Electronic Communications of the EASST
Volume 4 (2006)

Proceedings of the Second International Workshop on Graph and Model Transformation (GraMoT 2006)

Search Trees for Distributed Graph Transformation Systems

Ulrike Ranger and Mathias Lüstraeten

13 pages

Guest Editors: Gabor Karsai, Gabriele Taentzer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Search Trees for Distributed Graph Transformation Systems

Ulrike Ranger and Mathias Lüstraeten

Department of Computer Science 3 (Software Engineering)
RWTH Aachen University
Ahornstrasse 55, 52074 Aachen, Germany
[ranger|matlue]@i3.informatik.rwth-aachen.de

Abstract: Graph transformation systems, like PROGRES and Fujaba, can be used for modeling software systems of various domains, and support the automatic generation of executable code. A graph transformation rule is executed only if the pattern of the transformation's left-hand side is found in the graph. The search for the pattern has an exponential worst-case complexity. In many cases, the average complexity can be reduced using search tree algorithms in the code generation phase. When modeling distributed graph transformations, the communication overhead between the coupled applications largely affects the pattern matching performance. Therefore, we present an approach for adapting existing search tree algorithms for the efficient search of distributed graph patterns. Our algorithm divides the distributed graph pattern into several sub-patterns such that every sub-pattern affects solely the graph of exactly one coupled application. The results of these sub-patterns are used to determine the match for the entire graph pattern.

Keywords: Graph Transformations, Search Trees, Distributed Systems

1 Introduction

Graph transformation systems (GTS), like PROGRES [Sch91] and Fujaba [FNTZ00], can be used to model software systems in a visual way. Additionally, they facilitate the generation of executable code, like C or Java, for the modeled software system. Several large projects of various domains have been developed using GTS, but they lack support for specifying distributed systems. In our project, we extend GTS for appropriate concepts including the visual specification of distributed graph transformations, which affect several applications simultaneously.

To support distributed graph transformations, we have to consider three aspects: First, the syntax and semantics of distributed graph transformations have to be defined (*specification level*). Second, a concept for the generation of efficient code has to be developed considering special requirements like communication costs (*code generation level*). Third, a runtime environment must be designed and implemented, which supports the execution of the generated applications (*runtime level*). This paper focuses on the code generation level. The specification level is described in [RSM06].

Our code generation approach for distributed graph transformations is based on the existing *search tree algorithms* of PROGRES and Fujaba. Search trees allow to reduce the complexity for searching graph patterns specified within a graph transformation, as this search has an exponential worst-case complexity.

Regarding distributed graph transformations, also the high communication costs within a distributed system have to be considered within the search trees. Therefore, we adapt the code generation algorithm by the following approach: The distributed graph pattern specified within a graph transformation is divided into several sub-patterns, such that every sub-pattern affects solely the graph of exactly one coupled application. These sub-patterns are sent to the coupled applications, thus reducing the communication costs in comparison of querying the applications for every single remote element of the pattern. As sub-patterns may depend on formerly queried pattern elements, the dependencies to other sub-patterns have to be analyzed. Thus, the sub-patterns are executed parameterized with former results, and their results determine the match of the entire distributed graph pattern.

The paper is structured as follows: In Section 2 we shortly introduce distributed graph transformations considering a simple Publishing Trade as an example. Search trees, which are used to generate efficient executable code for local graph transformations, are described in Section 3. Section 4 presents our approach for adapting the presented search tree algorithm to distributed graph transformations. A summary and an outlook on future work are given in Section 5.

2 Specifying Distributed Graph Transformation Systems

In this section, we shortly introduce the visual modeling of distributed graph transformation systems at specification level [RSM06]. This comprises the graph schema of the distributed system, which will be described in Subsection 2.1, and the visual specification of the dynamic behavior presented in Subsection 2.2.

In our approach, a distributed system is modeled by different specifications, each modeling one module of the software system. At runtime, each specification is executed separately in an application storing the according host graph[1]. These applications are coupled at runtime by executing distributed graph transformations, which affect several applications simultaneously. The specificator only has to develop appropriate distributed transformations, as the GTS generates adequate source code, and the runtime environment automatically performs their execution.

2.1 Graph Schema of a Distributed Graph Transformation Systems

To illustrate our concepts, we use a simple *Publishing Trade* as example, whose static structure is depicted in Figure 1. Here, we assume that a book publisher already has an existing module Publisher, managing all Books and their Authors[2]. As the Publisher has decided to sell his Books in an online shop, a new module Publisher Shop for this purpose shall be developed, using the existing specification of the Publisher module. The new Publisher Shop manages the necessary data for selling the Books of the Publisher including Customers and their according Orders. Additionally, a Customer can have a wish list for desired Books, which is modeled by the wishes-edge. For advertising Books, the Publisher Shop memorizes the Authors liked by the Customers (likes-edge) when a Customer has bought a Book.

[1] Until now, our approach has the restriction that every specification is executed exactly once within a distributed system. We are developing appropriate concepts to fill this gap including the introduction of *roles* for executed applications and the usage of adequate *role restrictions* within graph transformations.

[2] For the sake of simplicity, every Book is written by exactly one Author.

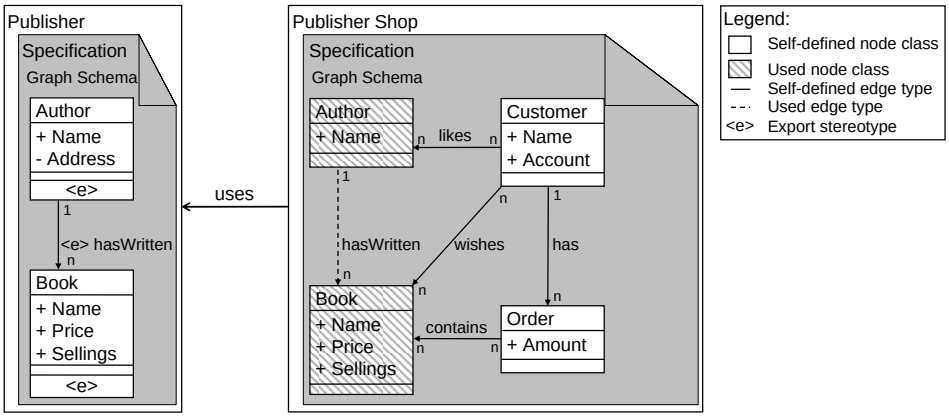


Figure 1: Graph schema for the distributed Publishing Trade

The Publisher and the Publisher Shop are modeled by different specifications shown in Figure 1. As the Publisher manages all information about the Books and the Authors, the Publisher Shop needs access to these data. Therefore, the Publisher defines an *export interface* by using the stereotype <e>. Each graph schema element marked with this stereotype forms the interface of the specification. An exported node class consists of the class name and the public attributes and methods defined within the node class.

The exported specification interface can be *used* by other specifications. To distinguish between used and self-defined schema elements, the used elements are depicted as striped rectangles resp. dashed arrows within the specification. Although the used graph schema elements are *read-only*, they can be applied within the specification in nearly the same way as local, self-defined schema elements. E.g. edges can be defined between self-defined and used node classes or just between used node classes. With this mechanism, the Publisher Shop of our example integrates the Publisher interface by defining new edges, e.g. the wishes-edge between the self-defined Customer node class and the used Book node class realizing the wish list.

2.2 Distributed Graph Transformations

The dynamic behavior of a distributed system has to be specified within *distributed graph transformations*. In distributed transformations, edges and nodes of used types can be applied in the same way as of self-defined types but they address remote objects instead of local objects. At runtime, every node and edge only exists once within a distributed system, i.e. in the host graph of the application, which is based on the specification defining its type. Coupled applications only store *reference nodes* on remote nodes instead of copying remote nodes with their data into the local host graph. They are explicitly inserted into the runtime graphs storing the location of their according remote nodes. As reference nodes are only helper structures for accessing remote nodes, they are implicitly managed by the runtime level and are not regarded in the graph transformations. Their usage supports the realization of self-defined edges incident to remote nodes. In contrast to reference nodes, we do not store reference edges.

A remote node (and thus an appropriate reference node in the local host graph) can be created

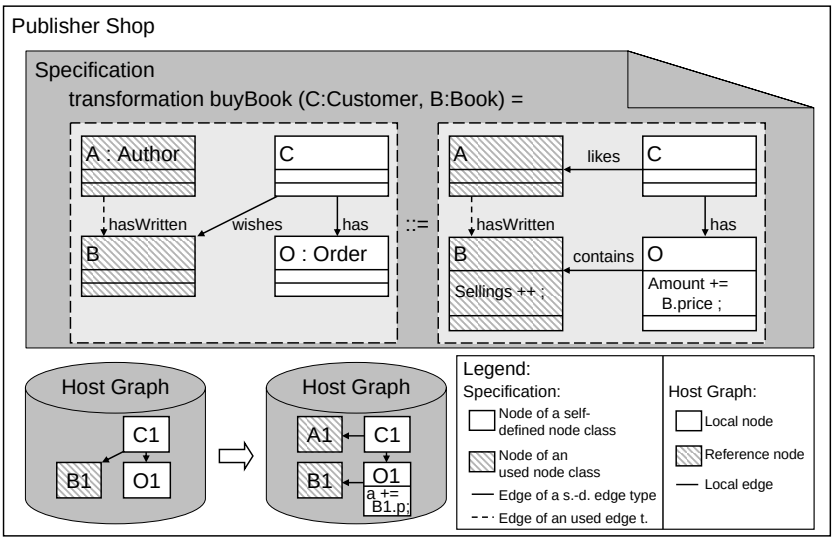


Figure 2: Graph transformation BuyBook

within a distributed graph transformation by using the according node only in the transformation's right-hand side. This concept corresponds to creating a local node of a self-defined type. Analogously, the deletion of a remote node (and thus of the corresponding reference node) is triggered by using the according node only on the left-hand side of the transformation.

Furthermore, a reference node is automatically created in the local host graph if an appropriate node is used in the left-hand side of the transformation, and no adequate reference node is locally available, although an according remote node exists. In this case, the remote application defining the node's type is searched for an adequate node, and an according reference node is created. Additionally, all reference nodes are automatically deleted if the actual remote node is deleted. Operations specified on remote objects, like the deletion of a node or an attribute assignment, are propagated transparently to the corresponding remote application[3]. To ensure the consistency of the host graphs by remote graph transformations, we will introduce a *rule engine* (cf. Section 5).

Figure 2 shows an example of the distributed graph transformation buyBook of the Publisher Shop. In this transformation, a Customer buys a Book (given as input parameters), which has been on the Customer's wish list. This leads to the deletion of the wishes-edge as it is not longer needed. Furthermore, the two edges likes and contains are created. The attribute Sellings counting the number of sales of B is incremented. Below the transformation, two example host graphs of the Publisher Shop are depicted showing the host graph before and after the execution of the transformation. According to the transformation edges are created and deleted and a new reference node A1 for the Author A is inserted. In the Publisher's host graph only the attribute Sellings of node B1 is modified, which is not shown in Figure 2.

[3] For propagating remote operations and the management of references, we develop an appropriate plug-in for the database DRAGOS [Böh04], which is used by PROGRES (and soon by Fujaba) prototypes for storing the graphs.

3 Search Trees

After modeling a software system with a GTS, appropriate code has to be generated. As the matching of the left-hand side of a graph transformation has an exponential worst-case complexity, PROGRES and Fujaba use *search trees* [Zün96] for their code generation. In this section, we will describe these search trees using the code generation plug-in CodeGen2 [GSR05] of Fujaba as an example. As we present the current mechanism, we ignore the remote nature of operations concerning a coupled application.

3.1 General Structure of Search Trees

To generate executable and efficient code for graph transformations, every transformation is translated into a search tree. A search tree is a tree, having *operations* as nodes and *precondition*-relations between an operation and its child operations. The search tree contains all operations, which have to be performed for the modeled graph transformation in an appropriate execution order. In this paper, we use a more general notion of a search tree, since not only search operations are used. CodeGen2 uses over 20 different operations, covering constraint checks and operations for creating and deleting nodes.

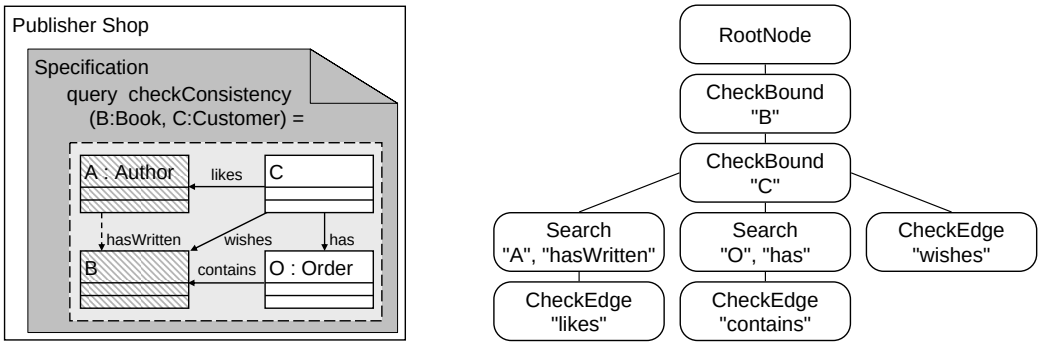
In general, the operations describe the runtime semantics of modeling elements of a transformation. There is no *1-to-1*-mapping between operations and modeling elements. One modeling element may result in several different operations in the search tree, because its runtime semantics is precisely defined in the code generation phase. On the other hand, one operation may also cover several modeling elements, as the semantics of one modeling element may be determined only in combination with other elements.

For translating the graph transformation into a search tree, every node of the transformation's left-hand side (*graph pattern*) has to be identified in the host graph. As a first step, a search tree covering all bound nodes[4] is created. The second step regards all unbound nodes calculating a *spanning forest* for the graph pattern. For every bound node, the forest contains a tree having the bound node as root node. Every unbound node has to be searched in the host graph by traversing an edge of the pattern incident to a bound node. The unbound node together with its edge is inserted into the tree of the corresponding bound node. After building the complete spanning forest, for all unbound nodes of the spanning forest a *search operation* representing the unbound node and the traversed edge is inserted into the search tree. This operation is placed as child of the operation, which binds the edge's source node.

Note that Fujaba does not allow isolated unbound nodes in graph patterns, because it uses the heap of the JAVA runtime environment instead of a graph database. Other graph transformation systems like PROGRES support this feature by querying the underlying database for nodes of a specific type. For the sake of simplicity, we do not regard the search of isolated nodes and patterns although this is covered by our approach using DRAGOS [Böh04].

After inserting the search operations into the search tree, the remaining modeling elements of the graph transformation (like edges and attribute assertions) have to be inserted. The position of

[4] A *bound node* is a node, which already has been uniquely identified in the host graph, for example nodes given as input parameters of a transformation. In contrast, *unbound nodes* are nodes, which have to be searched in the host graph. If an appropriate node for the unbound node is found, this node becomes a bound node.



(a) Graph query (b) Search tree

Figure 3: Graph query checkConsistency

such an operation is determined by the following overall invariant of the tree: All preconditions of an operation must be fulfilled by preceding operations. To satisfy this invariant, the search tree may have to be reorganized according to the invariant. The complete algorithm for the search tree generation is shown in [GSR05].

To illustrate the generation of search trees, we introduce the query checkConsistency depicted in Figure 3(a). This query may be used for consistency checks, e.g. to test if the wishes-edge has been deleted when the Book has been bought by the Customer and is thus part of an Order.

Figure 3(b) shows a possible search tree for checkConsistency. The tree generation starts with two CheckBound-operations for B and C, which are inserted as children of the RootNode. A CheckBound-operation checks the validity of a bound node, i.e. that the node is not equal to `null`.

To determine all search operations needed to cover the unbound pattern nodes, a spanning forest is computed consisting of two trees: One tree with B as root node and the other tree with C as root node. For our example, we assume that the spanning forest is given by all nodes with the hasWritten- and the has-edge. According to this assumption, the search operations for A and O using the hasWritten- resp. the has-edge are inserted into the search tree.

To cover the remaining edges, namely contains, wishes and likes, CheckEdge-operations have to be inserted. A CheckEdge-operation must have at least those operations in their parent hierarchy, which cover the source and the target node of the edge. To fulfill this invariant, the tree may have to be reorganized before the CheckEdge-operations can be inserted.

In our example, the CheckBound-operation of node C (CheckC) is moved below the CheckBound-operation of node B (CheckB) and all children of CheckB are moved below CheckC[5]. This reorganization is needed for the correct insertion of the CheckEdge-operation of the wishes-edge into the tree. All other CheckEdge-operations can be inserted without any reorganization.

3.2 Cost Model

A graph transformation may have several different valid search trees, because there are several ways to cover all modeling elements by operations. To evaluate the different search trees, the

[5] These reorganizations are possible, because siblings in the search tree are independent of each other.

Table 1: Cost table for search operations of query checkConsistency

index	source	edge	target	cost
1	C	has	O	25
2	B	contains	O	25
3	B	hasWritten	A	1
4	C	likes	A	25

code generation uses a *cost model* presented in the following.

Search operations may have different runtime costs, which can be estimated by exploiting e.g. the cardinalities of the graph schema. Search operations using a *to-n*-edge for matching an unbound node are very costly compared to those using a *to-1*-edge. Using a *to-1*-edge, the exact candidate node can be directly determined, whereas a search operation using a *to-n*-edge leads to n possible candidate nodes.

However, the costs for search operations can only be estimated, since the exact cardinality for a *to-n*-edge in the host graph is not known during the code generation phase. Therefore, CodeGen2 and the PROGRES code generation use default values for the expected costs. [VVF05] presents an approach for adapting the edge cardinalities by analysis of sample instance graphs leading to very precise cost estimations. So far, this approach is not considered by CodeGen2 and thus not regarded in this paper.

Even by considering only the static graph schema, the runtime behavior can be greatly improved. Before generating the search tree, the costs for each operation of the transformation are estimated. Afterwards, a minimum spanning forest for the graph pattern is computed, which determines the preliminary search tree. This preliminary tree is incrementally extended by operations until all modeling elements are covered choosing the cheapest operation in each step.

Considering our example checkConsistency (cf. Figure 3(a), Figure 3(b)), there are four possible search operations (cf. Table 1), which have to be considered for the search tree generation. According to the target cardinalities of the edges, the costs for the search operations are either 1 or n. Note that this cost model is highly simplified and n has to be a real value greater than 1. For this purpose, CodeGen2 contains different *cost strategies* for assigning concrete cost values. The default cost strategy for search operations based on *to-n*-edges assumes a cardinality of 50. As result, it computes a cost of 25, since this is the value having 1 as lower and 50 as upper cardinality.

To minimize the costs of the search tree and thus get the best runtime efficiency of the generated code, the algorithm chooses the cheapest search operations until all unbound nodes are covered. In our example, search operation 3 is chosen first due to the smallest costs. Since node O can be covered by two operations with the same costs, the algorithm chooses non-deterministically between the search operation 1 and 2. Given that operation 1 is chosen, the search tree depicted in Figure 3(b) is computed.

Table 2: Modified cost table for search operations of query checkConsistency

index	source	edge	target	cost
1	C	has	O	25
2	B	contains	O	25
3	B	hasWritten	A	51
4	C	likes	A	25

4 Search Trees for Distributed Graph Transformations

In Section 2 we presented our syntax and semantics for modeling distributed graph transformations. To execute such a transformation, it has to be translated into executable code. Therefore, we extend the CodeGen2-approach presented in Section 3.

In the following, only the code generation for queries is shown, because queries have the greatest impact on the runtime efficiency. A distributed query requires *remote operations*, like remote check operations and remote search operations. Each remote operation affects exactly one application which is called *home application*. After searching the distributed pattern, the transforming operations are performed, which require no modifications of CodeGen2.

4.1 Remote Offset

Remote operations have additional costs compared to their local correspondents, which results from the communication delay between the participating applications. The additional cost is called *remote offset* and depends on the network infrastructure. The remote offset is configurable for each pair of applications within a distributed system. By using a different cost strategy for remote operations, the offset is integrated into CodeGen2.

Revisit the query consistencyCheck introduced in Section 3. Every search tree that is computed without regarding a remote offset uses the hasWritten-edge as first search operation. Thus, a remote operation is performed even it is not necessary, since A can be found locally via likes.

In many cases, a remote operation is more costly than every local operation. Considering this in our cost model, we introduce a remote offset for remote operations, which is larger than the maximal cardinality of any local edge. Thus, assuming a cost of 25 for a search operation using a *to-n*-edge, we set the remote offset to 50. The modified costs are depicted in Table 2.

Figure 4 shows the search tree for the query checkConsistency, which is computed according to the modified cost table (cf. Table 2). Search operation 4 is chosen first instead of search operation 3 due to the remote offset. The only remote operation in the modified search tree is the CheckEdge-operation of likes, which is depicted striped in the search tree.

4.2 Boundary Nodes

In many cases, there exists a number of operations, which have the same home application. Instead of formulating queries for every operation, operations may be processed *en bloc* leading to one single query. Thus, the number of communication steps can be heavily reduced.

The crucial question is how to cut the graph pattern into separate sub-patterns (*remote blocks*),

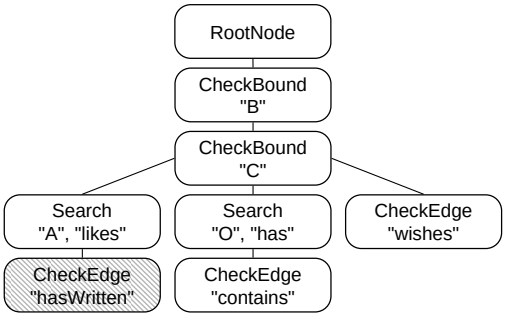


Figure 4: Modified search tree for query checkConsistency

such that every sub-pattern concerns solely one application. Additionally, several blocks for the same application may be needed, if contained elements are dependent on non-contained elements.

To illustrate this fact, the graph query AdvertiseAuthor is depicted in Figure 5. The query can be used to advertise an Author to a given Customer C, using the likes-relation of another Customer with the same interests. Starting at the only bound node C, the unbound nodes are searched in the following order: B, A1, C1 and A2. This is the only possible order, as every search for an unbound node needs at least an edge incident to a bound node. Because of these dependencies, two different remote blocks are necessary for the same application, i.e. the Publisher.

To define the boundary of a block, we introduce the notion of *boundary nodes*. A boundary node is a node which needs at least one remote operation for its identification, and is incident to a local edge. In Figure 5, B and A2 are boundary nodes due to the remote attribute check. Without the attribute check, the nodes B and A2 can be covered locally. A block is defined as the largest connected graph pattern, which is limited by boundary nodes and covers only remote operations concerning the same application. E.g. two blocks exist in the query AdvertiseAuthor: Block 1 contains the attribute check of node B, the node A1 and the hasWritten-edge. Block 2 contains the attribute check of A2. These blocks are sent to their home application separately preserving this order. They are *parameterized* with the results of local searches, in the example by B resp. A2.

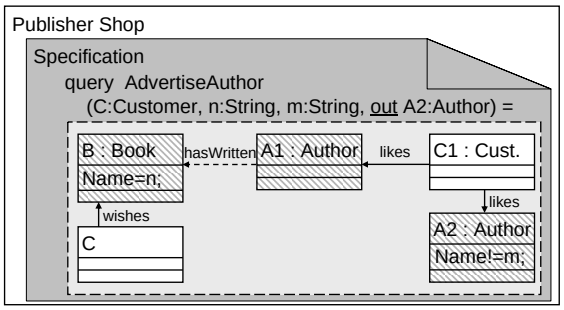


Figure 5: Graph query AdvertiseAuthor

```
linearize(Node RN, TreeSet LO){
   RO := all direct subtrees of RN rooted by remote operations concerning one HA
   LO := LO + all remaining subtrees of RN
   if (RO != ∅){
      for (i=2 to |RO|) move RO_i below RO_1
      RN := root node of RO_1
      linearize(RN, LO)
   } else{
      for (each l ∈ LO){
         insert l below RN
         RN := root node of l
         linearize(RN, ∅)
      }
   }
}
```

Listing 1: Pseudocode of `linearize`

4.3 Realization of Remote Blocks

In the following, we describe how the creation of remote blocks can be computed after generating the search tree with CodeGen2 (cf. Section 3) considering the remote offset. We developed an algorithm for creating remote blocks using the generated search tree as input. This algorithm uses the following two propositions: Let R be the set of all remote operations concerning the same home application forming one remote block. Then

1. every element of R is contained in a common subtree with root RN[6],
2. every element of R is reachable by RN by only traversing other elements of R.

We do not provide a formal proof of these propositions, but want to discuss their plausibility.

To 1: Since a remote block is connected, all elements of R are somehow dependent on each other. Due to the tree structure of the search tree, these operations must have a common root operation. This root operation must cover a boundary node. Then, the elements of R are contained in the subtrees having a child of RN as their root.

To 2: This is valid, because all remote operations inside the remote block are dependent on each other.

In the following, we present the algorithm depicted in Listing 1, which calculates the remote blocks. It rearranges all remote operations of the search tree in a chain, from which the blocks for the remote applications and their parameterization can be directly derived. The algorithm needs two sets, which are defined as follows: Let RO be the set of direct subtrees of RN, which are rooted by a remote operation concerning the same home application as RN. If RO is empty, RO is recomputed containing all direct subtrees of RN, which are rooted by a remote operation concerning the home application of one direct child of RN. Let LO be a set of subtrees within the search tree, which aggregates all direct subtrees of RN not contained in RO.

The algorithm `linearize` (cf. Listing 1) starts with the root node of the search tree as RN and $LO = \emptyset$. If $RO \neq \emptyset$, all subtrees contained in RO except of RO_1 are moved below the root node of RO_1. Then the algorithm is called with the root node of RO_1 as new parameter RN and

[6] The following abbreviations are used: RN for root node, and RO for remote operations having RN as direct predecessor.

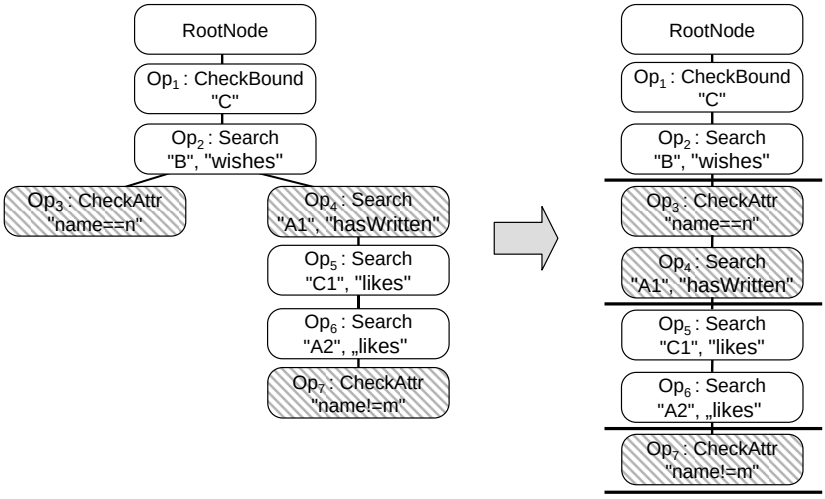


Figure 6: Search tree for query AdvertiseAuthor

LO. If $RO = \emptyset$, all subtrees contained in *LO* are inserted below *RN* and the algorithm is called recursively for all of their root nodes as *RN* and $LO := \emptyset$.

Due to proposition 1, every subtree containing a remote operation of block *R* is considered. According to proposition 2, every remote operation is considered by the algorithm. All needed rearrangements are possible, because siblings – and hence disjoint subtrees – in the search tree are independent of each other. The algorithm terminates, as every node of the search tree is used exactly once as parameter *RN*.

`linearize` does not improve the worst-case complexity of a distributed pattern matching. A worst-case scenario consists of many remote blocks each covering only one remote operation. On the other hand, a best-case scenario consists of a chain of *m* remote nodes, which are connected by *to-n*-edges, and the chain is connected by a *to-1*-edge to a local node. Without optimization, at most $n^{(m-1)}$ communication steps are necessary to determine the match of the remote block, since $m-1$ links have to be traversed and there exist *n* possible candidates for each node in the pattern. With the block creation, the whole query can be done within one communication step.

We will now illustrate the behavior of `linearize` considering the query blockExample, whose search tree is shown in Figure 6. The algorithm starts with $RN =$ RootNode and $LO = \emptyset$. Since all subtrees of *RN* start with a local operation, $RO = \emptyset$ and $LO = \{ST(\mathsf{Op_1})\}$[7]. Due to $RO = \emptyset$, `linearize` is called with $RN = \{\mathsf{Op_1}\}$ and $LO = \emptyset$. Now, *RO* and *LO* are recomputed, resulting in $RO = \emptyset$ and $LO = \{ST(\mathsf{Op_2})\}$. `linearize` is called again with $RN = \mathsf{Op_2}$ and $LO = \emptyset$. *RO* is recomputed as $\{ST(\mathsf{Op_3}), ST(\mathsf{Op_4})\}$ and *LO* remains $\emptyset$. The search tree is rearranged, i.e. $ST(\mathsf{Op_4})$ is moved below Op3. Afterwards, `linearize` is called with $RN =$ Op3 and $LO = \emptyset$. The algorithm has already computed the desired chain, and thus performs no further rearrangements of the search tree. Now it contains two chains of remote operations representing two remote blocks (cf. Subsection 4.2).

[7] $ST(\mathsf{Op_1})$ stands for subtree with root node $\mathsf{Op_1}$.

5 Conclusion

In this paper, we presented our approach for generating efficient code for distributed graph transformations. Our approach is based on existing search tree algorithms used by the code generations of PROGRES and Fujaba. The usage of search trees is advantageous for reducing the average complexity of searching graph patterns, as this search has an exponential worst-case complexity. To reduce also the communication costs arising in distributed systems, we extended the existing code generations: We have integrated a remote offset for remote operations in the cost heuristics giving priority to the execution of local operations. Additionally, we extract subpatterns from the generated search tree, which are sent to the appropriate remote applications. Thus, the remote applications are not queried for every single element of the graph pattern, decreasing the communication overhead.

As the coupled specifications export only schema parts, distributed graph transformations may be modeled, which violate certain local constraints. Thus, the execution of distributed graph transformations may lead to inconsistent host graphs within the distributed system. Therefore, we will introduce a *rule engine* as next step, which will facilitate the execution of *repair actions* [Win00]. Furthermore, this engine will be able to prevent the creation or deletion of a certain node as described in [HEET99].

Bibliography

[Böh04] Boris Böhlen. Specific graph models and their mappings to a common model. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, 2nd *International Workshop on Applications of Graph Transformations with Industrial Relevance, AGTIVE'03*, volume 3062 of *LNCS*, pages 45–60. Springer-Verlag, Heidelberg, Germany, 2004.

[FNTZ00] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story diagrams: A new graph rewrite language based on the Unified Modelling Language and Java. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, 6th *International Workshop on Theory and Application of Graph Transformations, TAGT'98*, volume 1764 of *LNCS*, pages 296–309. Springer-Verlag, Heidelberg, Germany, 2000.

[GSR05] Leif Geiger, Christian Schneider, and Carsten Reckord. Template- and modelbased code generation for MDA-tools. In Holger Giese and Albert Zündorf, editors, 3rd *International Fujaba Days*, volume tr-ri-05-259 of *Technical Report*. University of Paderborn, Germany, 2005.

[HEET99] Reiko Heckel, Hartmut Ehrig, Gregor Engels, and Gabrielle Taentzer. A view-based approach to system modeling based on open graph transformation systems. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, volume 2, pages 639–668. World Scientific, Singapore, 1999.

[RSM06] Ulrike Ranger, Erhard Schultchen, and Christof Mosler. Specifying distributed graph transformation systems. 2006. Presented at the 3rd International Workshop on Graph-Based Tools, GraBaTs'06, in Natal, Brazil.

[Sch91] Andy Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. PhD-Thesis, RWTH Aachen University, 1991.

[VVF05] Gergely Varró, Dániel Varró, and Katalin Friedl. Adaptive graph pattern matching for model transformations using model-sensitive search plans. In Gabor Karsai and Gabriele Taentzer, editors, 1st *International Workshop on Graph and Model Transformation, GraMoT'05*, volume 125 of *ENTCS*. Elsevier Science, 2005.

[Win00] Andreas Winter. *Visuelles Programmieren mit Graphtransformationen*. PhD-Thesis, RWTH Aachen University, 2000.

[Zün96] Albert Zündorf. Graph pattern matching in PROGRES. In Janice E. Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg, editors, 5th *International Workshop on Graph Grammars and Their Application to Computer Science*, volume 1073 of *LNCS*, pages 454–468. Springer-Verlag, Heidelberg, Germany, 1996.