Ensuring Consistency Within Distributed Graph Transformation Systems

Ulrike Ranger and Thorsten Hermes

RWTH Aachen University
Department of Computer Science 3 (Software Engineering)
Ahornstraße 55, 52074 Aachen, Germany
{ranger,thermes}@i3.informatik.rwth-aachen.de

Abstract. Graph transformation systems can be used for modeling the structure and the behavior of a software system in a visual way. In our project, we extend existing graph transformation systems to model and execute distributed systems. One challenge in this context is the simultaneous and correct modification of the local runtime graphs of the participating applications by visual distributed graph transformations.

As the execution of these transformations may cause inconsistencies in the local runtime graphs, we present an approach to avoid inconsistencies: A runtime mechanism translates invalid graph transformations into valid transformations. This translation is based on predefined rules describing the substitution of invalid transformation parts. Thus, new graph transformations are dynamically built at runtime. Furthermore, the runtime mechanism controls access within a distributed system.

1 Introduction

For the software development process, the use of visual modeling languages becomes more and more important. The most famous representative of such a language is the Unified Modeling Language (UML). By using different diagram types, like use-case, class and sequence diagrams, the UML supports the different phases of a software development process. These diagrams are advantageous as they serve as basic development artifacts and allow the visualization of different abstraction levels of the software system.

There are several tools enabling the drawing of UML diagrams, e.g. Rational Rose and Poseidon. They allow the generation of class templates according to UML class diagrams. Unfortunately, they do not support the generation of source code for UML behavior diagrams representing modifications on object structures.

Graph transformation systems (GTS) fill this gap, as they support the specification of static and dynamic software aspects and the generation of source code from the specification. Graphs are descriptive data structures and mathematically founded. A lot of efficient algorithms already exist for solving problems on graphs. Two representatives of GTS are PROGRES [1] and Fujaba [2], which have been used to model software system with complex data structures, e.g. process management systems, authoring tools, and systems for reverse

M.B. Dwyer and A. Lopes (Eds.): FASE 2007, LNCS 4422, pp. 368–382, 2007.

engineering. The structure of the software system is defined by a graph schema. The dynamic aspects are modeled as graph transformations, allowing the creation and modification of a runtime graph conforming to the graph schema. Both schema and transformations can be specified textually as well as visually.

Based on the specification, the GTS generates source code, from which a *visual prototype* may be created e.g. using the UPGRADE framework [3]. As UPGRADE is an extensive framework, the prototype can be configured and adapted to the user's need. With this abstraction from the runtime graph and the graph transformations, the prototype allows the developers to observe the modeled software system and its behavior from a desired view.

GTS are restricted to the modeling of local systems. Our project aims at the extension of GTS for the modeling and execution of *distributed systems*. In a distributed system, each participating application is based on a separate specification and stores its own runtime graph. Every application (acting as a *server*) defines its interface. It provides graph elements, which can be used in specifications of other applications (*clients*). A client can either call pre-defined transformations contained in interfaces or model new graph transformations visually, by using interface elements as remote objects. In this paper, we focus on the visual modeling of distributed transformations, as the textual modeling is studied in [4]. The execution of visually defined transformations modifies the client runtime graph as well as the server runtime graphs. Since an interface does not and even cannot cover *all* graph constraints of an application's specification, the execution may lead to inconsistencies in the different runtime graphs. A transformation causing inconsistencies is called an *invalid transformation*.

In [5] we described the modeling and execution of visual distributed graph transformations disregarding the mentioned inconsistencies. In this paper, we present our concepts to avoid the execution of invalid transformations by enabling the server to modify these transformations dynamically at runtime. With the use of queries on the runtime graph and predefined rules describing how invalid transformation parts are translated into valid parts, *valid graph transformations* are built from the invalid transformations. This mechanism can also be used for introducing access rights.

The paper is structured as follows: In Section 2 we introduce the general structure of a distributed system and show how such a system can be modeled with a GTS. We explain these concepts considering an example of a simplified process management system. Section 3 describes our approach for avoiding inconsistencies within the distributed system, which may be caused by the execution of invalid distributed graph transformations. We present similar approaches and compare them to our approach in Section 4. A summary and an outlook to future work is given in Section 5.

2 Specifying Distributed Systems with GTS

In this section, we introduce how distributed systems can be specified with GTS. We first describe the general architecture of a distributed system and then show

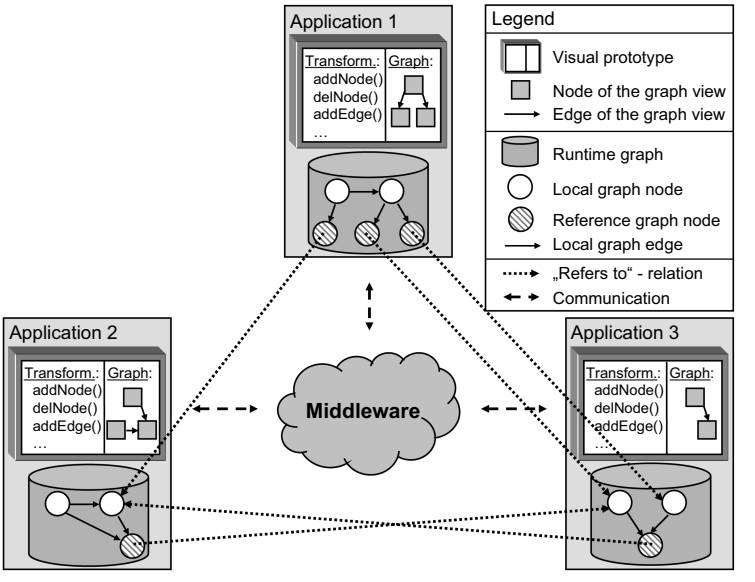

Fig. 1. Basic architecture of a distributed system

the modeling of its structure and behavior. The modeling is shown by means of an *abstract graph language* resembling the PROGRES language.

2.1 Architecture of a Distributed System

In our approach, a distributed system consists of several applications, which are all built according to the tool construction process described in section 1: (1) specifying the desired software application with the graph language of the GTS, (2) generating appropriate source code from the specification, and (3) creating a visual prototype. Figure 1 shows the basic architecture of a distributed system consisting of three applications[1]. For every application, the visual prototype and the runtime graph is depicted. The *prototype* shows the application's state, which is stored in the runtime graph. The user can call graph transformations modeled within the specification and observe their impacts on the runtime graph. The graph transformations can vary from simple graph modifications, like addNode for adding a new node of a certain type, to complex graph modifications, like produceDoc which will be described in Section 2.3.

The coupled applications perform different tasks and are used separately by different users. For coupling the applications and exchanging data between them, they require access to runtime graphs of other applications. For restricting the access to all data, every application has to specify an interface defining the nodes

[1] For sake of simplicity, we assume that all coupled applications have well-defined tasks and thus are based on different specifications. I.e. every specification is not executed more than once within a distributed system.

and edges which may be used by other applications (clients) (see Section 2.2). Our approach is general enough that an application may act as client and server simultaneously, but in this paper we restrict the applications to act either as client or as server. Instead of replicating remote nodes with their data, we use *reference nodes* in the client runtime graphs allowing the direct access on remote server nodes[2]. Reference nodes do not store any data but the location of the remote object. They are helper structures supporting the realization of relations between nodes located in different applications.

The communication between the applications, like propagating modifications to remote nodes, is done by a *middleware*. The middleware uses existing technologies like RMI or CORBA. At the moment, we use a synchronous communication model, which supports distributed transactions guaranteeing a defined and consistent state of the entire system. In the following, we will not focus on its implementation but on modeling a distributed system in a visual way. The code generation of the GTS is responsible to generate code for the middleware.

2.2 Structure of a Distributed System

In this section, we show how the static structure of a distributed system is modeled with the abstract graph language. For this purpose, we introduce the *simplified process management system* SPMS as example, which is the first distributed and extensive system we have modeled with our new concepts. The SPMS manages the tasks, documents, and resources needed for complex processes, like the development of a software system. These aspects are modeled and executed as separate applications, which have to be coupled at runtime to form a comprehensive process management system. In Figure 2 the class diagrams of the applications, the SPMS consists of, are depicted. The class diagrams represent the graph schemas[3] of the applications, in which the classes correspond to node types and associations between classes correspond to edge types.

The *Resource Manager* handles all human and computer resources needed for executing and performing the tasks of complex processes. A resource is modeled by node type R, which has an attribute rName for its name and a boolean attribute occupied indicating the activity state. Additionally, a resource can be assessed by using node type A (abbreviation for assessment). In the *Document Manager* the documents (modeled by node type D) are managed, which serve as input for tasks and are produced by tasks. The Document Manager stores all documents alphabetically ordered in a linear list using edge type nextElem. This list is needed for giving the local users of the Document Manager a clear overview of all existing documents. Furthermore, edge type basedOn models dependencies between documents. In the *Task Manager* the actual process is designed by dividing and structuring it into several smaller tasks using node type T. These tasks have to be executed in a specific order determined by edges of type nextTask.

In the SPMS, the Task Manager is used for coupling the applications, although another application could have also been used. As the applications are developed

[2] In contrast to reference nodes, we do not store reference edges as they are of no use.
[3] We use directed node- and edge-labeled graphs, in which nodes may be attributed.

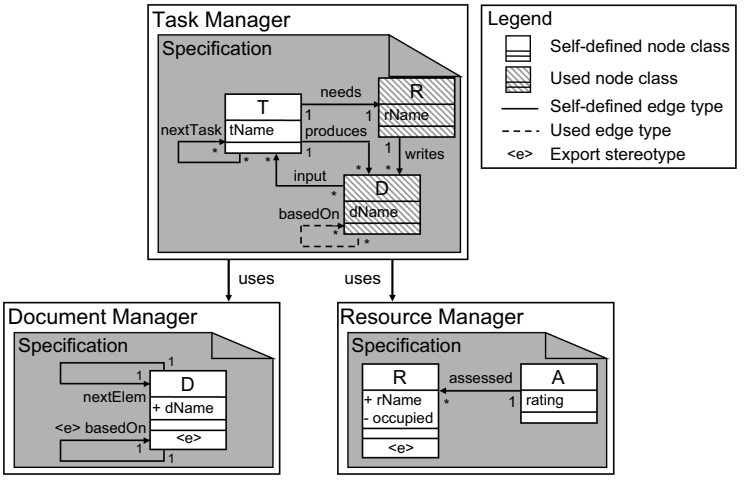

Fig. 2. Static structure of the SPMS

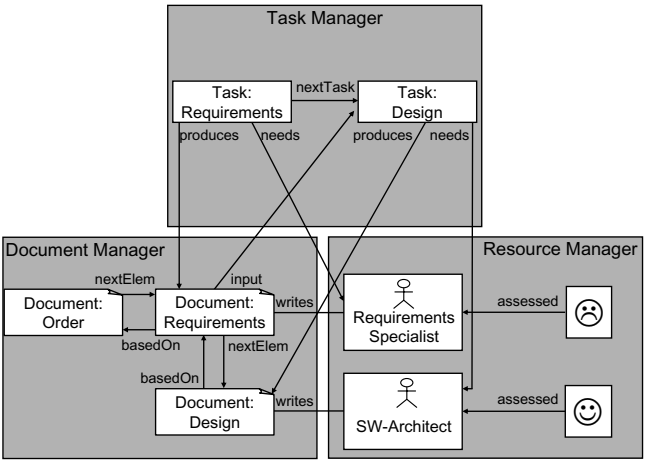

Fig. 3. Sample runtime-graph of the SPMS

separately, the Task Manager does not have any knowledge about the other applications. Therefore, the Resource Manager and the Document Manager define *interfaces* containing the types, which may be used by the Task Manager. The interfaces are not separated from the implementations, but implicitly defined by marking the respective visual elements with the <e>-stereotype (including the public attributes of node types). The interface of the Resource Manager consists of node type R and its attribute rName. Node type D, its attribute dName and the edge type basedOn compose the interface of the Document Manager.

Interface elements are *read only* and thus must not be changed by a client, e.g. by adding an attribute to a node type. This fact is emphasized in the specification

by illustrating the used graph elements by striped rectangles and dashed arrows. To integrate interface elements, the used graph elements and the self-defined elements can be related by defining edge types between them. For example, if a resource performs a certain task, this is modeled by an edge type needs in the Task Manager relating the self-defined node type T and the used node type R (see Figure 2). In that way, the interrelations between the different applications can be modeled using local edges.

Figure 3 shows an example of a SPMS runtime graph, which is a possible instantiation of the model (graph schema) depicted in Figure 2. The Task Manager has several tasks for the development of a software system, which refer to documents and resources in the other applications.

2.3 Modeling the Behavior

Based on the static structure of the SPMS shown in Figure 2, the behavior of the distributed system can be specified by graph transformations. In this paper, we focus on the *visual specification of distributed graph transformations*.

Basically, a visual graph transformation[4] consists of a left-hand side (LHS) and a right-hand side (RHS). The LHS defines a *graph pattern*, which is searched for in the runtime graph. A sub-graph in the runtime graph conforming to the LHS is called *match*. If several matches are found, one of them is chosen non-deterministically. The RHS of the transformation defines the modifications of the match, e.g. creating nodes. For the definition of visual transformations, graph languages offer an expressive variety of language constructs. The graph languages provide also the definition of *graph queries*, which only search for a match.

One essential advantage of specifying graph transformations visually is the modeling of the behavior in a *declarative* way, i.e. the modeling of *what* the transformation does instead of *how* the specified modifications can be achieved. In distributed graph transformations, the specified behavior lead to the simultaneous modification of several runtime graphs belonging to different applications. Distributed graph transformations are executed as *transactions*, i.e. either all modifications or no modification at all are performed, fulfilling the ACID properties known from databases. As the syntax and semantics of distributed transformations are described in [5], we only present a simple example here.

Figure 4 shows the distributed graph transformation produceDoc for producing a new document by a task. This transformation is specified in the Task Manager, but its execution affects also the state of the Document Manager. A task t and a document d, representing a remote node in the Document Manager, are given as input parameters. The LHS consists of these nodes and an edge of type input incident to them. Furthermore, it contains a resource r referencing a node in the Resource Manager, which is needed by the task t. On the RHS, all nodes and edges of the LHS are preserved and additionally a new document nd is created. For integrating the new document nd in the existing graphs, a writes-edge is created connecting r and nd showing that nd is written by r. Furthermore, a

[4] In some GTS approaches, a visual graph transformation is called a *production* or *rule*.

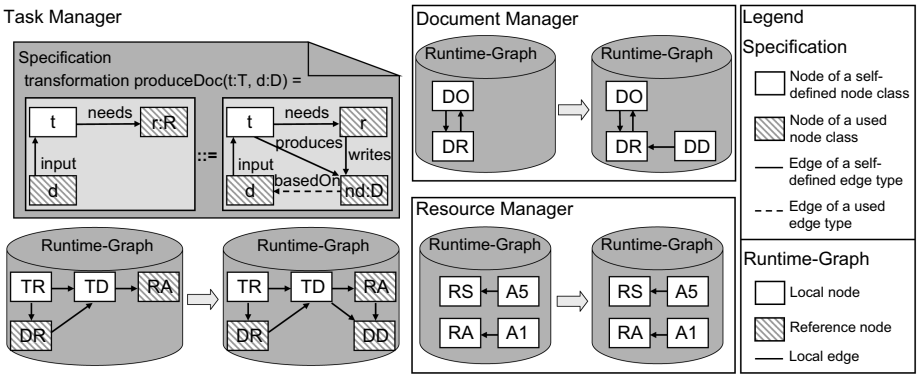

Fig. 4. Transformation for producing a new document

produces-edge incident to task t and the document nd is created. As nd is based on d, a edge of type basedOn is created connecting both documents.

When executing produceDoc, the runtime graphs of the Task Manager and the Document Manager are modified, as node and edge types of both are used in the transformation. These modifications are depicted in the runtime graphs in Figure 4 affecting local and reference nodes. For example, in the Document Manager a new local node with id DD for the new document nd in the transformation is created, while in the Task Manager only a reference node pointing to node DD is created. This behavior is founded in the fact that node type D is specified in the Document Manager and the Task Manager acts only as client for this type. Thus, every node of type D logically belongs to the Document Manager and coupled applications may only have references on these nodes. The resulting runtime graphs depicted in Figure 4 correspond to the SPMS state shown in Figure 3 (for lack of space only the initials of nodes are shown in Figure 4).

2.4 Execution of Distributed Transformations

The search of a graph pattern has an exponential worst-case complexity and becomes even more cost-intensive when concerning different applications. [6] presents an approach to divide the LHS specified in a client into several sub-patterns, each affecting exactly one application. The sub-patterns are sent to the server applications at runtime using GTXL [7], instead of querying the applications for every single pattern element. The server applications respond with appropriate matches, thus reducing the communication costs. After determining the match for the LHS in the client, the modifications are performed according to the RHS. We use a similar mechanism for them: We divide the distributed graph transformation of the client into several sub-transformations, each affecting one application. The sub-transformations are sent to the server applications using GTXL and are then executed. For example, the transformation on the left in Figure 5 is a sub-transformation of produceDoc sent to the Document Manager.

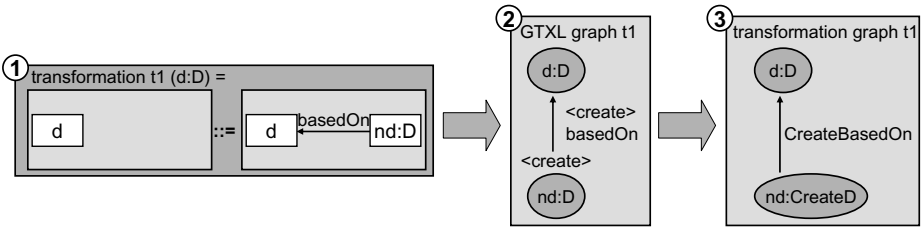

Fig. 5. Representations of a visual graph transformation

GTXL provides a XML-based format for exchanging graphs and transformations. The structure of a transformation in GTXL may be regarded as graph, in which the nodes and edges are marked with stereotypes describing their modification. Figure 5 shows an example of a graph transformation ① and its abstract graph representation in GTXL ② (③ will be explained in Section 3).

When executing a sub-transformation, inconsistencies in the runtime graph of the server application may occur. These can be caused by *create operations*, which insert new nodes and edges in the server's runtime graph, *delete operations*, destroying nodes and edges of a server, and *attribute operations*, which change the attribute values of server objects. All these operations transform the server runtime graph without considering its internal constraints. For example, when executing transformation produceDoc depicted in Figure 4, a new document is created in the Document Manager. As the Task Manager does not know that the Document Manger stores all documents in a linear list, he has not specified the insertion of the new document in the list. This causes local inconsistencies, because the linear list does not contain all documents of the Document Manager, but it relies on a consistent list structure. This problem is called *graph rewriting dilemma* [8]: The interfaces have to provide node and edge types, which are visually available in client specifications, but due to data abstraction the interfaces do not cover entire graph schemas with all specification constraints. Since this is an important aspect in software engineering, we present an approach for preserving information hiding and solving the graph rewriting dilemma.

3 Meta-transformations

In this section, we describe a mechanism to deal with invalid graph transformations specified in a client application: The server dynamically translates them into valid graph transformations, which also update the *internal data* hidden by the interface. The translation mechanism is invoked by the server application for every incoming query or transformation received from a client (via GTXL). This mechanism can also be used for access control (e.g. as described in [9]).

3.1 The Meta-transformation Approach

Since we use graph transformations to describe the applications's behavior, it is only natural to use the same approach for ensuring consistency. As illustrated

in Figure 5, an incoming graph transformation[5] can be viewed as a graph. The server first translates the incoming GTXL graph ② into a transformation graph ③. This graph stores the actions to be performed in the type information of each element instead of stereotypes. It is based on a *transformation graph schema* derived from the application schema: For every possible action and type in the application schema, a combined type is generated, e.g. type CreateD for creating an instance of type D. Nodes in the transformation graph are called *operations*.

The transformation graph is then transformed using a pre-defined set of rules, which we call *meta-transformations*. These are transformations that operate on transformations represented by transformation graphs. They are written by the specifier of the server application without knowledge of the transformations that might be modeled by clients. Their application is performed at runtime, when a transformation is received. Each meta-transformation deals with a single aspect of consistency or access control, and does not have to match the entire incoming transformation. Since a big transformation may be matched by multiple meta-transformations, the specifier defines a total order over the meta-transformations. Each meta-transformation works on the intermediate result of previous ones. The final result is a transformation graph that represents a valid transformation, which is then executed by the application.

We require that either at least the changes specified in the incoming transformation are performed (*minimal semantics*), or that no changes are performed at all. In the later case an error is reported to the client and the distributed transformation is aborted. This ensures that distributed transformations are predictable from the client's perspective. There are two types of meta-transformations:

Simple Meta-Transformations. The only difference to regular graph transformations is that simple meta-transformations are not defined over the application's graph schema, but the graph transformation schema described above. If a match for a LHS is found in the incoming transformation, the meta-transformation is applied, either once or to every match, as defined by the specifier.

Complex Meta-Transformations. In order to deal with large incoming transformations and perform sophisticated checks, complex meta-transformations use the control structures of the graph language to combine queries and simple meta-transformations (equivalent to PROGRES transactions). For their notation, we will use the visual flow notation of Fujaba in Section 3.2. Complex meta-transformations do not have a LHS or RHS, and are invoked for every incoming transformation. In addition to queries on the transformation graph, we provide runtime graph queries (specified using the application's graph schema). This allows reactions depending on the current runtime graph, e.g. rejection of a transformation that attempts to create a document if a document with that name already exists. The control flow together with the queries can be used to simulate advanced features like *negative application conditions* [10].

[5] We only discuss graph transformations, but the same mechanism applies to queries.

3.2 Examples

Maintaining an Ordered List. The Document Manager maintains a linked list of all stored documents, ordered by their dName attribute. This implementation detail is hidden, so any document created by the Task Manager will have to be inserted into this list by the Document Manager itself.

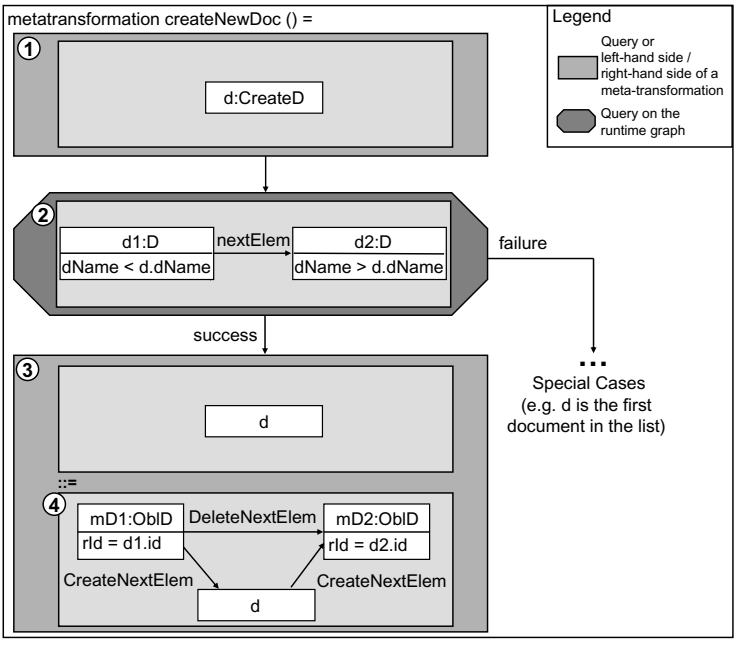

Fig. 6. Meta-transformation to maintain ordered linked list

The complex meta-transformation in Figure 6 begins with a query on the transformation graph ① that is matched when an incoming transformation attempts to create a document. The runtime graph query ② is then executed to find two documents between which the new document should be inserted. If a suitable position is found, the simple meta-transformation ③ is invoked. It modifies the incoming transformation so that it does not only create the document d, but also creates appropriate edges to d1 and d2 and deletes the existing edge between them. Figure 7 shows the effect of createNewDoc() on the transformation from Figure 5, translated into the standard transformation notation.

We use attributes to relate runtime graph elements with the operations on them, e.g. to ensure that we create edges to the same documents we found in the runtime query. The attribute id for documents in the runtime graph is used to access the internal identifier assigned by the GTS. On the transformation graph level, operations that affect a specific document store its identifier in the attribute rId. Using attribute assignments in ④, we add OblD operations for d1 and d2, meaning the corresponding documents will appear on both LHS and RHS

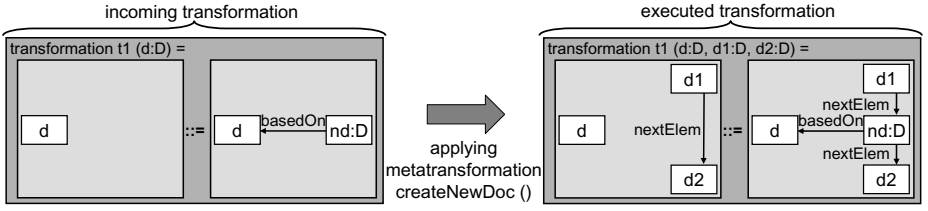

Fig. 7. Result of applying the meta-transformation createNewDoc()

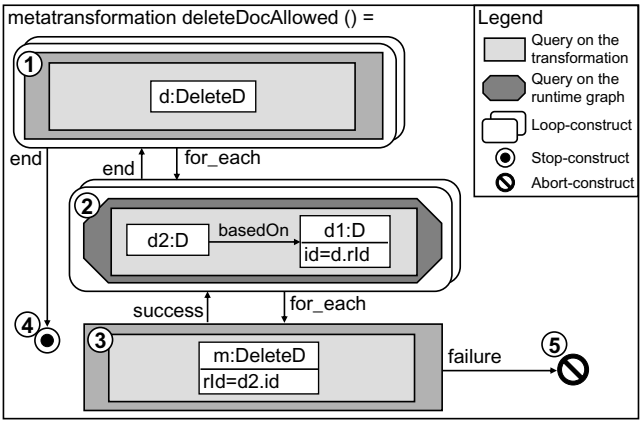

Fig. 8. Meta-transformation preventing deletion of documents that are still referenced

of the transformation. The failure branch for handling special cases is omitted due to space limitations. Usually, we would perform the initial query ① inside a loop (see next example) to deal with incoming transformations creating more than one document.

Access Control. Documents in the Document Manager must not be deleted while there are other documents based on them. In this case, we ensure consistency not through translation, but rejection of incoming transformations.

This access control is realized by the complex meta-transformation in Figure 8. The initial query ① matches the deletion of a document d in the incoming transformation. Its enclosing for_each loop iterates through all matches for a document deletion operation. For every such document d1, a runtime graph query ② searches for a document d2 based on it. d1 is identified using the rId from the corresponding deletion command d. Again, the for_each loop enclosing ② ensures that we process all possible matches for d2. We perform a query ③ on the transformation graph to check whether this document will also be deleted by the incoming transformation. If not, this would cause an inconsistency, and thus the failure branch leads to a special symbol ⑤, which results in the rejection of the incoming transformation. If no consistency violation is detected, the loop ends with ④, and processing of this meta-transformation is finished.

3.3 Evaluation

Existing GTS, like PROGRES, address some concerns of meta-transformations by other mechanisms. We will compare them to our approach and give some estimates regarding runtime and specification complexity.

Constraints and Repair Actions. Constraints specify invariants on the runtime graph that are verified after every transformation. If a violation is detected, processing is halted. This is not acceptable in a distributed system where the client specifier cannot keep the server graph consistent because of a limited interface. Repair actions [8] are an extension of constraints. Instead of terminating the application, a transformation may be triggered to return the graph to a consistent state. This is different from the meta-transformation approach, where inconsistencies are not allowed to occur in the first place. While this approach allows repair actions to perform iterative graph manipulation, they suffer from several disadvantages: First, they have less available information, e.g. when deleting a node, a meta-transformation can still evaluate its attributes and incident edges before the node is deleted. A repair action is invoked after the deletion, and thus all information about the node is lost. Second, repair actions are not independent: The specifier of a repair action must assume that every repair action is the first action performed on a runtime graph with multiple inconsistencies, making it hard to correctly perform the repairs. Third, constraints and repair actions require the whole graph to be searched for inconsistency patterns after every modification, meaning their execution time grows with the graph size.

Runtime Complexity of Meta-Transformations. Meta-transformations operate on the transformation graph, which typically contains only few elements. For runtime graph queries, knowledge of the incoming transformation can be used to avoid global searches on the runtime graph. Only when this is not possible, these queries exhibit the same complexity as constraints/repair actions. This means that, except for these worst cases, the runtime complexity of meta-transformation does not depend on the size of the runtime graph, making them much more scalable.

Number of Meta-Transformations. To cover all possible operations (creation, deletion, and, for nodes only, attribute modification), $3 * n + 2 * e$ meta-transformations have to be specified, where n is the number of node types and e the number of edge types in the interface. Operations that require neither consistency enforcement nor access control may be omitted. We are currently investigating possibilities to support the specifier in this task.

4 Related Work

The presented meta-transformations may not be confused with the *meta transformation rules* introduced in [11]. They translate generic graph transformations

applying type variables into several concrete transformations without type variables at specification time. At runtime, the concrete transformations are executed instead of the generic transformations, improving the performance. These rules can also be used for increasing the maintainability of transformations. In contrast to meta-transformations, the meta rules of [11] operate on static transformations known at specification time, and are performed at specification time.

In [12], the GTS Fujaba is extended by defining the life-cycle of software components and their interactions. This is modeled by using existing diagrams and introducing new diagram types in Fujaba, which e.g. offer the specification of deploying components located on different machines. Furthermore, fault tolerance in case of hardware failures is considered. For the interaction of components, distributed graph transformations are needed. In contrast to our approach, no means for modeling the distributed transformations in a visual way is provided, thus [12] is more related to the textual specification presented in [4]. As only textual interfaces and calls of remote procedures are used, the graph rewriting dilemma as presented in Section 2.4 does not arise in [12].

In context of GTS, many projects deal with the integration of different models, e.g. by using triple graph grammars [13]. Most of these approaches use only one specification containing the different models and one runtime graph. They focus on synchronizing the different models instead of modeling distributed systems as presented in this paper. Furthermore, they do not concentrate on the abstraction of implementation details, as provided by specification interfaces.

For modeling distributed systems, [14] introduces *hierarchically distributed graph transformations* for GTS. Basically, a network graph defines the topology and the relations between applications of a distributed system. Each application is represented as a node in the network graph and in turn stores its runtime state within a local graph. Instead of using references as described in Section 2.1, objects are replicated in the applications. Distributed transformations are modeled on the level of the network graph and not on application level, making the applications *passive* components within the distributed system. Another difference between the two approaches is the level of data abstraction: We provide abstraction on the graph schema level, while [14] offers the more fine-granular abstraction on object level. Thus, for every object shared by two applications an explicit relation has to be defined, which leads to the extensive definition of object relations. However, the approach is only rudimentarily implemented and does not offer means for solving the graph rewriting dilemma.

5 Conclusion

Our project extends existing GTS by modeling structure and behavior of a distributed system in a visual way. To achieve information hiding, applications of a distributed system only exchange interfaces. As a drawback, distributed graph transformations specified in a client cannot consider all constraints internally imposed by the servers. Since this may lead to inconsistencies within the runtime

graphs, we introduce meta-transformations for translating invalid transformations into valid transformations dynamically.

Meta-transformations are based on existing concepts for visual graph transformations, but operate on graph transformations themselves instead of modifying runtime graphs. As incoming transformations are not known at specification time, the server specifier defines a number of *translation rules*, which are applied by a runtime mechanism. This also allows queries on the runtime graph, so that the local state may be taken into account when performing the translation. Meta-transformations provide a general mechanism, which can also be used for realizing access control within a distributed system.

The graph rewriting dilemma [8] has been originally described for local systems, which are built from different *sub-systems* using one common graph. Meta-transformations can be used for ensuring the consistency of all sub-systems in this local case as well. As all graph transformations which will be performed on the sub-systems are known at specification time, meta-transformations can be applied in a pre-processing step just before generating code for the system. Thus, the runtime infrastructure is not needed for local systems.

We have tested our concepts on a large distributed process management system and they have shown to be suitable for solving the graph rewriting dilemma. As a next step, we will formalize the presented concepts and integrate them into the existing formalism of PROGRES. Additionally, we are analyzing concepts for improving the interface mechanism presented in Section 2.2 by introducing *virtual graph views*. They define an explicit view on an application instead of implicitly determine the interface by a proper part of the graph schema. For the realization, meta-transformations can be used for translating queries conforming to the virtual graph view into queries conforming to the internal structure.

References

1. Schürr, A.: Operationales Spezifizieren mit programmierten Graphersetzungssystemen. PhD-Thesis, RWTH Aachen University (1991)
2. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: A new graph rewrite language based on the Unified Modelling Language and Java. In Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: 6^{th} International Workshop on Theory and Application of Graph Transformations, TAGT'98. Volume 1764 of LNCS., Springer-Verlag, Heidelberg, Germany (2000) 296–309
3. Böhlen, B., Jäger, D., Schleicher, A., Westfechtel, B.: UPGRADE: A framework for building graph-based interactive tools. In Mens, T., Schürr, A., Taentzer, G., eds.: 1^{st} International Workshop on Graph-Based Tools, GraBaTs'02. Volume 72 of ENTCS., Elsevier Science Publishers (2002)
4. Böhlen, B., Ranger, U.: Concepts for specifying complex graph transformation systems. In Ehrig, H., Engels, G., Parisi-Presicce, F., eds.: 2^{nd} International Conference on Graph Transformations, ICGT'04. Volume 3256 of LNCS., Springer-Verlag, Heidelberg, Germany (2004) 96–111
5. Ranger, U., Schultchen, E., Mosler, C.: Specifying distributed graph transformation systems. (2006) , presented at the 3^{rd} International Workshop on Graph-Based Tools, GraBaTs'06.

6. Ranger, U., Lüstraeten, M.: Search trees for distributed graph transformation systems. In Karsai, G., Taentzer, G., eds.: 2^{nd} International Workshop on Graph and Model Transformation, GraMoT'06. Volume 4 of Electronic Communications of the EASST., European Association of Software Science and Technology (2006) (to appear).
7. Taentzer, G.: Towards common exchange formats for graphs and graph transformation systems. In Ehrig, H., Ermel, C., Padberg, J., eds.: 1^{st} International Workshop on Uniform Approaches to Graphical Process Specification Techniques, UNIGRA'01. Volume 44(4) of ENTCS., Elsevier Science Publishers (2001)
8. Winter, A.: Visuelles Programmieren mit Graphtransformationen. PhD-Thesis, RWTH Aachen University (2000)
9. Heckel, R., Ehrig, H., Engels, G., Taentzer, G.: A view-based approach to system modeling based on open graph transformation systems. In Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools. Volume 2. World Scientific, Singapore (1999) 639–668
10. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundamenta Informaticae **26** (1996) 287–313
11. Varró, D., Pataricza, A.: Generic and meta-transformations for model transformation engineering. In Baar, T., Strohmeier, A., Moreira, A., Mellor, S., eds.: 7^{th} International Conference on the Unified Modeling Language, UML'04. Volume 3273 of LNCS., Springer-Verlag, Heidelberg, Germany (2004) 290–304
12. Tichy, M.: Durchgängige Unterstützung für Entwurf, Implementierung und Betrieb von Komponenten in offenen Softwarearchitekturen mittels UML. Diploma Thesis, University of Paderborn (2002)
13. Schürr, A.: Specification of graph translators with triple graph grammars. In Mayr, E.W., Schmidt, G., Tinhofer, G., eds.: 20^{th} International Workshop on Graph-Theoretic Concepts in Computer Science, WG'94. Volume 903 of LNCS., Springer-Verlag, Heidelberg, Germany (1995) 151–163
14. Fischer, I., Koch, M., Taentzer, G.: Visual design of distributed object systems by graph transformation. Technical Report 98-15, Tech. University of Berlin (1998)