Softw Syst Model
DOI 10.1007/s10270-006-0045-5

SPECIAL SECTION PAPER

A graph-based algorithm for consistency maintenance in incremental and interactive integration tools

Simon M. Becker · Sebastian Herold · Sebastian Lohmann · Bernhard Westfechtel

Received: 19 February 2005 / Revised: 31 October 2006 / Accepted: 14 December 2006

Abstract Development processes in engineering disciplines are inherently complex. Throughout the development process, the system to be built is modeled from different perspectives, on different levels of abstraction, and with different intents. Since state-of-the-art development processes are highly incremental and iterative, models of the system are not constructed in one shot; rather, they are extended and improved repeatedly. Furthermore, models are related by manifold dependencies and need to be maintained mutually consistent with respect to these dependencies. Thus, tools are urgently needed which assist developers in maintaining consistency between inter-dependent and evolving models. These tools have to operate incrementally, i.e., they have to propagate changes performed on one model into related models which are affected by these changes. In addition, they need to support user interactions in settings where the effects of changes cannot be determined automatically and deterministically. We present an algorithm for incremental and interactive consistency maintenance which meets these requirements. The algorithm is based on graphs, which are used as the data model for representing the models to be integrated, and graph transformation rules, which describe the modifications of the graphs to be performed on a high level of abstraction.

Keywords Incremental consistency maintenance · Graph transformation · Triple graph grammars

1 Introduction

1.1 Background

Development processes in engineering disciplines are inherently complex. Throughout the development process, the product to be developed is modeled from different perspectives, on different levels of abstraction, and with different intents. In general, a *model* is an abstraction of some real world object; here, a model refers to the product which is the subject of engineering. For example, in software engineering models are created to represent the requirements, the design, or the implementation of a software system; furthermore, we may distinguish between structural and behavioral models.

Models are related by manifold *dependencies*. For example, the design of a software system depends on its requirements, and the implementation depends on the design. A model m_1 depends on some model m_2 if the contents of m_1 are constrained by rules which relate elements of m_1 to elements of m_2. If these constraints are satisfied, m_1 is said to be consistent with m_2. With respect to consistency, we have to distinguish between

Communicated by Dr. Francesco Parisi-Presicce.

This paper is an extended version of [6].

S. M. Becker (✉) · S. Herold · S. Lohmann
RWTH Aachen University, Computer Science III,
Ahornstr. 55, 52074 Aachen, Germany
e-mail: sbecker@i3.informatik.rwth-aachen.de

B. Westfechtel
University of Bayreuth, Applied Computer Science I,
95440 Bayreuth, Germany
e-mail: Bernhard.Westfechtel@uni-bayreuth.de

S. Herold
e-mail: herold@i3.informatik.rwth-aachen.de

S. Lohmann
e-mail: slohmann@i3.informatik.rwth-aachen.de

intra-model consistency, which refers to the correctness of the contents of some model with respect to local rules, and *inter-model consistency*, which refers to the correctness of the relationships between inter-dependent models with respect to cross-model rules. In the context of this paper, we will focus on inter-model rather than on intra-model consistency.

Development processes may be viewed as multiphase *transformation processes* from the initial problem statement to the final solution. However, this is a simplified view suggesting a waterfall-like process, where each phase is entered only when the preceding phase has been completed. In contrast, current development processes are highly incremental and iterative. In an *incremental* development process, the product is not developed in one shot; rather, it is developed in smaller parts called increments. In an *iterative* process, activities are executed repeatedly until the goals of development are reached eventually. Furthermore, development does not always proceed in forward direction (*forward engineering*). Rather, it may also involve activities working in backward direction (*reverse engineering*). Combining forward and reverse engineering results in *round-trip engineering*, where developers opportunistically mix both modes of development.

Maintaining inter-model consistency is a demanding task which requires sophisticated tool support. In this paper, we will subsume all kinds of tools for maintaining inter-model consistency under the notion of *integration tool*. These tools may be classified as follows:

- A *transformation tool* processes a source model and transforms it into a target model.
- A *consistency analysis tool* takes inter-dependent models and checks inter-model consistency.
- A *hyperlink tool* creates and maintains links between elements of inter-dependent models.
- A *browsing tool* traverses inter-model links.

Transformation tools have been studied extensively. For example, consider the *Model Driven Architecture* [32] initiative launched by the OMG. The mission of MDA is to generate platform-specific code from architectural models. Often, transformation tools of this kind are run in batch mode, processing the whole architectural model without requiring any user interactions. In this way, development effort may be reduced significantly.

Batch transformers are not always adequate, which is also recognized in the context of MDA [23].[1] For example, consider the transition from requirements to design, which involves moving from the problem space to the solution space. The design cannot be generated automatically from the requirements. Rather, the designer has to make deliberate *design decisions* which balance requirements such as efficiency, adaptability, portability, etc. Therefore, the design process is performed *interactively*. Furthermore, design usually proceeds *incrementally* and *iteratively*. As a consequence, integration tools which assist in maintaining consistency between requirements and design need to operate both interactively and incrementally. Moreover, tool support has to provide for *traceability*, i.e., each element of the design has to be traced back to its originating elements in the requirements.

1.2 Contribution

In this paper, we focus on incremental and interactive integration tools for inter-model consistency maintenance. A tool is called *incremental* if the effort to process a change is proportional to the size of the change. In other words: An incremental tool is designed to process small changes with low effort. The unit of change is called *increment*, which corresponds to some syntactic unit of the underlying modeling language. These definitions are drawn from the tool building literature [29]; consider e.g. incremental parsers or syntax-aided editors with incremental context-sensitive analysis. In contrast, in the context of incremental development processes an increment denotes a unit of work, whose size depends on the characteristics of the development process being executed.

Incremental tools are not necessarily *interactive*; consider e.g. incremental parsers operating silently in the background. In contrast, we focus on scenarios where user interaction is *inherently* required (as e.g. for the integration between requirements engineering and design). In such scenarios, user interaction cannot be avoided because not all decisions can be automated. Involving the user implies that the user can control the integration process. If automation is pushed too far, the results produced by non-interactive integration tools might be of limited value. Of course, there are other scenarios where automated tools are desired, such as e.g. code generators in model-driven development. But this is not the focus of the research presented in this paper.

To provide incremental and interactive integration tools for inter-model consistency maintenance, *links* are required which connect increments of inter-dependent models. These links are used for browsing to switch back and forth between inter-dependent models.

[1] In the cited reference, Chap. 7 lists desirable features of transformation tools which closely match the features listed below.

Furthermore, they are used to record design decisions and to propagate changes. Finally, links provide for traceability of the development process.

Inter-model links can be realized in various ways, e.g., by maintaining bidirectional pointers which are stored with the models to be integrated. In contrast, we maintain a separate data structure called *integration model*. In general, a link stored in the integration model relates sets of increments in the respective inter-dependent models. Thus, links may express $m : n$ relationships in the general case. Furthermore, links may be annotated with additional information, and we may even store relationships between links. Altogether, the integration model is much more powerful than a realization by bi-directional pointers, which has been ruled out because of its inherent limitations.

Another reason for storing links in a separate data structure is *a posteriori integration* of existing tools. In the case of heterogeneous tools, each tool maintains its own data base, which may not be extensible by external applications. In such a situation, it is not possible to store bidirectional pointers with the data of the tools.

The key contribution of this paper lies in a novel *algorithm* for consistency maintenance in incremental and interactive integration tools. The algorithm is driven by rules which relate patterns in inter-dependent models. Here, the term *pattern* refers to a set of increments and their connecting relationships. In contrast to previous approaches, rule execution is split into multiple steps. First, the algorithm searches for applicable rules and looks for *conflicts* among these rules. Informally, two rules stand in conflict if the execution of one rule disables the execution of the competing rule. Conflicts are presented to the user, who performs a selection among the conflicting rules. Only those rules which do not participate in any conflict are executed automatically (i.e., without further user interaction). This approach requires to separate pattern matching from rule execution. In contrast, in previous work (e.g. [11,20]) rules have been executed atomically (but not necessarily automatically) without making the user aware of conflicts among the offered rules.

The integration algorithm is based on *graphs*. Graphs consist of nodes representing objects and edges representing relationships between these objects. Graphs constitute the common meta model in which the inter-dependent models and the integration models are expressed. We use graphs because they are well suited for representing complex data with manifold relationships in a natural way. Furthermore, we make use of *graph transformation rules* [33] for describing modifications of these graphs on a high level of abstraction. Since graph transformation rules are executable, an implementation of the algorithm may be created by rapid prototyping. In fact, we have realized and evaluated the integration algorithm in *IREEN*, an *I*ntegration *R*ule *E*valuation *EN*vironment [27]. Additionally, a more industry-related framework for building real-world integration tools is being implemented based on the lessons learned from IREEN (cf. Sect. 9).

1.3 Structure

Section 2 presents a scenario which motivates our work by a practical example. Section 3 is devoted to the graph-based specification of integration tools. Sections 4 to 9, the core part of this paper, present our novel approach to rule execution. Section 10 discusses related work, and Sect. 11 presents a short conclusion.

2 Scenario

2.1 Context of research

The research presented in this paper is *domain-independent*, i.e., it can be applied to different engineering disciplines. In the introduction, we referred to software engineering, particularly to the relationships between requirements and design. We studied these relationships in the *IPSEN* project, which was concerned with tightly integrated, incremental, and graph-based software engineering environments [29]. Within the IPSEN project, an integration tool for maintaining consistency between requirements and design was developed [25,26] which may be viewed as a precursor of the IREEN prototype described in this paper.

Our current research is carried out within the *IMPROVE* project [30], which is concerned with models and tools for design processes in *chemical engineering*. IMPROVE is an interdisciplinary research project which is performed by computer scientists, chemical engineers, and plastics engineers at the RWTH Aachen University, Germany. IMPROVE cooperates closely with industrial partners in order to validate research concepts and to put research into practice. The research reported in this paper has been conducted in the subproject of IMPROVE which is dedicated to the development of integration tools. To this end, a cooperation was established with *innotec*, a German software company which provides solutions for chemical engineering. In the context of this cooperation, an integration tool was developed which assists in maintaining consistency between flow sheets and simulation models [5]. This tool is based on a general framework for building integration tools. The integration algorithm described here is part of this framework.

2.2 Development processes in chemical engineering

Development processes in chemical engineering are divided roughly into two parts: basic engineering and detail engineering. In *basic engineering*, the basic decisions concerning the design of the chemical plant are made. In *detail engineering*, the design is refined to a blueprint of the chemical plant to be built. In the sequel, we will focus on a small cutout of basic engineering. We will be concerned with design and simulation and will ignore other activities such as requirements analysis, cost estimation, and laboratory experiments.

In chemical engineering, the *flow sheet* plays a central role. A flow sheet is composed of *devices* (reactors, separation units, etc.) and *streams* representing the flow of chemical substances between devices. All elements of the flow sheet are typed. For example, there are different types of reactors such as plug flow reactors or continuous stirring reactors. Furthermore, elements are decorated by attributes. For example, a reactor may be described by attributes such as height and dimension, and a stream is characterized by some material and its flow rate. In basic engineering, many details are still left unspecified, e.g., the layout of the plant to be built. The flow sheet is refined later in detail engineering. As far as basic engineering is concerned, the flow sheet primarily serves to design the chemical process and the overall composition of the chemical plant.

Primarily, the flow sheet constitutes a structural model. However, since it is used as a central overview diagram, behavioral data are collected in the flow sheet, as well. These data comprise temperature, mass flows, heat flows, pressure profile, molecular weight, etc. Behavioral data are obtained by performing laboratory experiments, analytically by solving mathematical equations, and by simulations. Altogether, the flow sheet is used by chemical engineers as the central model which describes both the structure and the behavior of the chemical plant to be built. All crucial design decisions are performed with the help of and are documented in the flow sheet.

As mentioned, the behavior of the chemical process may be determined in multiple ways. In this paper, we will focus on *simulations*, which are based on mathematical models of the chemical process. These models are defined as sets of differential equations, which usually are too complicated to be solved analytically. Simulations are classified further into *steady-state* simulations, which describe the behavior of the chemical process in its equilibrium state, and *dynamic* simulations, which are used e.g. to determine the start-up behavior and the behavior in the case of technical faults.

Simulations are based on *simulation models*. How these models are structured, depends heavily on the respective simulation tool. In the worst case (not to be discussed further), the simulation expert has to develop the simulation model in terms of differential equations on his own. However, many simulation tools reduce the effort of developing simulation models considerably by providing pre-defined blocks. Simulators of this kind are called *block simulators*. The simulation expert composes a simulation from pre-defined blocks and connects them by streams. In a block simulation model, a stream defines a connection between the outputs of the source block and the inputs of the target block. This means that the outputs of the source block are set equal to the inputs of the target block. How the blocks are structured internally, is (intentionally) hidden from the users of the simulation tool.

Performing simulations of the chemical process is not straightforward. Usually, simulations are performed only for parts of the process, which are defined in connected *regions* of the flow sheet. Please note that the structure of a region in the flow sheet may deviate from the structure of the corresponding simulation model. In the case of a block simulator, the simulation model is composed of pre-defined blocks, which may or may not correspond to devices contained in the flow sheet. For example, multiple blocks may have to be composed in order to simulate a single device of the flow sheet.

The notion of *consistency* between a flow sheet and a simulation model is not straightforward to define; in this case, an "official" definition of consistency does not exist. There are some intuitive rules of thumb, e.g., a device may be mapped onto a block, or a stream in the flow sheet may be mapped onto a stream in the simulation model. Furthermore, each element of the region to be simulated has to mapped onto some element of the corresponding simulation model (and vice versa). However, as explained above, mappings are not always 1:1.

Therefore, we do not assume that consistency is defined once and for all in a universally accepted way. Rather, consistency is defined in terms of a set of *rules* which is extensible and modifiable. The rule base is fed by simulation experts in an iterative process. However, the knowledge acquisition process goes beyond the scope of this paper; therefore, the reader is referred to [4]. In this paper, we will assume that the rule base is given and static.

In basic engineering, the development process is highly *iterative*. Typically, the flow sheet is designed first.

After simulations have been performed, the simulation results are propagated back to the flow sheet. Based on the simulation results, the flow sheet may have to be extended or modified, and new simulations may have to be performed. This iterative process terminates when the requirements to the chemical process are eventually satisfied. Please note that the development process may also involve reverse engineering. For example, it may happen that an initial version of the flow sheet is derived from a pre-existing simulation model. Furthermore, a simulation expert may decide to extend or modify the simulation model and propagate these structural changes back into the flow sheet. Altogether, this implies a tight change propagation cycle between the flow sheet and the various simulation models.

2.3 Sample process

The sample process described below was designed in cooperation with chemical engineers and the software company *innotec*. In the cooperation with *innotec*, we built an integration tool for coupling *COMOS PT*, an environment for chemical engineering developed by *innotec*, and *Aspen Plus*, a commercial block simulation tool. Among other tools, *COMOS PT* provides a flow sheet editor. The integration tool maintains consistency between flow sheets prepared with *COMOS PT* and simulation models created in *Aspen Plus* (cf. Sect. 9).

The integration between *COMOS PT* and *Aspen Plus* was performed *a posteriori*, i.e., existing tools were integrated which were not designed for integration. Each tool stores models in *documents* which are owned by the tools. Links between these documents are stored in a separate *integration document*.

The design process refers to a simple chemical process which produces ethanol from ethen and water. For this simple process, we assume that there is no need to define different regions for different simulations, i.e., we assume that the region for the simulation model covers the complete flow sheet.

Figure 1 shows the flow sheet and the simulation model in different stages of development. Typically, flow sheets and simulation models are created by different users at different times with the help of respective tools; an integration tool is used to establish mutual consistency on demand. The design process consists of four steps:

1. An initial version of the simulation model is created.
2. An initial version of the flow sheet is derived from the simulation model (reverse engineering).
3. The flow sheet is extended with further elements. In parallel, the simulation is performed for the simulation model corresponding to the old version of the flow sheet (concurrent engineering).
4. Consistency between the flow sheet and the simulation model is re-established by propagating changes in both directions.

Flow sheets consist of devices and streams, which are represented in Fig. 1 as boxes and arrows, respectively. In the diagrams for the simulation model, blocks are shown as graphical icons indicating their types; streams connecting the blocks are displayed as arrows. Finally, inputs and outputs are marked by graphical arrows. All elements of the flow sheet and the simulation model are decorated with attributes, which have been omitted from the figure for the sake of readability.

The integration tool maintains a data structure which contains links between elements of the flow sheet and elements of the simulation model. These links are represented by ellipses which are located on the dashed line separating the flow sheet from the simulation model. Dotted lines are used to indicate the elements participating in a link. Furthermore, arrows located

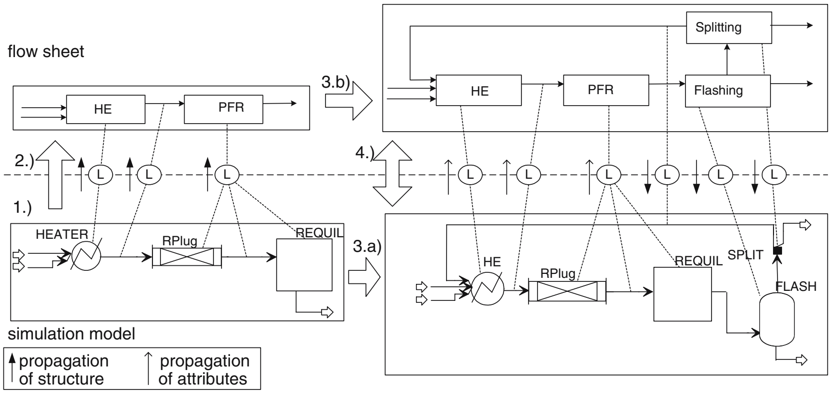


Fig. 1 Integration between flow sheet and simulation model

on the left indicate the directions of change propagations (structural and attribute changes, respectively) which are performed in the various steps of the design process.

In the sample design process, two users are involved: The *design expert* is responsible for the flow sheet, the *simulation expert* creates the simulation model and runs simulations. The steps of the design process are performed as follows:

Step 1. Initially, we assume that the simulation expert has already created a simulation model for a part of the chemical process (heating and reaction). The simulation model is composed of three blocks according to the capabilities of the respective simulation tool (please recall that the blocks are pre-defined by the simulation tool).

Step 2. The design expert transforms the simulation model into the flow sheet with the help of the integration tool. Multiple alternatives are available for this transformation. It turns out that the simplest one—a 1:1 transformation—does not result in an adequate flow sheet because the blocks do not correspond 1:1 to devices in the flow sheet. Rather, the design expert decides to group two blocks and their connecting stream into a single device (a plug flow reactor) in the flow sheet. The link between the PFR and the respective parts of the simulation model is established by firing a corresponding integration rule. In addition, another rule is available which just transforms the block called RPlug into a PFR. This 1:1 rule stands in conflict with the rule selected here: Applying one rule disables the competing rule. The integration tool presents conflicting rules to the user who may select the rule to be applied.

Step 3. Steps 3a and 3b are carried out in parallel, using different tools. Using the simulation model created so far, the simulation expert runs the simulation. The simulation results comprise flow rates, temperatures, etc. In parallel, the design expert uses the flow sheet editor to extend the flow sheet with the chemical process steps that have not been specified so far (flashing and splitting).

Step 4. Finally, the integration tool is used to synchronize the parallel work performed in the previous step by re-establishing consistency between flow sheet and simulation model. This step is performed cooperatively by the design expert and the simulation expert in a joint working session. Synchronizing the flow sheet and the simulation model involves information flow in both directions. First, the attributes containing the simulation results are propagated from the simulation model back to the flow sheet.[2] Second, the extensions are propagated from the flow sheet to the simulation model. In this way, mutual consistency is re-established.[3]

2.4 Requirements to integration tools

From this example, we derive several features of the kinds of integration tools we are addressing:

Functionality. An integration tool must manage links between elements of inter-dependent models. In general, links may be m:n relationships, i.e., a link connects m source elements with n target elements. 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 inter-dependent models. 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 model m_1 is changed, the changes are propagated into model m_2 and vice versa.

Mode of interaction. An integration tool may operate automatically in simple scenarios. However, user interactions are required in the case of non-deterministic transformations.

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

Conflict detection. In the case of non-deterministic choices, the integration tool must detect conflicting rules and report conflicts to the user. Otherwise, the user would have to apply a rule without being aware of conflicting alternatives.

Time of activation. In the case of reactive integration, the integration tool performs incremental transformations and consistency checks on each user command. In contrast, integration on demand means that the integration tool is explicitly activated by the user at appropriate points of time. In our work, we focus on integration on demand as it is appropriate in multi-user scenarios as presented above.

[2] For a description of the attribute propagation mechanism please refer to [5].

[3] Please note that Fig. 1 displays the simulation model after Step 4. The results of Step 3a are not visualized.

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.

3 Graph-based specification of integration tools

Integration tools are based on a formal specification which employs graphs as the underlying data model and graph transformation rules to specify operations on graphs on a high level of abstraction. Since graph transformation rules are executable, code may be generated from the formal specification. In this way, prototypes of integration tools may be generated rapidly.

The current section introduces the formal foundations of integration tools. First, a brief introduction into graph grammars and graph transformation systems is given. Then, we discuss a graph-based formalism which has been designed specifically for solving integration problems.

3.1 Graph grammars and graph transformation systems

Graphs are well suited for modeling complex structured data. In the context of this paper, we use graphs to represent the models to be integrated (the flow sheet and the simulation model in our running example) and the integration model placed in between these models. Please note that the diagrams of Fig. 1 are based on a graphical notation designed for the end user which differs from the internal representation introduced below. The integration algorithm to be described in subsequent sections requires a formal graph representation. How such a graph representation is actually provided, goes beyond the scope of this paper and is described elsewhere [4]. The basic idea is to realize a graph view on the data structures of the tools (flow sheet editors and simulators) by means of wrappers.

Graphs may be defined in many different ways. Here, we are dealing with directed, typed, and attributed graphs. A *graph* has a finite set of nodes which are identified uniquely by node numbers. Edges are binary relationships between nodes. A *directed* graph contains edges whose ends are distinguished as *source* and *target*, respectively. This distinction does not imply that edges can be traversed only from source to target; in contrast, edges may be traversed in both directions. In a *typed* graph, both nodes and edges are typed. In an *attributed graph*, the elements of graphs may be decorated with attributes. In the graph data model which we assume, attributes may not be attached to edges; furthermore, duplicate edges of the same type are not allowed. Therefore, edges are represented as triples (sn, tn, et), where sn, tn, et denote source node, target node, and edge type, respectively. In the sequel, directed, typed, and attributed graphs are briefly called graphs.

Please note that types of nodes and edges as well as attributes have to be defined in a *graph schema*, i.e., a database schema for graphs. Due to the lack of space, this is not further explained.

Figure 2 displays a cutout of the graph representation of the informal diagram shown in Fig. 1. Please note that this is an *internal graph representation*, i.e., it is not intended to show this graph to the end user. Rather, end users operate on diagrams similar to those shown in Fig. 1. The graph of Fig. 2 refers to the plug flow reactor (PFR) in the flow sheet, the corresponding cascade of blocks in the simulation model, and the link structure in between.

In Fig. 2, nodes and edges represent increments and relationships, respectively. Nodes and edges are drawn as boxes and arrows, respectively. Edge types are placed on the arrows for the edges. The box for a node contains its node number (any unique identifier is permissible, not just natural numbers) and its type. Attributes are omitted from the figure.

Let us now describe the elements of the graph representation, referring back to the diagram of Fig. 1:

- The plug flow reactor of the flow sheet and the corresponding blocks in the simulation model are represented as nodes (`f2`, `a2`, and `a10`, respectively).
- The stream connecting the simulation blocks is also represented as a node (`a6`) even though it is displayed as an arrow in the diagram. The stream cannot be represented as an edge because it has to be decorated with attributes (e.g., material flow or temperature).
- Both the flow sheet and the simulation model are based on a common port-based meta model. A *port* defines a connection point of some element and is represented by a node (e.g., `f1`). Ports are classified into *input ports* and *output ports*. For example, for a reactor ports define connection points for incoming and outgoing material flows; for streams, they define their ends and the flow direction. Each port is *owned* by the element to which it is attached. The ownership is expressed by edges from element nodes to port nodes.
- *Connections* are used to link ports of different elements. Each connection is represented by a node (e.g., `a4`) and two edges pointing to the connected ports. In the informal diagrams of Fig. 1, neither ports nor connections are represented explicitly.

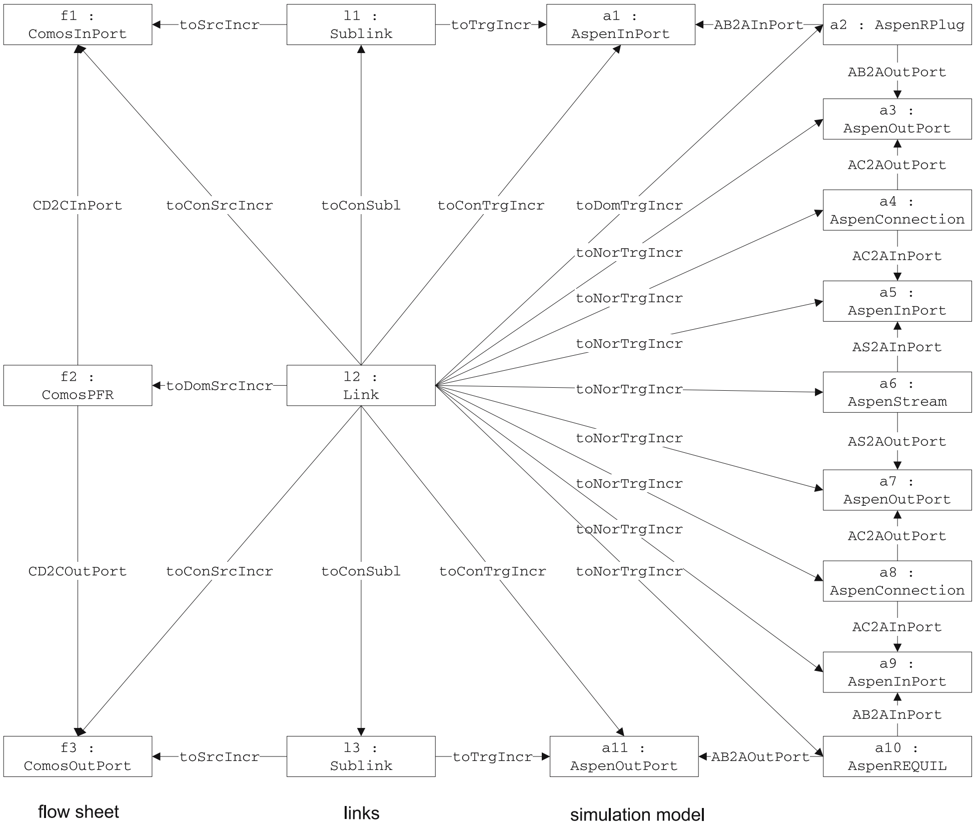


Fig. 2 Graph representation of a cutout of Fig. 1

- Finally, links are represented by *link nodes* (`l2`). A link node is connected to each node of the linked subgraphs by an edge. These edges are classified into different types which will be explained later. Primarily, links connect increments of flow sheets and simulation models, respectively. Additional *sublink nodes* (`l1`, `l3`) define correspondences between ports, which in general cannot be deduced uniquely from correspondences between elements. In our example, the outer ports of the cascade of simulation blocks are linked to the input port and the output port of the plug flow reactor.

A *graph grammar* is a generalization of a string grammar, i.e., it generates graphs instead of strings. The first papers on graph grammars were published around 1970. An overview of the theoretical foundations is given in [33]; applications of graph grammars are described in [13]. In a graph grammar, graphs are generated from a start graph by applying *productions*. A production consists of a left-hand side and a right-hand side, both of which are graphs. A production is applicable to a host graph when a match of its left-hand side is found. Applying the production means that the match of the left-hand side is replaced with a copy of the right-hand side.

A *graph transformation system* differs from a graph grammar since it is intended to transform graphs rather than to generate graphs. Therefore, a graph transformation system does not distinguish between terminal and non-terminal symbols. In a graph transformation system, *graph transformation rules* are applied to modify a host graph. While productions are used to generate graph structures, graph transformation rules may also specify deletions.

In the sequel, we will generally use the term “graph transformation rule” independently of whether we refer to graph grammars or to graph transformation systems.

Graph grammars and graph transformation systems will be discussed more concretely in the next subsections. As a notation for graph grammars and graph transformation systems, we will use the specification language *PROGRES* [35]. We have specified our graph algorithm for incremental and interactive consistency maintenance with the help of the PROGRES development environment, which offers - among other tools such as a syntax-aided editor and an interpreter - a compiler which generates code from the specification. The features of the specification language PROGRES will be explained on the fly throughout the presentation of the graph-based integration algorithm.

3.2 Triple graph grammars

The graph of Fig. 2 consists of three subgraphs corresponding to the flow sheet, the simulation model, and the integration model, respectively. The intermediate link data structure is required in the case of complex integration problems, where *m* : *n* correspondences have to be maintained, applied transformation rules have to be recorded, and dependencies between links or transformation rules have to be represented.

For the specification of sophisticated integration tools, a customized variant of graph grammars called *triple graph grammars* was developed [34]. A triple graph grammar deals with interrelated graph structures called *source graph*, *target graph*, and *integration graph*, respectively. The terms "source" and "target" denote distinct ends of the relationship between the models to be integrated, but this does not necessarily imply a unique direction of transformation (in fact, transformations are performed in both directions in our sample scenario).

The core idea behind triple graph grammars is to specify the relationships between source, target, and integration graph declaratively by *triple rules*. A triple rule defines a coupling of three rules operating on source, target, and integration graph, respectively. When a triple rule is applied, the coupled graphs are modified synchronously, taking their mutual relationships into account. Triple graphs generated from a start graph by applying triple rules are mutually consistent by definition, i.e., the triple rules define what inter-model consistency means.

Originally, a special-purpose formalism was introduced for triple rules in [34]. Instead, we specify triple rules as ordinary PROGRES rules. An example of a triple rule in PROGRES notation is given in Fig. 3. This rule describes the creation of corresponding connections, which are inserted synchronously into the source graph and the target graph and are connected via a link node in the integration graph. Throughout the rest of this paper, this rule will be used as a running example.

The rule is specified in PROGRES syntax (with the exceptions of the comments in italic, which were added for the sake of readability). In the first line, the rule is given a name. The dashed boxes contain the left-hand side and the right-hand side, respectively.[4] The left-hand side consists of nodes, which are numbered in a unique way, and edges, both of which are typed. Since this rule specifies an extension of the host graph, the right-hand side repeats all of these nodes, using the notation `i' = 'i`. Likewise, their connecting edges are repeated on the right-hand side. Furthermore, the right-hand side contains new nodes (notation `i' : type`, where `type` denotes the type of the new node) and edges.

The left-hand side is composed of port nodes in source and target graph, distinguishing between output ports and input ports.[5] Furthermore, it is required that the port nodes in both graphs correspond to each other. This requirement is expressed by the nodes of type `Sublink` in the integration graph and their outgoing edges, which point to nodes of the source and target graph, respectively. Port correspondences are established by other triple rules which transform the elements the ports belong to.

All elements of the left-hand side re-appear on the right-hand side. New nodes are created for the connections in source and target graph, respectively, as well as for the link between them in the integration graph. The connection nodes are embedded locally by edges to the respective port nodes. For the link node, the following types of adjacent edges are distinguished:

- `toDomSrcIncr` and `toDomTrgIncr` edges are used to connect the link to exactly one *dominant increment* in the source and target graph, respectively. In our example, the connections act as dominant increments.
- In general, the source and target pattern related through the triple rule may consist of more than one increment in each participating graph. Then, there are additional edges to *normal increments* (`toNorSrcIncr` and `toNorTrgIncr`; not needed in our running example).[6]
- `toConSrcIncr` and `toConTrgIncr` edges point to nodes which define the context for the new increments. These nodes are called *context increments*. In our example, the ports being connected serve as context increments.
- If a link is composed of sublinks, the composition relationships are expressed by edges of type `ToSubl`. These do not appear in our sample rule since the sublinks occurring here serve as context and are owned by other links for mapping devices or streams with respective ports.
- `ToConSubl` edges are used to connect a link to sublinks in its context (which are in turn connected to the related increments in source and target graph by `ToSrcIncr` and `ToTrgIncr` edges, respectively). In our example, sublinks between corresponding ports appear in the context of the link between

[4] For layout reasons, the left-hand side is shown above the right-hand side.

[5] Only ports of different orientation may be connected.

[6] The distinction between dominant and normal increments is not vital, but helpful for pragmatic reasons; see next section.

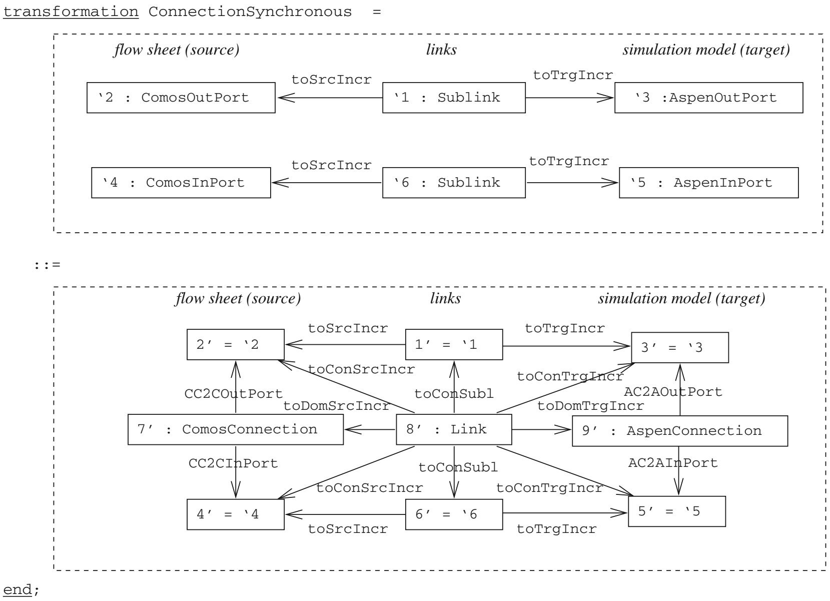


Fig. 3 Triple rule for a connection

connections. ToConLnk edges connect a link to links in its context but are not used here.

The triple graph grammar for the integration between flow sheets and simulation models contains further rules dealing with devices, streams, and ports. These rules are elementary inasmuch as they define 1:1 mappings between single elements of flow sheets and simulation models, respectively. In addition, there are rules for mapping composite patterns which may define *m*:*n* mappings in general. For example, in Fig. 2 a mapping is defined between the plug flow reactor in the flow sheet and a cascade of two blocks in the simulation model. It is a straightforward task to construct a triple rule from the graph of Fig. 2. Due to space restrictions, however, the following presentation is restricted to the running example shown in Fig. 3 (creation of a connection).

3.3 Rule execution in the original TGG approach

Triple graph grammars are *generative*, i.e., they describe mutually consistent triple graphs. Each triple graph which may be generated from the start graph by applying the triple rules consists of three mutually consistent subgraphs. Thus, a given triple graph may be checked for consistency by solving the word problem of the triple graph grammar. This could be done, but has not been the focus of our work.

For our purposes, triple rules cannot be executed as they stand. Our integration tools are designed to propagate changes between inter-dependent models after these have been modified in a potentially asynchronous way. In contrast, a triple rule such as given in Fig. 3 describes a *synchronous graph transformation*. Furthermore, the triple rule could be applied to any pair of corresponding ports. This is not desired for incremental integration tools for consistency maintenance. Instead, rules are desired which create a corresponding connection in the target model when a connection has been added to the source model and vice versa. When one connection is already present, rules of this kind operate in a deterministic way (while the synchronous rule would insert connections more or less randomly).

In [34], it is described how *asynchronous integration rules* may be derived from a synchronous triple rule by adding nodes and their edges to its left-hand side:

– A *forward rule* assumes that the source graph has been extended, and extends the integration graph and the target graph accordingly. Thus, the forward rule derived from our sample rule (Fig. 3) additionally contains node 7 on the left-hand side.
– Analogously, a *backward rule* is used to describe a transformation in the reverse direction. In our example, node 9 is added to the left-hand side of the backward rule.
– Finally, a *correspondence* (*analysis*) *rule* is used when both graphs have been modified in parallel. In our running example, this means that connections have been inserted into both the flow sheet and the simulation model and a link is created a posteriori. Thus, the correspondence rule additionally includes nodes 7 and 9 on the left-hand side.

Our approach goes beyond the derivation of asynchronous rules as proposed in the original TGG

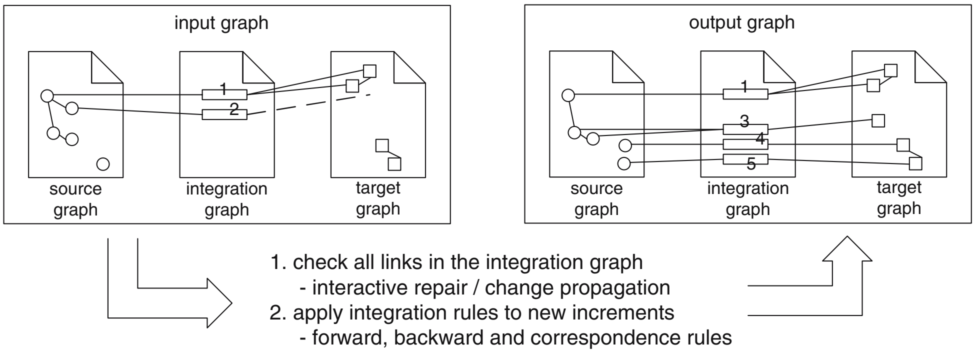

input graph
1
2
source graph
integration graph
target graph
output graph
1
3
4
5
source graph
integration graph
target graph
1. check all links in the integration graph
- interactive repair / change propagation
2. apply integration rules to new increments
- forward, backward and correspondence rules

Fig. 4 Overall integration algorithm

approach. An integration tool driven by asynchronous rules as described above would execute these rules in an atomic way. However, the user would not be aware of conflicts between these rules. These conflicts occur when transformation rules overlap with respect to their non-context increments. Therefore, the requirement for conflict detection (see end of Sect. 2) cannot be met in the case of atomic execution of asynchronous triple rules. In order to inform the user about conflicting rules, all applicable rules and their mutual conflicts have to be considered before a rule is selected for execution. To achieve this, we have to give up *atomic rule execution*, i.e., we have to decouple pattern matching from graph transformation. Breaking up rule execution constitutes the core contribution of this paper and will be discussed in the following Sects. 4–8.

4 Overview

4.1 Overall integration algorithm

In our interactive, incremental, and bidirectional integration tools, the execution of forward, backward, and correspondence analysis rules, as introduced in Sect. 3, is embedded in an overall integration algorithm. This algorithm is sketched in Fig. 4.

The input and output of the algorithm is an overall graph that contains graph representations of source, target, and integration subgraphs. Thus, graph edges between nodes belonging to different subgraphs can be handled as normal graph edges. When the algorithm is invoked, source and/or target graph contain some edges and nodes, with some of the nodes already being connected to links in the integration graph, if the algorithm is not invoked for the first time. For instance, in our scenario in Sect. 2 at the first invocation of the integration tool the target graph contains the simulation model for the core part of the plant, whereas source and integration graph are still empty. At the second invocation, a part of the simulation model is connected to a part of the flow sheet by links in the integration graph. Additional components have been added to the flow sheet that are not referenced by links yet.

In the abstract example in Fig. 4, two links (1, 2) are contained in the integration graph and reference some of the nodes in source and target graph. Other nodes in source and target graph have been added since the last run of the integration tool and are not referenced by links, yet. Furthermore, some nodes in the target graph have been deleted, resulting in a dangling reference from link 2.

Another implicit input to the algorithm is a set of forward, backward, and correspondence analysis rules. This input is implicit, as for the specification approach followed here (c.f. next Sect. 4.2), the rules are automatically hard-coded into the integration algorithm by a code generator.

The overall algorithm consists of two main phases: The first one deals with existing links, while the second one aims at handling nodes that are not referenced by links, yet, and at creating new links.

Each link in the integration graph has an associated state. When a link has been newly created by executing a rule or manually by the user, its state is initially set to consistent. In the first phase, for each link it is checked whether source and target patterns originally referenced by the link are still present in source and target graphs. If some parts of the patterns are missing due to modifications of source and/or target graphs, the state of the link is changed to damaged. In the example in Fig. 4, link 2 has a dangling reference to the target graph and thus is damaged.[7]

In the next step, it is attempted to repair damaged links. There are different possible repair strategies, most of which require user interaction. Some of these possibilities are explained in Sect. 8. In the example in Fig. 4,

[7] The underlying graph model does not support dangling edges, so additional information has to be stored in the integration graph to detect missing nodes, see Sect. 8.

all nodes of the target graph being referenced by the damaged link 2 have been deleted. Thus, the deletion is propagated to the source graph by deleting the link in the integration graph and the associated pattern in the source graph.

In the second main phase of the overall algorithm, forward, backward, and correspondence analysis rules as introduced in Sect. 3 are applied to nodes that are still available, i.e. are not referenced by a link, yet. In the example, a forward rule has been applied to create link 3, a backward rule for link 4, and a correspondence rule for link 5. This may have involved the manual resolution of conflicts between competing rules. After all rules have been applied, there may still be some nodes that have to be dealt with manually, due to the lack of appropriate rules. Additionally, links created by executing rules may be modified by the user later on. As already mentioned in Sect. 2, we do not intend to provide a fully automated transformation. Instead, we explicitly support the combination of manual and automatic steps to perform the transformation.

In this paper, the first phase of the algorithm is discussed only briefly in Sect. 8. We focus on the second part of the algorithm performing the execution of forward, backward, and correspondence analysis rules. In Sect. 5, the main idea of rule execution in our tools is informally explained for forward rules using an abstract example. The corresponding graph transformation rule definitions are described in Sect. 6 including some optimizations made to the algorithm originally introduced in [6]. Backward rules are handled analogously to forward rules. Correspondence analysis rules are addressed in Sect. 7. In this article, we concentrate on structural aspects of the integration. Besides handling of structure, our integration tools support handling of attributes and their values as well, which is not addressed here.

The integration algorithm is defined as graph transformation system using PROGRES. The following Sect. 4.2 illustrates how invariant parts of the algorithm together with triple rule-specific graph transformation rules and schema as well as graph transformation rule definitions for source and target graph are combined to the final graph transformation system and how it can be executed.

4.2 From triple rules to executable specifications

Our main goal is to formally define the algorithm for the execution of integration rules by defining a translation of integration rules into the specification of a graph transformation system. The approach is to combine some invariant specification parts with some specification parts that are derived from the set of synchronous triple graph grammar rules. The resulting overall specification is executable, which allows the evaluation of the integration algorithm and the integration rule set.

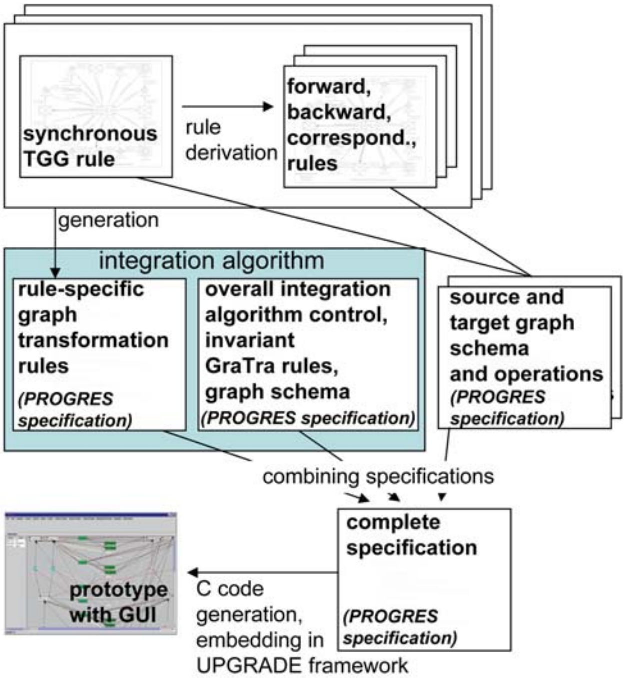

synchronous TGG rule
rule derivation
forward, backward, correspond., rules
generation
integration algorithm
rule-specific graph transformation rules
(PROGRES specification)
overall integration algorithm control, invariant GraTra rules, graph schema
(PROGRES specification)
source and target graph schema and operations
(PROGRES specification)
combining specifications
complete specification
(PROGRES specification)
C code generation, embedding in UPGRADE framework
prototype with GUI

Fig. 5 Overview of prototype generation

Figure 5 gives an overview of how the overall specification is derived following this approach. First, from each synchronous triple graph grammar rule contained in the rule set a forward, a backward, and a correspondence analysis rule are derived as explained in Sect. 3. In the sequel, the term *integration rule* (or just rule) will be used for such rules if its direction is not to be explicitly addressed.[8]

The integration algorithm for the execution of integration rules consists of rule-specific and generic graph transformation rules. The rule-specific graph transformation rules are automatically derived from the forward, backward, and correspondence analysis rules using a code generator. The generator output is an incomplete specification of a graph transformation system for the PROGRES system [35] containing the specific graph transformation rules for each integration rule. The translation of integration rules into the rule-specific graph transformation rules is not formally specified in this paper. Instead, in Sects. 6 and 7 some examples of the resulting graph transformation rules are described.

To obtain a complete and executable specification, the generated specification has to be combined with two

[8] To avoid confusing integration rules with graph transformation rules, the latter term is always written completely.

other specification parts: One specification contains the invariant integration-specific parts, which are the integration graph schema, the overall integration algorithm control, and some generic graph transformation rules for the algorithm. Additionally, for both source and target graph there are specifications each containing the graph schema and some graph transformation rules to realize operations allowing the user to modify the graph. They are needed because the executable specification has to be self-contained, i.e. it is not possible to access data of external applications. Thus, applications have to be simulated using graph transformation rules.

The complete specification is compiled by the PROGRES system [35] resulting in C code which is then embedded into the UPGRADE framework which provides a graphical user interface and persistent graph storage for the generated code [8]. This leads to a prototype which allows construction and modification of source and target graphs as well as simulating runs of the integration tool. Inspection of the current state of the graph and user interaction during the integration can be performed using the graphical user interface. Some additional coding is required for application-specific layout algorithms and user interfaces. However, these efforts can be kept small because the resulting prototypes are not targeted at the end user. Instead, they are intended only to serve as proof of concept and for the evaluation of integration rules as well as the integration algorithm itself. This prototyping methodology is called IREEN (*I*ntegration *R*ule *E*valuation *EN*vironment). Our approach for the integration of real-world applications and its relation to IREEN is briefly addressed in Sect. 9.

5 Incremental transformation (informal)

The main idea of our rule execution approach is to decouple pattern matching and graph transformation to allow conflict detection and user interaction as proposed in Sect. 3. Thus, the integration algorithm first finds possible applications of integration rules and explicitly stores the matches in the integration graph. Then, conflicts between these possible rule applications are determined. Conflict free rules are applied immediately, whereas the user is asked for conflict resolution for the remaining rule applications. Only then, the user-selected rules are executed.

The integration rule execution algorithm is defined by its overall control structure, depicted as UML activity diagram (with additional annotations) in Fig. 6, together with graph transformation rules realizing the single activities. The UML notation is only used for illustration, the overall control structure is implemented in PROGRES as well.

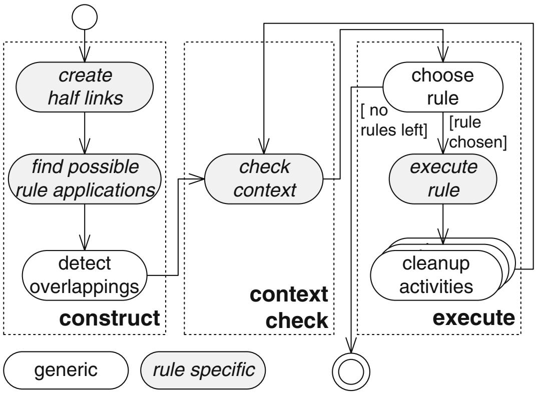


Fig. 6 Simplified integration algorithm (rule execution part)

The set of integration rules used in an integration tool is directly incorporated into the algorithm by realizing some of the activities with graph transformation rules specific for each integration rule contained in the set (labeled in italics in the figure). The remaining graph transformation rules are independent of specific rules.

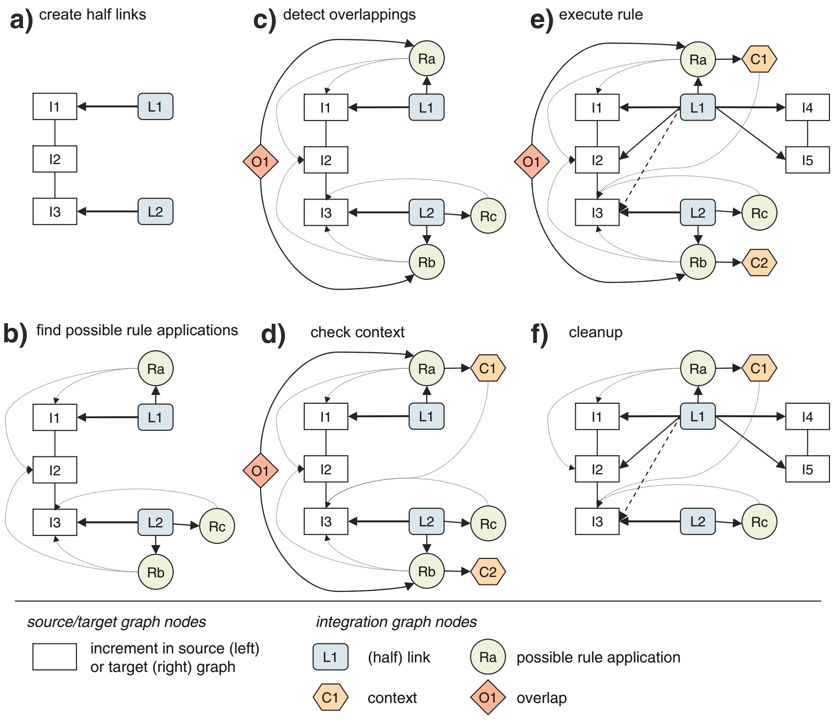
The algorithm is used to apply forward, backward, and correspondence analysis rules. In this and the following section, we present the algorithm focusing on forward rules only. The execution of backward rules works accordingly. The differences that have to be considered if correspondence rules are applied are addressed in Sect. 7.

The algorithm for the execution of integration rules consists of three phases. In the following, these phases are informally described with the help of the abstract example in Fig. 7. To keep it small, the example does not relate to specific types of source and target graphs as well as to specific rules.

During the first phase (construct), all possible rule applications and conflicts between them are determined and stored in the integration graph. First, for each increment in the source graph that has a type compatible with the dominant increment's type of any rule, a *half link* is created that references this increment. A half link is a link in the integration graph that references only one dominant increment in either the source or the target graph and no other increments. In the example, half links are created for the increments I1 and I3, and named L1 and L2, respectively (c.f. Fig. 7a).

Then, for each half link the possible rule applications are determined. This is done by trying to match the left-hand side of forward rules, starting at the dominant increments to avoid global pattern matching. In the example (Fig. 7b), three possible rule applications were

Fig. 7 Simplified example integration



found: Ra at the link L1 would transform the increments I1 and I2; Rb would transform the increments I2 and I3; and Rc would transform increment I3.

Each increment can be referenced by one link only as non-context increment (c.f. Sect. 3), as these increments (normal or dominant) are "consumed" by an integration rule. Thus, a *conflict* occurs if multiple possible rule applications reference the same non-context increment. After applying one of the conflicting rules, they are no longer available for the competing rules. Therefore, in the case of a conflict, the user has to choose one of the conflicting rules in the execute phase.

There are two types of conflicts: First, there can be multiple rule nodes at one half link. These share at least the dominant increment, so only one of the corresponding rules can be executed. This is the case for link L2 in the example in Fig. 7c: Rb and Rc are conflicting. Second, an increment can be referenced by possible rule applications of different links. In the example, the increment I2 is referenced by Ra and Rb. To prepare conflict-resolving user interaction, conflicts of the second type are explicitly marked in the graph by adding an edge-node-edge construct (e.g., O1 in Fig. 7b).

In the next phase (context check), the context is checked for all possible rule applications and all matches are stored in the graph. Only rules whose contexts have been found are ready to be applied. In the example in Fig. 7d, the context for Ra consisting of increment I3 in the source graph was found (C1). The context for Rb is empty (C2), the context for Rc is still missing.

In the last phase (execute), a rule is selected for execution and after its execution the data collected in the construct phase is updated. If any rule application whose context is present is unambiguous, i.e., it is not involved in any conflicts, it is automatically selected for execution. Otherwise, the user is asked to select one rule among the rules with existing context. If there are no executable rules left, the algorithm ends. In the example in Fig. 7d, no rule can be automatically selected for execution. The context of Rc is not yet available and Ra and Rb as well as Rb and Rc are conflicting. Here, it is assumed that the user selects Ra for execution.

Then, the selected rule is executed. In the example (Fig. 7e), this is the rule corresponding to the rule node Ra. As a result, increments I4 and I5 are created in the target graph, and references to all increments are added to the half link L1. Now, the half link has become a *consistent link*, also called full link. The result in source, target, and integration graph—concerning the link in question and its increments—is the same as if the corresponding forward triple graph grammar rule had been applied in an atomic way.

Afterwards, rules that cannot be applied and links that cannot be made consistent anymore are deleted. In Fig. 7f, Rb is deleted because it depends on the availability of I2 which is now referenced by L1 as a non-context increment. If there were alternative rule applications belonging to L1, they would be removed as well. Last, obsolete overlappings have to be deleted. In the example, O1 is removed because Rb was deleted. Please note that the cleanup procedure may change depending on how detailed the integration process has to be documented. For instance, if the user decisions are to be traced completely, all possible rule applications have to be preserved, even those not executed.

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

6 Specification of incremental transformation

In this section, the algorithm for the execution of integration rules is presented in detail. Graph transformation rules in PROGRES notation are shown for some of its activities. The algorithm described here is an optimized version of the one presented in [6] and used in the previous section to informally introduce the approach in a simplified way. The optimization provides two benefits: First, it is avoided to repeatedly check the context for each possible rule application. For each possible rule application, the algorithm determines a specific point in time when its context check is performed only once. This improves the performance of the algorithm a lot. Second, after the context check has been performed for a possible rule application, the rule application is deleted if the context has not been found. Only conflicts between already context checked rule applications are shown to the user. This prevents unnecessary user interaction. The optimization is based on the idea of keeping track of the dependencies between different potential rule applications as originally introduced in [25].

The optimized algorithm is depicted in Fig. 8. In the following subsections, its phases and the related activities are explained in detail. The most important graph transformation rules used to realize the activities are shown. To present the integration rule-specific graph transformation rules, the forward rule transforming a connection (c.f. Sect. 3) is used as running example.

Before explaining the phases of the algorithm in detail, we briefly introduce some additional language constructs of PROGRES that we make use of in the sequel. For a more detailed description, the reader is referred to [35].

PROGRES graph transformation rules can have a condition section. It consists of a list of expressions which refer to values of attributes of nodes contained in the left-hand side pattern (syntax to read attribute name of node `i: `i.name). If not all expressions evaluate to true for a match which has been found based on the left-hand side pattern, the match is ignored. For instance, a condition is used in the PROGRES graph transformation rule in Fig. 9 to ensure that the link's status is unchecked.[9]

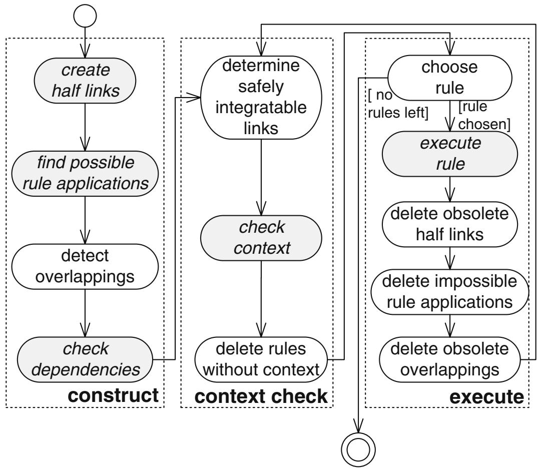

create half links
find possible rule applications
detect overlappings
check dependencies
construct
determine safely integratable links
check context
delete rules without context
context check
choose rule
[no rules left]
[rule chosen]
execute rule
delete obsolete half links
delete impossible rule applications
delete obsolete overlappings
execute

Fig. 8 Integration algorithm (rule execution part)

Another possibility to further restrict the possible matches of the left-hand side is to make use of *restrictions*. Restrictions can be applied to single nodes on the left-hand sides of graph transformation rules. The application of a restriction is visualized as a double arrow pointing at the respective node being labeled with the name of the restriction to apply. A restriction can be compared to a function that has to evaluate to true for the restricted node to be matched by the left-hand side. When applying a restriction, it can receive further parameters, e.g. constants, nodes, attributes of nodes. A restriction is applied, for instance, in the graph transformation rule in Fig. 10 at node '2. A restriction can be defined visually as graph test (comparable to the left-hand side of a graph transformation rule) or textually using specific expressions.

While attributes are read in conditions, they can be set in the transfer section of a PROGRES graph transformation rule (syntax to set attribute name of node i' to value val: i'.name:=val). The transfer section is used e.g. in the PROGRES graph transformation rule in Fig. 9.

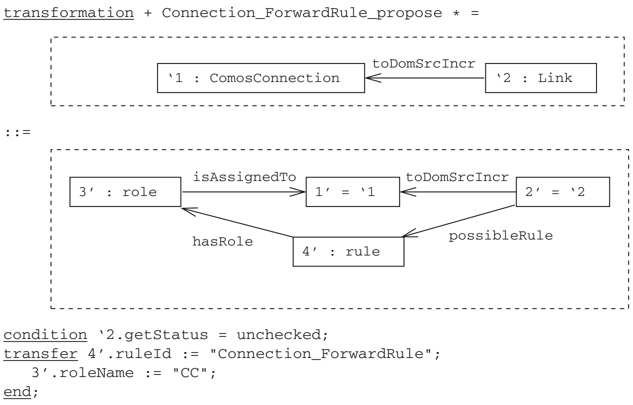
An asterisk (*) following the name of a PROGRES graph transformation rule implies that the graph transformation is performed for all possible matches when the graph transformation rule is executed. Without an

[9] The status of links (accessed by getStatus and setStatus) is used for the PROGRES implementation of the overall algorithm control which is not discussed in detail here.

asterisk, only one match is selected nondeterministically for execution.

PROGRES graph transformation rules can contain negative application conditions (NAC). Unlike in other graph transformation languages, they are not specified as additional patterns. Instead, they are contained in the left-hand side pattern in the form of *negative nodes* and *negative edges* which are depicted as crossed-out edges or nodes, respectively. For instance, node `‘3` in Fig. 12 is a negative node. The semantics are not explained in general here. Instead, for each graph transformation rule making use of NACs, the intended behavior will be presented.

Comparable to edges, *paths* can be used to express that nodes are connected by a sequence of edges and nodes. Paths are used in left-hand side patterns comparable to edges but are depicted by a double arrow. For instance, the graph transformation rule in Fig. 10 contains two applications of the `referencesNonContextIncrement` path. Paths can be defined textually and graphically, but their definitions are not shown here, due to the lack of space.



```
transformation + Connection_ForwardRule_propose * =
::=
condition ‘2.getStatus = unchecked;
transfer 4’.ruleId := "Connection_ForwardRule";
   3’.roleName := "CC";
end;
```

Fig. 9 Find possible rule applications

6.1 Construction phase

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

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

To create half links, for each rule a PROGRES graph transformation rule is derived that matches an increment with the same type as the rule's dominant increment in its left-hand side, with the negative application condition that there is no half link attached to the increment yet. Then, on its right-hand side the half link node is created and connected to the increment with an edge. All of these graph transformation rules are executed repeatedly, until no more left-hand sides are matched, i.e., half links have been created for all possibly dominant increments.

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

Figure 9 shows the PROGRES graph transformation rule for the example integration rule. The left-hand side consists of the half link and the respective dominant increment only because all other increments of this rule are context increments. In general, all non-context increments and their connecting edges are part of the left-hand side.

On the right-hand side, a rule node is created to identify the possible rule application (`4’`). This node carries the id of the rule and is connected to the half link. A role node is inserted to explicitly store the result of the pattern matching (`3’`).

If there are more increments matched, role nodes can be distinguished by the `roleName` attribute.

The asterisk (⋆) following the graph transformation rule's name tells PROGRES to perform the graph transformation for each possible match of the graph transformation rule's left-hand side. When executed together with the corresponding graph transformation rules for the other integration rules, as a result all possibly applicable rules are stored at each half link. Please note that if an integration rule is applicable for a half link with different matches of its source increments, multiple rule nodes with the corresponding role nodes are added to the half link.

During user interaction in later steps of the algorithm, for each link that is involved in a conflict all possible rule applications are presented to the user who has to resolve the conflict by selecting one of the rules. While conflicts between possible rule applications sharing the same dominant increment and thus belonging to the same link are immediately visible, others have to be marked with cross references (hyperlinks) between the conflicting rule applications. This is the case for all rules that belong to different links but share at least one non-context increment. Such rules are connected by an edge-node-edge construct with the node's type being `overlapping`. The corresponding graph transformation rule is not shown.

In general, not only conflicts between forward rules, but also conflicts between forward, backward, and correspondence analysis rules generated from the same synchronous rule are detected. As a result, it is not necessary to check whether the non-context increments of the right-hand side of the synchronous rule are already present in the target graph when determining possible rule applications in the second step of this phase (cf. Sect. 7).

To determine for each possible rule application when its context is to be checked, *dependencies* between rule applications and other links are searched and stored in the integration graph (activity check dependencies). These dependencies are based on the fact that the context required for a given possible rule application is either there from the beginning, will never be there, or is created by the execution of other rules. To handle the latter case, it is defined that a possible rule application depends on a link if parts of its context may be created by making the link consistent. For instance, in our scenario the integration rule transforming a connection can be applied only after the connected ports have been handled by other rules.

To be able to detect dependencies, some assumptions about the existence of parts of the context depending on the rule type (forward, backward, or correspondence analysis) have to be made:

- For forward rules, the source graph part of the context is assumed to be already contained in the source graph before the execution of any rule. In the synchronous rule, there must be a path from each target context increment to a source context increment traversing only target context increments, context links and sublinks, and one source context increment.
- Backward rules are handled analogously.
- For correspondence analysis rules, both source and target graph parts of the context have to be contained in the respective graph.
- For all rule types, the integration graph part of the context may still be missing in the integration graph. Instead, there are (inconsistent) half links connected to some increments which already reference their possible rule applications.

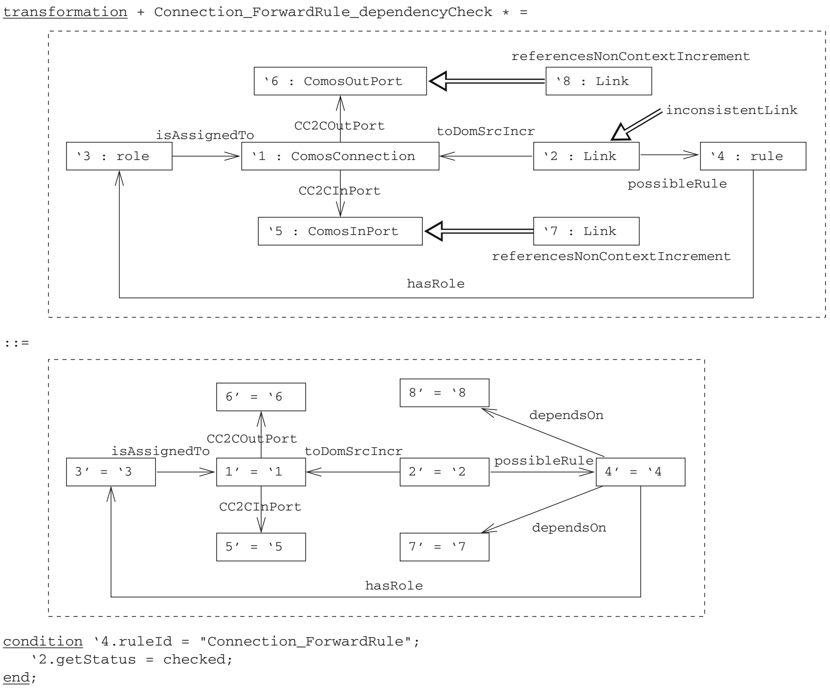
Figure 10 shows how these assumptions can be used to detect dependencies between possible rule applications and links using the forward rule of a connection as example: The source graph part of the left-hand side of the PROGRES graph transformation rule contains the pattern formed by all source increments—including source context—and their edges. The half link (`'2`) and the rule node (`'4`) together with all assigned roles as in the right-hand side of the `propose` graph transformation rule are contained in the integration graph part. For each source context increment that has an edge to any context element in the integration graph in the synchronous rule, a `Link` node connected to the increment by a `referencesNonContextIncrement` path is contained in the left-hand side. Here, the flow sheet ports (`'5`, `'6`) and the links `'7` and `'8` are part of the pattern because they are connected by an edge to a context sublink in the synchronous rule. The `referencesNonContextIncrement` path detects existing direct references to non-context increments by `toDomSrcInr` or `toNorSrcIncr` as well as indirect possible future references indicated by the existence of a rule node referencing the increment via a role. If a match for the left-hand side is found, `dependsOn` edges are created from the rule node (`'4`) to the links (`'7`, `'8`) on the right-hand side. The `dependsOn` edges explicitly model that finding the context for the current rule depends on the integration of the referenced links.

6.2 Context check

As the first activity of the context check phase, a set of links is determined which do not depend on other inconsistent links (definition not shown). A link is *ready for integration* (`safe IntLink`) if it neither has a rule that depends on a link which is still inconsistent, nor a rule that overlaps with a rule that belongs to a link that is not ready for integration. Only then can the context of all its possible rule applications be checked, and all information that is needed for user interaction later on is available.

The context check is performed for all links that are ready for integration in the next step of this phase (check context). The context is formed by all context increments from the synchronous rule. It may consist of increments of source and target graphs and of links contained in the integration graph.

Fig. 10 Dependency check for the connection rule



```
transformation + Connection_ForwardRule_dependencyCheck * =

::=

condition '4.ruleId = "Connection_ForwardRule";
   '2.getStatus = checked;
end;
```

Figure 11 shows the PROGRES graph transformation rule checking the context of the example integration rule. The left-hand side contains the half link (`2), the non-context increments (here, only `1), the rule node (`4), and the role nodes (`3). The non-context increments and their roles are needed to embed the context and to prevent unwanted folding between context and non-context increments. For the example rule, the context consists of the two ports connected in the source graph (`7, `8), the related ports in the Aspen graph (`5, `6), and the relating sublinks (`9, `10). The consistentLink restrictions make sure that the sublinks belong to a consistent link. The restriction safeIntLink applied on the link (`2) ensures that the context is checked only for links that can be safely integrated.

On the right-hand side, a new context node is created (`11). It is connected to all nodes belonging to the context by role nodes (12′, 13′, 14′, 15′, 16′, 17′) and appropriate edges. If multiple matches of the context are found, multiple context nodes with their roles are created as the graph transformation is performed for all matches.

To make sure that the context belonging to the right integration rule is checked, the rule id is constrained in the condition part of the graph transformation rules. After the context of a possible rule application has been found, the rule can be applied.

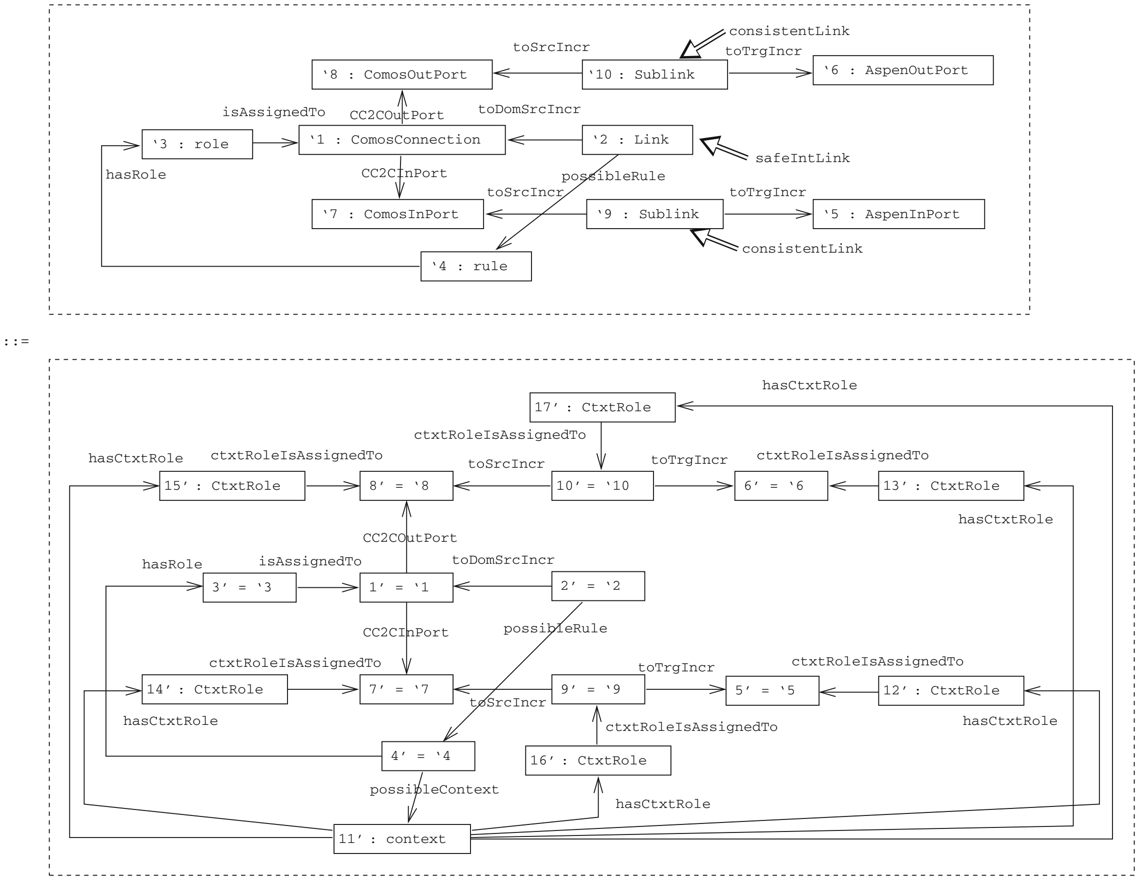
In the last step of this phase (delete rules without context), for all links that are ready for integration the possible rule applications whose contexts were not found are deleted. As they do not depend on inconsistent links, it is not possible that the missing contexts will be created later.

6.3 Execution phase

The execute phase starts with the selection of one rule for execution (choose rule). This activity comprises automatic as well as manual selection of rules. First, it is checked whether there is a rule that can be executed without user interaction. Therefore, the graph transformation rule in Fig. 12 is used to find a rule application that is not involved in any conflict. On the left-hand side of the graph transformation rule, a rule node is searched (`1) that has only one context node and is not related to any overlap node. If there are multiple context nodes, the rule cannot be executed automatically, instead, the user has to choose a context node. The rule node has to be related to exactly one half link (`2) that does not have another rule node. For forward rules, a rule node always belongs to one link only, while nodes of correspondence analysis rules are referenced by two half links (see Sect. 7).

If a match is found in the host graph, the rule node and the context node are selected for execution by substituting their referencing edges by selectedRule and selectedContext edges, respectively. The rule node is returned in the output parameter selRule.

```
transformation + Connection_ForwardRule_contextCheck * =
```



```
condition `4.ruleId = "Connection_ForwardRule";
   `2.getStatus = checked;
   `3.roleName = "CC";
transfer 12'.roleName := "AC.In";
   13'.roleName := "AC.Out";
   14'.roleName := "CC.In";
   15'.roleName := "CC.Out";
   16'.roleName := "CLPM.In";
   17'.roleName := "CLPM.Out";
end;
```

Fig. 11 Check context

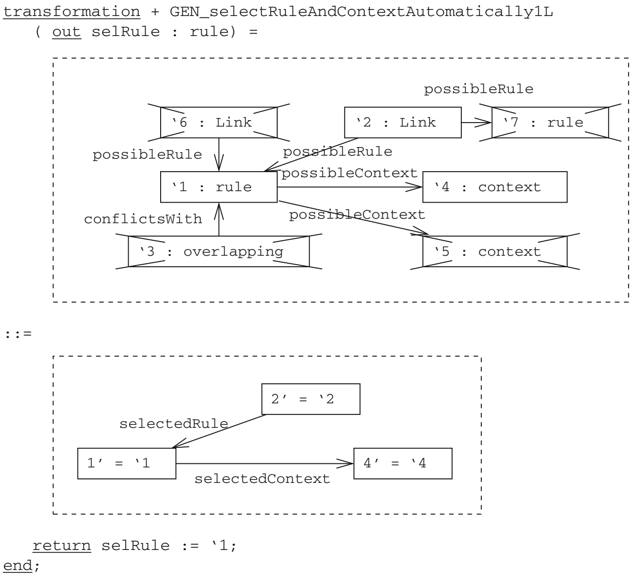
The corresponding rule can be applied in the next step.

If no rule could be selected automatically, i.e. all rules are involved in conflicts, the user has to resolve one of these conflicts by selecting a rule for execution. Therefore, all conflicts are collected and presented to the user. For each half link, all possible rule applications are shown. If a rule application conflicts with another rule of a different half link, this is marked as annotation (hyperlink) at both half links. The result of the user interaction is stored in the graph with the help of `selectedRule` and `selectedContext` edges, as with the automatic rule selection.

If there is neither a rule that can be automatically selected nor a conflict left, the algorithm terminates. If there are still inconsistent links left at the end of the algorithm, the user has to perform the rest of the integration manually.

The selected rule is now executed by a rule-specific PROGRES graph transformation rule, for our example rule see Fig. 13. The left-hand side pattern is nearly identical to the right-hand side of the context check graph transformation rule in Fig. 11. The main difference is that the edge from the link (`` `2 ``) to the rule node (`` `4 ``) is now a `selectedRule` edge and the edge from the rule node to the context node (`` `11 ``) is a `selectedContext` edge.

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



```
transformation + GEN_selectRuleAndContextAutomatically1L
   ( out selRule : rule) =
```

```
   return selRule := `1;
end;
```

Fig. 12 Select unambiguous rule

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

The following steps of the algorithm are performed by generic graph transformation rules that update the information about possible rule applications and conflicts. First, obsolete half links are deleted (delete obsolete half links). A half link is obsolete if its dominant increment is referenced by another consistent link as non-context increment. Then, possible rule applications are removed which are conflicting with the rule just executed (delete impossible rule applications). Finally, overlap nodes referencing rules that have been deleted and are thus dangling are removed (delete obsolete overlappings).

7 Correspondence analysis

Correspondence analysis rules are used to detect correspondences between existing increments in source and target graph. Therefore, they search for existing patterns in source and target graphs and create links in the integration graph. One application would be to use a rule set consisting only of correspondence analysis rules and to apply it to two complete graphs. As a result, all corresponding parts were connected by consistent links and all inconsistencies were marked by inconsistent half links. Although supported by our approach as well, this scenario is quite unrealistic.

Instead, during the creation and modification of source and target graphs an integration tool is used from time to time to propagate changes between the graphs. For propagation, all kinds of rules, i.e. forward, backward, and correspondence rules are applied together. They detect already corresponding increments and propagate increments between source and target graphs that do not have a corresponding counterpart in the other graph, yet.

During the execution of correspondence analysis rules, the same requirements regarding conflict detection and user interaction have to be fulfilled as with forward and backward rules. Especially, conflicts between both types of rules have to be detected. Therefore, correspondence analysis rules are executed together with forward and backward rules using the same integration algorithm.

In the following subsection, we will describe only the differences between the execution of forward and correspondence rules (c.f. Fig. 8). As in the previous section, the rule for a connection is used as running example.

7.1 Applying correspondence analysis rules

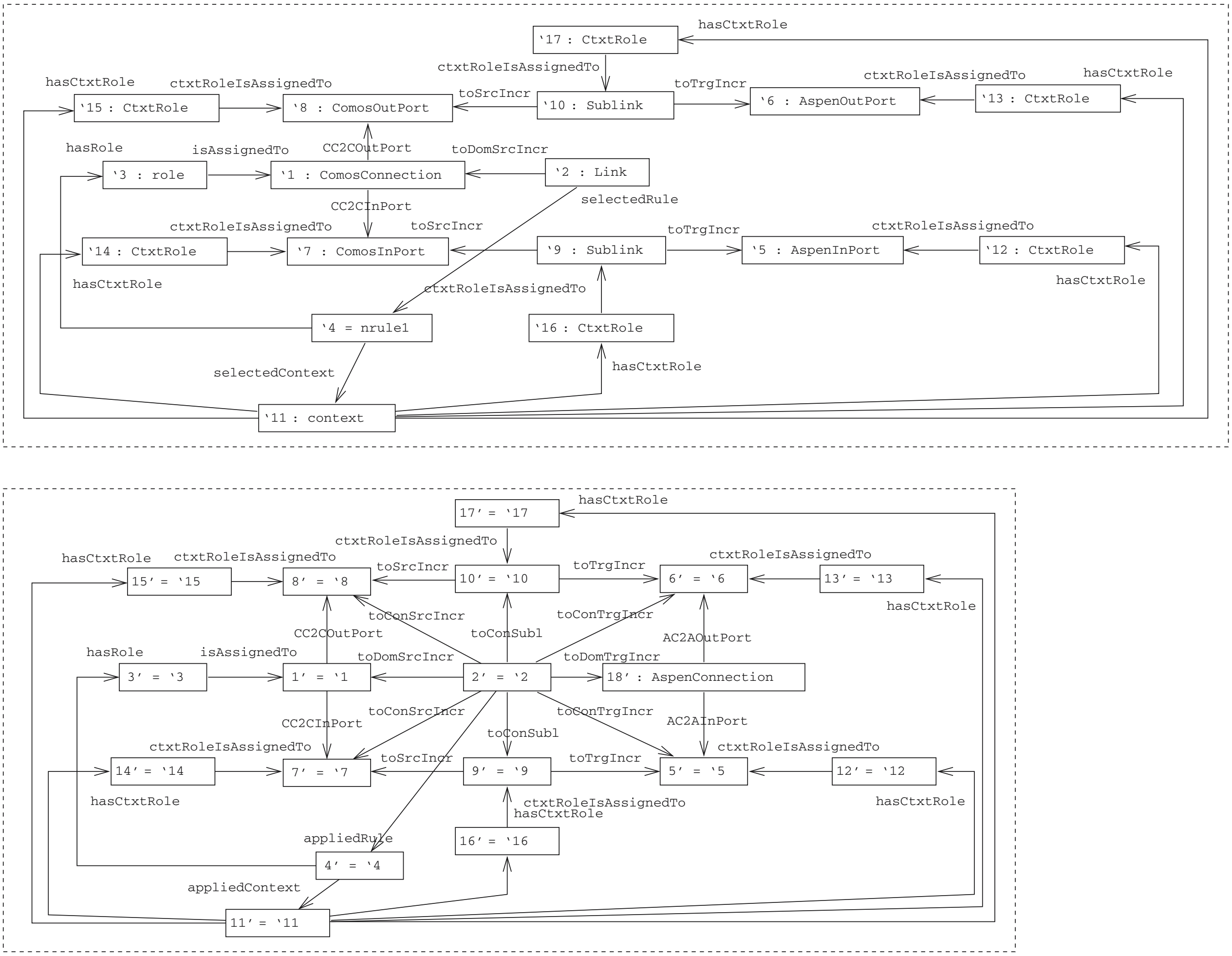
The intention of correspondence analysis rules is to detect already existing patterns of increments in source and target graph and to relate them by a new link in the integration graph. Thus, during the find possible rule applications activity of the integration algorithm, the patterns are searched in source *and* target graph. The corresponding rule-specific graph transformation rule for our running example is depicted in Fig. 14.

Its left-hand side looks for all dominant and normal increments in *both* source and target graph (the connections '1 and '2). Additionally, *two* nodes of type Link are searched ('3, '4). These links have been created during the create half links activity by the according graph transformation rules for forward and backward rules, if connections exist in source and target graph.

On the right-hand side, as in Fig. 9, a role node is created for each increment found (5', 6') and a rule node referencing all role nodes is created (7'). In contrast to forward and backward rules, the rule node is referenced by *two* links. Thus, the potential correspondence between the two connection increments is established.

All following activities of the algorithm preserve both links, until finally, when the rule is executed, the two links are merged into a single link. This is done by deleting one of them and redirecting its edges to the other

```
transformation + Connection_ForwardRule_apply
   ( nrule1 : rule) =
```



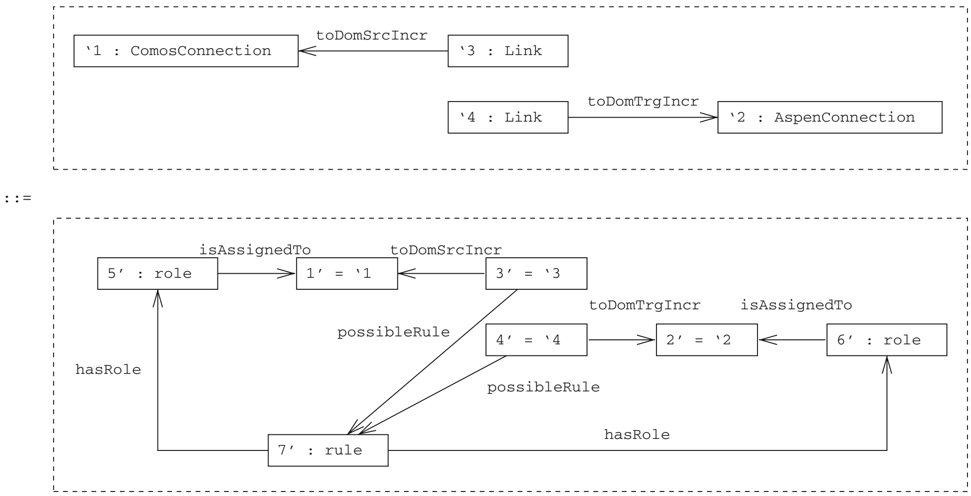
```
::=

condition `4.ruleId = "Connection_ForwardRule";
   `2.getStatus = checked;
   `3.roleName = "CC";
   `12.roleName = "AC.In";
   `13.roleName = "AC.Out";
   `14.roleName = "CC.In";
   `15.roleName = "CC.Out";
   `16.roleName = "CLPM.In";
   `17.roleName = "CLPM.Out";
transfer 2'.setStatus := ruleBased;
end;
```

Fig. 13 Execute rule

one. As both increment patterns are already contained in source and target graph, when executing the rule just the integration graph is modified.

In general, finding all potential correspondences leads to a combinatorial explosion of possible rule applications. This is addressed by different techniques, depending on the integration rule in question. First, conditions based on attribute values of increments can be added. For instance, simply the user-given names of target and source increments can be compared. More complex conditions can be defined, too, e.g. equations on attributes containing chemical data like the substances flowing in streams to find matching streams in our scenario. Second, defining rules with complex patterns on both source and target graph sides makes finding too many matches more unlikely. Third, as with the rule for connections, there are no attributes and the pattern cannot be extended but there is context comprising source, target, and integration graph parts contained in the rule. In this case, all matches are found but the context check in the next phase eliminates "wrong" possible rule applications. Current work aims at providing a fourth possibility using artificial contexts for rules that neither have attributes nor context.

```
transformation + Connection_CorrespondenceRule_propose * =
```



```
condition '3.getStatus = unchecked;
   '4.getStatus = unchecked;
transfer 7'.ruleId := "Connection_CorrespondenceRule";
   5'.roleName := "CC";
   6'.roleName := "AC";
end;
```

Fig. 14 Find possible rule applications for correspondence analysis

7.2 Executing correspondence analysis rules together with forward and backward rules

Besides executing a set of integration rules containing only rules all being of the same type (i.e., either forward, backward or correspondence analysis rules), the mixed execution of all types of rules is supported.

If at least one forward or backward variant as well as the correspondence analysis variant of a rule can be applied to the same set of increments, this leads to a conflict. This conflict can be used to ensure that corresponding increments are not duplicated by applying a forward and a backward rule as proposed in the beginning of this section. Figure 15 demonstrates this using an abstract example: Increment I1 in the source graph is the dominant increment for link L1 and increment I4 in the target graph is the dominant increment for link L2. In this situation, a rule R is applicable in its forward variant (→) at L1, in its backward variant (←) at L2 and as correspondence analysis rule (↔) at both links. Obviously, R → and R ↔ are conflicting because they are both referenced by the same link, as well as R ← and R ↔. These two conflicts prevent the automatic execution of any of the three rules.

In general, user interaction is used to resolve conflicts. Concerning the connection rule, the conflict could be automatically eliminated by assigning the correspondence analysis rule a higher priority than the forward and backward rules and adapting the generic find unambiguous rule step of the algorithm to react according to priorities. This is possible because only one connection may exist between two ports, thus the duplication of connections by applying forward and backward rules is not allowed. In other cases, only the user can decide which rule should be executed. As a heuristic, correspondence rules can be given a higher priority anyway, forcing manual post-processing if this caused a wrong result. This heuristic implies that as many matches as possible are found automatically, assuming that it does not make sense to transform an increment when a corresponding increment is already available.

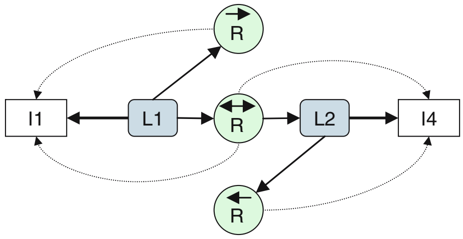


Fig. 15 Conflict between forward, backward, and correspondence analysis rule

8 Consistency check and repair

In the first main phase of the overall integration algorithm (c.f. Sect. 4.1), before other rules are executed it is checked whether already existing links in the integration graph have been damaged by modifications of source and target graphs. If a link is damaged, this has to be propagated: All other links referencing it as context have to be set to damaged as well.

There are different causes for a link to become damaged:

- Increments in source or target graph have been deleted, resulting in dangling references from the link.
- Attribute values of source or target increments have been changed in a way that the attribute conditions tested by the propose and the context check graph transformation rule no longer hold.
- Edges in source or target graph have been deleted, resulting in the patterns searched by the propose and the context check graph transformation rule being no longer present.
- A link contained in the integration context of the current link has changed its state to damaged.
- A link contained in the integration context of the current link has been changed manually (references to increments were removed), resulting in the pattern that was searched during the context check being no longer present.

To be able to detect changes induced by the causes listed above, additional information has to be kept in the integration graph. For links that have been established by the execution of integration rules, at least a reference to the applied rule has to be stored to be able to check whether the originally referenced pattern is still there. For links that have been created manually, a description of the whole patterns that are referenced by the link has to be copied to the integration graph. Only with this information is it possible to detect changes that have been made to the graphs without tracing them. Tracing changes would be possible as well for the PROGRES prototypes but with respect to our a-posteriori integration tool implementations, where tracing depends on the possibilities of the integrated tools, it is not supported. Just checking for dangling edges is not possible for both types of links because dangling edges are not supported by the graph model underlying PROGRES and thus are immediately deleted.

The simplest way to deal with a damaged link is to delete it and thereby make the remaining increments available again for the execution of other rules. Though possible in general, deleting the link is not a good option. The modification resulting in the damage of the link was most probably done on purpose. The link—even if it is damaged—contains valuable information on which parts of the other graph may be affected by the modifications. So in most cases just asking the user to resolve the inconsistency manually is a better option than deleting the link.

If the inconsistency has been caused by the deletion of increments, it is possible to propagate it by first deleting all remaining increments in source and target graphs and then deleting the link. This behavior could be restricted to situations where all increments of a link have been deleted in one of the graphs or where one of the dominant increments has been deleted.

Another option is to restore consistency by removing the cause for the inconsistency. For instance, missing increments or edges may be created. This option is desirable only in those cases where the operation causing the damage was carried out accidentally, because it would be undone. For attribute values, the attribute conditions of the synchronous rule can be used to propagate the change.

If only some parts of the patterns in source and target graphs are missing, it is possible to perform a pattern matching for the whole pattern of the corresponding rule using still existing nodes to initialize some of the patterns' nodes. This can be helpful e.g. if the user first deletes an increment and then recreates it.

In general, it cannot be determined automatically which alternative for repairing damaged links is appropriate. Because of that, user interaction is necessary here as well. The integration tool can determine all the possibilities for dealing with the inconsistency and let the user decide. It is even possible to temporarily delete the link, collect all the alternatives of how normal integration rules can be applied to the freed increments and then present them together with other repair actions to the user.

In our current implementation, damaged links are just detected and the user is notified to repair them manually. Using the damaged link, his attention is directed towards the part of the document that needs to be adapted. Therefore, an integration rule-specific PROGRES restriction is generated. It searches the whole right-hand side of the synchronous rule including the links and edges between links and increments in source and target graphs. A corresponding PROGRES graph transformation rule searches for links that were created by the rule in question but do not fulfill the restriction. On its right-hand side, it changes the state of the link to `damaged`. Current work aims at evaluating other alternatives for detection and repair of inconsistencies and how they can be implemented with graph transformation rules.

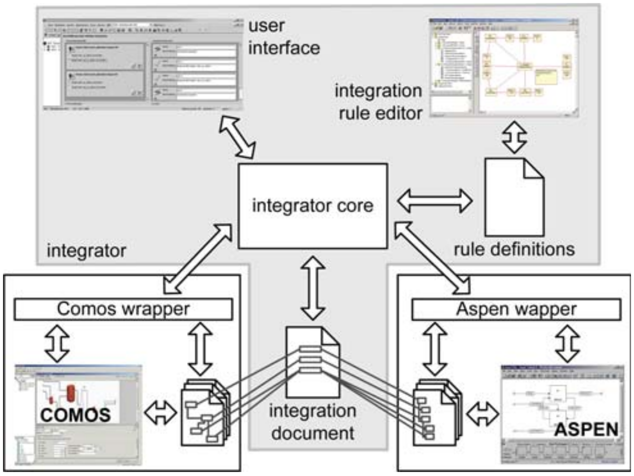


Fig. 16 System architecture of framework-based integrator

9 Integrator framework and prototype

Using executable IREEN specifications to evaluate the integration algorithm and test possible extensions is very convenient, as very low implementation effort is required to gain experimental prototypes. Nevertheless, for several reasons this approach is not suitable for the realization of industrial prototypes: First, the execution of graph transformation specifications requires a heavy-weight, grown infrastructure (PROGRES, UPGRADE, and the underlying graph database GRAS [22]) which is not well suited for an implementation in an industrial context. Second, IREEN prototypes cannot be connected to external tools, prohibiting a-posteriori integration, which is urgently needed. Third, the IREEN user interface shows the internal structure of the overall graph consisting of source, target, and integration graph. While this view is very helpful when debugging the integration algorithm, it is nearly incomprehensible from the end user's perspective.

Thus, we are continuously developing a C++-based framework [5,19] that reflects all results gained by applying the IREEN methodology. The coarse-grained system architecture of the integrator for our evaluation scenario (cf. Sect. 2) that was realized based on the framework is depicted in Fig. 16. Framework-based integrators are controlled by integration rules as well. Therefore, the framework also includes tool support (upper right corner) for *rule modeling* which is based on the UML [4,7] (i.e., the graph grammar formalism is not exposed to domain experts).

Existing applications and their documents are connected to the framework using *tool wrappers* that provide a graph interface on their data. In our example, these tools are *Comos PT* and *Aspen Plus*. The corresponding wrappers both use the applications' COM interfaces to access their API.

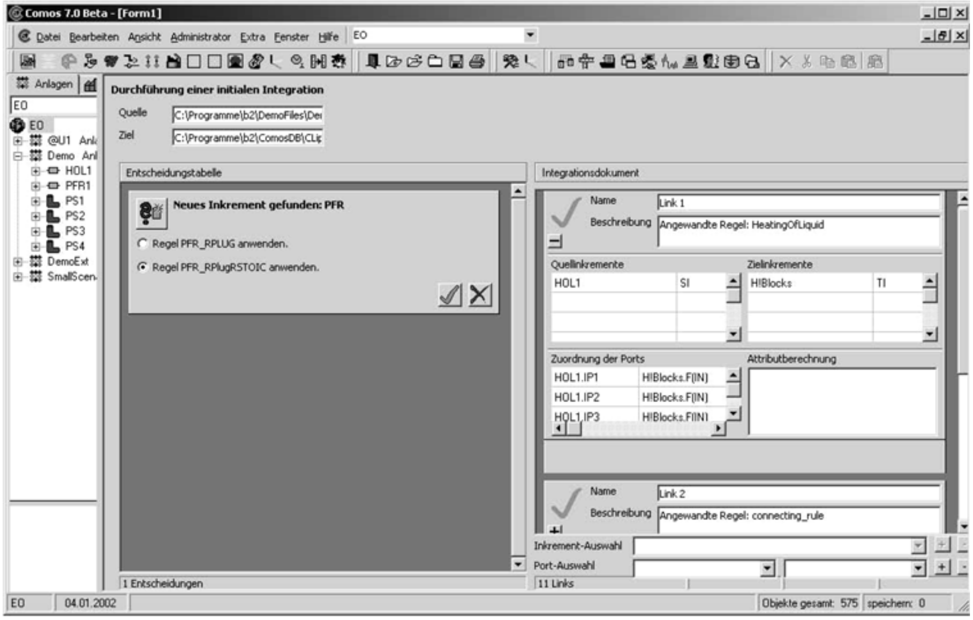
As there is no overall graph as in IREEN, and source and target documents' internal data structures cannot be modified, the relationships between the documents' data are stored in an additional *integration document*. It is serialized as XML file, but kept in memory during runtime of the integrator providing optimized access to its content, e.g. via indexes.

Unlike with the IREEN approach, integration rules are interpreted at runtime by the *integrator core*, which is the main component of the framework. This allows for extending the integration rule set at runtime without having to recompile the integrator. Thus, the integrator can 'learn' new rules from manual interaction when a complete rule set is not available in advance [7].

The integrator core is reused for all framework-based integrators. Its implementation is a one-to-one realization of the integration algorithm underlying IREEN, with the rule-specific graph transformations being executed by a specific *integration rule interpreter*. This rule interpreter is based on a graph transformation engine that is part of the integrator core, too. This engine can be kept much more light-weight than, e.g., PROGRES, as only very simple graph transformations have to be executed. All pattern matching is done starting from dominant increments, in most cases only locally traversing the graph avoiding global pattern matching. Thus, the—in theory—high complexity of pattern matching does not affect the integrators' performance. The rule-independent graph transformation rules of the integration algorithm are manually hard-coded into the integrator core, making use of the optimized storage of links in the integration document.

User interaction is performed using a user interface that is tailored to the specific integrator application (cf. Fig. 17). For the integrator between *Aspen Plus* and *Comos PT*, the user interface is incorporated as plug-in into the *Comos PT*. The menu bar and the tree view shown in the screenshot are a part of *Comos PT*, while the main part of the window is currently occupied by the integrator. In its left part, all pending decisions are presented to the user. The screenshot shows the decision between two alternatives for processing the reactor of the process flow diagram (cf. Sect. 2.3). In its right part, the integration document can be inspected. It contains a list of all links, each of which can be maximized to show all its details, e.g. the list of all related increments.

Currently, in a follow-up project of IMPROVE (cf. Sect. 11) we are, among other aspects, investigating further possibilities for integrator user interfaces. The most promising idea is to hide as much of the integration document as possible from the user. The integrated



Durchführung einer initialen Integration

Quelle: C:\Programme\b2\DemoFiles\De

Ziel: C:\Programme\b2\ComosDB\CLip

Neues Inkrement gefunden: PFR

○ Regel PFR_RPLUG anwenden.

● Regel PFR_RPlugRSTOIC anwenden.

Name: Link 1

Beschreibung: Angewandte Regel: HeatingOfLiquid

Quellinkremente		Zielinkremente	
HOL1	SI	H!Blocks	TI

Zuordnung der Ports	
HOL1.IP1	H!Blocks.F(IN)
HOL1.IP2	H!Blocks.F(IN)
HOL1.IP3	H!Blocks.F(IN)

Attributberechnung

Name: Link 2

Beschreibung: Angewandte Regel: connecting_rule

1 Entscheidungen

11 Links

EO | 04.01.2002 | Objekte gesamt: 575 | speichern: 0

Fig. 17 Screenshot of the integrator user interface

documents could be presented directly side by side using the original views of the integrated applications. Links could be only implicitly represented by enhancing the original views by additional functionality as highlighting corresponding increments in both documents upon selection or browsing between related increments. Unrelated increments and inconsistent links could be marked using a certain color, and pending decisions between conflicting rules could be directly annotated at the dominant increments. For a user interface like this it is necessary to be able to extend the user interface of at least one of the integrated applications. In our follow-up project, this is possible for *Comos PT*, as it is carried out in tight cooperation with its developer *innotec*.

10 Related work

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

10.1 Related work on triple graph grammars

The triple graph grammar approach was introduced by Schürr in [34], where the theoretical foundations for building TGG-based integration tools were laid. The work was motivated by integration problems in software engineering. For example, [26] describes how triple graph grammars were applied in the IPSEN project [29], which dealt with integrated structure-oriented software development environments.

The Ph.D. thesis of Lefering [25] built upon the theoretical foundations of Schürr. In this thesis, Lefering developed an early framework for building integration which was based on triple graph grammars. The framework was implemented in C++, rules had to be transformed manually into C++ code to make them operational. The framework was applied to the integration of requirements engineering and software architecture documents.

Unfortunately, the work of Lefering was not continued after the termination of the IPSEN project. Rather, applications of triple graph grammars were built using the PROGRES environment. In the reengineering projects VARLET [20] and REFORDI [11], synchronous triple rules were transformed manually into forward rules (for transforming the old system into a renovated one being based on object-oriented concepts). The PROGRES system was used to execute forward rules in an atomic way.

Our work on rule execution differs from systems such as REFORDI and VARLET inasmuch as a single triple rule is executed in multiple steps. Decomposition of rule execution was motivated in particular by the need for detecting conflicts and resolving them interactively. Rather, we draw upon early work performed by Lefering. In particular, this applies to the design of the basic data structures (such as distinction between dominant, normal, and context increments), the idea to split up rule execution, and the introduction of dependencies between links for the topological sorting of rule applications and the elimination of pseudo conflicts. On the other hand, our work contributes the following improvements:

- We added detection, persistent storage, and resolution of conflicts between integration rules.
- We provide a precise formal specification of the integration algorithm. In [25], the algorithm was described informally and implemented in a conventional programming language.
- Likewise, rules had to be hand-coded in Lefering's framework. In contrast, synchronous triple rules are converted automatically into specific rules for execution in our approach.
- We used the specification in two ways: First, IREEN was constructed by rapid prototyping by generating code from the formal specification (Fig. 5). Second, an implementation designed for industrial use was derived from the formal specification [4].

To conclude this subsection, let us briefly discuss other current work on triple graph grammars:

The PLCTools prototype [3] allows the translation between different specification formalisms for programmable controllers. The translation is inspired by the triple graph grammar approach [34] but is restricted to 1:n mappings. The rule base is conflict-free, so there is no need for conflict detection and user interaction. It can be extended by user-defined rules which are restricted to be unambiguous 1:n mappings. Incremental transformations are not supported.

In [24], triple graph grammars are generalized to handle integration of multiple documents rather than pairs of documents. From a single synchronous rule, multiple rules are derived in an analogous way as in the original TGG approach as presented in [34]. The decomposition into multiple steps such as link creation, context check, and rule application is not considered.

In [39,10], a plug-in for flexible and incremental consistency management in Fujaba is presented. The plug-in is specified using story diagrams [17], which may be seen as a UML-inspired notation for graph rewrite rules. From a single triple rule, six rules for directed transformations and correspondence analysis are generated in a first step. In a second step, each rule is decomposed into three operations (responsibility check, inconsistency detection, and inconsistency repair). The underlying ideas are similar to our approach, but they are tailored towards a different kind of application. In particular, consistency management is performed in a reactive way after each user command. Thus, there is no global search for possible rule applications. Rather, modifications to the object structure raise events which immediately trigger consistency management actions.

10.2 Other approaches

Related areas of interest in computer science are (*in-*) *consistency checking* [36] and *model transformation*. Consistency checkers apply rules to detect inconsistencies between models which then can be resolved manually or by inconsistency repair rules. Model transformation deals with consistent translations between heterogeneous models. Our approach contains aspects of both areas but is more closely related to model transformation.

In [15], a consistency management approach for different view points [16] of development processes is presented. The formalism of distributed graph transformations [38] is used to model view points and their interrelations, especially consistency checks and repair actions. To the best of our knowledge, this approach works incrementally but does not support detection of conflicting rules and user interaction.

Model transformation recently gained increasing importance because of the model-driven approaches for software development like the model-driven architecture (MDA) [32]. In [18] and [21] some approaches are compared and requirements are proposed. While the requirements on transformations for the MDA can be compared to the ones on our approach sketched in this paper, there is still a gap to transformations available in current practice. Still, most tools work batch-wise and do not support user interaction to influence the transformations.

In [37], an approach for non-incremental and non-interactive transformation between domain models based on graph transformations is described. The main idea is to define multiple transformation steps using a specific meta model. Execution is controlled with the help of a visual language for specifying control and parameter flow between these steps.

In the AToM project [12], modeling tools are generated from descriptions of their meta models. Transformations between different formalisms can be defined using graph grammars. The transformations do not work incrementally but support user interaction. Unlike our approach, control of the transformation is contained in the user-defined graph grammars.

The QVT Partner's proposal [2] to the QVT RFP of the OMG [31] is a relational approach based on the UML and very similar to the work of Kent [1]. While Kent is using OCL constraints to define detailed rules, the QVT Partners propose a graphical definition of patterns and operational transformation rules. These rules operate in one direction only. Furthermore, incremental transformations and user interaction are not supported.

BOTL [9] is a transformation language based on UML object diagrams. Comparable to graph transformations, BOTL rules consist of an object diagram on the left-hand side and another one on the right-hand side, both describing patterns. Unlike graph transformations, the former one is matched in the source document and the latter one is created in the target document. The transformation process is neither incremental nor interactive. There are no conflicts because of very restrictive constraints on the rules.

In [14], some of the approaches sketched here and others are compared. They are based on graph transformations but not all of them use triple graph grammars. In [28], it is discussed why graph transformations in general are a useful device for the implementation of model transformations.

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

11 Conclusion

We have presented a novel approach to the execution of integration rules in incremental and interactive integration tools using graph transformations. The approach is based on triple graph grammars. Rule execution is broken up into multiple phases to take care of conflict detection and user interaction. For a set of triple rules, each single triple rule is translated into a set of graph transformation rules which are plugged into a generic algorithm for rule execution.

For the *usability* of incremental and interactive integration tools, it is crucial that user interaction is minimized. User interaction cannot be avoided in the case of conflicting rules, but the integration tool must not request further interactions because it is "not smart enough". Here, the optimization which we added to the integration algorithm plays an important role. The concept of safe integration ensures that rules are not processed too early. Without safe integration, it might happen that pseudo conflicts are reported. A *pseudo conflict* occurs between two rules if these rules overlap with respect to non-context increments, but only one of the rules will finally be able to fire (please recall that conflicts are detected before the context check is performed). The optimization we added ensures that links are processed in topological order and rules which will never be applicable are eliminated as soon as possible.

The *performance* of the integration algorithm depends on how efficiently graph transformation rules are executed. In the case of the IREEN prototype presented in Sect. 4.2, the PROGRES environment generates fast code from graph transformation rules by employing sophisticated heuristics for graph pattern matching [35]. In the C++ framework presented in Section 9, the integration algorithm was implemented even more efficiently by exploiting specific properties of the integrator rule set and by hard-coding the domain-independent core part of the integration algorithm.

The research presented in this paper was carried out under the umbrella of IMPROVE, a long-term research project which is concerned with models and tools for engineering design processes [30]. The integrator framework described in Section 9 was developed in close cooperation with an industrial partner, namely the German software company *innotec*, which develops tools for chemical engineering. Recently, a follow-up project to IMPROVE has been launched which addresses technology transition into industrial practice. The project has a duration of three years and builds upon the framework and prototypes developed so far.

The work we have performed so far was driven by requirements defined by our industrial partners and chemical engineers involved in the IMPROVE project (see Sect. 2). Within the technology transfer project, we are going to perform a comprehensive *case study* to evaluate our approach. In particular, this case study will further investigate usability issues (see also Sect. 9) and open problems concerning e.g. correspondence analysis (Sect. 7) and repair actions (Sect. 8).

Acknowledgments 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* and the constructive comments provided by the unknown reviewers.

References

1. Akehurst, D., Kent, S., Patrascoiu, O.: A relational approach to defining and implementing transformations between metamodels. J. Softw. Systems Modeling **2**(4), 215–239 (2003)
2. Appukuttan, B.K., Clark, T., Reddy, S., Tratt, L., Venkatesh, R.: A model driven approach to model transformations. In: Proceedings of the 2003 Model Driven Architecture:

Foundations and Applications (MDAFA2003), CTIT Technical Report TR-CTIT-03-27. University of Twente, The Netherlands (2003)

3. Baresi, L., Mauri, M., Pezzè, M.: PLCTools: Graph transformation meets PLC design. Electron. Notes Theor. Comput. Sci. **72**(2), (2002)
4. Becker, S.M., Haase, T., Westfechtel, B.: Model-based a-posteriori integration of engineering tools for incremental development processes. J. Softw. Systems Modeling **4**(2), 123–140 (2005)
5. Becker, S.M., Haase, T., Westfechtel, B., Wilhelms, J: Integration tools supporting cooperative development processes in chemical engineering. In: Proceedings of the 6th Biennial World Conference on Integrated Design and Process Technology (IDPT-2002), Pasadena. Society for Design and Process Science (2002)
6. Becker, S.M., Lohmann, S., Westfechtel, B.: Rule execution in graph-based incremental interactive integration tools. In: Proceedings of the 2nd International Conference on Graph Transformations (ICGT 2004), LNCS, vol. 3256, pp. 22–38. Springer, Heidelberg (2004)
7. Becker, S.M., Westfechtel, B.: UML-based definition of integration models for incremental development processes in chemical engineering. J. Integr. Des. Process Sci. Trans. SDPS **8**(1), 49–63 (2004)
8. Böhlen, B., Jäger, D., Schleicher, A., Westfechtel, B.: UPGRADE: building interactive tools for visual languages. In: Proceedings of the 6th World Multiconference on Systemics, Cybernetics, and Informatics (SCI 2002), vol. I (Information Systems Development I), pp. 17–22, USA (2002)
9. Braun, P., Marschall, F.: Transforming object oriented models with BOTL. Electron. Notes Theor. Comput. Sci. **72**(3), (2003)
10. Burmester, S., Giese, H., Niere, J., Tichy, M., Wadsack, J.P., Wagner, R., Wendehals, L., Zündorf, A.: Tool integration at the meta-model level: the Fujaba approach. Int. J. Softw. Tools Technol. Transf. (STTT) **6**(3), 203–218 (2004)
11. Cremer, K., Marburger, A., Westfechtel, B.: Graph-based tools for re-engineering. J. Softw. Maintenance Evolut. Res. Practice **14**(4), 257–292 (2002)
12. de Lara, J., Vangheluwe, H: Computer aided multi-paradigm modelling to process petri-nets and statecharts. In: Proceedings of 1st International Conference on Graph Transformations (ICGT 2002), LNCS, vol. 2505, pp. 239–253. Springer, Heidelberg (2002)
13. Ehrig, H., Engels, G., Kreowski, H., Rozenberg, G. (eds.): Handbook on Graph Grammars and Computing by Graph Transformation: Application, Languages, and Tools, vol. 2. World Scientific, Singapore (1999)
14. Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovszky, T., Prange, U., Taentzer, G., Varró, D., Varró-Gyapay, S.: Model transformation by graph transformation: a comparative study. In: MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005) Available from: http://www.inf.mit.bme.hu/FTSRG/Publications/varro/2005/mtip05.pdf (2005)
15. Enders, B.E., Heverhagen, T., Goedicke, M., Tröpfner, P., Tracht, R.: Towards an integration of different specification methods by using the ViewPoint framework. Trans. SDPS **6**(2), 1–23 (2002)
16. Finkelstein, A., Kramer, J., Goedicke, M.: ViewPoint oriented software development. In: International Workshop on Software Engineering and its Applications, pp. 374–384 (1990)
17. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: A new graph rewrite language based on the unified modeling language. In: Proceedings of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT), LNCS vol. 1764, pp. 296–309. Springer, Heidelberg (1998)
18. Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: the missing link of MDA. In: Proceedings of 1st International Conference on Graph Transformations (ICGT 2002), LNCS vol. 2505, pp. 90–105, Barcelona. Springer, Heidelberg (2002)
19. Herold, S.: Ein Rahmenwerk für graphbasierte Integrationswerkzeuge. Master's thesis, RWTH Aachen University, Germany (2005)
20. Jahnke, J., Zündorf, A.: Applying graph transformations to database re-engineering. In Ehrig et al. [13], pp. 267–286
21. Kent, S., Smith, R.: The bidirectional mapping problem. Electron. Notes Theor. Comput. Sci. **82**(7) (2003)
22. Kiesel, N., Schürr, A., Westfechtel, B.: GRAS: a graph-oriented software engineering database system. Infor. Systems **20**(1), 21–51 (1995)
23. Kleppe, A.G., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Pearson Education, Boston (2003)
24. Königs, A., Schürr, A.: Multi-domain integration with MOF and extended triple graph grammars [online]. In: Bezivin, J., Heckel, R. (eds.) Language Engineering for Model-Driven Software Development, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005. <http://drops.dagstuhl.de/opus/volltexte/2005/22> [date of citation: 2005-02-01]
25. Lefering, M.: Integrationswerkzeuge in einer Softwareentwicklungsumgebung. Berichte aus der Informatik. Shaker Verlag, Aachen (1995)
26. Lefering, M., Schürr, A.: Specification of integration tools. In Nagl [29], pp. 324–334
27. Lohmann, S.: Ausführung von Integrationsregeln mit einem Graphersetzungssystem. Master's thesis, RWTH Aachen University, Germany (2004)
28. Mens, T., van Gorp, P., Karsai, G., Varró, D.: Applying a model transformation taxonomy to graph transformation technology. In: Karsai, G., Täntzer, G. (eds.) GraMot 2005, International Workshop on Graph and Model Transformations, vol. 152 of Electron. Notes Theor. Comput. Sci. pp. 143–159 (2006)
29. Nagl, M. (ed.): Building Tightly-Integrated Software Development Environments: The IPSEN Approach. LNCS, vol. 1170. Springer, Berlin (1996)
30. Nagl, M., Marquardt, W.: SFB-476 IMPROVE: Informatische Unterstützung übergreifender Entwicklungsprozesse in der Verfahrenstechnik. In: Informatik '97: Informatik als Innovationsmotor, Informatik aktuell, pp. 143–154, Aachen Springer, Heidelberg (1997)
31. OMG. MOF 2.0 query / view / transformations, request for proposal (2002)
32. OMG Architecture Board ORMSC. Model driven architecture (MDA) (2001)
33. Rozenberg, G. (ed.): Handbook on Graph Grammars and Computing by Graph Transformation 1 (Foundations). World Scientific, Singapore (1997)
34. Schürr, A.: Specification of graph translators with triple graph grammars. In: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 1994), LNCS, vol. 903, pp. 151–163, Herrsching. Springer, Heidelberg (1995)

35. Schürr, A., Winter, A., Zündorf, A.: The PROGRES approach: Language and environment. In Ehrig et al. [13], pp. 487–550
36. Spanoudakis, G., Zisman, A.: Inconsistency management in software engineering: Survey and open research issues. In: Handbook of Software Engineering and Knowledge Engineering, vol. 1, pp. 329–380. World Scientific, Singapore (2001)
37. Sprinkle, J., Agrawal, A., Levendovszky, T., Shi, F., Karsai, G.: Domain model translation using graph transformations. In: Proceedings of the 10th International Conference on Engineering of Computer-Based Systems (ECBS 2003), pp. 159–167. IEEE Computer Society (2003)
38. Taentzer, G., Koch, M., Fischer, I., Volle, V.: 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, pp. 269–340. World Scientific, Singapore (1999)
39. Wagner, R., Giese, H., Nickel, U.A.: A plug-in for flexible and incremental consistency mangement. In: Proceedings of the International Conference on the Unified Modeling Language 2003 (Workshop 7: Consistency Problems in UML-Based Software Development), San Francisco. Blekinge Institute of Technology (2003)

Authors' Biographies

Bernhard Westfechtel received his diploma degree in 1983 from University of Erlangen-Nuremberg, his doctoral degree (Ph.D.) in 1991 and his habilitation degree in 1999 from RWTH Aachen, Germany. Since 2004, he has been a full professor of computer science at University of Bayreuth, Germany. He is interested in software engineering environments, software configuration management, process modeling, model-driven development, software architectures, tool integration, and graph technology.

Sebastian Herold received his degree in computer science from the RWTH Aachen University in 2005. This paper is partly based on his diploma thesis. Since then, he has been working at the Software Architecture Group of the University of Kaiserslautern as research assistant. His main research interest are software architectures and model-driven development.

Sebastian Lohmann received his degree in computer science from the RWTH Aachen University in 2004. This paper is partly based on his diploma thesis. Since then, he has been with sd&m AG in Ratingen.

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.