Int J Softw Tools Technol Transfer (2004) 6: 203–218 / Digital Object Identifier (DOI) 10.1007/s10009-004-0155-8

# Tool integration at the meta-model level: the Fujaba approach

**Sven Burmester[1,*], Holger Giese[1,**], Jörg Niere[2], Matthias Tichy[1,**], Jörg P. Wadsack[1], Robert Wagner[1,***], Lothar Wendehals[1,****], Albert Zündorf[3]**

[1] Software Engineering Group, Department of Computer Science, University of Paderborn, Germany
e-mail: {burmi,hg,mtt,maroc,wagner,lowende}@uni-paderborn.de
[2] Software Engineering Group, Department of Electrical Engineering and Computer Science, University of Siegen, Germany
e-mail: joerg.niere@uni-siegen.de
[3] Software Engineering Research Group, Department of Electrical Engineering and Computer Science, University of Kassel, Germany
e-mail: albert.zuendorf@uni-kassel.de

Published online: 11 November 2004 – 

**Abstract.** Today's development processes employ a variety of notations and tools, e.g., the Unified Modeling Language UML, the Standard Description Language SDL, requirements databases, design tools, code generators, model checkers, etc. For better process support, the employed tools may be organized within a tool suite or integration platform, e.g., Rational Rose or Eclipse. While these tool-integration platforms usually provide GUI adaption mechanisms and functional adaption via application programming interfaces, they frequently do not provide appropriate means for data integration at the meta-model level. Thus, overlapping and redundant data from different "integrated" tools may easily become inconsistent and unusable. We propose two design patterns that provide a flexible basis for the integration of different tool data at the meta-model level. To achieve consistency between meta-models, we describe rule-based mechanisms providing generic solutions for managing overlapping and redundant data. The proposed mechanisms are widely used within the Fujaba Tool Suite. We report about our implementation and application experiences.

**Keywords:** Tool coupling – Meta-Model Extension – Meta-Model Integration – Patterns – Consistency

## 1 Introduction

In recent decades, the usage of different software tools has grown massively in nearly all industrial areas of product development. The various aspects of the product under development are described using different software tools specialized for one particular purpose. Altogether the software tools produce documents, and often the output of one software tool is used as input for another tool. This inevitably leads to the idea of tool coupling.

Coupling tools is usually done by exchanging the produced documents. Unfortunately, most tools have their own document format. This makes the exchange of documents between tools or the sharing of document parts, in means of interfaces, hard to achieve. In recent years XML has made the exchanging task easier but left unresolved the problem of using different tools for different purposes.

Most tools are domain specific, i.e., they offer certain functionality for their domain. In the best case, one document, produced by one or a group of tools, describes only one specific part of the product, but usually the information overlaps.

Overlapping information requires consistency for all documents. Unfortunately, such a product-consistency management system exists only for closely related documents. Therefore, the documents are often inconsistent in their overlapping parts, which results in expensive process overhead for managing the inconsistencies.

Another observation often made in practice is that over time the product is extended, which frequently forces the need for new tool functionalities. Usually, these new functionalities are not supported by the existing tools and the existing tools do not allow for the enhancement of their functionality. Switching to a completely new tool is an expensive task and usually not an option. Adding a specific new tool for just the required functionality and establishing and maintaining a consistency management

* Supported by the International Graduate School of Dynamic Intelligent Systems, University of Paderborn

** This work was developed partially in the course of the Special Research Initiative 614 – Self-Optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft (DFG).

*** This work has been supported by the DFG grant GA 456/7 ISILEIT as part of the SPP 1064.

**** This work is part of the FINITE project funded by the DFG, project no. SCHA 745/2-2.

system for the new tool and the existing ones takes too much time. Therefore, the resulting inconsistencies produce additional process overhead and costs.

To reduce the process overhead, and thereby the additional costs, one solution to overcome the overlapping information and the inherent consistency problem is an integrated development approach. In the following discussion we call such an integrated development environment a *Tool Suite*. In Tool Suites different tools interoperate on a common meta-model with a common consistency management system. The problem to be solved is how to handle the dependencies between the different tools in the Tool Suite and to keep them separated as much as possible in order to prevent one monolithic application.

In this article we present a flexible mechanism based on our *Meta-Model Extension* and *Meta-Model Integration* patterns to integrate different tools on the meta-model level. The benefits of our patterns are that they may be instantiated in a new system design as well as in an old system design. In addition, we present a flexible consistency management system, especially suited for the integration of different or enhanced meta-models.

The paper is organized as follows. Section 2 first presents general problems coming up when integrating tools in a common environment and then presents related work. Section 3 gives an overview of our approach to integrating tools at the meta-model level. Then, Sects. 4 and 5 go into details of the *Meta-Model Extension* and *Meta-Model Integration* patterns as well as into details of our consistency management system. As an application of our proposed solution to coupling tools at the meta-model level, Sect. 6 presents our experiences with the Fujaba Tool Suite, which overcomes the problems mentioned above by means of an extensible tool-integration framework. We close the article with a conclusion.

## 2 Problem description

In general, tool integration has four main problems: (1) integration of (graphical) user interfaces, (2) integration of tool functionality, (3) integration of tool data and metadata, and (4) consistency management during tool integration.

The first problem area is the adaption of the (graphical) user interface of one tool by another. This includes the extension of menus, tool bars, and dialog windows as well as the extension of the representation of certain data at the user interface. The former points are frequently supported, e.g., via (extensible) GUI configuration files, whereas the last point is seldom supported.

The second problem area is the integration of functionality. First, this means that one tool uses some functionality of another tool. This is usually easy to achieve via some application programming interface (API). Second, one tool wants to modify and enhance a certain functionality of another tool. This requires mechanisms like the template method, chain of responsibility design patterns [9], or a listener concept for command executions. A tighter coupling of different functionality or even a replacement of existing functionality with an alternative implementation is not always supported. Another opportunity that has come about with the advent of net-centric computing is services installed on one or more instances and retrievable via a central lookup service [1, 31]. Enhancing or exchanging of functionality then means providing updated or new services or altering service connections.

The previous two problem areas are well understood. There are several solutions and implementations, e.g., XML-based GUI configuration files describing the adaption of GUIs, especially for menus and actions, or application server solutions.

The third problem area comprises the access and extension of a tool's data and metadata. Especially for a tight integration of the tools the integration of the meta-model is a remaining challenge, which is the focus in this article.

Concerning the access and extension of a tool's data and metadata, we differentiate between two categories of tools. The first category contains tools designed and developed without a facility to access their meta-model. Most older tools belong to this category, and a posteriori integration means coupling the tools via their import and export interfaces. For integration of data in different formats, typical solutions are standardized exchange formats, e.g., XMI. A comparison on how and where the schemas are defined, i.e., organization of the model data to be exchanged, is given in [17]. In this category integration of functionality is not feasible and usually an independent tool serves as a GUI integration platform, e.g., a favorite folder, which is a very simple solution.

The second category contains tools allowing other tools to access their meta-model via a defined interface, or allowing other tools to enhance their meta-model and access the data directly. Such an integration often provides means to deal with overlapping artifacts.

Integrating and exchanging overlapping artifacts is, however, not a new problem. The Ipsen approach [21] presented a framework to integrate tools through a common meta-model. However, this approach is not easily applicable to the integration of existing tools because integration often means duplicating the meta-models – one copy in the original tool and one in the Ipsen approach underlying the common meta-model.

The Corum approach [32] suggests the usage of a common information model that is used by all tools. For the integration of tools that are not based on the Corum approach and that cannot use the Corum API, input can be generated by means of transformation tools. In [18] the Corum II approach is presented, integrating different reengineering tools operating on different levels.

A prominent tool-integration platform is Eclipse, which is promoted by several midsize and major IT enterprises. The supported integration aspects are restricted to a framework built on a mechanism for discovering, integrating, and running plug-ins. One plug-in is the Eclipse Modeling Framework (EMF). It is a Java/XML framework for generating tools and other applications based on simple class models. Although EMF provides the foundation for interoperability with other EMF-based tools, support for tight meta-model coupling and integration between these tools is not provided. Further, to the best of our knowledge, most UML tools like Together Control Center or Rational Rose are also only extensible via APIs.

The fourth problem area when integrating tools is consistency management. On the one hand, it is clear that a certain consistency management system is closely related to the solution taken to integrate metadata and the possibility of accessing the data. On the other hand, consistency management with respect to tool integration consists of some general problems and requirements. For example, separation of concerns on the one hand reduces the complexity of the overall specification. On the other hand, the increasing number of used notations very often leads to a wide range of inconsistencies [10, 11], e.g., syntactical inconsistencies violating the well-formedness of models, behavior inconsistencies between different diagrams [6], or inconsistencies during refinement of diagrams [4]. Of course, the general problem of inconsistency is much broader and comprises a large number of disciplines. For a survey we refer the reader to [23] and for a research agenda to [7].

The built-in consistency management of most of today's tools is not satisfactory. One reason for this is that tools often try to enforce consistency and do not support temporal inconsistencies during development. In [2, 23], Nuseibeh et al. emphasize the importance of tolerating inconsistencies and propose a conceptual framework for inconsistency management that allows inconsistencies to be ignored, deferred, circumvented, ameliorated, or resolved. In addition, tools that support consistency checking on user demands only [24, 26] have to check the whole specification over and over again, even if no changes were made.

A database reengineering approach that supports temporary inconsistencies is presented in [14, 15], hence reengineering means handling partly correct information, hypotheses, or contradictory indications. The presented consistency management system uses so-called Triple Graph Grammars, cf. [20, 28], which are more or less hard-coded, because the application domain is fixed, i.e., database reengineering.

In cases of a flexible environment for various application areas, we need a consistency management system that keeps different connected models of different meta-models consistent. This demands a flexible consistency rule set extension and management accompanied by the corresponding rule execution mechanism.

## 3 The Fujaba integration approach

The integration approach presented in this paper was developed by the Fujaba Team. We implemented this approach in the Fujaba Tool Suite, which is presented in Sect. 6.

### *3.1 Plug-in mechanism*

Plug-ins are a common mechanism for adding third-party software to a basic environment. Throughout this article we call such a basic environment a Tool Suite. A plug-in has to provide information about its version, its dependencies on other plug-ins, and its developers. Additionally, a plug-in can change menus and menu entries as well as popup menus and toolbars via XML-based GUI configuration files. Additionally, it is possible for plug-ins to notice the execution of menu actions of other plug-ins. Every tool that participates in a Tool Suite is a plug-in. Fujaba's Integration Approach supports two plug-in interoperability variants.

First, an existing tool (plug-in) is extended by a new plug-in, i.e., the meta-model of the existing tool is extended in the new plug-in (Fig. 1). Second, two existing tools (plug-ins) are integrated, i.e., the meta-model instances of two plug-ins are linked together and kept consistent by a third plug-in (Fig. 2).

Figure 1 shows a meta-model extension for an existing ToolSuite. The ≪uses≫ dependency is unidirectional; thus the ToolSuite is not dependent on its plug-ins. The ToolA plug-in is a tool with its own meta-model. ToolB is a plug-in that extends the meta-model of ToolA. In terms of compiling, this means that ToolA can be compiled with the ToolSuite only, whereas ToolB needs the ToolSuite and the ToolA plug-in for compilation.

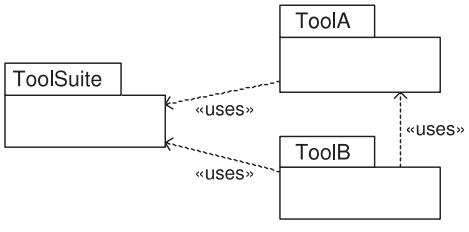


**Fig. 1.** Tool extension

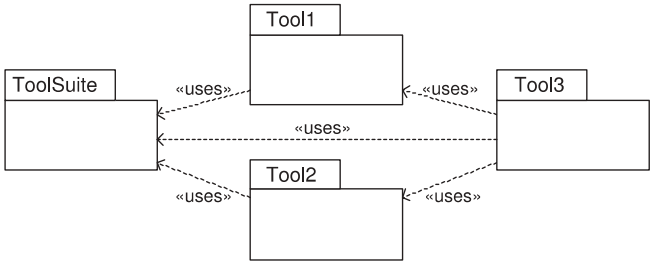


**Fig. 2.** Tool integration

Assume that two independent (existing) tools, plug-in Tool1 and plug-in Tool2, should be used together in a consistent manner. The meta-models of these two plug-ins are independent. The aim is to enable the use of both Tool1 and Tool2, while integrating their meta-models. Figure 2 shows such a meta-model integration within the Tool3 plug-in. Tool3 establishes the link between the two disjunct meta-models of Tool1 and Tool2. Hence, Tool1 and Tool2 are separately compilable. Of course, Tool3 needs Tool1 and Tool2 to compile.

### *3.2 Bidirectional associations*

A fundamental problem for extensible frameworks is providing the possibility to integrate meta-models even though the different meta-models might be in independently compilable parts (plug-ins) of the software system. In particular, bidirectional associations produce mutual dependencies between classes.

There are different ways to implement associations between meta-model elements (classes). One way is to use an attribute in a class A to reference another class B. This models a unidirectional association where class B does not know about its reference from class A. Keeping unidirectional associations consistent is a difficult task. Deleting an object of class B may result in an invalid reference within an object of class A.

Another solution is to use bidirectional associations. They can be created by mutual references between both classes, which is usually done by adding a public attribute to each class referencing the other class. When creating a link between two objects, both attributes are set to reference each other. Even in this scenario there could be problems preserving the consistency. Since the attributes can be set independently of each other, invalid references are still possible.

We propose a smarter solution. Instead of using public attributes, we recommend private attributes with public access methods. The access methods can be used to encapsulate a mechanism that ensures a consistent mutual referencing. When setting a link between two objects by using the access method of one object, the access method of the other method will be implicitly called to set the opposite reference. Even when deleting a link between two objects, both references will be erased. See [8] for more details.

Other solutions to implementing bidirectional associations use adapters or observers (cf. [9]) or generate code according to JMI [16]. All solutions embed considerable overhead, i.e., (external) tables that store the bidirectional associations (adapter) or at least an observer for the generated association-interface that listens to the participating classes. However, these solutions result in different ways of dealing with intra- and intertool associations, which in turn results in maintenance and consistency problems.

In monolithic applications, such a tight integration of different meta-models is not problematic since the classes just have to be connected via bidirectional associations. In an extensible framework this solution is obviously not appropriate since the different parts (plug-ins) would not even compile separately.

Therefore, we propose a solution that preserves the separation of core and plug-in as well as allows the tight integration of the meta-models. Technically speaking, it must be possible to connect both meta-models bidirectionally without explicitly adding references to the meta-models.

### *3.3 Running example*

We present our approach using a running example. The ISILEIT project [19], related to embedded systems, requires a hybrid specification with the Unified Modeling Language (UML) and the Specification and Description Language (SDL). SDL was developed for telecommunications engineering with a special focus on formal semantics for process communication. We used the FUJABA TOOL SUITE with support for UML class diagrams as a basic environment and added support for SDL.

SDL block diagrams are used to specify the static communication structure where processes are connected by channels and signal routes. From an SDL block diagram an initial UML class diagram is derived, where each process is represented by a class. In addition, each signal received by a process in the SDL block diagram is mapped to a method of the corresponding class in the UML class diagram. The class diagram can then be refined by state charts for each class.

We added an SDL block diagram meta-model as a plug-in that extended the UML class diagram meta-model. Further, the SDL plug-in restricted the UML class diagram meta-model by prohibiting multiple inheritance for code generation reasons. This scenario corresponds to Fig. 1. Consistency maintenance, by means of meta-model restriction, has to be handled in a flexible manner by the new plug-in.

A second embedded-system-related project raises the need for a stand-alone SDL tool (plug-in). Nevertheless, the integration with UML is still a major concern. Thus, we constructed a plug-in that integrates and manages the consistency between UML and SDL (Fig. 2). The integration plug-in has to incorporate a flexible mechanism that guarantees consistency between the integrated meta-models.

## 4 Meta-Model Integration approach

In the following discussion our proposed *Meta-Model Extension* and *Meta-Model Integration* patterns will be described in the style introduced by Gamma et al. [9].

### *4.1 Meta-Model Extension pattern*

Intent

Extend tools on the meta-model level with tight bidirectional but compile-time-independent coupling between meta-model elements.

Motivation

Connecting meta-models is a fundamental task when extending a tool by another tool or plug-in. To handle data consistency between meta-models, data updates have to be propagated in both directions between the meta-models. This requires bidirectional associations between corresponding meta-model elements of the different tools.

Figure 3 shows such a tight connection between two elements in different meta-models using a bidirectional 1-to-N association. Each of both classes references the other class. Links between objects of those meta-model elements can then be navigated in both directions. Thus, a bidirectional ≪uses≫ dependency exists.

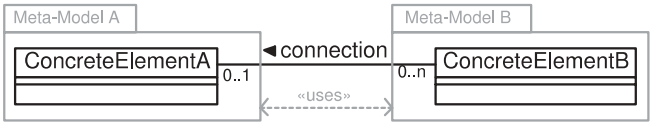


**Fig. 3.** Tight connection between meta-models

However, this bidirectional association creates a mutual dependency. It requires a modification of both tools and particularly prevents a separate compilation of the tools. In addition, the meta-models cannot be deployed separately. These dependencies have to be avoided in order to provide independent tool development and platforms that will be extensible by tools to be developed in the future.

The Meta-Model Extension pattern describes how an existing meta-model can be extended by a new meta-model. The extension does not affect the existing tool but still provides a bidirectional coupling between the new and the existing tool. This scenario is depicted in Fig. 1.

Applicability

Use the Meta-Model Extension pattern when you want to (1) provide an extensible tool platform using a meta-model framework and (2) extend a meta-model of an existing tool based on the meta-model framework.

Structure

In Fig. 4 the Meta-Model Extension pattern is presented. A Meta-Model A is extended by a Meta-Model B without affecting the interface and the compilation of Meta-Model A, but still a bidirectional association between elements of the two meta-models is given.

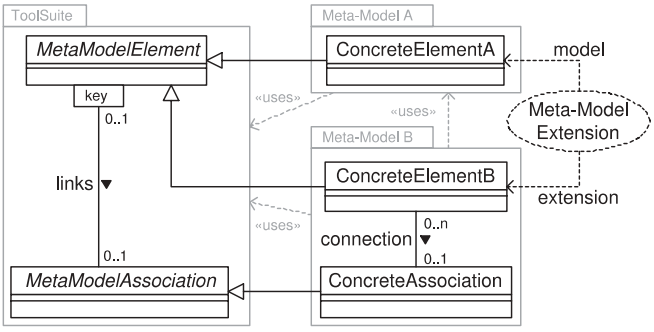


**Fig. 4.** Meta-model extension pattern

The two classes MetaModelElement and MetaModelAssociation are part of the the meta-model framework. This framework is provided by Tool Suites that enable extensions on the meta-model level. All elements of meta-models have to subtype the class MetaModelElement. It provides a qualified association to the class MetaModelAssociation. To create a bidirectional association between ConcreteElementA of Meta-Model A and ConcreteElementB of Meta-Model B as depicted in Fig. 3, a subclass of MetaModelAssociation has to be created and a connection association between ConcreteElementB and ConcreteAssociation has to be established. This connection association should have the same cardinalities as the intended tight association between the meta-model elements. All associations in this pattern are bidirectional. The (name of the) ConcreteAssociation class is used as the key for the qualified links association.

Participants

MetaModelElement. Provided by the meta-model framework. Superclass for all meta-model elements.
MetaModelAssociation. Provided by the meta-model framework. Superclass for all inter-meta-model association classes.
ConcreteElementA, ConcreteElementB. Elements from different meta-models, provided by different tools.
ConcreteAssociation. Association class provided by the extending tool for inter-meta-model links.

Collaborations

The connection association modeled by the additional class ConcreteAssociation can be navigated in both directions. Figure 5 shows how objects of ConcreteElementA and ConcreteElementB are linked via an association object.

It is obvious how to navigate from a ConcreteElementB object b1 via the object link of type ConcreteAssociation to an object a1 of type ConcreteElementA. In the opposite direction from a ConcreteElementA object a1 you can

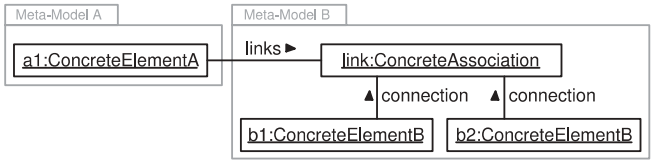


**Fig. 5.** Objects linked via an association object

get the ConcreteAssociation object link by using the class name of the ConcreteAssociation as key and then navigate to an object b1 of type ConcreteElementB.

Consequences

The Meta-Model Extension pattern (1) realizes bidirectional associations between meta-model elements and (2) avoids mutual compile-time dependencies by using generalizations.

Implementation

The above-described navigation via objects of the association class is clumsy compared to navigation via ordinary bidirectional associations. For ordinary bidirectional associations well-known approaches (cf. [8]) exist that use access methods to ease the navigation of associations and the creation and deletion of links. In principle, the same approach can be used to navigate via the association class. Unfortunately, the usage of the Meta-Model Extension pattern implies the navigation of two associations. Additionally, the generic association between MetaModelElement and MetaModelAssociation imposes an additional burden on the developer. Therefore, the source code, which navigates over the object structures built by the pattern, may deteriorate, and the readability suffers.

Thus, for the implementation of the Meta-Model Extension pattern we propose a variant of the access methods described in [8] to hide as much as possible the special nature of the pattern-based association. Typically, access methods are used for creation, check, and deletion of links as well as navigation over links. In the example above, we are creating a ConcreteElementB class that has an association to a not-to-be-changed class ConcreteElementA. Therefore, we may add ordinary access methods to the first class but not to the second. In the case of a to-many association, the access methods for the class ConcreteElementB have the following signatures:

```
public void addToConcreteElementA
            (ConcreteElementA value)
public void removeFromConcreteElementA
            (ConcreteElementA value)
public boolean hasConcreteElementA
            (ConcreteElementA value)
public Iterator iteratorOfConcreteElementA()
```

In case of a to-one association, we employ two access methods:

```
public void setConcreteElementA
            (ConcreteElementA value)
public ConcreteElementA getConcreteElementA ()
```

Inside these methods the usage of an object of the association class can be completely encapsulated. Thus, the application of the pattern is hidden from ConcreteElementB's point of view. The other way around the association class cannot be completely encapsulated by access methods since the ConcreteElementA class must not be changed. Therefore, the access methods must be added to another class. The association class is a natural location for these access methods. Since the exact object of the association class is typically not known and is dependent on the object of the ConcreteElementA class, we propose to add static access methods to the association class that include the object of the ConcreteElementA class as parameter. These static access methods completely hide the existence of an object of the association class from the developer. Therefore, the developer does not need to treat the association based on the Meta-Model Extension pattern in a special way. In particular, the error-prone manual handling of objects of the association class is completely encapsulated in the static access methods. For example, the addTo method first uses the links association between the MetaModelElement and MetaModelAssociation classes to reach the object of the ConcreteAssociation class and then adds the valueB object to the connection association. The encapsulation keeps the two associations consistent with each other such that the two bidirectional associations are equivalent to the needed one bidirectional association between both meta-model element classes.

```
public static void addToConcreteElementB
            (ConcreteElementA valueA,
            ConcreteElementB valueB)
public static void removeFromConcreteElementB
            (ConcreteElementA valueA,
            ConcreteElementB valueB)
public static boolean hasConcreteElementB
            (ConcreteElementA valueA,
            ConcreteElementB valueB))
public static Iterator
            iteratorOfConcreteElementB
            (ConcreteElementA valueA)
```

In the case of a to-one association, appropriate set and get methods are used.

Example

Figure 6 gives an example where the Meta-Model Extension pattern was used to extend a UML meta-model by an SDL meta-model (Sect. 3.3).

In this example a framework for implementing abstract syntax graphs (ASGs) is given. The classes ASGElement and ASGAssociation are part of the framework and implement the MetaModelElement and MetaModelAssociation classes of the pattern. Two meta-models are

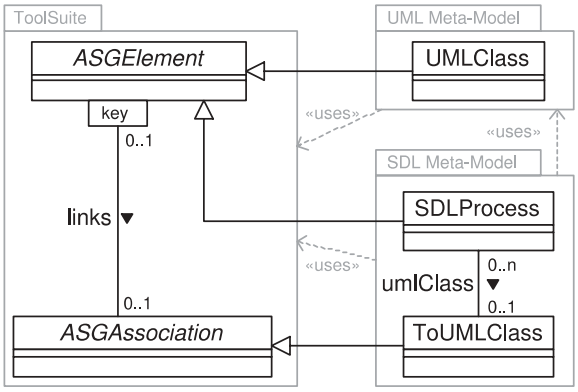


**Fig. 6.** Extension of a UML meta-model by an SDL meta-model

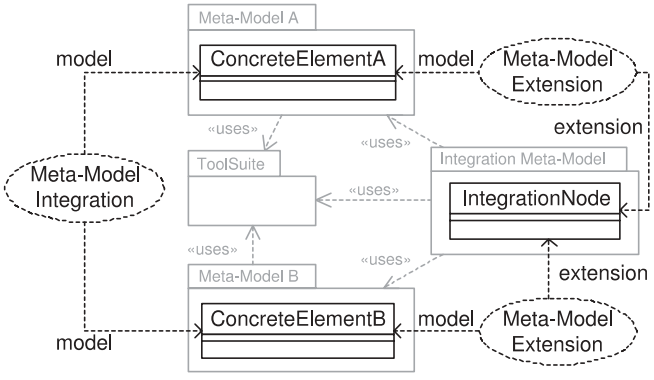


**Fig. 7.** Meta-Model Integration pattern

built on top of this framework, which are a UML class diagram meta-model and a meta-model for SDL.

This example corresponds to Fig. 1, where one plug-in extends another plug-in. Here a plug-in providing SDL block diagrams extends another plug-in providing UML class diagrams by associating SDL processes with UML classes.

### *4.2 Meta-Model Integration pattern*

Intent

Integrate independent tools on the meta-model level with tight bidirectional but compile-time-independent coupling between meta-model elements.

Motivation

The presented Meta-Model Extension pattern limits the reuse of the extending tool as it cannot be used without the extended tool. For the example given in Sect. 4.1 this means that the SDL block diagrams cannot be used without the UML class diagrams. Therefore, a mechanism is needed to integrate two independent tools on the meta-model level without affecting their dependencies. The Meta-Model Integration pattern proposes to use a third integration meta-model that extends the existing tools by using the Meta-Model Extension pattern twice. This scenario is depicted in Fig. 2. The integration meta-model can then be used to preserve consistency between the two meta-models.

Applicability

Use the Meta-Model Integration pattern when you want to integrate two independently developed tools based on the meta-model framework.

Structure

The Meta-Model Integration pattern is based on the double application of the Meta-Model Extension pattern. Figure 7 shows the structure of the pattern.

An Integration Meta-Model is used to connect two independent meta-models A and B. When connecting two elements from Meta-Models A and B an IntegrationNode is created. It is connected via association objects using the Meta-Model Extension pattern. Note that Meta-Models A and B are still independent of each other and can be used separately. Only the Integration Meta-Model depends on Meta-Model A and Meta-Model B.

Participants

ConcreteElementA, ConcreteElementB. Elements from different meta-models, provided by different independent tools.
IntegrationNode. Element from third integration meta-model to connect elements from the existing meta-models.
Meta-Model Extension pattern. Applied twice. Provided partially by the meta-model framework. Two concrete meta-model association classes between the ConcreteElements and the IntegrationNode have to be provided by the Integration Meta-Model.

Collaborations

The integration nodes can be used for preserving consistency between elements of the two meta-models. The integration tool has to ensure the existence of the integration nodes so that changes can be propagated between the connected elements. The integration tool has to create the integration node when meta-model elements are created and delete the nodes when meta-model elements are destroyed.

Consequences

The Meta-Model Integration pattern realizes a tight bidirectional connection between elements of two independent meta-models.

Implementation

The pattern can be implemented using the *Observer* pattern [9]. The integration tool observes the creation and

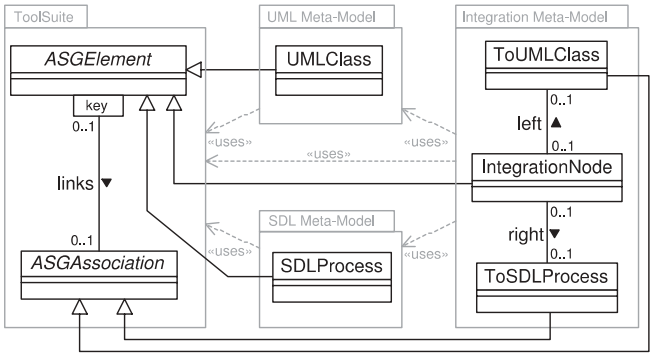


**Fig. 8.** Integration of SDL and UML meta-models

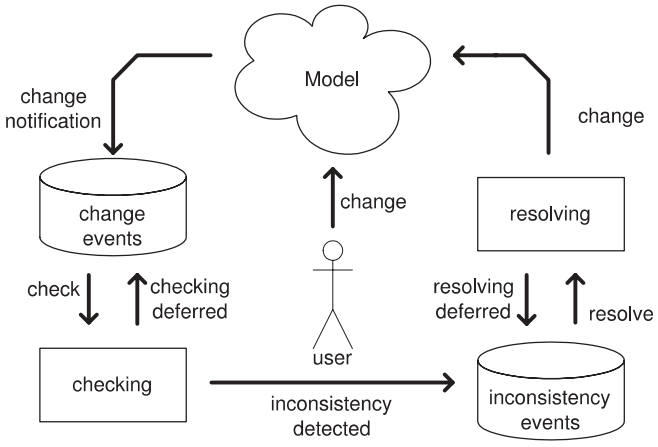


**Fig. 9.** Consistency life cycle of a model modification

deletion of meta-model elements. The integration nodes observe changes of the connected elements.

Example

Figure 8 shows how the UML and SDL meta-models are connected by a third integration meta-model. The first Meta-Model Extension pattern is applied to the components ToolSuite, UML Meta-Model, and the Integration Meta-Model. The second Meta-Model Extension pattern is applied to the components ToolSuite, SDL Meta-Model, and the Integration Meta-Model.

Consistency is a vital part of the Meta-Model Integration pattern. Ensuring consistency by manually implementing observers for each constraint is a tedious and error-prone task. To reduce the effort, a consistency mechanism for checking declarative rules is presented in the following.

## 5 Consistent tool integration

As outlined in Sect. 2, the problem of inconsistency detection and management is central to the development of large and complex models. Thus, sufficient support for model consistency is a crucial prerequisite for the successful and effective application of computer-aided development tools.

The requirements for a suitable consistency management approach have to provide an incremental checking of consistency rules as well as the ability to tolerate detected inconsistencies. In the case of tool integration, the set of required consistency rules and integration constraints results from the individual combination of independently developed tools. Thus, it is not feasible for a tool developer to identify consistency rules for all possible integration scenarios in advance. It must be possible to specify required consistency rules outside each individual tool. Resulting conflicts between contradictory rules and repair actions have to be managed at runtime.

We first outline how the execution of such rules has to be organized and then present two alternatives specifying the required consistency rules and repair actions.

### *5.1 Consistency checking*

To understand what a consistency rule evaluation should look like, we have to understand the different phases of (1) detecting a possible inconsistency, (2) checking that it is indeed an inconsistency, and (3) resolving the inconsistency with respect to a given consistency rule. The general life cycle of a model modification concerning consistency is depicted in Fig. 9.

To detect a possible inconsistency, tools usually wait for the user command to check all rules. A better approach is to observe the model modifications and employ an on-demand consistency management approach that reacts to different change events or specific user commands in a flexible manner. An incremental checking algorithm has to analyze only that fraction of the system that has been modified. Thus, the response time for consistency checking can be reduced drastically.

For each change event we simply have to check whether it results in a modification that is inconsistent. The change events can be processed by a single rule application that usually reports detected inconsistencies. In more complex cases, multiple rules may cooperate via resulting higher-level change events that permit one to decompose the detection of inconsistencies into a sequence of less complex steps.

We distinguish required and optional rules. For required rules the consistency check has to be successfully applied. Thus, if the rule detects an inconsistency, either it is automatically resolved by a related repair action or a compensating undo is required.

For optional rules we allow temporary inconsistencies. If the automatic check does indicate a problem and only a user inspection can determine whether or not it is a real inconsistency, the consistency management should simply keep track of the required manual inconsistency checks and inform the user.

For the final step of resolving an inconsistency, support for automatic as well as manual handling is required. It is required to keep these two last phases separated,

as a proper consistency management system must tolerate for optional rules that a detected inconsistency remains in the model until it is resolved in a later stage of development when the required context information is available.

If an inconsistency is detected, it can be resolved either automatically or manually. In the case of automatic resolution of inconsistencies, some consistency rules and repair actions may introduce new inconsistencies. Even worse, contradictory consistency rules and repair actions can result in a nonterminating chain of change events, checking activities, inconsistency events, and repair activities. In particular, such situations occur often in environments with several integrated tools restricting the meta-model of one tool in different ways. For example, if two tools A and B were integrated using some consistency rules restricting the meta-model of tool B, integrating a third tool C into this environment may introduce further consistency rules on B's meta-model. If the additional consistency rules are contradictory to the already existing rules, an automated repair will result in a cyclic execution of the consistency rules.

An appropriate approach to evaluating a given set of rules thus has to detect such problems and stop the repeated rule application. To prevent such a cyclic execution and to detect contradictory consistency rules and repair actions, our algorithm works in two phases.

In the first phase, all modified elements are checked by executing the appropriate consistency rules. In the case of found inconsistencies, the necessary repair activities are executed. As a matter of course, repair activities modify the model elements themselves, causing further change events. Thus, during a repair all elements modified by a repair action are stored, and it is noticed which consistency rule caused the repair action to be executed.

The second phase works similar to the first phase except for the fact that now the consistency rules are executed only for the previously stored elements collected during the repair activity in the preceding phase. Additionally, the second phase is executed in a cyclic manner where the modified elements collected during a repair serve as input for the subsequent execution phase cycle. Within each phase cycle three different cases can be distinguished:

The first case is if further inconsistencies are detected and if the considered element was modified by another rule in a previous phase cycle. This means that multiple rules cooperate with each other. Thus, the appropriate repair activity can be executed without any risk. Note that, as in the first phase, all modified elements will be stored and used for the next phase cycle.

The second case is if further inconsistencies are detected and if the considered element was modified by the same rule in a previous phase cycle. If this situation holds, either a repair action is faulty or a contradictory rule introduced the inconsistency a second time. In that case a further repair action is not executed. Instead, the user is informed about the detected problem and has to resolve the conflict in the specified consistency rule(s) and/or repair action(s).

In the third case no further inconsistencies are found. This means that in the preceding phase cycle all consistency checking and repair activities were successful. Hence, no further steps are required. Note that the execution terminates if no further modifications are performed during a phase cycle.

### *5.2 Simple graph grammar rules*

To achieve a successful and effective interoperability between integrated tools, additional integration constraints and consistency rules have to be specified. Typical examples for such integration constraints and rules are restrictions on the meta-model of one tool that are needed to achieve a clear interoperability with another tool.

As an example, we employ a constraint from our previously mentioned case study. In our case study, we are generating code for programmable logic controllers (PLC's) [22]. To keep the transformation of the class diagram into a non-object-oriented target language quite simple, we have to restrict the used class diagrams to not include any multiple inheritance between classes.

The Unified Modeling Language (UML) is defined by the abstract syntax of the underlying meta-model [25]. We will use a simplified fragment of this logical model as an example (Fig. 10).

In Fig. 11, a class diagram is represented as an instance of the simplified meta-model fragment depicted in Fig. 10. The objects of type Class represent three classes A, B, and C. The inheritance relationships are represented by instances of Generalization and links connecting the parent classes A and B with the child class C. Note that, since class C inherits from class A as well as from class B, this meta-model instance violates our constraint forbidding multiple inheritance.

For the specification of such constraints and appropriate repair actions we have decided to use graph rewriting

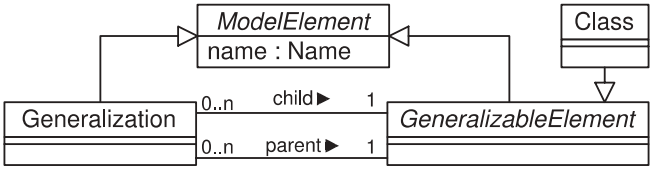


**Fig. 10.** Simplified fragment of the UML meta-model

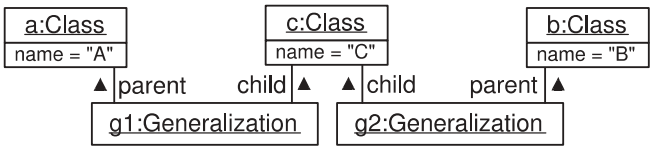


**Fig. 11.** Meta-model instance

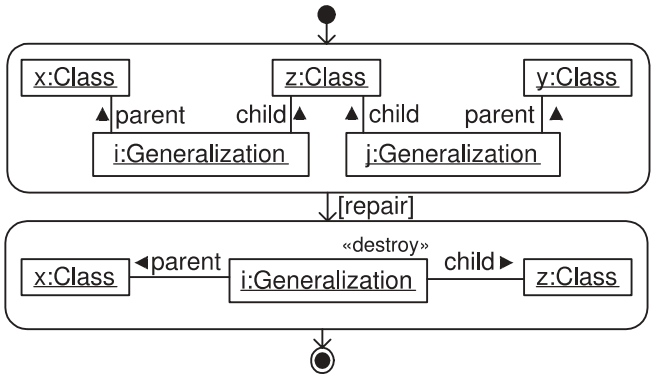


**Fig. 12.** Integration constraint and repair action

rules based on graph grammars [27]. The idea of using graph grammars for the visual specification of consistency rules and appropriate repair actions is not new [3, 5]. We rely on these concepts and show in the following discussion how they can be applied in our tool-integration approach (cf. [29]).

The specification depicted in Fig. 12 consists of two parts. The upper part is the consistency rule itself. It contains a specification of a counterexample, i.e., it specifies an inconsistent situation in the meta-model instance that is not allowed.

In a graph-grammar-based consistency check, a graph rewriting system will search for the pattern, and as soon as a match has been found, an inconsistency will be reported to the user.

The consistency checking can be performed by applying the graph grammar rule as follows: After a change event has occurred, the applicability of the rule is checked by looking for a match of the modified model element in the specified graph grammar. If the element can be mapped to one of the graph grammar nodes, the remaining structure is searched for. If there is no such match in the consistency rule, or if the remaining structure cannot be found, the consistency rule is not applicable for this element. Otherwise, if a match for the whole structure is found, an inconsistency was detected and will be reported to the user.

The lower part of the specification contains the repair action. The repair action is executed only if the upper part of the rule can be applied and automatic consistency resolving is activated. In our example, the repair is simply performed by deleting the Generalization object (stereotype ≪destroy≫) and the links between the parent and the child objects.

Handling inconsistencies may be as simple as adding information to the description or deleting it. Thus the inconsistency handling can also be automated by specifying repair activities using graph grammars [30]. However, in some cases additional information from the user is needed. Thus resolving inconsistencies immediately is not always possible or even desired [2, 23]. Therefore, the automated repair may remain unspecified.

### *5.3 Triple graph grammar rules*

For a tight integration of tools on the meta-model level it is often not sufficient to restrict the individual meta-model of each tool itself. What is additionally needed is a flexible mechanism to specify semantic relationships between syntactically unrelated objects without modifying the meta-models of the tools. This relationship is used to propagate changes in one model to the corresponding model in the other tool and vice versa. This way, consistency between both models can be ensured.

Simple graph grammars as presented in the previous section are usually restricted to the specification of operations on a single meta-model. Therefore, they are not suitable for the specification of integration constraints and rules among different meta-models.

For specifying relationships between different meta-models, we are using a technique based on triple graph grammar theory [28]. In the original work [20], triple graph grammars are used for transformations between different documents. An additional integration document enables a clear distinction between the source and the target document and keeps the extra links needed to preserve the consistency between both documents. For tool integration, each document can be seen as a separate meta-model of the integrated tools (Fig. 13).

A triple graph grammar specification is a declarative definition of a mapping between two meta-models. A triple graph grammar rule is shown in Fig. 14. The rule stems from the already mentioned mapping example where each SDL process is mapped to a UML class and vice versa.

The rule specifies a consistent state between the objects of the integrated tools. It consists of a left-hand and a right-hand rule side. Each rule side contains objects of the underlying meta-model of the tool, i.e., the left-hand side contains instances of the SDL block diagram meta-model and the right-hand side contains instances of the UML class diagram meta-model. The objects are connected by links using the integration model in between

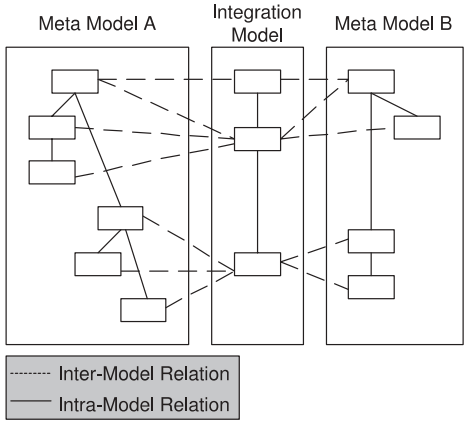


**Fig. 13.** Triple Graph Grammar

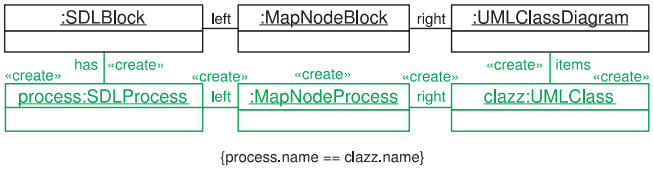


**Fig. 14.** Triple Graph Grammar Rule

both rule sides. An additional constraint specifies that the process and the class name have to be equal.

This declarative specification can be translated into simple graph rewriting rules that are used for consistency checking as well as for automated repair actions. We are creating three graph rewriting rules for each transformation direction. The rules for the consistency preserving operations are presented in Fig. 15. These rules are applied if changes in the left model, i.e., the SDL block diagram, are made. Therefore, these rules are called left to right rules or forward rules.

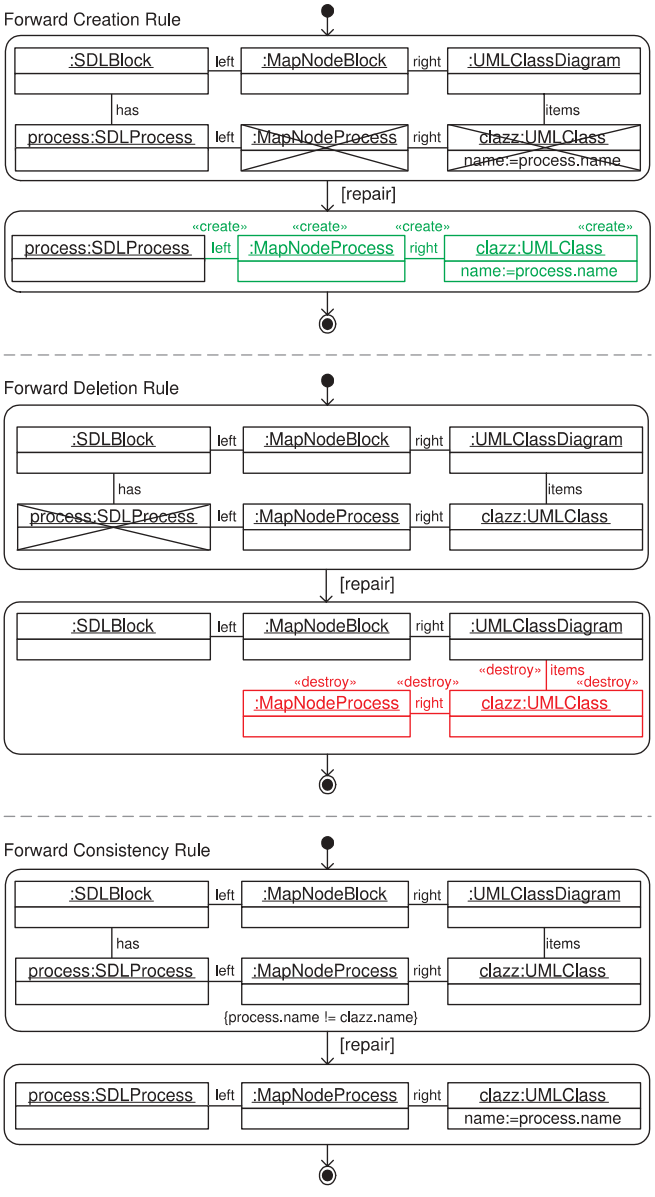


**Fig. 15.** Derived forward rules

The *Forward Creation Rule* is applied if a new SDL process was created and no corresponding class and integration objects are present. The operational semantic is similar to the already presented graph rewriting rules. Therefore, the graph rewriting system will search for all objects except the crossed-out nodes. The crossed-out nodes are so-called negative application conditions [12]. They are used to express that an object must not exist. Thus the graph rewriting system will check for the absence of the class and the integration object. If this condition holds, i.e., a matching pattern was found, an inconsistent situation was detected. To resolve the inconsistency, the missing class, the integration object, and the links are created. The creation of objects and links is specified by the stereotype ≪create≫. In the last step the name of the new class is set to the name of the process.

The *Forward Deletion Rule* is applied if an SDL process was deleted. This inconsistent situation is denoted by a crossed-out process object. If a matching object structure is found, both the integration object and the class object with its links are deleted. The deletion is denoted by the stereotype ≪destroy≫.

The *Forward Consistency Rule* is applied only if the process object and the class object are already connected through an integration object and the state of the process object has changed. In our concrete example, this means that the name of the process was modified. In this case the new name is propagated to the class object. This is specified by the assertion name := process.name within the UMLClass object.

In contrast to the forward rules, the rules that handle modifications of the right-hand model, i.e., the UML class diagram, are called right to left or *backward rules*. Since they are created in a way that is quite similar to the way in which the forward rules are created (just by exchanging the left and the right model elements), they are omitted here due to lack of space. For a detailed description see [29].

## 6 Integration within the Fujaba Tool Suite

As an application of our proposed solution to integrate tools at the meta-model level, we present the Fujaba Tool Suite, which is an open source UML CASE Tool project.

### *6.1 Fujaba Tool Suite*

The Fujaba project was started by the software engineering group at the University of Paderborn in fall 1997. It was designed as one monolithic application including several functionalities from different domains. In 2002 Fujaba was redesigned and became the Fujaba Tool

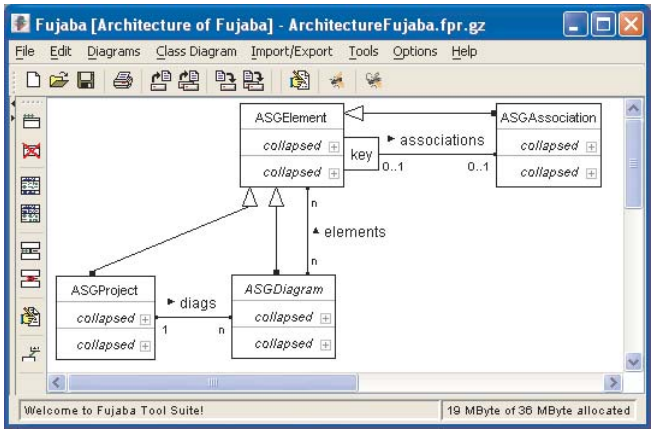


**Fig. 16.** FUJABA's meta-model framework

SUITE. It now provides a plug-in architecture that allows developers to add functionality easily while retaining full control over their contributions. It supports building menus, drawing diagrams, preserving consistency between models and views, etc. Therefore, the FUJABA TOOL SUITE is a platform for the integration of third-party tools.

### Meta-model framework

FUJABA's core meta-model provides a framework for abstract syntax graphs that is easily extensible by plug-ins using the Meta-Model Extension and Integration patterns. Figure 16 depicts the core's simplified meta-model framework.

ASGElement and ASGAssociation implement the Meta-Model Extension pattern. According to Fig. 4 the MetaModelElement is implemented by ASGElement and the MetaModelAssociation by ASGAssociation. Thus ASGElement is the superclass for all meta-model elements. All subtypes of ASGElement are connection points for other meta-models.

Through the elements association all meta-model elements can be assigned to a diagram that is an ASGElement itself. ASGDiagram serves as a base class for meta-model-specific diagrams such as a UML class diagram. Multiple diagrams build up a project. Hence ASGProject manages the diagrams through the diags association.

### Plug-ins

Plug-ins for the FUJABA TOOL SUITE using the meta-model framework can connect to meta-models of existing plug-ins through the Meta-Model Extension pattern. To accomplish a tool integration in practice, there is need not only for an integration at the meta-model level but also for extending the graphical user interface (GUI) and providing functionality to visualize the model.

The extension of the GUI is done by XML-based documents specifying the appearance of menu bars, popup menus, and toolbars. Menus and toolbars are filled with actions. An action may cause the creation, deletion, or modification of the model or open self-defined dialogs to interact with the user.

To help the plug-in developer visualize his model, FUJABA provides the so-called *Fujaba Swing Adapter* (FSA). FSA is an implementation of the Model View Controller mechanism and thus automatically propagates model changes to the visualization and vice versa.

Owing to FUJABA's history it provides additionally sophisticated code generation. This can easily be extended by adding additional code generators.

All XML documents and class files that specify the GUI extensions, the meta-model, the functionality, and the visual representation form the plug-in. They are shipped as a single jar file and can be downloaded with FUJABA from a list of servers and installed automatically.

### Consistency management environment

For the specific requirements of the outlined tool-integration framework, a flexible consistency management environment has been implemented (cf. [30]). The part of the consistency management architecture that is relevant for execution is summarized in Fig. 17.

Consistency rules can be organized in the form of catalogs and different categories to permit each plug-in to add its own rules in an organized manner. Such predefined catalogs can then be loaded on demand or at startup of a plug-in to configure the FUJABA TOOL SUITE for the specific needs of an individual tool-integration scenario.

For automated detection, checking, and repair, consistency rules are executed within so-called analysis engines. We have tool support to develop such engines for the two formalisms presented in Sect. 5, namely, simple graph rewrite rules and triple graph grammars. Additionally,

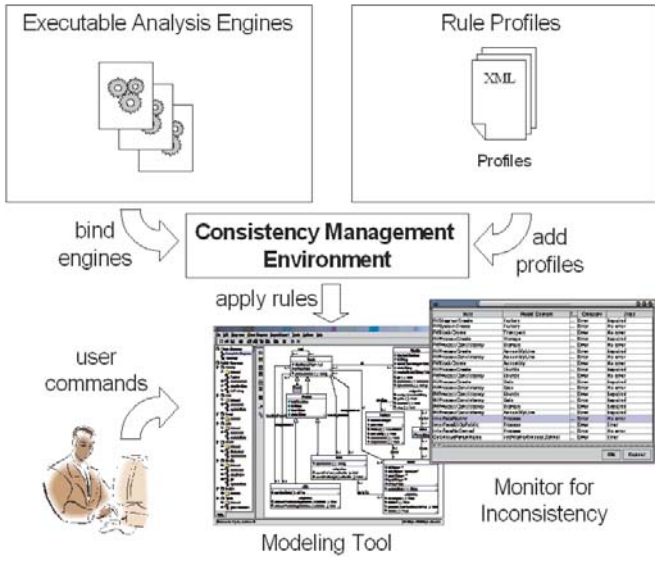


**Fig. 17.** Elements of consistency management plug-in

external tools such as model checkers can be integrated for checking and repair (cf. [13]) by simply realizing such engines.

Starting from a consistent model, inconsistencies can only be introduced by model changes. In our tool, models are changed by executing user commands from predefined pull-down or popup menus. In general, a user command is not an atomic operation and usually consists of more than one model modification. Thus inconsistencies during the execution of a user command are not unusual and are often temporary. Therefore, the automatic consistency checking within our plug-in is not executed until a user command is completed. The changes are collected, and after the command has been finished only relevant rules concerning the modified model elements are executed (Sect. 5).

### *6.2 Integration example*

In the following example we show how the Meta-Model Integration framework is used to integrate two standalone editors into our Fujaba Tool Suite. This integration will be used to show how the Meta-Model Integration pattern and consistency management are combined to achieve interoperability between both editors.

For the specification of controller software we are using SDL block diagrams and UML class diagrams (Sect. 3). For a tight integration both editors and their meta-models have to be integrated into the Fujaba Tool Suite using the Meta-Model Integration framework.

#### UML editor integration

A part of our UML class diagram meta-model and how it is embedded into Fujaba's meta-model framework is shown in Fig. 18. The class diagram is implemented by UMLClassDiagram extending ASGDiagram. The diagram elements representing classes, methods, and attributes are derived from UMLDiagramItem (which itself extends ASGElement). Thus other plug-ins can connect to the UML class diagram meta-model, as shown in Sect. 4.

#### SDL editor integration

The SDL meta-model is integrated in Fujaba in a similar manner. Figure 19 shows a part of its architecture.

As in the UML class diagram meta-model there exists a diagram class SDLDiagram extending ASGDiagram and multiple meta-model elements derived from ASGElement. Elements for structuring the SDL system are SDLSystem, SDLBlock, and SDLProcess. The superclass SDLElement was designed to model a system composed of blocks and processes. Blocks can be hierarchically nested. It is forbidden syntactically, for example, for a block to contain a process that is ensured through intramodel consistency rules. Connections between elements are modeled through the source and target associations to SDLConnection.

### *6.3 Consistency management*

Since both editors can be used in parallel, it is necessary to keep both specifications consistent. Thus changes in the UML model have to be reflected in the SDL model and vice versa.

For this purpose we are using the triple graph grammars presented in Sect. 5. The rule specification is supported by a special editor where required rules can be specified in an easy way. The needed bidirectional associations between the integration model and the meta-models of the tools are realized using the Meta-Model

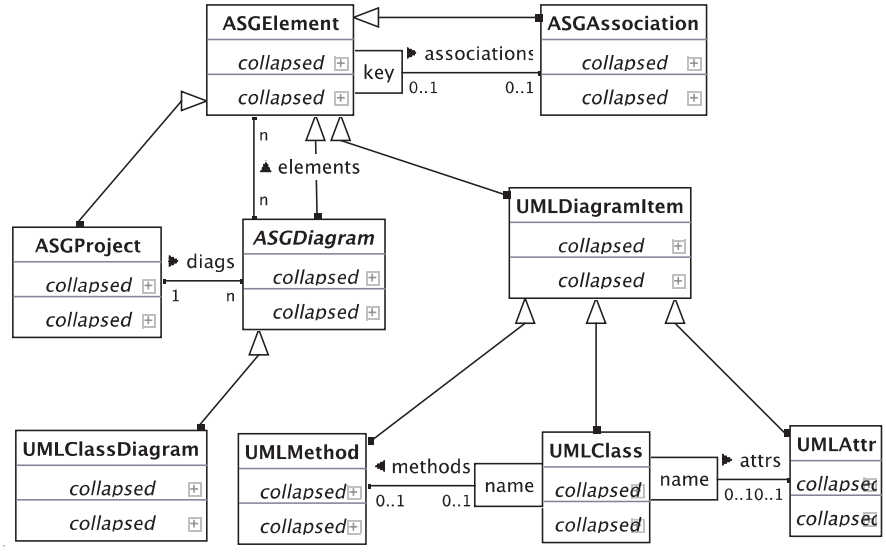


**Fig. 18.** Meta-model of the UML class diagram plug-in

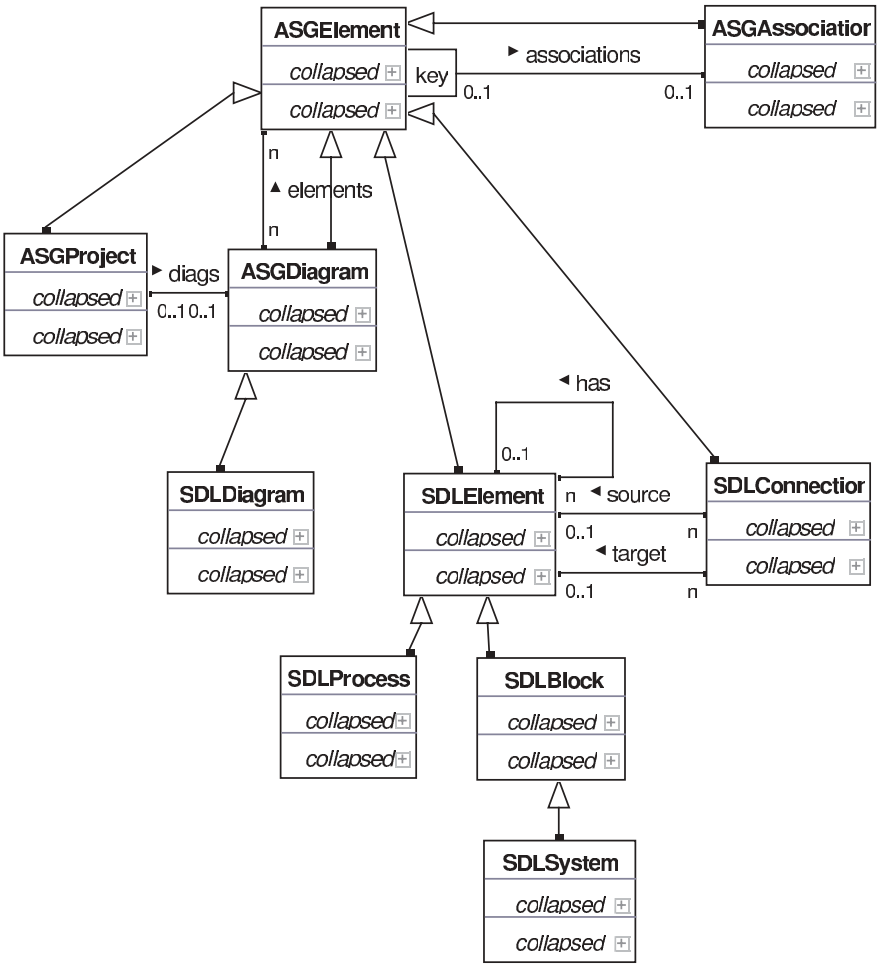


**Fig. 19.** Meta-model of the SDL plug-in

Integration pattern from Sect. 3. For a simpler usage of the editor, the extra nodes used within the integration pattern are hidden from the user.

The triple graph grammar editor and the rule for the process to class mapping are shown in Fig. 20. This rule can then be translated automatically into the simple graph grammar rules presented in Sect. 5. From these graph grammar rules executable analysis engines are generated.

The consistency management binds the rule catalog and its analysis engines at runtime to the Fujaba Tool Suite. These rules are then applied to the currently developed models.

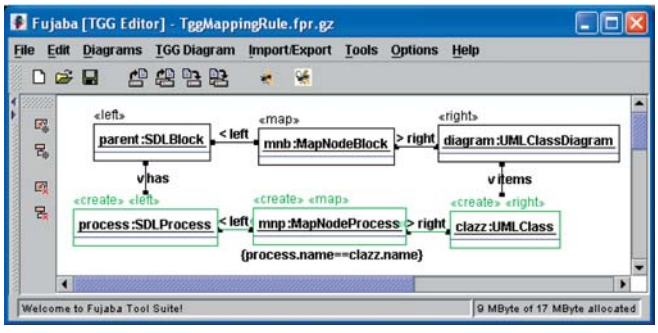


**Fig. 20.** TGG Rule editor

The consistency management can also be configured and adapted at runtime. For example, consistency rules and categories can be activated or deactivated on demand by the user. If a category is deactivated, its entries are not considered and thus no rule within the category is checked. This way, the engineer can adjust the consistency checking to his current needs.

The user can specify whether or not he will be informed immediately about any detected inconsistency. If this option is selected, the system informs the user as soon as an inconsistency is detected by displaying an information dialog. In most cases, especially for minor inconsistencies, the option will be deactivated so as not to hinder the user.

An inconsistency can be resolved either automatically or manually. This behavior can be controlled by the repair option. If no repair action is specified or the automatic repair is disabled, the developer is informed about the unresolved inconsistency and must resolve it manually. This way, the automatic resolution of inconsistencies may be circumvented by the user if required.

## 7 Conclusion

Tool integration is a hot topic among tool vendors and users. The integration of graphical user interface artifacts and the integration of functionality via a special application programming interface (API) is supported by most integration platforms. Unfortunately, a prominent problem in tool integration originates from data stored by the different tools. The problem of dealing with overlapping data and ensuring the consistency of data while maintaining the flexibility of different tools remains unsolved in all proposed platforms.

In this paper we describe the Fujaba approach to tool integration. The Fujaba approach tackles the different aspects of tool integration. While also supporting the mentioned aspects of integration at the graphical user interface level and providing the integration of functionality, the presented approach focuses in particular on data integration.

The proposed Meta-Model Extension and Meta-Model Integration patterns enable the integration of data in different scenarios on the meta-model level. Both patterns maintain flexibility by ensuring compile-time independence. The Meta-Model Extension pattern is used for the extension of one meta-model, whereas the Meta-Model Integration pattern enables the integration of two different meta-models by an integration meta-model.

Based on the data integration provided by the patterns, a consistency mechanism is presented. This consistency mechanism uses triple graph grammars for a graphical, though formal, specification of structural consistency between instances of the integrated meta-model elements. This formal specification is used for managing consistency during runtime. Automatic repair actions may be included in the consistency specification.

The presented approaches to data integration and additional approaches for graphical user interface and functionality integration have been widely employed within the Fujaba Tool Suite and all related plug-ins with great success.

*Acknowledgements.* We thank all students, Ph.D. students, and all those who have helped to build the current Fujaba Tool Suite, which is available at `http://www.fujaba.de`.

## References

1. Arnold K, O'Sullivan B, Scheifler RW, Waldo J, Wollrath A, (1999) The Jini specification. The Jini Technology Series, June 1999
2. Balzer R (1991) Tolerating inconsistency. In: Proc. 13th international conference on software engineering, Austin, TX. IEEE Press, New York, pp 158–165
3. Bottoni P, Koch M, Parisi-Pressice F, Taenzer G (2000) Consistency checking and visualization of OCL constraints. In: UML 2000. Lecture notes in computer science, vol 1936. Springer, Berlin Heidelberg New York
4. Egyed A (2000) Automatically validating model consistency during refinement. Technical report, Center for Software Engineering, University of Southern California, Los Angeles, October 2000
5. Ehrig H, Tsiolakis A (2000) Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In: Ehrig H, Taentzer G (eds) Proc. ETAPS 2000 workshop on graph transformation systems, Berlin, Germany
6. Engels G, Küster J, Groenewegen L, Heckel R (2001) A methodology for specifying and analyzing consistency of object-oriented behavioral models. In: Gruhn V (ed) Proc. 8th European software engineering conference (ESEC), September 2001, pp 186–195
7. Finkelstein A (2000) A foolish consistency: technical challenges in consistency management. In: Ibrahim MT, Küng J, Revell N (eds) Proc. 11th international conference on database and expert systems applications (DEXA'00), London, UK, September 2000. Lecture notes in computer science, vol 1873. Springer, Berlin Heidelberg New York, pp 1–5
8. Fischer T, Niere J, Torunski L, Zündorf A (1998) Story diagrams: a new graph rewrite language based on the unified modeling language. In: Engels G, Rozenberg G (eds) Proc. 6th international workshop on theory and application of graph transformation (TAGT), Paderborn, Germany, November 1998. Lecture notes in computer science, vol 1764. Springer, Berlin Heidelberg New York, pp 296–309
9. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object oriented software. Addison-Wesley, Reading, MA
10. Ghezzi C, Nuseibeh B (1998) Special issue on managing inconsistency in software development (1). IEEE Trans Softw Eng 24(11):906–1001
11. Ghezzi C, Nuseibeh B (1999) Special issue on managing inconsistency in software development (2). IEEE Trans Softw Eng 25(11):782–869
12. Habel A, Heckel R, Taenzer G (1996) Graph grammars with negative application conditions. Fund Inf 26(3,4):287–313
13. Hirsch M, Giese H (2003) Towards the incremental model checking of complex realtime UML models. In: Proc. Fujaba Days 2003, Kassel, Germany, October
14. Jahnke J (1999) Management of uncertainty and inconsistency in database reengineering processes. PhD thesis, University of Paderborn, Paderborn, Germany, September
15. Jahnke J, Schäfer W, Wadsack J, Zündorf A (2002) Supporting iterations in exploratory database reengineering processes. J Sci Comput Programm 45(2-3):99–136
16. Java Community Process (2002) JSR 040: Java Metadata Interface (JMI) Specification, June 2002
17. Jin D, Cordy J, Dean T (2002) Where's the schema? A taxonomy of patterns for software exchange. In: Proc. 10th international workshop on program comprehension (IWPC), Paris, June 2002
18. Kazman R, Woods S, Carrière SJ (1998) Requirements for integrating software architecture and reengineering models: CORUM II. In: Proc. working conference on reverse engineering (WCRE'98), Honolulu, HI, October, pp 154–163
19. Köhler H, Nickel U, Niere J, Zündorf A (2000) Integrating UML diagrams for production control systems. In: Proc. 22nd international conference on software engineering (ICSE), Limerick, Ireland. ACM Press, New York, pp 241–251
20. Lefering M (1994) Software document integration using graph grammar specifications. In: Proc. 6th international conference on computing and information. J Comput Inf 1(1):1222–1243
21. Nagl M (ed) (1996) The IPSEN approach. Lecture notes in computer science, vol 1170. Springer, Berlin Heidelberg New York
22. Nickel U, Schäfer W, Zündorf A (2003) Integrative specification of distributed production control systems for flexible automated manufacturing. In: Nagl M, Westfechtel B (eds) DFG Workshop: Modelle, Werkzeuge und Infrastrukturen zur Unterstützung von Entwicklungsprozessen, pp 179–195
23. Nuseibeh B, Easterbrook S, Russo A (2000) Leveraging inconsistency in software development. IEEE Comput 33(4):24–29
24. Object International () Together Control Center, the Together case tool. `http://www.togethersoft.com` [last accessed October 2004]
25. OMG (2002) Unified Modeling Language Specification version 1.5. Needham, MA
26. Rational Rose. the Rational Rose case tool. `http://www.rational.com` [last accessed October 2004]

27. Rozenberg G (ed) (1999) Handbook of graph grammars and computing by graph transformation, vol 1. World Scientific, Singapore
28. Schürr A (1994) Specification of graph translators with triple graph grammars. In: Proc. 20th international workshop on graph-theoretic concepts in computer science. Herrsching, Germany. Springer, Berlin Heidelberg New York
29. Wagner R (2001) Realisierung eines diagrammübergreifenden Konsistenzmanagement-Systems für UML-Spezifikationen. Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, November
30. Wagner R, Giese H, Nickel U (2003) A plug-in for flexible and incremental consistency management. In: Proc. international conference on the Unified Modeling Language 2003 (Workshop 7: Consistency problems in UML-based software development), San Francisco, October
31. Waldo J (1999) The Jini architecture for network-centric computing. Commun ACM 42(7):76–82
32. Woods S, O'Brian L, Lin T, Gallagher K, Quilici A (1998) An architecture for interoperable program understanding tools. In: Proc. 6th international workshop on program comprehension (IWPC), Ischia, Italy, July 1998, pp 54–63