Fine-grained and Structure-Oriented Document Integration Tools are Needed for Development Processes

*S. Gruner * / M. Nagl / A. Schürr*
Lehrstuhl für Informatik III, Aachen University of Technology
Ahornstr. 55, D–52074 Aachen, Germany
E -Mail: {stefan / nagl / andy}@i3.informatik.rwth–aachen.de

Abstract: Development processes of various engineering disciplines are usually rather complex. They consist of many interacting subprocesses, which are carried out by different developers. Each subprocess delivers its own documents, which are part of the overall result. All involved documents and their mutual, fine-grained dependencies are subject to permanent changes during the life-time of their development process. Keeping these documents in a consistent state is one of the most important prerequisites for the success of any engineering project.

As completely automatic change control between documents is often impossible, interactive consistency monitoring and (re-)establishing tools are necessary, which we call integration tools. This paper reports about experiences in building integration tools for software engineering environments and about ongoing efforts to build similar integration tools for chemical process engineering. Furthermore, the paper presents an object-oriented and graph-grammar-based formal method for specifying integration tools and sketches how their implementations are derived from their high-level specifications.

Key words: development processes, product integration, tool specification, fine-grained interdocument relations, coupled graph grammars

1. Development Processes and Their Results

Development Processes (DP) in areas such as software development, computer integrated manufacturing, or chemical process control usually involve different developers. Each developer produces a certain set of documents, which is part of the overall DP result. His documents have to be kept in a consistent state with documents produced by other developers. Between documents, directed and mutual consistency dependencies have to be taken into account. A software design specification, which depends on a requirement specification, is an example of a directed dependency. The different perspectives of a requirement specification—such as a data-oriented view and a function-oriented view—are an example for mutual document dependencies. Simultaneous engineering /BW 96/ aims at accelerating DPs by starting dependent subprocesses as early as possible with preliminary results (prereleases) of preceding subprocesses. Concurrent engineering /Re 93/, on the other hand, allows to develop different perspectives of the same product part in parallel.

* Stipendiat der Deutschen Forschungsgemeinschaft im Graduiertenkolleg für Informatik und Technik der Rheinisch-Westfälischen Technischen Hochschule Aachen

1.1 Development Subprocesses and Their Results

A key problem in the development of any engineering product is *change control*, especially in the case of simultaneous or concurrent engineering. Changes are carried out due to detected errors, due to changed design decisions but also, in the extreme case, due to changed requirements in an ongoing project. In the course of a usual development process many errors are made and the construction of required results is often not straightforward. Therefore, we can state that both development and maintenance of engineering products have to deal with permanent changes of intermediate or final DP results.

In any application area mentioned above *complex document configurations* are built up and maintained. They do not only consist of the final configuration (e.g. the source code of a software system) but also of many further subconfigurations responsible for describing the requirements, the architectural plan, documenting the developed ideas and met decisions, assuring quality, or managing the whole development process. Such an overall configuration consists of many documents which, in turn, may have a complex inner structure.

Produced documents often have many fine-grained dependencies between their constituents. For the quality of the whole product and the efficiency of the total process these dependencies are of minor importance. A single developer is usually responsible for the internal consistency of a document. He should be able to keep all local consistency requirements in mind and, in many cases, he is supported by suitable document processing tools such as a syntax-directed diagram editor or a CASE analysis tool. The *fine-grained dependencies between documents* correspond to the interfaces between the work of different developers. As we shall see in section 2, no suitable support is available for keeping these interdocument dependencies in a consistent state. Therefore, we concentrate on this problem in the following. We use the relations between a requirement specification and the design of a software system as a running example.

For *coordinating* a *team* of *developers,* management information (administration configuration) about a project has to be built up and maintained. We distinguish between process, product (configuration and version), resources, and department or company information, and we regard their mutual relations in order to coordinate the labour of a team of developers. Management in this sense has to be supported by suitable tools. There, interesting problems arise as the administration configuration is changed in its structural form, when a development process is carried out /Kra 98/. However, in this paper we concentrate on the support of technical developers and the interfaces of their results.

Many DP documents, which are the results of technical development subprocesses, have a *semiformal contents*. Some documents are completely informal, as the nonfunctional requirements specification, where we find plain text possibly presented in a standardized (sub-)chapter form. Very few documents are formal, as e.g. a specification in a logic-based language such as Z. A standard case is that we find documents in diagrammatic, tabular, pictorial, or textual form which altogether possess an underlying structure and a more or less well-defined syntax. Examples of this kind are OO-analysis diagrams, module descriptions in a software design description language, and so on. So, ”semiformal” either means that formalization of a certain DP result is not carried out completely or, even more, that the underlying document description language is only formalized to a certain extent.

Having most documents in a semiformal form, the *fine-grained relations* between documents are *semiformal*, too. Often, we can state that a certain increment (subpart) of one document may be related to an increment of another document if both increments are instances of corresponding types, have compatible properties, and appear in a certain context within their documents. A technical documentation may, for instance, contain a section for each module of a related software design document, and a section may contain a subsection for each exported resource of the related module.

1.2 Preserving Consistency of Dependent Documents

Language correspondences have to be elaborated, which define legal interdocument relations, before supporting tools can be built /Jan 92/. This is usually called *method integration* /Kro 93/. In some cases method integration rules require that instances of some type T_A in a document A are always related to instances of type some T_B in a document B, as it was the case with modules and sections in software design and technical documentation documents, respectively. This is called a bijective (1:1) correspondence between increment types of documents. In many cases, we have (m:n) correspondences on the type level such that further information is needed in order to decide whether an increment of type T_A of document A may be related to an increment of type T_B in document B. Such a decision may depend on

- local properties of inspected increments,
- their contexts in the regarded pair of documents,
- and manual design decisions of an involved developer.

Usually, consistency establishing subprocesses of a development process *cannot be automated*. The development of some document B_i, which is the result of one subprocess, often depends on the result of another subprocess, some master document A, in a rather imprecisely defined way. Exceptions are generating an NC program from a CAD document, generating module frames from a software design document etc. In these cases, the contents of involved documents are closely related to each other and a complete formal definition of the corresponding interdocument dependencies is feasible.

The standard case is that *subprocesses are creative* in the sense that a developer is not able to come up with a precise and complete formal definition of a procedure (method, plan) how changes of a master document A have to be translated into corresponding changes of dependent documents B_i (cf. figure 1). As an example, regard the development of a coarse-grained software design for a given requirements specification. The design may be one of the ”structured” world, an object-based or an object-oriented one /Nag 90/, and it is influenced by many factors such as the underlying middleware or the decision to (re-)use certain libraries or frameworks. As a consequence, there is no chance to automate the transition from requirements engineering to software design completely. However, the transition can be simplified by tools, which perform trivial subtasks on their own and keep track of once established relations between the requirements for and the design of a software system (see below).

Changes within a master document A require changes in a dependent document $B_{i\,,}$ which then require further changes in dependent documents of B_i. There are many different possibilities how to translate an update of a master document into updates of its dependent documents. Furthermore, rather different strategies may be used to propagate necessary changes along chains of dependent documents. One possibility is called *batch-oriented*. It reestablishes first complete consistency between a document A and all its directly dependent documents B_i w.r.t. a sequence of updates on A, before proceeding with the dependent documents of B_i. Another possibility is called *trace-oriented*. It propagates one performed update on a document A to all directly or indirectly affected dependent documents after the other. Both strategies have their specific advantages and disadvantages and should be supported by integration tools.

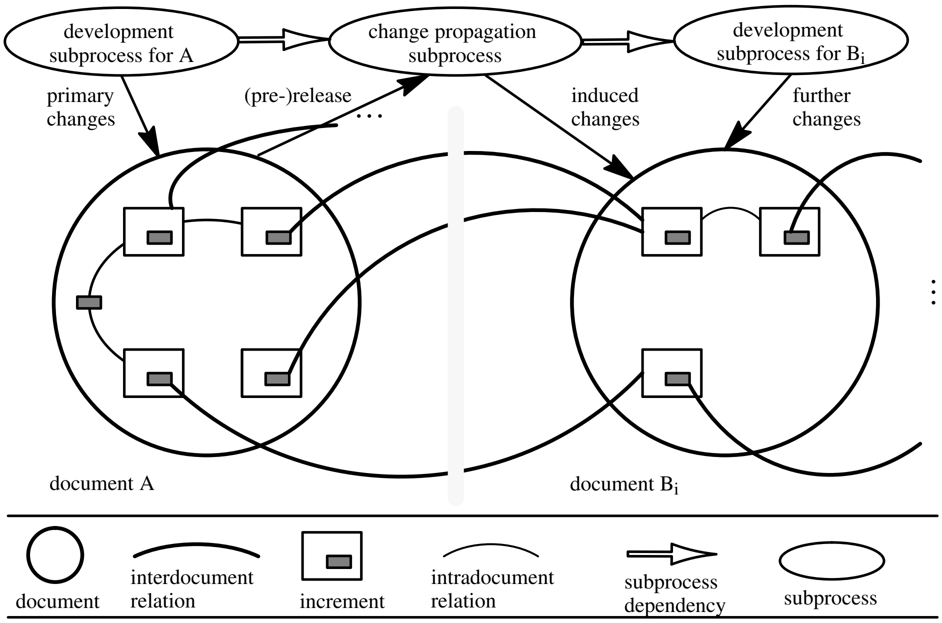

Fig. 1: Dependent documents and their development subprocesses.

The following sections discuss the specification and realization of various types of *integration tools,* which are responsible for monitoring and (re-)establishing consistency of dependent documents. These tools have to regard the semiformal structure of corresponding documents. They have to give substantial support for fine-grained integration by regarding the current form of documents and offering different possibilities how to propagate changes, which are selected based on creative design decisions. Furthermore, they must not enforce certain orders of process steps as quite different consistency reestablishing strategies are possible. Finally, it should be possible to work with existing and a posteriori integrated tools, when manipulating the corresponding documents.

2. Available Support for Interdocument Consistency Control

The available support for monitoring and maintaining consistency of related documents on a fine-grained technical level is usually on a considerably lower level than the kind of tight integration sketched in the previous section. The standard procedure is that developers exchange documents in some low-level *standard format* (Postscript, SGML, HTML etc.). In any case, the developer of a dependent document has to find out which changes have taken place on a master document A and then to perform the right changes on the dependent documents B_i.

Another wide-spread approach, especially in the software engineering community /SB 93/, is to write *batch-oriented and automatically working converters* after the corresponding method integration has taken place. So, neither incremental changes of some master document A can be handled, nor do they regard that its dependent documents B_i are already elaborated to a certain state, nor can creative design decisions of developers be taken into account. Furthermore, such transformers are often hand-coded. As many documents in different languages as well as different method integration approaches exist, hand-coding of integration tools has to be replaced by generating them from high-level specifications.

A similar problem occurs with *document exchange standards* such as STEP with its data modeling language EXPRESS /ISO/ or CDIF /CDIF 94/. They define huge class diagrams (data models) for certain types of engineering documents, but disregard consistency relations between different types of documents to a great extent. The data modeling language EXPRESS allows, for instance, to define the data model of each type of documents as a separate module and to import the data model of one module into another one. Furthermore, EXPRESS offers rules for defining static integrity constraints across document boundaries. But it is very difficult to derive consistency establishing operations from these static integrity constraints. As a consequence, STEP/EXPRESS data models are not a suitable source of input for generating integration tools.

Also *hypertext systems,* as introduced in /Co 87/, do not offer an appropriate solution for preserving the consistency of a set of related documents. They have no knowledge about the semantics of links and just offer basic mechanisms to insert unspecific links and to browse along them. Consistency control is on a low level, namely detection of dangling links. Finally, all links have to be established manually by the user. He has no profit if he is forced to insert them manually and then only gets warnings that a part of them are dangling.

More refined concepts can be found in (meta) software development environments, where documents are internally represented as *attributed syntax trees*. Support is given for propagating changed attribute values up and down the syntax tree. This allows to specify and generate analysis tools, which check interdocument consistency constraints /KKM 87/ if all regarded documents are modeled as subtrees of a common syntax tree. Other systems offer better support for the required nesting of documents. Gate nodes and door attributes of a distributed syntax tree model the transition from one document's language to another one /Bo 88/.

All syntax tree based systems mentioned above have problems with the specification of *active transformation tools* (in contrast to passive consistency checking tools). Attribute coupled grammars and variants thereof /GG 84, RT 88/, tree pattern matchers /AGT 89/, and context-sensitive tree transformation tools /CC 93/ are promising attempts to overcome these problems. They are useful for purposes like concrete syntax generation (unparsing) or compiler back-end generation. The still remaining problem with these approaches is that generated transformation tools are unidirectional, batch-oriented, and not interactive.

Federated database systems /SL 90/ represent another form of data integration. They offer a common global schema for different local database systems, which usually is used for retrieval of data but not for updates (update problem). Active database systems /DHW 95/ offer event-trigger mechanisms to keep databases consistent. Event-trigger mechanisms play about the same role for the data-oriented integration paradigm as messages do for the control-oriented integration paradigm of broadcast message servers /Rei 90, Fro 90, Fra 91/. Event-trigger mechanisms and broadcasted messages are still rather *low level means* to simplify the implementation of integration tools and to propagate updates between related documents or document processing tools.

For the sake of completeness we should mention the existence of tools, which coordinate development subprocesses and their results on a *coarse-grained level.*, i.e. without taking the internal structure of documents and the fine-grained interdocument relations between them into account. There, we find CAD frameworks /HN 90/ or EDM systems /McI 95/ for managing project databases. Workflow systems /GHS 95/ coordinate development subprocesses. Furthermore, we find configuration and version control tools on coarse- as well as fine-grained level. However, the latter are usually unspecific w.r.t. the structure of their documents /NW 98/.

3. Experiences in Building Integration Tools

Our experiences in building integration tools date back to 1988, when the implementation of our first integrated CASE tool prototype was finished /Lew 88/. Its most important integration tool has the task to keep a program's design in a consistent state with its technical documentation. This tool can be used in two different modes. Its *free format mode* allows to create arbitrary links between increments of the design document and sections or paragraphs of the accompanying technical documentation. In this case, the tool supports browsing along hyperlinks and issues warnings that once created hyperlinks are now dangling or that sources or targets of hypertext links are modified. The integration tool's more sophisticated *fixed format mode* enforces a structure of the technical documentation, which is closely related to the structure of its software design document. Any module of the design document corresponds to a section of the technical documentation, any exported module resource to a paragraph of the enclosing section. Section headlines are automatically derived from module names.

The (software design, technical documentation) integration tool realizes therefore a combination of a hyperlink *browser*, a consistency *checker*, and a consistency re-establishing *transformation tool* (as any other integration tool presented here). Updates of the design document are immediately propagated as *change messages* to the dependent technical documentation. These messages are then asynchronously processed and—partly automatically, partly manually—translated into appropriate updates of the technical documentation. The integration tool was manually implemented without any kind of reuse of basic components and without a formal specification of its expected behavior. Needed hyperlinks were stored as pairs of unidirectional pointers in related documents.

Some years later an incrementally and automatically working (software design, Modula-2) integration tool was built using a more elaborate specification and implementation approach /Wes 91/. First of all, the EBNF syntax definitions of both involved types of documents were related to each other and then manually translated into a *programmed graph rewriting specification* of the integration tool's functional behavior. Based on the graph rewriting specification, an integration tool was handcrafted, which keeps a software system's design and the corresponding configuration of Modula-2 implementation documents in a consistent state. The main progress of this integration tool—compared with the (software design, technical documentation) integrator—is that hyperlinks are now stored in a separate *integration document*. This allows the integration of already implemented types of documents more easily, simplifies multi-user access to related documents, and offers the appropriate database for storing information about an ongoing integration process.

The needs to build an integration tool, which keeps requirements engineering documents and software design documents in a consistent state, forced us to generalize the integration tool specification and implementation approach /Jan 92/. The most challenging feature of the (requirements engineering, software design) integration tool was the required interaction between the computation of applicable transformations rules and the manual selection of actually applied rules. This is due to the fact that consistency relations between Structured Analysis (SA) diagrams and Entity Relationship (ER) diagrams on one hand and design documents on the other hand are rather vague and context-dependent. An SA data flow diagram (DFD) may be translated into a module or a procedure of a design document depending on the number of applied occurrences of the DFD in its SA document, the way how related SA increments were already translated into software design increments, and creative design decisions of the involved integration tool user.

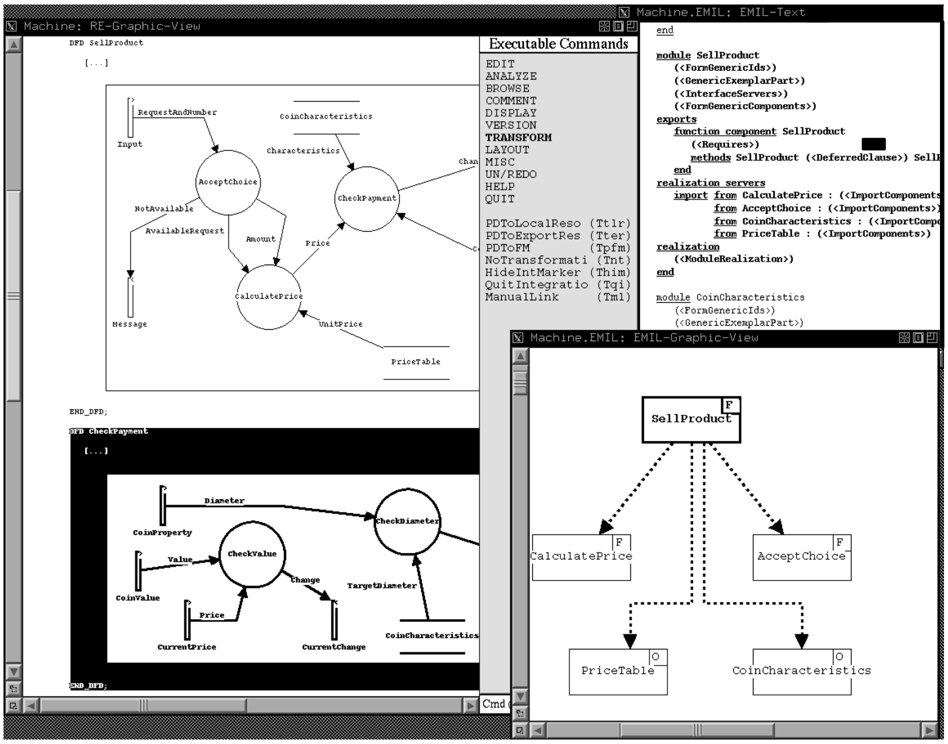

Fig. 2: Requirements engineering and program design integration tool.

Figure 2 shows one example of an SA document and a graphical as well as a textual view of its related design document. The top left DFD SellProduct contains the three processes AcceptChoice, CheckPayment, and CalculatePrice as well as the two data stores CoinCharacteristics and PriceTable. The latter two provide needed input data for CheckPayment and CalculatePrice, respectively. The DFD in the left bottom corner of figure 2 represents the refinement of the process CheckPayment. All involved DFDs—except CheckPayment—are translated into so-called F(unction) modules, all data stores into so-called O(bject) modules, and the occurrence of a certain process or data store Y in a DFD X corresponds to an import between the related modules X and Y. The data flows between processes as well as the input and output ports of DFD SellProduct (the small rectangles labeled Input and Message on its left-hand side) are disregarded on this level of granularity. They may be useful later on for determining formal parameter lists of generated procedures.

The missing module CheckPayment together with all its dependent components is just under construction. The black background color of DFD CheckPayment in the SA document informs the integration tool's user that this SA increment has not yet any counterpart in the corresponding program design. It is up to the user to decide whether it is worthwhile to build a separate module for CheckPayment, too, or whether the functionality of CheckPayment may be implemented as a single local function of module SellProduct.

The syntax of the regarded requirements engineering and software design documents with their diagrammatic notations was no longer defined in the form of EBNFs, but in the form of (extended) ER diagrams. As a consequence, a new *meta modeling approach* had to be invented to identify corresponding entity types of ER diagrams instead of corresponding nonterminal classes of previously used EBNFs. This approach to relate entity types of different (data base) schemas by deriving them from the same meta class has been adopted by the data base community for solving data base migration problems /JJ 95/.

Unfortunately, it is often not possible to derive consistency checking or document transformation code directly from constructed meta models. The problem is that entity type correspondences, such as DFD is either related to module or to procedure, are not precise enough to define the wanted behavior of integration tools. Therefore, we returned to the idea presented in /Wes 91/ to describe the syntax of documents by means of grammars and to specify dependencies between documents by coupling these grammars. The main difference between the old proceeding in /Wes 91/ and the new proceeding in /Lef 95/ is that the former one uses context-free string grammars (EBNFs) for this purpose, whereas the latter one is based on *context-sensitive graph grammars*, as already suggested in /Pra 71/.

Following this approach, the specification and realization of document integration tool proceeds now as follows:

1. The internal structures of dependent documents are modeled as *directed graphs*, which contain different classes (types) of attributed nodes and edges for different kinds of increments and intradocument relations.
2. UML[1] *class diagrams* are then constructed to define the relevant components of regarded document graphs from a static point of view.
3. Next, *correspondences* between UML class diagrams are established in accordance with the meta modeling approach of /Jan 92/. They identify possibly related increment classes (node types) of dependent documents.

1. UML, the Unified Modeling Language, is the new standard notation for object–oriented analysis and design. It was developed by Rational Rose and is now an accepted standard of the Object Management Group OMG /FS 97/.

4. Afterwards, *object diagrams* are used to define corresponding substructures (subgraphs) of related document graphs more precisely on the instance level.
5. These object diagrams are translated into *graph grammars* that generate those subsets of schema consistent document graphs, which consist of previously defined object diagrams only (thereby excluding intermediate inconsistent or incomplete document editing results as valid integration tool inputs).
6. Finally, the constructed *graph grammars are coupled* such that each production of a master document graph is related to a set of productions of a dependent document graph.

All needed kinds of integration tools may be derived from a single coupled graph grammar specification of the corresponding interdocument consistency relations. This includes a *forward transformation* tool, which propagates updates from a master document to the dependent document, a *reverse transformation* tool, which propagates updates from a dependent document back to its master document, or a pure *analysis* tool, which checks (traces) consistency of dependent documents without changing their contents.

Up to now, all integration tools are manually derived from a given coupled graph grammar specification, based on a *reusable framework* for the construction of integration tools /Nag 96/. The reusable framework offers, for instance, various forms of document traversing and comparing strategies as well as a standard implementation of integration documents. These integration documents, which were first introduced in /Wes 91/, are now used for storing all hyperlinks between two dependent documents together with all design decisions of users how to translate updates of master documents into updates of dependent documents.

It is the subject of ongoing research activities to translate coupled graph grammar specifications, which define interdocument consistency relations in a purely declarative manner, automatically into *programmed graph rewriting specifications,* which define the functional behavior of specific integration tools in the form of complex document graph transformations /JSZ 96, Gru 97/. These graph rewriting specifications may be translated into equivalent C or Modula-2 code, using the compiler of the PROGRES graph grammar development environment /SWZ 95/.

The presented integration tool development process is explained in more detail in the following section 4. It was already successfully used for realizing another version of the (requirements engineering, software design) integration tool of figure 2 as well as for realizing new integration tools between software designs and Eiffel programs or between SA and ER diagrams /Lef 95/. Furthermore, related graph grammar based approaches were used for translating relational DBMS schemas into object-oriented DMBS schemas /JSZ 96/ and for integrating commercial SA editors with a research prototype for high-level timed Petri nets /BOP 97/. Last but not least a refined version of the presented integration approach is currently used for integrating chemical process engineering tools /NM 97/.

4. Specifying Interdocument Consistency Preserving Integration Tools

The previous section presented a number of graph grammar based integration tools and sketched their development. It is the purpose of this section to explain the graph grammar based development of integration tools in more detail, using the running example of the (requirements engineering, software design) integration tool. As already mentioned, the development of such a tool starts with modeling all involved types of documents as *directed graphs*. Different types of increments—such as DFD or Process—correspond to different types (classes) of nodes, different types of intradocument relations—such as DFD contains Process or DFD defines Process—are introduced as different types of directed edges (associations). Furthermore, node attributes are needed to represent local properties of increments, such as the Name of a DFD or a Process.

4.1 UML Class and Object Diagrams Define Interdocument Relations

The needed components of document graphs and their relations are introduced as so-called *graph schemas*. It is a matter of taste whether an ER-diagram-like notation or an UML-class-diagram notation is used for this purpose. Within this paper, we prefer the upcoming standard OO notation UML, which allows to draw class diagrams as well object diagrams /FS 97/. Furthermore, UML offers the concept of packages, which allows to encapsulate document graph schemas and to distinguish between local and externally visible document graph components.

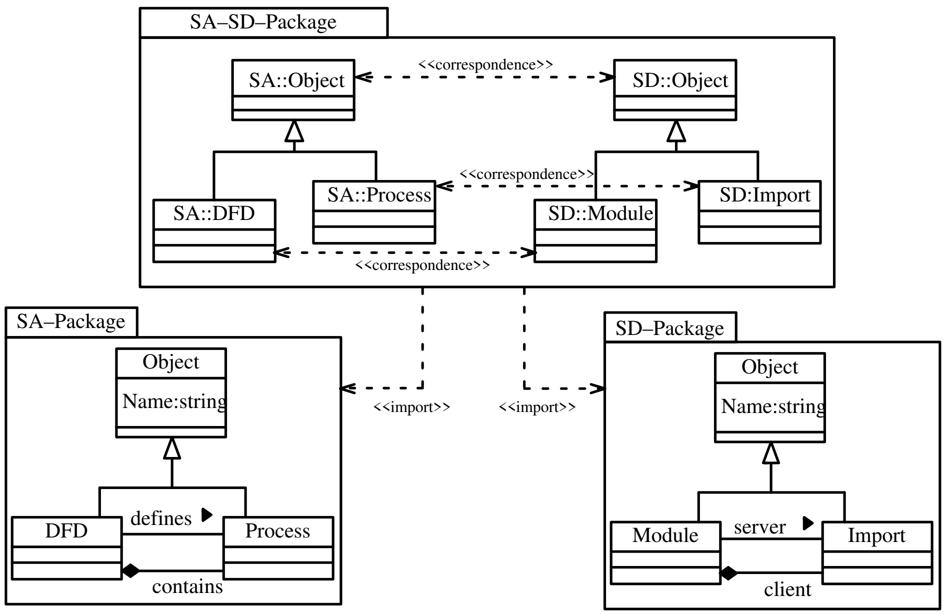

Fig. 3: Corresponding SA and SD document graph schema definitions.

Figure 3 shows a cut-out of the graph schema definitions for Structured Analysis (SA) and software design (SD) documents. Its packages SA and SD display only those definitions of classes and associations which are needed for the translation of a DFD and a Process into a Module and an Import, respectively. Both packages introduce a superclass Object to declare a Name attribute for DFD and Process on one side and for Module and Import on the other side. Please note that the Import relation is modeled as part of its client Module (source) and possesses the Name of its server Module (target) as its own Name. This reflects the way how import relations (clauses) are defined in the textual software design document representation of figure 2.

The additional SA–SD package imports all externally visible classes of its dependent packages SA and SD and introduces the required *class correspondences* between the two (abstract) Object superclasses and between their subclasses DFD and Module as well as between their subclasses Process and Import. Many classes and class correspondences have been omitted in figure 4 due to lack of space, as e.g. the SD class Procedure and its correspondence to the SA class DFD.

Some *constraints* have been developed concerning legal and illegal combinations of generalization relationships with graph schema crossing correspondence relationships. These constraints, defined in /Gru 97/, prohibit e.g. the definition of a correspondence relationship between the classes SA::DFD and SD::Object in the presence of a correspondence relationship between the classes SA::Object and SD::Module. This is considered as a contradiction between the requirement that any SA::Object is mapped onto a SD::Module and the fact that a SA::DFD, a special kind of SA::Object, may be mapped onto any SD::Object.

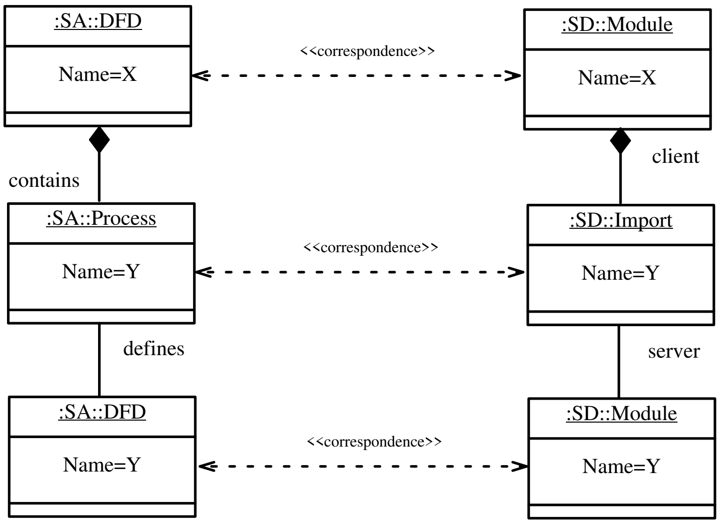

Fig. 4: Object diagram definition of corresponding SA and SD subgraphs.

Based on graph schema correspondences, which define a superset of all possible relations between SA increments and SD increments, it is now necessary to identify existing interdocument consistency relations more precisely. Experiences showed that *object diagrams* are the most appropriate notation for this purpose. They allow one to define pairs of subgraphs (subpatterns, substructures) on the instance level, which relate certain configurations of SA increments to corresponding configurations of SD increments and vice-versa. Figure 4 presents one example of this kind. It states that a DFD X, which contains a Process Y with its own DFD definition, may be related to a Module X, which contains an Import clause for Module Y. Furthermore, it requires that the DFD instances X and Y correspond to Module instances X and Y, and that the Process instance of Y in DFD X corresponds to the Import clause for Module Y in Module X.

Many object diagrams of this kind are necessary to define the set of all relevant interdocument relations between SA and SD document graphs. Unfortunately, it is not possible to establish useful *consistency or completeness criteria* for the resulting set of object diagrams in the general case. This is due to the fact that subgraphs of different object diagrams may overlap and that certain document subgraphs on one side may not have corresponding document subgraphs on the other side. Our running example requires e.g. the construction of another object diagram, which relates the same SA subgraph as in figure 4 to a Module X, which contains a Procedure Y. As a consequence, it is not possible to interpret the constructed set of object diagrams as the consistent definition of a deterministic function, which translates SA documents into SD documents. The completeness criteria is violated by the fact that there may be some SA documents without corresponding SD documents. There is perhaps no rule how to translate a DFD X, which contains a Process X as a forbidden self-reference, into a corresponding SD document substructure.

4.2 From Class and Object Diagrams to Coupled Graph Grammars

The UML class diagrams and object diagrams of the previous subsection are the appropriate means to discuss the functionality of an integration tools with its future users. These users have the needed knowledge about the regarded application domain for building the appropriate set of class and object diagrams as well as for checking consistency and completeness of these diagrams. The following step of the integration tool building process is on a more technical level and may be performed without any assistance of application domain experts. It concerns the translation of a set of object diagrams into a *coupled graph grammar specification*. Each object diagram is translated into one or more coupled graph grammar productions. Each coupled graph grammar production is a pair of two regular graph grammar productions plus the definition of a correspondence relationship between the nodes on the left- and right-hand sides of the combined productions.

Figure 5 shows two examples of coupled graph grammar productions, which were produced by taking the object diagram of figure 4 as input. The main problem of the transition from object diagrams to coupled graph grammars is that we have to distinguish between *context nodes*, which are part of a production's left- and right-hand side, and *new nodes*, which are only part of a production's right-hand side. It is, for instance, not useful to define a graph grammar production, which creates a Process Y as part of a DFD X together with its DFD definition Y. As a consequence, we would not be able to deal with a DFD without any applied Process occurrences or with more than one occurrence. It is, therefore, better to translate the object diagram of figure 4 into two coupled graph grammar productions, as presented in figure 5. The first one, Create–DFD&Module, consists of two regular graph grammar subproductions, which have connected grey rectangles as background. Both subproductions have an empty left-hand side and a single node on the right-hand side. The SA subproduction creates an isolated DFD node with Name = Y in the SA document graph, the SD subproduction a corresponding Module node with the same Name in the related SD document graph.

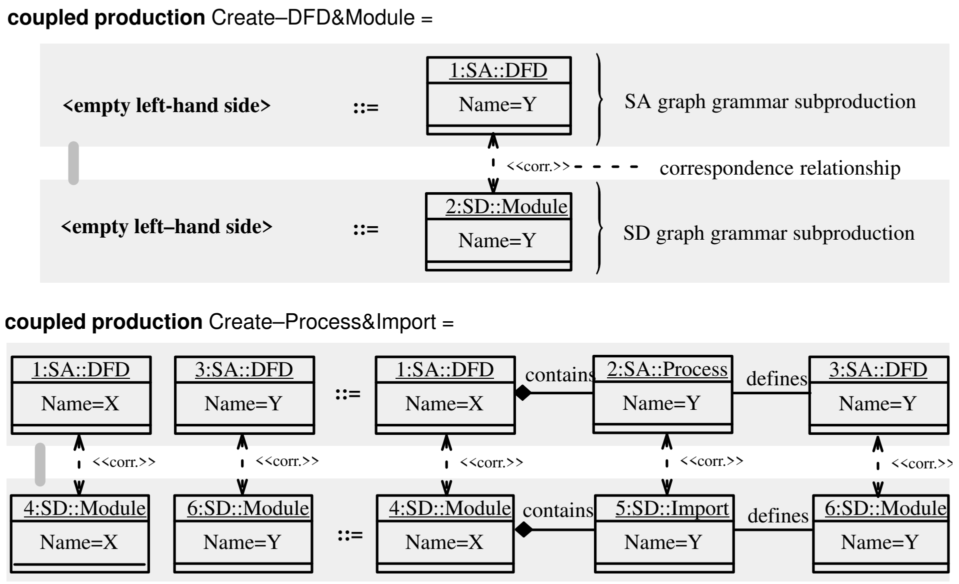

Fig. 5: Two coupled graph grammar productions derived from figure 4.

The following coupled graph grammar production Create–Process&Import of figure 5 matches two pairs of corresponding DFD and Module nodes with the left-hand sides of its two subproductions. These nodes together with their correspondence relationship are preserved by the given subproductions due to the fact that the defined right-hand sides contain their left-hand sides as subgraphs. Furthermore, the coupled subproductions create a new Process occurrence of DFD Y in DFD X as well as a corresponding Import clause with Module X as client and Module Y as server.

4.3 From Coupled Graph Grammars to Integration Tool Specifications

As already mentioned, one coupled graph grammar serves as the specification for a number of related but nevertheless quite differently behaving integration tools. The same coupled SA–SD graph grammar may for instance be used to develop an incrementally working forward engineering tool, which translates SA document updates into SD document updates, and a batch-oriented reverse engineering tool, which takes a complete SD document as input and produces a corresponding SA document as output. This a consequence of the fact that coupled graph grammar productions do not distinguish between master documents and dependent documents and that they do not prescribe how and when consistency between dependent documents is (re-)established.

Coupled graph grammars, which are constructed using a set of object diagrams as input, never contain node or edge deleting productions, i.e. the left-hand sides of their productions are always subgraphs of their right-hand sides. The restriction to non-deleting productions is not as severe as it seems to be at a first glance. This is due to the fact that the productions of a coupled graph grammar are *not the editing operations* for the involved types of documents. Document graph editing operations are defined without having certain interdocument consistency relations in mind. Furthermore, they do not only create, but modify and delete document components. Last but not least the editing operations for a certain type of document graphs may be used to manipulate subgraphs, which are irrelevant and therefore hidden for the regarded document integration task (cf. /Nag 96/ for further details concerning the definition of document views for integration tools).

The restriction to non-deleting productions is the necessary prerequisite for being able to derive *different kinds of (efficiently working) integration tools* from one coupled graph grammar specification. Otherwise, we would be forced to parse pairs of document graphs w.r.t. to unrestricted types of graph grammars in order to be able to check interdocument consistency relations (i.e. to solve the membership problem for type 0 grammars, which is undecidable in the general case). Having the restriction to non-deleting productions in mind, it is possible to translate coupled graph grammar productions into different sets of ordinary graph transformation rules. One set of transformation rules defines the functional behavior of a forward transformation integration tool, another one the behavior of the corresponding reverse transformation integration tool, a third one the behavior of a consistency checking tool, and so on.

Figure 6 shows one example of a derived SA–SD *forward transformation rule*. It translates a Process occurrence of DFD Y in DFD X of the SA document into an Import between the related Modules X and Y of the SD document. This rule was constructed as follows: its left-hand side is the combination of the right-hand side of the corresponding SA production with the left- and side of the corresponding SD production in figure 5, its right-hand side is the combination of the right-hand sides of the two coupled SA and SD productions.

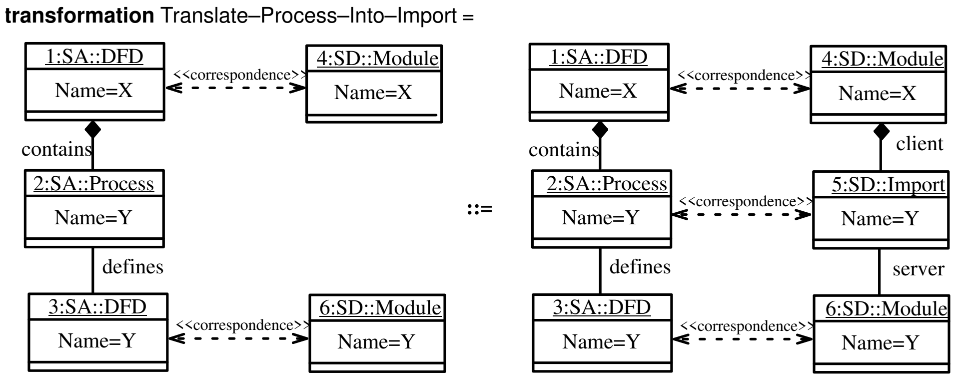

Fig. 6: A forward graph transformation rule from SA to SD.

An SD–SA *reverse transformation rule* may be constructed by simply exchanging the roles of SA and SD productions in the previous paragraph. A consistency checking and correspondence relationships establishing transformation rule may be built by merging the constructed forward and reverse transformation rules.

4.4 From Integration Tool Specifications to Implementations

All generated transformation rules have to be combined with a *reusable framework*, which determines the order of rule applications and which processes needed user interactions. Furthermore, the framework provides a certain bookkeeping strategy for not yet transformed document parts or already transformed but afterwards changed document parts. Please note that ”real” graph transformation rules are more complex than the one presented in figure 6. They use a more sophisticated representation of correspondence relationships (as nodes and edges of separate integration document graphs) and manipulate therefore three related subgraphs of a hierarchical graph instead of one flat graph only /Nag 96/.

The finally needed translation from forward or backward graph transformation rules to *efficiently executable C or Modula-2 code* is supported by the PROGRES graph grammar environment /SWZ 95/. This environment is available as free software on the world-wide-web page

http://www–i3.informatik.rwth–aachen.de/research/progres/index.html.

It represents an integrated set of tools for syntax-directed editing, analyzing, and executing single graph transformation rules or complex graph transformation programs. Two execution modes are supported: (1) direct interpretation of created specifications and (2) compilation into lower level programming languages such as C and Modula-2. Generated program fragments may be combined with a hand-crafted code, such as the integration tool framework mentioned above.

For further details concerning the construction of graph transformation rules from coupled graph grammars and the implementation of the needed framework the reader is referred to /Nag 96, JSZ 96/.

5. Formal Background of Coupled Graph Grammars

The preceding sections introduced a graph grammar based method for the specification and implementation of document integration tools on a rather informal level. A complete formal definition of graph grammars, their underlying graph data models, and the definition of appropriate graph grammar coupling mechanisms is outside the scope of this paper. The interested reader is referred to /RS 96/ for a formal treatment of restricted types of *graph grammars* as a visual language syntax definition and parsing formalism and to the chapter 7 of the Handbook of Graph Grammars /Sch 97/ for the formal definition of a very general class of *programmed graph transformation systems*.

For further details concerning the usage of programmed graph transformation systems as a tool specification and implementation mechanism the reader is referred to /Nagl 96/. Further information concerning the design and implementation of the very high-level programming *language PROGRES* and its programming environment may be found in /SWZ 95/.

Last but not least the reader is referred to /Sch 94/ for a formal definition of *coupled graph grammars*, which is based on a very simply graph data model (without different types of nodes or edges and without attributes) and a simple form of graph grammar productions. A formal definition of coupled graph grammars for a more complex UML-compatible graph data model and more complex forms of productions is under development. Its first version, published in /Gru 97/, provides the formal background for the definition of correspondences between graph schemas (UML class diagrams) and the definition of coupled graph grammar productions, which respect the previously defined graph schema correspondences.

6. Summary and Future Work

Development processes of various engineering disciplines are usually rather complex. They consist of many interacting subprocesses, which are carried out by different developers. Various approaches are propagated nowadays how to support developers in executing their subprocesses and how to guarantee the overall consistency of their results, rather complex configurations of dependent technical documents. One may use the experience of human developers

- by recording traces of successfully executed subprocesses and transforming them into repeatable process chunks /PDJ 94/,
- by offering means for direct multi-media communication, which are tightly integrated with technical document manipulating tools /Her 96/,
- and by realizing interactive integration tools, which help their users to monitor and (re-)establish interdocument consistency relations.

These three different approaches complement each other and may be combined with appropriate management tools to *support engineering processes* on a very high level /NW 98/.

This paper had a main focus on *document integration tools*. It presented a graph grammar based method for deriving efficiently working integration tool implementations from very high-level interdocument consistency specifications. It is worth-while to notice that the presented method requires *not a complete formal specification* of the (semantics of the) considered types of documents and the involved modeling languages and methods. It is sufficient if all regarded documents have a well-defined internal structure (syntax definition) and if we have some knowledge about possibly corresponding patterns of increments in related documents.

The *functionality of presented integration tools* varies from a low-level hypertext editor, where all interdocument relations have to be created and maintained manually (if almost no knowledge about interdocument consistency relations is available) to an automatically working document transformation tool (if interdocument consistency relations may be defined as a total and deterministic function). In many cases the available knowledge about regarded document dependencies lies between these two extremes, such that the resulting integration tools are able to perform trivial consistency (re-)establishing tasks on their own and to compute sets of possible consistency (re-)establishing actions in the general case.

It is the subject of ongoing research activities to generalize the presented coupled graph grammar formalism w.r.t. the form of permitted coupled subproductions. Furthermore, we are planning to realize a new generation of *graph grammar coupling tools*, based on the experiences reported in /JSZ 96/ with a rather ad hoc approach to use the PROGRES environment for this purpose. These tools will offer appropriate support for entering graph schema correspondences and coupled graph grammar productions, for checking the consistency between graph schema correspondences and coupled productions, and for translating coupled graph grammar specifications into ordinary PROGRES specifications of needed integration tools.

Finally, we are making our first experiences with changing our focus from tightly integrated software engineering environments to tightly *integrated chemical process engineering environments* /NM 97/. The new application domain forced us to complement coupled graph grammars with UML class and object diagrams as more appropriate means of communication between domain experts and future users of integration tools and their developers. But the main challenge of the new application domain is that we have to generalize the presented integration approach from the a-priori integration of self-developed (software engineering) tools to the *a-posteriori integration* of already existing chemical process engineering tools.

References

/AGT 89/ A. V. Aho / M. Ganapathi / S.W.K. Tijang: Code Generation Using Tree Matching and Dynamic Programming, TOPLAS 11, 4, 491–516 (1989).

/Bo 88/ P. Borras et al.: Centaur: The System, in P. Henderson, Proc. 3rd, ACM Softw. Eng. Symp. on Practical Software Development Environments, ACM Softw. Eng. Notes 13,5, 14–24 (1988).

/BOP 97/ L. Baresi / A. Orso / M. Pezzè: Introducing Formal Specification Methods in Industrial Practice, in: Proc. 19th In.t Conf. on Software Engineering (ICSE'19), IEEE Computer, 56–66 (1997).

/BW 96/ H.J. Bullinger / J. Warschatt (Eds.): Concurrent Simultaneous Engineering Systems, Berlin: Springer (1996).

/CC 93/ J.R. Cordy / J.H. Carmichael: The TXL Programming Language Syntax and Informal Semantics Version 7, Technical Report 93–355, Computing and Information Science at Queen's University, Kingston (1993).

/CDIF 94/ EIA / CDIF: CDIF Family of Standards (1994).

/Co 87/ J. Conklin: Hypertext: An Introduction and Survey, IEEE Computer, 17–41 (1987).

/DHW 95/ U. Dayal / E. Hanson / J. Widom: Active Database Systems, in Kim (Ed.): Modern Database Systems, 434–456, New York: ACM (1995).

/Fra 91/ B. Frankel: The ToolTalk Service, Sun Microsystems (1991).

/Fro 90/ D. Fromme: HP Encapsulator: Bridging the Generation Gap, HP–Journal, 59–68 (1990).

/FS 97/ M. Fowler / K. Scott: UML Distilled, New York: Addison Wesley (1997).

/GG 84/ H. Ganzinger / R. Giegerich: Attribute Coupled Grammars, in Proc. ACM Symp. on Compiler Construction, SIGPLAN Notices 17, 6, 172–184 (1984).

/GHS 95/ D. Georgakopoulos / M. Hornick / A. Sheth: An Overview of Workflow Management: From Process Modelling to Workflow Automation Infrastructure, Distributed and Parallel Databases 3, 119–153 (1995).

/Gru 97/ S. Gruner: Schema Correspondences Support the Specification of Incremental Integration Tools based on pair graph grammars (in German), TR AIB–97/5, RWTH Aachen (1997).

/Her 96/ O. Hermanns: Multicast Communication in Cooperative Multimedia Systems (in German), Diss. RWTH Aachen (1996).

/HN 90/ D. Harrison / A. Newton / R. Spickelmeir / T. Barnes: Electronic CAD Frameworks, Proc. IEEE 78–2, 393–419 (1990).

/ISO/ ISO CD 10303: Product Data Representation and Exchange, NIST Gaithersburg, USA.

/Jan 92/ Th. Janning: Requirements Engineering and Programming in the Large (in German), Diss. RWTH Aachen, Wiesbaden: Dt. Univ. Verlag (1992).

/JJ 95/ M. Jeusfeld / U. Johnen: An Executable Metamodel For Re-Engineering of Database Schemas, Int. Journal of Cooperative Information Systems, 4 (2/3), 237–258 (1995).

/JSZ 96/ J. Jahnke / W. Schäfer / A. Zündorf: A Design Environment for Migrating Relational to Object-Oriented Data Base Systems, in: Proc. Int. Conf. on Software Maintenance 1996, IEEE Computer Society Press, 163–170 (1996).

/KKM 87/ G. E. Kaiser / S. M. Kaplan / J. Micallef: Multi-user, Distributed, Language-based Environment, IEEE Software 4, 6, 58–67 (1987).

/Kra 98/ C. A. Krapp: An Adaptable Environment for the Management of Development Processes, Diss. RWTH Aachen, to appear.

/Kro 93/ K. Kronlöf: Method Integration, Chicester: Wiley (1993).

/Lef 95/ M. Lefering: Integrating Tools in a Software Development Environment (in German), Diss. RWTH Aachen, Aachen: Shaker (1995).

/Lew 88/ C. Lewerentz: Interactive Design of Large Program Systems (in German), Diss. RWTH Aachen, IFB 194, Berlin: Springer (1988).

/McI 95/ K.G. McIntosh: Engineering Data Management – A Guide to Successful Implementation, Maidenhead: McGraw-Hill (1995).

/Nag 90/ M. Nagl: Software Engineering: Methodical Programming in the Large (in German), Berlin: Springer (1990).

/Nag 96/ M. Nagl (Ed.): Building Tightly Integrated Software Development Environments: The IPSEN Approach, LNCS 1170, Berlin: Springer (1996).

/NM 97/ M. Nag l /W. Marquardt: SFB 476 IMPROVE: Support for Overlapping Developing Processes in Chemical Process Engineering (in German), in Jarke/Pasedach/Pohl (Eds.): Informatik '97, Informatik aktuell, 143–154, Berlin: Springer (1997).

/NW 98/ M. Nagl / B. Westfechtel: Integration of Development Systems in Engineering Applications (in German), to appear, Berlin: Springer (1998).

/PDJ 94/ K. Pohl / R. Dömges / M. Jarke: Decision-oriented Process Modelling, Proc. 9th Intern. Software Process Workshop, ISPW 9, Airlie, Virginia (1994).

/Pra 71/ T.W. Pratt: Pair Grammars, Graph Languages and String-to-Graph Translations, Journal of Computer and System Sciences 5, 560–595 (1971).

/Re 93/ R. Reddy et al.: Computer Support for Concurrent Engineering, IEEE Computer 26–1, 12–16 (1993).

/Rei 90/ S. Reiss: Interacting with the Field Environment, Software–Practice and Experience 20–S1, 89–115, (1990).

/RS 96/ J. Rekers / A. Schürr: Defining and Parsing Visual Languages with Layered Graph Grammars, Journal of Visual Languages and Computing 7, 3 (1996).

/RT 88/ T. Reps / T. Teitelbaum: The Synthesizer Generator Reference Manual, New York: Springer (1988).

/SB 93/ D. Scheström/G. v. d. Broek (Eds.): Tool Integration, Chicester: Wiley (1993).

/Sch 94/ A. Schürr: Specification of Graph Translators with Triple Graph Grammars, in Tinhofer (Ed.): Proc. WG '94 Int. Workshop on Graph-Theoretic Concepts in Computer Science, LNCS 903, Berlin: Springer Verlag, 151–163 (1994).

/Sch 97/ A. Schürr: Programmed Graph Replacement Systems, in Rozenberg (Ed.): Handbook of Graph Grammars and Computing by Graph Transformation, vol. 1, Singapore: World Scientific, 479–546 (1997).

/SL 90/ A. P. Sheth / J. A. Larson: Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases, Comp. Surveys 22–3, 183–236 (1990).

/SWZ 95/ A. Schürr / A.J. Winter / A. Zündorf: Graph Grammar Engineering with PROGRES, in Schäfer/Botella (Eds.): Proc. 5th ESEC, LNCS 989, 219–234 (1995).

/Wes 91/ B. Westfechtel: Revision and Consistency Control in an Integrated Software Development Environment (in German), Doct. Diss., RWTH Aachen, IFB 280 (1991).