3.2 Incremental and Interactive Integrator Tools for Design Product Consistency

S. Becker, M. Nagl, and B. Westfechtel

Abstract. Design processes in chemical engineering are inherently complex. Various aspects of the plant to be designed are modeled in different logical documents using heterogeneous tools. There are a lot of fine-grained dependencies between the contents of these documents. Thus, if one document is changed, these changes have to be propagated to all dependent documents in order to restore mutual consistency.

In current development processes, these dependencies and the resulting consistency relationships have to be handled manually by the engineers without appropriate tool support in most cases. Consequently, there is a need for incremental integrator tools which assist developers in consistency maintenance. We realized a framework for building such tools. The tools are based on models of the related documents and their mutual relationships. Realization of integrators and their integration with existing tools is carried out using graph techniques.

3.2.1 Integrator Tools for Chemical Engineering

Introduction

Development processes in different engineering disciplines such as mechanical, chemical, or software engineering are highly complex. The product to be developed is described from multiple perspectives. The results of development activities are stored in *documents* such as e.g. requirements definitions, software architectures, or module implementations in software engineering or various kinds of flow diagrams and simulation models in chemical engineering (cf. Sect. 1.1). These documents are connected by mutual dependencies and have to be kept consistent with each other. Thus, if one document is changed, these changes have to be propagated to dependent documents in order to restore mutual consistency.

Tool support for maintaining inter-document consistency is urgently needed. However, conventional approaches suffer from severe limitations. For example, batch converters are frequently used to transform one design representation into another. Unfortunately, such a transformation cannot proceed automatically, if human design decisions are required. Moreover, batch converters cannot be applied to propagate changes incrementally. Current tool support in chemical engineering is mainly characterized by numerous software tools for *specific purposes* or *isolated parts* of the design process. However, a sustainable improvement of the design process can only be achieved by the integration of single application tools into a comprehensive design environment [548]. During the last years, commercial environments like Aspen Zyqad [517] or Intergraph's SmartPlant [605] have been developed. They are mainly restricted to the tools of the corresponding vendor. The adaptation

M. Nagl and W. Marquardt (Eds.): Collaborative and Distributed Chemical Engineering, LNCS 4970, pp. 224–267, 2008.

of the tools to specific work processes of developers within a company or the integration of arbitrary tools, especially from other vendors, are unsolved issues.

In this paper, we present *incremental integrator tools* (integrators) which are designed to support concurrent/simultaneous engineering and can be tailored to integrate any specific interdependent documents from any application domain. The only restriction is that documents have to be structured such that a graph view on their contents can be provided.

The key concept of our tools is to store fine-grained relationships between interdependent documents in an additional *integration document* which is placed in between the related documents. This integration document is composed of *links* for navigating between fine-grained objects stored in the respective documents. Furthermore, these links are used to determine the impact of changes, and they are updated in the course of change propagation. Changes are propagated and links are established using an extension of the triple graph grammar formalism originally introduced by Schürr [413].

Integrator tools are driven by *rules* defining which objects may be related to each other. Each rule relates a pattern of source objects to a pattern of target objects via a link. Rules may be applied automatically or manually. They are collected in a rule base which represents domain-specific knowledge. Since this knowledge evolves, the rule base may be extended on the fly. The definition of rules is based on domain knowledge. Rules for our integrators for chemical engineering design processes are defined using the model framework CLiP [20] (cf. Subsect. 2.2.3).

Integrator tools for specific documents are built based on a universal *integrator framework* and by specifying a corresponding rule base. Additionally, some tool-specific extensions, like wrappers for the tools to be integrated, have to be implemented.

There has been a tight *cooperation* of this subproject with the CLiP project at the department of process systems engineering (LPT) [15]. All chemical engineering examples used throughout this paper have been elaborated in cooperation with the LPT and our industrial partner innotec [745].

Motivating Example

We will use the sample scenario in Fig. 3.21 to illustrate how integrator tools assist the design team members. The scenario deals with the integration of *process flow diagrams* (PFD) and *simulation models*. A PFD describes the chemical process to be designed, while a simulation model serves as input to a tool for performing steady-state or dynamic simulations. Different tools may be used for creating flowsheets and simulation models, respectively. In the following, we assume that the flowsheet is maintained by Comos PT [745] and simulations are performed in Aspen Plus [516], both of which are commercial tools used in chemical engineering design.

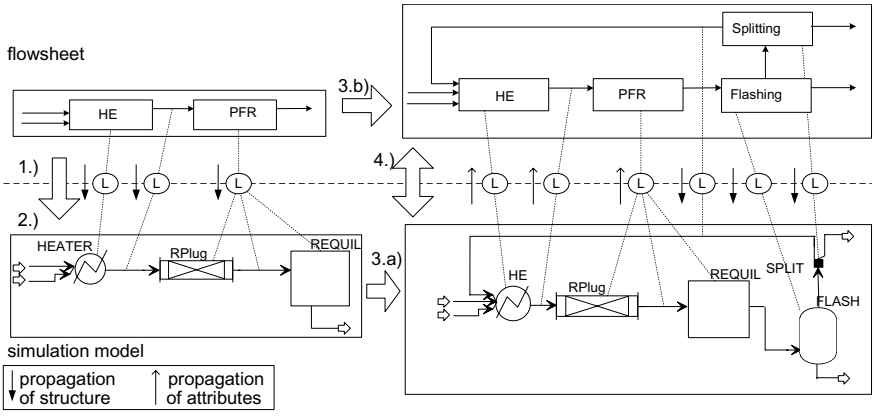

Fig. 3.21. Sample integration scenario: integration of PFD and simulation model

PFDs act as central documents for describing chemical processes. They are refined iteratively so that they eventually describe the chemical plant to be built. Simulations are performed in order to evaluate design alternatives. Simulation results are fed back to the PFD designer, who annotates the flowsheet with flow rates, temperatures, pressures, etc. Thus, *information* is *propagated back* and *forth* between flowsheets and simulation models.

Unfortunately, the *relationships* between both results are *not* always *straightforward*. To use a simulator such as Aspen Plus, the simulation model has to be composed from pre-defined blocks. Therefore, the composition of the simulation model is specific to the respective simulator and may deviate structurally from the PFD.

The chemical process taken as example produces ethanol from ethen and water. The PFD and simulation models are shown above and below the dashed line, respectively. Two *subsequent versions* of *both models* are depicted side by side. The integration document for connecting both models contains links which are drawn on the dashed line[26]. The figure illustrates a design process consisting of four steps:

1. An initial *PFD is created* in Comos PT. This PFD is still incomplete, i.e., it describes only a part of the chemical process (heating of substances and reaction in a plug flow reactor, PFR).
2. The integrator tool is used to *derive a simulation model* for Aspen Plus from the initial PFD. Here, the user has to perform two decisions. While the heating step can be mapped structurally 1:1 into the simulation model, the user has to select the most appropriate block for the simulation to be performed. Second, there are multiple alternatives to map the PFR. Since the most straightforward 1:1 mapping is not considered to be sufficient, the

[26] This is a simplified notation. Some details of the document and integration model introduced later are omitted.

user decides to map the PFR into a cascade of two blocks. These decisions are made by selecting among different possibilities of rule applications which the tool presents to the user.

3. a) The simulation is performed in Aspen Plus, resulting in a *simulation model* which is augmented with simulation *results*.
   b) In parallel, the *PFD* is *extended* with the chemical process steps that have not been specified so far (flashing and splitting).
4. Finally, the integrator tool is used to *synchronize* the parallel *work* performed in the previous step. This involves information flow in both directions. First, the simulation results are propagated from the simulation model back to the PFD. Second, the extensions are propagated from the PFD to the simulation model. After these propagations have been performed, mutual consistency is re-established.

An integrator tool prototype has been realized to carry the design process out in this example. This was part of an industrial cooperation with innotec [745], a German company which developed Comos PT.

Requirements

From the motivating example presented so far, we derive the following requirements:

Functionality An integrator tool must manage *links between objects* of interdependent documents. In general, links may be m:n relationships, i.e., a link connects m source objects with n target objects. They may be used for multiple purposes: *browsing*, *correspondence analysis*, and *transformation*.

Mode of operation An integrator tool must operate incrementally rather than batch-wise. It is used to *propagate changes* between interdependent documents. This is done in such a way that only *actually affected parts* are *modified*. As a consequence, manual work does not get lost (in the above example the elaboration of the simulation model), as it happens in the case of batch converters.

Direction In general, an integrator tool may have to work in *both directions*. That is, if a source document is changed, the changes are propagated into some target document and vice versa.

Integration rules An integrator tool is driven by *rules* defining which *object patterns* may be related to each other. There must be support for defining and applying these rules. Rules may be interpreted or hardwired into software.

Mode of interaction While an integrator tool may operate automatically in simple scenarios, it is very likely that user *interaction* is *required*. On the one hand, user interaction can be needed to resolve non-deterministic situations when integration rules are conflicting. On the other hand, there can be situations where no appropriate rule exists and parts of the integration have to be corrected or performed manually.

Time of activation In single user applications, it may be desirable to propagate changes eagerly. This way, the user is informed promptly about the consequences of the changes performed in the respective documents. In multi user scenarios, however, *deferred propagation* is usually required. In this way, each user keeps control of the export and import of changes from and to his local workspace.

Traceability An integrator tool must *record a trace* of the rules which have been applied. This way, the user may reconstruct later on, which decisions have been performed during the integration process.

Adaptability An integrator tool must be adaptable to a *specific application domain*. Adaptability is achieved by defining suitable integration rules and controlling their application (e.g., through priorities). In some cases, it must be possible to modify the rule base on the fly.

A-posteriori integration An integrator tool should work with *heterogeneous tools* supplied by different vendors. To this end, it has to access these tools and their data. This is done by corresponding wrappers which provide abstract and unified interfaces.

Not every integrator tool has to fulfill all of these requirements. E.g., there are some situations where incrementality is not needed. In other situations, the rule base is unambiguous such that there will never be user interaction. In such cases, it has to be decided whether a full-scale integrator tool based on the integrator framework is used anyway, or some other approach is more suitable. In IMPROVE, we implemented both types of integrator tools for different parts of our overall scenario. For instance, in addition to the integrator tool described in the motivating example, we created a tool that generates the input file for heterogeneous process simulation with CHEOPS ([409], see Subsect. 3.2.6). This tool only uses small parts of the integrator framework and most of its behavior is hand-coded instead of being directly controlled by rules. Other tools have been realized using completely different approaches like XML and XSLT [567, 602]. Even for the realization of these simpler tools, the experiences gained with the full-scale integrator tools were helpful.

Organization of This Paper

The rest of this paper is structured as follows: In the next Subsect. 3.2.2, we give a short overview of our integrator framework. Our modeling formalism for integration is explained in Subsect. 3.2.3. Subsection 3.2.4 addresses the integration algorithm and its relation to the triple graph grammar approach. We support two approaches for the implementation of integrators, which are introduced and compared in Subsect. 3.2.5. In our project, furthermore, some integrators have been realized following a modified or an entirely different approach. They are sketched in Subsect. 3.2.6. Subsection 3.2.7 compares our approach to other integration R&D work. In Subsect. 3.2.8, we give a summary and an outlook on open problems.

3.2.2 Integrator Framework

Architecture Overview

In each application domain, e.g. in chemical engineering, there are a lot of applications for integrator tools. As a consequence, the *realization* of a specific integrator tool has to require as *little effort* as possible.

We are addressing this by two means: First, our approach allows to define *rules* for integrator tools based on already-existing domain models (cf. Sect. 6.3) and to *derive* an *implementation* for such tools (process reuse within the integrator development process). Second, we provide a *framework* for integrator tools that offers most parts of the integrator functionality in predefined and *general components* (product reuse) [27, 251]. To create a specific integrator tool, only some additional components have to be implemented and integration rules have to be defined.

Figure 3.22 provides an overview of the *system architecture* for integrator tools. It shows an integrator tool between Comos PT (source, lower left corner) and Aspen Plus (target, lower right corner) as example. Note that the terms "source" and "target" denote distinct ends of the integration relationship between the documents, but do not necessarily imply a unique direction of transformation. For each pair of related Aspen and Comos documents, their fine-grained relationships are stored as links in an integration document. The structure of the integration document is the same for all integrator tools realized using the framework and will be discussed below.

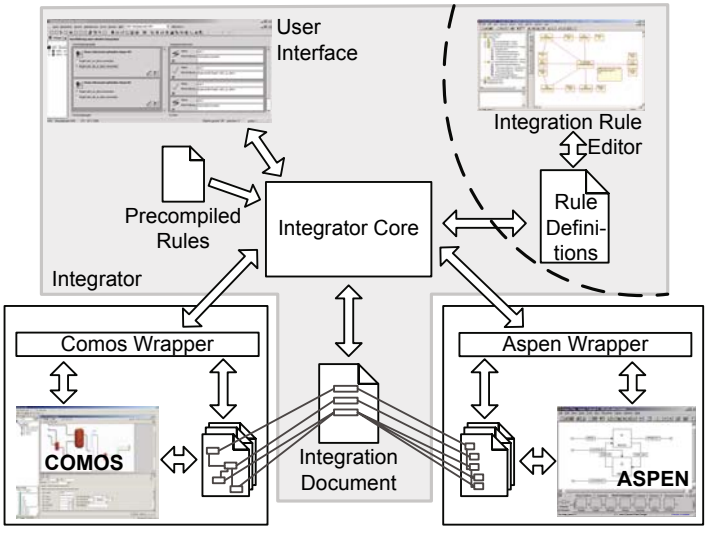

Fig. 3.22. System architecture of an integrator

The integration is performed by the *integrator core*. It propagates changes between source and target documents and vice-versa and modifies the links in the integration document. The integrator core includes an implementation of the integration algorithm which will be explained in detail in Subsect. 3.2.4. It is a universal component that is used by all framework-based integrator tools.

The core does not access source and target tools directly but uses *tool wrappers*. These wrappers provide a standardized graph view on the tools' documents to keep the integrator code independent of the tools' specific interfaces. Additionally, they provide functions for launching tools and locating specific documents. For each tool, a new wrapper has to be implemented. To minimize the required programming efforts, the model-based wrapper specification approach described in [136] and in Sect. 5.7 of this book can be used.

During integration, the integrator core is controlled by *integration rules*, which can be provided following *different approaches*:

First, they can be defined using the integration rule editor (upper right corner of Fig. 3.22), be exported as rule definition files, and then be executed by a *rule interpreter* being part of the core. The formalism for defining integration rules will be explained in Subsect. 3.2.3. Depending on whether the integrator supports the definition of integration rules on the fly, the rule editor is considered either a part of the framework or a part of the development environment for integrators.

Second, they can be *implemented manually* in a programming language, compiled, and linked to the integrator tool in order to be executed. This can lead to a better performance during execution and allows to define rules whose functionality goes beyond the rule definition formalism and the predefined execution algorithm.

Third, as a *combination*, rules can be specified using the rule editor, then translated automatically into source code, and then compiled and linked to the integrator tool.

The integrator *user interface* (upper left corner of Fig. 3.22) is used to control the integrator interactively. Here, the user has the possibility to choose between different rule applications or to manipulate links manually. Although there is a generic user interface implementation, in most cases an application-specific GUI should be provided to facilitate the integration process for the user.

Integration Document

An integration document contains a set of *links* which represent the relationships mentioned above. Each link relates a set of syntactic elements (*increments*) belonging to one document with a corresponding set belonging to another document. A link can be further structured by adding *sublinks* to a link. A sublink relates subsets of the increments referenced by its parent link and is created during the same rule execution as its parent.

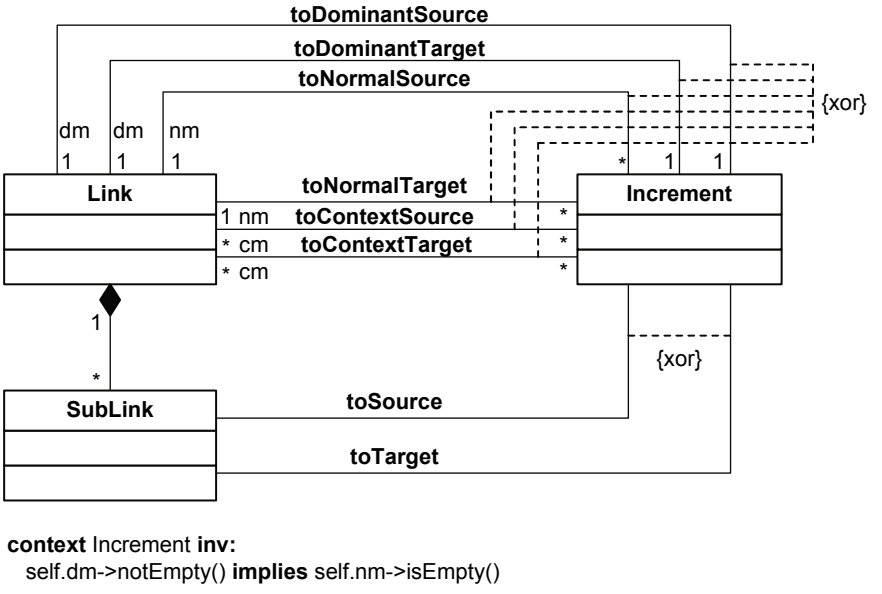

Fig. 3.23. Link model

Figure 3.23 shows the *structure of links* in a UML class diagram [560]. Most constraints needed for a detailed definition are omitted, only examples are shown. An increment can have different roles w.r.t. a referencing link: Increments can be *owned* by a link or be referenced as *context* increments. While an increment can belong to at most one link as owned increment, it can be referenced by an arbitrary number of links as context increments. Owned increments can be created during rule execution, whereas only existing increments can be referenced by new links as context increments.

Context increments are needed when the execution of a rule depends on increments belonging to an already existing link that was created by the application of another rule. Context is used for instance to embed newly created edges between already transformed patterns. Owned increments can be further divided into *dominant* and *normal* increments. Dominant increments play a special role in the execution of integration rules (see Subsect. 3.2.4). Each link can have at most one dominant increment in each document. A link can relate an arbitrary number of normal increments.

There is additional *information* stored *with a link*, e.g. its state and information about possible rule applications. This information is needed by the integration algorithm but not for the definition of integration rules.

3.2.3 Definition of Integration Rules

Overview: Different Levels for Modeling Rules

To create a specific integrator, a set of integration rules specifying its behavior is needed. Therefore, we provide a modeling formalism for such rule sets. The *explicit modeling* of rules has several *advantages* over implicitly coding them within an integrator: First of all, it is much easier to understand rules and their interdependencies if they are available as a human readable visual model. Additionally, our modeling approach is multi-layered and allows consistency checking between the layers. This cannot guarantee the complete correctness of rule sets but at least prevents some major mistakes and thereby ensures the executability of rules.

Another advantage is that the *source code* of integrators is *independent* of specific rules or – if rules are linked to the integrator (see above) – dependencies are limited to certain sections of the code. This *facilitates* the *adaptation* of integrators to new applications or changed rules. If integration rule sets are interpreted using the rule interpreter of the integrator framework, even learning new *rules on the fly* is possible.

In most application domains, *domain models* already exist or are at least under development. Consequently, the information has to be used when defining integration rules. For instance, in another project of IMPROVE the product data model framework CLiP [20] was defined (see Sect. 2.2). Such domain models normally are not detailed enough to allow for the derivation of integration rules or even integrator tools. Nevertheless, they can serve as a starting point for integration rule definition [15]. Together with some company-specific refinements, they can be used to identify documents that have to be integrated and to get information about the internal structure of the documents as well as about some of the inter-document relationships. Of course, the models have to be further refined to be able to get executable integration rules. The process of refining domain models to detailed definitions of the behavior of integrator tools is described in detail in Sect. 6.3. In this section, the focus is on defining integration rule sets without a strong relation to domain models.

For the definition of integration rules, we follow a *multi-layered approach* as postulated by OMG's meta object facility (MOF) [874], based on the Unified Modeling Language (UML) [560]. Figure 3.24 provides an overview of the different modeling levels and their contents for the running example. Using MOF as meta-meta model, on the meta level the existing UML meta model is extended. Additional elements are added that form a language to define models of the documents to be integrated and to express all aspects concerning the documents' integration.

The *extension* on the *meta level* comprises *two parts*: First, *graph-* and *integration-related* definitions are added. These are used by all integration rule sets. Second, *domain-specific* extensions can be made. They can facilitate the definition of integration models when being combined with domain-specific

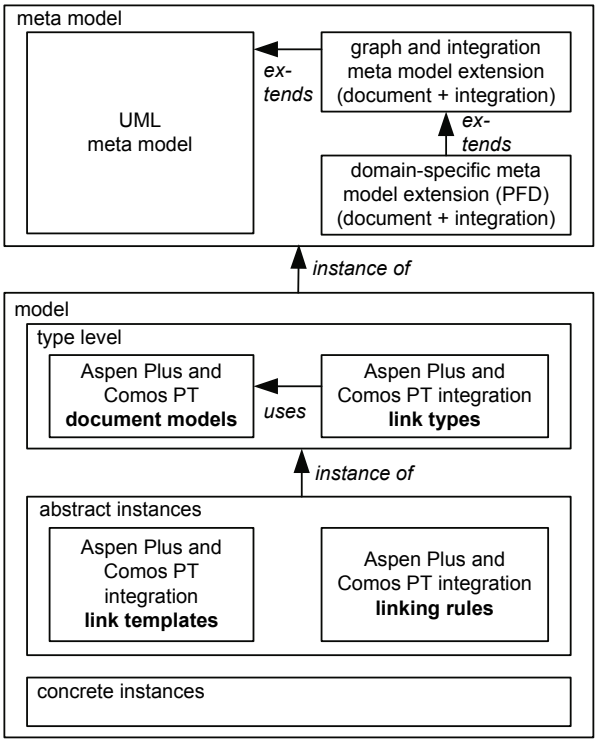

Fig. 3.24. Levels of modeling

visualization. To express the relation between meta model and model, we use the approach sketched in [391], where UML stereotypes are used on the *model level* to express the instance-of relationship between meta model elements and model elements. A more detailed description of the meta level can be found in [26].

On the *model level*, we distinguish between a type (or class) level and an instance level, like standard UML does. On the *type level*, *document models* for specific types of documents are defined. They are expressed as class hierarchies describing the documents' underlying type systems. In our example, documents containing simulation models for Aspen Plus and flowsheets for Comos PT are defined. To be able to perform an integration of these documents, *link types* that relate classes contained in the documents' class hierarchies are defined. All occurrences of links in further definitions on lower levels are instances of these link types and are thereby constrained by these types.

The *instance level* is divided into an abstract and a concrete level. On the *abstract instance level*, *link templates* and *linking rules* are specified using collaboration diagrams. Link templates are instances of link types relating

a pattern (which is a set of increments and their interrelations) that may exist in one document to a corresponding pattern in another document. A link template only defines a possible relation between documents. It is not imposed that the relation always exists for a given set of concrete documents.

Link *templates* can be *annotated* to define executable linking rules. The annotations provide information about which objects in the patterns are to be matched against existing objects in concrete documents and which objects have to be created, comparable to graph transformations. Linking *rules* are *executed* by integrators and are thus also called integration rules. Rule execution is described in detail in Subsect. 3.2.4.

While on the abstract instance level only patterns are defined that may appear in source, target, and integration document, on the *concrete instance level*, concrete existing documents and integration documents can be modeled. The resulting models are snapshots of these real documents, which can be used as examples, e.g., to document rule sets. As this is not vital for defining integration rules, models on the concrete instance level are not further described here.

In the following, selected aspects of the integration rule model are described in more detail. For further information, we refer to [26, 39, 40]. The modeling approach presented here is based on work dealing with a purely graph-oriented way of specifying integration rules and corresponding tools [131–134].

Type Level Modeling

Before integration rules for a given pair of document types can be defined, *document models* describing the documents' contents have to be created. In our current modeling approach, this is done by providing class hierarchies defining *types* of *entities* relevant for the integration process (increment types) and the possible *interrelations* of increments being contained in a document. To facilitate the definition of integration rules, it is planned to use more advanced document models that address further structural details (see Sect. 6.3). Here, simple UML-like class diagrams are used to express the type hierarchies.

To illustrate our modeling approach, we use excerpts of the underlying rule base of the motivating scenario presented in Subsect. 3.2.1. Figure 3.25 shows a part of the *Aspen Plus* type hierarchy. The figure is simplified, as it does not show stereotypes, cardinalities, and association names. It only shows an *excerpt* of the *simulation document model*. The type hierarchy does not reflect the whole Aspen Plus data model as offered by Aspen's COM interface. Instead, it is the model offered by our Aspen tool wrapper which supplies only information relevant for our integration between simulation models and flowsheets.

On the lowest layer on the left side of the type hierarchy we find increment *types* for some of the simulation *blocks* and *streams* that are predefined

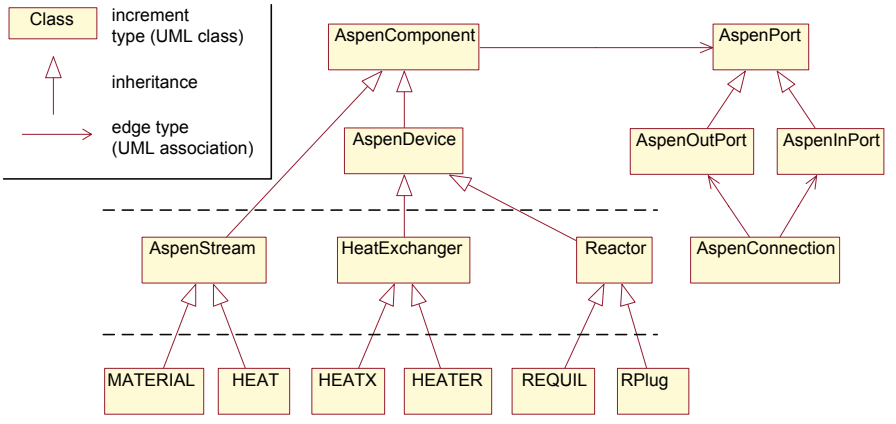

Fig. 3.25. Aspen Plus document model (UML class diagram)

in Aspen Plus (e.g. RPlug modeling the behavior of plug flow reactors and MATERIAL, the type of stream transporting substances between blocks).

One layer higher, the blocks are further classified by grouping them into *categories* that correspond to the tabs holding the blocks in the Aspen user interface. All blocks and streams inherit from AspenComponent (top layer). Each *component* can have an arbitrary number of ports (AspenPort) which can be further refined regarding their orientation (AspenInPort and AspenOutPort). AspenConnection is used to express that two ports of opposite orientation are connected by referencing them via associations.

In Fig. 3.21, blocks are represented as *rectangles*, streams are shown as *arrows* inside source and target document. Connections and ports are not represented explicitly (rather, they may be derived from the layout), but they are part of the internal data model.

The *document model* for *Comos PT* is not presented here, since it is structured quite similarly. This similarity is due to two reasons: First, both types of documents, simulation models and PFDs, describe the structure of chemical plants by modeling their main components and their interconnections. Second, technical differences between the internal models of both applications are eliminated by tool wrappers. Nevertheless, the remaining mapping between simulation models and PFDs is not straightforward, as we have already discussed above.

As a first step for defining integration rules, *link types* are modeled. They are used for two different purposes: First, each link that is contained in an integration document or occurs in an integration rule on the instance level has to be an *instance* of a link *type*. As a result, it is possible to *check* the *consistency* of integration rules against a set of link types. This cannot ensure the semantical correctness of integration rules but facilitates the definition process by eliminating some mistakes.

Second, it is very likely that for a new integrator a basic set of integration *rules* is defined *top-down using domain knowledge* [15]. For instance, in our running example it is common domain knowledge that devices in a PFD are simulated by predefined blocks in Aspen Plus. More concrete statements can be made as well: A reactor in the PFD could be simulated by some reactor block in Aspen Plus. Each of these statements can be translated easily in a link type definition. Thereby, our modeling formalism provides a means of formalization and communication of domain knowledge concerning the relationships between documents. The resulting link types can be further refined and, finally, on the abstract instance level, integration rules can be defined accordingly.

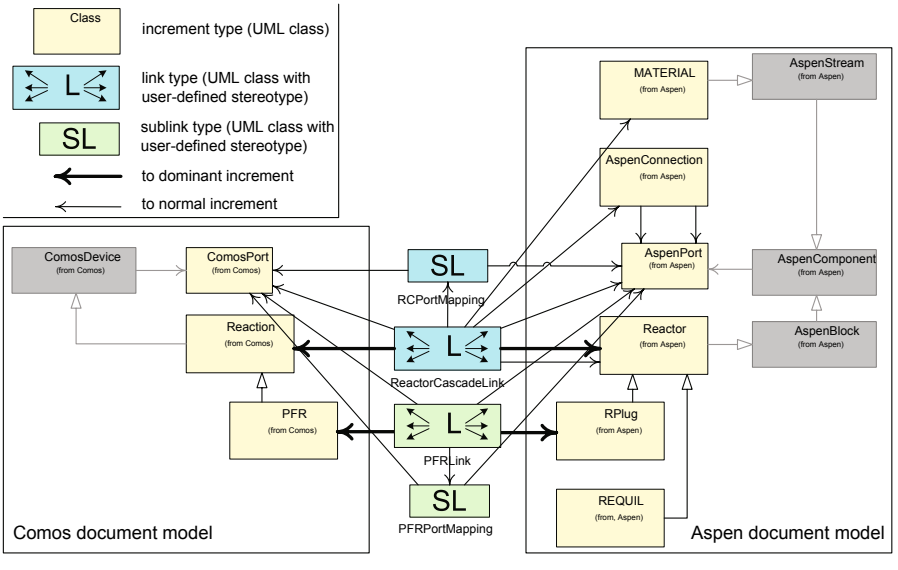

Fig. 3.26. Reactor link types (UML class diagram)

Figure 3.26 shows an example class diagram defining two *link types* concerning the relation between *reactors* in *PFDs* and simulator *blocks* in *Aspen Plus*. The left side of the figure contains an excerpt of the Comos document model, the right side one of the Aspen document model. In between, link types and sublink types are depicted. While the documents' increment types are shown as plain UML classes, link and sublink types are presented using graphical stereotypes. This can be applied to increment types as well if domain-specific elements are defined in the meta model together with corresponding graphical stereotypes. For instance, all reactor increment types could be displayed with a reactor symbol. The usage of graphical stereotypes facilitates the readability of models for domain experts having little UML knowledge.

In the figure, a link type called PFRLink is defined. This *example link type* expresses that exactly one instance[27] of the specific reactor type PFR (plug flow reactor) can be simulated by exactly one instance of the specific reactor block RPlug in the simulation model. The reactors in both documents can have an arbitrary number of ports which are mapped by the link as well. To assign corresponding ports, the link may have sublinks each mapping pairs of ports[28]. Both reactors are referenced as dominant increments. This link type is rather specific, as it forms relatively tight constraints for all of its instances.

In general, it is not always possible to simulate a reactor in the PFD by one single simulator block, but rather a cascade of reactor blocks can be necessary. This is the case in our running example. Therefore, *another link type*, namely for mapping reactor devices to reactor block cascades is defined (ReactorCascadeLink in Fig. 3.26). It assigns *one* Reaction (or one of its subtypes') *instance* to *multiple instances* of Reactor subtypes[29]. These instances are connected via their ports and connections with MATERIAL streams transporting the reacting substances. Again, sublinks help identifying related ports.

The ReactorCascadeLink *type* is *rather generic* compared to the PFRLink. For instance, it does not specify the number of reactor blocks used in the simulation, nor does it specify their concrete types. Even how they are connected by streams, is not further specified. To get an executable rule, a lot of information is still missing, which is supplied on the abstract instance level (see below).

The type level definitions of the rule set for our running example comprise much more link types. Some of them are more concrete, others are more generic than those discussed above. Even from the link types described so far, it can be seen that the definition of the relations between given document types is quite ambiguous. This reflects the complexity of the domain and the need for engineers' knowledge and creativity during integration. Thus, our integration *models* only *provide heuristics to support* the engineers at their work.

Abstract Instance Level Modeling

In general, the information modeled on the type level is not sufficient for gaining executable integration rules. Link types constrain links but do not fully specify the structures to be related by links. Therefore, on the abstract instance level, a detailed definition of the corresponding *patterns related by a link* is made. This is done by defining so-called *link templates* in UML collaboration diagrams. These instances are abstract because they do not describe concrete documents but situations that *may* occur in one concrete document at runtime and how the corresponding situation in the other document could

[27] Cardinalities are not shown in the figure.

[28] To keep the figure simple, it is not distinguished between ports of different orientation, as it is done in the real model.

[29] The Reactor class is abstract, thus no instances of it are allowed.

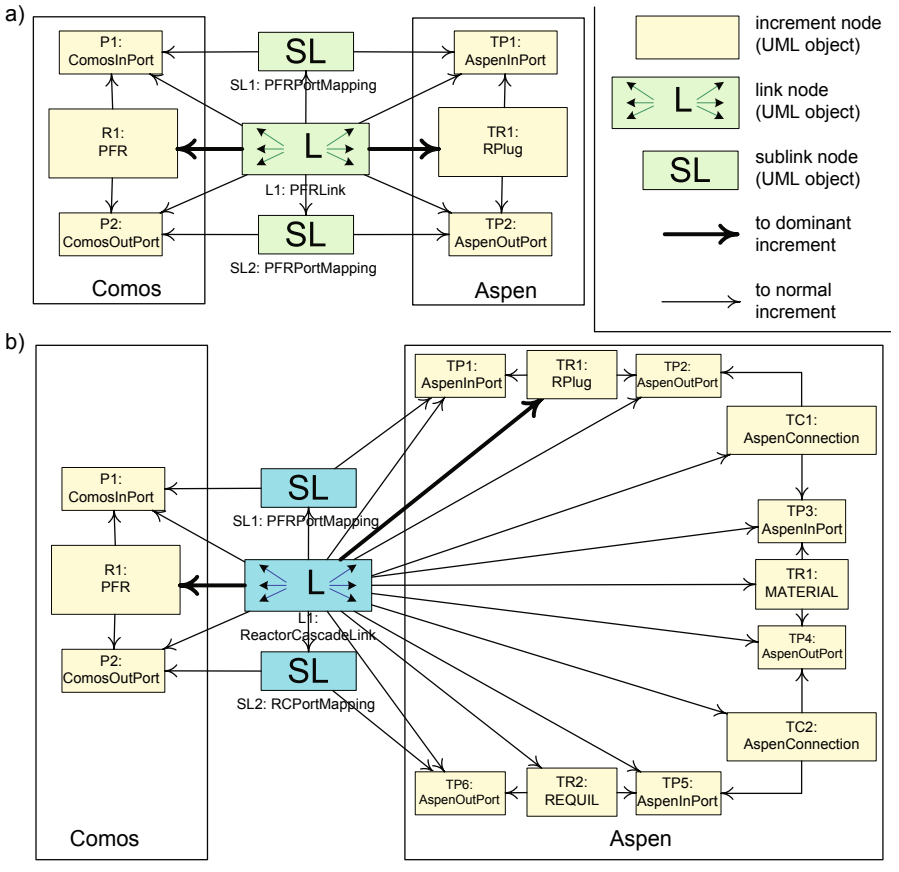

Fig. 3.27. Reactor link templates (UML collaboration diagram)

look like. From link templates operational integration rules that describe actions like "if the described pattern was found in the source document, create a corresponding pattern in the target document" can be derived easily.

To illustrate the definition of link templates, we again use the rule set of our running *example*. Figure 3.27 shows two UML *collaboration diagrams* defining link templates which are instances of the link types introduced in the previous subsection.

The *link template* in Fig. 3.27 a) *relates* a plug flow *reactor* (PFR) with two ports *to a* RPlug *block* with two ports in the simulation model. All increments are referenced by the link L1, which is refined by the sublinks SL.1 and SL.2 to map the corresponding ports. The port mapping is needed later to propagate connections between ports (see below). The link L1 is an instance of the PFRLink type. It could be an instance of the more generic link type ReactorCascadeLink as well, with the cascade just consisting of one reactor block.

However, a link should always be typed as instance of the most concrete link type available that fits its definition. Link types reflect domain knowledge and the more concrete a link type is the more likely it is that its instances are supported by domain knowledge.

In the running example, the simple link template of Fig. 3.27 a) was not applied, because a single simulator block does not suffice to study the reaction of the plug flow reactor. Instead, the PFR was mapped to a *reactor cascade*. Figure 3.27 b) contains the *link template* that is the basis of the corresponding integration rule. The PFD-side pattern is the same as in Fig. 3.27 a). On the simulation side, a cascade of a RPlug and a REQUIL block are defined. Substances are transported by a MATERIAL stream, whose ports are connected to the reactors' ports. Again, all increments in both documents are referenced by the link, which is an instance of ReactorCascadeLink, and the external ports of the cascade are associated with the PFR ports by sublinks.

The link types and templates discussed so far and some other rules in our running example address the mapping of devices and blocks together with their ports. If rules are derived from these templates and are then used, e.g. to transform the devices of a PFD into a simulation model, it is necessary to *transform* the *connections between* the *ports* as well. This is done with the help of the definitions in Fig. 3.28. Part a) of the figure contains the link template for mapping a ComosConnection to an AspenConnection. While the mapping

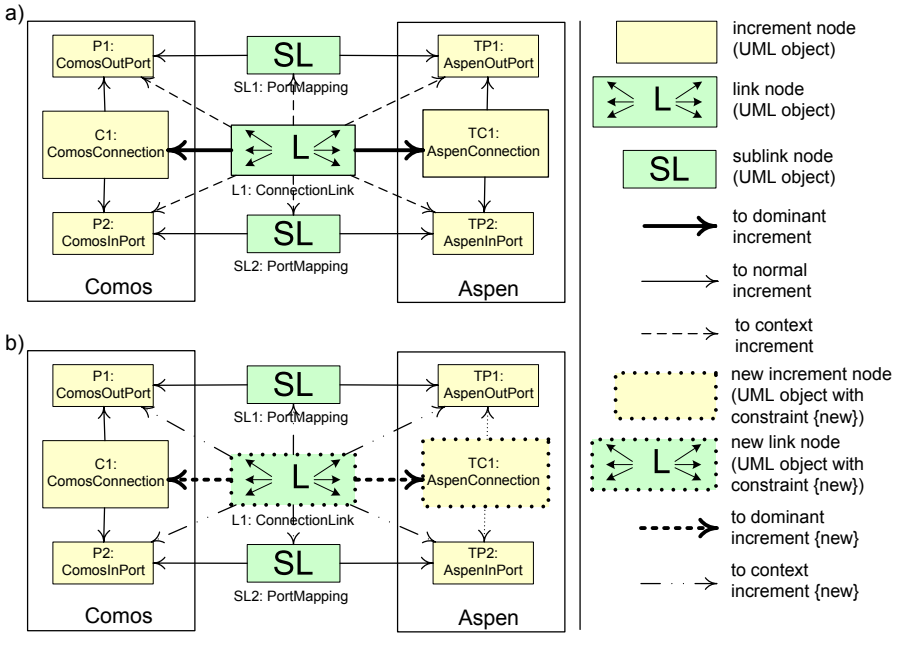

Fig. 3.28. Connection link template (a) and forward rule (b)

is quite simple with regard to its 1:1 structure, there is a particularity about this link template: To be able to embed the edges leading from the newly created connection in the target document to the correct ports, the source document ports have to be already mapped to their corresponding ports in the target document. As a result, the ports and their mapping are referenced as context information by the link L1. This ensures that the rule transforming connections is applied only after the ports have been mapped by applications of other rules and thus it can be determined which ports to connect. Then, the edges from the newly created connection to the ports can be created easily.

Modeling Operational Rules

The link templates described so far are *purely declarative* and just describe which patterns could be related by fine-grained inter-document links. They do *not* contain *operational* directives for transforming a document into another one or for establishing links between documents. Following the triple graph grammar approach [413], operational integration rules can be easily derived from link templates.

For each *link template*, *three integration rules* can be derived[30]:

- *Forward* transformation *rules* look for the context in source, target, integration document, and the non-context increments in the source document, as well as for all related edges. For each match, it creates the corresponding target document pattern and the link structure in the integration document.
- *Backward* integration *rules* do the same but in the opposite direction from target to source document.
- *Correspondence* analysis *rules* search the pattern in source and target document including the whole context information. For each match, the link structure in the integration document is created.

The derivation of a forward transformation rule from a link template is illustrated in Fig. 3.28, as an example, using the rule to transform a connection. Part b) shows the *forward rule* corresponding *to* the *link template* in part a). All dotted nodes (L1 and TC1) and their edges are created when the rule is executed. To determine whether the rule can be applied, the pattern without these nodes is searched in the documents. Here, the already related ports and the connection in the PFD are searched and the corresponding connection in the simulation model is created.

The notation of integration rules can be compared to *graph transformations* [328, 652] with left-hand and right-hand sides compressed into one diagram. The dotted nodes and edges are contained on the right-hand side only and thus are created during execution. The other nodes and edges are contained on both sides and are thus searched and kept.

[30] Additional rules can be derived if consistency checking and repairing existing links are taken into account.

So far, only the structural aspects of link templates and rules were addressed. In practice, each increment is further defined by some attributes and their values. Using a subset of the OCL language (Object Constraint Language [879], part of the UML specification), *conditions based on* these *attributes* that further constrain the applicability of the resulting integration rules can be added to link templates.

To deal with the consistency of attributes, link templates can be enriched with different *attribute assignment statements* using a subset of the OCL language as well. An attribute assignment can access all attributes of the increments referenced by a link. There are different situations in development processes in which an integration is performed. Depending on the situation, an *appropriate* attribute assignment is *chosen*. For instance, for each correspondence (i.e., for the set of resulting rules) there is one attribute assignment for the initial generation of the simulation model, one to propagate the simulation results back into the flowsheet, etc.

Rule Definition Round-Trip

Figure 3.29 shows the interrelations between the different *parts* of the *modeling formalism* from a practical point of view. The meta model serves as basis both for the implementation of integrator tools and the rule modeling process. It is defined according to domain-specific knowledge like, in our case, the information model CLiP [20] for chemical engineering and the requirements concerning integration functionality.

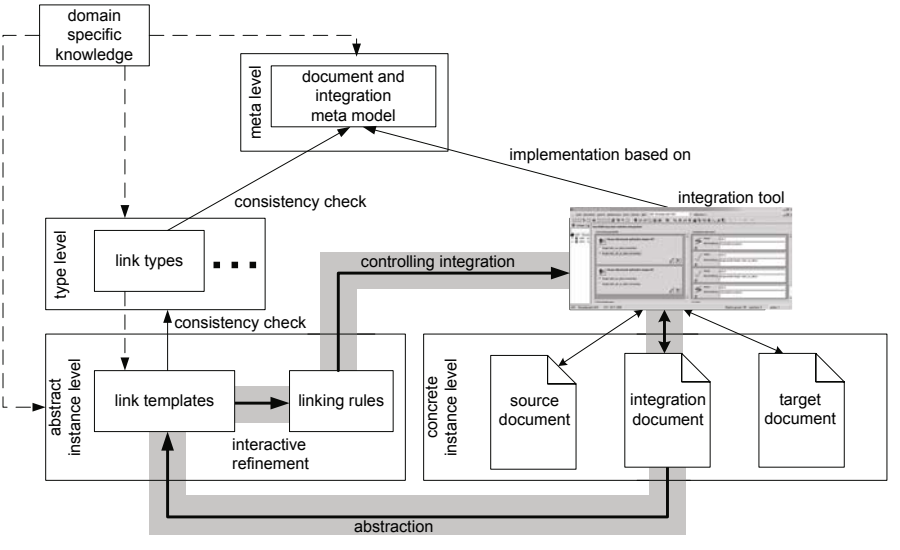

Fig. 3.29. Layers of modeling and modeling round-trip

Basically, there are two ways to define integration rules: *top-down*, before the integrator tool is applied, or *bottom-up*, based on situations occurring during the usage of the tool. It is most likely that in practice first a basic set of rules is defined top-down by a *modeling expert* and then the rule base is extended bottom-up by the *engineer using* the integrator tool.

Before any further definitions can be made, the *documents* to be integrated have to be *modeled* on type level as described above, which is not shown in this figure. Next, *link types* have to be *defined* on type level that declare types for possible correspondences on the abstract instance level. Again, for both tasks domain-specific knowledge has to be used.

Following a top-down approach, *link templates* on the abstract instance level are modeled based on the link types of the type level. These are then refined to *linking rules*. The resulting set of rules is *used by* the *integrator* to find corresponding parts of source and target document and to propagate changes between these two documents. The corresponding document parts are related by *links* stored in the *integration document*. If no appropriate rule can be found in a given situation, the chemical engineer performing the integration can *manually modify* source and target document and add links to the integration document.

To extend the rule base *bottom-up*, the links entered manually in the integration document can be automatically *abstracted to* link *templates*. Next, a consistency check against the link types on the type level is performed. If the link templates are valid, the engineer is now guided through the interactive *refinement* of the link templates to *linking rules* by a simplified modeling tool. The rules are *added* to the rule base and can be *used* for the following integrations.

This can be *illustrated* by an *extension* of the *scenario* presented above: Initially, there is no rule for mapping a plug flow reactor to a cascade of two reactors. Instead, the first time the situation occurs, the mapping is performed manually: The user creates the reactor cascade in the simulation model and adds a link to the integration document. From this link, the link template in Fig. 3.27 b) is abstracted. The link template is consistency-checked against the available link types. It is detected that the link template fits the ReactorCascadeLink type from Fig. 3.26 and, therefore, it is permanently added to the rule base and applied in further runs of the integrator.

3.2.4 Rule Execution

In this subsection, the *execution algorithm* for integration *rules* is presented. First, the triple graph grammar approach which serves as the basis for our approach is briefly sketched. Furthermore, it is explained how our work relates to this approach and why it had to be extended. Second, an overview of our integration algorithm is given, using a simple and abstract scenario. Third, the individual steps of the algorithm are explained in detail, using the integration rule for a connection as an example. In this subsection, the execution

of the algorithm with PROGRES [414] is considered. Please note that the execution with PROGRES is only one approach for building integrator tools (cf. Subsect. 3.2.5).

Triple Graph Grammars and Execution of Integration Rules

For modeling an integration, the source and target documents as well as the integration document may be modeled as graphs, which are called *source graph*, *target graph*, and *correspondence graph*, respectively. If the tools operating on source and target documents are not graph-based, the graph views can be established by tool wrappers (cf. Subsect. 5.7.4). Moreover, the operations performed by the respective tools may be modeled by graph transformations.

Triple graph grammars [413] were developed for the high-level specification of graph-based integrator tools. The core idea behind triple graph grammars is to specify the relationships between source, target, and correspondence graphs by *triple rules*. A triple rule defines a coupling of three rules operating on source, target, and correspondence graph, respectively. By applying triple rules, we may modify coupled graphs synchronously, taking their mutual relationships into account. In the following, we give a short motivation for our integration algorithm only. For a detailed discussion of the relation between our approach and the original work by Schürr, the reader is referred to [37].

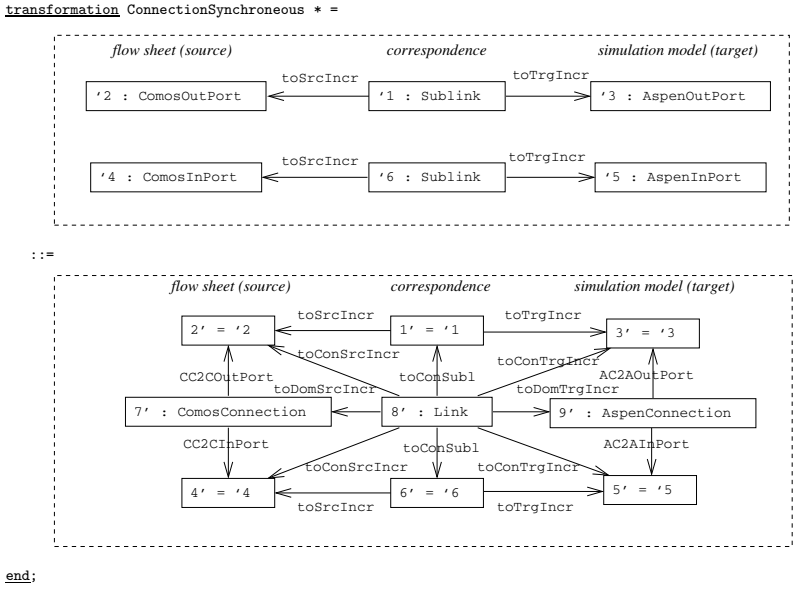

Fig. 3.30. Triple rule for a connection

An *example* of a *triple rule* is given in Fig. 3.30. The rule refers to the running example to be used for the integration algorithm, namely the *creation* of *connections* as introduced in Fig. 3.28 a). Here, the triple rule is presented as a graph transformation in PROGRES [414] syntax.

A *graph transformation* consists of a left-hand and a right-hand side, which are displayed on top or bottom, respectively. Each side contains a graph pattern consisting of nodes and interconnecting edges. When a graph transformation is applied to a graph, the left-hand side pattern is searched in the graph (pattern matching) and replaced by the right-hand side pattern. All nodes on the left-hand side and new nodes on the right-hand side are further specified by giving their type. The nodes on the right-hand side that appear on the left-hand side as well are related to the corresponding left-hand side nodes by their node numbers.

Both sides of the triple rule `ConnectionSynchronous` span all participating subgraphs: the source graph (representing the PFD) on the left, the correspondence graph in the middle, and the target graph (for the simulation model) on the right. The triple rule can be seen as a different representation of the link template in Fig. 3.28 a). Its *left-hand side* is composed of all context nodes of the link template: It contains the port nodes in the source and target graphs, distinguishing between output ports and input ports. Furthermore, it is required that the port nodes in both graphs correspond to each other. This requirement is expressed by the nodes of type `subLink` in the correspondence graph and their outgoing edges which point to nodes of the source and target graph, respectively.

The *right-hand side* contains all nodes of the link template: All elements of the left-hand side reappear on the right-hand side. New nodes are created for the connections in the source and target graph, respectively, as well as for the link between them in the correspondence graph. The connection nodes are embedded locally by edges to the respective port nodes. For the link node, three types of adjacent edges are distinguished. `toDom`-edges are used to connect the link to exactly one dominant increment in the source and target graph, respectively. In general, there are additional edges to normal increments (not needed for the connection rule). Finally, `toContext`-edges point to context increments.

Figure 3.30 describes a *synchronous graph transformation*. As already explained earlier, we cannot assume in general that all participating documents may be modified synchronously. In the case of asynchronous modifications, the triple rule shown above is not ready for use. However, we may derive asynchronous forward, backward, or correspondence analysis rules as explained in Subsect. 3.2.3. Figure 3.31 shows the *forward* rule for a connection from Fig. 3.28 b) in PROGRES syntax. In contrast to the synchronous rule, the connection in the PFD is now searched on the left-hand side, too, and only the connection in the simulation model and the link with its edges are created on the right-hand side.

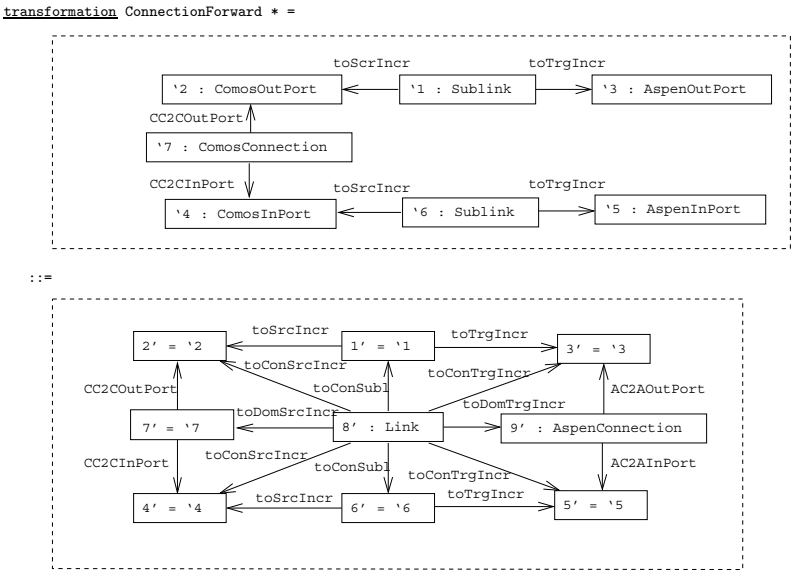

Fig. 3.31. Forward rule for a connection

Unfortunately, even these rules are not ready for use in an integrator tool as described in the previous section. In case of non-deterministic transformations between interdependent documents, it is crucial that the user is made aware of conflicts between applicable rules. A conflict occurs, if multiple rules match the same increment as owned increment. Thus, we have to consider all applicable rules and their mutual conflicts before selecting a rule for execution. To achieve this, we have to *give up atomic rule execution*, i.e., we have to decouple pattern matching from graph transformation [33, 255].

Integration Algorithm

An integration rule cannot be executed by means of a single graph transformation. To ensure the correct sequence of rule executions, to detect all conflicts between rule applications, and to allow the user to resolve conflicts, all integration rules contained in the rule set for an integrator are automatically translated to a set of graph transformations. These rule-specific transformations are executed together with some generic ones altogether forming the *integration algorithm*.

While the algorithm supports the concurrent execution of forward, backward, and correspondence analysis rules, we *focus* on the *execution* of *forward rules* here. Also, we present the *basic* version of the algorithm only, without optimizations. A full description of all aspects can be found in [29] which is an extended version of [33].

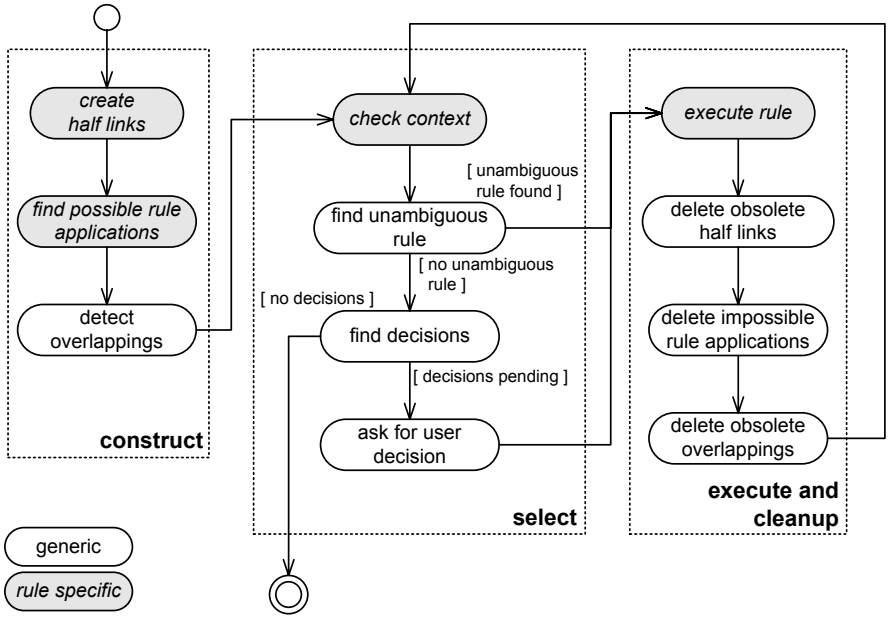

Fig. 3.32. Integration algorithm

The execution of integration rules is embedded in an *overall algorithm* which can be briefly sketched as follows: When performing an integration, first all links already contained in the integration document are checked for consistency. Links can become inconsistent due to modifications applied to the source and target documents by the user after the last run of the integrator tool. In this case, they can be interactively repaired by applying repair actions proposed by the integrator tool or fixed manually by adding or removing increment references to the link or by simply deleting the link. For an initial integration the integration document is empty, so this applies only to subsequent integrations. After existing links have been dealt with, rules are executed for all increments that are not yet referenced by links. In case of a subsequent integration, these increments have been added by the user to source and target documents since the last run of the integrator tool.

Figure 3.32 shows a UML *activity diagram* depicting the integration algorithm. To perform each activity, one or more graph transformations are executed. Some of these graph transformations are generic (white), others are specific for the integration rule being executed (grey and italics). Thus, the algorithm is composed of all *generic* and *rule-specific graph transformations*, the latter for all integration rules contained in the rule set. The overall algorithm is divided into *three phases*, which are described informally in the following using the example of Fig. 3.33. The example is rather abstract and is not related to specific rules of our scenario.

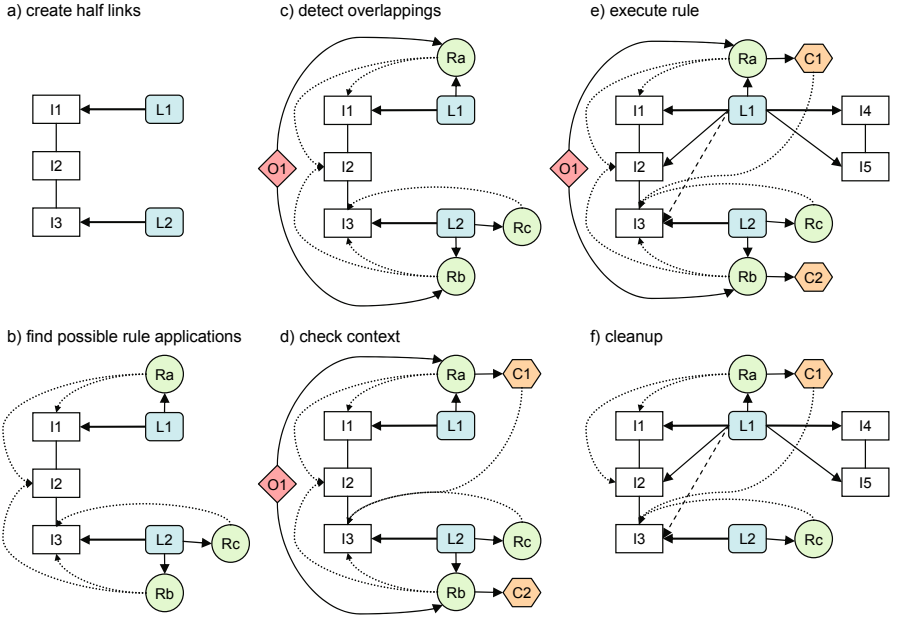

Fig. 3.33. Sample integration

During the *first phase* (construct), all possible rule applications and conflicts between them are determined and stored in the graph. First, for each increment in the documents that has a type compatible with the dominant increment's type of any rule, a *half link* is *created* that references this increment. In the example, half links are created for the increments I1 and I3, and named L1 and L2, respectively (cf. Fig. 3.33 a).

Then, for each half link the possible *rule applications* are *determined*. This is done by trying to match the left-hand side of forward transformation rules, starting at the dominant increments to avoid global pattern matching. In the example (Fig. 3.33 b), three possible rule applications were found: Ra at the link L1 would transform the increments I1 and I2; Rb would transform the increments I2 and I3; and Rc would transform increment I3.

Here, two types of conflicts can be found. First, the rules Rb and Rc share the same dominant increment. Second, the rules Ra and Rb share a normal increment. Both situations lead to conflicts because each increment may only be transformed by one rule as normal or dominant increment. To prepare conflict-resolving user interaction, *conflicts* of the second type are *explicitly marked* in the graph by adding an edge-node-edge construct (e.g. O1 in Fig. 3.33 b).

In the *second phase* (select), the *context* is *checked* for all possible rule applications and all matches are stored in the graph. Only rules whose context has been found are ready to be applied. In the example in Fig. 3.33 d), the

context for Ra consisting of increment I3 in the source document was found (C1). The context for Rb is empty (C2), the context for Rc is still missing.

If there is a possible rule application, whose context has been found and which is not involved in any conflict, it is *automatically selected* for execution. Otherwise, the user is asked to *select* one rule among the rules with existing context. If there are no executable rules the algorithm ends. In the example in Fig. 3.33 d), no rule can be automatically selected for execution. The context of Rc is not yet available and Ra and Rb as well as Rb and Rc are conflicting. Here, it is assumed that the user selects Ra for execution.

In the *third phase* (execute and cleanup), the selected *rule* is *executed*. In the example (Fig. 3.33 e), this is the rule corresponding to the rule node Ra. As a result, increments I4 and I5 are created in the target document, and references to all increments are added to the link L1. Afterwards, *rules* that cannot be applied and *links* that cannot be made consistent anymore are *deleted*. In Fig. 3.33 f), Rb is deleted because it depends on the availability of I2, which is now referenced by L1 as a non-context increment. If there were alternative rule applications belonging to L1 they would be removed as well. Finally, *obsolete overlappings* have to be deleted. In the example, O1 is removed because Rb was deleted. The cleanup procedure may change depending on how detailed the integration process has to be documented.

Now, the *execution goes back* to the select phase, where the context check is repeated. Finally, in our example the rule Rc can be automatically selected for execution because it is no longer involved in any conflicts, if we assume that its context has been found.

In the following, some of the rule-specific and generic graph transformations needed for the execution of the connection rule will be explained in more detail.

Construction Phase

In the construction phase, it is determined which *rules* can be possibly applied to which *subgraphs* in the source document. Conflicts between these rules are marked. This information is collected once in this phase and is updated later *incrementally* during the repeated executions of the other phases.

In the first step of the construction phase (create half links), for each increment, which type is the type of a *dominant increment* of at least one rule, a link is created that references only this increment (*half link*). Dominant increments are used as anchors for links and to group decisions for user interaction. Half links store information about possible rule applications; they are transformed to consistent links after one of the rules has been applied.

To *create half links*, a *rule-specific* PROGRES *production* (not shown) is executed for each rule. Its left-hand side contains a node having the type of the rule's dominant increment, with the negative application condition that there is no half link attached to it yet. On its right-hand side, a half link

node is created and connected to the increment node with a `toDomSrcIncr`-edge. All these productions are executed repeatedly, until no more left-hand sides are matched, i.e., half links have been created for all possibly dominant increments.

The second step of the construction phase (find possible rule applications) determines the integration rules that are possibly applicable for each half link. A rule is *possibly applicable* for a given half link if the source document part of the left-hand side of the synchronous rule without the context increments is matched in the source graph. The dominant increment of the rule has to be matched to the one belonging to the half link. For potential applicability, context increments are not taken into account, because missing context increments could be created later by the execution of other integration rules. For this reason, the context increments are matched in the selection phase before selecting a rule for execution.

Figure 3.34 shows the PROGRES transformation for the example forward rule for a connection of Fig. 3.28 b). The left-hand side consists of the half link (`‘2`) and the respective dominant increment (`‘1`), as all other increments of this rule are context increments. In general, all non-context increments and their connecting edges are part of the left-hand side. The link node is further constrained by a *condition* that requires the attribute `status` of the link to have the value `unchecked`. This ensures that the transformation is only applied to half links that have not already been checked for possible rule applications.

On the right-hand side, a rule *node* is created to *identify* the possible *rule application* (`4’`). A transfer is used to store the id of the rule in its attribute `ruleId`. A `possibleRule`-edge connects it to the half link. A role node is inserted to explicitly store the result of the pattern matching (`3’`). If there are more increments matched, role nodes can be distinguished by the `roleName`-attribute. The asterisk (`*`) behind the production name tells PROGRES to

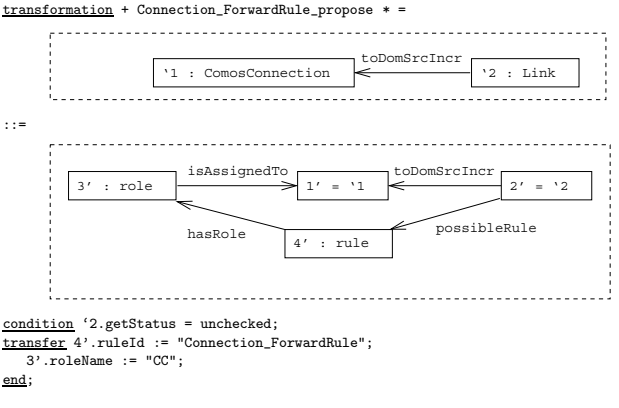

Fig. 3.34. Find possible rule applications

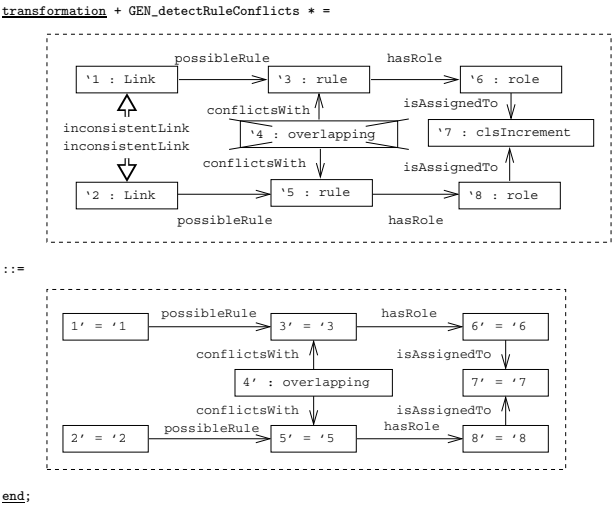

Fig. 3.35. Detect overlappings

apply this production for each possible matching of its left-hand side. When executed together with the corresponding productions for the other rules, as a result all possibly applicable rules are stored at each half link. If a rule is applicable for a half link with different matchings of its source increments, multiple rule nodes with the corresponding role nodes are added to the half link.

In the selection phase, for each link that is involved in a conflict all possible rule applications are presented to the user who has to resolve the conflict by selecting one. Thus, these conflicts are directly visible. *Conflicts* where possible rule applications share the *same normal increment* are *marked* with cross references (hyperlinks) between the conflicting rule applications belonging to different links. This is done with the help of the generic PROGRES production in Fig. 3.35. The pattern on the left-hand side describes an increment (`'7`) that is referenced by two roles belonging to different rule nodes which belong to different links. The negative node `'4` prevents the left-hand side from matching if an overlap is already existing and therefore avoids multiple markings of the same conflict. The arrows pointing at the link nodes, each labeled `inconsistentLink`, call a PROGRES *restriction* with one of the link nodes as parameter. A restriction can be compared to a function that has to evaluate to true for the restricted node to be matched by the left-hand side. The definition of the restriction is not shown here. It evaluates to true, if the link's attributes mark it as being inconsistent.

On the *right-hand* side, the *conflict* is *marked* by adding an overlap node (`4´`) is inserted between the two rule nodes. Again, this production is marked with an asterisk, so it is executed until all conflicts are detected. Besides detecting conflicts between different forward transformation rules, the depicted

production also detects *conflicts between forward, backward, and correspondence analysis rules* generated from the same synchronous rule. Thus, to prevent unwanted creation of redundant increments, it is not necessary to check whether the non-context increments of the right-hand side of the synchronous rule are already present in the target document when determining possible rule applications in the second step of this phase.

Selection Phase

The goal of the selection phase is to *select* one possible *rule* application for execution in the next phase. If there is a rule that can be executed without conflicts, the selection is performed *automatically*, otherwise the *user* is *asked* for his decision. Before a rule is selected, the contexts of all rules are checked because only a rule can be executed whose context has been found.

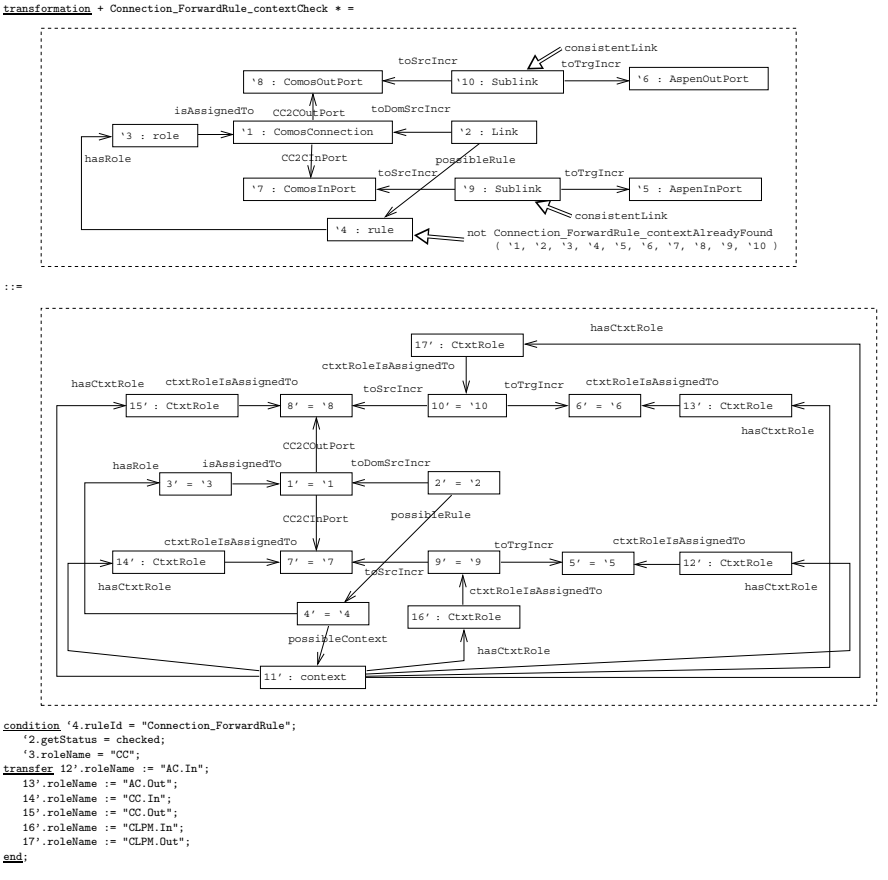

Fig. 3.36. Check context

The *context check* is performed in the first step of this phase. The context is formed by all context elements from the synchronous rule. It may consist of increments of source and target documents and of links contained in the integration document.

Figure 3.36 shows the PROGRES *production checking the context* of the example integration rule. The left-hand side contains the half link (‘2), the non-context increments (here, only ‘1), the rule node (‘4), and the role nodes (‘3). The non-context increments and their roles are needed to embed the context and to prevent unwanted folding between context and non-context increments. For the example rule, the context consists of the two ports connected in the source document (‘7, ‘8), the related ports in the Aspen document (‘5, ‘6), and the relating sublinks (‘9, ‘10). The restrictions make sure that the sublinks belong to a consistent link.

On the right-hand side, to mark the matched context, a new *context node* is created (‘11). It is *connected* to all nodes belonging to the context by *role nodes* (12’, 13’, 14’, 15’, 16’, 17’) and appropriate *edges*. If the matching of the context is ambiguous, *multiple* context nodes with their roles are created as the production is executed for all matches.

As the selection phase is executed repeatedly, it has to be made sure that each *context match* (context node and role nodes) is *added* to the graph only *once*. The context match cannot be included directly as negative nodes on the left-hand side because edges between negative nodes are prohibited in PROGRES. Therefore, this is checked using an additional graph test which is called in the restriction on the rule node. The graph test is not presented here as it is rather similar to the right-hand side of this production[31].

The *context* is *checked* for all possible *rule applications*. To make sure that the context belonging to the right rule is checked, the rule id is constrained in the condition part of the productions. After the context of a possible rule application has been found, the rule can be applied.

After the context has been checked for all possible rule applications, some rules can be applied, others still have to wait for their context. The next step of the algorithm (find unambiguous rule) tries to find a *rule application* that is *not* involved in any *conflict*. The conflicts have already been determined in the construction phase. As any increment may be referenced by an arbitrary number of links as context, no new conflicts are induced by the context part of the integration rules. The generic PROGRES production in Fig. 3.37 finds rule applications that are not part of a conflict. On the left-hand side a rule node is searched (‘1) that has only one context node and is not related to any overlap node. It has to be related to exactly one half link (‘2) that does not have another rule node.

For forward transformation rules, a rule node belongs to one link only, whereas nodes of correspondence analysis rules are referenced by two half

[31] In the optimized version of the integration algorithm, the context check is performed only once for each rule, thus this test is avoided.

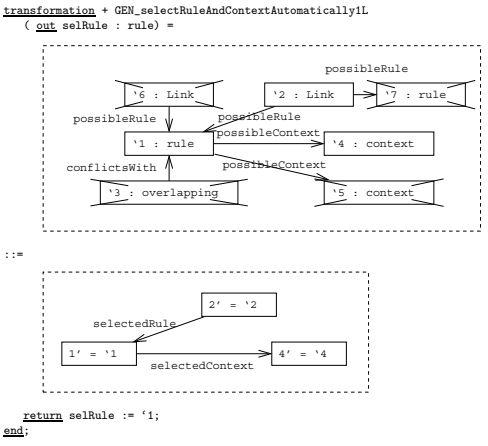

Fig. 3.37. Select unambiguous rule

links. Therefore for *correspondence* analysis *rules*, *another production* is used which is not shown here.

A rule node is not selected for execution if there are *conflicting rules*, even if their *context* is still *missing*. As the context may be created later, the user has to decide whether to execute this rule and thereby making the execution of the other rules impossible.

If a *match* is *found* in the host graph, the *rule* node and the context node are *selected* for execution by substituting their referencing *edges* by `selectedRule` and `selectedContext` edges, respectively (cf. right-hand side of production in Fig. 3.37). The rule node is returned in the output parameter `selRule`. Now, the corresponding rule can be applied in the execution phase.

If *no rule* could be selected *automatically*, the *user* has to decide which rule is to be executed. Therefore, in the next step (find decisions), all conflicts are collected and presented to the user. For each half link, all possible rule applications are shown. If a rule application conflicts with another rule of a different half link, this is marked as annotation at both half links. Rules that are not executable due to a missing context are included in this presentation but cannot be selected for execution. This information allows the user to *select* a rule *manually*, knowing which other rule applications will be made impossible by his decision. The result of the user interaction (ask for user decision) is stored in the graph and the selected rule is executed in the execution phase.

If *no rule* could be selected automatically and there are no decisions left, the *algorithm terminates*. If there are still half links left at the end of the algorithm, the user has to perform the *rest* of the integration *manually*.

Execution Phase

The *rule* that was selected in the selection phase is *executed* in the execution phase. Afterwards, the *information* collected during the construction phase has to be *updated*.

Rule *execution* is performed by a *rule-specific* PROGRES *production*, see Fig. 3.38. The *left-hand side* of the production is nearly identical to the right-hand side of the context check production in Fig. 3.36. The main difference is that, to identify the previously selected rule, the edge from the link (`‘2`) to the rule node (`‘4`) is now a `selectedRule` edge and the edge from the rule node to the context node (`‘11`) is a `selectedContext` edge. The `possibleRule` and `possibleContext` edges are replaced when a rule together with a context is selected for execution either by the user or automatically in the previous phase of the algorithm (see above).

On the *right-hand side*, the new *increments* in the *target* document are *created* and embedded by *edges*. In this case, the connection (`18’`) is inserted

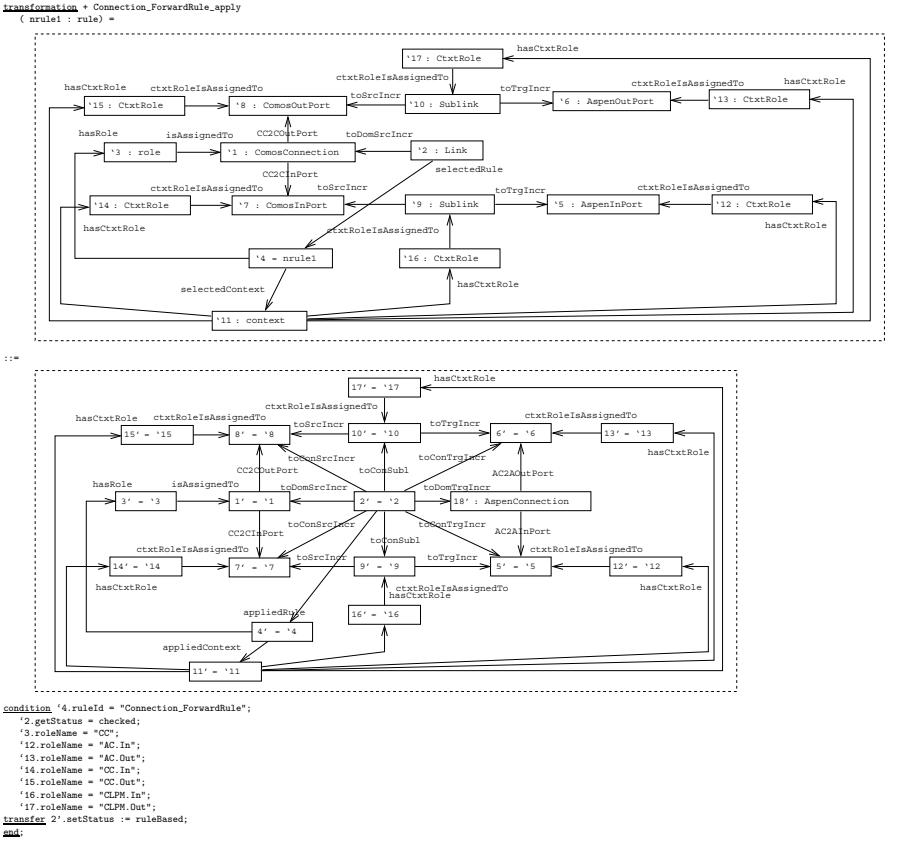

Fig. 3.38. Execute rule

and connected to the two Aspen ports (`5', 6'`). The half *link* (`2'`) is *extended* to a full link, referencing all context and non-context increments in the source and the target document. The information about the applied rule and roles etc. is kept to be able to detect inconsistencies occurring later due to modifications in the source and target documents.

The last steps of the algorithm are performed by generic productions not shown here that update the information about possible rule applications and conflicts. First, obsolete *half links* are *deleted*. A half link is obsolete if its dominant increment is referenced by another link as non-context increment. Then, potential *rule applications* that are no longer possible because their potentially owned increments are used by another rule execution are *removed*.

3.2.5 Implementation

Besides realizing integrators completely on an ad-hoc hardwired basis, there are *four* different *alternatives* for implementing a given set of integration rules (see Fig. 3.39).

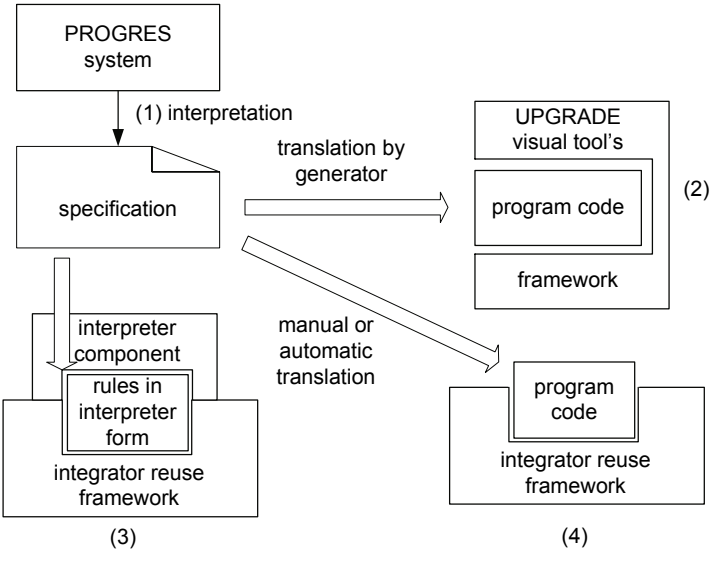

Fig. 3.39. Four different possibilities for realizing integrators based on a specification, (2) and (3) being discussed in detail below

Different Realization Strategies

Two of them are based on the academic PROGRES system which allows for defining and executing graph transformation systems [414]. Both have in common that the set of integration rules has to be available as a PROGRES graph

transformation specification, which can be generated as described in the previous subsection. Following alternative (1), the *specification* is *interpreted* within the PROGRES system. As the PROGRES interpreter does not offer domain- and application-specific visualization and graph layout, this alternative is not considered in this paper.

The second alternative (2) is based on first *generating* program *code* from the PROGRES specification and then compiling it together with the *UPGRADE* visual tool's framework [49]. This results in a PROGRES-independent prototype with graphical user interface.

Alternatives (3) and (4) are based on a specific *framework* designed for the realization of *integrator tools* by making use of reuse. These alternatives follow a more industrial approach. So, they do not need the academic platform used for alternatives (1) and (2). Following alternative (3), integration rules are *interpreted* at runtime by a specific interpreter component. Alternative (4) is to include program *code* for all integration rules – either generated automatically or written manually – into the framework.

In the following, we are focusing on alternatives (2) and (3) which are compared in Fig. 3.40. We give a short overview and explain the common ground as well as the differences of both approaches. In the following subsections, we will present each of them in more detail.

Both implementation approaches rely on a set of integration rules as having been described in Subsect. 3.2.3. For *modeling* these *rules*, we provide a special *editor* which is shared by both approaches. To reduce the implementation efforts, the rule editor uses the commercial case tool Rational Rose [743] as a basis, since it already provides basic modeling support for all types of UML diagrams needed by our modeling approach. To supply modeling features *specific* for *integration* rules, a plug-in implemented in C# (about 8600 lines of code) was created.

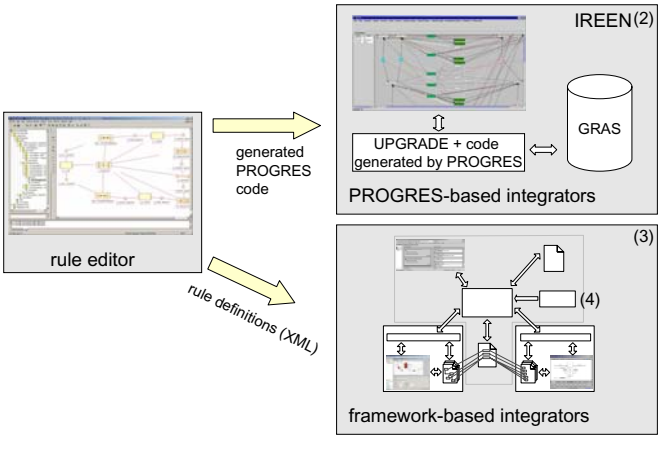

Fig. 3.40. Different implementation approaches

The integration rule editor comprises *all layers* of our *modeling approach* which are organized in different model packages: The meta layer package contains class diagrams for the generic integration meta model, which are fixed, as well as for domain-specific meta model extensions, which can be added and modified. In the type layer package, class diagrams for document type definitions and class diagrams for link type definitions can be created. In the abstract instance package, link templates and rules can be defined using UML collaboration diagrams.

The plug-in provides consistency *check* support *between* the *layers*: the type layer is checked against the meta layer and the instance layer is checked against the type and meta layers. For both checks, all detected inconsistencies are listed in the user interface and the affected model elements are highlighted. This helps the user to remove the inconsistencies. Additionally, the plug-in is able to derive forward, backward and consistence analysis rules from a given link-template. After the model is checked for its consistency and integration rules are derived, integration rules are exported in the XML dialect for graphs GXL [567, 732].

Both implementation approaches *apply* the *integration algorithm* presented in Subsect. 3.2.4. They differ in how the algorithm is executed.

Following the approach in the upper half of Fig. 3.40 (2), *PROGRES code* is derived for the rule-specific steps of the algorithm from the GXL rule definitions and combined with a predefined specification of the generic ones. Then, using PROGRES' *code generation* capabilities and the UPGRADE framework [49], an integrator prototype with a GUI is derived. Up to now, integrators realized by this approach are not connected to existing real-world applications. Instead, they are used for the quick evaluation of integration rules and of the integration algorithm itself. This realization method is called IREEN (Integration Rule Evaluation ENvironment). Current work at our department aims at providing a distributed specification approach for PROGRES [50] and interfaces to arbitrary data sources for UPGRADE [46]. Forthcoming results of this work could be used in the future to connect integrators realized by this approach to existing tools.

Up to now, integrator tools to be used in an industrial context and integrating existing applications are realized differently. *Integration rules* contained in GXL files are *interpreted* at runtime by a *C++-based integrator framework* (lower half of Fig. 3.40, (3). This approach was already sketched in Subsect. 3.2.2 (cf. Fig. 3.22). Besides interpreting rules, which is done for most rules, pre-compiled rules can be linked to the integrator as well (4). Up to now, these rules have to be hand-coded, but a C++ code generation comparable to the PROGRES code generation could be realized. The integrator is connected to the existing applications by tool wrappers which provide graph views on the tools' data.

Realization with PROGRES and Prototyping

Figure 3.41 gives an overview of how a PROGRES-based *integrator prototype* including a graphical user interface can be derived from a given integration rule set, following the *IREEN method*. First, from each synchronous triple graph rule being contained in the integration rule set to be executed a forward, a backward, and a correspondence analysis rule is derived as explained in Subsects. 3.2.3 and 3.2.4.

As mentioned before, the integration algorithm for rule execution consists of rule-specific and generic graph transformations. The *rule-specific graph transformations* are automatically derived from the UML collaboration diagrams containing all forward, backward, and correspondence analysis rules using a code generator. The generator output is an incomplete PROGRES graph transformation system [412, 414, 481] containing all rule-specific transformations for all rules.

To obtain a complete and executable specification, the partial specification has to be *combined* with three generic *specification parts*: One specification contains the static integration-specific parts, which are the integration graph scheme, the overall integration algorithm control, and the generic transformations for the algorithm. Additionally, for both source and target document there is a specification containing the document's graph scheme and some operations allowing the user to modify the document. Currently, the specifications for source and target documents are constructed manually. In general, it is possible to – at least partially – derive these specifications from UML models as well.

The *complete specification* is *compiled* by the PROGRES system resulting in C code which is then *embedded* in the UPGRADE framework [49, 206]. This

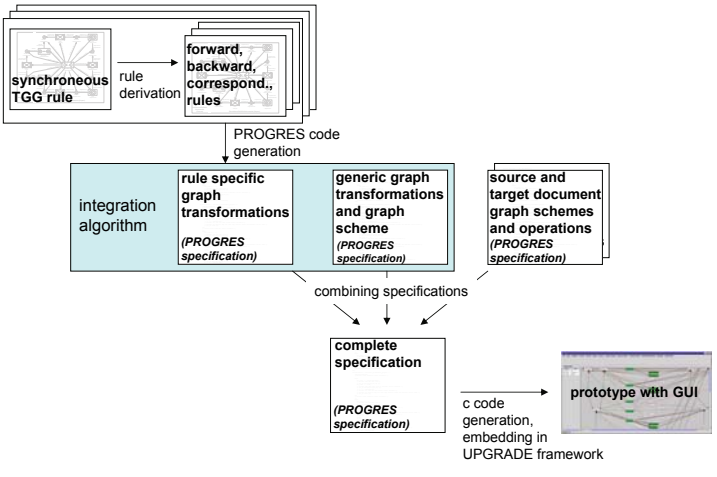

Fig. 3.41. Composition of PROGRES specification

leads to a *prototype* with a graphical user interface which allows construction and modification of source and target documents as well as performing runs of the integrator tool. All documents reside in the underlying graph database GRAS [8]. Some additional coding is required for application-specific layout algorithms and user interfaces. However, these efforts can be kept small because the resulting prototypes are not targeted at the end user. Instead, they are intended to serve as proof of concept, i.e., for the evaluation of integration rules without having to deal with real applications.

Industrial Realization

For the practical realization of integrators, i.e. for demonstrators in industry, an *integrator framework* is used. The framework is implemented in C++ and comprises about 14.000 lines of code. We focus here on a sketchy survey, because an overview of the system components of this framework has already been given in Subsect. 3.2.2 (cf. Fig. 3.22),

The *architecture* of the integrator framework is displayed in Fig. 3.42. The package IntegratorCore contains the execution mechanism and the overall control of the integrator. The package IntegrationDoc provides functionality for storing, retrieving, and modifying links in the integration document. DocumentWrapper consists of interface definitions that all tool wrappers have to implement. As there are no generic wrappers, there is no implementation in this package. The packages mentioned so far correspond to the main components in the system architecture of Fig. 3.22.

There are two additional packages: First, IntegrationGraphView provides an *integrated graph view* on source, target, and integration document. Therefore, it uses the corresponding document packages. Second, GraphPatternHandling supplies graph *pattern matching* and *rewriting functionality*. This functionality is rather generic and could be used for arbitrary graph rewriting tasks.

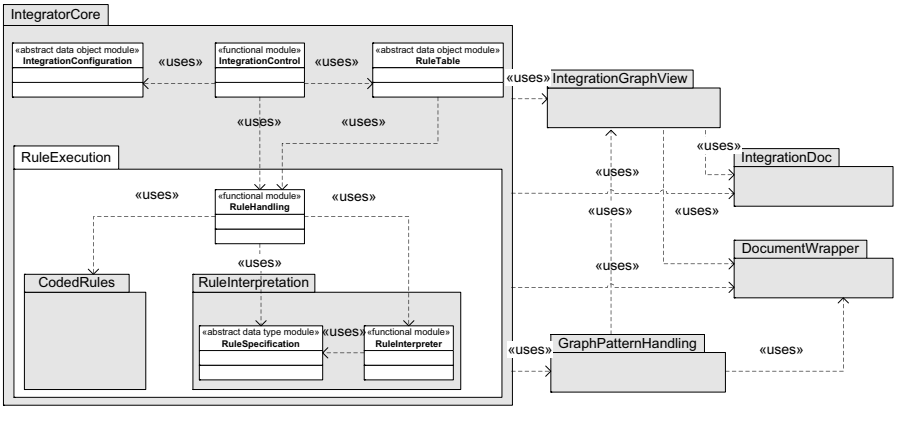

Fig. 3.42. Integrator framework

Indeed, some of the pattern matching algorithm optimizations incorporated in PROGRES have been re-implemented in the package. But as the graph rewriting formalism for integration rules is limited, so far the package only supplies the features needed for integrators.

The module IntegrationControl in the integrator core provides the *overall control* for enacting integrations. For instance, its interface offers methods that are used by the GUI to start or restart the integrator. To customize a run of the integrator, administrative settings are stored in IntegrationConfiguration and read by IntegrationControl. At the start of the integrator, all integration rules are stored in the module RuleTable. The sub-package RuleExecution implements the integration algorithm introduced in Subsect. 3.2.4. Rule-independent steps of the algorithm are implemented in RuleHandling. Rule-specific steps are either implemented directly in the sub-package CodedRules or executed by the rule interpreter (sub-package RuleInterpretation). The module RuleHandling serves as a "router", either calling a rule-specific piece of code in CodedRules for coded rules, or handing the execution over to the rule interpreter. For either type of rule, the realization of the algorithm steps is mostly based on graph pattern handling. But unlike the PROGRES-based implementation, some algorithm steps can be implemented by calling methods of the integration document package directly to provide a more specific and, thereby, more efficient implementation.

Prototype Demonstrator

Our integration approach described so far has been *applied* in a *cooperation* with our industrial partner innotec GmbH. Innotec is a German software company and the developer and vendor of the integrated engineering solution Comos PT. In our cooperation, the integrator for Comos PT process flow diagrams and Aspen Plus simulation models as described in the motivating example (cf. Subsect. 3.2.1) has been implemented.

The *integrator realization* is based on an early version of the C++ integrator framework which is interpreting integration rules at runtime. Integration rules are modeled using the formalism described in Subsect. 3.2.3 with Rational Rose, the rule modeling plug-in is used to export the rules to XML files. Figure 3.43 shows the graphical user interface of the integrator.

The *user interface* is integrated into the Comos PT environment as a plug-in. It is divided into *two parts*: On the left-hand side, all pending decisions between alternative rule applications are listed. The user has to choose a rule before the integration can proceed. On the right-hand side, all links in the integration document are shown. Symbols illustrate the links' states and for each link a detailed description showing the related increments can be opened. The integrator realized so far only addresses the integration of PFDs and simulation models. Future work aims at providing an integration platform for Comos PT for arbitrary integration problems (cf. Sect. 7.6).

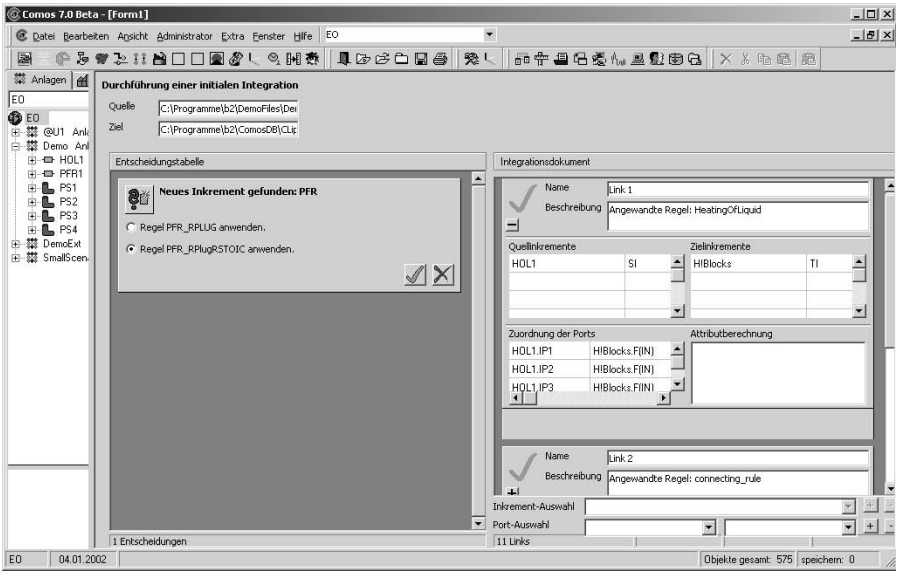

Fig. 3.43. User interface of the integrator for Comos PT PFDs and Aspen Plus simulation models (in German)

3.2.6 Additional Integrators

Apart from integrators described so far, some *further integrators* have been realized in IMPROVE and for predecessor projects that only partially make use of the described concepts and infrastructures. In this section, we will provide a short overview of these tools. Some tools will be described in more detail in Sect. 5.5 and Sect. 7.6.

CHEOPS Integrator

In the IMPROVE scenario (cf. Sect. 1.2), the tool CHEOPS [409] (cf. Subsect. 5.3.5) is used for the *simulation* of the *overall* chemical *process*. This simulation consists of multiple simulation models for different heterogeneous simulation tools. The task of CHEOPS is to perform partial simulations with the appropriate tools and to exchange simulation results between them.

An *integrator* was implemented to generate the *XML file* that is used to control CHEOPS for a specific process. This integrator differs from the ones described so far. Unlike other integrators, it deals with *more* than two *documents*. It unidirectionally generates the XML document out of multiple input documents: It reads the top-level PFD, the AHEAD product model (cf. Sect. 3.4) and the single simulation models. Additionally, links contained in integration documents between PFDs and the simulations are exploited. Another difference is the kind of user interaction taking place. There are *no*

conflicts as explained in Subsect. 3.2.4. Instead, the user has to *select* which simulation *model* to *use* if more than one is available for a part of the process and to supply initial values for external input streams.

Therefore, the integrator was *implemented manually* using only parts of the integrator framework. The rules that describe how the XML file has to be integrated are hard-coded into the prototype. This does not lead to problems here as the rules are simple and static. Additionally, *rule execution* is much *simpler* as in other integrators because of the lack of real conflicts. This integrator is described in more detail in Sect. 5.5.

Integrating Technical and Design Level

Most integrators integrate the results of technical activities in the development process, like PFDs and simulation models. Unlike that, the PFD-AHEAD integrator performs the *integration* of a *technical master document*, the PFD, with the administrative configuration of the development process in the AHEAD system (cf. Sect. 3.4) being some *organizational master document*.

As the PFD is a master document that serves as overview of the whole chemical plant to be designed, it can be used to provide an *interface* to the administration of the development process. The PFD-AHEAD integrator does this to help the *chief engineer* in a development process to *determine* the *consequences* of *changes* that are made to a part of the plant design. To do so, the chief engineer marks components in the PFD that have to be redesigned due to a change. After that, the integrator interactively *determines* how project *coordination* in AHEAD has to be adapted to contain tasks that deal with these changes. The *process manager* reviews the changes to the AHEAD process and either modifies them or directly applies them.

For this integrator as well, a *realization* approach *different* from the one for normal integrators has been applied. The main reason for this are the peculiarities of the user interaction needed: Instead of selecting between conflicting rule applications, the chief engineer *annotates* the *source document* (PFD) to make his decisions. Later, he interactively *refines* his *annotations* with the help of the integrator.

Additionally, user *interaction* is performed by two *different roles* in the development process. The first is the chief engineer, who uses a PFD-related user interface. The second is the project manager, whose user interface is closely related to the AHEAD system. As a result, the integrator was implemented manually. Nevertheless, the experience with other integrators was quite helpful as some concepts of the original approach could be applied resulting in a clean architecture and a straight-forward integration algorithm. This integrator is also described in more detail in Sect. 5.5.

Other Integrators within and Outside of IMPROVE

Some additional integrators have been built which are listed below. Due to space restrictions, they are explained very briefly only.

- An integrator collecting *data from production control* has been realized in an industrial cooperation with the German company Schwermetall [956]. It integrates a large number of heterogeneous data sources into a centralized database with the mapping being quite simple. This integrator is not based on the framework sketched above.
- Two *XML-based integration tools* have been built in the area of *process management*. The first translates AHEAD (cf. Sect. 3.4) process definitions into the petri net dialect used as workflow language by the commercial workflow management tool COSA [615]. The second handles the import of workflow definitions made with WOMS (cf. Sect. 2.4) into AHEAD. Both made use of the XML transformation language XSLT [602].
- During the first phase of IMPROVE, an integrator between *Aspen Plus* and the *flowsheet editor* (FBW) of IMPROVE (cf. Sect. 3.1.3) and one between the modeling tools *gPROMS* and *ModKit* have been implemented manually [84]. The experience gained with their implementation was important for the design of the integrator framework.
- After the first version of the framework was implemented, the integrator between FBW and Aspen Plus has been reimplemented to evaluate the framework.
- At our department, *integrator tools for other domains* have been built: In the ConDes project, an integration between a *conceptual* model of a building with the *concrete building architecture* is performed. Additionally, different ontologies modeling common knowledge about building architecture are integrated [234, 241].
  In the CHASID project, written *text* is integrated with a *graph structure* describing its *contents* [128].
  In the domain of *reverse-* and *reengineering*, triple graph grammars have been applied to integrate different *aspects* [81–83, 88, 89].
  The integration of different logical documents was first studied for development processes in *software engineering* [109, 260]. For instance, the relationship between requirements engineering and software architecture has been studied [74, 184, 185, 254]. Theses studies have been broadened during the IPSEN project [334] dealing with a tightly integrated development environment [229, 256–259].
- In our cooperation with innotec, we currently develop an integrator tool between the *data* structure *definition* of Comos PT and corresponding *UML models*.

3.2.7 Related Work

Our approach to the specification of incremental and interactive integrator tools is based on triple graph grammars. Therefore, we will discuss the relationships to other research on *triple graph grammars* in the next subsection. Subsequently, we will address competing approaches to the *specification* of integrator tools which do not rely on the triple graph grammar approach.

Related Work Based on Triple Graph Grammars

The triple graph grammar approach was invented in our group by Schürr [413], who gave the theoretical foundations for building TGG-based integrator tools. The work was motivated by *integration problems in software engineering* [349]. For example, [259] describes how triple graph grammars were applied in the IPSEN project [334], which dealt with integrated structure-oriented software development environments.

Lefering [255] built upon these theoretical foundations. He developed an early framework for building integrators which was based on triple graph grammars. The framework was implemented in C++, rules had to be transformed manually into C++ code to make them operational. The framework was applied to the integration of requirements engineering and software architecture documents, but also to integrate different views of requirements engineering [229].

Other applications of triple graph grammars have been built using the PROGRES environment. In our *reengineering* project REFORDI [88], synchronous triple rules were transformed manually into forward rules (for transforming the old system into a renovated one being based on object-oriented concepts). The PROGRES system was used to execute forward rules – which were presented as PROGRES productions – in an atomic way.

Our work on rule execution differs from systems such as REFORDI (or, e.g., VARLET [766] from another department) inasmuch as a single *triple rule* is executed in *multiple steps* to detect and resolve conflicts, as originally introduced by Lefering.

Our work contributes the following improvements:

- We added detection, persistent storage, and resolution of *conflicts* between integration rules.
- We provide a precise formal *specification* of the *integration algorithm*. In [255], the algorithm was described informally and implemented in a conventional programming language.
- Likewise, rules had to be hand-coded in Lefering's framework. In contrast, synchronous triple rules are *converted automatically* into specific rules for execution in our approach.
- We *used* the *specification* in *two ways*: First, IREEN was constructed by generating code from the formal specification (Fig. 3.41). Second, an implementation designed for industrial use was derived from the formal specification (Fig. 3.42).

To conclude this subsection, we will briefly discuss related work on triple graph grammars:

The PLCTools prototype [528] allows the *translation* between *different specification formalisms* for programmable controllers. The translation is inspired by the triple graph grammar approach [413] but is restricted to 1:n

mappings. The rule base is conflict-free, so there is no need for conflict detection and user interaction. It can be extended by user-defined rules which are restricted to be unambiguous 1:n mappings. Incremental transformations are not supported.

In [786], *triple* graph *grammars* are *generalized* to handle *integration* of *multiple documents* rather than pairs of documents. From a single synchronous rule, multiple rules are derived [787] in a way analogous to the original TGG approach as presented in [413]. The decomposition into multiple steps such as link creation, context check, and rule application is not considered.

In [579, 1033], a plug-in for *flexible* and *incremental consistency management* in Fujaba is presented. The plug-in is specified using story diagrams [670], which may be seen as the UML variant of graph rewrite rules. From a single triple rule, six rules for directed transformations and correspondence analysis are generated in a first step. In a second step, each rule is decomposed into three operations (responsibility check, inconsistency detection, and inconsistency repair). The underlying ideas are similar to our approach, but they are tailored towards a different kind of application. In particular, consistency management is performed in a reactive way after each user command. Thus, there is no global search for possible rule applications. Rather, modifications to the object structure raise events which immediately trigger consistency management actions.

Other Data Integration Approaches

Related areas of interest in computer science are (*in-*)*consistency checking* [975] and *model transformation*. Consistency checkers apply rules to detect inconsistencies between models which then can be resolved manually or by inconsistency repair rules. Model transformation deals with consistent translations between heterogeneous models. Our approach contains aspects of both areas but is more closely related to model transformation.

In [658], a *consistency* management approach for *different view points* [669] of development processes is presented. The formalism of distributed graph transformations [992] is used to model view points and their interrelations, especially consistency checks and repair actions. To the best of our knowledge, this approach works incrementally but does not support detection of conflicting rules and user interaction.

Model transformation recently gained increasing importance because of the model-driven approaches for software development like the *model-driven architecture* (MDA) [876]. In [689] and [776] some approaches are compared and requirements are proposed.

In [977], an approach for non-incremental and non-interactive *transformation between domain models* based on graph transformations is described. The main idea is to define multiple transformation steps using a specific meta model. Execution is controlled with the help of a visual language for specifying control and parameter flow between these steps.

In the *AToM project* [627], modeling tools are generated from descriptions of their meta models. Transformations between different formalisms can be defined using graph grammars. The transformations do not work incrementally but support user interaction. Unlike our approach, control of the transformation is contained in the user-defined graph grammars.

The *QVT Partner's proposal* [509] to the QVT RFP of the OMG [875] is a relational approach based on the UML and very similar to the work of Kent [498]. While Kent is using OCL constraints to define detailed rules, the QVT Partners propose a graphical definition of patterns and operational transformation rules. These rules operate in one direction only. Furthermore, incremental transformations and user interaction are not supported.

BOTL [565] is a transformation language based on UML object diagrams. Comparable to graph transformations, BOTL rules consist of an object diagram on the left-hand side and another one on the right-hand side, both describing patterns. Unlike graph transformations, the former one is matched in the source document and the latter one is created in the target document. The transformation process is neither incremental nor interactive. There are no conflicts due to very restrictive constraints for the rules.

Transformations between documents are urgently needed, not only in chemical engineering. They have to be incremental, interactive, and bidirectional. Additionally, transformation rules are most likely ambiguous. There are a lot of transformation approaches and consistency checkers with *repair actions* that can be used for transformation as well, but none of them fulfills all of these requirements. Especially, the detection of conflicts between ambiguous rules is not supported. We address these requirements with the integration algorithm described in this contribution.

3.2.8 Summary and Open Problems

In this section, we presented the results of the IMPROVE subproject B2. The *main contributions* of this section are the integration algorithm defined in the PROGRES specification of IREEN, the specification method for integration rules, and the integrator framework. Our framework-based integrator prototypes realized so far could be implemented with considerably lower effort than those that were built from scratch. The explicit specification of integration rules helped improving the quality of the resulting tools.

First practical experiences have been gained in a *cooperation* with our industrial partner innotec. The cooperation will be continued in a DFG transfer project, see Sect. 7.6.

Another important aspect of *methodolodical* integrator *construction* is the step-wise refinement of coarse-grained domain models or ontologies to fine-grained specifications defining the behavior of operational tools. In this section, this topic has only been sketched, focusing mostly on the fine-grained definition of integration rules. The relationship to domain models will be discussed in more detail in Sect. 6.3.

Besides evaluation in industry, current and future work will address some major *extensions* to the integration approach. For instance, more language constructs of graph transformations, e.g. paths and restrictions, are to be incorporated into the integration rule language. Additionally, the framework will be extended to offer repair actions for links that have become inconsistent due to modifications of documents. Further research will be conducted to support the integration of multiple documents considering complex multi-document dependencies.