Towards Model Structuring Based on Flow Diagram Decomposition

Arend Rensink
Department of Computer Science,
University of Twente
P.O. Box 217, 7500 AE, The Netherlands
a.rensink@utwente.nl

Maria Zimakova
Department of Computer Science,
University of Twente
P.O. Box 217, 7500 AE, The Netherlands
m.v.zimakova@utwente.nl

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.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques – *flow charts, state diagrams.*

General Terms
Algorithms, Management, Measurement, Performance, Design, Languages, Theory.

Keywords
Model transformations, graph transformations, model structuring, flow diagram decomposition, data flow graph, complexity measure.

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 [6] and attempts to compile UMLA to BPEL programs; the last issue is discussed, for instance, in [16]. 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 [3].

There exists 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 Section 2, we discuss the flow graph decompositions and complexity measure problem in Section 3. We consider these to be the heart of our contribution. In Section 4 we implement those methods with graph transformations, employing the graph-transformation tool Groove [14] for rule execution. Finally, in the conclusion (Section 5) we come back to the above considerations, evaluate our results and discuss plans for future work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
BM-MDA'09, June 23, 2009, Enschede, the Netherlands
Copyright © 2009 ACM ISBN 978-1-60558-503-1/ 09/06... $10.00

2. BASIC NOTIONS

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

Definition 1. 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 where Λ is a finite set of node and edge labels;
- λ is a *labeling function* $\lambda: N \cup E \to \Lambda$.

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. A *path* in a graph G is an alternating sequence of nodes and edges 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 $tgt(e_i) = v_{i+1}$.

Let G be a labeled directed graph as above with a labeling 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 following:

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

We call this the *word representation* of p.

Definition 2. A *flow graph* Φ is a triple (G, s, t), where

- $G = (N, E, \lambda)$ is a 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.

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.

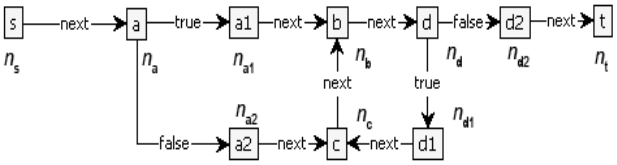


Figure 1. Flow graph example

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 $v \in N$.
- 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 $v \in N$.

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).

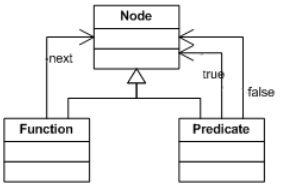


Figure 2. The types in the flow diagram

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 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 3. Let p be a full path in a flow graph $\Phi = (G, s, t)$. Then an *execution sequence* $\texttt{Seq}(p)$ is the word representation of p.

For instance, sequence (*s next a true* a_1 *next b next d false* d_2 *next t*) is an execution sequence for our example in Figure 1.

Now, let `Path` be a set (maybe infinite) of all possible full paths in the flow graph $\Phi = (G, s, t)$ in the light of the discussion above. The word representation of `Path` thus regarded as a *language* $\texttt{Lang}(\Phi) = \lambda(\texttt{Path})$ defined over the alphabet Λ.

Definition 4. Two flow graphs Φ_1 and Φ_2 are *equivalent* (denote it as $\Phi_1 \sim \Phi_2$) if they define the same languages: $\texttt{Lang}(\Phi_1) = \texttt{Lang}(\Phi_2)$.

Algebra of flow diagrams. 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 [2], 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 four elementary types $\Gamma = \{\Pi, \Delta, \Omega\}$ *base subdiagrams*.

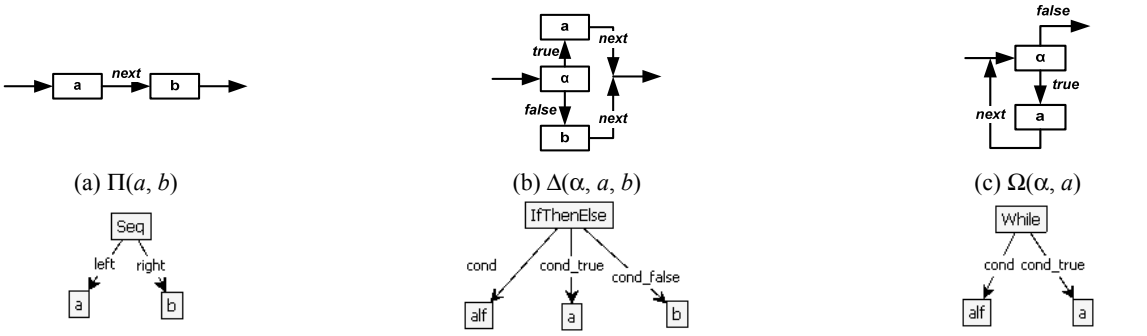
For our subsequent definitions we also use the notions of a signature and algebra, as defined in [5]. The ingredients of these definitions that are important here are:

- A collection of data *sorts Sort*.
- A collection of *carrier sets Data*, partitioned into subsets for each of the sorts in *Sort*.
- A mapping *par*: $Oper \to Sort^+$ that associates to every operation $op \in Oper$ a non-empty string of sorts.

Note that an operation *op* with no parameters represents a constant value.

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 5. 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 $\texttt{Sub}: N' \to \Theta$ 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



(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

following rules:

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

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 sorts *fg*, *pred* and *func*, representing the arbitrary flow graphs, predicative nodes and functional nodes, respectively. We also define a constant *empty* for the empty flow graph and operation symbols for the elementary flow graphs (for each functional node) and the base subdiagrams $\Gamma = \{\Pi, \Delta, \Omega\}$:

`Sig` =
Sort: *fg*, *pred*, *func*;
Oper: *empty*, *elem*, Π, Δ, Ω;
par: *empty*:→ *fg*,
elem: *func* → *fg*,
Π: *fg fg* → *fg*,
Δ: *pred fg fg* → *fg*,
Ω: *pred fg* → *fg*.

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

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

Definition 6. A flow diagram $\Phi = (G, s, t)$ where $G = (N, E, \lambda)$ is *strongly decomposable* (or *well-formed* in terms of [6] and [13]) if there exists an expression *exp* in the `Sig`-algebra `FlowGraph` such that `FlowGraph`$[\![exp]\!] \cong \Phi$.

Together with a strong decomposition, [2] considered another decomposition which is obtained by operating on an *equivalent* strongly decomposable flow graph. Formally, a flow graph Φ is *weakly decomposable* if $\Phi \sim \Phi'$ for some strongly decomposable flow graph Φ'.

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* [11].

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 7. 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 constructions `Seq`, `IfThenElse` and `While` n Figure 3 (d)-(f), respectively. The different node types that are supported by syntax trees, together with their relationships, are shown in Figure 4.

Similarly to the algebra `FlowGraph` above, we can implement a signature `Sig` with a different algebra `SyntaxTree` on a set ϑ of syntax tree constructions {`Seq`, `IfThenElse`, `While`}.

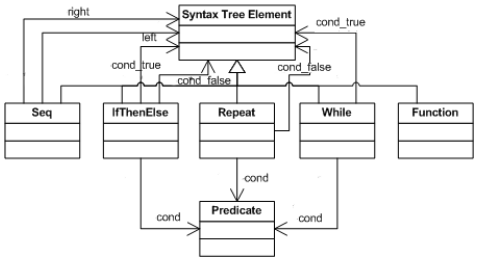


Figure 4. The types in the syntax trees

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

Definition 8. 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.

3. FLOW DIAGRAM DECOMPOSITION

In this section we consider the structuring problems imposed by our common example. The first problem is a very complicated graph structure of the flow diagram. One of the decomposition approaches to solve that kind of problem was provided in [2]. We discuss details of this approach in Section 3.1. The main flow diagram also has one meaningful (more than one node) *strongly connected component* (SCC); therefore we can improve the decomposition quality applying the method of [13]. The application of this algorithm as a part of the general approach is discussed in Section 3.2. The complexity measure to evaluate the advantage of different methods is considered in Section 3.3 and some concrete implementation results are presented in Section 4.

A concise review of many of other results developed in this field has been prepared in [7]. We also come back to that discussion in the closing remarks about future work in Section 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 [2] were presented in Section 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 $w = (x_1\, x_2 \ldots x_i \ldots x_j \ldots x_n) \in$ `Lang`(Φ) where $x_k \in \Lambda \cup \{T, F, K, \omega\}$, $k \in [1, n]$

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

where $\widetilde{x} = \begin{cases} true & \text{if } x = T; \\ false & \text{if } x = F; \\ x & \text{otherwise} \end{cases}$.

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

$$\texttt{Restrict}(w) = (x_1\, x_2 \ldots x_{i-1}\, x_{i+2} \ldots x_n).$$

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

Definition 9. Two flow graphs Φ_1 and Φ_2 extended by functions *T*, *F*, *K* and predicate ω are *equivalent* if they define the same restricted languages, that is $\overline{\texttt{Lang}}(\Phi_1) = \overline{\texttt{Lang}}(\Phi_2)$.

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.

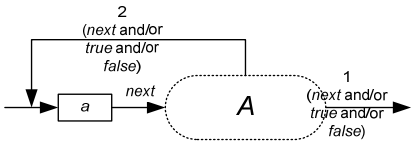
Theorem 1. For any flow graph Φ_1 there exists (at least) one equivalent strongly decomposable flow graph Φ_2 extended by the functions *K, T, F* and predicate ω; in other words, any flow graph is weakly decomposable.

The proof of the theorem and the decomposition algorithm is based on the flow diagram classification represented in Figure 5 (a)-(c). The equivalent strongly decomposed flow diagrams of type I and II are shown in Figure 6 (for more details see [14]).

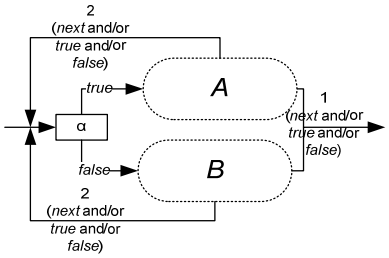
3.2 SCC decomposition

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 [13].

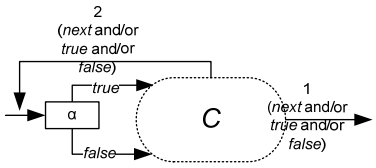
Theorem 2. Every flow diagram can be transformed into an equivalent strongly decomposable (well-formed) flow diagram by node duplication (proof see [13]).



(a) Structure of a type I diagram



(b) Structure of a type II diagram



(c) Structure of a type III diagram

Figure 5. Three types of flow diagrams

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

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 (d) 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 requires 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 review [3].

A well-known and often used complexity measure was proposed by McCabe in [10].

Definition 10. The *cyclomatic number* $v(\Phi)$ of flow graph Φ with *n* nodes, *e* edges, and *p* connected components is

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

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

Let a *decomposition degree* $m(\Phi)$ of a flow graph Φ be a number of substitutions Φ_{v_i}, $i = 1, \ldots, n$, such that $\Phi_{v_i} \in \Gamma \setminus \{\Pi\}$. Then

Definition 11. The following definition of *essential complexity* $v_e(\Phi)$ is used to reflect the lack of structure:

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

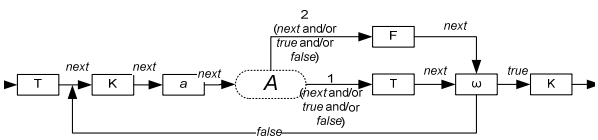
In the large, we propose to measure a full complexity of the flow diagram as follows:

Definition 12. 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* $V(\Phi)$:

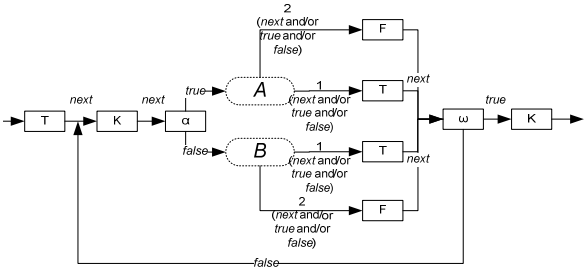
$$V(\Phi) = [v(\Phi) + v_d(\Phi)] \times v_e(\Phi).$$

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

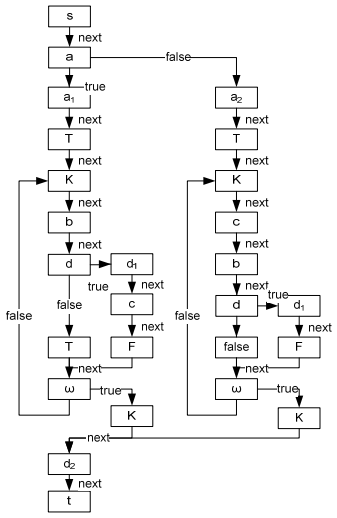


(a) Transformation of a type I diagram

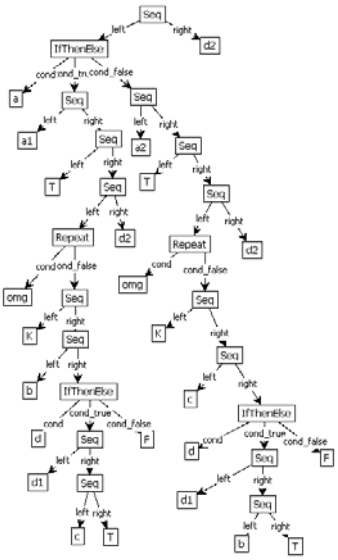


(b) Transformation of a type II diagram

Figure 6. Transformation of a type I and type II diagrams



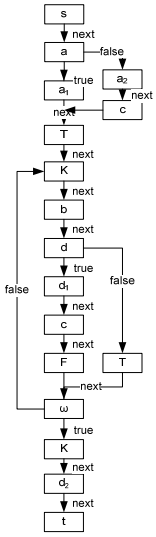
(a) Böhm-Jacopini decomposition



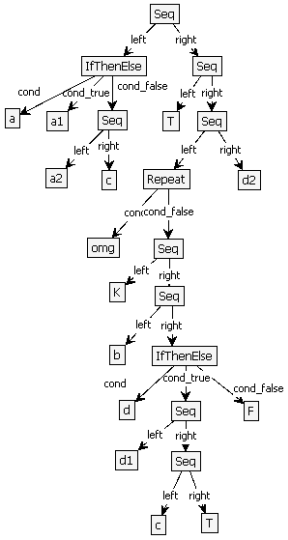
(b) The syntax tree decomposition of graph (a) with $V = 11$

```
begin
  if a then begin
     a1;
     var_bool := true;
     repeat
       b;
       if d then begin
          d1;  c;
          var_bool := false;
       end else
          var_bool := true;
     until var_bool;
  end else begin
     a2;
     var_bool := true;
     repeat
       c;  b;
       if d then begin
           d1;
           var_bool := false;
       end else
           var_bool := true;
      until var_bool;
  end;
  d2;
end.
```

(c) The text code representation of the syntax tree (b)



(d) Decomposition using SCC method



(e) The syntax tree decomposition of graph (d) with $V = 5$

```
begin
  if a then
       a1;
  else begin
       a2;
       c;
  end;
  var_bool := true;
  repeat
       b;
       if d then begin
           d1;
           c;
           var_bool:= false;
       end else
           var_bool:= true;
  until var_bool;
  d2;
end.
```

(f) The text code representation of the syntax tree (e)

Figure 7. Two strongly decomposable (well-formed) extended flow graphs equivalent to the flow graph in Figure 1, respective final syntax trees and text code representations

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.

4. GROOVE IMPLEMENTATION

We implemented techniques described in Section 3 within the Groove (see [5], [9], [14]) 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*.

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

- *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 as was discussed in Section 3.1.

- *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.

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

Unfortunately, we cannot explain the precise workings of the Groove implementation in the available space; however, the rules and some example cases are available at [15] for the reader to try out.

The example of the final syntax tree for the straight Böhm-Jacopini method applied to the initial flow diagram in Figure 1 is shown in Figure 7 (b) 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 7 (e) 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 7 (b) and Figure 7 (e) are presented in Figure 7 (c) and Figure 7 (f), 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 at [15].

Two flow graphs with 50 and 100 random nodes and edges are interesting as performance and scaling test cases. The results comprise about 1500 and 2500 transitions, respectively (as compared with 8 transitions for the first simple case). This shows that the potential advantages of the approach, in terms of graph transformations, could be applied in practice.

5. CONCLUSIONS

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 [7].

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

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

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.

6. ACKNOWLEDGEMENTS

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

7. REFERENCES

[1] Allen, F.E., 1970. Control Flow Analysis. In ACM Sigplan Notices.

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

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

[4] Dumas, M., ter Hofstede A.H.M., 2001. UML Activity Diagrams as Workflow Specification Language. In Proceedings of the UML'2001 Conference. Toronto, Canada.

[5] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G., 2006. Fundamentals of Algebraic Graph Transformation. Springer-Verlag, Berlin, Germany.

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

[7] Erosa, A.M., Hendren L.J., 1994. Taming control flow: A structured approach to eliminating goto statements. In Proceedings of ICCL, Toulouse, France, pp 229–240.

[8] Kiepuszewski, B., ter Hofstede, A.H.M., Bussler, C., 2000. On Structured Workflow Modelling. In Proceedings CAiSE'2000, Stockholm, Sweden, Vol. 1789, pp. 431-445.

[9] Kleppe, A.G., Rensink, A., 2008. 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.

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

[11] Object Management Group, 2005. Abstract Syntax Tree Metamodel, Request For Proposals (RFP). http://www.omg.org/cgi-bin/doc?admtf/05-02-02.pdf.

[12] Object Management Group, 2008. Business Process Modeling Notation, V1.1. http://www.omg.org/docs/formal/08-01-17.pdf

[13] 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.

[14] Rensink, A., 2004. The GROOVE Simulator: A Tool for State Space Generation. In AGTIVE 2003. Springer, Heidelberg, Germany, Vol. 3062, pp. 479–485.

[15] Rensink, A., Zimakova, M., 2009. Examples of Implementation in Groove. Available at http://ewi.utwente.nl/~mzimakova/bm-mda_2009.

[16] Zhao, W., Hauser, R., Bhattachaya, K., Bryant B., 2005. Compiling Business Processes: Untangle Unstructured Loops in Irreducible Flow Graphs. Technical report UABCIS-TR-2005-0505-1, Birmingham, USA.