Incremental Integration Tools for Chemical Engineering: An Industrial Application of Triple Graph Grammars

Simon M. Becker and Bernhard Westfechtel

Lehrstuhl für Informatik III, RWTH Aachen
Ahornstraße 55, D-52074 Aachen
`(sbecker|bernhard)@i3.informatik.rwth-aachen.de`

Abstract. Triple graph grammars, an extension of pair graph grammars, were introduced for the specification of graph translaters. We developed a framework which constitutes an industrial application of triple graph grammars. It solves integration problems in a specific domain, namely design processes in chemical engineering. Here, different design representations of a chemical plant have to be kept consistent with each other. Incremental integration tools assist in propagating changes and performing consistency analysis. The integration tools are driven by triple rules which define relationships between design documents.

1 Introduction

Triple graph grammars, an extension of pair graph grammars [1], were introduced at the WG '94 workshop [2]. Originally, they were motivated by integration problems in software engineering; later, they were applied to other domains as well. In general, triple graph grammars may be used for the specification of graph translations, coupling of graph structures, and consistency maintenance.

This paper reports on an *industrial application* of triple graph grammars. The Collaborative Research Centre IMPROVE [3] is concerned with the development of models and tools for chemical engineering design. In IMPROVE, we realized a framework for building incremental and interactive integration tools [4, 5]. The framework was developed in close cooperation with an industrial partner (innotec, a Germany software company, which offers an engineering database system called COMOS PT).

In chemical engineering design, a chemical plant is described from different perspectives by a set of interrelated *design documents*, including various kinds of flow sheets for describing the chemical process and the components of the chemical plant, simulation models for steady-state and dynamic simulations, etc. Design proceeds incrementally, i.e., the design documents are gradually refined and improved. Throughout the whole design process, interrelated design documents have to be kept consistent with each other. Design documents may be represented as graphs in a natural way. Triple graph grammars are used to define correspondences between graph structures. They serve as specifications for rule-based integration tools.

Section 2 briefly recalls triple graph grammars. Section 3 introduces a motivating example from the chemical engineering domain. Section 4 derives general requirements from the motivating example. Section 5 presents our framework for building incremental and interactive integration tools. Section 6 explains how triple rules are defined in this framework. Section 7 is devoted to implementation issues. Section 8 discusses the way we applied triple graph grammars and the experiences we made. Section 9 compares related work. Section 10 presents a short conclusion.

2 Triple Graph Grammars

Pair graph grammars were introduced as early as 1971 by Pratt to specify graph-to-graph translations [1]. A pair grammar defines a set of pair productions which modify the participating graphs and update correspondences between nodes. *Triple graph grammars* [2] are an extension of pair graph grammars. They were motivated by the study of integration problems in software engineering environments [6]. These studies showed the need for a separate *correspondence graph* to be placed in between *source* and *target graph*. The terms “source” and “target” denote distinct ends, but do not imply a direction. A *triple production* consists of productions operating on source, correspondence, and target graph, respectively, as well inter-graph mappings which are used to relate elements of the correspondence graph to elements of the source and the target graph, respectively.

Let us briefly recall some definitions from [2]:

- A *graph* is a quadruple $G = (V, E, s, t)$, where V and E are finite sets of vertices (nodes) and edges, and $s, t : E \to V$ assign source and target nodes to edges.
- A *graph morphism* from G to G' is a pair $h = (h_V, h_E)$, where $h_V : V \to V'$, $h_E : E \to E'$ are defined such that they “preserve” source and target nodes.
- A (monotonic) *graph production* is a pair of graphs $p = (L, R)$, where $L \subset R$.
- A graph production p is *applicable* to a graph G if there is a morphism $h : L \to G$. *Application* of p results in a graph G' which is extended with (copies of) nodes and edges in $R \setminus L$.
- A *triple graph* is a structure $G = (SG \leftarrow h_{SG} - CG - h_{TG} \to TG)$, where SG, CG, and TG denote source, correspondence, and target graph, respectively, and h_{SG} and h_{TG} are graph morphisms.
- A *triple production* is a structure $p = (sp \leftarrow h_{sp} - cp - h_{tp} \to tp)$, where sp, cp, and tp denote source, correspondence and target productions, respectively. h_{sp} and h_{tp} are pairs of graph morphisms which map the left-hand and right-hand sides, respectively.
- A triple production p is *applicable* to a triple graph G if its component productions are applicable to the component graphs and the production mappings may be mapped onto the graph mappings. *Application* of p results in a triple graph G' such that the component productions are applied to the component graphs and the graph mappings are updated according to the production mappings.

Based on these definitions, the following propositions were proved in [2]:

- A given triple production
  $p = ((SL, SR) \leftarrow h_{sp} - (CL, CR) - h_{tp} \to (TL, TR))$

may be split into *source-local production*
$p_{sl} = ((SL, SR) \leftarrow \epsilon - (\emptyset, \emptyset) - \epsilon \rightarrow (\emptyset, \emptyset))$
and a *source-to-target production*
$p_{st} = ((SR, SR) \leftarrow h_{sp} - (CL, CR) - h_{tp} \rightarrow (TL, TR))$
such that $p = p_{sl}\, p_{st}$.

– A sequence of applications of triple productions $p_1 \dots p_n$ is equivalent to the application of all source-local productions $p_{1_{sl}} \dots p_{n_{sl}}$, followed by the application of all source-to-target productions $p_{1_{st}} \dots p_{n_{st}}$.

Triple graph grammars are used for the specification of graph-based integration tools which may be classified as follows:

Synchronous coupling Source, correspondence, and target graph are modified synchronously by applying triple productions.
Source-to-target translation Given a source graph SG and a sequence of source-local productions, apply source-to-target productions, yielding a correspondence graph CG and a target graph TG.
Incremental change propagation Starting from a triple graph $G = (SG, CG, TG)$, first apply a sequence of source-local productions to SG and then propagate the changes to CG and TG by applying corresponding source-to-target productions.

In the context of this paper, we will focus on incremental change propagation, which in general may be performed bidirectionally.

3 Motivating Example

Incremental change propagation is essential in chemical engineering design, where chemical plants are described in design documents from different perspectives. Below, we focus on a problem which we have been studying in cooperation with an industrial partner. innotec, a Germany software company, offers an engineering database system called COMOS PT [7]. In particular, COMOS PT maintains *flow sheets* describing the chemical process and the composition of the chemical plant to be designed. The problem was to integrate COMOS PT with Aspen Plus [8], a simulation environment provided by another vendor. In Aspen Plus, *simulation models* are created (and executed) which have to be kept consistent with the corresponding flow sheets.

In chemical engineering, the flow sheet acts as a central document for describing the chemical process. The flow sheet is refined iteratively so that it eventually describes the chemical plant to be built. Simulations are performed in order to evaluate design alternatives. Simulation results are fed back to the flow sheet designer, who annotates the flow sheet with flow rates, temperatures, pressures, etc. Thus, information is propagated back and forth between flow sheets and simulation models. Unfortunately, the relationships between them are not always straightforward. To use a simulator such as Aspen Plus, the simulation model has to be composed from pre-defined blocks. Therefore, the composition of the simulation model is specific to the respective simulator and may deviate structurally from the flow sheet.

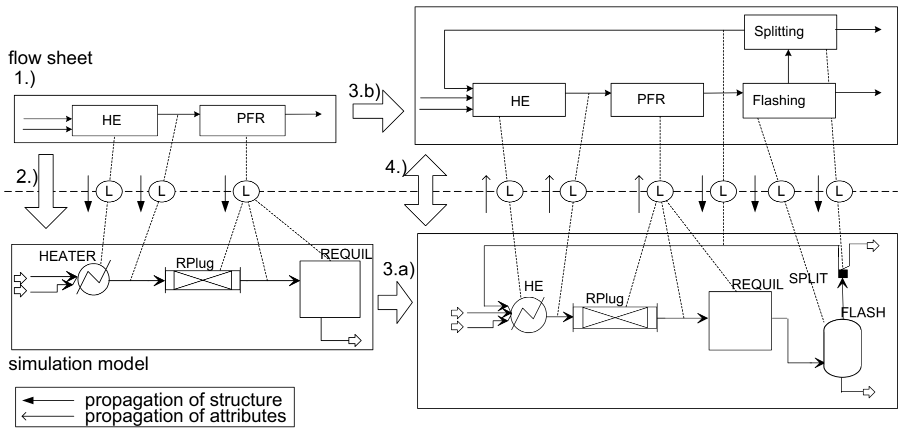

Fig. 1. Integration between flow sheet and simulation model

Figure 1 illustrates how an incremental integration tool assists in maintaining consistency between flow sheets and simulation models. The chemical process taken as example produces ethanol from ethen and water. Flow sheet and simulation model are shown above and below the dashed line, respectively. The integration document for connecting them contains links which are drawn on the dashed line. The figure illustrates a design process consisting of four steps:

1. An initial flow sheet is created in COMOS PT. This flow sheet is still incomplete, i.e., it describes only a part of the chemical process (heating of substances and reaction in a plug flow reactor, PFR).
2. The integration tool is used to transform the initial flow sheet into a simulation model for Aspen Plus. Here, the user has to perform two decisions. While the heating step can be mapped structurally 1:1 into the simulation model, the user has to select the most appropriate block for the simulation to be performed. Second, there are multiple alternatives to map the PFR. Since a straightforward 1:1 mapping is not sufficient, the user maps the PFR into a cascade of two blocks.
3. The simulation is performed in Aspen Plus, resulting in a simulation model which is augmented with simulation results. In parallel, the flow sheet is extended with the chemical process steps that have not been specified so far (flashing and splitting).
4. Finally, the integration tool is used to synchronize the parallel work performed in the previous step. This involves information flow in both directions. First, the simulation results are propagated from the simulation model back to the flow sheet. Second, the extensions are propagated from the flow sheet to the simulation model. After these propagations have been performed, mutual consistency is re-established.

4 Requirements

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

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

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

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

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

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

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

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

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

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

5 Framework for Building Integration Tools

Figure 2 provides an overview of the *framework* for *tool integration* which we have developed with our industrial partner. At the heart of this framework, the integrator core offers basic functionality. In particular, it includes the basic control logic, i.e., the algorithms for document integration. The integrator core accesses the integration document which stores fine-grained links and records the application of integration rules. Furthermore, it is connected to the tools and documents to be integrated via respective *wrappers*, which are used to abstract from tool-specific details (a posteriori integration). The integrator user interface is used to control the integrator interactively. The rules which drive the integrator are specified in a *rule definition tool*. Rules are interpreted by the integrator core; alternatively, they may be hard-coded and compiled for more efficient execution[1].

[1] Currently, the latter requires manual programming.

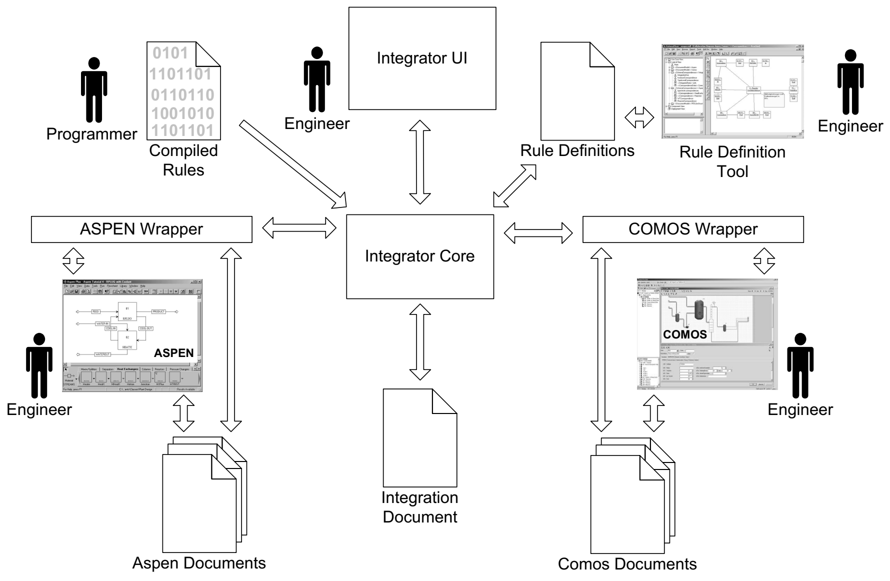

Fig. 2. Framework for tool integration

Let us illustrate the operation of this framework by the example of Section 3:

1. The flow sheet designer creates an initial flow sheet in COMOS PT. Here, COMOS PT is used as it stands.
2. The simulation expert uses the integrator to create a simulation model. The integrator accesses the flow sheet through the COMOS wrapper which provides a graph-based view on the source graph. Similarly, the ASPEN wrapper offers an updatable view on the target graph. The simulation expert activates source-to-target productions through the interactive interface of the integrator. Source-to-target relationships are stored in the integration document, which plays the role of the correspondence graph.
3. The flow sheet designer and the simulation expert operate in parallel locally on their respective documents.
4. The changes are synchronized with the help of the integrator. To synchronize the changes, both source-to-target and target-to-source productions are applied.

6 Definition of Rules

For the definition of rules, we decided to rely on the *Unified Modeling Language* [9] primarily for pragmatic reasons. The UML is a wide-spread modeling language which is supported by CASE tools such as Rational Rose, TogetherJ, etc. Although the UML is based on an object-oriented rather than on a graph-based data model, there are strong relationships to graphs and graph rewriting systems. For example, an *object diagram* showing a set of objects connected by links may be viewed as a graph. Likewise,

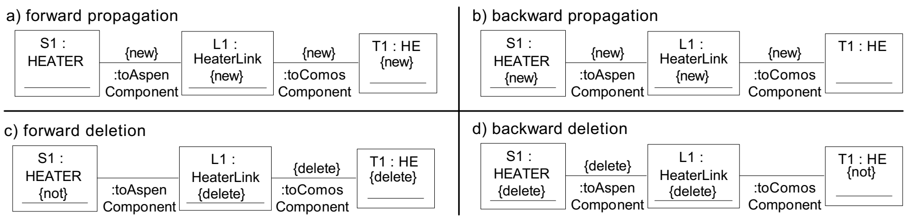

Fig. 3. Encoding graph rewrite rules with collaboration diagrams

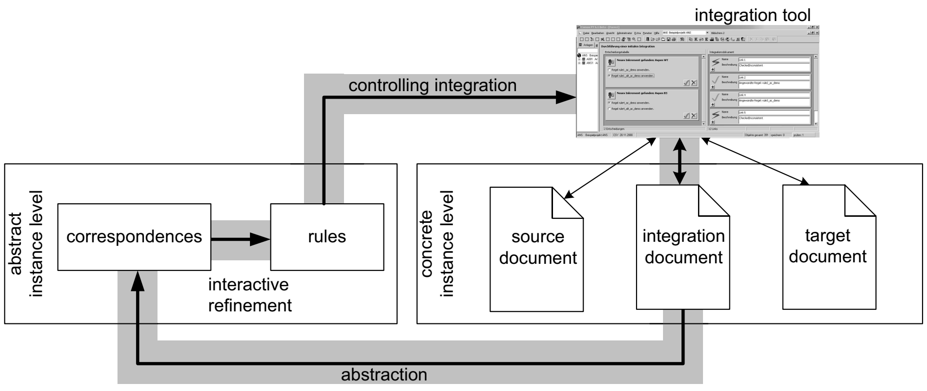

Fig. 4. Modeling process

a *collaboration diagram* extending a static object diagram with operations for creating/deleting objects or links corresponds to a graph rewrite rule.

Figure 3 illustrates how graph rewrite rules are expressed as collaboration diagrams. All of these rules deal with simple 1:1 correspondences between heater elements in flow sheets and heater blocks in simulation models. Left- and right-hand side of a graph rewrite rule are merged into a single diagram. Creation and deletion of objects and links are indicated by annotations new and delete, respectively. Rules a) and b) are *constructive* since they insert objects and links into the target (source) document after the source (target) document has been extended. In contrast, the *destructive* rules c) and d) are applied to propagate deletions: If the source (target) object is not present any more, the target (source) object as well as the link object have to be deleted. Please note that in general users may perform not only insertions, but also changes and deletions to source and target documents. Thus, we have to deal with general *graph rewrite rules* rather than only with generating productions.

So far, we have tacitly assumed that the rule base is given when the integrator is applied. In fact, it is fairly difficult to define an appropriate and comprehensive rule set beforehand. Rather, the rules have to be learned through experience. This is achieved through a *round-trip modeling process* which is illustrated in Figure 4. Let us assume an initial rule base to start with. The user may apply these rules to establish correspondences between source and target document. If the user wishes to establish a certain cor-

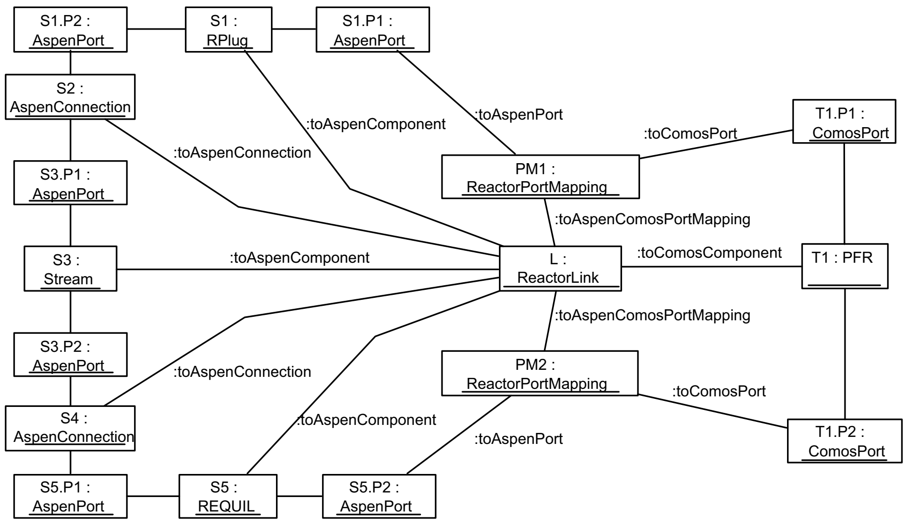

Fig. 5. Complex correspondence

respondence even though a specific rule is not available, he may resort to built-in ad-hoc rules by means of which "untyped" correspondences may be created. Here, the user has to specify correspondences manually. Concrete correspondences stored in actual integration documents may be transformed into abstract correspondences defining mutually related graph patterns. Subsequently, these static correspondences may be transformed into dynamic rules. After that, the integrator may be used with the improved rule set.

Figure 5 gives an example of a complex correspondence which is represented by an object diagram. This correspondence is abstracted from the link between the plug flow reactor in COMOS and the cascade of reactor blocks in ASPEN, as illustrated in Figure 1[2]. From an object diagram, we may derive collaboration diagrams by introducing new and delete annotations. This may be performed in two steps. In the first step, a synchronous rule is defined. In the second step, source-to-target and target-to-source rules may be derived from the synchronous rule.

So far, only the structural aspects of correspondences and rules have been addressed. In addition, *attributes* have to be considered. In practice, elements of flow sheets and simulation models may carry a large number of attributes which have to be kept consistent with each other. Therefore, rules for attribute assignments have to be provided as well. In the UML, the relationships between attribute values may be defined in the Object Constraint Language (OCL). For further details on attribute assignments, the reader is referred to [5].

[2] Please note that a simplified notation was employed in Figure 1, while Figure 5 shows the actual internal graph representation. In particular, connections are represented as objects, as well as the end points (ports) of both connections and devices.

7 Implementation

The framework introduced in Section 5 was implemented in cooperation with our industrial partner innotec, the provider of COMOS. The implementation was performed such that the integrator would interact smoothly with COMOS (and ASPEN). Furthermore, it was required to keep the implementation as slim as possible and to avoid rucksacks of infrastructure software. For these reasons, we did not use the *PROGRES* environment [10], which is still heavily used in other projects carried out in our group. Rather, a *light-weight implementation* was realized which is tailored towards the specific requirements of our application domain and does not provide a general and powerful graph rewriting machinery.

For the points to be made in the next section, it is not important to go into the details of the implementation. However, we do have to convey an overall understanding of how the integrator works. The integrator is provided with the source document, the target document, and the integration document. Both the source document and the target document may have been modified after the last run of the integrator (see e.g. the last step of our example in Section 3). To re-establish consistency, the integrator searches source and target documents for elements which do not participate in correspondences. These elements are scheduled for source-to-target or target-to-source transformations. In the next step, the set of candidate rules is identified for each scheduled element. If there is no such rule, the user may apply a built-in ad hoc rule. If there is more than one rule, the user has to select the appropriate one. If there is exactly one rule, the rule is applied automatically. In addition, the integrator performs a run through the integration document to check the consistency of the correspondences already stored in the integration document. Each correspondence stores a reference to the respective rule. All correspondences whose source or target patterns were modified as marked as inconsistent. If essential elements of those patterns were deleted, the correspondence is deleted as well. If possible, repair actions are initiated to re-establish consistency. In addition to structural rules, the integrator also handles attribute rules (through the execution of script code).

8 Discussion

After having recalled the theoretical foundations of triple graph grammars in Section 2 and having presented a practical application in the following sections, we now reflect on the experiences we have made in the described application.

By and large, triple graph grammars constitute a powerful conceptual framework for addressing integration problems for the following reasons:

- For complex m:n relationships, it pays off to introduce a correspondence graph in between the source and the target graph.
- Triple productions declaratively specify coupling of graph structures and abstract from the different possible modes of use: synchronous coupling, source-to-target (and target-to-source) translation, and incremental change propagation.

On the other hand, the actual definitions as given in [2] bear some restrictions which prevented their use in our context:

– Graphs are not typed, and nodes do not carry attributes. Both types and attributes are very important in our application domain.
– Inter-graph relationships are represented by graph morphisms. Usually, morphisms are defined between graphs of the same type. Furthermore, they have to preserve not only source and target nodes of edges, but also types. This is not the case for the relationships between the correspondence graph and source or target graph. In addition, a correspondence node may be related to only one source and target node, respectively. Thus, complex correspondences cannot be modeled in the way we have done it (see Figure 5); rather, they have to spread over multiple correspondence nodes which are grouped only implicitly[3]. Altogether, it seems more appropriate to represent inter-graph relationships by *inter-graph edges* instead.
– The definitions deal only with graph *grammars*. However, we are concerned with graph *rewriting systems*: The user may also apply deleting or modifying transformations. Grammars are adequate for batch translations: Given a source graph SG, construct a target graph TG. However, we have to deal with general editing rather than merely with constructing operations.

Finally, we faced some practical problems in our application domain:

– The original proposal tacitly assumes that we may start from given grammars for the source and target graphs. In practice, these grammars are not available. Usually, tools provide a procedural interface (e.g., OLE) for reading and writing the data stored in native data structures. There is no definition of the underlying graphical language. At best, the tool builder may provide a documented textual interchange format (typically XML).
– Likewise, it is by no means straightforward to define the triple rules. In fact, the correspondences between flow sheets and simulation models may be defined only through practical experience. Therefore, we introduced our round-trip modeling process illustrated in Figure 4.
– In our application domain, we only have a fairly weak notion of *consistency*. The rules describing correspondences between flow sheets and simulation models are of heuristic nature. Similar observations apply e.g. to the relationships between requirements definitions and software architectures in software engineering [6].
– In [2], it is assumed that the decoupling of transformations is achieved with the help of graph parsers: Only when we know the production sequence on the source graph can we apply the corresponding productions on the target graph. Building of graph parsers is complicated anyway, but it completely breaks down when the parsing problem is undecidable[4]. For these reasons, we have never considered building graph parsers. Rather, the changes performed on source and target graph are determined in a completely different way by traversing source, target, and correspondence graph, as described in Section 7.

[3] In the example given in [2], this grouping is introduced informally by composite node identifiers.
[4] The original proposal assumes monotonic productions to guarantee decidability.

9 Related Work

Triple graph grammars have their roots in the IPSEN project [11] which dealt with tightly integrated software development environments. Originally, only a priori integration was considered, i.e., tools were designed for integration from the very beginning. Lefering [6] used triple graph grammars to develop incremental integration tools for the coupling of requirements definitions and software architectures. Later on, triple graph grammars were applied in several software engineering projects outside the scope of the IPSEN project. In particular, they were used for the re-engineering of software systems. In the Varlet project [12], incremental integration tools were built for mapping relational to object-oriented database schemas. In ReforDi [13], tools were developed for migrating mainframe applications to a client-server architecture. Here, the structure graph of the original Cobol application was mapped to an object-based architecture with the help of triple rules. Both projects relied on the PROGRES environment [10] as the underlying specification and implementation machinery. Finally, [14] reports on an application of triple graph grammars in chemical engineering. To some extent, this work served as a starting point for the project described in this paper.

Graph transformations have been used for the specification of integration tools in a couple of other projects as well. Some of this work is devoted to *model transformations*, where a given model is transformed into another notation [15, 16]. Model transformation tools usually operate in batch mode without user interaction, i.e., they work like a compiler. In contrast, the applications we study demand for incremental, interactive integration tools. Closely related problems are studied in the ViewPoints project [17], which investigates methods and tools for maintaining consistency between related view points (documents in our terminology) in software engineering. Enders et al. [18] describe how consistency analysis and repair actions in the ViewPoints approach are specified with the help of distributed graph transformations. Here, the more restricted pair graph grammar approach coined by Pratt [1] is applied.

10 Conclusion

We have presented an industrial application of triple graph grammars. Incremental integration tools are used to maintain consistency between inter-dependent design documents in chemical engineering. We have also discussed the strengths and limitations of the triple graph grammar approach. In our future work, we will evaluate the tools which we built in cooperation with our industrial partner. Furthermore, we will generalize the framework such that it can be applied outside the chemical engineering domain.

References

1. Pratt, T.W.: Pair grammars, graph languages and string-to-graph translations. Journal of Computer and System Sciences **5** (1971) 560–595
2. Schürr, A.: Specification of graph translators with triple graph grammars. In Mayr, E.W., Schmidt, G., Tinhofer, G., eds.: Proceedings 20th Workshop on Graph-Theoretic Concepts in Computer Science WG 1994. LNCS 903, Herrsching, Germany, Springer (1994) 151–163

3. Nagl, M., Marquardt, W.: SFB-476 IMPROVE: Informatische Unterstützung übergreifender Entwicklungsprozesse in der Verfahrenstechnik. In Jarke, M., Pasedach, K., Pohl, K., eds.: Informatik '97: Informatik als Innovationsmotor. Informatik aktuell, Aachen, Germany, Springer (1997) 143–154
4. Becker, S., Haase, T., Westfechtel, B., Wilhelms, J.: Integration tools supporting cooperative development processes in chemical engineering. In: Proceedings International Conference on Integrated Design and Process Technology (IDPT-2002), Pasadena, California (2002)
5. Becker, S., Westfechtel, B.: UML-based definition of integration models for incremental development processes in chemical engineering. In: Proceedings International Conference on Integrated Design and Process Technology (IDPT-2003), Kumming, China (2003)
6. Lefering, M.: Tools to support life cycle integration. In: Proceedings of the 6th Software Engineering Environments Conference, Reading, UK, IEEE Computer Society Press (1993) 2–16
7. innotec GmbH: COMOS PT Documentation, http://www.innotec.de. (2003)
8. Aspen-Technology: Aspen Plus Documentation, http://www.aspentech.com. (2003)
9. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison Wesley, Reading, Massachusetts (1999)
10. Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES approach: Language and environment. [19] 487–550
11. Nagl, M., ed.: Building Tightly-Integrated Software Development Environments: The IPSEN Approach. LNCS 1170. Springer, Berlin, Germany (1996)
12. Jahnke, J., Zündorf, A.: Applying graph transformations to database re-engineering. [19] 267–286
13. Cremer, K., Marburger, A., Westfechtel, B.: Graph-based tools for re-engineering. Journal of Software Maintenance and Evolution: Research and Practice **14** (2002) 257–292
14. Cremer, K., Gruner, S., Nagl, M.: Graph transformation based integration tools: Applications to chemical process engineering. [19] 369–394
15. Baresi, L., Mauri, M., Pezzè, M.: PLCTools: Graph transformation meets PLC design. Electronic Notes in Theoretical Computer Science **72** (2002)
16. de Lara, J., Vangheluwe, H.: Computer aided multi-paradigm modeling to process petri-nets and statecharts. In Corradini, A., Ehrig, H., Kreowski, H.J., Rozenberg, G., eds.: Proceedings of the 1st International Conference on Graph Transformations. LNCS 2505, Barcelona, Spain, Springer (2002) 239–253
17. Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., Goedicke, M.: Viewpoints: A framework for integrating multiple perspectives in system development. International Journal of Software Engineering and Knowledge Engineering **2** (1992) 31–58
18. Enders, B., Heverhagen, T., Goedicke, M., Tröpfner, P., Tracht, R.: Towards an integration of different specification methods by using the Viewpoint framework. Transactions of the SDPS **6** (2002) 1–23
19. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools. Volume 2. World Scientific, Singapore (1999)