Tool Integration with Triple Graph Grammars - A Survey

Alexander Königs, Andy Schürr

Real-Time Systems Lab
Darmstadt University of Technology
D-64283 Darmstadt, Germany

Abstract

Nowadays, typical software and system engineering projects in various industrial sectors (automotive, telecommunication, etc.) involve hundreds of developers using quite a number of different tools. Thus, the data of a project as a whole is distributed over these tools. Therefore, it is necessary to make the relationships of different tool data repositories visible and keep them consistent with each other. This still is a nightmare due to the lack of domain-specific adaptable tool and data integration solutions which support maintenance of traceability links, semi-automatic consistency checking as well as update propagation. Currently used solutions are usually hand-coded one-way transformations between pairs of tools. In this article we present a rule-based approach that allows for the declarative specification of data integration rules. It is based on the formalism of triple graph grammars and uses directed graphs to represent MOF-compliant (meta) models. As a result we give an answer to OMG’s request for proposals for a MOF-compliant “queries, views, and transformation” (QVT) approach from the “model driven application development” (MDA) field.

Keywords: tool integration, model integration, triple graph grammars, QVT, MDA

1 Introduction

Development processes of complex system engineering projects nowadays typically involve hundreds of geographically distributed developers. Commonly used process models (e.g. waterfall model, Rational Unified Process, V-model) subdivide these processes into distinct interrelated development phases or workflows. Developers are assigned to a specific phase or workflow; they often

1 Email: {koenigs|schuerr}@es.tu-darmstadt.de
Fig. 1. Different tools in one specific software system engineering process

make use of CASE tools which are specialized in supporting a specific task and manipulating specific types of documents. Figure 1 depicts a small example of this kind, a subset of a tool chain that is used in various automotive systems engineering projects. The functional and non-functional requirements of a system are stored in the data-based requirements tool Doors. These requirements ask for software functionality which is specified and modeled in Matlab and Together. The hardware design of the system is done with HDL Author. Catia is used for computer-aided design, engineering, and manufacturing. Functional tests are specified using CTE XL. Finally, the data of the entire project is managed with Windchill.

Thus, the development documents of a project as a whole are usually distributed over proprietary data repositories of different tools. As the developers work on these separated data repositories that evolve concurrently, they face the risk of working on increasingly inconsistent sets of documents. Therefore, tool support is urgently needed to keep the data of separate tool repositories in a consistent state, taking sometimes rather domain-specific integrity constraints into account. This kind of tool/data integration support has been a “hot research topic” for about 15 years now and is still not addressed satisfactorily.

One popular integration approach introduced in the past was based on so-called message servers as used in commercial products like HP-Soft-Bench [16] or ToolTalk [15]. These systems offered basic control- and event-oriented [4] integration mechanisms, but did not allow for the specification of functional dependencies between documents. A more data-oriented approach was based on so-called compatibility maps [17]. This approach worked on a common database, but provided only poor support for the specification of functional document dependencies. Other approaches introduced at about the same time modeled whole project databases as attributed syntax trees and implemented consistency checks as attribute evaluation rules. For instance Mercury [21]
offered some support for attribute value propagation from one document to another one. Gandalf [19] and Centaur [3] used a more sophisticated version of the concept of attributed syntax trees and supported arbitrarily nested documents. The main disadvantage of all these approaches was the lack of any tool support for consistency recovering data updates. To overcome this deficiency of standard attribute grammars based solutions attribute coupled grammars [18] have been introduced. They use attribute evaluation rules of one syntax tree to construct another syntax tree, i.e. another document. A severe drawback of attribute coupled grammars as well as of the presented data integration approaches so far is that they are batch-oriented and unidirectional. They translate a source document into a target document, but not vice-versa. TransformGen [46] was a first attempt to regard source and target documents in a bidirectional way. TransformGen semi-automatically generated unidirectional and batch-oriented transformation tools for both directions using a single specification of related string grammars as input. This idea of relating grammars for different string/tree languages to graph languages has been generalized and builds the basis for the integration approaches presented in this paper. The initial work in this area has been done by Pratt, who introduced the concept of pair grammars [38] already many years ago.

Recently, the tool/data integration problem has been re-addressed, but not yet solved, by the Object Management Group (OMG) and its Request For Proposals (RFP) for a MOF-compliant “queries, view, and transformation” standard (QVT) [32]. The RFP requires first of all that OMG’s Meta-Object Facility (MOF) [33] is used as a basis. MOF is a standard modeling language specification for “defining, manipulating, and integrating meta-data and data in a platform independent manner”. In this paper we give a survey of a tool/data integration approach based on MOF and so-called triple graph grammars with regard to the QVT-RFP. Triple graph grammars (TGGs) have been invented as a stand-alone declarative model integration formalism 10 years ago [43]. It is topic of ongoing research activities to adopt TGGs to the world of OMG standards [22]. It is the purpose of this paper to summarize the basic ideas of the TGG-based tool integration approach using the terminology of the OMG standard MOF and to relate this presentation to the previously published formal definition of a rather simple form of TGGs. Thus, this paper is a combination of excerpts from [22] and [43]. TGGs have been adopted for migrating relational to object oriented database systems [20], for tool integration in the IPSEN context [25], and for diagram consistency management in the FUJABA project for instance [51].

For demonstration purposes we will use a running example that is related to a real-world automotive project. The project deals with the development of
a windscreen wiper with a rain sensor. The tool integration scenario selected for this paper involves the tools Doors from Telelogic [49] for the definition of system requirements in natural language, and Together from Borland [6] for the visual definition of use cases and class diagrams from the Unified Modeling Language (UML) \(^2\) [35].

Figure 2 shows parts of the data\(^3\) kept in Doors and Together. Figure 2a is a screenshot of Doors and shows a small cut-out of the windscreen wiper’s system requirements. First of all some enumeration types are introduced which are used at various places to distinguish between different operating modes of our system. Furthermore, a hierarchy of subsections helps to organize the system requirements as related groups and subgroups of so-called features. Each feature description has a structure adopted from use case driven requirements engineering approaches. Among other things they distinguish between activation triggers, preconditions and postconditions, a more or less detailed description of the regarded system feature, and so forth. Figure 2b presents a use case diagram created with Together that visualizes our requirements. It graphically depicts the relationships between system features (use cases) and stakeholders (actors) as well as interactions between different features in the form of use case dependencies. Furthermore, it shows that we have met the decision that the hierarchical relationships between the feature group 1.3.1 Wiping and its subfeatures 1.3.1.1 Drop-arm switch wiping and 1.3.1.2 Wiping for washing do not represent a decomposition of a complex feature into interacting subfeatures, but a kind of refinement relationship. Drop-arm switch wiping and Wiping for washing are two different operating modes of the windscreen wiper which “inherit” some pre- and postconditions as well as invariants from the common supermode Wiping. Therefore, the just regarded feature hierarchy in Figure 2a is translated into a generalization hierarchy in Figure 2b instead of mapping it onto a hierarchy of packages containing use case diagrams. This is a good example for the case, where the user of a tool integration solution has to choose between different options how to translate a submodel created by one tool into a related submodel of another tool.

Finally, Figure 2c shows a class diagram that represents a first rather naive system design based on the use case diagram of Figure 2b and the system fea-

\(^2\) The data integration rules presented in this paper have been invented for demonstration purposes only. Rules used in practice are more complex, but do not systematically make use of all available features of our model integration formalism. They are, therefore, less appropriate for the purpose of explaining our approach.

\(^3\) In the context of tools we use the terms documents and data. In the context of metamodeling we correspondingly talk about models and objects. Finally, we use the terms graphs and nodes in the context of graph grammars.
Fig. 2. Example requirements, use case, and class diagram documents
ture definitions of Figure 2a. In this case, we made the decision to introduce an interface wrapper class for any (sensor/actuator) actor of the system and to implement each feature in the form of a separate class. Sensors send their data to the corresponding feature classes, whereas feature classes compute and propagate the needed data to control all actuators. Furthermore, it is rather obvious to translate generalization relationships between use cases into generalizationships between classes, whereas it is a matter of debate whether the translation of extend and includes dependencies into aggregation of classes is the most obvious solution. It is the topic of related ongoing research activities to come up with sets of more sophisticated domain-specific model mapping rules which support coevolution of system requirements and architectures.

In chapter 2 we explain our document and tool integration approach as presented in [22]. We introduce meta-modeling as a well-known technique for describing the structure of documents. We add the concept of triple graph grammars in order to write down rules which declaratively describe correspondences between documents in order to integrate them. From these declarative rules we derive operational rules in chapter 3. These operational rules can be applied in order to fulfill rather different integration tasks. In chapter 4 we recall the formalism of triple graph grammars as presented in [43]. We continue in chapter 5 by presenting OMG’s QVT request for proposals as a possibility to classify document and tool integration approaches. On this basis, we introduce related approaches, and compare them with each other as well as with our own approach. Finally, chapter 6 summarizes and concludes our paper, discusses open issues and future work.

2 Presentation of our approach

In this chapter we present our tool/data integration approach that allows for the declarative specification of consistency checking and recovering integration rules. We start by introducing meta-modeling as a well-known technique to specify the abstract syntax and static semantics of data kept in software system engineering tools. Furthermore, we use meta-modeling for declaring correspondence link types between the data repositories of different tools. Finally, we introduce the idea of triple graph grammars for specifying declarative data integration rules. We will see later on that these triple graph grammar rules may automatically be translated into different sets of operational model transformation rules which are then responsible for checking consistency of meta-models and propagating changes between related meta-models in all directions.
2.1 Meta-modeling

Meta-modeling is a well-known technique for defining the abstract syntax and static semantics of data kept in software system engineering tools [45]. We use MOF version 2.0 [33] which is OMG’s standard language specification for this purpose. Figure 3 shows a part of one MOF 2.0-compliant meta-model of UML use case diagrams as defined in Together. This meta-model is not a cut-out of the “real” UML meta-model for use case diagrams for the following reasons: First of all the real meta-model is too complex to serve as a running example for this paper. Furthermore, it offers (of course) no means to represent the diagram elements of UML tools, which are often (mis-) used to cluster logically related modeling elements and should, therefore, not be neglected when data integration rules are defined.

The meta-model of Figure 3 states that a UML use case diagram in Together consists of Diagrams that are the leaves of a Package hierarchy. Each Diagram owns a number of UseCases. Furthermore, UseCases can be related to each other by Generalization relationships. Having such a meta-model for a document we can represent the data contained in the document as a UML object diagram which conforms to the meta-model. Figure 4 shows a part of the UML object diagram representation of the UML use case diagram

---

4 Extend and Include relationships as well as Actors are not used in the following examples and, therefore, omitted here.
document of our running example (cf. Figure 2b). Accordingly, we can write a meta-model for requirements documents stored in Doors.

![Diagram]

Fig. 4. Part of the UML object diagram representation of the UML use case diagram

The basic idea of our tool/data integration approach is to create and maintain correspondence links (traceability links) between elements of the different tools’ data repositories. Using the meta-models of the tools that we want to integrate with each other we can write a meta-model for the correspondence links as well. This meta-model maps elements from the meta-model of one tool to elements of the meta-model of the other tool and vice-versa. Figure 5 depicts the three meta-models that we need for declaring correspondence links in our windscreen wiper project.

Figure 5a shows the meta-model for requirements documents. Figure 5c recalls the meta-model for use case diagrams. Finally, Figure 5b uses both tools’ meta-models in order to declare the needed correspondence link types. The imported classes from the to be integrated meta-models defined in package RequirementsDocument and UseCaseDocument are rendered in gray in the package IntegrationSchema.

In our approach we demand that to be integrated meta-models are organized as shown in Figure 6. The meta-model of each tool is defined in a separate package which may contain subpackages if needed. Each of these packages may contain any MOF 2.0-compliant meta-model describing the data structure of a tool. Additionally, we have a third hierarchy of packages marked with the stereotype Integration. These packages contain the meta-model of the needed correspondence links. We demand that each correspondence link type declaration must match the one depicted in Figure 7. The declared link type may have an arbitrary name and must be marked with the stereotype Integration. This stereotype indicates the fact that we currently (mis-)use UML classes stereotyped with Integration to define MOF 2.0 associations, when we use a standard UML 1.x CASE tool for the definition of meta-models.
The implementation of a proper MOF 2.0 editor with direct support for all meta-modeling concepts is almost completed as a new plugin for the UML CASE tool framework FUJABA [30]. It will be used in the future for meta-modeling purposes and offer more sophisticated support for organizing meta-models in packages, specializing associations, etc.

Any (MOF) association like the one depicted in Figure 7 has exactly two ends. The ends point to the corresponding elements from the tools’ meta-models. Each end may carry a multiplicity. As usual these multiplicities specify how many elements from tool B correspond to one element from tool A and vice versa. If no multiplicity is provided for an association end we assume the multiplicity 1 as default. These multiplicities are later on used to check consistency and completeness of (semi-)automatically created correspondence links. In our running example the implicitly defined multiplicity constraints require e.g. that all requirement document instances of class FeatureGroup, Feature, and Dependency are mapped onto one and only one instance of the corresponding classes Package, UseCase, and Relationship of a use case document.
Additionally, a link type declaration may inherit from another link type declaration such that we are able to construct modeling-domain specific hierarchies of correspondence link types. Please notice that MOF 2.0 now offers the therefore required means for generalization and refinement of associations which are not discussed in this paper. They are of great importance for the definition of reusable and adaptable meta-model integration specifications. For a discussion of still open problems with the semantics of generalization and refinement (redefinition and subsetting) of MOF 2.0 associations the reader is referred to [1].

2.2 Specification of integration rules

Having declared the types of the correspondence links we now need a language for specifying the conditions that must hold in order to consider a correspondence link as consistent. One obvious possibility is to use a textual language like the Object Constraint Language (OCL) [34] for that purpose. For example, if we want to state that a correspondence link of type FeatureGroupPackageRelation is consistent if the name of the attached FeatureGroup with the identifier \( fg \) is the same as the name of the attached Package with the identifier \( p \), we would write something like \( fg.\text{name} = p.\text{name} \). This is straightforward and easy to understand. Let us now regard a second example. We would like to state that a correspondence link of the type FeatureUseCaseRelation is consistent if the name of the attached Feature with the identifier \( f \) is the same as the name of the attached UseCase with the identifier \( uc \) and \( f \) is contained in a FeatureGroup \( fg \) which is linked to a Package \( p \) by a correspondence link of type FeatureGroupPackageRelation \( fgpr \), and the Package \( p \) finally contains \( uc \). In a textual constraint language the needed constraint definition looks like \( f.\text{name} = f.fg.fgr.p.uc.\text{name} \). Although the regarded constraint is still very basic this expression is harder to imagine and to understand, especially without a figure which we omit here.
intentionally. The situation gets worse if the condition whether a correspondence link is consistent or not becomes more complex. The way out is to use a graphical notation for the definition of consistency conditions. Only parts of a condition which cannot reasonably be expressed in a graphical way should still be expressed textually. This concerns conditions like the equality of names for instance.

About 30 years ago graph grammars have been invented for the purpose to define the syntax and the static semantics of visual languages graphically [37,42]. Used in combination with meta-models they exactly offer the appropriate means to define the constraints which determine the consistency of our correspondence links. Graph grammars or, more precisely, programmed graph transformation systems have been adapted and implemented by various integrated visual programming environments like PROGRES [47], FUJABA [30], AGG [48], or DiaGen [26]. These environments have been and still are successfully used as meta-CASE tools for prototyping integrated sets of CASE and programming tools [29]. For further details concerning various forms of graph grammars, available implementations, and related success stories the reader is referred to the “Handbooks of Graph Grammars and Computing by Graph Transformation [11,12] as well as to the proceedings of two workshops on “Applications of Graph Transformations with Industrial Relevance” [31,36].

In order to be able to combine OMG’s meta-modeling world with the concept of graph grammars we interpret MOF meta-model instances and UML object diagram representations of the tools’ data as directed node and edge labeled graphs. Objects (MOF class instances) obviously correspond to attributed nodes with type labels, whereas links (MOF association instances) are interpreted as directed binary edges with type labels. Tool meta-models are interpreted as graph schemas or type graphs which define the types of nodes and edges as well as the associated attributes [47]. Besides a graph schema a graph grammar provides a set of graph rewriting rules. These rules describe how any graph that conforms to the graph grammar can be created. Each rule provides a left-hand side which specifies a pattern in the graph under construction that must be found in order to apply the rule. Additionally, each rule provides a right-hand side which specifies a pattern that replaces the selected match of its left-hand side as a result of the rule application. Figure 8 depicts a first example of a graph rewriting rule.

Figure 8a uses a graph rewriting rule notation with separated left- and right-hand sides. Such a notation is used by PROGRES and many other graph rewriting system approaches. In order to apply the parametrized rule createUseCase to a graph under construction, the graph must possess at least
one `Diagram` node. After one of the possible matches of the rule’s left-hand side has been selected the difference between the rule’s right-hand side and its left-hand side is added to the graph\(^5\). After rule application the graph has got a new `UseCase` node with a `name` attribute value provided by the rule’s parameter `x`. This new `UseCase` is linked to the previously selected `Diagram` node. Figure 8b depicts the same rule in a collapsed style as used in FUJABA. The left-hand side of such a collapsed rule is determined by the nodes and edges without any stereotype\(^6\), the right-hand side consists of all unmarked nodes and edges as well as of all nodes and edges marked with the stereotype `new`. We use the collapsed style in the following. It is the preferred notation as long as monotonic rules are used which do not delete any nodes and edges from the graph.

As an example we now present a subset of the rules of a graph grammar for our UML use case diagrams. We recall that the meta-model shown in Figure 3 acts as the graph’s schema. Figure 9 lists the needed graph grammar rules. Please notice that we have used two rules with an empty left-hand side to define the additionally needed start graphs or axioms of such a grammar. These rules may only be applied to the empty graph thereby creating a graph which consists either of one `Package` or one `Diagram` node only.

The rule `createPackage` allows for the creation of a new `Package` with the provided name. Accordingly, the rule `createDiagram` allows for the creation of a new `Diagram`. The rule `addPackage` describes the addition of a new `Package` which is part of an existing `Package`. Correspondingly, `addDiagram` describes the addition of a new `Diagram` which is part of an existing `Package`. By using the rule `addUseCase` we can add new `UseCases` to existing `Diagrams`. Finally, we can connect existing `UseCases` by Generalization relationships by applying the rule `addGeneralization`. Figure 10 demonstrates how a

---

\(^5\) Deletion of nodes and edges is handled by removing the difference of a rule’s left-hand side and its right-hand side from the regarded graph.

\(^6\) In general, nodes and edges marked with the stereotype `delete` also belong to a rule’s left-hand side, but not to its right-hand side.
Fig. 9. Graph rewriting rules

sequence of applications of the introduced graph rewriting rules from Figure 9 looks like which creates the UML object diagram representation from Figure 4.

We start by applying the rule `createPackage("WipingWashing")`. This step creates a new `Package` with the name `WipingWashing`. We then add a new `Diagram` to this package by using the rule `addDiagram()`. By applying the rule `addUseCase("Wiping")` we add a `UseCase` with the name `Wiping` to the `Diagram`. We apply this rule twice more with different parameter values to create the other two `UseCases`. Finally, we use the rule `addGeneralization()` twice to create the `Generalization` relationships. Accordingly, we can write a graph grammar for the requirements document as well. For further details concerning various forms of graph grammars and their formal definition based on category theory, first/second order logic, and set theory the reader is referred to [11] and to Chapter 4 of this paper.

Our two graph grammars used in isolation now create and thereby characterize the languages of all meta-model consistent requirement documents or use case documents (object diagrams). In general the condition that a given graph grammar creates schema consistent (meta-model consistent) graphs only cannot be checked statically. The previously mentioned system PROGRES uses e.g. a static type system to guarantee that created edges do only connect nodes of proper types. It resorts to runtime checking as soon as complex multiplicity constraints or attribute constraints are involved. It is the topic of
ongoing research activities to develop verification techniques and tools which
guarantee at compile time that a given set of graph rewriting rules does not
violate a given set of static semantic rules (expressed e.g. in the form of condi-
tional graph patterns or OCL expressions) \[5,50,40\]. Furthermore, we cannot
give any guarantees that a specified graph grammar generates all schema con-
sistent graphs. On the contrary, we are using graph grammars to characterize
those subsets of all meta-model compliant use case diagrams and requirements
documents which can be mapped onto each other without violating any inter-
model consistency conditions.

More precisely, we are using triple grammars (TGGs) to specify consistency
conditions for correspondence links between related graphs (models, develop-
ment documents, tool data repositories). Triple graph grammars generalize
the idea of pair grammars introduced about 30 years ago by Pratt [38] for the
simultaneous specification of parsers from text files to abstract syntax graphs
and unparsers from syntax graphs back to text files. TGGs as introduced in
[43] offer appropriate means for the declarative specification of bidirectional
transformations between pairs of graphs, which are connected using a third
so-called correspondence graph. A TGG rule describes first of all the simulta-
neous derivation of two graphs using a pair of graph rewriting rules; a third
graph rewriting rule is used to check and create correspondence links between
related nodes of both regarded graphs as a side-effect. This combination of
three graph rewriting rules which have to be applied simultaneously was the reason for choosing the name “triple graph grammar”.

Applying the idea of triple graph grammars as it is to practice in order to keep documents consistent with each other is utopistic. This would mean that all development documents or models would evolve simultaneously. Additionally, all documents would be consistent all the time. Unfortunately, the situation is usually as follows: We have lots of documents that evolve concurrently. We would like to automatically create correspondence links for traceability purposes and to automatically fix any inconsistencies from time to time when a consistent state of the project database is needed. Thus, TGGs are declarative specifications of integration rules which are not directly useful for tool integration purposes. From these declarative specifications operational graph rewriting rules can be derived automatically. These operational graph rewriting rules are then used to realize needed data/tool integration services (cf. Chapter 3).

Like common graph grammars triple graph grammars have graph schemas [25]. A triple graph grammar schema simply consists of a pair of simple graph schemas plus a correspondence graph schema which introduces all additionally needed node and edge types for the realization of the connections between nodes of different graphs. In our example this role is played by the meta-model from Figure 5. Additionally, a TGG provides a set of triple graph rewriting rules. Figure 11 lists examples of such triple graph rewriting rules. The createFeatureGroupAndPackage TGG rule creates the initial graph (axiom) which consists of three subcomponents. The left part of the TGG rule in Figure 11 creates a new FeatureGroup node which belongs to a requirements subgraph, its right part creates a new Package node which belongs to a separate use case subgraph, whereas its middle part establishes the needed correspondence structure (one node and two edges) between the two new nodes as a third subgraph of the graph of all documents. Or to rephrase its effects in MOF meta-modeling terms, the rule creates two objects in different models and relates these new objects to each other by creating a correspondence link between them.

A constraint of the form $fg\text{.name} = p\text{.name}$ ensures that the names of the corresponding objects are the same. This equality is enforced using a separate constraint attached to the new correspondence link instead of just using a single rule parameter which determines the value of both object names for the following reasons: the clear separation between attribute values needed for the creation of the requirements document and the use case document on one hand and the constraints crossing document boundaries on the other hand simplifies
later on the derivation of operational graph rewriting rules from TGG rules, significantly.

The second and third TGG rule of Figure 11 add related subcomponents of class FeatureGroup or Feature and Package or UseCase to an already related pair of objects of class FeatureGroup and Package. Finally, the rule addDependencyAndGeneralization is responsible for creating a pair of Dependency and Generalization links that connect already related pairs of objects of the appropriate classes. Please notice that we have omitted the additional negative rule application condition that guarantees that not more than one Dependency link (UseCase link) is created between a given pair of Feature objects (UseCase objects). Multiple links of this kind between a single pair of objects are not only useless from the local point of view of the related models, but would also cause problems for the automatic creation of correspondence links. It would be unclear how to connect parallel links in one model to the corresponding parallel links in the other model.

Triple graph rewriting rules as introduced in Figure 11 can be sequentially applied in order to create a triple graph that conforms to the triple graph grammar, i.e. to create a consistent pair of documents or models simultaneously together with the needed corresponding links between them. We demonstrate the rule application in Figure 12.

In Figure 12a we have applied the rule createFeatureGroupAndPackage for simultaneously creating a FeatureGroup, a Package, and a correspondence link of the type FeatureGroupPackageRelation between them. We then apply the rule addFeatureAndUseCase to derive the situation depicted in Figure 12b.

In our approach we demand that triple graph rewriting rules must conform to the triple graph rewriting rule prototype in Figure 13. That means that each triple graph rewriting rule has exactly two primary objects a and b which are simultaneously created and linked by a new correspondence link abr. Additionally, both primary objects may be provided with a local context into which the new objects are inserted. Furthermore, the rule may possess an arbitrary number of correspondence links. They describe relationships which must exist between the local context objects of both documents; otherwise the regarded rule cannot be applied. Moreover, a constraint may be attached to the new correspondence link which must hold, too. This attribute constraint may only consist of simple equations involving pairs of attributes of objects belonging to different models. It is the subject of future research activities to permit more general forms of attribute constraints and to use well-known constraint programming techniques to derive attribute assignments which repair violated constraints.
Fig. 11. Example of triple graph rewriting rules

trary number of secondary objects which are connected to the selected unique primary object on each side. Secondary objects are created while rule application, but not linked by correspondence links. Figure 13 does not depict these
additional objects which are sometimes needed, when a single object of one model is translated into a set of related objects in the other model.

Finally, our TGG rules may never delete any objects for the following two reasons: First of all, TGGs are used to generate languages of consistent pairs of documents and not to manipulate pairs of documents, i.e. deletion of objects and edges are operations which are not needed. Furthermore, restricting ourselves to graph grammars with monotonic rules only we avoid the pitfalls of the “graph grammar parsing problem”. As soon as node or edge deleting rules are permitted, the graph grammar parsing problem, i.e. the problem to recognize all graphs generated by a given graph grammar, becomes almost unfeasible. Only very restricted classes of graph grammars are known until today which guarantee polynomial space and time complexity for their associated parsing algorithms [41]. Unfortunately, checking the consistency of pairs of documents or translating one document into another one based on a TGG specification essentially requires the solution of the parsing problem for the simple graph grammar components of the given TGG. Relying on all the constraints mentioned above we only have to visit all objects of the regarded
models in an appropriate order and to identify the TGG rule with the right context definition for the just selected primary object. Further details of this procedure will be explained in Chapter 3, for further details concerning the formal definition of TGGs the reader is referred to Chapter 4.

Figure 14 presents an example of a previously introduced TGG rule and shows the partition of this rule in its subcomponents.

When applied this rule creates the primary objects \( f \) and \( uc \) with the provided values for their \( name \) attributes. For this purpose the triple graph must contain a FeatureGroup and a Package to which the primary objects will be attached to. Additionally, the FeatureGroup and the Package must already be linked by a correspondence link of the type FeatureGroupPackageRelation. Finally, the value of the attribute \( name \) of \( f \) must equal the value of the attribute \( name \) of \( uc \). This rule does not create any secondary objects.

### 3 Deriving operational rules

Model-integrating consistency rules are specified using the declarative TGG approach and cannot be used directly for model integration purposes. As already mentioned above we translate declarative TGG rules into operational graph model transformation rules. These operational rules should cover the following model/tool integration scenarios:

(i) Rules are needed which check whether an existing correspondence link between two models is still valid or not. This includes checking whether the local contexts of connected elements still exists, whether the inter-model
relationships are fulfilled, and whether the attached attribute constraint still holds.

(ii) Other rules should be available which create all possible consistency links between matching elements of regarded model pairs.

(iii) Furthermore, there should be rules that propagate the creation of objects from one model to the other combined with the automatic creation of appropriate correspondence links.

(iv) Finally, we want to have a set of repair actions that can be applied when a consistency check fails. This includes a rule that propagates attribute changes from one element to the linked one if the constraint is violated in the selected direction, a rule that removes an invalid correspondence link, and rules that propagate the deletion of linked objects from one model to the other. These repair actions are not considered in this paper. The reader is referred to [22] for more details.

It is the main advantage of triple graph grammars that they may automatically be translated into various sets of regular operational graph transformation rules which support all tool integration scenarios listed above. The general idea of this translation process is as follows: First of all the two regarded models plus the correspondence links between them are considered to be a single graph (with three distinct subgraphs). Then all TGG rules are translated into regular graph transformation rules which manipulate different parts of the new graph as needed.
Figure 15 shows some of the most important operational rules that can be derived from the single declarative TGG integration rule of Figure 14. For additional operational rules the reader is referred to [22]. Figure 15a depicts an operational rule which tests whether a link is consistent or not. The link under test is marked with the stereotype if. Without this stereotype this rule would test whether the pattern as a whole can be found or not. This is not what is intended. With the if stereotype added to the correspondence link node this rule works as follows: the rule loops through the set of all existing FeatureUseCaseRelation correspondence links and tests whether a Feature and a UseCase are attached to the just regarded link. It tests furthermore whether the Feature is connected to a FeatureGroup which is linked to a Package by a FeatureGroupPackageRelation. Finally, it checks whether the package is connected to the UseCase and whether the constraint holds or not.

Figure 15b shows a rule which creates a FeatureUseCaseRelation between a Feature and a UseCase. In order to create the correspondence link the Feature must be connected to a FeatureGroup which is linked to a Package by a FeatureGroupPackageRelation. Additionally, the Package must be connected to the UseCase and the constraint must hold. Again we have omitted a negative application condition which prohibits multiple applications of this rule and the creation of more than one FeatureUseCase correspondence link between a regarded pair of Feature and UseCase nodes.

Figure 15c depicts a LR-transformation rule (forward transformation rule) which translates the regarded part of a requirements document into the corresponding cut-out of a use case diagram. More precisely, this rule adds a new UseCase to the use case diagram and links it to a Feature from the requirements document by a FeatureUseCaseRelation. The new UseCase will be connected to the Package which is linked to the FeatureGroup by a FeatureGroupPackageRelation which is connected to the Feature. The attribute values of the UseCase are set by the attribute assignment on the one hand and by default values on the other hand. Again a negative application condition has been suppressed to increase the readability of the picture which guarantees a one-to-one correspondence between Feature and UseCase nodes. Accordingly, Figure 15d depicts a RL-transformation rule (backward transformation rule).

4 Simple Triple Graph Grammars and LR-Translators

In the previous sections we have seen how triple graph grammars are used to specify correspondence relationships with associated integrity constraints between pairs of graphs (documents). The presented graphical notation gave
the reader the impression that regarded pairs of graphs plus their additional correspondence relationships are modeled as one graph which contains these different parts as subgraphs. But in reality to be integrated documents as well as the derived correspondence relationships are stored in different tool repositories. Our formal definition of triple graph grammars, presented here, reflects this situation appropriately. The productions of a triple graph grammar consist of pairs of simple graph grammar productions called left and right productions. An additional correspondence production plus morphisms (mappings) between productions are used to define the needed correspondence relationships.

As a consequence triple graph grammars specify languages of graph triples. Elements \((LG, CG, RG)\) belonging to these languages represent related graph structures \(LG\) and \(RG\), respectively, which are linked to each other by means of an additional correspondence graph \(CG\). The grammar for such a graph triple language consists of triples of productions \((lp, cp, rp)\), where each production component is responsible for generating or extending the corresponding graph component.

In principle, any graph model and any graph grammar approach may be used as the underlying basic formalism of triple graph grammars. To emphasize this, we will use a very simple class of graphs without any node and edge labels and rather straightforward rewriting rules, such that we are able to explain the principles of the new formalism both within the framework of the algebraic and the algorithmic graph grammar approach [13,28] without getting stuck into technical details. Further details like node and edge labels that point to nodes (classes) and edges (association) of a type graph (meta model) may be added easily as long as a set-theoretic and category-theoretic definition for these graph grammar extensions does exist.\(^8\)

We are running into problems as soon as complex application conditions in the form of (first-order predicate) logic expressions are added to triple graph grammar productions. In this case, a more powerful logic-based graph grammar formalism has to replace the set- or category-theoretic approaches used in this publication [44]. Unfortunately, it is still unclear how to extend the concept of monotonic productions for logic-based graph grammars with negative application conditions and how to adapt the translation of monotonic triple graph grammar productions into left-to-right (LR/forward) or right-to-left (RL/backward) translators in the general case. Therefore, we did not use

---

\(^8\) Typed graph grammars have been introduced in the literature several times; the latest publication related to a formal definition of typed graph grammars with node and edge attributes we are aware of is [14]
Fig. 16. Application of Simple Productions and Triple Productions

any complex application conditions in the preceding sections except of simple attribute equations

In the presented version of triple graph grammars the definition of a monotonic production is rather straight-forward. This means that any production’s left-hand side must be part of its right-hand side, i.e. productions do not delete vertices and edges. In this case, a given graph directly contains all necessary information about its derivation history, and graph parsing simply means covering a given graph with right-hand sides of productions (for further details cf. [43]). This simplifies the development of LR- or RL-translators considerably which

- take a given left (right) graph as input and compute its derivation history,
- determine the related sequence of productions for the missing right (left) graph, and
- and apply the determined sequence of productions to generate the corresponding right (left) graph.

Requiring monotonicity is not as restrictive as it seems to be at a first glance, since triple graph grammars are not intended to model editing processes on related graphs (with insertions as well as deletions and modifications of graph elements), but are a generative description of graph languages and their relationships. Following this line, we start with the definition of simple graphs, graph morphisms, and monotonic productions:

\[ \subseteq \quad \subseteq \quad \subseteq \quad \subseteq \]

Diagram a) Diagram b)

Remark:
The sign “\( \subseteq \)” labels any arrow which represents an inclusion.

9 In this case node attributes may be encoded as pointers to auxiliary nodes with unique labels as values; in this scenario testing the equality of two attributes simply means checking whether they are pointers to the same auxiliary attribute node.
Definition 4.1 A quadruple $G := (V, E, s, t)$ is a graph

- $V$ being a finite set of vertices,
- $E$ being a finite set of edges, and
- $s, t : E \to V$ as source and target vertex assigning functions.

Definition 4.2 Let $G := (V, E, s, t)$, $G' := (V', E', s', t')$ be two graphs. A pair of functions $h := (h_V, h_E)$ with $h'_V : V \to V'$ and $h_E : E \to E'$ is a graph morphism from $G$ to $G'$, i.e. $h : G \to G'$, iff:

$$\forall e \in E : h_V(s(e)) = s(h_E(e)) \land h_V(t(e)) = t(h_E(e)).$$

Furthermore, we will assume that the operators $\subseteq, \cup, \cap, \setminus$ are defined as usual for graphs as the pairwise application to their corresponding edge and node set components.

Definition 4.3 Any tuple of graphs $p := (L, R)$ with $L \subseteq R$ is a monotonic production and $p$ applied to a given graph $G$ produces another graph $G' \supseteq G$, denoted by: $G \sim p \sim G'$, with respect to redex selecting morphisms $g : L \to G$ and $g' : R \to G'$, iff:

- $g'|_L = g$, i.e. $g$ and $g'$ are identical w.r.t. the left graph $L$.
- $g'$ maps new vertices and edges of $R \setminus L$ onto unique new vertices and edges of $G' \setminus G$.

Using the categorical framework [13], the two conditions of Def. 4.3 may be replaced by requiring the existence of the pushout diagram a) of Figure 16. Based on this fundamental terminology we are now able to define graph triples as well as triple productions and their application to graph triples:

Definition 4.4 Let $LG, RG$, and $CG$ be three graphs, and $lr : CG \to LG$, $rr : CG \to RG$ are those morphisms which represent correspondence relationships between the left graph $LG$ and the right graph $RG$ via the correspondence graph $CG$ in the following way:

$$x \in LG \text{ is related to } y :\iff \exists z \in CG : x = lr(z) \cap rr(z) = y.$$ 

The resulting graph triple is denoted as follows:

$$GT := (LG \leftarrow lr - CG - rr \rightarrow RG).$$

Definition 4.5 Let $lp := (LL, LR)$, $rp := (RL, RR)$, and $cp := (CL, CR)$ be monotonic productions. Furthermore, $lh : CR \to LR$ and $rh : CR \to RR$ are graph morphisms such that $lh|_{CL} : CL \to LL$ and $rh|_{CL} : CL \to RL$ are
morphisms, too, which relate the left- and right-hand sides of productions \(lp\) and \(rp\) via \(cp\) to each other. The resulting **triple production** is:

\[
p := (lp \leftarrow lh - cp - rh \rightarrow rp).
\]

And the **application** of such a triple production to a graph triple

\[
GT := (LG \leftarrow lr - CG - rr \rightarrow RG)
\]

with redex selecting morphisms \((lg, cg, rg)\) produces another graph triple

\[
GT' := (LG' \leftarrow lr' - CG' - rr' \rightarrow RG'),
\]

i.e.: \(GT \sim p \sim GT'\), which is uniquely defined (up to isomorphism) by the existence of the “pair of cubes” in diagram b) of Figure 16. Its new morphisms \((lg', cg', rg')\) are already determined by Def. 4.3. Furthermore, the left-hand side diagram of Figure 17 proofs the existence and uniqueness of

\[
lr' : CG' \rightarrow LG' \text{ with } lr = lr'|_{CG} \text{ and } lh \circ lg' = cg' \circ lr'.
\]

This is a direct consequence of the pushout property for the square with corners \(CL, CG, CG',\) and \(CR\). In the same way, the existence and uniqueness of \(rr'\) can be shown (an algorithmic version of the proof may be found in [43]).

\(\Box\)

In the sequel, we often have to deal with triple production applications, where the redex or result for their left or right production application is already known in the form of a morphism \(g\). We denote these **restrictions for rewriting** \(GT\) into \(GT'\) by \(GT \sim p(g) \sim GT'\) in the sequel.

Having defined the application of triple productions to graph triples we are now able to model processes which extend related graphs (and their inter-
relationships) synchronously. But how can we handle the case, where a left
graph is given and we have to construct the missing right graph including all
inter-graph relationships or vice versa? For symmetry reasons, the solution is
the same in both directions. Therefore, the construction of \textit{LR-translators}
will be discussed in detail and the solution for \textit{RL-translators} may be obtained by
simply exchanging the roles of “left”- and “right”-hand side components.

Informally speaking we have to split a triple production \( p \) into a pair of
triple productions \( p_L \) and \( p_{LR} \). \( p_L \) is a \textit{left-local triple production} which rewrites
the left graph only. \( p_{LR} \) is a \textit{left-to-right translating triple production} which
keeps the new left graph unmodified but adjusts its correspondence and right
graph. Within the following propositions we will show how to split a triple
production into a left-local production and a left-to-right transformation. Fur-
thermore, we will prove that the application of a sequence of triple productions
is equivalent to the application of the corresponding sequence of left-local pro-
ductions followed by the sequence of left-to-right transformations.

\textbf{Theorem 4.6} \ A given triple production

\[ p := ((LL, LR) \leftarrow lh - (CL, CR) - rh \rightarrow (RL, RR)) \]

may be split into the following pair of equivalent triple productions:

\[ p_L := ((LL, LR) \leftarrow \epsilon - (\emptyset, \emptyset) - \epsilon \rightarrow (\emptyset, \emptyset)) \]

is the \textbf{left-local production} for \( p \), where \( \emptyset \) is the empty graph and \( \epsilon \) is an
inclusion of the empty graph \( \emptyset \) into any graph.

\[ p_{LR} := ((LR, LR) \leftarrow lh - (CL, CR) - rh \rightarrow (RL, RR)) \]

is the \textbf{left-to-right translating production} for \( p \). For these triple produc-
tions and any graph triples \( GT \) and \( GT' \) (as in Def. 4.5), and a morphism
\( lg' : LR \rightarrow LG' \) the following proposition holds:

\[ GT \sim p(lg') \sim GT' \iff \exists HT : GT \sim p_L(lg') \sim HT \land HT \sim p_{LR}(lg') \sim GT'. \]

\textbf{Proof.} The following equivalences prove that the vertical sides of the cubes of
Figure 16 b) and Figure 18 imply each other if all production applications use
the same morphism \( l\gamma' \) to select an image of graph \( LR \) in \( LG' \) (and thereby of \( LL \) in \( LG \)):

\[
LG \sim (LL, LR) \sim LG' \Leftrightarrow LG \sim (LL, LR) \sim LG' \land LG' \sim (LR, LR) \sim LG'.
\]

\[
CG \sim (CL, CR) \sim CG' \Leftrightarrow CG \sim (CL, CL) \sim CG \land CG \sim (CL, CR) \sim CG'
\]

\[
RG \sim (RL, RR) \sim RG' \Leftrightarrow RG \sim (RL, RL) \sim RG \land RG \sim (RL, RR) \sim RG'.
\]

And Def. 4.5 guarantees existence and uniqueness of all horizontal arrows. Furthermore, diagram 16 b) is equivalent to \( GT \sim p(l\gamma') \sim GT' \). We can even merge the two rows of cubes in the upper part of the diagram of Fig. 18 to a single row of cubes. Then, the new upper part of the diagram of Fig. 18 is equivalent to \( GT \sim p_L(l\gamma') \sim HT \). Finally, the lower part of the diagram of Fig. 18 is equivalent to \( HT \sim p_{LR}(l\gamma') \sim GT' \).

Please note that we used the name \( lr \) in Figure 18 to denote a morphism from \( CG \) to \( LG \) as well as its range extension to a morphism from \( CG \) to \( LG' \supseteq LG \). Furthermore, all arrows without any label denote inclusions and the domain restrictions of \( lh \) and \( rh \) from \( CR \) to \( CL \) have been omitted in order to keep the diagram as legible as possible.

In a similar way the splitting of a triple production into a right-local production followed by a right-to-left translating production may be defined, but we have still to show that we can use these locally equivalent splittings for the definition of graph transformations which create first a left graph completely and add a corresponding right graph and the accompanying correspondence graph afterwards or vice versa, i.e. we have to prove:

**Theorem 4.7** Given \( n \) triple productions \( p_1 \) through \( p_n \) and morphisms \( l\gamma_1 \) to \( l\gamma_n \), which determine the application results of left production components of \( p_1 \) through \( p_n \), we can prove that

\[
p_1(l\gamma_1) \circ \ldots \circ p_n(l\gamma_n) = (p^1_L(l\gamma_1) \circ \ldots \circ p^n_L(l\gamma_n)) \circ (p^1_{LR}(l\gamma_1) \circ \ldots \circ p^n_{LR}(l\gamma_n)).
\]

**Proof.** This follows directly from proposition 3.6 that ensures

\[
p_1(l\gamma_1) \circ \ldots \circ p_n(l\gamma_n) = (p^1_L(l\gamma_1) \circ p^1_{LR}(l\gamma_1)) \circ \ldots \circ (p^n_L(l\gamma_n) \circ p^n_{LR}(l\gamma_n))
\]

and the fact that
a triple production $p^k_L := ((LL, LR) \leftarrow \epsilon - (\emptyset, \emptyset) - \epsilon \rightarrow (\emptyset, \emptyset))$ modifies left graph components only and has no requirements with respect to correspondence or right graphs,

a simple production $(LR, LR)$ may be applied to a graph $LG'$ without causing any modifications, whenever $LG'$ is the result of applying first a monotonic production $(LL, LR)$ followed by an arbitrary number of different monotonic productions,

and $p^k_{LR} := ((LR, LR) \leftarrow lh - (CL, CR) - rh \rightarrow (RL, RR))$ keeps its left graph unmodified.

Therefore, we are allowed to exchange the application order of triple productions freely as long as for any natural numbers $i \leq k$ the application of $p^i_L$ precedes the application of $p^k_L$ for $i \neq k$, the application of $p^i_{LR}$ precedes the application of $p^k_{LR}$ for $i \neq k$, and the application of $p^i_L$ precedes the application of $p^k_{LR}$.

□
In a similar way, we can prove that a sequence of triple productions may be replaced by an equivalent sequence of corresponding right-local and right-to-left translating productions. Therefore, the problem of constructing LR- or RL-translations is solved in principle. The realization of such a translation process is divided into two steps:

- The given source graph is analyzed and a sequence of left-local (right-local) productions is computed, which creates the given source graph (if possible).
- Afterwards, the corresponding sequence of LR-translating (RL-translating) productions is applied to the initial (empty) target graph.

For further details concerning the first step of this algorithm the reader is referred to [43]. Furthermore, the topic of correspondence analysis is discussed on an informal level in [24]. An extension of the presented formalism of TGGs to n graphs (instead of pairs of graphs only) is out of the scope of this paper and discussed on an informal level in [22].

5 Related work

In this chapter we explain the relationships of our approach to the field of model driven application development (MDA) [23] and OMG’s request for proposals (RFP) [32] for a MOF-compliant “queries, views, and transformation” (QVT) approach. Furthermore, we compare our proposal to those model transformation and integration approaches that address OMG’s QVT, too. For a detailed comparison of the TGG model integration approach in its original form with “classical tool/data integration approaches” developed in the last millennium (like attribute-coupled grammars, broadcast message query server, etc.) the reader is referred to [29]. For a more comprehensive survey of elder and rather recently developed tool integration techniques the reader is referred to the CASE tool integration monography of Brown et al. [4] and the just published special sections on tool integration issues of the two Springer journals SoSym [45] and STTT [10].

5.1 OMG’s QVT-RFP

Although the field of data and tool integration has been studied for about twenty years now there still is a lack of domain-specific adaptable tool/data integration solutions which support consistency checking as well as incremental update propagation and which are not restricted to one-way transformations between pairs of tools only. This problem is addressed by OMG’s Request for proposals: MOF 2.0 Query/View/Transformation (QVT). This RFP demands
a number of features which can be used for classifying data and tool integration approaches. Each response to the RFP must:

• offer a language for specifying queries for selecting and filtering of model elements.

• provide a language for model transformation definitions. These definitions can be used to generated a target model from a given source model.

• have a MOF 2.0-compliant abstract syntax for each language.

• have an expressive transformation language allowing automatic transformations.

• support the creation of views.

• support incremental change propagation between source and target model.

Additionally, a response may:

• offer transformations which can be executed in both directions.

• provide traceability information.

• use generic transformation definitions for reuseability purposes.

• provide some sort of transactional mechanism.

• support the use of additional data which is not contained in the source model.

• allow transformations for the case that source and target model coincide.

For an extensive survey of model-transformation and integration approaches in general the reader is referred to [9]. This survey distinguishes among other things (e.g. features of transformation rules, features of rule application scoping, source-target relationship) between the following main categories of approaches:

• **declarative** approaches usually offer a logic-based language for the definition of consistency constraints between related models and are able to derive consistency checking and inconsistency removing update operations from these constraints.

• **graph transformation** approaches interpret models as graphs and use graph transformation rules to describe one-way translations of one model into another one.

• **hybrid** approaches that combine different techniques from the other categories.
With respect to Czarnecki’s categorization of model transformation approaches, TGGs and MDI are hybrid approaches that combine the advantages of declarative and graph transformation approaches.

5.2 Related approaches

In the following, we will select some model transformation approaches which are typical representatives of the above listed categories of Czarnecki and compare them based on the list of requirements of OMG’s RFP QVT.

The submission from the QVT-partners [39] to OMG’s QVT-RFP aims at model integration by means of so-called relations and mappings. The approach is similar to our own in the way that it allows for model consistency checking based on relation definitions, and doing forward and backward model transformations by defining mappings. One difference to our approach is that the model checking and model transformation rules are textually denoted. Furthermore, the rules are not declarative. That means that the rules for forward and backward model transformations are not automatically derived and must be specified manually. As a consequence there are no guarantees that consistency checking relations, forward and backward transformations for a pair of models implement the same set of constraints. Figure 19 gives an impression of how relation and mapping specifications look like. The relation from Figure 19a describes the correspondence between a method in a UML class diagram and an XML element which should represent this method. The relation says that for each method in the UML class diagram there exists an XML element with the name Method. The XML element provides an attribute with the name name and a value n which corresponds to the name of the method in the UML class diagram. Furthermore, the XML element contains the method’s body b. Accordingly, the mapping from Figure 19b describes in which way a XML element is created from a given method in a UML class diagram.

The goal of the graph transformation system GReAT [2] is to allow for the operational specification of rather complex model transformations. As we do, this approach uses graph rewriting rules based on a UML-like notation. In contrast to our own approach GReAT is not designed to keep existing models consistent with each other. GReAT takes one model as input and completely transforms it to another model. GReAT aims at a very expressive language for graph rewriting rules. Besides multiplicities for graph nodes the language introduces multiplicities for edges in graph rewriting rules for simultaneous

---

10 An additional graphical visualization has been defined, but its role for the definition of model transformations is unclear
Fig. 19. Relation and mapping specifications in QVT-partners’ approach

Fig. 20. Example rule specification in GReAT

manipulation of sets of rule matches. Additionally, it offers sophisticated control structures as sequences, non-determinism, hierarchical expressions, recursion, and branching. Figure 20 shows how rules in GReAT look like. Each rule may provide in and out parameters for passing objects to the rules. In the example the object TLR acts as the in and out parameter. The pattern of each rule is a graph rewriting rule. The ticks read like our new stereotypes and designate elements that are created during rule application. Additionally, each rule may provide a Guard that is checked before rule application. The Attribute Mapping expresses which values the attributes of the created objects get.

Like GReAT the BOTL approach [7] aims at model transformations. Similar to our approach it offers a UML-like notation for graph rewriting rules working on pairs of models/graphs. One difference is that this approach cannot be used for checking consistency of related models. BOTL just bidirectionally transforms one model to another model. Furthermore, it uses a different rule application strategy. Instead of applying one rule from of a set of concurrently applicable rules (selected by taking user preferences into account), the BOTL approach applies all matching rules in parallel and merges
the resulting set of graphs if possible. This is in our opinion a rather strange behavior, at least in those cases, where different rules represent competing model transformation options and where rule-application conflicts should be resolved by a human being. An impression how rules in BOTL look like is given by Figure 21.

The xlinkit approach [27] is a typical example of a logic-based declarative approach. It is designed to detect inconsistencies on tools’ data represented by XML-files and to generate proposals how to remove existing inconsistencies between pairs of models (manually). It uses a completely textual notation based on first order logic and XML. Figure 22 demonstrates how a consistency checking rule in xlinkit looks like. The rule states that for all elements in Adverts there must be an element in Catalogue/Products with the same name.

Finally, we should mention IMPROVE [8] which is also based on TGGs and has the same roots as our approach. It may be considered as a predecessor of the MDI approach presented here. It is mainly used for the incremental integration of pairs of tools in the field of chemical engineering and was as far as we know the first TGG-based approach with a UML-like notation. Unfortunately, it uses its own meta-modeling concepts inherited from the graph transformation system PROGRES [47] and is not compatible with OMG’s meta-modeling world. From IMPROVE we have learned how to derive object-deletion rules from a given TGG and how to offer incremental update
propagation support. Furthermore, IMPROVE is a source of inspiration for the semi-automatic creation of tool wrappers which hide proprietary tool APIs behind standard model manipulation interfaces.

The IMPROVE approach currently is implemented as a prototype for one tool integration example using the graph rewriting system PROGRES. In contrast to our approach IMPROVE interprets triple graph rewriting rules at run-time with all the pros and cons of an interpretative versus a compiled approach. Since, IMPROVE uses triple graph grammars as well Figure 23 only shows a very simple example of rule specification.

Figure 24 classifies the presented related approaches with respect to OMG’s QVT-RFP and summarizes their features. As we can see the strengths of our approach are that it conforms to standards like MOF and JMI. Additionally, our approach is the only one which can easily be extended for multi-document integration as explained in the last chapter. On the other hand, the currently implemented tool integration framework is not incrementally working as the IMPROVE approach; most tools used in practise do not provide change events which are urgently needed to trigger the execution of update propagation rules. Furthermore, it is a matter of debate whether true incremental and continuous propagation of changes of one document to another document is useful in a scenario, where different persons manipulate these documents and where usually version management systems are used to avoid the immediate propagation of document changes of one person to the rest of a project’s team. We currently simulate a kind of incremental consistency-checking behavior using a batch-oriented approach. In this approach we compare the results of two runs of correspondence link creating or checking integration tool activations.

6 Conclusion

In this paper we have presented a declarative model integration approach which combines OMG’s meta-modeling standard MOF and the triple graph grammar based tool/data integration approach invented about 10 years ago. This approach does not require that the data (models) of all regarded tools are stored in one database, but directly accesses the data repositories of these tools using JMI-standard compliant interfaces. An additionally needed database is
only responsible for storing any created traceability relationships (correspondence links) between data elements of different tools. We have shown how one rule-based visual specification can be used to derive code for creating and consistency checking of correspondence links as well as for forward and backward propagation of changes. Furthermore, we have argued that we thus address the most important requirements of OMG’s request for proposal QVT of a standard for querying, viewing, and transforming MOF meta models. In addition, we have recalled the formalism of triple graph grammars as introduced in [43]. This formal definition has been presented in two versions using the framework of algorithmic set-theoretic as well as of the algebraic category-theoretic graph grammar families. As discussed the formal definition may easily be extended to deal with concepts like node and edge labels or attributes which are needed for a graph encoding of MOF models.

Finally, we have mentioned some open problems which have to be solved in the future. Until now our model integration rules may only make use of simple equations for the description of dependencies of attribute values of different models. In most cases this restriction does not cause any problems, since different mechanisms are used to deal with the structural part of the involved models. Nevertheless, it is our plan to apply standard constraint programming techniques to translate more complex attribute constraints into directed equations, which then propagate attribute changes from one model to another one.

References


