*Integrated Design and Process Technology*, **IDPT-2002**
Printed in the United States of America, June, 2002
©2002 Society for Design and Process Science

# Integration Tools Supporting Cooperative Development Processes in Chemical Engineering

**Simon Becker, Thomas Haase, Bernhard Westfechtel, Jens Wilhelms**
**Computer Science III, Aachen University of Technology, D-52056 Aachen**
***(sbecker—thaase—bernhard—jensw)@i3.informatik.rwth-aachen.de***

***ABSTRACT:***
Development processes in chemical engineering involve many engineers who are dealing with different working areas. They use tools that produce heterogeneous documents. For instance, one engineer creates a process flow diagram (PFD) with a tool like Comos PT. Another engineer has to simulate parts of this chemical process using a simulation tool like Aspen Plus, using a document describing the simulator input and producing a document containing the simulator output. A lot of dependencies exist between the documents created in development processes. For example, the simulator input file has to be consistent to the selected part of the PFD. In current development processes, these dependencies and the resulting consistency relationships have to be handled manually by the engineers without appropriate tool support. Some tools exist that allow the generation of one document out of other documents but at each generation step the target document is overwritten, and all changes applied to it are lost. We are developing tools that explicitly model the inter-document relationships . Moreover, they work incrementally, i.e., only those parts of the target document are altered that depend on parts of the source document that were modified. In this paper we describe the resulting tools and their benefit for chemical engineering.

## I. INTRODUCTION

*Development processes* in different disciplines such as software, mechanical, chemical, or electrical engineering are inherently complex. In order to design a sophisticated technical product, engineers create a large number of interdependent documents. To this end, they make use of potentially heterogeneous tools supplied by different vendors. It is not only difficult to design documents which are internally consistent. In addition, external consistency between interdependent documents has to be ensured.

This paper deals with integration tools for maintaining consistency between interdependent documents created by heterogeneous tools. That is, we are concerned with *a posteriori integration* (which is also called bottom up integration): tools are integrated only after they have been implemented independently from each other.

Frequently, heterogeneous tools are integrated with the help of batch converters. A *batch converter* transforms an input document $d$ created by some tool $t$ into an output document $d'$ to be processed by another tool $t'$. Often, the output produced by a batch converter is modified interactively by a human user.

While batch converters work fine in the case of waterfall-like development processes, they are not suited for incremental processes. For example, the user working on $d'$ may detect problems in $d$. As a consequence, $d$ needs to be modified, and these changes have to be propagated into $d'$. Unfortunately, another run of the batch converter destroys the changes which the user has applied to $d'$ in the meantime. Instead, an *incremental integration tool* is required which propagates only the changes performed in $d$ into $d'$ without implying the loss of manual changes in $d'$.

In this paper, we present incremental integration tools which are designed to support concurrent/simultaneous engineering. Relationships between interdependent documents are stored in an *integration document* which is placed in between the related documents. The integration document is composed of *links* for navigating between fine-grained objects stored in the respective documents. Furthermore, these links are used to determine the impact of changes, and they are updated in the course of change propagation.

Integration tools are driven by *rules* defining which objects may be related to each other. Each rule relates a pattern of source objects to a pattern of target objects via a link. Rules may be applied automatically or manually. They are collected in a rule base which represents domain-specific knowledge. Since this knowledge evolves, the rule base may be extended on the fly.

We demonstrate our approach to tool integration by a case study in the domain of chemical engineering. In collaboration with an industrial partner (innotec, a German software company; one of its products is the computer aided engineering solution *Comos PT*), we are building a tool for integrating process flow diagrams with simulation models. A chemical engineer may use the integration tool to construct a simulation model from a process flow diagram and vice versa. To this end, he may apply predefined integration rules and may insert new ones on the fly. In general, m:n relationships may be established between objects of the participating documents. Changes may be propagated in both directions.

This specific tool is based on a general *framework* for building incremental integration tools. The framework provides integration documents for storing links, rule bases

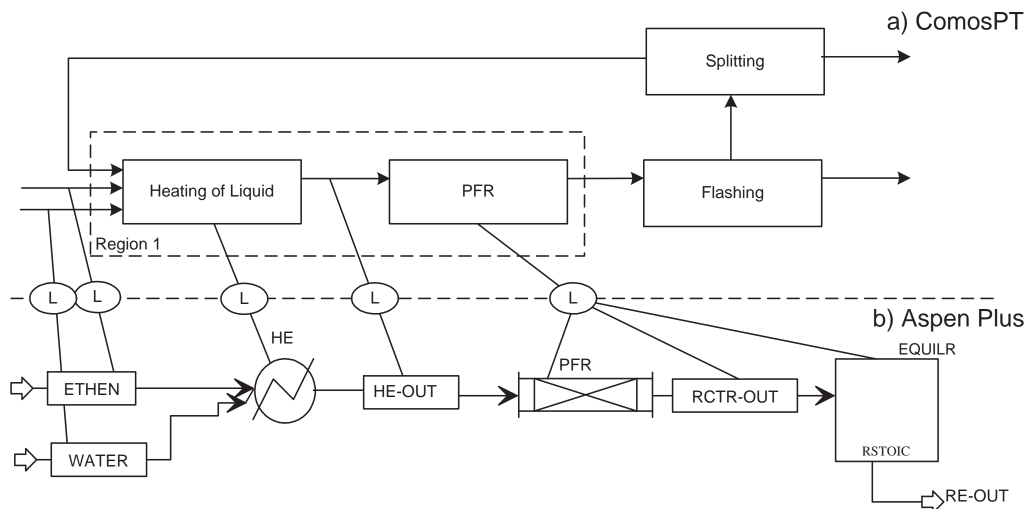


Fig. 1. PFD in Comos PT, Aspen Plus simulation corresponding to Region 1

collecting integration rules, algorithms for applying these rules, and wrappers for tool encapsulation. It is funded on theoretical work in the area of graph transformations [11]. With the help of the framework, the effort for building specific integration tools is reduced considerably.

The rest of this paper is structured as follows: Section II introduces a motivating example, from which requirements to integration tools are derived in Section III. Section IV, which constitutes the core part of this paper, describes our approach to tool integration. Section V discusses related work, and Section VI concludes this paper.

## II. Motivating Example

The running example to be used in this paper is drawn from the domain of chemical engineering. Here, we are concerned with the design of plants in which chemical processes are performed. We focus on *basic engineering*, where the fundamental decisions concerning the design of the process and the respective plant are made. In basic engineering, *process flow diagrams* (PFDs) play a central role since they represent the chemical process at a conceptual level. However, PFDs merely provide structural views of the chemical process. Its behavior is studied with the help of laboratory experiments and computer simulations based on detailed mathematical models. Only the latter will be included into our running example.

Please note that the overall design process is highly *iterative*. Typically, the designer starts out with a fairly coarse-grained PFD representing different process alternatives. Then, simulations are performed to get initial insight into the behavior of the chemical process. The results of simulations can be fed back into the PFD so that the designer may view process parameters such as streams, temperature, etc. in the PFD. The simulation results are used to select the most suitable design alternative, which is refined further in subsequent steps. More detailed simulations are performed on the basis of the extended PFD, etc.

Two tools from different vendors are involved in our example: *Comos PT*, which includes an editor for PFDs, and *Aspen Plus*, which supports both stationary and dynamic simulations. In between, we place an integration tool which supports consistency control between PFDs created in Comos PT and simulation models created in Aspen Plus. We sketch the functionality that the user expects from the integration tool, but we refrain from showing screenshots.

Our case study deals with an organic process, namely the production of ethanol from ethen and water. The designer represents this process with the help of a PFD, in which the overall process is decomposed into four steps (Figure 1a): heating of liquid, reaction (in a plug flow reactor, PFR), flashing (to separate the output), and splitting (to feed back most of the valuable input substances which are existent in the reactor outlet).

To get more detailed information on the process, the designer delegates its simulation to a modeler. Since it is too complex to simulate the whole process in one shot, it is decided to start with the simulation of a certain part of it. This part — called `Region 1` — is designated by the dashed rectangle in Figure 1a.

The modeler uses an integration tool to create the simulation model in a semi-automatic way. The integration tool creates a simulator input file which can be loaded into Aspen Plus. In addition, it creates fine-grained links (labeled with `L` in the figure) between Comos PT and Aspen Plus objects. These links serve multiple purposes: browsing (navigation across the links), traceability (records of mapping decisions), and change propagation (identification of objects affected by a change).

In our example, the input flows of the heating step are transformed automatically into corresponding inlets. For

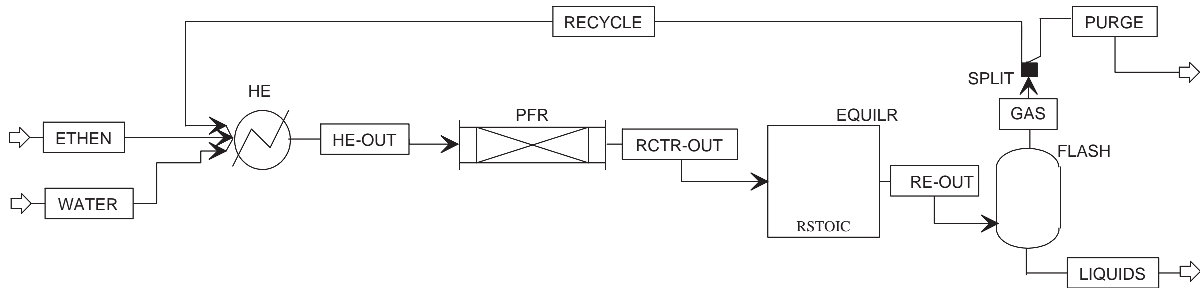


Fig. 2. Aspen Plus simulation corresponding to the whole PFD in Figure 1a

the heating step, Aspen Plus provides multiple blocks that can be used alternatively for simulation. Here, a user interaction is necessary. The modeler selects the block he considers most appropriate for the simulation task at hand.

So far, predefined integration rules have been applied either automatically (in the case of a deterministic transformation) or interactively (in the non-deterministic case). Concerning the reaction part, the situation is different. The modeler has to devise a suitable model fragment in Aspen Plus on his own. It turns out that no single block is available for the simulation of the reaction. Therefore, the designer decides to use two blocks each of which handles part of the overall reaction. This is an example of a 1:n relationship; in general, m:n relationships are possible and allowed.

The designer records his mapping decision by applying a so-called ad hoc integration rule (meaning that any sets of source and target objects may be related to each other). *Ad hoc rules* may be used whenever domain knowledge is not yet available (domain-specific rules have not yet been defined). Domain-specific rules may then be derived from instances of ad hoc rules. Since the modeler expects that the same transformation will be used repeatedly in future simulations, he invokes a command that generates a specific integration rule from the instance of the ad hoc rule.

After that, the modeler reads the input file into Aspen Plus and runs the simulation. The results of the simulation are propagated back into the PFD with the help of the integration tool. To this end, the integration tool analyzes the simulation results and relates them to the structural elements of the PFD. As a consequence, the PFD is enriched with simulation data.

As a next step, it is decided to simulate the overall process rather than just heating and reaction. Thus, the simulation region is extended to cover the whole PFD. The integration tool is used to extend the simulation model accordingly; the result is shown in Figure 2. In contrast to a batch converter, the integration tool does not generate a simulation model from scratch. Rather, it retains those parts which have already been transformed, implying that manual work and user decisions are not lost. To this end, it employs the links established between PFD and simulation model. Subsequently, the whole process is simulated in Aspen Plus, and the results are propagated back into the PFD.

Please note that the example presented above is simplified and does not fully exploit the capabilities of the integration tool. In general, the integration tool may propagate not only extensions, but also modifications and deletions from the PFD into the simulation model. Likewise, backward propagation may involve not only extensions (enrichment of the PFD with simulation data). In particular, the integration tool may be employed to import already existing simulation models, and the modeler may also perform structural changes in Aspen Plus which are propagated back into the PFD. Finally, PFD and simulation model may be modified concurrently. Thus, the integration tool supports full, bidirectional, and incremental change propagation.

## III. Requirements

From the motivating example presented in the previous section, we derive the following requirements:

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

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

*Direction* In general, an integration tool may have to work in both directions. That is, if $d_1$ is changed, the changes are propagated into $d_2$ and vice versa.

*Mode of interaction* While an integration tool may operate automatically in simple scenarios, it is very likely that user interactions are required to resolve non-deterministic choices.

*Time of activation* In single user applications, it may be desirable to propagate changes eagerly. This way, the user is informed promptly about the consequences of the changes performed in the respective documents. In multi user scenarios, however, *deferred propagation* is usually required.

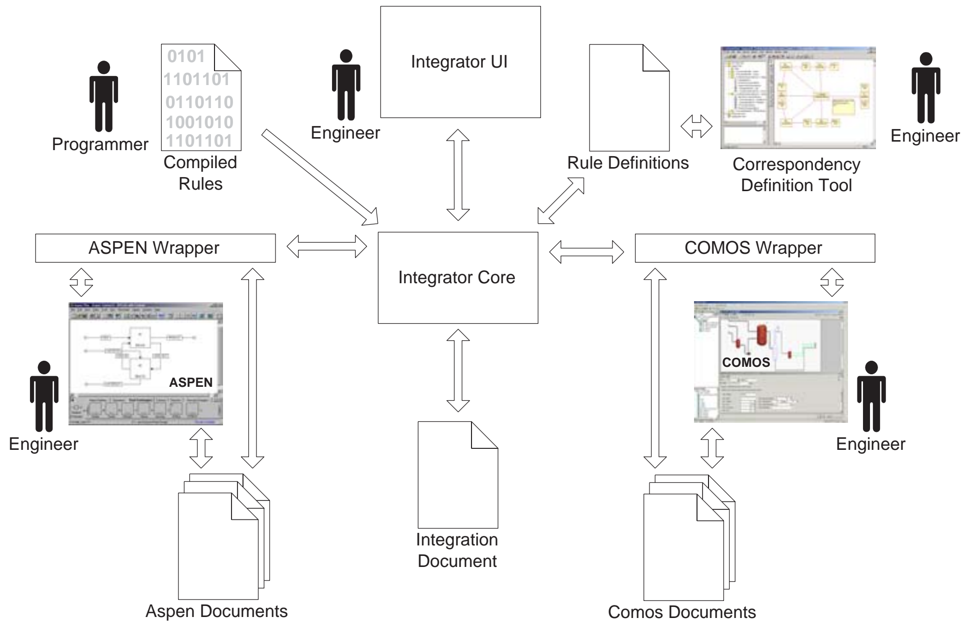


Fig. 3. System architecture

In this way, each user keeps control of the export and import of changes from/to his local workspace.

*Integration rules* An integration tool is driven by rules defining which *object patterns* may be related to each other. It must provide support for defining and applying these rules.

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

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

*A posteriori integration* An integration tool must work with heterogeneous tools supplied by different vendors. To this end, it has to access these tools via corresponding wrappers which provide abstract and unified interfaces.

## IV. Integration Approach

### A. System Architecture

Figure 3 provides an overview of the *system architecture* for integration tools. At the heart of this architecture, the integrator core offers basic functionality. In particular, it includes the basic control logic, i.e., the algorithms for document integration (Subsection IV-D). The integrator core accesses the integration document which stores fine-grained links and records the application of integration rules (Subsection IV-B). Furthermore, it is connected to the tools and documents to be integrated via respective *wrappers*, which are used to launch the tools, navigate and update their documents, etc. (Subsection IV-E). The integrator user interface is used to control the integrator interactively.

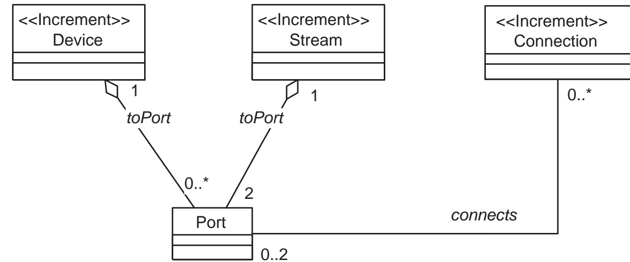


Fig. 4. Document model

Integration rules (Subsection IV-C) may be defined in two ways. First, the user may define a rule ad hoc based on an actual correspondence of object patterns. Second, a more systematic approach may be followed by employing a correspondence definition tool based on the UML. A rule may be stored in a rule definition base, where it is interpreted by the integrator core. This way, rules may be added on the fly. Alternatively, a rule may be compiled for more efficient execution. This results in a hybrid execution approach.

### B. Document Integration Model

To simplify the integration of Comos PT and Aspen Plus the proprietary data model of both applications is translated into a generic graph model for PFDs by the tool wrappers. Figure 4 shows the structural part of this data model in UML. Comos PT devices and Aspen Plus blocks are

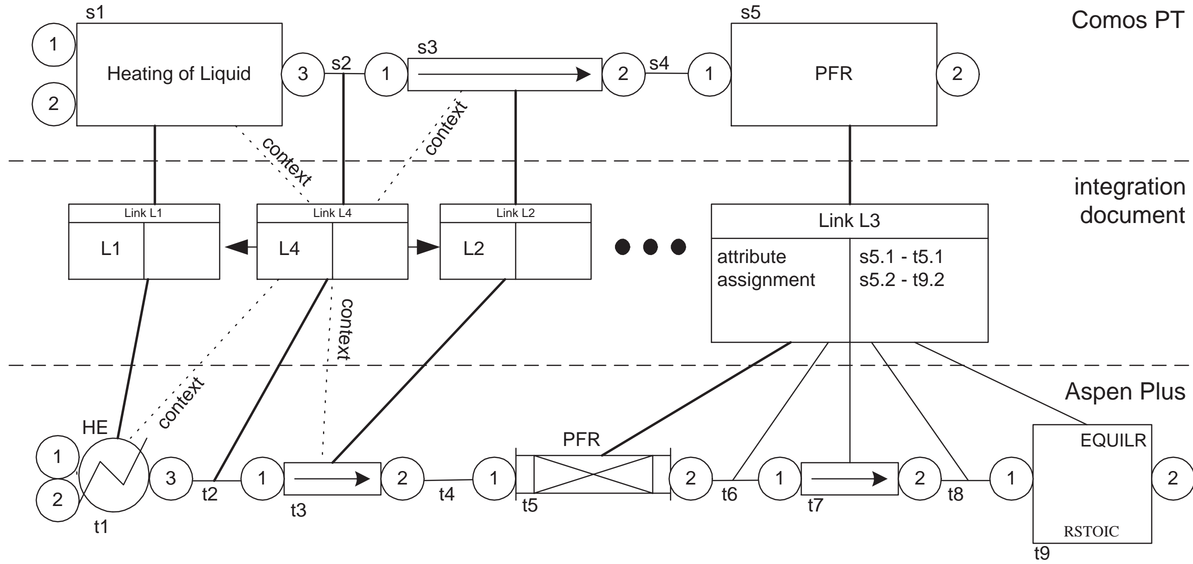


Fig. 5. Integration document

mapped to objects of the class `Device`. The class `Stream` represents the stream classes of both application. Streams aggregate exactly two objects of the class `Port`, one input and one output, devices can aggregate zero to many ports. Ports are used to model the connections between devices and streams. Ports are connected by objects of the class `Connection`, via `connects` associations. One connection connects at most two ports; one port can participate in many `connects` relationships.

Most PFD-based applications have their own type hierarchy for devices and streams. These are reflected by application specific subclasses of the appropriate class in the generic model. For example, the Comos PT PFR is modeled by a subclass `PFR` of the class `Device`. According to the applications' data model, these subclasses can have user defined as well as predefined attributes.

From the integration point of view, `Device`, `Stream`, and `Connection` are *increments*, i.e. fine-grained document parts that can be related to increments in other documents via fine-grained inter-document relationships. This fact is expressed by the stereotype `Increment`. Ports are regarded as belonging to the aggregating device or stream, so they do not have this stereotype.

As mentioned in Section IV-A, integration tools store the fine-grained inter-document links in an integration document. Figure 5 shows an excerpt of the integration document that is created by the integrator run that transforms `Region 1` in Figure 1 into the corresponding Aspen Plus input. In this figure, the Comos PT and Aspen Plus documents are presented in the generic PFD graph model. The ports that belong to streams and devices are represented by numbered circles. In between the two documents the integration document with the fine-grained links is depicted. For instance, the link `L3` relates the reaction (PFR) in the Comos PT document to the corresponding simulation blocks (PFR, EQUILR) and the connecting stream (`t7`) in the Aspen Plus document. The connections (`t6`, `t8`) are referenced by the link, too. While `L3` relates one increment of the source document to five increments in the target document, in general, many-to-many links between increments are supported.

Referenced increments can play three different roles: *increment*, *dominant increment*, and *context increment*. Any increment can be referenced by zero to many links as context increment but only by one link as increment or dominant increment. One increment of each link can be marked as dominant increment (c.f. `t5`), which is done at the application of integration rules, which will be explained in the next section. Context increments are needed to access the attributes of an increment, without including it in the link as regular increment.

In addition to the references to the increments in both documents, links contain a set of attribute assignment specifications (see also Section IV-C). Each specification contains the assignment of multiple attributes. For each integrator run, a different specification can be used. An attribute assignment can be specified in two ways: simple assignments in tabular form and complex calculations with scripting code like Visual Basic Script. In addition to the set of assignment specifications, there has to be a at least one table or script that checks if the attributes are still in a consistent state or a new assignment has to be performed.

The link `L3` contains additional information. To identify the ports in the Aspen Plus document corresponding to port `1` and `2` of the PFR, the port mapping has to be specified. In this case, the first port of the PFR (`s5.1`) is mapped

to the first port of the PFR in the Aspen Plus document (`t5.1`). Similar, the second port (`s5.2`) is mapped to the port `t9.2`. In general, there can be one-to-many relationships between ports.

The port mapping is used by links that map the connections in the different documents. For example, the link `L4` relates the connection `s2` to the connection `t2`. In addition to the two connections, it references the increments `s1`, `s3`, `t1`, and `t3` as context increments. If, as in our example, `t2` is generated by a transformation from `s2`, it is necessary to know which are the ports corresponding to `s1.3` and `s3.1`. This information can be looked up in the port mapping of the links that reference the context increments, `L1` and `L2`. This results in a dependency of `L4` upon `L1` and `L2`, which is expressed by the arrows between them.

### C. *Integration Rules*

Although the manual definition of links is possible, in most cases integration tools use rules to decide automatically which sets of increments from both documents are to be connected by a link. If the possible rule applications are ambiguous, user interaction is needed.

Our prototype supports two ways of defining rules: on the fly generation of rules based on manually drawn links and rule definition in a UML-based correspondence editor. In the first case, the integration tool adds the new rule to its rule base which is interpreted at runtime. In the second case, the user can choose whether the rule is fed into the rule base via an XML file and interpreted, or it is coded in a programming language by hand and linked to the integration tool as compiled rule. This can be more efficient for complex transformations. In future versions of our prototype, automatic code generation will be possible, too.

To define a rule on the fly based on a given link, as the first step, the source and target patterns are generated. These patterns describe the increment structure in the source and target document. Each increment is specified by its type, its role and a placeholder name. The latter is substituted by the identifier of the specific increment when the pattern is matched against the document to apply rules. The port mapping can be derived from the link's correspondence information. If not already present in the link, one increment on each link side has to be marked as dominant. The result of this step is the description of a *pair of corresponding increment patterns* on the abstract instance level.

In the next step, based on these corresponding patterns, multiple *structural integration rules* can be defined. For each rule the user has to decide if the actual type of the increments is to be used in the pattern description or any type that is located in the path from the increments' types to the root of the type hierarchy (i.e. `Stream` or `Device`). Now it has to be specified for each increment placeholder if it has to be matched to existing increments or has to be created in one of the documents. By doing so, different kinds of rules can be defined:

*Consistency check* Rules where no increments are marked for creation can be used to check the consistency between two documents. When the rules are executed the documents' structures remain unchanged, only the links are created in the integration document.

*Forward transformations* Rules where only increments on the target document side are marked for creation are used to create the target document's structure corresponding to the source document.

*Backward transformations* These rules are comparable to forward transformations but work in the opposite direction.

*Synchronous transformations* Rules where increments on both sides are marked for creation modify the structure of both documents.

To simplify dealing with large sets of rules, a short rule description and a rule name should be entered.

For example, the link `L3` in Figure 5 can be translated automatically into an abstract instance correspondence by replacing the identifiers with placeholders. Based on this correspondence, a consistency checking rule can be defined that adds the link `L3` to the integration document if the source and target patterns are matched in the documents while performing an integration. To specify a forward transformation, all target increments have to be marked for creation. If a PFR is found in the source document, this rule generates the link `L3` in the integration document and the whole target pattern in the target document. Similarly, backward and synchronous transformations can be defined.

In addition to the structural integration rules, a *specification of the attribute assignment* is necessary. The attribute assignment takes place after the rule's structural transformations have been performed and the link between the corresponding increments has already been drawn. As a result, the attribute assignment does not have to be specified for each rule. Instead, for each pair of corresponding increment patterns a set of attribute assignments as tables or as scripting code can be defined. In addition to the assignments there has to be at least one table or one script that checks the consistency of the attribute values. Only if attribute values are inconsistent, a new assignment has to be performed. Independently of the kind of the applied structural rule, attribute values can be propagated in any direction between the two documents. At each integrator run, the user can choose which of the defined attribute assignments and which consistency check should be used. For instance, one assignment is used at the generation of the Aspen Plus simulator input and another one is used to import the simulation results into Comos PT.

The UML-based correspondence editor allows to define correspondences between different document models on the type and (abstract) instance level. On the type level generic `relates-to` relationships can be specified in UML class diagrams to define on a coarse-grained level which of the documents' concepts relate to each other. This information can be used to check the consistency of the correspondence

definition on the abstract instance level. Here UML object diagrams are used to define static correspondences between increments and sets of attribute assignments as OCL expressions. In a next step these correspondences can be enriched with the necessary information to define multiple dynamic integration rules with the help of collaboration diagrams. The correspondence editor is still under construction and will be described in a more detailed way in forthcoming publications.

### *D. Algorithm*

This section gives a brief overview of the algorithm that is used to perform an integrator run. For a more detailed description please refer to [18].

Before an integrator run can be performed, a Comos PT document, an Aspen Plus document, a set of integration rules and–if already present–an integration document have to be selected. At the first integrator run between two documents no integration document is available so a new one has to be created.

To perform the integration, all increments that are not referenced by a link as increment or dominant increment in both documents are identified. For each of these increments, an incomplete link is created that references it but does not reference increments in the other document. The state of the resulting links is set to *unchecked inconsistent*.

In the next step, each link in the integration document is inspected. Depending on the state of the link, different actions are performed:

*Consistent* First the source and target patterns and the port assignment is mapped against the documents' structures. If the structures were not changed after the link's creation, the link is still structurally consistent. If it was changed, the link has to be deleted or marked as unchecked inconsistent. If the link is structurally consistent, the attribute assignment can be performed by using one of the specified attribute assignments. Which of the assignments has to be used can be determined by asking the user at the processing of each link, or by selecting the assignments in advance for the whole integration.

*Unchecked inconsistent* All rules that contain the increment that is referenced by the incomplete link and whose patterns match the document structure have to be identified and stored with the link. Now user interaction is necessary to chose one of these possible rules for execution. Until the user makes his decision, the link is marked as checked inconsistent.

*Checked inconsistent* If the link's dominant source increment was deleted, the link has to be deleted, too. If a user decision is available and the selected rule is still applicable, the rule is executed and the link's state is set to consistent. Otherwise the link remains inconsistent. If a link is set to consistent, attribute assignment is performed.

These steps are repeated until all links are consistent and all increments are referenced by links.

### *E. Wrapping of Applications*

As we mentioned in the requirements (see Section III) one aspect of our integration tool is the a posteriori integration of existing data sources[1]. A widely known technique to provide uniform access to heterogeneous data sources is the encapsulation of these data sources by a software component called *wrapper* [13]. A wrapper serves as an adapter which has the responsibility to convert the interface of the existing data source into an interface required by a client using this data source. Therefore each data source is linked to an appropriate wrapper (see Figure 6).

According to its task, a wrapper consists of two layers: a common external interface expected by the client, in our scenario the integrator tool, and a data source specific internal interface. In the case of a wrapped tool the internal interface could be an interface to an API (Application Programming Interface) offered by the tool, in the case of a wrapped document it could be an interface to a parser for the underlying structure of the document. The client communicates only with the wrapped data sources via the well-known external interface of the wrapper. This way, we achieve access transparency for the client with regard to the various data sources. For a new data source to be integrated by the integrator tool just the specific internal interface has to be developed.

Offering some kind of location transparency for the client is another point of interest. A data source requested by the client is identified based on a logical name rather than by exact location, such as a host name or file path. This is one aspect that leads to the concept of a mediator. A definition for a mediator is given in [17]: "A mediator is a software module that exploits encoded knowledge about … data to create information for a higher layer of application.". In the context of our architecture this task is done by the wrappers. Hence we apply the mediator more in the sense of Gamma et al. [4]: the mediator encapsulates how the client and the wrappers interact, i.e. the communication between client and wrappers takes place indirectly via the mediator. For this purpose every wrapper has to announce itself to the mediator. The mediator then associates the location of the wrapper resp. the wrapped data source with its logical name, e.g. document $d$. A client, requesting data included in document $d$, submits its query to the mediator, which forwards the query to the appropriate wrapper. The results are delivered vice versa. This type of communication is an example for synchronous communication between a client and a wrapped data source, called Request-Response (see the arrow between Integrator and Mediator in Figure 6). The integrator makes use of it, e.g., to browse the target and source documents and to modify their contents. It is the wrappers' task to translate the proprietary tool data into the integration tool's document model (see Subsection IV-

[1] In our context we use the term "data source" for the documents to be integrated as well as for the tools processing these documents.

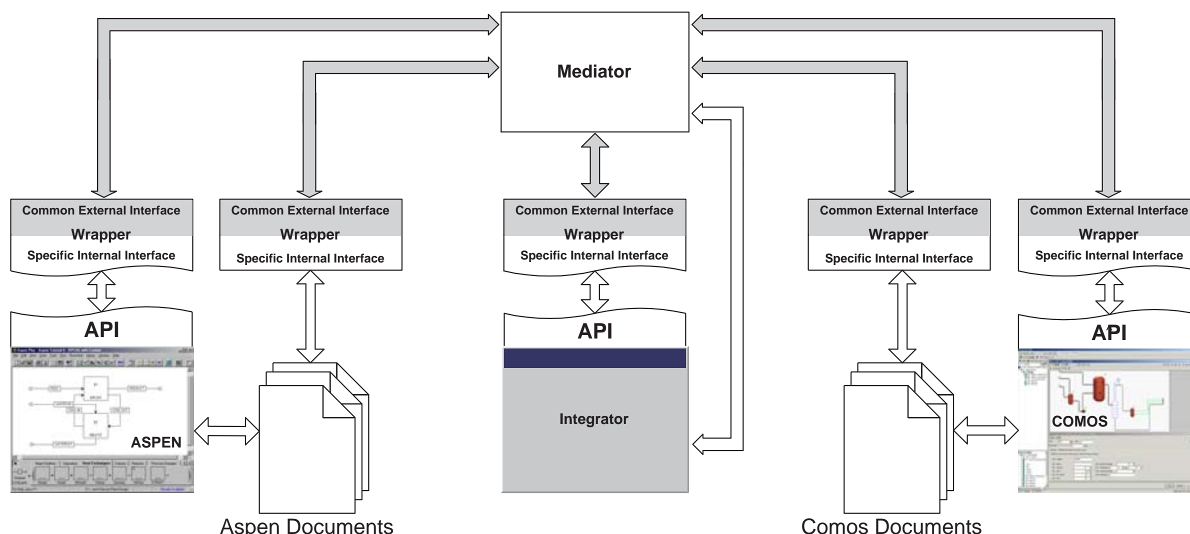


Fig. 6. Wrapper architecture

B) while performing these requests.

The mediator architecture also supports asynchronous communication, called Publish-Subscribe. For example, the integrator resp. its wrapper can subscribe to the mediator, the publisher, to receive notifications about modifications in the integrated documents. Adding or removing a block in the Aspen Plus document will cause an event, which is propagated via its wrapper to the mediator. The mediator then publishes this event to its subscribers, namely the integrator. Consequently the integrator now updates the Comos PT document, if necessary.

### *F. Implementation*

Our prototype of an integration tool for Aspen Plus and Comos PT is implemented based on a universal framework for integration tools. Until now, this framework is tailored to deal with PFD-based applications in chemical engineering but will be extended to support any applications whose data model can be mapped to graphs. The framework and the application-specific parts are largely implemented in C++. The integration document and the rule base are stored as XML files. The whole storage layer is wrapped in a universal interface so that the XML storage can be replaced easily, e.g., by a database. Comos PT and Aspen Plus are accessed by the wrappers via COM interfaces.

The graphical user interface is implemented in Visual Basic and allows to edit the integration document directly and offers a decision table to resolve ambiguities. The prototype can be integrated into the GUI of Comos PT as COM add-in. Figure 7 shows a screenshot of the prototype running inside Comos PT[2]. On the left side, the decision table offers two alternative rules for two links. On the right side, a list with all existing links is presented. Depending on the links' states, they are marked with a flash (inconsistent) or a tick (consistent). With a click on the plus sign in each link, the link's details (source and target pattern, etc.) can be shown.

[2]The screenshot shows the German version of the prototype. Though some identifiers are different, the depicted integration is a part of our running example.

## V. Related Work

Our work is concerned with *tool integration*, which can be addressed with respect to multiple dimensions [16]. We are mainly concerned with data integration (external consistency between interdependent documents), but to some extent we also deal with control integration (propagation of changes) and process integration (simultaneous engineering). Presentation integration plays a minor role. Moreover, we assume some form of platform integration (e.g., OLE [1]).

Numerous approaches to *data integration* have been developed (the following discussion does not strive for completeness):

- All data are stored in a logically centralized database [15]. However, this solution is viable only in the case of a priori integration since all tools have to agree upon a common data model.
- Tools still use their own data management facilities, but check their data in a global repository. This approach has been realized in systems for engineering/product data management [9] or software configuration management [14]. However, data integration is confined to the coarse-grained level (no fine-grained integration between the contents of different documents).
- Neutral data formats provide standards for data exchange between tools on the fine-grained level. STEP [6] addresses the standardization of data in different engineer-

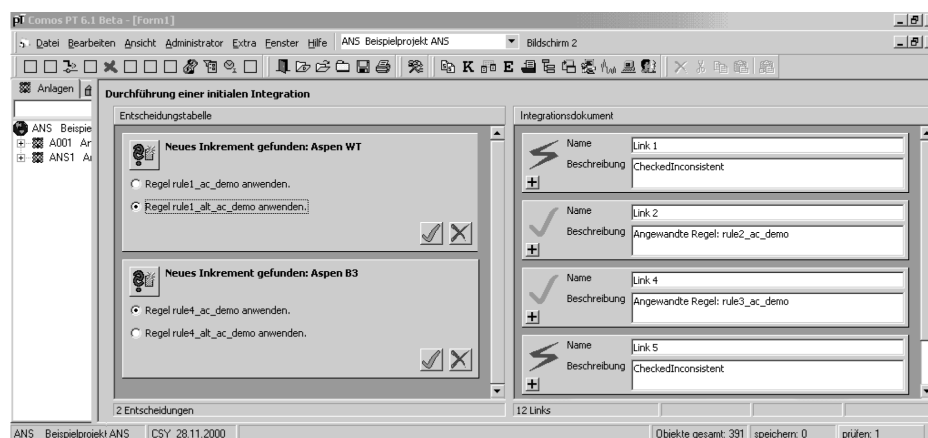


Fig. 7. Screenshot of the integration tool running as Comos PT plug-in

ing domains. Data integration is supported through the exchange of text files in STEP format. XML [5] has also become popular for data exchange. Data integration is typically supported through batch converters between the neutral data format and proprietary data formats of the respective tools. In contrast, our integration tools support incremental change propagation.

The work described in this paper is based on previous projects carried out in the software engineering domain. Within the IPSEN project, we built incremental tools for integrating requirements definitions, software architectures, programs, and technical documentations (see [8] and [10], Chapter 2). However, tools were integrated a priori, i.e., a common user interface, database, etc. was assumed. Moreover, integration tools were hardcoded. In contrast, our current work addresses a posteriori integration. Rules can be defined on the fly, and they can be interpreted or compiled as desired.

*Triple graph grammars* [11] provide the theoretical background of our work. This approach was also applied in the REforDI project [2], which was concerned with re-design of legacy applications, and in Varlet [7], which addressed database reengineering. In both projects, integration tools were specified in PROGRES, a specification language for programmed graph transformations [12]. Integration tools were implemented by rapid prototyping (generation of code from the specification).

There are the following differences to the work described in this paper: First, integration rules may be defined and added to the rule base on the fly, while a separate compilation step was needed in REforDI and Varlet. Second, integration rules may be defined by end users (engineers), while they had to be defined by experts being familiar with the PROGRES language in REforDI and Varlet. Third, a posteriori tool integration was not addressed in REforDI and Varlet. Finally, both REforDI and Varlet provided specific integration tools, while we strive for a general framework for building integration tools (partly based on [8]).

## VI. Conclusion

We have presented an integration tool supporting cooperative development processes in chemical engineering. This tool is based on a general infrastructure for building incremental integration tools. We are currently completing the implementation of the integration tool between Comos PT and Aspen Plus. We are intending to validate our work in an industrial context (in cooperation with innotec).

Future work will first address several extensions with respect to the support of PFD-based tools (e.g., hierarchical PFDs, integration of other simulation tools). Moreover, we will study other application domains as well. As a result, we expect the framework for building integration tools to evolve such that it covers more general types of graphs (so far, we assume a model tailored towards PFD-based applications).

Acknowledgements. We are indebted to Jochen Schüler, Bernd Kokkelink, and Marcus Elo from innotec for their constant support and encouragement. Furthermore, Birgit Bayer (LPT, Aachen University of Technology) provided us with the example presented in Section II.

## References

[1] K. Brockschmidt. *Inside OLE*. Microsoft Press, Redmond, Washington, 1995.
[2] K. Cremer. A tool supporting the re-design of legacy applications. In P. Nesi and F. Lehner, editors, *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering*, pages 142–148, Florence, Italy, Mar. 1998. IEEE Computer Society Press.
[3] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, volume 2. World Scientific, Singapore, 1999.
[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Software*. Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1995.
[5] C. Goldfarb and Prescott. *The XML Handbook*. Prentice Hall, Englewood Cliffs, New Jersey, 1998.
[6] ISO, Geneve, Switzerland. *ISO 10303: Product Data Representation and Exchange*.
[7] J. Jahnke and A. Zündorf. Applying graph transformations to database re-engineering. In Ehrig et al. [3], pages 267–286.

[8] M. Lefering. *Integrationswerkzeuge in einer Softwareentwicklungsumgebung*. Shaker, Aachen, Germany, 1995. In German.

[9] K. G. McIntosh. *Engineering Data Management — A Guide to Successful Implementation*. McGraw-Hill, Maidenhead, England, 1995.

[10] M. Nagl, editor. *Building Tightly-Integrated Software Development Environments: The IPSEN Approach*. LNCS 1170. Springer-Verlag, Berlin, Germany, 1996.

[11] A. Schürr. Specification of graph translators with triple graph grammars. In E. Mayr, G. Schmidt, and G. Tinhofer, editors, *Proceedings WG '94 Workshop on Graph-Theoretic Concepts in Computer Science*, LNCS 903, pages 151–163, Herrsching, Germany, June 1994. Springer-Verlag.

[12] A. Schürr, A. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In Ehrig et al. [3], pages 487–550.

[13] H. M. Sneed. Encapsulation of legacy software: A technique for reusing legacy software components. *Annals of Software Engineering*, 9(1-4):293–313, 2000.

[14] W. F. Tichy, editor. *Configuration Management*, volume 2 of *Trends in Software*. John Wiley & Sons, New York, 1994.

[15] L. Wakeman and J. Jowett. *PCTE — The Standard for Open Repositories*. Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[16] A. Wasserman. Tool integration in software engineering environments. In F. Long, editor, *Proceedings of the 2nd International Workshop on Software Engineering Environments*, LNCS 467, pages 137–149, Chinon, France, Sept. 1990. Springer-Verlag.

[17] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, Mar. 1992.

[18] J. Wilhelms. Inkrementelle Integratoren zur Kopplung verfahrenstechnischer Entwicklungswerkzeuge. Master's thesis, Aachen University of Technology, Aachen, Germany, Mar. 2002. In German.