $Im(v_1, v_2) \in T$ in the syntax tree with the edge label *flow*;
- every node $v \in \Phi$ has to be connected with its image $Im(v) \in T$ by edge $(v, Im(v))$ with the edge label *synt*;
- for each rule we create and support a connection between flow diagram elements and syntax tree elements by edges with the label *synt*.

Priority 12. CopyNodes


?[left,right,cond,cond_true,cond_false]
?[Seq,IfThenElse,IfThen,While,Repeat]
at
∀>0
at
?a
in
synt
?a
∃
at
synt
at

Priority 11. CopyEdges



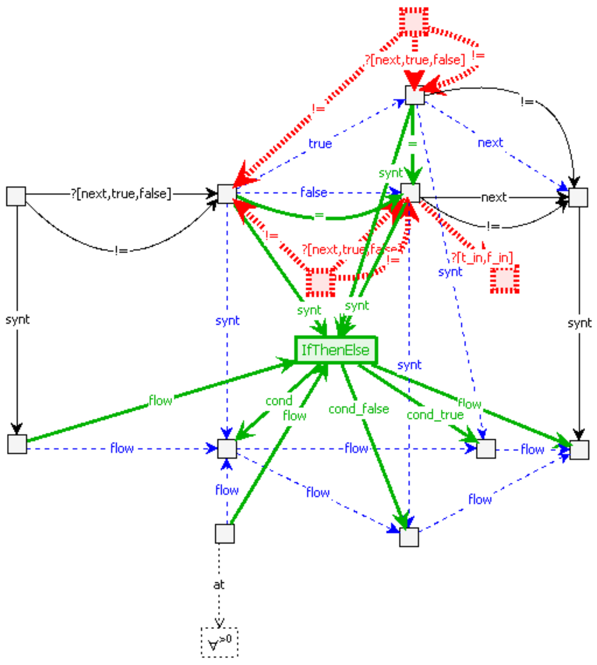
2. Contraction rules. For each type of elementary flow diagrams Π, Ω, Δ and Ξ, we design one flow diagram *contraction rule* that introduce the necessary syntax tree elements and contracts elementary flow diagram to one node.

The notation for contraction rules is follows:

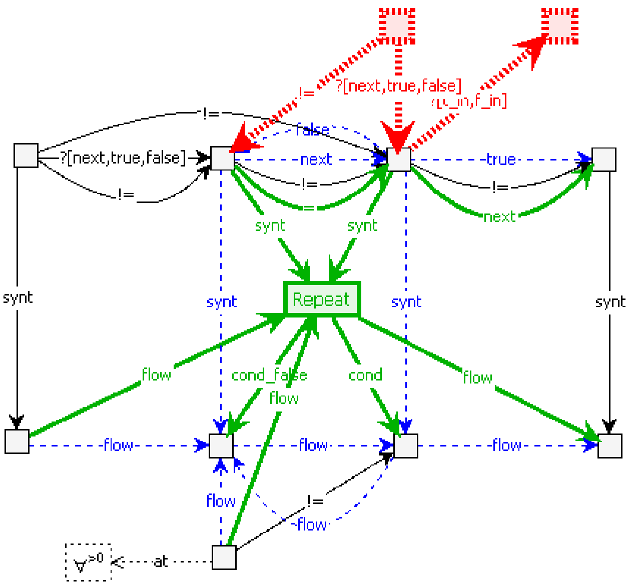
- the node types are used in compliance with the paper notation;
- to itemize all possible edge labels for this connection we are using separate labels or regular expressions such as "?[*next*, *true*, *false*]";
- to emphasize discrepancy of two nodes we are using an edge with the label "!=" between them.

Priority 10.

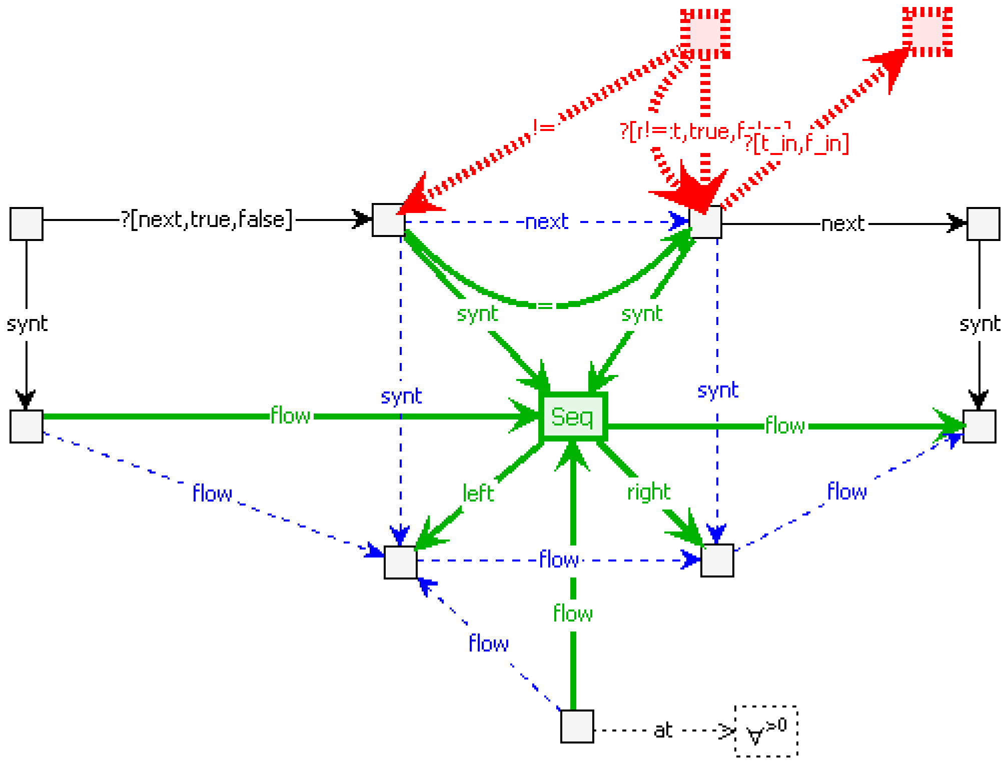
IfThenElseNodes



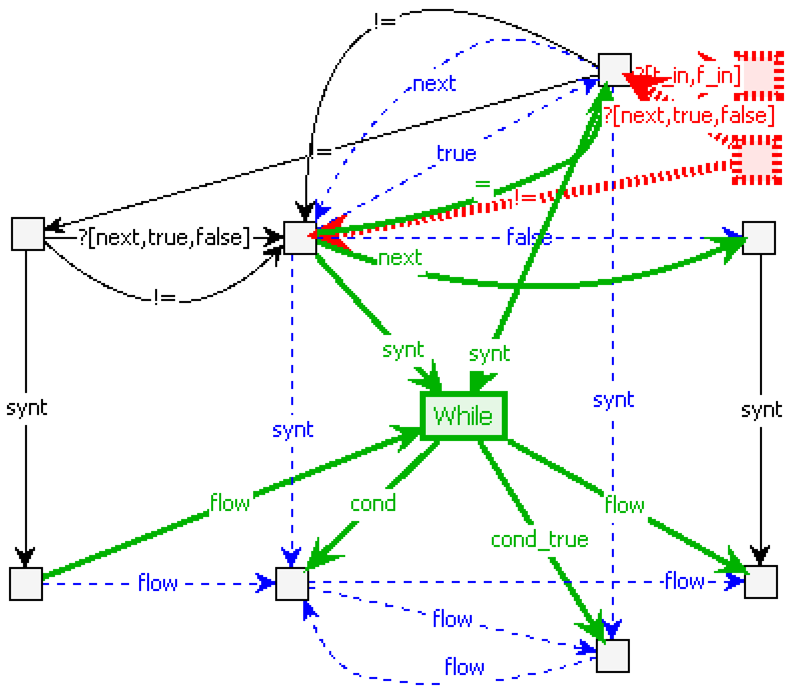
RepeatNodes



SequenceNodes



While Nodes



3. Decomposition rules. The flow diagram *decomposition process* operates top-down, starting from the root-node of the flow diagram under construction and choosing an appropriate type of flow diagram (I, II or III) as was discussed in the paper.

The notation for decomposition rules is follows:

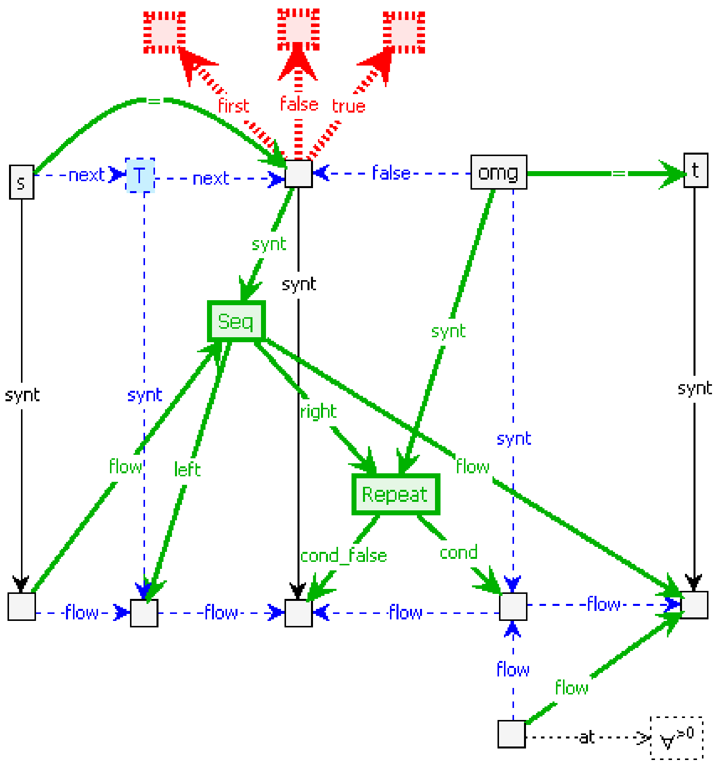
- the node types are used in compliance with the paper notation;
- to itemize all possible edge labels for this connection we are using separate labels or regular expressions such as "?[*next*, *true*, *false*]";
- to emphasize discrepancy of two nodes we are using an edge with the label "!=" between them;
- to define a logical behavior we are using the universal quantifier ∀ and existential quantifier ∃; the quantifier $\forall^{>0}$ restricts usability of universal quantifier to existing nodes.

The flow diagram decomposition process operates as following:

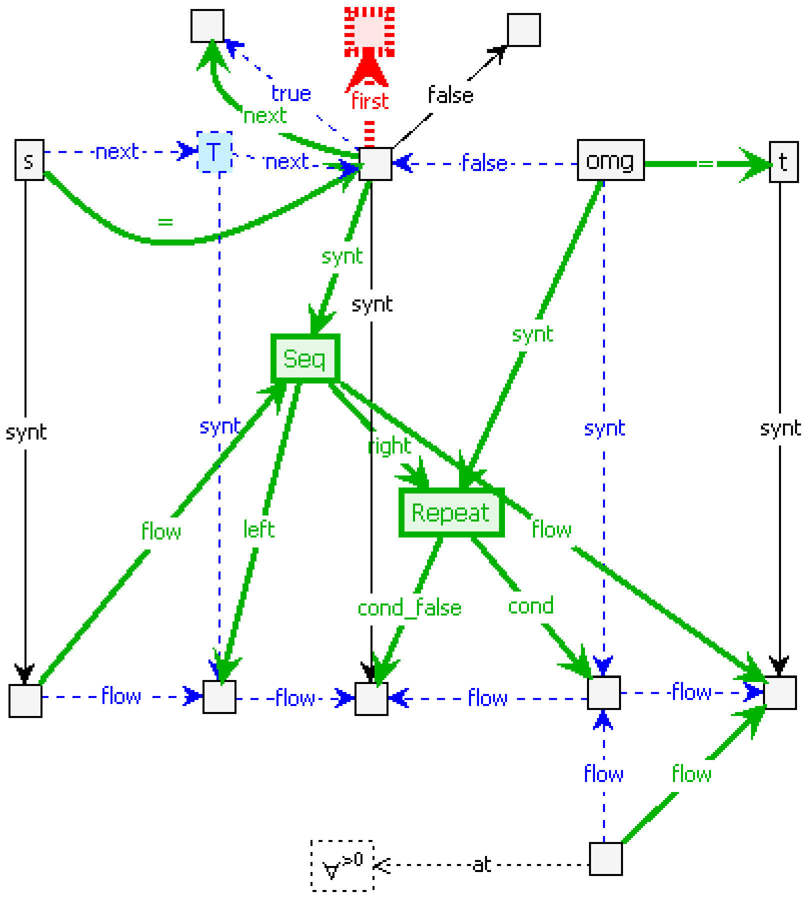
- on the first step of the decomposition process the new node `Group` is added and it should be connected with the first and last flow diagram elements by edges *first* and *last*, respectively; in the case of types II or III it also should be connected with two first elements in conditional branches *true* and *false* by edges *t_in* and *f_in*, respectively;
- in the case of types II or III we have the iterative process to find all elements in conditional branches *true* and *false* and connect them with node `Group` by edges *t_in* and *f_in*, respectively;
- on the last step of the decomposition process the node duplication for nodes located in both conditional branches *true* and *false* and the extension of the flow diagram by special functions *T*, *F*, *K* and predicate *omg* is executing.

Priority 9.

GroupType1-Normalization

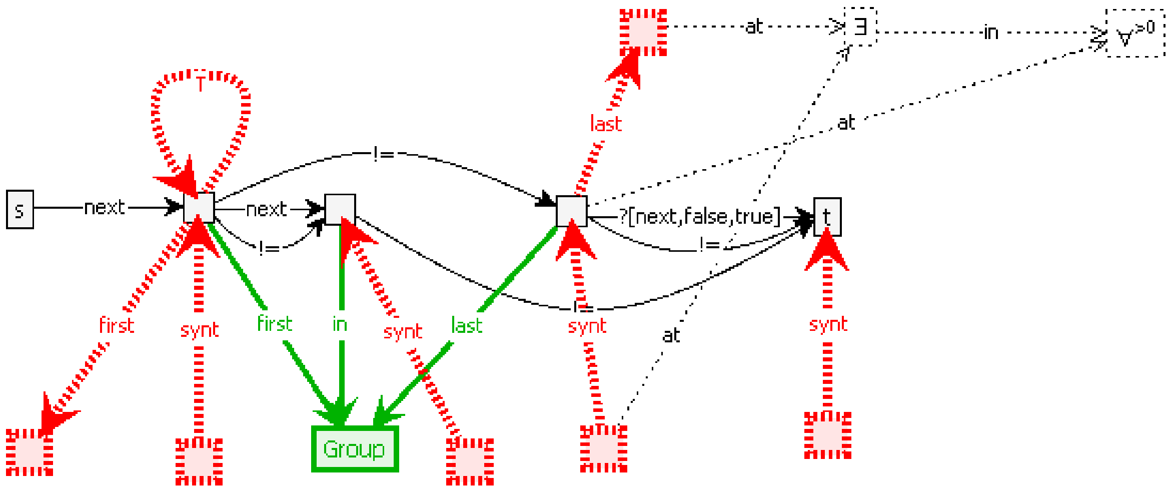


GroupType2-Normalization

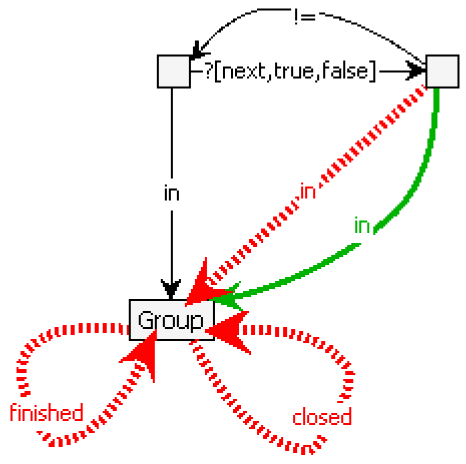


Priority 8.

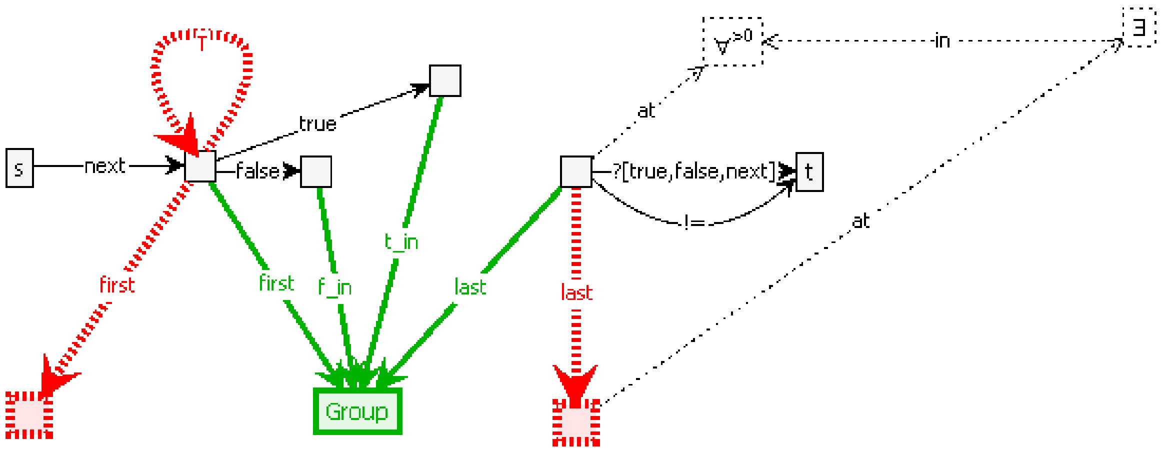
GroupType1-Step1



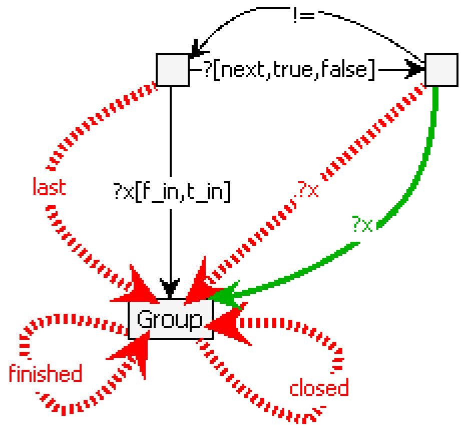
GroupType1-Step2



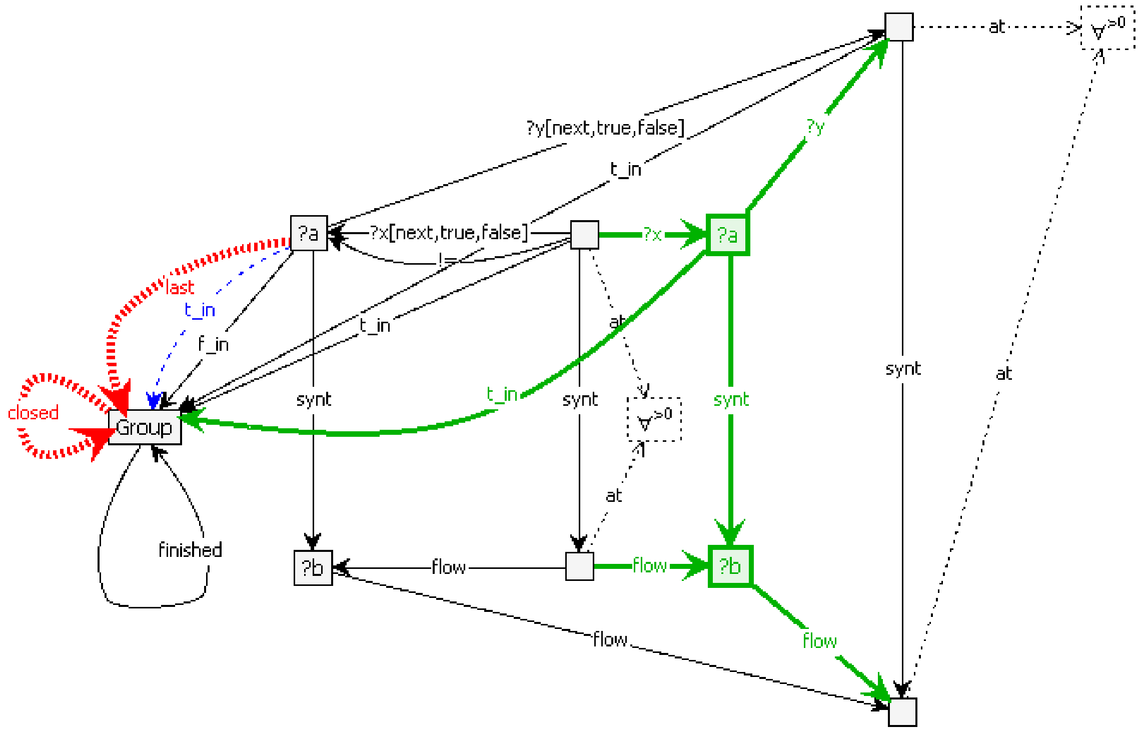
GroupType2-3-Step1



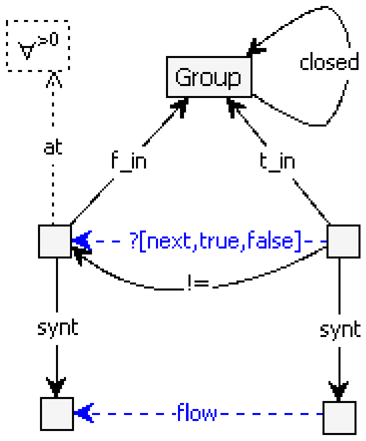
GroupType2-3-Step2



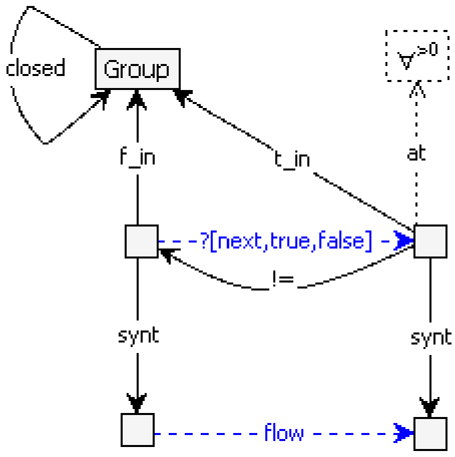
GroupType3-Step3-In



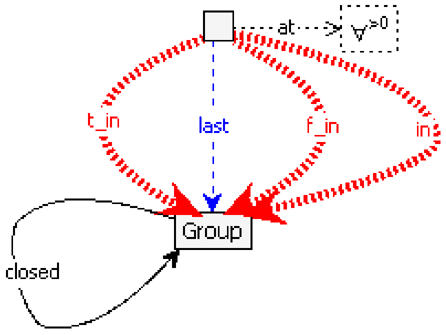
GroupType3-Step3-Del1



GroupType3-Step3-Del2

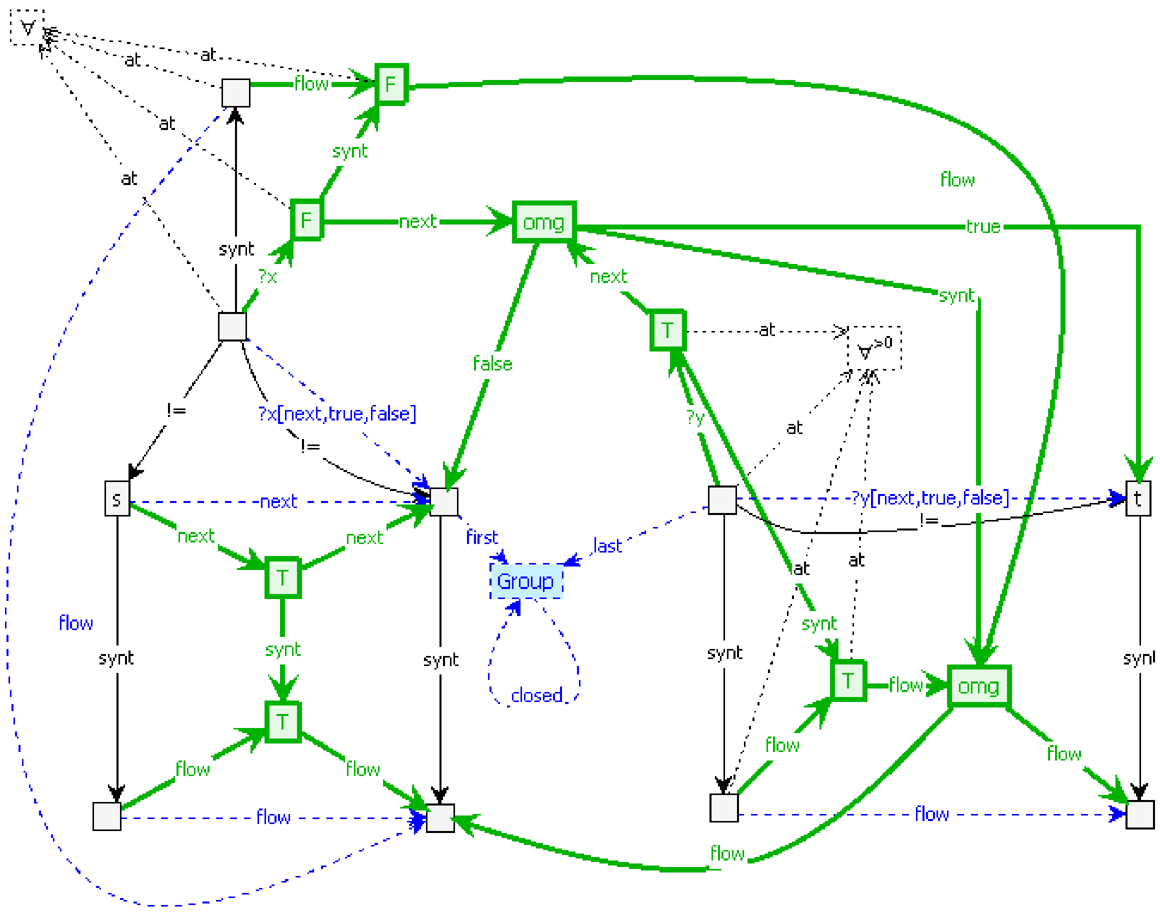


GroupType3-Step3-Del3



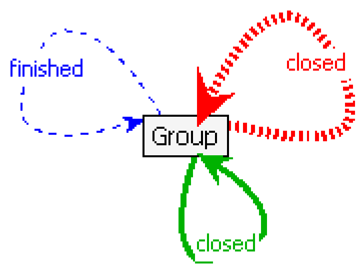
Priority 7.

GroupType2Normal-2Edges

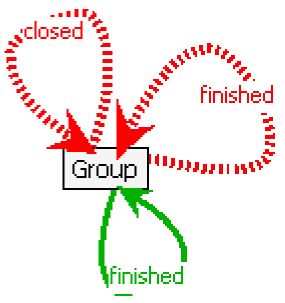


Priority 6.

GroupType-CloseGroup

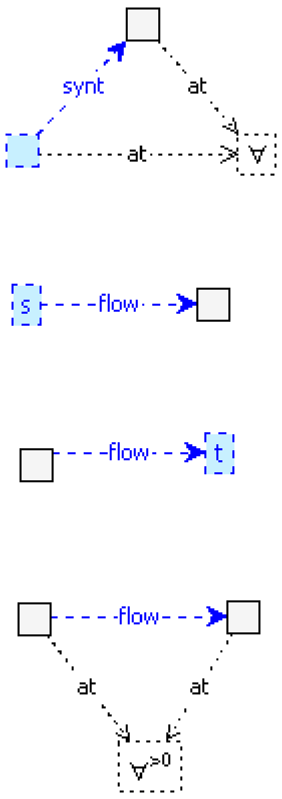


GroupType-FinishGroup



4. Syntax trees. On the last step of our transformation we delete the contracted flow diagram elements and get a final syntax tree.

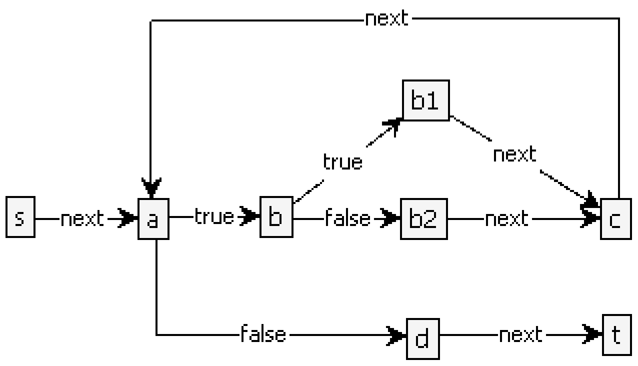
Priority 5.
GetSyntaxTree



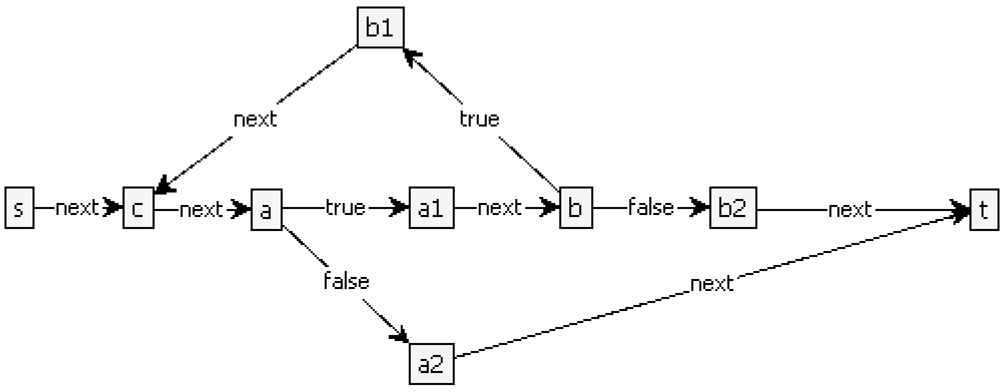
Appendix B.

Examples of Implementation within GROOVE

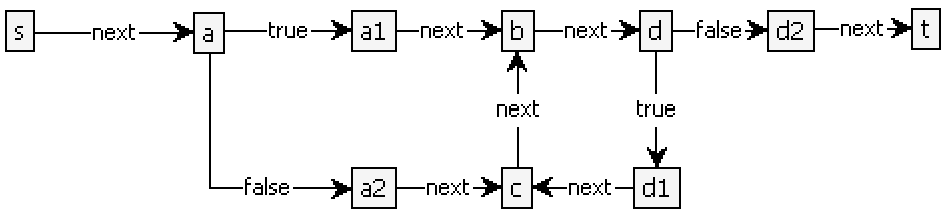
Example 1. An example of strongly decomposable flow graph with $n = 8$ nodes. In this case $v = 3$, $v_e = 1$, $v_d = 0$ and $V = (3 + 0) \times 1 = 3$. Normalization methods yield the same results.



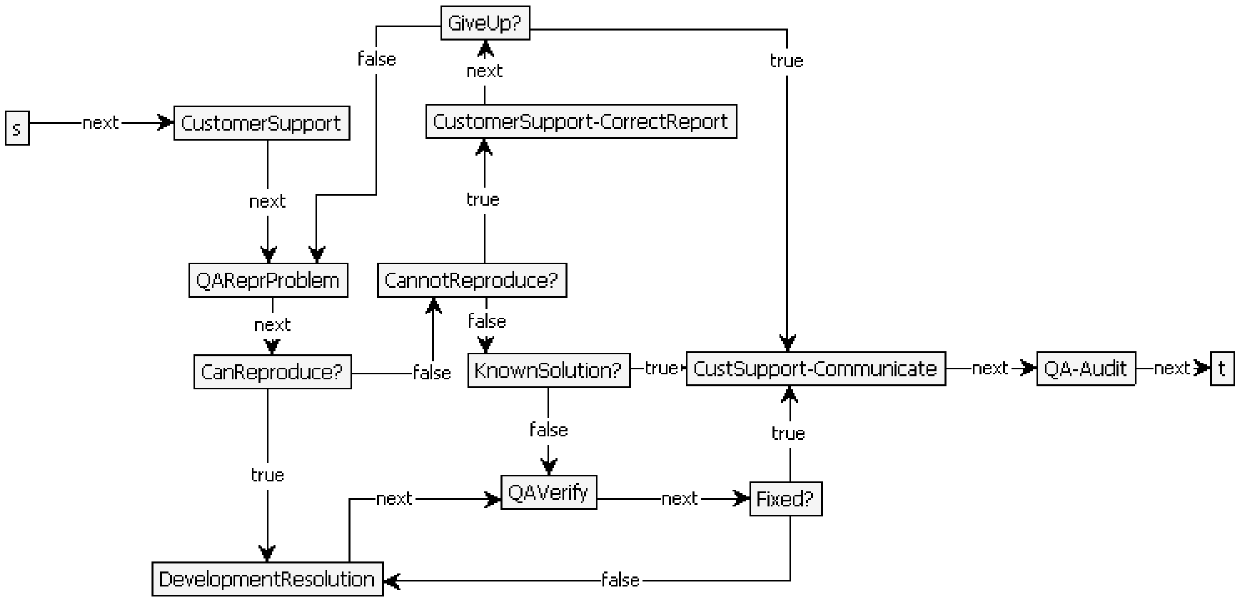
Example 2. The example of not strongly decomposable flow graph with $n = 9$ nodes from the paper (Peterson, W.W., Kasami, T., Tokura, N., 1973. On the capabilities of while, repeat and exit statements. In *Communications of ACM*, Vol. 16, No. 8, pp. 503-512). For the initial flow graph $v = 3$, $v_e = 3$, $v_d = 0$ and $V = (3 + 0) \times 3 = 9$. The final syntax tree for the straight Böhm-Jacopini method has $v = 4$, $v_e = 1$, $v_d = 0$ and $V = 4$. The SCC method is not applicable in this case.



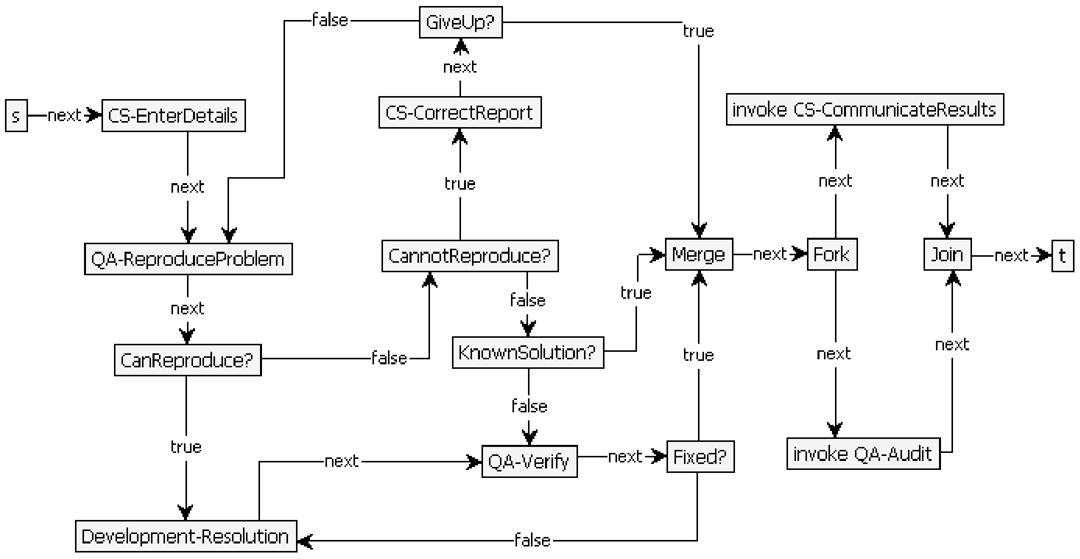
Example 3. The example of a flow graph with $n = 10$ nodes which was used throughout the paper. For the initial flow graph $v = 3$, $v_e = 3$, $v_d = 0$ and $V = 9$. The final syntax tree for the straight Böhm-Jacopini method has $v = 6$, $v_e = 1$, $v_d = 4$ and $V = (6 + 5) \times 1 = 11$. The best of five final syntax trees obtained by the nondeterministic SCC method has $v = 4$, $v_e = 1$, $v_d = 1$ and $V = (4 + 1) \times 1 = 5$.



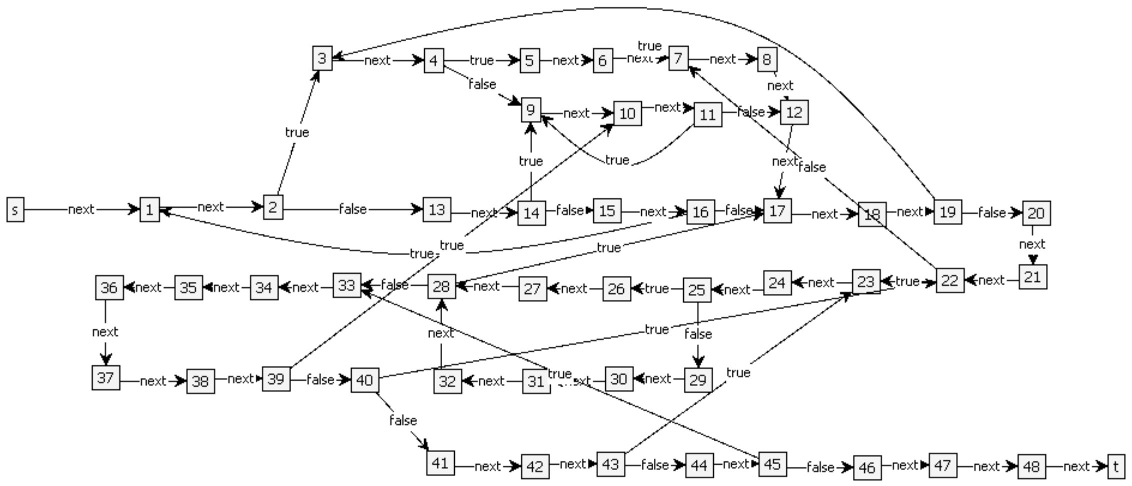
Example 4a. The example of a real flow graph with n = 14 nodes as provided in (JSPWiki, 2008. Trouble Ticket Scenario, http://www.xpdl.org/wiki/Wiki.jsp?page=TroubleTicketScenario) by Workflow Management Coalition without fork/join part. For the initial flow graph v = 6, v_e = 6, v_d = 0 and V = 36. The final syntax tree for the straight Böhm-Jacopini method has v = 12, v_e = 1, v_d = 6 and V = 18. The best of 12 final syntax trees obtained by the nondeterministic SCC method has v = 8, v_e = 1, v_d = 3 and V = 11.



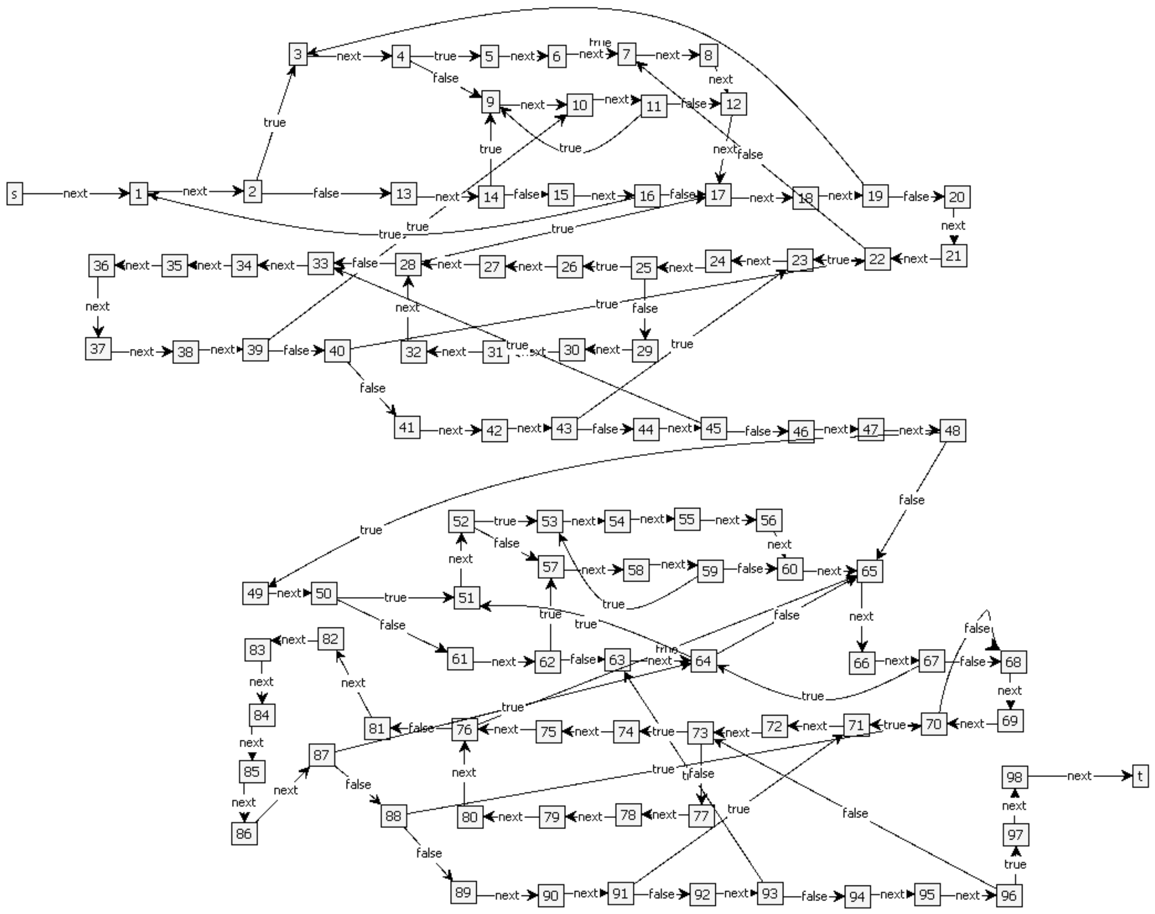
Example 4b. The example of a real flow graph with n = 17 nodes as provided in (JSPWiki, 2008. Trouble Ticket Scenario, http://www.xpdl.org/wiki/Wiki.jsp?page=TroubleTicketScenario) by Workflow Management Coalition with fork/join part. For the initial flow graph v = 10, v_e = 10, v_d = 0 and V = 100. The final syntax tree for the straight Böhm-Jacopini method has v = 12, v_e = 6, v_d = 4 and V = 96. The best of 12 final syntax trees obtained by the nondeterministic SCC method has v = 8, v_e = 4, v_d = 2 and V = 40.



Example 5. This example (a flow graph with $n = 50$ random nodes and random edges) is interesting as a performance and scaling test case. For the initial flow graph $v = 14$, $v_e = 11$, $v_d = 0$ and $V = 156$. The final syntax tree for the straight Böhm-Jacopini method has $v = 27$, $v_e = 1$, $v_d = 37$ and $V = 64$. The best of 52 final syntax trees obtained by the nondeterministic SCC method has $V = 32$.



Example 6. This example (a flow graph with n = 100 random nodes and random edges) is interesting as a performance and scaling test case. For the initial flow graph V = 276. The final syntax tree for the straight Böhm-Jacopini method has V = 154. The best of 72 final syntax trees obtained by the nondeterministic SCC method has V = 84.


s
t
next
true
false