Softw Syst Model (2005) 4: 123–140 / Digital Object Identifier (DOI) 10.1007/s10270-004-0071-0

# Model-based a-posteriori integration of engineering tools for incremental development processes

**Simon M. Becker, Thomas Haase, Bernhard Westfechtel**

Department of Computer Science III, University of Technology Aachen, Ahornstr. 55, 52074 Aachen, Germany
e-mail: {sbecker,thaase,bernhard}@i3.informatik.rwth-aachen.de

Received: 1 December 2003/Accepted: 28 June 2004
Published online: 10 November 2004 – © Springer-Verlag 2004

**Abstract.** A-posteriori integration of heterogeneous engineering tools supplied by different vendors constitutes a challenging task. In particular, this statement applies to incremental development processes where small changes have to be propagated – potentially bidirectionally – through a set of inter-dependent design documents which have to be kept consistent with each other. Responding to these challenges, we have developed an approach to tool integration which puts strong emphasis on software architecture and model-driven development. Starting from an abstract description of a software architecture, the architecture is gradually refined down to an implementation level. To integrate heterogeneous engineering tools, wrappers are constructed for abstracting from technical details and for providing homogenized data access. On top of these wrappers, incremental integration tools provide for inter-document consistency. These tools are based on graph models of the respective document classes and graph transformation rules for maintaining inter-document consistency. Altogether, the collection of support tools and the respective infrastructure considerably leverage the problem of composing a tightly integrated development environment from a set of heterogeneous engineering tools.

**Keywords:** A-posteriori integration – Incremental consistency management – Graph transformation – UML – Wrapping – Software architecture

## 1 Introduction

*Development processes* in different engineering disciplines are hard to support. Throughout the development process, a large number of *documents* are created which constitute the inputs and outputs of development tasks. These documents describe the product to be developed from different perspectives and at different levels of abstractions. They are connected by manifold dependencies and have to kept consistent with each other. In this respect, it has to be taken into account that development processes often are highly *incremental*: Rather than creating documents in a phase-oriented order, activities in different phases are performed in an intertwined way, implying that small changes have to be propagated back and forth between inter-dependent documents. While this constitutes a major challenge in its own, a further complication results from the fact that different documents may be processed by *heterogeneous tools* supplied by different vendors. *A-posteriori integration* of heterogeneous tools requires highly sophisticated modeling and implementation techniques in order to construct a development environment for incremental development processes with feasible effort.

In response to these challenges, we have developed an approach to tool integration which puts strong emphasis on *software architecture* and *model-driven development*. The term "approach" is not confined to the conceptual level, i.e., we have not merely defined concepts for tool integration. Rather, we have realized our approach by a collection of support tools and a respective tool infrastructure. In this way, we considerably leverage the problem of composing a tightly integrated development environment from a set of heterogeneous engineering tools. This is achieved through a *model-based tool construction process* which consists of the following steps:

1. *Architecture modeling and refinement* (Sect. 3). The software architecture of the integrated development environment to be constructed is modeled initially at a high level of abstraction. The initial architecture is refined gradually by means of architectural transformations which take care of technical details and introduce technical components such as tool wrappers required to make integration work. The transformation process results in a concrete architecture consisting of the components which need to be implemented (either manually or automatically).

2. *Modeling and construction of wrappers* (Sect. 4). In the case of a-posteriori integration, we have to deal with tools supplied by different vendors, using different data management systems, etc. To make use of these tools, we have to provide components which abstract from technical details and make them available at a conceptual level. These components, which are called *wrappers*, are constructed in a semi-automatic way with an interactive tool which supports the exploration of the interface of the development tool to be integrated.
3. *Construction of executable models for incremental consistency management* (Sect. 5). By providing tool wrappers, we decompose the problem of tool integration into manageable pieces. In the third (and final) step, we develop integration tools for incremental consistency management which make use of the wrappers' interfaces. Here, we follow a model-based approach: First, the documents to be integrated are described by corresponding document models. Subsequently, an integration model is constructed which defines correspondences between documents and rules for maintaining inter-document consistency. The integration model, which is based on graphs and graph transformations, is executable. In this way, we manage to construct integration tools with acceptable effort.

The rest of this paper is structured as follows: Section 2 provides an overview of our approach. Sections 3–5, which constitute the core part of this paper, are devoted to the steps of the model-based tool construction process. Section 6 discusses related work. Finally, Sect. 7 concludes this paper.

## 2 Overview

### 2.1 Context

Although our tool integration approach is generic (it may be applied in different engineering disciplines), it has been developed in the context of a research project which is concerned with a specific domain. The Collaborative Research center *IMPROVE* [43], a long-term research project carried out at Aachen University of Technology, deals with design processes in *chemical engineering*. The mission of this project is to develop models and tools for supporting design processes in chemical engineering, focusing on early phases (conceptual design and basic engineering).

Figure 1 illustrates the overall vision of IMPROVE with respect to tool support for the engineering design process[1]. As in other disciplines, in chemical engineering many tools are already available each of which supports a certain part of the overall design process. However, these tools are provided by different vendors and are not integrated with each other. This

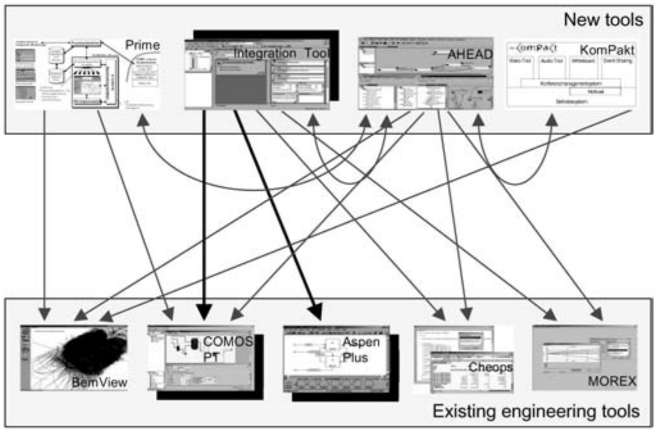

**Fig. 1.** Integrated tool environment

is illustrated by the bottom layer of Fig. 1, which shows several tools for designing and simulating chemical plants. These design tools are integrated into an overall environment for supporting engineering design processes. To this end, an infrastructure for tool integration is provided (not shown in the figure). Furthermore, new tools are developed which compose the existing tools and add further innovative functionality (top layer of Fig. 1).

In the sequel, we will focus on one type of tools of the upper layer: incremental integration tools for maintaining inter-document consistency. We will discuss the development of integration tools using a specific example: a-posteriori integration of a tool for editing flowsheets (Comos PT) with a tool for performing simulations (Aspen Plus).

### 2.2 Example

In chemical engineering, the *flowsheet* acts as a central document for describing the chemical process. The flowsheet is refined iteratively so that it eventually describes the chemical plant to be built. Simulations are performed in order to evaluate design alternatives. Simulation results are fed back to the flowsheet 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 them 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 flowsheet.

Figure 2 illustrates how an incremental integration tool assists in maintaining consistency between flowsheets and simulation models. The chemical process taken as example produces ethanol from ethen and water. Flowsheet and simulation model are shown above and below the dashed line, respectively. The integration document for connecting them contains links which are drawn on the dashed line. The figure illustrates a design process consisting of four steps:

[1] In the sequel, we prefer the term "design" to the term "development" when referring to chemical engineering because it is more appropriate in that context. On the other hand, we will continue to use the term "development" in the context of software engineering (development of tools for chemical engineers).

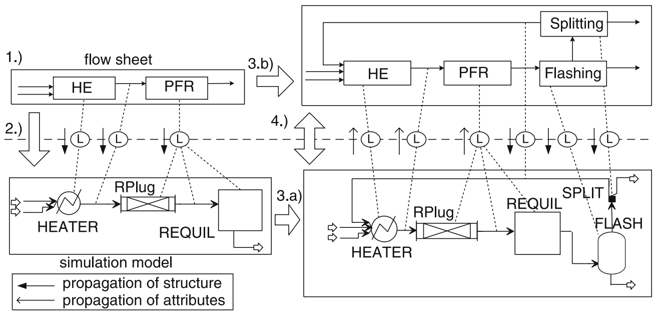

**Fig. 2.** Integration between flowsheet and simulation model

1. An initial flowsheet is created in Comos PT. This flowsheet 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 integration tool is used to transform the initial flowsheet into a simulation model for Aspen Plus. 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 sufficient, the user decides to map the PFR into a cascade of two blocks. These decisions are made by selecting among the different possibilities of rule applications the tool presents to the user.
3. The simulation is performed in Aspen Plus, resulting in a simulation model which is augmented with simulation results. In parallel, the flowsheet is extended with the chemical process steps that have not been specified so far (flashing and splitting).
4. Finally, the integration 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 flowsheet. Second, the extensions are propagated from the flowsheet to the simulation model. After these propagations have been performed, mutual consistency is re-established.

From this example, we may derive several features of the kinds of integration tools that we are addressing. Concerning the mode of operation, our focus lies on incremental integration tools rather than on tools which operate in a batch-wise fashion. Rather than transforming documents as a whole, incremental changes are propagated – in general in both directions – between inter-dependent documents. Often, the integration tool cannot operate automatically because the transformation process is non-deterministic. Then, the user has to resolve non-deterministic choices interactively. In general, the user also maintains control on the time of activation, i.e., the integration tool is invoked to re-establish consistency whenever appropriate. Finally, it should be noted that integration tools do not merely support transformations. In addition, they are used for analyzing inter-document consistency or browsing along the links between inter-dependent documents.

A-posteriori integration constitutes an important challenge being addressed by our approach. For example, it is crucial that chemical engineers may continue to use their favorite tools for flowsheet design and simulation. In our case study, we assume two wide-spread commercial tools, namely Comos PT (for flowsheet design) and Aspen Plus (for simulation). Both tools are fairly typical with respect to their technical features: Both maintain proprietary databases for storing flowsheet designs and simulation models, respectively. In addition, they both offer COM interfaces for tool integration. These interfaces allow to query the respective tool's functionality as well as to invoke operations to read and manipulate the tool's database, etc. Integration tools have to cooperate harmoniously with such existing tools, adding the "glue" which has been missing so far.

## *2.3 Tool development process*

In the following, we will sketch the approach that we have followed in order to solve the integration problems described above. Please note that we have actually implemented this approach, i.e., we have realized both the tools used for the tool development process and the integration tool between Comos PT and Aspen Plus [7, 28, 30].

### 2.3.1 Architecture modeling and refinement

The generic architecture of integration tools is sketched in an informal way in the grey region at the center of Fig. 3. Inter-document links are stored in integration documents separately from the native data structures used by the tools to be integrated. The integration tool is driven by rules which are created in a correspondency definition tool. The integrator core applies these rules, modifying the integration document as well as the documents to be integrated. The document integration process may be controlled through the user interface

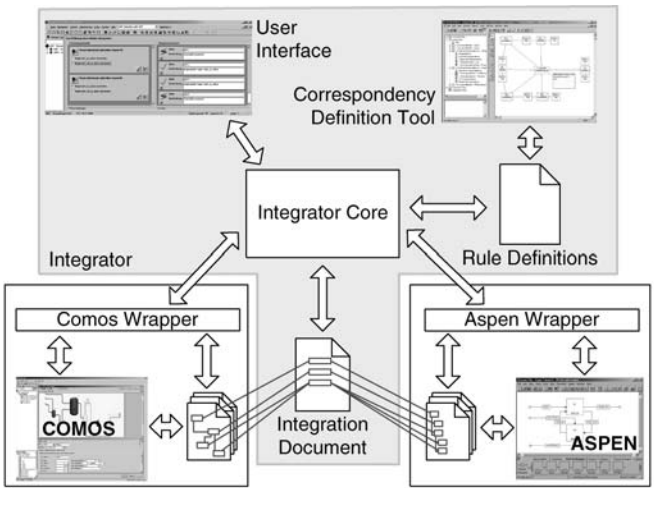

**Fig. 3.** Informal architecture of the integration tool

of the integration tool. Tools and their documents are accessed via wrappers which provide homogenized data access and abstract from technical details. As a consequence, the integration tool may focus on logical issues. Furthermore, the integrated tools may be replaced without affecting the integration tool provided that the wrapper interfaces need not be changed.

The first step of the tool integration process – architecture modeling and refinement – is concerned with embedding the integration tool to be developed into the architecture of the overall environment depicted in Fig. 1. Please note that we will focus in this paper on just these integration aspects. We will consider neither the architecture of the overall environment nor the architecture of integration tools in detail. Rather, we will investigate the "gluing parts" needed for performing the integration.

To this end, we essentially refine the wrapper components displayed in the informal architecture of Fig. 3. In fact, it turns out that the refinement results in a fairly sophisticated subsystem which is designed systematically by applying architectural transformations. Here, we distinguish between a *logical architecture* abstracting from technical details and a *concrete architecture* which realizes the logical architecture. Starting from a high-level simple architectural configuration, wrappers are introduced and decomposed, resulting in a refined logical architecture (Fig. 5). Subsequently, the logical architecture is further refined into a concrete architecture which eventually takes care of all of the details of the underlying technical infrastructure (Fig. 6).

### 2.3.2 Interactive modeling and construction of wrappers

In the previous step, the architecture is refined such that the problem of wrapper construction is decomposed into two levels. *Technical wrappers* are responsible for hiding the technical details of the interfaces provided by the tools. For example, the clients of technical wrappers are shielded from the underlying communication infrastructure such as COM or CORBA. This is illustrated in Fig. 7 for the wrapper of Aspen Plus, which provides a COM interface.

Apart from this abstraction, the operations provided by the tools are mapped 1:1 onto the interface of the technical wrapper. In contrast, the *homogenizer wrapper* located on top of the technical wrapper realizes the data abstraction required by the integration tool. In our running example, both flowsheets and simulation models can be mapped onto a common meta model, namely *process flow diagrams* (PFDs) [6]. The wrapper establishes a view which is based on the PFD meta model. For example, Aspen Plus documents are modeled in terms of ports and components (Fig. 8).

The method implementations for the homogenizer wrapper may be constructed in a semi-automatic way in the case of tools providing a COM interface (Figures 9–11). To this end, the implementer of the wrapper executes a method interactively through the COM interface. Method invocations are traced and visualized by sequence diagrams, which the implementer generalizes into method implementations.

### 2.3.3 Executable models for incremental consistency management

So far, we have considered the development of wrappers for accessing the tools to be integrated as well as their documents. Now, we address the adaptation of the integration tool, which makes use of the homogenizer wrappers. As illustrated in Fig. 3, the integration tool consists of a generic core which is driven by domain-specific rules. These rules are defined with the help of the UML. Executable rules are constructed as follows (Fig. 13):

1. The integration tool is based on a generic *meta model* which defines graph-based documents as well as the contents of integration documents. If required, the meta model may be extended to define base concepts for a specific domain. For the integration of COMOS PT and Aspen Plus, we have defined the PFD meta model as a common meta model. In this way, integration rules may be expressed in terms of this meta model. Moreover, the PFD meta model determines the interface of the homogenizer wrapper.
2. Documents are integrated on the type level by defining *link types* which relate types of *increments* being parts of the respective documents (Fig. 15). To define these link types, the type hierarchies of the related documents are retrieved through the homogenizer interface and are made available in the correspondency definition tool.
3. On the abstract instance level, *link templates* relate corresponding patterns of the related documents. Initially, link templates are modeled as object diagrams (static collaboration diagrams, Fig. 16). Subsequently, they are refined by adding dynamic information. In this way, *linking rules* are constructed which describe graph transformations by dynamic collaboration diagrams (Fig. 17).
4. Finally, linking rules are converted into an executable form. Then, a generic integration algorithm realized as part of the integrator core executes them (Fig. 18). The integration algorithm operates interactively: The user of the integration tool is provided with a set of applicable rules to resolve conflicts and non-determinism; unique rules are executed automatically to reduce user interactions.

### 2.3.4 Discussion

So far, we have described the architecture- and model-based integration tool development process in a simplified way as a sequence of three steps. However, the actual process may deviate from this simplified structure in the following ways:

*Reuse.* The development process need not be performed from scratch for each integration tool to be developed. Rather, results of previous processes may be reused. For example, the architectural patterns created through architectural modeling and refinement may be reused, in a potentially adapted way, from previous developments.

*Parallelism.* Some steps of the development process may be executed in parallel. For example, wrapper construction (Step 2) and development of executable integration rules

(Step 3) may proceed in parallel after the interfaces of the homogenizer wrappers have been negotiated and fixed.

## 3 Architecture modeling and refinement

### *3.1 Overview*

Ordinarily, the term *software architecture* is defined as a description of "the structure of the components of a program/system (and) their interrelationships" [26]. This description serves different purposes, among other things e.g. analyzing certain software qualities, such as adaptability, maintainability, or portability, or managing the software development process [5]. What this simple definition disregards, when developing a large, complex system as it was introduced in Sect. 2.1, more than one structural perspective together with the dependencies among them will be necessary with respect to the goals mentioned above. These structural views, for example, include a *conceptual*, a *development*, and a *process view* [15]. Therefore, "high-level"-diagrams as in Figs. 1 or 3 are helpful to get a first impression of the overall systems structure, but are not an adequate description of a software system in the sense of a software architecture.

In our modeling process, the architecture refinement process is not performed in an ad-hoc manner, rather it is controlled by domain-specific knowledge about a-posteriori integration. Therefore, we defined the relevant concepts in this area, like e.g. Application, Wrapper, Application Programming Interface, or Document, their relationships, and additional constraints and transformation rules by a graph-based (meta) model (see Fig. 4 for a cutout of the meta model concerning the logical architecture modeling part). This model again was implemented using a programmed graph rewriting system (PROGRES [62]). Using a framework for building graph-based interactive tools (UPGRADE [11]) we finally implemented an architecture modeling tool, called Fire3 (*Friendly Integration Refinement Environment* [29]).

In the following Sect. 3.2 we demonstrate how the coarse-grained "architecture" of the integration tool (see Fig. 3) is stepwise refined towards a detailed architecture description considering aspects like decomposing components, introducing wrappers, and distributing components via certain middleware-techniques. We call the first and the second aspect *logical architecture refinement*, respectively; the third aspect is called *concrete architecture refinement*.

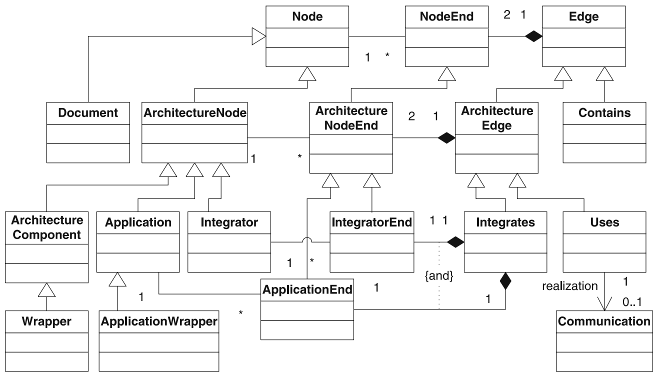

**Fig. 4.** Logical architecture meta model (cutout)

### *3.2 Stepwise architecture refinement*

#### 3.2.1 Logical architecture refinement

The system description sketched in Fig. 3 serves as starting point for the architecture refinement process. The structure described there is modeled in Box 1 of Fig. 5 with the concepts provided by the logical architecture meta model (see Fig. 4).

As the first refinement step the access to the application to be integrated by the integration tool is defined[2]. This can be done either by accessing the application via an API (*application programming interface*) (see Box 2a of Fig. 5) or, in the case no API is offered by the application, via the documents produced by the application (see Box 2b of Fig. 5). The latter refinement alternative is applicable e.g. when the application is equipped with an XML import and export function and only data integration is intended. Mixtures of alternative 2a and 2b are possible, as well (not shown in Fig. 5): if the API, for example, is a read-only interface, as in earlier releases of Aspen Plus, read access is realized via the API, while for write access the document solution is used.

Choosing alternative 2a leads to the model shown in Box 2a of Fig. 5: the ≪Application≫ Aspen Plus is extended with an additional ≪ArchitectureComponent≫ representing the API, that is used by the Comos_Aspen API Accessor, which is an ≪ArchitectureComponent≫ of the Integrator. This ≪Uses≫ relation between the Comos_Aspen API Accessor and the Aspen API is subsequently refined in step 3 and 4: An ≪ApplicationWrapper≫ is introduced (see Box 3 of Fig. 5) which is subdivided into a so-called *homogenizer wrapper* (AspenHomWrapper) and a *technical wrapper* (AspenTecWrapper) (see Box 4 of Fig. 5). This is done for the following reasons: The integration tool expects the tool's data to be provided as attributed, directed, node- and edge-labeled graphs (see Fig. 13 and Fig. 14). Therefore, the proprietary data model provided by the tool's API has to be transformed by the homogenizer wrapper into the graph model. In this context the technical wrapper offers the homogenizer wrapper a location- and implementation-independent access to the tool's API. How the homogenizer and the technical wrapper can be further refined, will be explained in Sect. 4.

Please note that for the refinement steps shown in Box 2a (or alternatively in Box 2b) user interactions are necessary: it is the software engineer's knowledge to decide how the ≪Application≫ is accessed by the ≪Integrator≫. After defining this, the transformations shown in Box 3 and Box 4 are performed by the architecture modeling tool automatically. When, for example, the software engineer decides later

[2] For the following explanations we will focus on the right ≪Integrates≫ relation between Comos_Aspen and Aspen Plus. The left ≪Integrates≫ relation can be refined analogously.

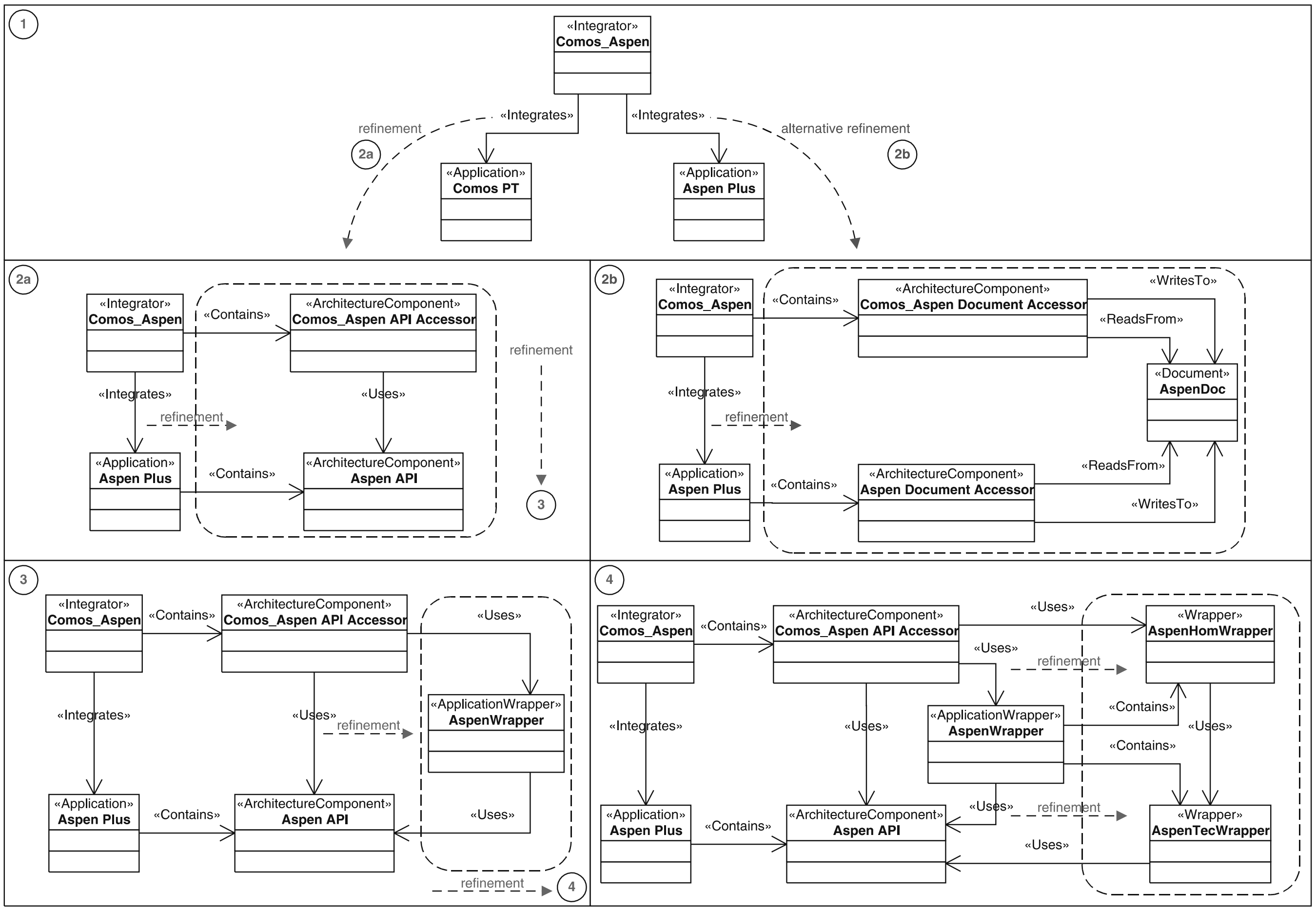
«Integrator» Comos_Aspen
«Integrates»
«Application» Comos PT
«Application» Aspen Plus
refinement
alternative refinement
«ArchitectureComponent» Comos_Aspen API Accessor
«ArchitectureComponent» Aspen API
«Contains»
«Uses»
«ArchitectureComponent» Comos_Aspen Document Accessor
«ArchitectureComponent» Aspen Document Accessor
«Document» AspenDoc
«ReadsFrom»
«WritesTo»
«ApplicationWrapper» AspenWrapper
«Wrapper» AspenHomWrapper
«Wrapper» AspenTecWrapper

**Fig. 5.** Logical architecture refinement

that no homogenizer wrapper is necessary, he can delete this component manually.

### 3.2.2 Concrete architecture refinement

So far we have specified the logical architecture of our system. The next step is to define the concrete architecture. Therefore, the logical structure is transformed in an isomorphic, concrete one (see Box 5 of Fig. 6): instances of the classes Application and Integrator, respectively instances of subclasses, are transformed into instances of class ProcessNode. They represent a process in the operating system sense. Instances of class ArchitectureComponent, respectively instances of subclasses, are transformed into instances of class ComponentImplementation, respectively into instances of corresponding subclasses. While the ≪Contains≫ relations are kept, the ≪Uses≫ relations are as well transformed into equivalent ≪MethodInvocation≫ or ≪Interprocess_Call≫ relations. A ≪MethodInvocation≫ represents a local communication, an ≪Interprocess_Call≫ represents a distributed one. The architecture modeling tool again carries out these transformations automatically.

Specifying how the ≪Interprocess_Call≫ relations will be implemented are the final steps of architecture refinement. Because the Aspen API is implemented by the tool's vendor using COM (*Component Object Model* [40]), the ≪Interprocess_Call≫ relation between the AspenTecWrapper and the Aspen API is simply refined into a ≪COM_Call≫ and additionally the Aspen API is transformed into an instance of class ≪COM_Interface≫ (see Box 5 of Fig. 6). In the case of the ≪Interprocess_Call≫ relation between the Comos_Aspen API Accessor and the AspenHomWrapper, different alternatives are possible. Realizing the AspenWrapper as an independent operating system process is one alternative[3]. The interprocess communication between the ≪Integrator≫ and the ≪Wrapper≫ can be implemented e.g. using CORBA (*Common Object Request Broker Architecture* [47]). This alternative offers the opportunity to distribute the ≪Integrator≫ and the ≪Application≫ to be integrated over various nodes in a computing network. If the software engineer decides so, the architecture is refined as shown in Box 6a of Fig. 6. If no distributed solution is desired, the independent ≪ProcessNode≫ AspenWrapper is resolved, i.e. the AspenWrapper is deleted, the AspenHomWrapper and the AspenTecWrapper are realized as components of the integration tool, and the communication between them is refined to a local ≪MethodInvocation≫ (see Box 6b of Fig. 6).

[3] This alternative was already suggested by the initial transformation of the logical into the concrete architecture.

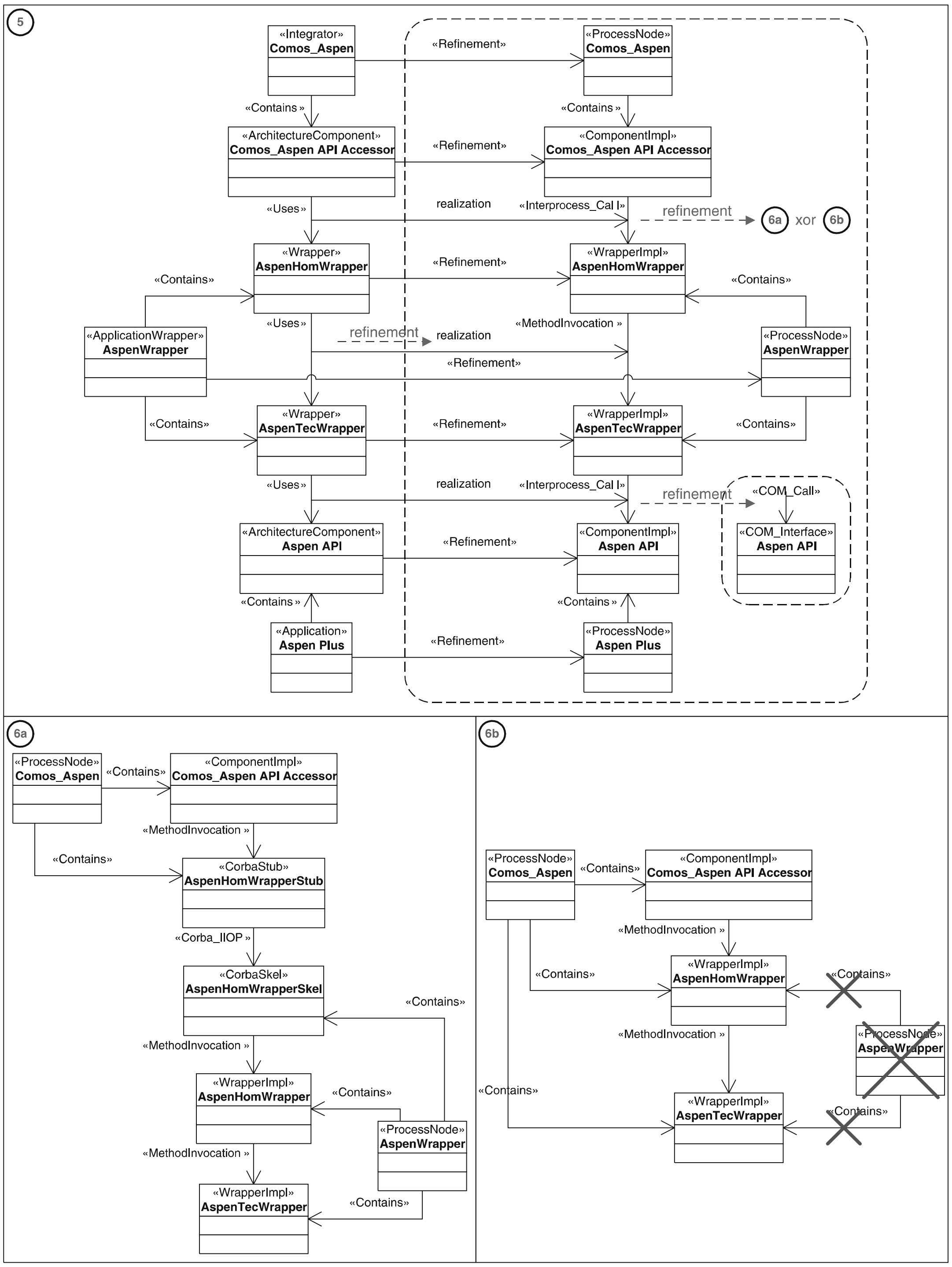
5
«Integrator» Comos_Aspen
«Refinement»
«ProcessNode» Comos_Aspen
«Contains»
«ArchitectureComponent» Comos_Aspen API Accessor
«ComponentImpl» Comos_Aspen API Accessor
«Uses»
realization
«Interprocess_Call»
refinement
6a xor 6b
«Wrapper» AspenHomWrapper
«WrapperImpl» AspenHomWrapper
«ApplicationWrapper» AspenWrapper
«ProcessNode» AspenWrapper
«MethodInvocation»
«Wrapper» AspenTecWrapper
«WrapperImpl» AspenTecWrapper
«COM_Call»
«COM_Interface» Aspen API
«ArchitectureComponent» Aspen API
«ComponentImpl» Aspen API
«Application» Aspen Plus
«ProcessNode» Aspen Plus
6a
«CorbaStub» AspenHomWrapperStub
«Corba_IIOP»
«CorbaSkel» AspenHomWrapperSkel
6b

**Fig. 6.** Concrete architecture refinement

## 4 Interactive modeling and construction of wrappers

### *4.1 Overview*

A wrapper acts as an *adapter* "convert(ing) the interface of a class into another interface clients expect" [23]. Therefore, application of wrappers enables the reuse of existing software in a new context [63].

Developing a wrapper includes several tasks: the syntax and the semantics of the given tool's interface to be wrapped, the *source interface*, as well as of the interface required by the client, the *target interface*, have to be specified. Furthermore, the transformation of the source into the target interface has to be defined. According to these tasks a wrapper is not a monolithic component, rather it is a subsystem consisting of several subcomponents. Our architecture takes this into consideration by dividing a wrapper into a technical wrapper, realizing the access to the source interface, and a homogenizer wrapper, realizing the target interface (see Box 4 of Fig. 5).

In Sect. 4.2 we present the modeling of the technical and the homogenizer wrappers' interfaces and an example of our approach to interactively specify the transformation of the source into the target interface.

## *4.2 Interface modeling and interactive exploration*

As described in the previous section, in our scenario Aspen Plus, the tool to be integrated, contains an API implemented using COM. The syntax of such a COM interface is documented in a standardized way through a so-called *type library*. A type library contains a static description of the classes and their attributes and operations offered by the interface in form of signatures. By simply parsing this type library our architecture can automatically be refined as shown in Fig. 7[4]. This class diagram defines the internal data model of Aspen Plus.

Figure 2 shows an example of an Aspen Plus simulation model. As this example illustrates, a simulation model consists of several components, e.g. *blocks*, representing chemical devices, *streams*, connecting blocks, or *ports*, modeling the connection between a block and a stream. Each of these components has its own type, for example a block represents a *heater* or *reactor*.

According to the requirements of the integration tool, the internal data model of Aspen Plus has to be transformed into a graph model in conformity with the graph layer in Fig. 14. Therefore, an Aspen Plus simulation model component is modeled as a Node containing an attribute meta type, which is the component's meta type, e.g. block, stream or port, and an attribute type, which is the component's concrete type, e.g., in the case of a block, heater or reactor (see Fig. 8)[5]. The necessary operations to access instances of this model are offered by the export interface of the homogenizer wrapper. An example for such an operation is shown in Fig. 8.

So far, we have modeled the internal, static data structures of the technical and the homogenizer wrapper. Now we have to define the transformation between these two structures. This is usually done by implementing it manually. To support the software engineer carrying out this implementation, we have developed a tool allowing the exploration of the dynamic behavior of an application offering a COM interface at runtime interactively. Using this tool, parts of the code to be implemented can be generated automatically.

[4] The figure shows a simplified model of the COM interface. The COM interface of Comos PT, for example, consists of 101 classes with a total of 7680 methods.

[5] For the following example we use this simplified model of the real model. It is simplified in the sense that the relations between the nodes are not considered.

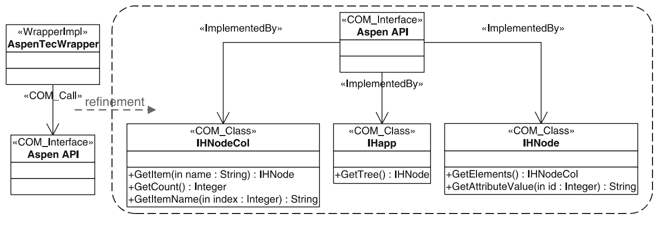
«WrapperImpl» AspenTecWrapper
«COM_Call»
refinement
«COM_Interface» Aspen API
«ImplementedBy»
«COM_Interface» Aspen API
«ImplementedBy»
«ImplementedBy»
«COM_Class» IHNodeCol
+GetItem(in name : String) : IHNode
+GetCount() : Integer
+GetItemName(in index : Integer) : String
«COM_Class» IHapp
+GetTree() : IHNode
«COM_Class» IHNode
+GetElements() : IHNodeCol
+GetAttributeValue(in id : Integer) : String

**Fig. 7.** Technical wrapper refinement

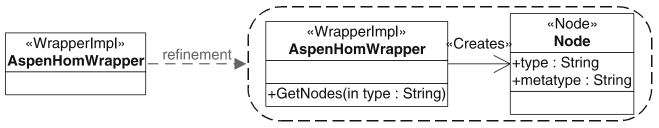
«WrapperImpl» AspenHomWrapper
refinement
«WrapperImpl» AspenHomWrapper
+GetNodes(in type : String)
«Creates»
«Node» Node
+type : String
+metatype : String

**Fig. 8.** Homogenizer wrapper refinement

The following example illustrates how the internal data model of the homogenizer wrapper is materialized. Therefore, specifications for the internal operations of the homogenizer wrapper realizing this materialization are created with the help of our tool in the following way: Firstly, the tool generates specifications for basic operations by tracing user interactions. Subsequently, these operation specifications are generalized and extended[6].

After parsing the type library of a COM interface, the tool starts the underlying application as an operating system process via a generic start operation that every COM interface implements. The return value of this operation is a reference to the initial object of the COM interface. The tool determines the object's class and, based upon the static information parsed out of the type library, a GUI (*graphical user interface*) for the given COM object is generated by the tool. Using this GUI, the software engineer can query the values of object attributes or can invoke the object's methods.

An attribute value or a return value of a method can be either an atomic value, like a string or an integer, or a reference to another COM object. In this case another GUI according to the referenced object is generated allowing to inspect the referenced object. In this way the software engineer can explore the COM interface of an application interactively.

Furthermore, the user's interactions are traced by the tool and can be visualized as UML sequence diagrams. The trace shown in Fig. 9, for example, illustrates how to access the collection of blocks included in an Aspen Plus simulation model using the operations offered by the tool's API. This trace serves as a specification for an internal operation called GetAspenBlocks of the homogenizer wrapper[7].

By composing such traces we get further specifications for internal operations, e.g. for retrieving the number of blocks included in the simulation model (see the left sequence diagram of Fig. 10) or for accessing the (concrete) type of a single block (see the right sequence diagram of Fig. 10)[8].

[6] At the moment the latter has to be done by the software engineer manually. Extending our tool in this way is our current work.

[7] With respect to our previous explanations the homogenizer wrapper communicates with the COM objects via the technical wrapper, which is not shown to keep the figure simple.

[8] These UML sequence diagrams as well as their composition are generated by the tool automatically.

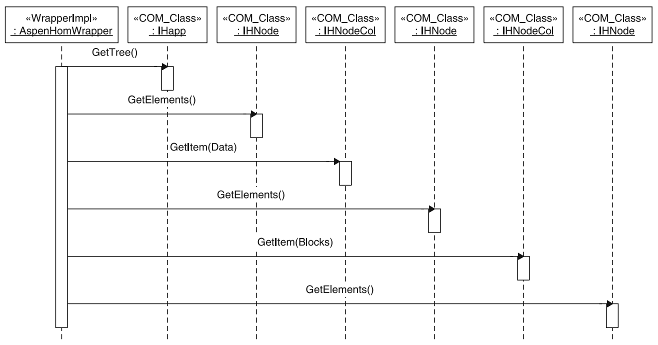

**Fig. 9.** Specification of operation GetAspenBlocks

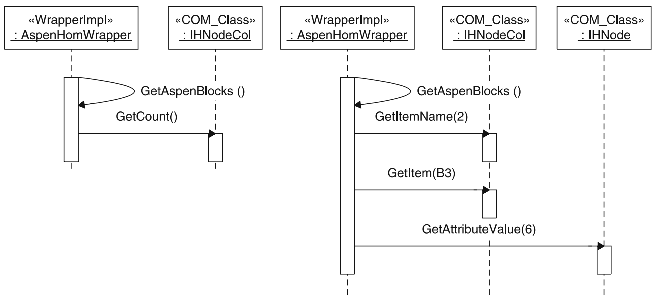

**Fig. 10.** Specification of operation NumberOfBlocks (*left sequence diagram*) and of operation GetAspenBlockType (*right sequence diagram*)

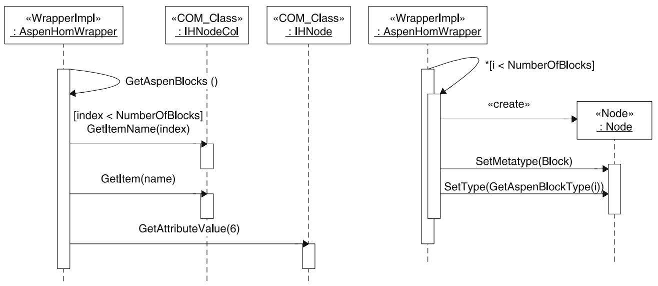

**Fig. 11.** Generalization of operation GetAspenBlockType (*left sequence diagram*) and specification of operation CreateBlockNodes (*right sequence diagram*)

By substituting the constant 2 with a parameter and specifying the range of the parameter (the range is given through the specification of the operation NumberOfBlocks) we get a generalized specification for accessing the (concrete) type of any block included in an Aspen Plus simulation model (see the left sequence diagram of Fig. 11). In contrast to the traced operation GetAspenBlockType():String, the generalized operation has the signature GetAspenBlockType(index:-Integer):String. Based upon this operation, we can finally specify an operation (see the right sequence diagram of Fig. 11) that instantiates the Node classes. This operation materializes the internal data model of the homogenizer wrapper regarding the blocks in an Aspen Plus simulation model. Analogously, operations to handle streams and ports can be specified.

## 5 Executable models for incremental consistency mangement

### 5.1 Overview

Whereas in Sect. 3 the overall architecture of an integrated system was discussed, in this section we describe how the consistency management between existing tools is performed in detail. In Fig. 5 a class Comos_Aspen was contained in the coarse-grained architecture of the system, connecting Comos PT and Aspen Plus. This class is a placeholder for an incremental integration tool, which in general supports incremental transformation and change propagation, browsing, and consistency check between dependent documents. In our running example, the integration tool in question supports the consistency management between flowsheets in Comos PT and simulation models in Aspen Plus with focus on the bidirectional, incremental transformation and change propagation between the two documents.

Unlike other approaches to rule-based consistency management which first check for inconsistencies and then apply inconsistency resolving rules, our approach is transformation-centered: New elements and changes of existing ones in one document are detected and propagated to the other one.

The propagation works rule based, with the rules being defined in a special modeling environment called rule editor. The set of rules controlling an integration tool in general is neither complete nor unambiguous w.r.t. the documents that are to be integrated. As a result of that integration tools do not work batch-wise but rely on two different kinds of user interaction: First, if a rule is missing for a given situation, transformation can be performed manually. Second, if multiple rules are applicable and their execution is conflicting, one of the rules has to be chosen for execution by the user.

The main idea behind the realization of our consistency management approach is to keep track of the fine-grained inter-dependencies between the contents of dependent documents. The resulting inter-document relationships are explicitly stored, which is an essential prerequisite for incremental change propagation between the documents. This is done in an additional document which is called *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. The integration is controlled by rules: One link is created by the execution of one integration rule. If a link has to be further structured, this can be done 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.[9]

Figure 12 shows the structure of links in a UML class diagram. Most constraints needed for a detailed definition are omitted, only examples are shown. An increment can have

[9] The usage of sublinks could be avoided by creating additional links by additional rules but this often leads to integration rules that are hard to understand.

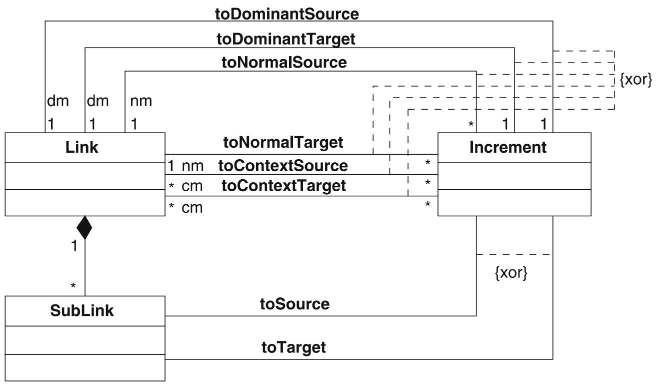

**Fig. 12.** Structure of links

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 be referenced by 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 created by an existing link, for instance to embed newly created edges. Owned increments can be further divided into *dominant* and *normal* increments. *Dominant* increments play a special role in the execution of integration rules (see Sect. 5.3). Each link can have at most one dominant increment in each document. A link can relate an arbitrary number of normal increments.

Please note that there is additional information stored at a link like its state and information about possible rule applications. This information is needed by the integration algorithm but not for the definition of integration rules.

The documents' contents are provided as attributed, directed, node- and edge-labeled graphs by the tool wrappers introduced in Sect. 4. The integration document forms an additional graph with inter-graph edges leading to the nodes in the other documents. To simplify the integration, it is performed on the union of all single graph documents and the inter-graph edges (otherwise, distributed graph transformations [65] would have to be used). The execution of integration rules dealing with the graph structure is done by using graph transformations. Rule definition and execution is inspired by the triple graph approach [37, 60] which was modified to fit the practical requirements in our domain of application [9]. Our approach supports transformation and consistency check of node attributes as well, but in this paper, we only present the structural part of the integration.

If an integration tool is started for the first time for two documents, a new integration document is created. Then, all integration rules that are unambiguously applicable are applied to the documents, modifying them and creating new links in the integration document. After that, a list of all rules that are applicable but conflicting to other rules is presented to the user, who has to select a rule that is to be executed. Again, unambiguously applicable rules are applied and decisions are made by the user until there are neither decisions nor unambiguous rules. Now, the user can manually complete the integration if the result is not satisfying, yet.

When the integration tool is activated again, each existing link in the integration document is checked for consistency first. I.e. the referenced increments are checked for modifications that violate the rule that created this link. Then, inconsistency resolving rules are applied to all inconsistent links, which can lead to user interaction, too. Next, integration rules are applied to the new increments which are not referenced by a link, yet, as when started for the first time.

In this paper, we will focus on the presentation of the definition and enactment of integration rules. A detailed description of the realization of the integration tool is provided in [7]. The rest of this section is structured as follows: In Sect. 5.2 our approach for the definition of integration rules using UML is presented, and Sect. 5.3 gives an overview of how integration rules can be executed using a graph transformation system.

## *5.2 UML-based modeling of integration rules*

For the definition of integration rules, we follow a multi-layered approach as postulated by OMG's meta object facility (MOF) [49]. Figure 13 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 by additional elements that form a language to define models of the documents to be integrated and to express all aspects concerning the documents' integration. The meta model itself is layered, too, which will be explained in detail below.

On the model level, we distinguish between a type (or class) level and an instance level, like standard UML does.

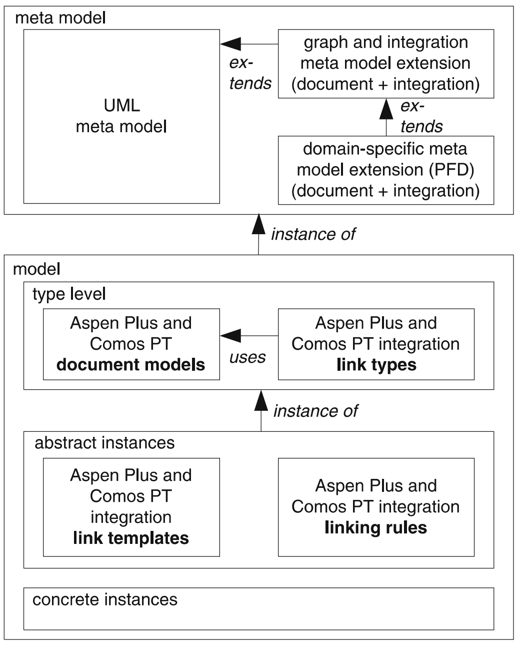
meta model
UML meta model
extends
graph and integration meta model extension (document + integration)
extends
domain-specific meta model extension (PFD) (document + integration)
instance of
model
type level
Aspen Plus and Comos PT document models
uses
Aspen Plus and Comos PT integration link types
instance of
abstract instances
Aspen Plus and Comos PT integration link templates
Aspen Plus and Comos PT integration linking rules
concrete instances

**Fig. 13.** Levels of modeling

On the type level, class hierarchies describing the different types of documents and *link types* constraining which increments can be related by a link are defined using UML class diagrams. The instance level is divided into an abstract and a concrete level. On 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. Link templates can be annotated to define linking rules. The annotations provide information about which objects in the patterns are to be matched against existing objects in concrete documents and which have to be created, comparable to graph transformations. On the concrete instance level, existing documents and integration documents can be modeled, which is not further described here. In this paper we will give a coarse overview of our modeling approach only, for a detailed description including more examples, the reader is referred to [10][10].

The modeling process to define a concrete integration tool is enacted as follows: First, if needed, specific meta model extensions for the types of documents to be integrated have to be created (see Sect. 5.2.1). Next, document models and link types have to be defined. To define the meta model extensions and link types on type level, domain knowledge like contained in the conceptual data model for chemical engineering CLiP [6] is required. The document models can be generated with the help of the tool wrappers described in Sect. 4. Now, the integration tool could be applied in the domain of application, because link templates and integration rules can be defined on the fly [10]. Nevertheless, a basic set of integration rules should be defined first to keep the additional effort low for the user. Therefore, link templates are to be modeled, which are consistency checked against the link types. From each link template a set of standard linking rules is automatically derived. All rules are stored in a rule base and finally, are interpreted by the integration tool at runtime, using the algorithm described in Sect. 5.3.

### 5.2.1 Meta model

To extend the UML meta model, we make use of restrictive meta model extension as described in [59]. The meta classes introduced by our meta model extension are referred to by stereotypes on the model level. Figure 14 depicts an excerpt of the meta model underlying our modeling formalism that shows its layered structure. It is presented using MOF, but a lot of details are omitted. The top layer contains the standard UML meta model which is extended by the next lower layer to a meta model for attributed, directed, node- and edge-labeled graphs. Replaces-constraints, in the figure depicted as dashed arrows marked with {r}, ensure that the

[10] Please note that we changed our terminology: In former publications we used *association* instead of *link type* and *correspondency* instead of *link template*.

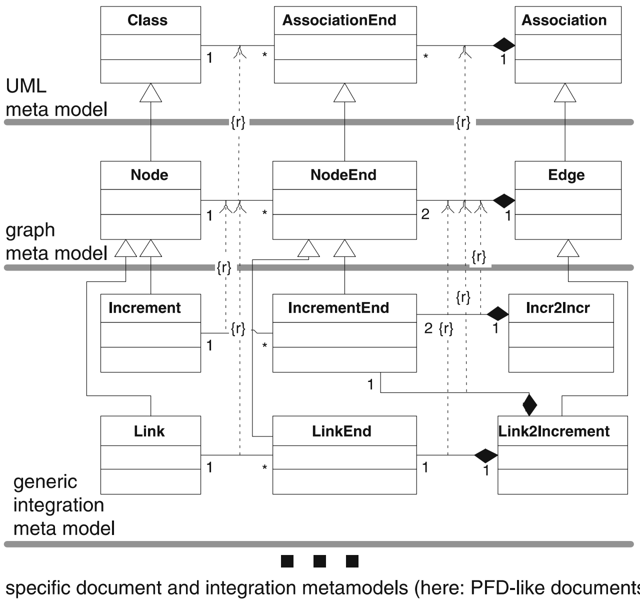

**Fig. 14.** Extension of the UML meta model

model elements of different layers cannot be used together in an unwanted fashion, e. g. resulting in an Edge connecting a Node and a UML Class.

The layer below consists of a generic integration meta model with all elements being subclasses of the elements from the graph layer. As a result of that, documents that are to be integrated and integration documents can both be regarded as being graphs, and they can be dealt with using graph techniques like graph transformations. The tool wrappers described in Sect. 4 provide us with access to the documents' contents via an interface that conforms to the graph layer of the meta model during the integration process. The constraints and multiplicities in Fig. 12 concerning links and their associations to increments are contained in the meta model but not explicitly visible because they have to be defined as constraints on the meta model restricting the associations' cardinalities on the type level.

The meta model layers presented so far are fixed for all integration models and tools. That is, our implementation of the integration framework and the modeling formalism relies on this model. If needed, on the layers below extensions for specific types of documents can be made, which are interpreted by all our tools.

In our running example, we deal with Comos PT flowsheets and Aspen Plus simulation models. The overall structure of both types of documents is rather similar, because both describe technical systems consisting of pieces of equipment that are connected by via connection ports. We call these types of documents PFD-like (process flow diagram like). Therefore, we defined a common meta model for PFD-like documents, to make the modeling of rules more user-friendly. The definition of the meta model for chemical engineering is based on the conceptual data model CLiP [6]. Here, both documents to be integrated are instances of PDF-like documents but in general, our approach supports the integration of documents being instances of different specific meta models. The PFD-like metamodel is provided by the tool wrappers when importing the tools' document models into the rule editor model.

### 5.2.2 Type level

Detailed document models are specified on the type level using UML class diagrams. Here, the increment types and their intra-document relations are defined. Increment types are modeled as classes in the documents' class hierarchies, their intra-document relations are modeled as associations. Increments can be attributed but in this paper we only deal with the structural aspects of the integration. Most parts of the document models can be directly imported from the tools' internal type systems using the tool wrappers.

Using the increment types from the detailed document models, link types are defined. Because all link templates have to be instances of link types, link types constrain the types of increments that can be related by a link template. They also define whether a link template can be further structured by sublinks and which types of increments can be related by them. Figure 15 shows an example of a link type definition. Increments and links are modeled as classes and their inter-relations as associations. Instances of the new link type StreamLink can relate one stream in a Comos PT flow sheet (an increment of type PhaseSystem) with one stream in an Aspen Plus simulation model (an increment of type MaterialStream). The stream increments are the dominant increments in both documents for the StreamLink. Additionally, up to two ports connected to the stream in each document can be referenced by the link as normal increments. In most flowsheets and simulation models, streams have one input and one output port, where other pieces of equipment can be connected. To map the corresponding ports on both sides of the link, up to two sub links of type StreamPortMapping can be added to the StreamLink. The sample link type definition described here is rather concrete because it is specifically tailored to relate streams. Other link types can be defined that are more generic. For instance, in [10] a link type is presented that allows to link any pattern in a flowsheet to any other pattern in a simulation model.

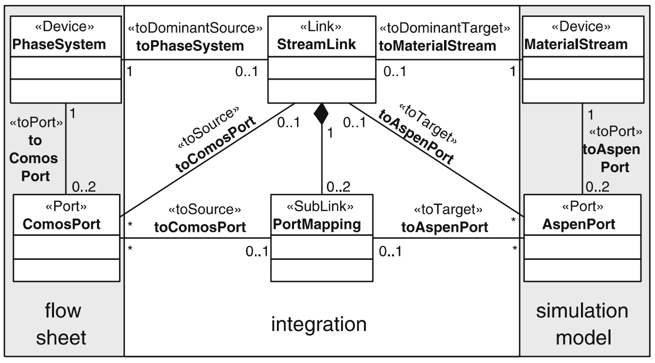

**Fig. 15.** Link type definition

### 5.2.3 Abstract instance level

Link templates relate corresponding patterns of the documents. A pattern is an abstract description of a situation that may occur in a concrete document. From a semantic point of view, link templates describe that if the patterns are present in the documents they may be related by a link. There is no information about how this link is established or whether parts of the patterns are generated. Link templates are modeled as (static) UML collaboration diagrams containing instances of link and sublink types that are connected via UML links with instances of increment types. Additional constraints can be defined concerning attributes and their values.

Figure 16 depicts a link template if the constraints are not taken into account. This link template is an instance of the link type in Fig. 15. A StreamLink relates a PhaseSystem and a Material Stream and their input and output ports. Two sublinks map the input and the output ports, which is needed for handling connections between ports during the transformation of documents as explained below. Please note that all object names are placeholders and no concrete identifiers belonging to existing documents. The instances' stereotypes referencing the meta model elements are not explicitly shown in the figure, but they can be derived from the instances abstractions on type level.

Link templates are purely declarative, but they can be extended to executable linking rules. Therefore, UML constraints are added, for example {new} marks an object of the template that currently is not present and has to be created. Other constraints are {not} for objects that are not present and {delete} for objects that have to be present and are deleted. By applying these constraints to the objects of a link template, a graph transformation rule is created. The execution of linking rules is presented in the next subsection.

Comparable to the triple graph grammar approach, different linking rules can be automatically derived from a link template. A *forward transformation rule* finds an increment structure in the source document (here: the flowsheet) and creates a corresponding structure in the target document (here: the simulation model). Both structures are then related

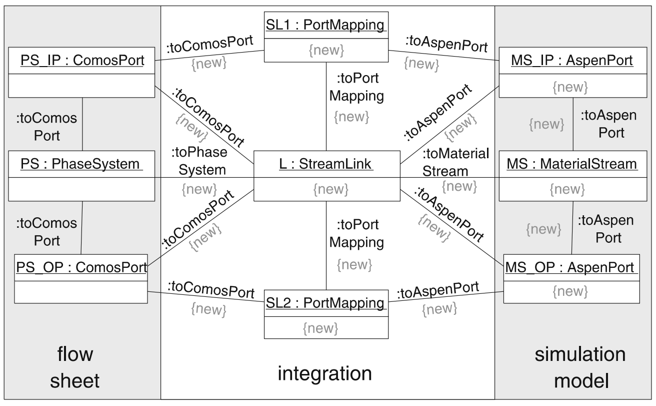

**Fig. 16.** Link template (without constraints printed in gray), linking rule (with constraints)

by a link. To derive a forward transformation rule, all increments, links, and sublinks are marked with the constraint {new}, except the context increments of all documents and the owned increments in the flowsheet document. Association instances that have a least one end outside the flowsheet document are marked with {new}, too. The linking rule in Fig. 16 is the resulting forward transformation rule. Please note that in this example link template there are no context increments. A *backward transformation rule* and other rules can be derived similarly. For special needs, link templates can be manually extended to rules.

Another linking rule is presented in Fig. 17. Please note that the corresponding link type definition is not shown. In flowsheets and simulation models, pieces of equipment are connected via their ports by special increments called connection. The depicted rule is used to ensure that, if two substructures of a source document are connected and are transformed to a target document, the resulting substructures are connected according to the original topology. It is a forward transformation rule propagating a connection in the flowsheet to the simulation model. The rules that transformed the two substructures already established sublinks to map the corresponding ports of the source and the target structures. The rule in Fig. 17 references the ports and the sublinks that map them as context (dashed lines). With the help of this information the rule is able to locate the ports in the target document that have to be connected by creating a new connection. The new link references the connections in both documents as dominant increment and the ports and sublinks as context increments.

The integration rules presented so far reference rather similar structures of the two documents. In general, rules can relate arbitrary substructures of the documents. For instance, the rule used in the running example (Sect. 2) in step 2 to transform the reactor relates a reactor in the flowsheet to a cascade of two reactors in the simulation model. Beside linking rules, there are inconsistency repair rules that can be applied after an existing link has become inconsistent because of modifications in the documents. Another aspect of integration not presented in this paper is the transformation of attribute values. This is done executing special scripts defined for each link template [7].

### 5.3 Execution of integration rules

In this subsection an overview of our approach for the execution of integration rules is given. A detailed description is provided in [8].

Although the integration rules presented so far can be interpreted as graph transformations, they *cannot* be executed straightforward by simply translating each rule to one graph transformation rule. Instead, the integration rules are executed by a complex algorithm consisting of rule-independent and rule-specific steps. This is necessary for several reasons:

– The sequence of rule applications has to be determined.
– Different rule applications may be possible for a given set of increments.
– The sets of increments used by two rule applications may intersect.
– To resolve conflicts and ambiguities, user interaction is necessary.

Figure 18 shows the integration algorithm for new links. As stated in Sect. 5.1, existing links are dealt with in a different phase of the overall integration algorithm which is

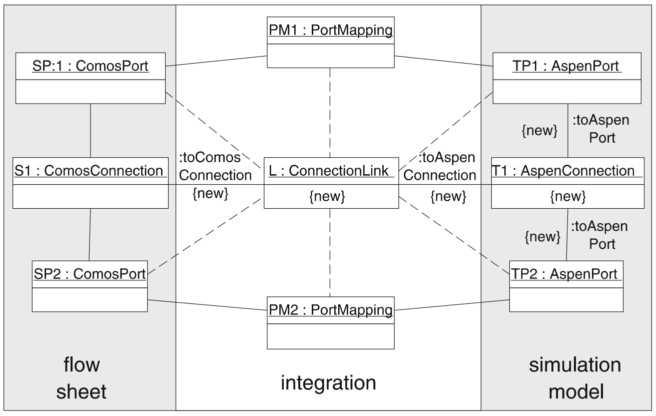

**Fig. 17.** Linking rule (forward) transforming a connection

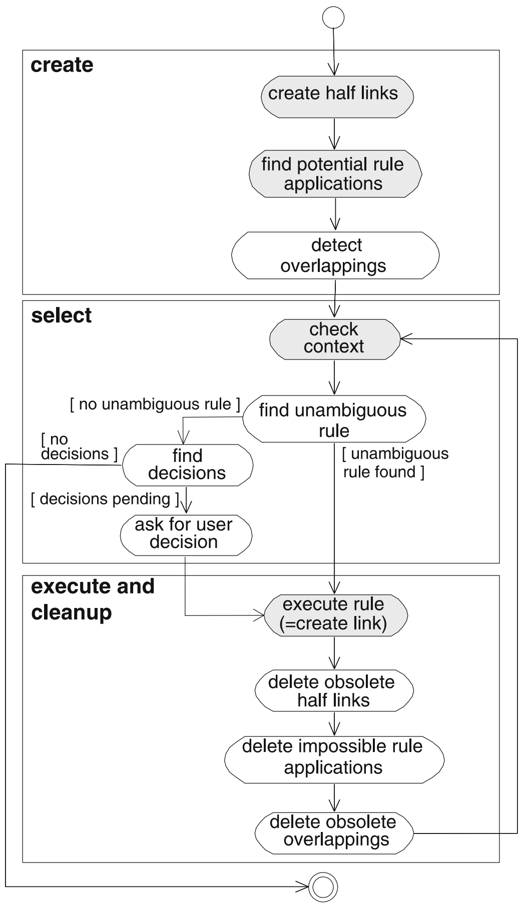

**Fig. 18.** The integration algorithm

not presented here. The algorithm is presented using a UML activity diagram. The activities are informally grouped into three phases by rectangles. Activities that are rule-specific are filled gray whereas rule-independent activities are filled white. The rule-specific activities are derived from the specification of the linking rules. In the algorithm's first phase (create), information about all potential rule applications is collected. In the second phase (select), all potential rule applications are checked for their applicability and then one rule is automatically or manually chosen for application. The third phase (execute and cleanup) consists of the application of the chosen rule and some cleanup activities adapting the information gathered in the create-phase to the changes made during the execution of the chosen rule. Then the execution is continued in the second phase until there are neither user decisions nor rules that can be executed automatically. In the following paragraphs, the three phases will be explained in more detail.

During the first activity in the create phase (create half-links), for each increment, that could be the dominant increment for at least one rule, a link referencing this increment as dominant is created (*half link*). The linking rules as presented in the previous subsections can be regarded as graph transformation rules with both sides of the transformation being compressed into one diagram. They can be divided into a left-hand side containing a graph pattern that has to be matched against the host graph before the transformation and a right-hand side describing the transformation result for the matched pattern. In the next activity (find potential rule applications), which is the first rule-specific activity, for each half link the left-hand sides of all rules that comply to the dominant increment are tried to be matched against the graph consisting of the documents to be integrated and the integration document. All possible matchings are stored at the half link. Please note that the context increments are not matched, yet, because these increments may still be created later by other rule applications. The other rule-specific activities are carried out similarly based on parts of the graph transformations described by the linking rules. In the next activity (detect overlappings), all intersections between the mapped increments of possible rule applications belonging to different links are found and stored in the graph.

In the first activity of the select phase, the context increments for all potential rule applications are tried to match and the result is stored in the graph. By doing this in this phase, the context increments are checked after each rule execution. Next, a potential rule application is searched that can be executed unambiguously. This is the case if all context increments are mapped, the rule is the only potential rule application connected to its half link, and there is no potential application residing at another half link with intersecting normal or dominant increments. If there is at least one unambiguous rule application, one of them is executed automatically. If there is no such rule, all ambiguities are collected and presented to the user, who can now manually select a potential rule application for execution. If there are no decisions, the algorithm terminates.

In the third phase (execute and cleanup), the selected rule is executed by replacing the already mapped left side in the host graph by its right side (execute rule). The next activities propagate the consequences of the changes to the data structure created during the create phase: First, all half links are deleted that cannot be made consistent because their dominant increment is now used by the executed rule. Next, all potential rule applications are deleted that became impossible because increments they needed are used by the executed rule. Last, all overlappings that are obsolete because one of the overlapping potential rule application involved was deleted are removed. Afterwards, the execution returns to the check context activity in the select phase.

The loop of the last two phases is continued until no unambiguous rules and no user decisions are present. It is possible that the integration is not completed after the termination of the algorithm. In such cases, the user has to complete the integration by modifying the documents and manually adding links to the integration document. From manually modified links, link templates can be interactively derived as is explained in [10].

## 6 Related work

We use UML [52] to express both the architectural refinement model and the multi-layered integration model. Therefore, extensions to the UML meta model have to be made [59] using the MOF [49]. Extensions of the OCL concerning graph transformation concepts like proposed in [61] will be used in future work to extend the modeling formalism.

### *6.1 Architecture modeling and refinement*

The observation that specifying the structure of a software system as coupled units with precise interfaces is a major contributing factor for developing a software system is almost as old as the software engineering discipline itself [53]. Due to the definition of a software architecture given in Sect. 3.1 it is not surprising that graph grammars were identified as a simple and natural way to model software architectures. Consequently, the rules and constraints for the dynamic evolution of the architecture, e.g. adding or removing components and relations between them, can be defined as graph transformations. Following this idea we use PROGRES [62] to describe these both aspects in an unified way.

Several related approaches are described in literature: Le Métayer [36] uses graph grammars to specify the static structure of a system. However, the dynamic evolution of an architecture has to be defined independently by a so-called coordinator. A uniform description language based on graph rewriting covering both aspects is presented by Hirsch et al. [31]. But in contrast to PROGRES this approach is limited to the use of context-free rules for specifying dynamic aspects. Similarly to us, Fahmy and Holt [21] also applies PROGRES to specify software architectural transformations.

Despite of this, these and other approaches for architecture modeling [34] claim to be usable to specify universal architectures independent from the domain and do not consider the needs for domain-specific architectures [39]. Therefore, PsiGene [58] allows to combine design patterns as presented in [13] and to apply them to class diagrams. A technique to specify patterns in the area of distributed applications and to combine them to a suitable software architecture is shown in [56].

While these approaches offer solutions for architectural patterns on a technical level, e.g. distributing components and defining patterns for their communication, they do not overcome the problem of semantic heterogeneity. This problem is addressed by numerous standardization efforts to define domain-specific interfaces based on corresponding architectural frameworks, e.g., OMG domain specifications [48], ebXML (*electronic business using eXtensible Markup Language* [18]) or OAGIS (*Open Applications Group* [46]). However, to adapt legacy systems to such standards they have to be wrapped. In this paper, we have shown how wrapping can be performed systematically at the architectural level.

### 6.2 Interactive modeling and construction of wrappers

An architecture-based approach for developing wrappers, similar to the one of us, is described by Gannod et al. [24]: interfaces to command line tools are specified as architectural components by using ACME [25], a generic architecture description language. Subsequently, based upon the specification the wrapper source code for the interface is synthesized. In comparison with our method presented in Sect. 4, Gannod et al. only cover the construction of the technical wrapper; any kind of data homogenization is not considered.

To enrich the expressiveness of a given interface to be wrapped, Jacobsen and Krämer modified CORBA IDL (*interface definition language*) [47] for adding specifications of semantic properties, so that a wrapper's source code can be extended by additional semantic checks automatically [32]. When wrapping tools in an a-priori manner, i.e. the semantics of the tool's interface is well-known, such descriptions are applicable for synthesizing the wrapper. Unfortunately, in the context of a-posteriori integration the semantic properties to be specified for generating the wrapper are unknown. This was one reason for developing our interface exploration tool.

Other attempts to discover the structure and behavior of a software system automatically come from the field of software reengineering. Cimitile et al. [14] describe an approach that involves the use of data flow analysis in order to determine various properties of the source code to be wrapped. A necessary prerequisite for this and the most other techniques in the area of software reengineering is the availability of the source code that is to be analyzed. Again, a-posteriori integration as presented in this paper does not satisfy this constraint.

The solution we have chosen is an application of the "programming by example" principle [38]. Several approaches for wrapping semi-structured data sources, e.g. web pages, following this principle can be found in literature. Turquoise [41] is a prototype of an intelligent web browser creating scripts to combine data from different web pages. The scripts are demonstrated by the user's browsing and editing actions, which Turquoise traces and generalizes into a program. Similar, NoDoSE [1] combines automatic analysis with user input to specify grammars for unstructured text documents. An automation of the generalization step, necessary in every programming by example approach, is presented in [35]. For a set of web pages single wrappers are specified manually, then an automatic learning algorithm generates a generalized wrapper by induction.

While these "programming by example" approaches concentrate on data integration, we are moreover interested in function and event integration, e.g. for offering the integration tool a visualized browsing functionality between integrated documents in the future.

### 6.3 Consistency management

Our approach to consistency Management is based on different foundations in computer sciences:

The idea of relating dependent pieces of information by links is borrowed from hypertexts [16]. The Chimera system [3] is an application of hypertext concepts to software engineering. In most hypertext systems – including Chimera – links have to be established manually. Some approaches to traceability, e.g. [57], follow the same principles.

Execution and definition of integration rules is based on graph transformation [19, 62], in particular pair grammars [55] and triple graph grammars [60]. Early work at our department concerning integration in software engineering [37] was carried out on the basis of these techniques during the construction of the integrated software engineering environment IPSEN [42]. We adapted the results to our current domain of application and extended the original approach: now, we are dealing with the problem of *a-posteriori* integration, the rule definition formalism was modified to the UML-based one described in this contribution (see Sect. 5.2) and the rule execution algorithm was further elaborated (see Sect. 5.3).

Related areas of interest in computer science are (in-)consistency checking [64] 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. Both fields of research recently gained increasing importance because of the model driven approaches for software development like the model driven architecture (MDA) [50]. In [27] the need for model transformations in the context of the MDA is described, some basic approaches are compared, and requirements on transformation languages and tools are proposed. In [33] requirements for mapping languages between different models are elaborated.

Without claiming the list to be exhaustive, here are some references to important projects in these areas:

XlinkIt [44,45] is a project dealing with consistency checks. XML technology is used to check for inconsistencies and to repair them. Because of the structure of the documents in our domain, we believe that UML and graph grammars are better suited to model and execute integration functionality.

The ViewPoint framework [22] dates back to the early nineties. Its main idea is to identify different perspectives (view points) of a product that is to be developed and to examine their inter-relations. This framework is applied as a basis in [20] using the formalism of distributed graph transformations [65] to detect and repair inconsistencies. To the best of our knowledge, this approach does not support conflicting rules and user interaction.

In the context of the Fujaba project, a consistency management approach was developed [67]. It checks intra-model as well as inter-model consistency. Parts of the inter-model consistency check [66], which can be used to transform models, are based on the triple graph grammar approach [60] like ours, but offer restricted transformation functionality only, w.r.t. the detection of conflicting rules and user interaction.

As part of the Kent Modeling Framework, in [2] a relational approach to the definition and implementation of model transformations is proposed. The definition of transformation rules is accomplished using UML class diagrams enriched with OCL constraints. This approach has the formal background of mathematical relations. It is not applicable in our domain of application because the rules are not intuitively understandable.

The QVT Partner's proposal [4] to the QVT RFP of the OMG [51] is very similar to Kent's approach but complements it with graphical definition of patterns and operational transformation rules. It does not support incrementality and user interaction.

BOTL [12] is a transformation language based on UML object diagrams. A BOTL rule consists of a left hand side diagram being matched in the source document and a right hand side diagram being created in the target document if a left hand side matching was found. The transformation process is neither incremental nor interactive.

In general, it can be observed that most approaches from the area of consistency management support an incremental mode of operation and user interaction, whereas most model transformation approaches work batch-wise. In our domain of application, chemical engineering, incremental transformation between models is urgently needed. But unlike MDA, in chemical engineering a complete and unambiguous definition of the translation between for instance flowsheets and simulation models is not feasible. Instead, a lot of decisions have to be made by the user and a lot of situations have to be dealt with manually. Consistency checking approaches with repair actions could be used for transformation, but there are problems like conflicting transformation rules and termination of the transformation process, which require further extensions of the approaches. We address these problems with the integration algorithm described in Sect. 5.3.

The advantage of our integration approach is that it uses standard UML for intuitive definition of integration rules and supports an incremental mode of operation as well as bidirectional transformation including conflict detection and user interaction.

## 7 Conclusion

We have presented an architecture-based and model-driven approach to the a-posteriori integration of engineering tools for incremental development processes. We have realized this approach in the context of the CRC 476 IMPROVE, which is concerned with models and tools for supporting design processes in chemical engineering. The case study presented in this paper – an integration tool for consistency management between flowsheets and simulation models – demonstrates that tight integration can be achieved even in the case of a-posteriori integration of heterogeneous tools developed by different vendors. Furthermore, since the tool development process is strongly architecture- and model-driven, the process can be performed at a fairly high level of abstraction with considerably reduced effort.

An important goal of IMPROVE is to transfer research into industrial practice. The work reported in this paper constitutes a contribution towards this goal. The integration tool between Comos PT and Aspen Plus has been developed in close cooperation with innotec, a software company which develops and sells Comos PT. Tool development has been performed in close cooperation with innotec, taking the requirements of the industrial partner into account. Thus, the integration tool developed in this cooperation provides a test case for our approach to tool development. The experiences we have gained so far are promising, but clearly further work on practical applications will have to be performed to obtain substantial feedback on the advantages and drawbacks of our approach to tool development.

*Acknowledgements.* This work was in part funded by the CRC 476 IMPROVE of the Deutsche Forschungsgemeinschaft (DFG). Furthermore, the authors gratefully acknowledge the fruitful cooperation with innotec.

## References

1. Adelberg B (1998) NoDoSE – a tool for semi-automatically extracting structured and semistructured data from text documents. In: Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data, Seattle, Washington, USA. ACM, pp 283–294
2. Akehurst D, Kent S, Patrascoiu O (2003) A relational approach to defining and implementing transformations between metamodels. Journal on Software and Systems Modeling 2(4):215–239
3. Anderson KM, Taylor RN, Whitehead EJ (2000) Chimera: Hypermedia for heterogeneous software development environments. ACM Transactions on Information Systems 18(3):211–245
4. Appukuttan BK, Clark T, Reddy S, Tratt L, Venkatesh R (2003) A model driven approach to model transformations. In: Proc. of the 2003 Model Driven Architecture: Foundations and Applications (MDAFA2003), CTIT Technical Report TR-CTIT-03-27. Univ. of Twente, The Netherlands
5. Bass L, Kazman R (1999) Architecture-based development. Technical Report CMU/SEI-99-TR-007, Carnegie Mellon University, Software Engineering Institute (SEI)

6. Bayer B (2003) Conceptual information modeling for computer aided support of chemical process design. Fortschritt-Berichte VDI, Reihe 3, 787. VDI Verlag, Düsseldorf, Germany
7. Becker S, Haase T, Westfechtel B, Wilhelms J (2002) Integration tools supporting cooperative development processes in chemical engineering. In: Proc. of the 6th Biennial World Conf. on Integrated Design and Process Technology (IDPT-2002), Pasadena, California, USA. Society for Design and Process Science, p 24
8. Becker SM, Lohmann S, Westfechtel B (2004) Rule execution in graph-based incremental interactive integration tools. In: Proc. of the 2nd Intl. Conf. on Graph Transformations (ICGT 2004), LNCS, vol 3256. Springer, pp 22–38
9. Becker SM, Westfechtel B (2003) Incremental integration tools for chemical engineering: An industrial application of triple graph grammars. In: Proc. of the 29th Workshop on Graph-Theoretic Concepts in Computer Science (WG 2003), LNCS, vol 2880. Springer, pp 46–57
10. Becker SM, Westfechtel B (2003) UML-based definition of integration models for incremental development processes in chemical engineering. In: Proc. of the 7th World Conf. on Integrated Design and Process Technology (IDPT-2003), Austin, Texas, USA. Society for Design and Process Science, p 46
11. Böhlen B, Jäger D, Schleicher A, Westfechtel B (2002) UPGRADE: Building interactive tools for visual languages. In: Proc. of the 6th World Multiconference on Systemics, Cybernetics, and Informatics (SCI 2002), Orlando, Florida, USA, pp 17–22
12. Braun P, Marschall F (2003) Transforming object oriented models with BOTL. In: Electronic Notes in Theoretical Computer Science, vol 72. Elsevier
13. Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M (1996) Pattern-Oriented Software Architecture: A System of Patterns, vol 1. Wiley
14. Cimitile A, de Lucia A, de Carlini U (1998) Incremental migration strategies: Data flow analysis for wrapping. In: Proc. of the 5th Working Conf. on Reverse Engineering (WCRE'98), Hawaii, USA. IEEE, pp 59–68
15. Clements PC, Northrop L (1996) Software architecture: An executive overview. Technical Report CMU/SEI-96-TR-003, Carnegie Mellon University, Software Engineering Institute (SEI)
16. Conklin J (1987) Hypertext: an introduction and survey. IEEE Computer 20(9):17–41
17. Donohoe P (ed) (1999) Software Architecture (TC2 1st Working IFIP Conf. on Software Architecture, WICSA1). Kluwer, San Antonio, Texas, USA
18. ebXML (2004) Technical architecture specification. Available from World Wide Web: `http://www.ebxml.org/specs/index.htm` [cited July 2004]
19. Ehrig H, Engels G, Kreowski HJ, Rozenberg G (eds) (1999) Handbook on Graph Grammars and Computing by Graph Transformation: Application, Languages, and Tools, vol 2. World Scientific
20. Enders BE, Heverhagen T, Goedicke M, Tröpfner P, Tracht R (2002) Towards an integration of different specification methods by using the viewpoint framework. Transactions of the SDPS 6(2):1–23
21. Fahmy H, Holt RC (2000) Using graph rewriting to specify software architectural transformations. In: Proc. of the 15th Intl. Conf. on Automated Software Engineering (ASE 2000). IEEE, pp 187–196
22. Finkelstein A, Kramer J, Goedicke M (1990) ViewPoint oriented software development. In: Intl. Workshop on Software Engineering and Its Applications, pp 374–384
23. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley
24. Gannod GC, Mudiam SV, Lindquist TE (2000) An architectural based approach for synthesizing and integrating adapters for legacy software. In: Proc. of the 7th Working Conf. on Reverse Engineering (WCRE'00), Brisbane, Australia
25. Garlan D, Monroe R, Wile D (1997) Acme: An architecture description interchange language. In: Proc. of the 1997 Conf. of the Centre for Advanced Studies on Collaborative Research (CASCON'97), Toronto, Ontario, Canada. IBM, pp 169–183
26. Garlan D, Perry DE (1995) Introduction to the special issue on software architecture. IEEE Transactions on Software Engineering 21(4):269–274
27. Gerber A, Lawley M, Raymond K, Steel J, Wood A (2002) Transformation: The missing link of MDA. In: Proc. of 1st Intl. Conf. on Graph Transformations (ICGT 2002), LNCS, vol 2505, Barcelona, Spain. Springer, pp 90–105
28. Haase T (2003) Semi-automatic wrapper generation for a-posteriori integration. In: Workshop on Tool Integration in System Development (TIS 2003), Helsinki, Finland, pp 84–88
29. Haase T, Meyer O, Böhlen B, Gatzemeier F (2004) A domain specific architecture tool: Rapid prototyping with graph grammars. In: Pfaltz et al. [54], pp 236–242
30. Haase T, Meyer O, Böhlen B, Gatzemeier F (2004) Fire3: Architecture refinement for a-posteriori integration. In: Pfaltz et al. [54], pp 461–467
31. Hirsch D, Inverardi P, Montanari U (1999) Modeling software architectures and styles with graph grammars and constraint solving. In: Donohoe [17], pp 127–143
32. Jacobsen H-A, Krämer BJ (1998) A design pattern based approach to generating synchronization adaptors from annotated idl. In: Proc. of the 13th Intl. Conf. on Automated Software Engineering (ASE'98), Hawaii, USA. IEEE, pp 63–72.
33. Kent S, Smith R (2003) The Bidirectional Mapping Problem. Electronic Notes in Theoretical Computer Science 82(7)
34. Klein P (2001) Architecture Modeling of Distributed and Concurrent Software Systems. PhD thesis. Wissenschaftsverlag Mainz, Aachen, Germany
35. Kushmerick N (2000) Wrapper induction: Efficiency and expressiveness. Artificial Intelligence 118(1–2):15–68
36. Le Métayer D (1998) Describing software architecture styles using graph grammars. IEEE Transactions on Software Engineering 27(7):521–533
37. Lefering M, Schürr A (1996) Specification of integration tools. In: Nagl [42], pp 324–334
38. Lieberman H (ed) (2001) Your wish is my command: Programming by example. Academic Press
39. Mettala E, Graham MH (1992) The domain-specific software architecture program. Technical Report CMU/SEI-92-SR-009, Carnegie Mellon University, Software Engineering Institute (SEI)
40. Microsoft (2004) The component object model specification. Available from World Wide Web: `http://www.microsoft.com/com/resources/comdocs.asp` [cited July 2004]
41. Miller RC, Myers BA (1999) Creating dynamic world wide web pages by demonstration. Technical Report CMU-CS-97-131, Carnegie Mellon University, School of Computer Science
42. Nagl M (ed) (1996) Building Tightly-Integrated Software Development Environments: The IPSEN Approach. LNCS, vol 1170. Springer
43. Nagl M, Marquardt W (1997) SFB 476 IMPROVE: Informatische Unterstützung übergreifender Entwicklungsprozesse in der Verfahrenstechnik. In: Informatik '97: Informatik als Innovationsmotor, Informatik aktuell. Springer, pp 143–154. in German.
44. Nentwich C, Capra L, Emmerich W, Finkelstein A (2002) xlinkit: A consistency checking and smart link generation service. Transactions on Internet Technology 2(2):151–185
45. Nentwich C, Emmerich W, Finkelstein A (2003) Consistency management with repair actions. In: Proc. of Intl. Conf. on Software Engineering (ICSE). ACM, pp 455–464
46. OAGIS (2004) Open Applications Group. Available from World Wide Web: `http://www.openapplications.org` [cited July 2004]
47. OMG (2004) CORBA/IIOP specification. Available from World Wide Web: `http://www.omg.org/technology/documents/formal/corba_iiop.htm` [cited July 2004]
48. OMG (2004) Domain specifications. Available from World Wide Web: `http://www.omg.org/technology/documents/domain_spec_catalog.htm` [cited July 2004]
49. OMG (2004) Meta object facility (MOF) specification. Available from World Wide Web: `http://www.omg.org/technology/documents/formal/mof.htm` [cited July 2004]
50. OMG (2004) Model driven architecture (MDA) specifications. Available from World Wide Web: `http://www.omg.org/mda/specs.htm` [cited July 2004]
51. OMG (2004) MOF 2.0 query / view / transformations, request for proposal. Available from World Wide Web: `http://www.omg.org/techprocess/meetings/schedule/MOF_2.0_Query_View_Transf._RFP.html` [cited July 2004]

52. OMG (2004) Unified modeling language (UML) specification. Available from World Wide Web: `http://www.omg.org/technology/documents/formal/uml.htm` [cited July 2004]
53. Parnas DL (1972) A technique for software module specification with examples. Communications of the ACM 15(5):330–336
54. Pfaltz JL, Nagl M, Böhlen D (eds) (2004) Applications of Graph Transformations with Industrial Relevance (Proc. 2nd Intl. Workshop AGTIVE 2003). LNCS, vol 3062. Springer
55. Pratt TW (1971) Pair grammars, graph languages and string-to-graph translations. Journal of Computer and System Sciences (JCSS) 5(6):560–595
56. Radermacher A (2000) Support for design patterns through graph transformation tools. In: Applications of Graph Transformation with Industrial Relevance (Proc. Intl. Workshop AGTIVE'99), LNCS, vol 1779. Springer, pp 111–126
57. Ramesh B, Jarke M (2001) Toward reference models of requirements traceability. Software Engineering 27(1):58–93
58. Riegel JP, Kaesling C, Schütze M (1999) Modeling software architecture using domain-specific patterns. In: Donohoe [17], pp 273–301
59. Schleicher A, Westfechtel B (2001) Beyond stereotyping: Metamodeling approaches for the UML. In: Proc. of the 34th Annual Hawaii Intl. Conf. on System Sciences (HICSS-34), Hawaii, USA. IEEE
60. Schürr A (1995) Specification of graph translators with triple graph grammars. In: Proc. of the 20th Intl. Workshop on Graph-Theoretic Concepts in Computer Science (WG 1994), LNCS, vol 903, Herrsching, Germany. Springer, pp 151–163
61. Schürr A (2001) Adding graph transformation concepts to UML's constraint language OCL. In: Electronic Notes in Computer Science, vol 44. Elsevier
62. Schürr A, Winter A, Zündorf A (1999) The PROGRES approach: Language and environment. In: Ehrig et al. [19], pp 487–550
63. Sneed HM (2000) Encapsulation of legacy software: A technique for reusing legacy software components. Annals of Software Engineering 9(1–4):293–313
64. Spanoudakis G, Zisman A (2001) Inconsistency management in software engineering: Survey and open research issues. In: Handbook of Software Engineering and Knowledge Engineering, vol 1. World Scientific, pp 329–380
65. Taentzer G, Koch M, Fischer I, Volle V (1999) Distributed graph transformation with application to visual design of distributed systems. In: Handbook on Graph Grammars and Computing by Graph Transformation: Concurrency, Parallelism, and Distribution, vol 3. World Scientific, pp 269–340
66. Wagner R (2001) Realisierung eines diagrammübergreifenden Konsistenzmanagement-Systems für UML-Spezifikationen. Master's thesis, University of Paderborn. in German.
67. Wagner R, Giese H, Nickel UA (2003) A plug-in for flexible and incremental consistency mangement. In: Proc. of the Intl. Conf. on the Unified Modeling Language (UML 2003), San Francisco, California, USA. Springer

**Simon M. Becker** received his degree in computer science from the RWTH Aachen University in 2001. Since then, he has been working at the Department of Computer Science III of the RWTH Aachen University as research assistant. His main area of research is data integration, especially concerning the a-posteriori integration of dependent documents in development processes. The research is carried out under the umbrella of the collaborative research centre 476 IMPROVE which deals with computer science support for development processes in chemical engineering.

**Thomas Haase** is research assistant at the Department of Computer Science III at the University of Technology Aachen, Germany. He is a member of the collaborative research centre 476 IMPROVE, dealing with computer science support for development processes in chemical engineering. His research areas include architectures and frameworks for the a-posteriori integration of software tools. He holds a diploma degree in computer science from the University of Koblenz-Landau, Germany.

**Bernhard Westfechtel** received his diploma degree in 1983 from University of Erlangen-Nürnberg and his doctoral degree (PhD) in 1991 from Aachen University of Technology, where he has been working as a senior researcher since then. He is interested in software engineering environments, software configuration management, process modeling, workflow management, object-oriented modeling, engineering/product data management, database systems for engineering applications, and software architectures.