

MDI - a Rule-Based Multi-Document and Tool Integration Approach

Alexander Königs, Andy Schürr

Merckstr. 25

Real-Time Systems Lab

University of Technology Darmstadt

D-64283 Darmstadt, Germany

{koenigs|schuerr}@es.tu-darmstadt.de

March 3, 2006

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 incremental update propagation. Currently used solutions are usually hand-coded one-way transformations between pairs of tools only. In this article we propose a new rule-based approach that allows for the declarative specification of data integration rules concerning multiple data repositories. Hence, we call our approach “Multi Document Integration” (MDI). It generalizes the formalism of triple graph grammars and replaces the underlying data structure of directed graphs by the more general data structure of MOF-compliant meta models. Our integration rule specifications are translated into JMI-compliant Java code which is used for various purposes by a tool integration framework. 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.

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 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*.

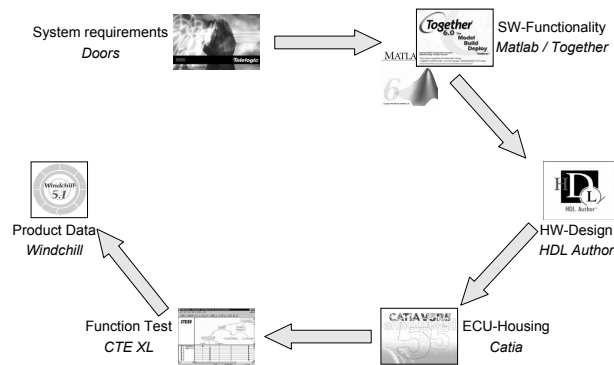


Figure 1: Different tools in one specific software system engineering process

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. For a detailed history the reader is referred to [KS05b, LS96, B⁺94].

Today available tool/data integration solutions suffer from the fact that the involved CASE tools usually are *commercial off the shelf* (COTS) and have been built without taking the requirements of data integration frameworks into account [A⁺03]. In most cases, they offer poor if any support at their interfaces (APIs) for keeping data consistent across tool (vendor) boundaries. As a consequence, most COTS tool data integration solutions are still hand-coded, uni-directional batch processes, which make use of rather primitive export and import interfaces. Usually, these data integration solutions have problems with the (semi-)automatic creation of traceability relationships as well as with bidirectional and/or incremental propagation of updates. Furthermore, they are hard to maintain and difficult to reuse in a different domain or for another set of tools with similar functionalities. This is mainly due to the fact that often rather domain-specific integrity constraints are hard-wired and mixed with code dealing with parsing and unparsing of proprietary data formats or other strange technicalities of available tool interfaces.

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) [OMG02]. The RFP requires first of all that OMG’s *Meta-Object Facility* (MOF) [OMG03a] 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”. This specification is currently available in version 2.0. In combination with language bindings for Java, Corba, etc. it provides to a certain extent the needed standard for COTS tool data interfaces.

In this paper we extend our ideas from [KS05b] and [KS05a] and propose a new 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 [Sch94], but never been adapted to the world of OMG’s standards beforehand. TGGs have been adopted for migrating relational to object oriented database systems [JSZ96], for tool integration in the *IPSEN* project [LS96] and *PROGRES* context [BW03], and for diagram consistency management in the *FUJABA* project for instance [Wag01]. For a detailed introduction to and overview of TGGs the reader is referred to [KS05b, Sch94].

TGGs and the QVT-RFP are restricted to the specification of integration and transformation rules between pairs of documents only. On the contrary, our multi document integration (MDI) approach aims for the specification of integration rules between an arbitrary number of documents.

We start by giving a running example in Section 2 in order to illustrate our proposal. In Section 3 we introduce meta-modeling as a well known technique for describing the structure of documents as well as for describing integration schemas. We explain how we can write declarative integration rules based on the tools’ meta-models and the integration schema in Section 4. In Section 5 we derive operational rules from our declarative specification. Finally, we show how we can generate code from these operational rules and present rule application strategies in order to execute the generated code by a tool integration framework in Section 6.

We continue in Section 7 by presenting OMG’s QVT request for proposals as a possibility to classify document and tool integration approaches. On this basis, we shortly introduce related approaches, and compare them with each other as well as with our own approach. Finally, Section 8 summarizes and concludes our paper, discusses open issues and future work.

2 Running example

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* [Tel] for the definition of system requirements in natural language, and *Together* from *Borland* [Bor] for the visual definition of use cases and class diagrams from the *Unified Modeling Language* (UML)¹ [OMG03c]. Thus, our running example deals with the integration of three sources of information (documents). Nevertheless, our approach is applicable for an arbitrary number of regarded documents.

Figure 2 shows parts of the data² 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 made 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

¹The 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.

²In the context of tools we use the terms *documents* and *data*. In the context of *meta-modeling* we correspondingly talk about *models* and *objects*. Finally, we use the terms *graphs* and *nodes* in the context of graph grammars.

a)

ID	Text
1	1 System Requirements
9	1.1 Types
54	WiperMode: Off, Interval, Slow, Fast, Wash
55	IgnitionMode: Off, On
56	DropArmSwitchMode: Off, Interval, Slow, Fast, Wash
67	Pump: Off, On
10	1.2 Interfaces
11	1.3 Featuregroup: Wiping&Washing
13	1.3.1 Featuregroup: Wiping
14	1.3.1.1 Feature: Drop-arm switch wiping
60	1.3.1.1.1 Parameter
61	In: DropArmSwitch : DropArmSwitchMode
42	1.3.1.1.2 Activation
43	1.3.1.1.3 Preconditions
44	1.3.1.1.4 Invariants
45	1.3.1.1.5 Description
46	1.3.1.1.6 Postconditions
47	1.3.1.1.7 Deactivation
16	1.3.1.2 Feature: Wiping for washing
25	1.3.1.2.1 Activation
27	1.3.1.2.2 Preconditions
28	1.3.1.2.3 Invariants
29	1.3.1.2.4 Description
30	1.3.1.2.5 Postconditions
31	1.3.1.2.6 Deactivation
57	1.3.1.3 Parameter
58	In: Ignition : IgnitionMode
59	Out: Wiper : WiperMode
32	1.3.1.4 Activation
33	1.3.1.5 Preconditions
36	1.3.1.6 Invariants
39	1.3.1.7 Description
40	1.3.1.8 Postconditions
41	1.3.1.9 Deactivation
15	1.3.2 Feature: Washing with wiping
64	1.3.2.1 Parameter
65	In: Ignition : IgnitionMode
66	In: DropArmSwitch : DropArmSwitchMode
22	1.3.2.2 Relationships

Username: akoenigs Exclusive edit mode

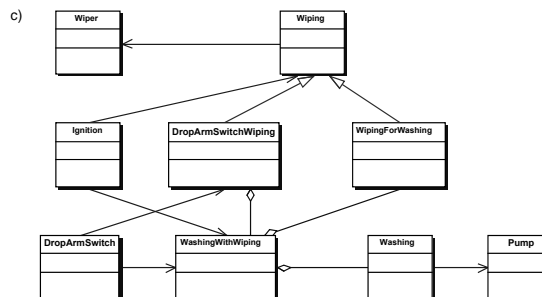
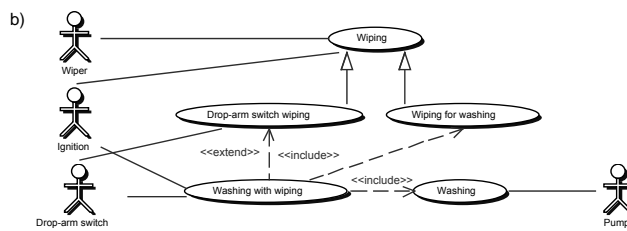


Figure 2: Example requirements, use case, and class diagram documents

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. Since this choice should hold for all regarded documents it is desirable to integrate multiple documents at the same time instead of integrating them pairwise.

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 feature 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 controller 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 generalization relationships between classes, whereas it is a matter of debate whether the translation of *extends* 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.

3 Meta-modeling

In this section we introduce *meta-modeling* as a well-known technique for defining the abstract syntax and static semantics of data kept in software system engineering tools [SD04]. Furthermore, we use meta-modeling for declaring *correspondence link types* between the data repositories of different tools. We use MOF version 2.0 [OMG03a] since it is OMG's standard language specification for this purpose. In contrast to UML (class diagrams) MOF is accompanied by the *Java Meta Interface* (JMI) standard of Sun which defines precisely how APIs of tools (data repositories) have to look like [Mos02]. Furthermore, the QVT-RFP demands the usage of MOF [OMG02]. 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)-

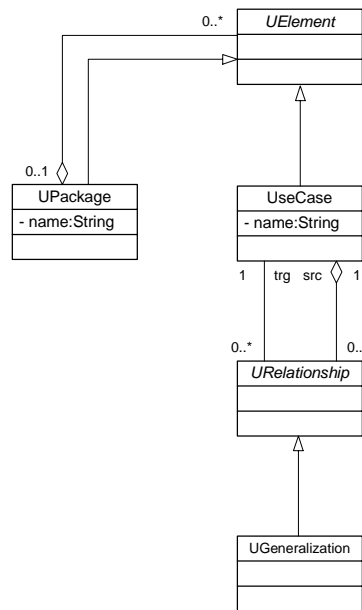


Figure 3: MOF 2.0-compliant meta-model of *UML use case diagrams*

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 **UseCases** that are the leaves of a **UPackage** hierarchy. **UseCases** can be related to each other by **UGeneralization** relationships³. Having such a meta-model for a document we can represent the data contained in the document as a *object diagram* which conforms to the meta-model. Figure 4 shows a part of the object diagram representation of the UML use case diagram document of our running example (cf. Figure 2b). Accordingly, we can write meta-models for *requirements documents* stored in Doors and *class diagrams* stored in Together.

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

³Extend and Include relationships as well as Actors are not used in the following examples and, therefore, omitted here.

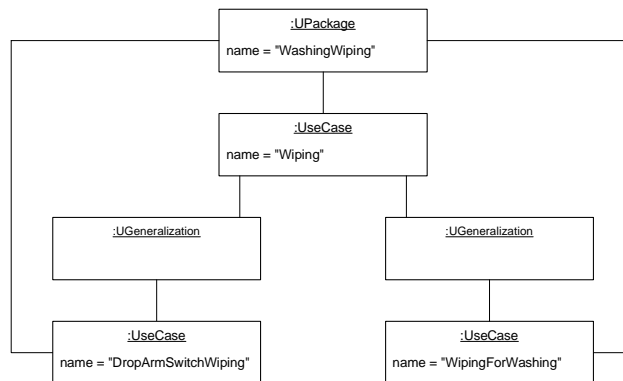


Figure 4: Part of the *UML object diagram* representation of the *UML use case diagram*

meta-model of the other tools. Figure 5 depicts the four meta-models that we need for declaring correspondence links in our windscreen wiper project.

Figure 5a shows the meta-model for class diagrams. Accordingly, Figure 5b depicts the meta-model for requirement documents. The meta-model for use case diagrams is recalled in Figure 5d. Finally, Figure 5c uses all 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 `ClassDiagram`, `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 further 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 7a. The declared link type may have an arbitrary name and must be marked with the stereotype `Integration`. Link types are introduced as classes with stereotype `Integration` because MOF 2.0 does not allow for the definition of n-ary associations. Switching from MOF to UML which supports n-ary associations as a metamodeling language would solve this problem but introduces another one: UML is not equipped with a standard for the definition of tool APIs comparable to JMI for MOF. Therefore, our notation has the character of an abbreviation that introduces a real class for the

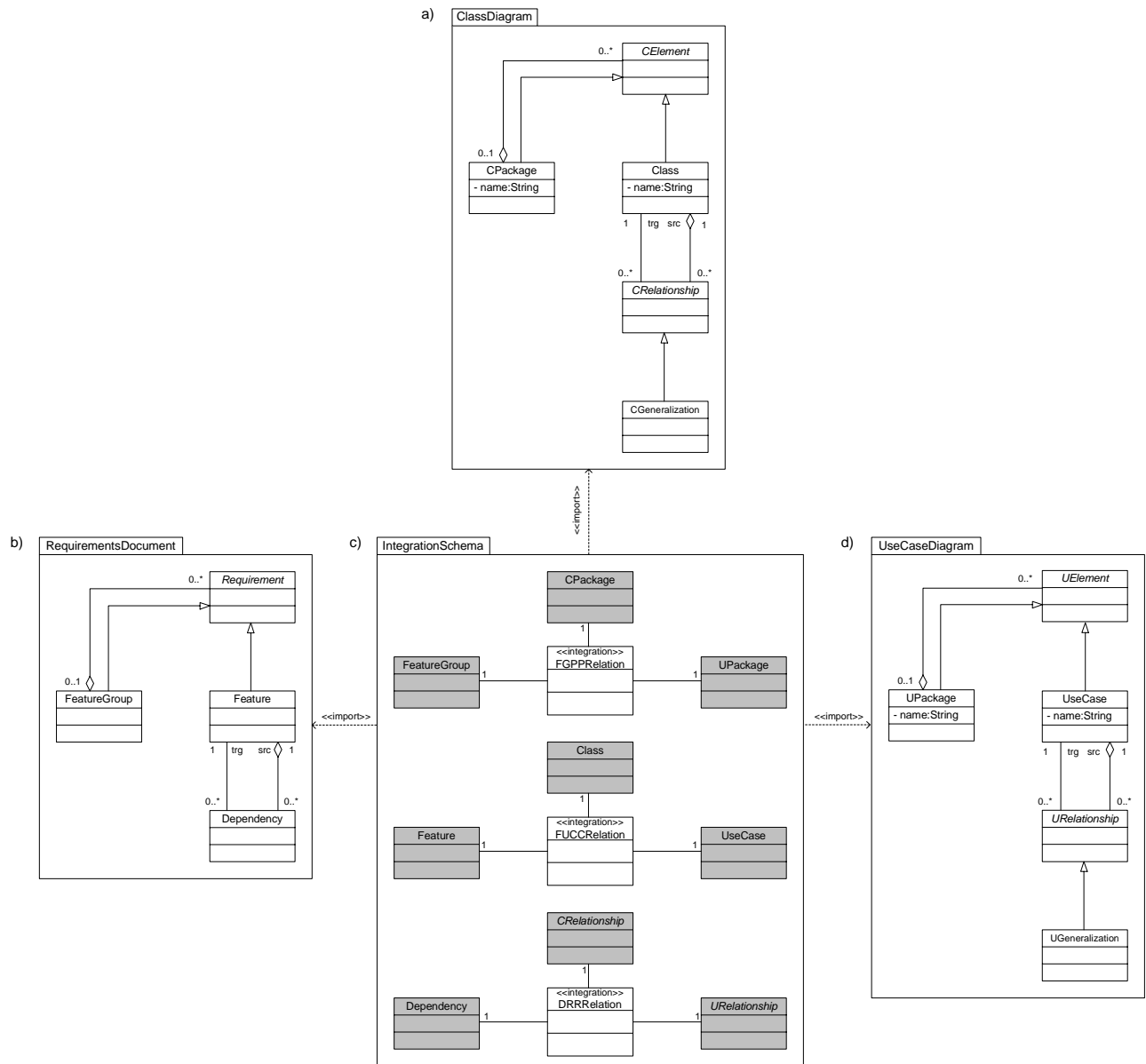


Figure 5: Tools' and integration meta-models

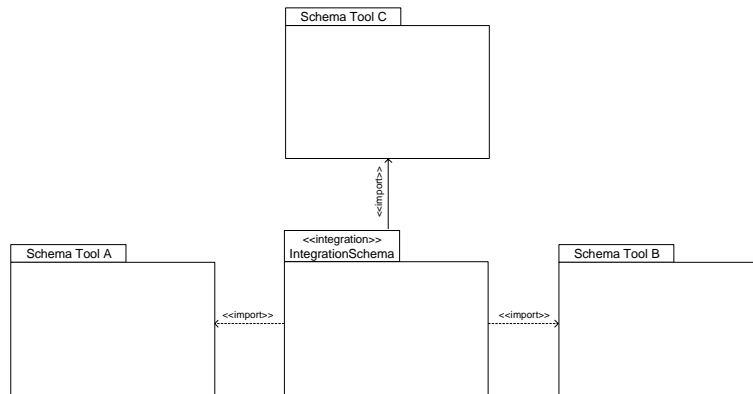


Figure 6: Organization of meta-models in our approach

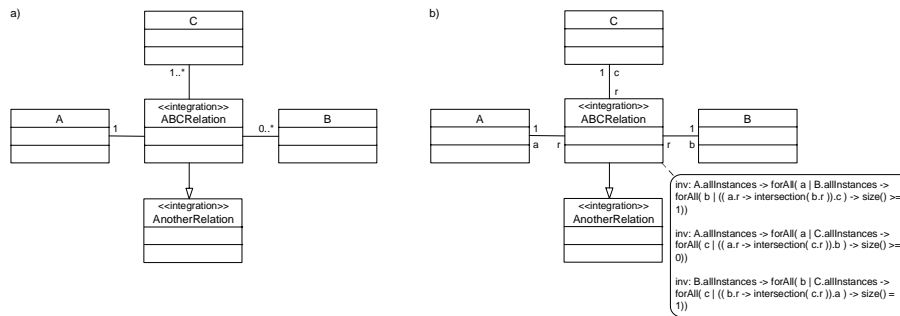


Figure 7: Prototype of a correspondence link type declaration

correspondence link and translates the given multiplicities into (OCL-)constraints as shown in Figure 7b.

Any link like the one depicted in Figure 7a has an arbitrary number of ends. The ends point to the corresponding elements from the tools' meta-models. Each end may carry a multiplicity. If no multiplicity is provided for an link end we assume the multiplicity $0..*$ as default. These multiplicities are later on used to *check consistency and completeness* of (semi-)automatically created correspondence links.

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. This is of great importance for the definition of reusable and adaptable meta-model integration specifications.

4 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) [OMG03b] for that purpose. For example, if we want to state that a correspondence link of type `FGPPRelation` is consistent if the name of the attached `FeatureGroup` with the identifier `fg` is the same as the name of the attached `UPackage` with the identifier `up` and as the name of the attached `CPackage` with the identifier `cp`, we would write something like `fg.name = up.name = cp.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 `FUCRelation` 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 as the name of the attached `Class` with the identifier `c`, and `f` is contained in a `FeatureGroup` `fg` which is linked to a `UPackage` `up` and to a `CPackage` `cp` by a correspondence link of type `FGPPRelation` `fgppr`, and the `UPackage` `up` finally contains `uc`, whereas the `CPackage` `cp` contains `c`. In a textual constraint language the needed constraint definition looks like `f.name = f.fg.fgppr.up.uc.name = f.fg.fgppr.cp.c`. 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 or the number of to be integrated documents increases. 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* [PR69, Sch70]. 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* [SWZ99] or *FUJABA* [NNZ00]. These environments have been and still are successfully used as meta-CASE tools for prototyping integrated sets of CASE and programming tools

[Nag96]. 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 [EEKR97, EEKR99] as well as to the proceedings of two workshops on “Applications of Graph Transformations with Industrial Relevance” [NSM99, PNB03].

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 [SWZ99]. 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. For more details on simple graph grammars and an example the reader is referred to [KS05b, Nag79].

As we pointed out in [KS05b], we are using *triple grammars* (TGGs) to specify consistency conditions for correspondence links between pairs of related graphs (models, development documents, tool data repositories). Triple graph grammars generalize the idea of *pair grammars* introduced about 30 years ago by Pratt [Pra71] 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 [Sch94] 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 simultaneous 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”. For more information on triple graph grammars the reader again is referred to [KS05b, Sch94, LS96]. In this paper we generalize this “triple graph grammar” approach to “multi graph grammars” which support the integration of an arbitrary number of graphs.

Like common graph grammars and triple graph grammars [LS96] multi graph grammars have graph schemas. A *multi graph grammar schema* simply consists of a set 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 multi graph grammar provides a set of *multi graph rewriting rules*. Figure 8 lists examples of such multi graph rewriting rules. The `createFeatureGroupPackagePackage` rule creates the initial graph (axiom) which consists of four subcomponents. The left part of the rule in Figure 8a creates a new `FeatureGroup` node which belongs to a requirements subgraph, its right part creates a new `UPackage` node which belongs to a separate use case subgraph, its upper part creates a new `CPackage` node which belongs to a separate class diagram subgraph, whereas its middle part establishes the needed correspondence structure (one node and three edges) between the three new nodes as a fourth subgraph of the graph of all documents. Or to rephrase its effects in MOF meta-modeling terms, the rule creates three 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.name = up.name = cp.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 the object names for the following reasons: the clear separation between attribute values needed for the creation of the requirements document, the use case document, and the class diagram document on one hand and the constraints crossing document boundaries on the other hand simplifies later on the derivation of operational graph rewriting rules (cf. Section 5) from multi graph grammar rules, significantly.

The rule of Figure 8b add related subcomponents of class `Feature`, `UseCase`, and `Class` to an already related set of objects of class `FeatureGroup`, `UPackage`, and `CPackage`.

Multi graph rewriting rules as introduced in Figure 8 can be sequentially applied in order to create a multi graph that conforms to the multi graph grammar, i.e. to create a consistent set of documents or models simultaneously together with the needed corresponding links between them. We demonstrate the rule application in Figure 9.

In Figure 9a we have applied the rule `createFeatureGroupPackagePackage` for simultaneously creating a `FeatureGroup`, a `UPackage`, a `CPackage`, and a correspondence link of the type `FGPPRelation` between them. We then apply the rule `createFeatureUseCaseClass` to derive the situation depicted

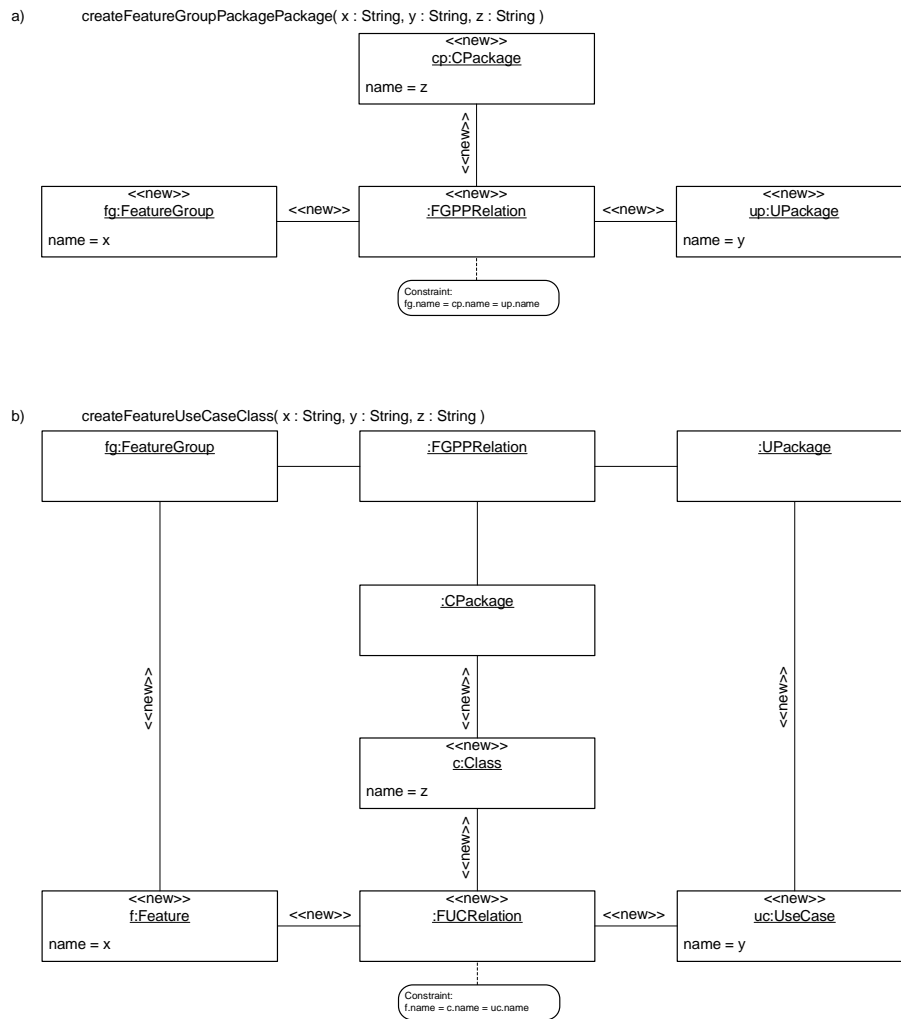


Figure 8: Example of multi graph rewriting rules

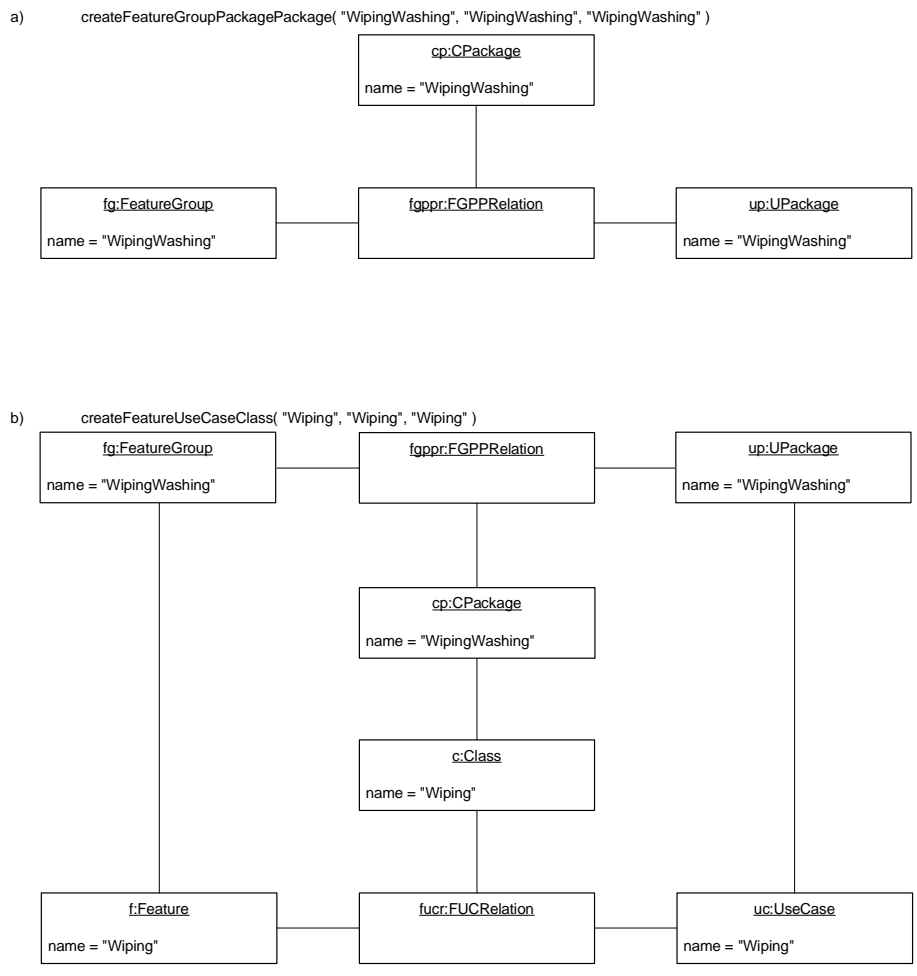


Figure 9: Application of multi graph rewriting rules

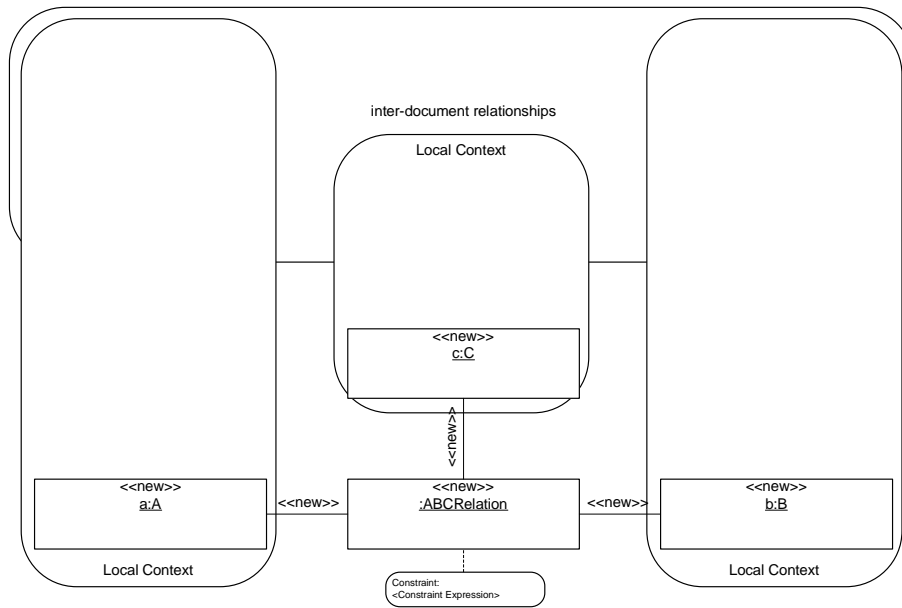


Figure 10: Prototype of a multi graph rewriting rule

in Figure 9b.

In our approach we demand that multi graph rewriting rules must conform to the multi graph rewriting rule prototype in Figure 10. That means that each multi graph rewriting rule has a set of *primary objects* which are simultaneously created and linked by a new correspondence link. Additionally, each primary object 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 the regarded 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 attributes of objects belonging to different models⁴. Furthermore, the rule may create an arbitrary number of *secondary objects* which are connected to the selected unique primary objects. Secondary objects are created while rule application, but not linked by correspondence links. Figure 10 does not depict these additional objects which are sometimes needed, when a single object of one model is

⁴It 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.

translated into a set of related objects in other models.

Finally, our multi graph grammar rules may never delete any objects for the following two reasons: First of all, multi graph grammars are used to generate languages of consistent sets of documents and not to manipulate sets of documents, i.e. deletion of objects and edges are operations which are not needed for the specification of the document integration. We will see in Section 5 how operations which propagate updates or object deletion from one document to the related documents are derived automatically. 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 [RS97]. Unfortunately, checking the consistency of sets of documents or translating one document into others based on a multi graph grammar specification essentially requires the solution of the parsing problem for the simple graph grammar components of the given multi graph grammar. 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 multi graph grammar rule with the right context definition for the just selected primary object. Further details of this procedure will be explained in Section 6.

Figure 11 presents an example of a previously introduced multi graph grammar rule and shows the partition of this rule in its subcomponents.

When applied this rule creates the primary objects `f`, `uc`, and `c` with the provided values for their `name` attributes. For this purpose the multi graph must contain a `FeatureGroup`, a `UPackage`, and a `CPackage` to which the primary objects will be attached to. Additionally, the `FeatureGroup`, the `UPackage`, and the `CPackage` must already be linked by a correspondence link of the type `FGPPRelation`. Finally, the value of the attribute `name` of `f` must equal the value of the attribute `name` of `uc` and `c` respectively. This rule does not create any secondary objects.

5 Deriving operational rules

Applying the idea of multi 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

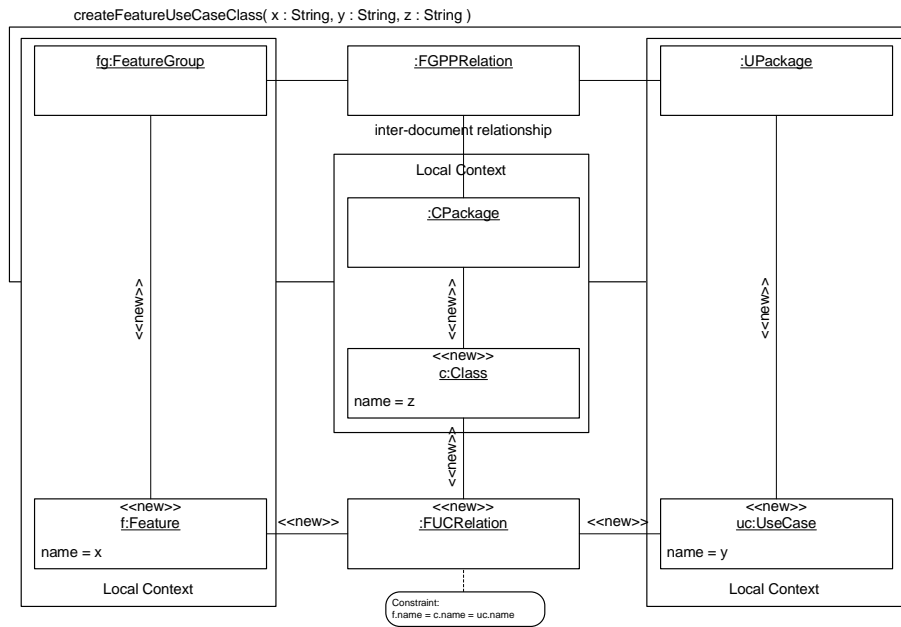


Figure 11: Example of a multi graph rewriting rule in our approach

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, multi graph grammars 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.

The operational rules should cover the following model/tool integration scenarios:

1. Rules are needed which check whether an existing correspondence link between a set of models is still valid or not. This includes checking whether the local contexts of connected elements still exist, whether the inter-model relationships are fulfilled, and whether the attached attribute constraint still holds.
2. Other rules should be available which create all possible consistency links between matching

elements of regarded models.

3. Furthermore, there should be rules that propagate the creation of objects from one model to the others combined with the automatic creation of appropriate correspondence links.
4. 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 ones 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 others.

It is the main advantage of multi 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 regarded models plus the correspondence links between them are considered to be a single graph (with distinct subgraphs). Then all multi graph grammar rules are translated into regular graph transformation rules which manipulate different parts of the new graph as needed.

Figure 12 shows some of the most important operational rules that we derived from the single declarative multi graph grammar integration rule of figure 11. Figure 12a 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 stereotype `if` added to the correspondence link node this rule works as follows: the rule loops through the set of all existing `FUCRelation` correspondence links and tests whether a `Feature`, a `UseCase`, and a `Class` are attached to the just regarded link. Furthermore, it tests whether the `Feature` is connected to a `FeatureGroup` which is linked to a `UPackage` and a `CPackage` by a `FGPPRelation`. Finally, it checks whether the `UPackage` is connect to the `UseCase`, whether the `CPackage` is connected to the `Class`, and whether the constraint holds or not.

Figure 12b shows a rule which creates a `FUCRelation` between a `Feature`, a `UseCase`, and a `Class`. In order to create the link the `Feature` must be connected to a `FeatureGroup` which is linked to a `UPackage` and a `CPackage` by a `FGPPRelation`. Additionally, the `UPackage` must be connected to the `UseCase`, whereas the `CPackage` must be connected to the `Class`, and the given constraint must hold. Again we have omitted a negative application condition which prohibits multiple applications of this

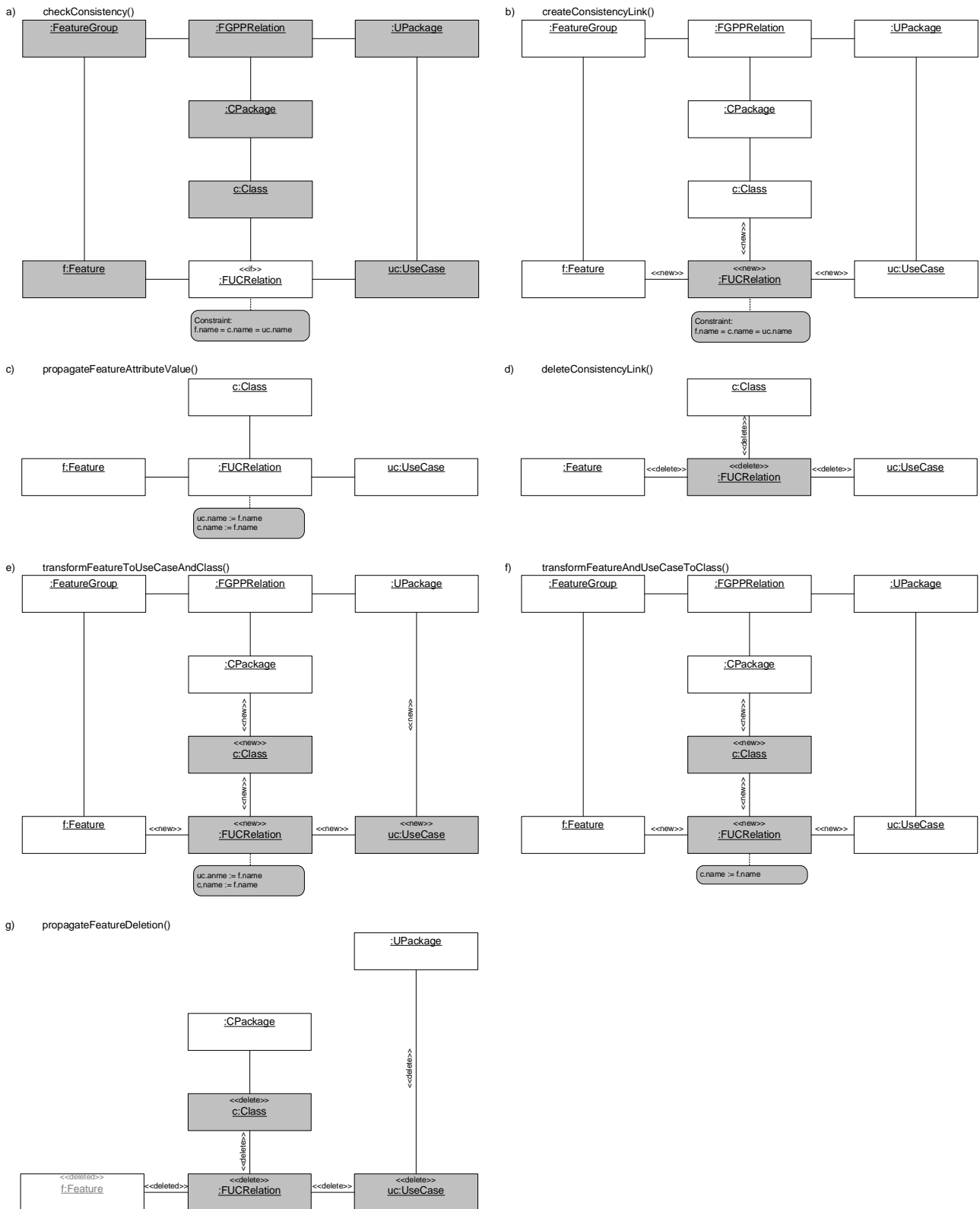


Figure 12: Derived operational rules

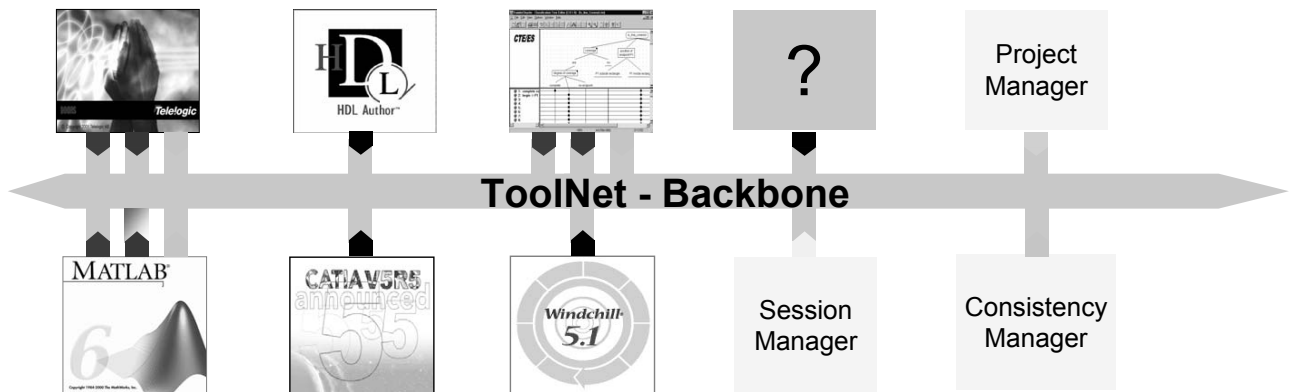
rule and the creation of more than one `FUCRelation` correspondence link between a regarded set of `Feature`, `UseCase`, and `Class` nodes.

The rule of Figure 12c represents the propagation of an attribute value from a `Feature` to a `UseCase` and a `Class` respectively. In order to perform the attribute propagation we must derive attribute assignments from the constraint. This is not trivial in the general case as already mentioned previously. We do not address this problem here, but are planning to use constraint solving techniques in the future in order to be able to deal with more general forms of attribute constraints. Accordingly, we derive two attribute propagation rules starting from a `UseCase` or a `Class`. Figure 12d depicts a rule which just deletes the `FUCRelation` link between the attached `Feature`, `UseCase`, and `Class`.

The rule shown in Figure 12e transforms a `Feature` into a `UseCase` and a `Class`. More precisely, this rule adds a `UseCase` and a `Class` and links them to the `Feature` by a new `FUCRelation`. The new `UseCase` will be connected to the `UPackage`, whereas the `Class` will be connected to the `CPackage` which are linked to the `FeatureGroup` by a `FGPPRelation` which is connected to the `Feature`. The attribute values of the `UseCase` as well as the attribute values of the `Class` are set by the attribute assignments 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`, `UseCase`, and `Class` nodes. Again, we derive two more transformation rules starting from a `UseCase` or a `Class`.

A similar transformation rule is illustrated in Figure 12f. This rule transforms a `Feature` and a `UseCase` into a `Class`. The new `Class` will be linked to the given `Feature` and `UseCase`. The `Class` will be inserted in a `CPackage` which is linked to a `FeatureGroup` containing the `Feature` and to a `UPackage` containing the `UseCase` by a `FGPPRelation`. Likewise, we get two more transformation rules starting from a `Feature` and a `Class` or a `UseCase` and a `Class` respectively.

Finally, Figure 12g shows a rule that examines a dangling `FUCRelation`. This `FUCRelation` is still connected to a `UseCase` and a `Class` but the `Feature` has been deleted after the creation of the correspondence link some time ago. The fact that the regarded `FUCRelation` is no longer connected to a `Feature` is expressed by drawing a gray `Feature` node with stereotype `deleted`. The rule propagates this deletion by deleting the `FUCRelation`, the `UseCase`, and the `Class`. Another time, we derive two additional deletion propagation rules starting from a `UseCase` or a `Class`. Variants of these rules may

Figure 13: Architecture of *Toolnet*

be defined which just delete a dangling correspondence link leaving the possibly still existing objects at the other ends intact.

We have shown that we are able to automatically derive fifteen operational integration rules from one declarative multi graph grammar rule in our running example. This number exponentially increases with the number of regarded graphs (documents, tool data repositories). This fact is one of the most important strengths of our declarative approach because the user only has to specify one declarative rule by himself instead of all operational integration and transformation rules.

6 Code generation and application to practice

In order to realize the desired tool integration we implement our approach as part of the *Toolnet* framework [A⁺03]. Toolnet provides the infra-structure for inter-tool communication, access to the tools' data by means of JMI-compliant tool adapters, and a guided user interface to make use of correspondence links and their underlying consistency rules. Toolnet is designed as a distributed tool integration framework. The communication between different network nodes is realized on top of *SOAP services*. Figure 13 roughly illustrates the architecture of Toolnet.

From the derived operational rules we introduced in Section 5 we generate executable code which can be plugged into Toolnet. Figure 14 presents the whole rule derivation and code generation process. Our approach is based on top of the meta-modeling and graph rewriting environment *FUJABA* [Sof05].

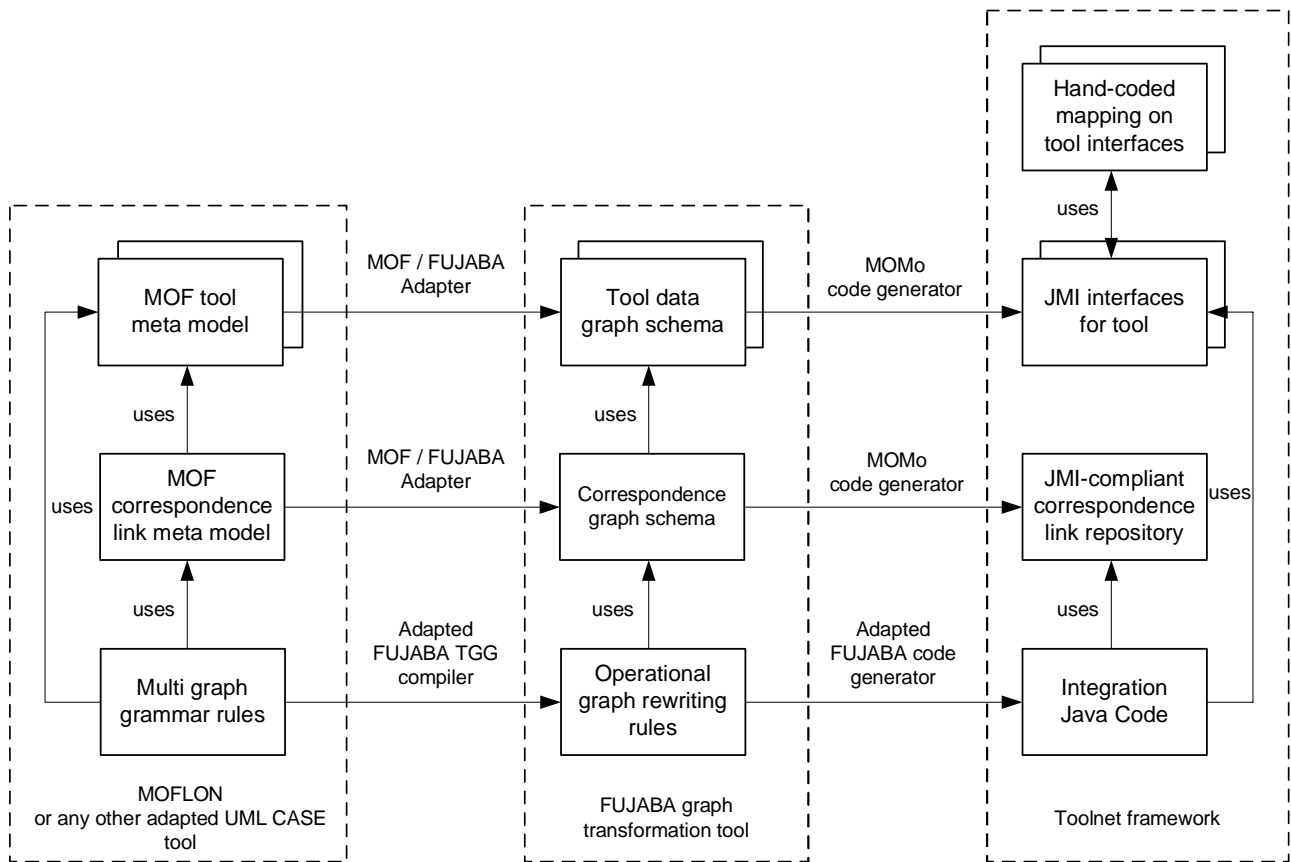


Figure 14: Code generation process in our approach

Since FUJABA only supports UML 1.4 our group implemented an editor plug-in which allows for the specification of MOF 2.0-compliant meta models as part of our *MOFLON* project [Rea05]. Furthermore, we adapted the existing TGG plug-in [Wag01], and implemented the rule derivation strategy presented in the last section. At the moment the generation of executable code from the operational rules has to be done by hand. We plan to adapt *MOFLON*'s code generator which is able to generate JMI interfaces and implementations for a given meta model to generate JMI-compliant code for our graph rewriting rules as well. Currently, *Toolnet*'s *Consistency Manager* component makes use of the operational integrations rules *a*, *b*, and *d* shown in Figure 12. The integration of the remaining rules is ongoing work.

In order to apply the code generated from the different operational rules for doing the integration

tasks we need strategies how to process the data stored in each tool.

The basic idea of our approach is that we visit the data objects of one tool one by one with respect to a hierarchical structure of the document (specified in its meta model). That means that we start with the root objects which are objects that do not have a parent. Having processed these objects we iterate by regarding the children of the root objects. We continue by processing their children, and so on. Topological sorting “on the fly” guarantees that objects on the same hierarchical level are visited in the right order. That means that processing of an object that requires other objects to be processed first will be postponed.

For each regarded object we determine the set of applicable link creation rules based on the type of this object. For each rule of the set of applicable rules we compute the set of possible link partner objects in the other documents based on the given rule’s pattern. If there are more applicable rules which provide multiple possible link partners we have to choose which rules will be applied and which link partners will be connected. For this purpose we introduce priorities for rules, and choose the rules with the highest priority. Another possibility is to ask for user interaction. Having computed which rules will be applied we link all possible link partners provided by these rules. At this point we could ask for user interaction as well.

We can reuse the idea of this strategy for applying the forward and backward transformation rules as well. The consistency checking, the attribute and deletion propagation, and the consistency link deletion rules are based on given correspondence links. Therefore, we do not need sophisticated application strategies for them.

7 Related work

In this section we explain the relationships of our approach to the field of *model driven application development* (MDA) [KWB03] and OMG’s request for proposals (RFP) [OMG02] 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 [Nag96]. 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. [B⁺94] and the just published special sections on tool integration issues of the two Springer journals SoSym [SD04] and STTT [DS04].

7.1 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 generate 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 [?]. 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 combines the advantages of declarative and graph transformation approaches.

7.2 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 results of this comparison are shown in Figure 15.

The submission from the *QVT-partners* [QVT03] 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 denoted⁵. Furthermore, the rules are not declarative. That means that the rules for forward and backward model transformations are not

⁵an additional graphical visualization has been defined, but its role for the definition of model transformations is unclear

automatically derived and must be specified manually⁶. 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.

The goal of the graph transformation system *GReAT* [AKS03] 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 the data of the regarded tools as a simple model (graph) and defines transformations (graph rewriting rules) on it. 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 manipulation of sets of rule matches. Additionally, it offers sophisticated control structures as *sequences*, *non-determinism*, *hierarchical expressions*, *recursion*, and *branching*.

Like GReAT the *BOTL* approach [Bra04] 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.

The *xlinkit* approach [N⁺02] 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*.

Finally, we should mention *IMPROVE* [BW03] 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.

⁶The latest upcoming revised QVT submission looks more similar to our TGG approach. It is future work to do a detailed comparison

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 approach inherited from the graph transformation system PROGRES [SWZ99] 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.

Figure 15 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 has been extended for multi-document integration as explained in the last section. On the other hand, the currently implemented tool integration framework is not incrementally working as the IMPROVE approach; most COTS tools 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.

8 Conclusion

In this paper we have presented a *declarative model integration approach* which combines OMG's meta-modeling standard MOF and the multi graph grammar based tool/data integration approach based on triple graph grammars which have been invented about 10 years ago. This approach does not

	QVT	GReAT	BOTL	xlinkit	IMPROVE	MDI
Declarative Specifications	(+)	-	+	+	(+)	+
Consistency Checks	+	-	(+)	+	+	+
Model Transformations	Unidirectional	Unidirectional	Bidirectional	-	Bidirectional	Multidirectional
Notation	Textual (Graphical)	Graphical	Graphical	Textual	Graphical	Graphical
Meta-Model Based	+	+	+	-	+	+
MOF-compliant	+	-	-	-	-	+
Code Generation	+	+	+	-	-	JMI-compliant
Traceability Support	(+)	-	-	+	+	+
Change Propagation	?	-	-	-	(Incremental)	(Incremental)
Multi Document Support	-	+	-	-	-	+

Figure 15: Classification of tool integration approaches

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. Finally, we have discussed the state of the implementation of our meta-modeling tools which support the presented integration approach. It has become clear that the adaption of these tools to the world of OMG standards is still ongoing work.

In addition, 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 the others. In addition we still have problems with the incremental propagation of updates of models for the following reasons: First of all, JMI interfaces to (meta) data repositories do not offer a notification mechanism for model updates. Furthermore, it is the subject of ongoing joint research activities in the EU network SegraVis⁷ to develop incremental pattern matching mechanisms for attribute graph grammars which use graph update events to keep track of all matches of a set of rules in a given graph. As a consequence, we have to resort to batch-oriented techniques right now, which compare models on demand only and generate reports for missing or inconsistent correspondence links, still needed updates, etc. Thus, we are able to deal with tool integration scenarios, where consistency checking of the data of different tools is not a continuous process, but triggered from time to time at certain points in an engineering process.

Finally, we will have to do a detailed comparison with and adaption to the latest upcoming revised QVT submission from the *QVT-Merge Group* since it looks similar to our TGG approach and,

⁷<http://www.segravis.org>

probably, can be extended to a multi-domain-integration approach in a similar fashion as TGGs have been extended to multi graph grammars in this paper.

References

- [A⁺03] Altheide et al. An Architecture for a Sustainable Tool Integration. In Dörr and Schürr, editors, *TIS 2003 Workshop on Tool Integration in System Development*, pages 29–32, 2003. <http://www.es.tu-darmstadt.de/english/events/tis/documentation/Proceedings.pdf>.
- [AKS03] Agrawal, Karsai, and Shi. Graph Transformations on Domain-Specific Models. Technical report, Institute for Software Integrated Systems at Vanderbilt University, 2003.
- [B⁺94] Brown et al. *Principles of CASE Tool Integration*. Oxford University Press, 1994.
- [Bor] Borland. Together. <http://www.borland.com/together>.
- [Bra04] Braun. *Metamodellbasierte Kopplung von Werkzeugen in der Softwareentwicklung*. Logos Verlag, 2004. German, PhD thesis.
- [BW03] Becker and Westfechtel. Incremental Integration Tools for Chemical Engineering: An Industrial Application of Triple Graph Grammars. In *29th Intl. Workshop Graph-Theoretic Concepts in Computer Science*, volume 2880 of *LNCS*, pages 46–57, 2003.
- [DS04] Dörr and Schürr, editors. Special Section on Tool Integration Applications and Frameworks. *Journal of Software Tools for Technology Transfer*, 2004. Springer-Verlag.
- [EEKR97] Ehrig, Engels, Kreowski, and Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific Publishing, 1997.
- [EEKR99] Ehrig, Engels, Kreowski, and Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2. World Scientific Publishing, 1999.
- [JSZ96] Jahnke, Schäfer, and Zündorf. A Design Environment for Migrating Relational to Object Oriented Database Systems. In *Proc. of the International Conference on Software Maintenance*, pages 163–170. IEEE Computer Society Press, 1996.

- [KS05a] A Königs and A Schürr. Multi-domain integration with mof and extended triple graph grammars. In J Bezivin and R Heckel, editors, *Language Engineering for Model-Driven Software Development*, volume 04101 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum, 2005.
- [KS05b] A Königs and A Schürr. Tool integration with triple graph grammars - a survey. *Electronic Notes in Theoretical Computer Science*, 2005. submitted for publication.
- [KWB03] Kleppe, Warmer, and Bast. *MDA Explained*. Addison-Wesley, 2003.
- [LS96] Lefering and Schürr. Specification of Integration Tools. In Nagl, editor, *Building Tightly-Integrated Software Development Environments: The IPSEN approach*, volume 1170 of *LNCS*, pages 440–456. Springer Verlag, 1996.
- [Mos02] Mosher. *A New Specification for Managing Metadata*. Sun Microsystems, 2002. <http://java.sun.com/developer/technicalArticles/J2EE/JMI/>.
- [N⁺02] Nentwich et al. xlinkit: A Consistency Checking and Smart Link Generation Service. In *ACM Transactions on Internet Technology*, volume 2, pages 151–185, 2002.
- [Nag79] Nagl. *Graph-Grammatiken*. Vieweg Press, 1979. German.
- [Nag96] Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, volume 1170 of *LNCS*. Springer Verlag, 1996.
- [NNZ00] Nickel, Niere, and Zündorf. The FUJABA Environment. In *Proc. of the 22nd International Conference on Software Engineering*, pages 742–745, 2000.
- [NSM99] Nagl, Schürr, and Münch, editors. *Proc. of 1st Intl. Workshop Applications of Graph Transformations with Industrial Relevance*, volume 1779 of *LNCS*. Springer-Verlag, 1999.
- [OMG02] OMG. *Request for Proposal: MOF 2.0 Query/Views/Transformations RFP*, 2002. <http://www.omg.org/cgi-bin/doc?ad/2002-04-10>.
- [OMG03a] OMG. *MOF 2.0 Specification*, 2003. <http://www.omg.org/cgi-bin/doc?ad/2003-04-07>.

- [OMG03b] OMG. *OCG specification*, 2003. <http://www.omg.org/cgi-bin/doc?ad/2003-01-07>.
- [OMG03c] OMG. *UML 2.0 superstructure specification*, 2003. <http://www.omg.org/cgi-bin/doc?ptc/2003-08-02>.
- [PNB03] Pfaltz, Nagl, and Böhlen, editors. *Proc. of 2nd Intl. Workshop Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *LNCS*. Springer-Verlag, 2003.
- [PR69] Pfaltz and Rosenfeld. Web Grammars. In *Int. Joint Conference on Artificial Intelligence*, pages 609–619, 1969.
- [Pra71] Pratt. Pair Grammars, Graph Languages and String-to-Graph Translations. In *Journal of Computer and System Sciences*, volume 5, pages 560–595. Academic Press, 1971.
- [QVT03] QVT-partners. *QVT-partners revised submission to QVT*, 2003. <http://qvtp.org/downloads/1.1/qvtpartners1.1.pdf>.
- [Rea05] Real-Time Systems Lab, Darmstadt University of Technology. *MOFLON*, 2005. <http://www.moflon.org>.
- [RS97] Rekers and Schürr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing*, 8(1):27–55, 1997.
- [Sch70] Schneider. Chomsky-Systeme für partielle Ordnungen. Arbeitsbericht 3,3, Institut für Mathematische Maschinen und Datenverarbeitung, Erlangen, 1970. German.
- [Sch94] Schürr. Specification of graph translators with triple graph grammars. In Mayr and Schmidt, editors, *Proc. WG'94 Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163, 1994.
- [SD04] Schürr and Dörr, editors. Special Section on Model-based Tool Integration. *Journal of Software&Systems Modeling*, 2004. Springer-Verlag.
- [Sof05] Software Engineering Group, University of Paderborn. *FUJABA*, 2005. <http://www.fujaba.de>.

- [SWZ99] Schür, Winter, and Zündorf. The PROGRES Approach: Language and Environment. In Ehrig, Engels, Kreowski, and Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, pages 487–550. World Scientific Publishing, 1999.
- [Tel] Telelogic. DOORS. <http://www.telelogic.com/products/doorsers/doors>.
- [Wag01] Wagner. Realisierung eines diagrammübergreifenden Konsistenzmanagement-Systems für UML-Spezifikationen. Master's thesis, Universität Paderborn, 2001. German.