IEEE Computer Society Press, p. 2-13 (1992)

A Graph–Based Approach to the Construction of Tools for the Life Cycle Integration between Software Documents

Bernhard Westfechtel
Lehrstuhl für Informatik III
Technical University of Aachen, W–5100 Aachen

Abstract

In the course of the construction of a software system, software documents (requirements definitions, software architectures, module implementations, etc.) are produced which are written in different languages and describe (parts of) the system from different points of view. While the construction of structure–oriented editors for such documents seems to be well understood, the construction of tools for the integration between them has severely been neglected. In this paper, we present a formal approach to the semi–automatic construction of incremental integration tools which provide both passive and active support for inter–document dependencies (consistency analyses and generation/updating of templates for dependent documents, respectively).

1 Introduction

In the course of the construction of a software system, **software documents** are produced which describe (parts of) the system from different points of view. For example, a requirements definition specifies the requirements which have to be met, a software architecture describes the structure of the system in terms of modules, subsystems and their interconnections, and a technical documentation contains an informal description which assists software developers in understanding and maintaining the system.

Depending on the purpose of the description, various **languages** are used for the notation of software documents. For example, requirements may be specified using Structured Analysis, the software architecture may be described in a module interconnection language, and the technical documentation may consist of a structured text which is composed of chapters, sections, and paragraphs. The languages differ considerably with respect to their presentation (textual or graphical) and with respect to the degree of their formalization (ranging from well understood documents such as programs to highly informal documents such as technical documentations).

In order to construct large, complex software systems, developers have to be supported by powerful and comprehensive **software development environments** [8, 21] covering the whole life cycle. We believe that – among others – the following requirements are essential for the success of software development environments:

1. They have to be **structure–oriented**, i.e. the tools must know the logical structures of the documents they are operating on.
2. They have to be **integrated**, i.e. the tools must understand the interrelationships between different software documents[1].
3. They have to be **adaptable to new languages** (or to modifications of existing languages) with little effort.

A lot of projects have been concerned with the construction of **structure–oriented software development environments** [13, 24, 3, etc.]. These projects have focused on building structure–oriented editors for textual or graphical languages. The effort to build such editors is considerably reduced by the application of generators which are fed with language descriptions. While environments of this kind meet the requirements 1 and 3 stated above, they fall short of providing adequate integration support (requirement 2) because of their limited capabilities for representing and maintaining inter–document relationships.

In particular, **integration across the software life cycle** has severely been neglected. To a great extent, the user is in charge of controlling the interrelationships between documents such as requirements definitions, software architectures, module implementations, and technical documentations. For example, he has to take care that the software architecture conforms to the requirements definition, and that modules are implemented according to their interfaces specified in the software architecture. This leads to serious problems especially when large and complex software systems have to be maintained over a long period: Without adequate automated support, relationships between software documents produced in different working areas (requirements engineering, programming in the large, programming in the small, etc.) eventually get lost, and the

1. There are also other important aspects of integration (e.g. uniform user interface) which are not relevant for this paper.

various descriptions of a software system tend to diverge more and more.

In this paper, we present an approach to the construction of **incremental integration tools** which are embedded in an integrated and structure–oriented environment. These tools assist the user in controlling the interrelationships between software documents that are produced in different working areas and therefore are written in different languages. We will demonstrate that such integration tools may partially be generated from descriptions of language correspondences. Thus, our approach aims at constructing software development environments which meet all requirements stated above.

Some important features of our approach are the following:

- It is based on **hypertext technology** [7]. In our approach, a document management system supporting links serves as a base layer on top of which integration tools are built which perform consistency analyses, generate frames and update dependent documents. These links assist in providing traceability across the software life cycle.
- It takes both the **coarse–grained level** (on which documents are considered as atomic units) and the **fine–grained level** (on which the internal structure of documents is modeled) into account.
- In contrast to most hypertext systems, our approach relies on a well–defined **formal data model**. For representing the database of a software system, we use attributed, hierarchical graphs. The structure of and operations on such graphs are formally defined by means of an operational specification language which is based on graph rewriting systems.

The remainder of this paper is organized as follows: In section 2, we describe the functionality of integration tools from the user's point of view. In section 3, we adopt the tool builder's view and present our approach to the construction of integration tools. In section 4, we compare our approach to other work described in the literature. Finally, section 5 concludes this paper with a short summary and a description of future activities.

2 Functionality of integration tools : the user's view

The research presented in this paper has been carried out within the **IPSEN** project [18, 20] which is dedicated to the construction of structure–oriented software development environments covering the whole software life cycle. It has been one of the major concerns of this project to study the integration between software documents which are produced in different working areas. In the sequel, we first describe the basic principles underlying our approach to integration (subsection 2.1). Subsequently, we illustrate these principles by means of some examples (subsection 2.2). Due to the lack of space, we only describe the basic functionality, but not the user interface of integration tools (see [18]).

2.1 Principles

With respect to their functionality, IPSEN integration tools have the following outstanding features:

- **Active support**: In addition to providing passive support by means of analysis tools which check the consistency between interdependent documents, there is also active support for the transition between different working areas. To a certain extent, information produced in one working area may be used for the automatic construction of software documents in another working area. Hence, the user is supported by tools which automatically create templates for dependent documents. For example, from the software architecture templates are generated which contain procedure headings which have to be filled out by the implementer.
- **Change propagation**: It is well known that activities in the software life cycle are not organized into sequential phases, but are highly interwoven. Therefore, it is not sufficient to build compilative integration tools which merely generate new templates for dependent documents. IPSEN tools operate **incrementally** inasmuch as incremental changes are propagated from master documents to their dependents. Thereby, manual extensions of previously generated templates are preserved.

Our incremental integration tools make intensive use of **links** between interdependent documents. On the **coarse–grained level**, links represent dependencies between whole documents which are considered as atomic units. These links alone are not sufficient for an incremental integration. Therefore, links are also maintained on the **fine–grained level**, where the structure of documents is taken into account. On this level, links represent relationships between increments (i.e. syntactical units such as variable declarations, statements, etc.) that belong to different documents. As will be explained in more detail in section 3, fine–grained links are used to determine the consequences which changes to a master document imply in its dependent documents.

With respect to the use of links, our work is closely related to **hypertext systems** [7] which have originally been developed for writing books, encyclopedia, documentations, etc., but have recently also been applied to other tasks (e.g. CAD and software development). However, there are at least two important differences regarding the handling of links:

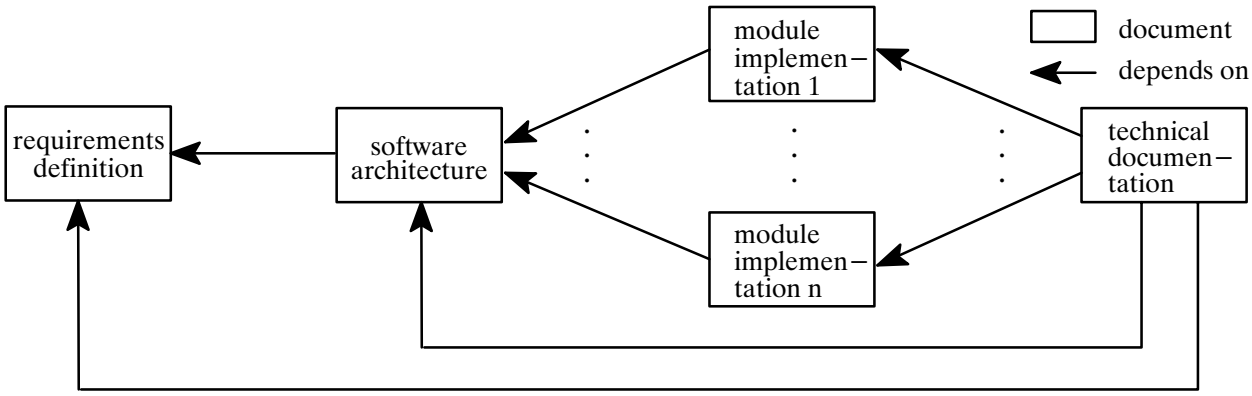

Fig. 1 Software documents and their interdependencies

- In most hypertext systems, links are always created **manually**. On the other hand, in IPSEN links are mostly created **automatically** by integration tools: For example, increments which belong to templates are related to the increments from which they are derived.
- Furthermore, links may usually be created between any portions of a hypertext. By way of contrast, the IPSEN system controls which (parts of) documents may be connected through links. On the coarse–grained level, the underlying life cycle model determines between which types of documents dependency links may be created. On the fine–grained level, links between increments of different documents may only be created if there is a corresponding dependency link on the coarse–grained level and further structural constraints (e.g. regarding the types of increments which may be connected) are satisfied. For these reasons, we denote IPSEN as a **syntax–directed hypertext system**.

An integration tool maintains interrelationships between master documents and dependent documents of certain types. Clearly, its functionality depends on the respective pair of document types. The following factors play a significant role:

- **Degree of formalization**: The less the respective document types are formalized, the less the user may be assisted in controlling external consistency (consistency of a dependent document with respect to its master document)[2].
- **Conceptual distance**: Software documents produced in different working areas describe (parts of) software systems from different points of view. Even if the corresponding document types are well formalized, it may be difficult to support integration between them in case their underlying concepts differ from each other considerably.

2. Internal consistency refers to the correctness of the contents of one document.

2.2 Examples

In the IPSEN project, we have been studying various examples for the integration between software documents. Fig. 1 depicts the **integration scenario** which we have been studying so far. As will be described below, the corresponding integration tools differ according to the factors discussed above (degree of formalization, conceptual distance).

The tightest form of integration has been realized between **software architectures** and **module implementations**. In this case, software documents are highly formalized, and the conceptual distance between source and target language is relatively small. We built an incremental integration tool which automatically generates (and updates) Modula–2 implementation templates from software architecture descriptions which are written in a module interconnection language called IPSEN–MIL [19]. Such templates consist of procedure headings, type declarations, and import clauses; they are filled out by the implementer. Templates are generated and modified on demand; modifications preserve (as far as possible) manual extensions performed by the implementer. In order to support change propagation, fine–grained links between architecture and module increments are created and maintained automatically by the integration tool.

The interrelationships between **technical documentations** and the **documents described therein** are much looser than the interrelationships between software architectures and module implementations. Technical documentations are semi–formal documents which consist of chapters, sections, and paragraphs the latter of which contain plain text. While it is possible to derive the coarse structure of a documentation from the contents of the documents to which it refers, this mechanism alone does not suffice to control the external consistency between the technical documentation and its master documents. Therefore, the user may

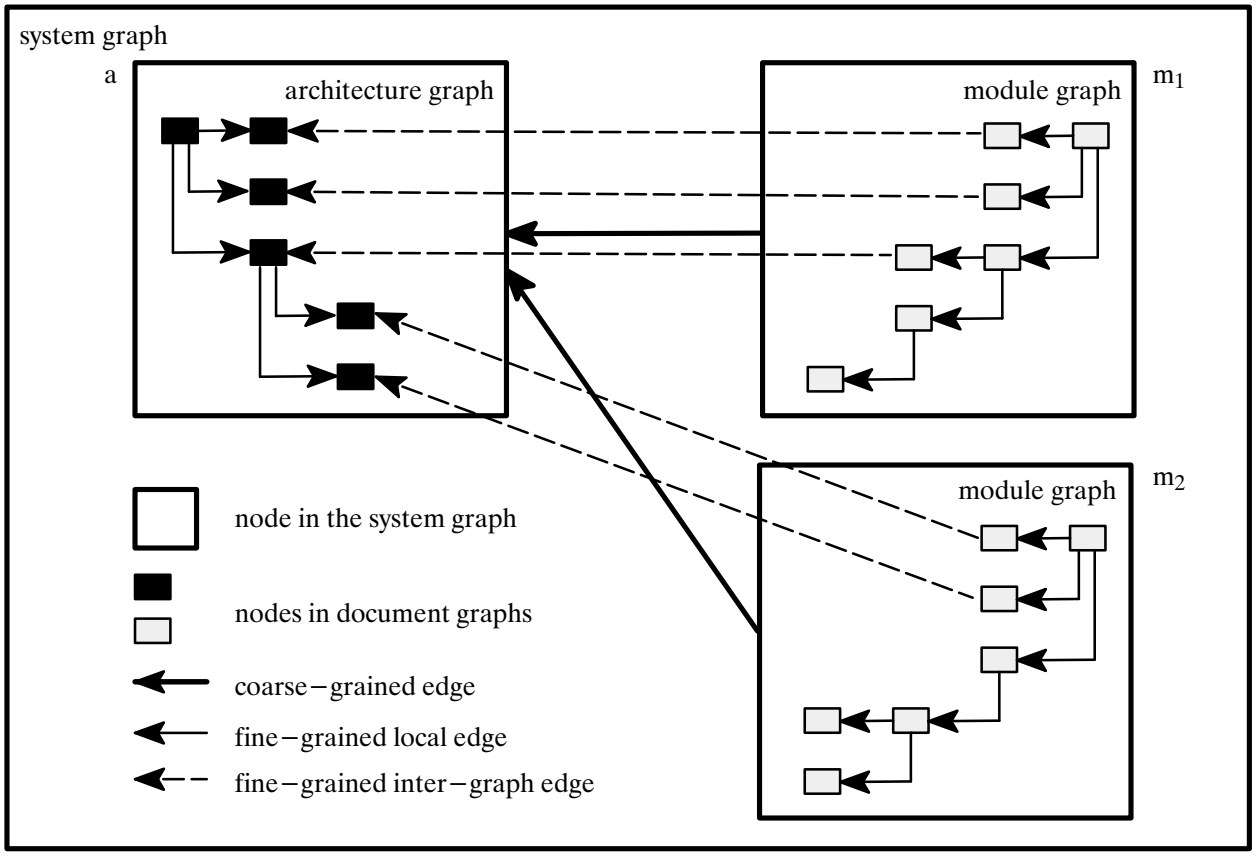

Fig. 2 Graph model of a software system

attach additional links to increments of the documentation which refer to increments of the master documents. By means of these manually created links, inconsistencies in the documentation may be detected after changes to the master documents have been performed. Furthermore, the links may be used for browsing. Thus, manually created links play a similar role in IPSEN as links in traditional hypertext systems. For further information on this topic, the reader is referred to [18].

With respect to its tightness, the integration between **requirements definitions** and **software architectures** lies between the cases discussed above. For the definition of requirements, we have developed a combination of SA and ER. Software architectures are specified in the IPSEN–MIL. Using these languages, requirements definitions and software architectures are formalized to a higher degree than technical documentations. On the other hand, the conceptual distance is greater than between software architectures and module implementations. While the problem space is investigated in the requirements analysis, the design is the first important step into the solution space. Due to the gap between requirements analysis and design, we have built an interactive tool for the construction of architecture templates. This tool is driven by transformation rules. Whenever an increment of the requirements definition may be transformed in multiple ways, the user controls which rule is to be applied. In contrast to implementation templates, architecture templates are not protected against modifications and may be changed by the user in arbitrary ways. A detailed discussion of the integration between requirements definitions and software architectures is beyond the scope of this paper; therefore, the interested reader is referred to [4].

What can be learnt from these examples? The **functionality** of an integration tool heavily depends on the degree of formalization of the respective document types as well as their conceptual distance. The choice space is at least characterized by the following options:

- An integration tool may provide **active** or **passive support** (i.e. generation/updating of frames or consistency analyses).
- An active integration tool may be either **phase–oriented** or **incremental**. In the former case, it may only generate empty frames; in the latter case, it supports the propagation of incremental changes.

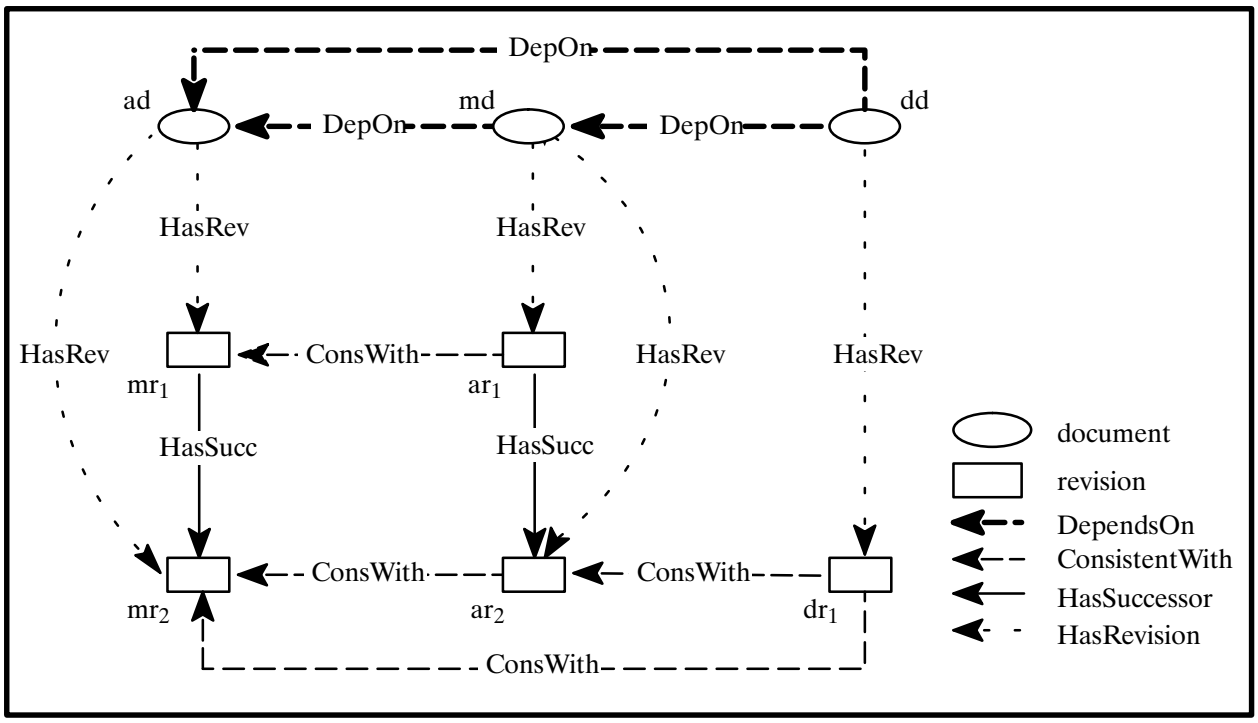

Fig. 3 Example of a System Graph Based on a Model for Revision and Consistency Control

- An active integration tool may operate either **deterministically** or **non–deterministically**. In case of deterministic operation, frames are updated/generated automatically. Otherwise, user interactions are required to resolve ambiguities (i.e. to select among multiple transformation rules).
- Frames may be either **protected** or **modifyable.** In the first case, the user is prevented from changing a frame. In the second case, the frame acts as a proposal which may be adapted by the user (or even thrown away completely).
- An integration tool may create **links automatically**, or it may support the **manual creation** of links in a hypertext–like fashion. The former case applies to active tools; the latter case occurs whenever active support is only possible to a limited extent.

3 Construction of integration tools : the tool builder's view

While the last section described the functionality of integration tools, we now take a look at integration tools from the tool builder's point of view. We present a formal approach to the **construction of integration tools** which is based on graphs. In subsection 3.1, we give a general survey of the formal specification of integration tools within the IPSEN framework. In subsection 3.2, we present an approach to the semi–automatic construction of integration tools which relieves the tool builder from routine tasks which are time–consuming and error–prone when they have to be carried out manually.

3.1 Graph–based formal specification of integration tools

Within the IPSEN project, **hierarchical attributed graphs** are used internally for modeling the structure of documents and the relationships between them. Fig. 2 shows how a (cutout of a) software system is internally represented as a hierarchical graph. On the coarse–grained level, the **system graph** consists of nodes and edges representing documents and their interdependencies. To each node, a **document graph** is associated which represents the internal structure of the corresponding document. Typically, the structure of a document is modeled as an abstract syntax graph, i.e. an abstract syntax tree which is augmented with context–sensitive edges representing bindings of identifiers, control flow, data flow, etc. Fine–grained links between increments that belong to different documents are modeled by non–local edges, which are denoted as **inter–graph edges**.

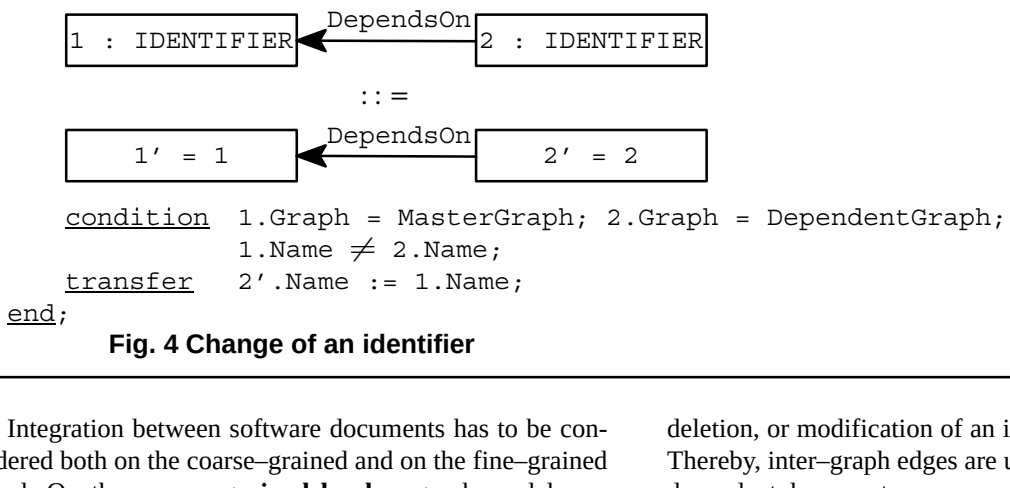

Fig. 4 Change of an identifier

Integration between software documents has to be considered both on the coarse–grained and on the fine–grained level. On the **coarse–grained level**, a graph model was developed which also takes the evolution histories of software documents into account [30, 31]. During its lifetime, a software document evolves into revisions which represent successive states of its development. In addition to **revision control** (which is essential for maintenance), the graph model also deals with **consistency control** which is concerned with the interrelationships between revisions of interdependent documents. In contrast to systems such as SCCS [25] and RCS [29], consistency relationships are maintained which aid in constructing consistent configurations: only those revisions of interdependent documents may be combined which are connected through consistency relations (e.g. architecture revision ar_2, module revision mr_2, and documentation revision dr_1 in fig. 3).

In the following, we focus on integration on the **fine–grained level**. We sketch how an **incremental transformer** is formally specified which propagates changes from a master document to a dependent document. We do not explicitly go into the compilative generation of templates which is included as a special case: the increment which is to be transformed is the root of the (relevant part of the) master document. Furthermore, we do not address the specification of passive tools which perform consistency analyses. However, note that certain analysis operations are also performed within an incremental transformer because it has to localize parts of the dependent document which have to be adapted to the master document.

For the specification of an incremental transformer, we use the language **PROGRESS** [26, 27, 28] which has been developed within the IPSEN project. PROGRESS supports the rule–based specification of graph transformations. A graph transformation is declaratively specified by means of a **graph rewrite rule** (also called production) which describes a pattern and its replacement. An incremental transformer consists of a set of such rules each of which handles a certain change to the master document, namely insertion, deletion, or modification of an increment of a certain type. Thereby, inter–graph edges are used to correctly update the dependent document.

A simple **example** of a **transformation rule** is shown in fig. 4. This rule updates a dependent identifier in the following situations:

- The name attribute of the master identifier has been changed.
- The node for the dependent identifier has been created by another transformation rule which handles a structural change to the master document (insertion of the node for the master identifier).

Note that in the second situation the structural transformation rule does not update the name attribute in order to achieve a clear modularization, i.e. orthogonal distribution of tasks among the transformation rules.

Let us briefly explain how the transformation rule is specified in PROGRESS: The **left–hand side** of the graph rewrite rule describes a pattern to be replaced. Here, the pattern consists of two `IDENTIFIER` nodes connected by a `DependsOn` edge. The **condition part** specifies additional constraints that the pattern has to fulfil: nodes `1` and `2` must belong to the `MasterGraph` and the `DependentGraph`, respectively, and their `Name` attributes must be distinct. The **right–hand side** describes the replacing subgraph which (accidentally) is structurally identical to the pattern. The **transfer part** specifies how the values of node attributes of the right–hand side are computed. Here, the value of the name attribute of node `2′` (which is identical to node `2`) is modified; all other attributes remain unchanged.

Another **example** of a **transformation rule** is shown in fig. 5. In contrast to the last example, this transformation rule is language–dependent. It is part of the incremental transformer between software architectures written in the IPSEN–MIL and module implementations written in Modula–2. Interfaces of modules contained in a software architecture comprise (among other things) specifications of functions which consist of an identifier, a formal parameter part, and a type of the return value. Such function specifica-

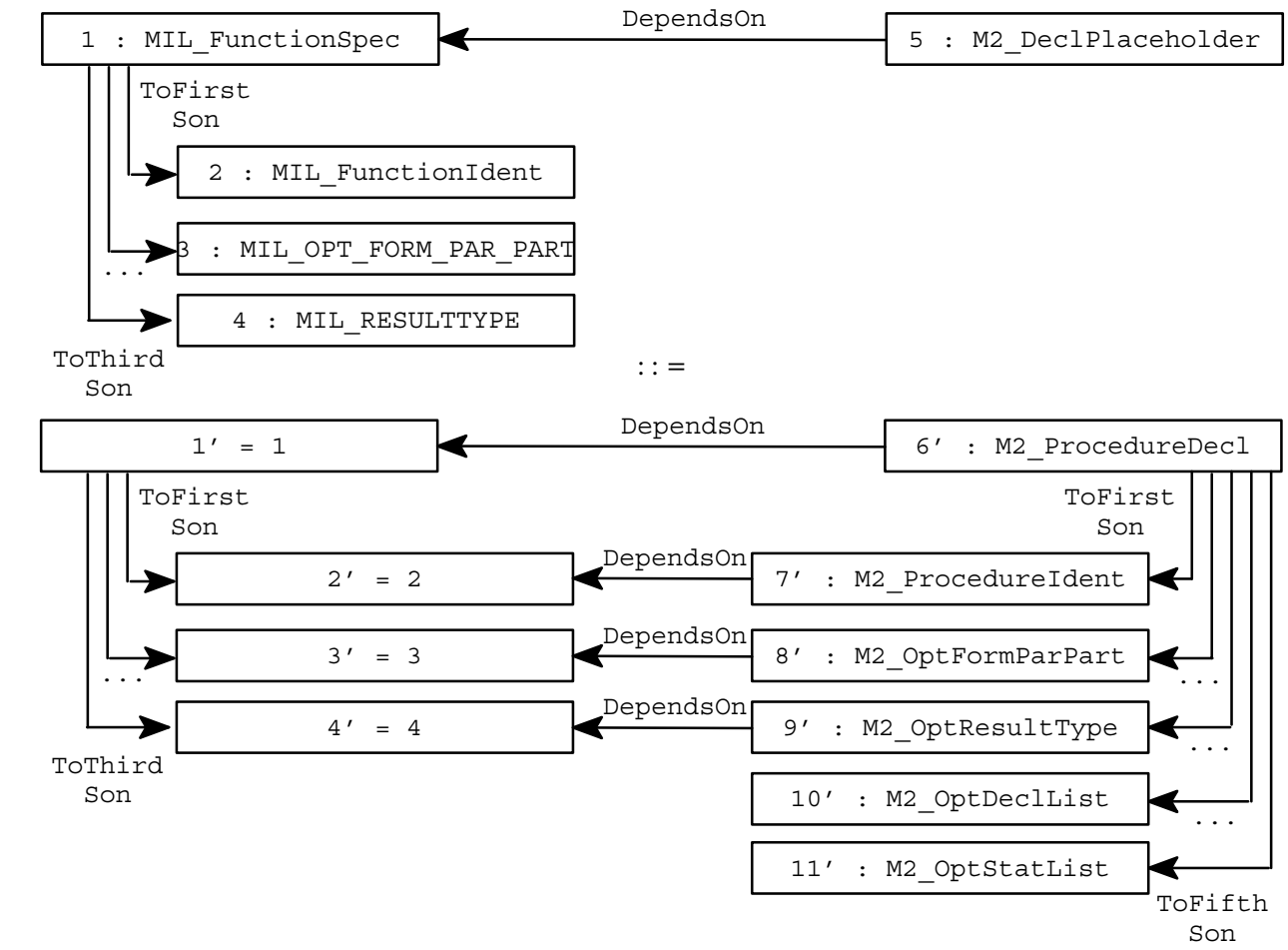

Fig. 5 Transformation of a function specification

tions are mapped onto procedure declarations which are contained in Modula–2 module implementations and consist of an identifier, a formal parameter part, a return type, a declaration list, and a statement list.

The transformation rule searches for a function specification which has not yet been transformed (nodes `1–4` of the left–hand side), but for which a placeholder (node `5`) has already been generated in the module implementation[3]. By application of the rule, the placeholder is replaced with a Modula–2 procedure declaration (nodes `6'–11'` on the right–hand side). The correspondences between (components of) the IPSEN–MIL function specification and the Modula–2 procedure declaration are represented by `DependsOn` edges. The `transfer` part specifies that all new nodes belong to the `DependentGraph`. Furthermore, nodes `6'–9'` are protected against modifications: the `Protected` attribute (which is defaulted to `false`) is used to direct the Modula–2 editor to filter change commands which would destroy the generated template. Finally, the `embedding` part of the production specifies that all incoming (`<-`) and outgoing (`->`) list edges are redirected

3. Note that it is the task of another transformation rule (which handles insertions into lists) to generate such a placeholder.

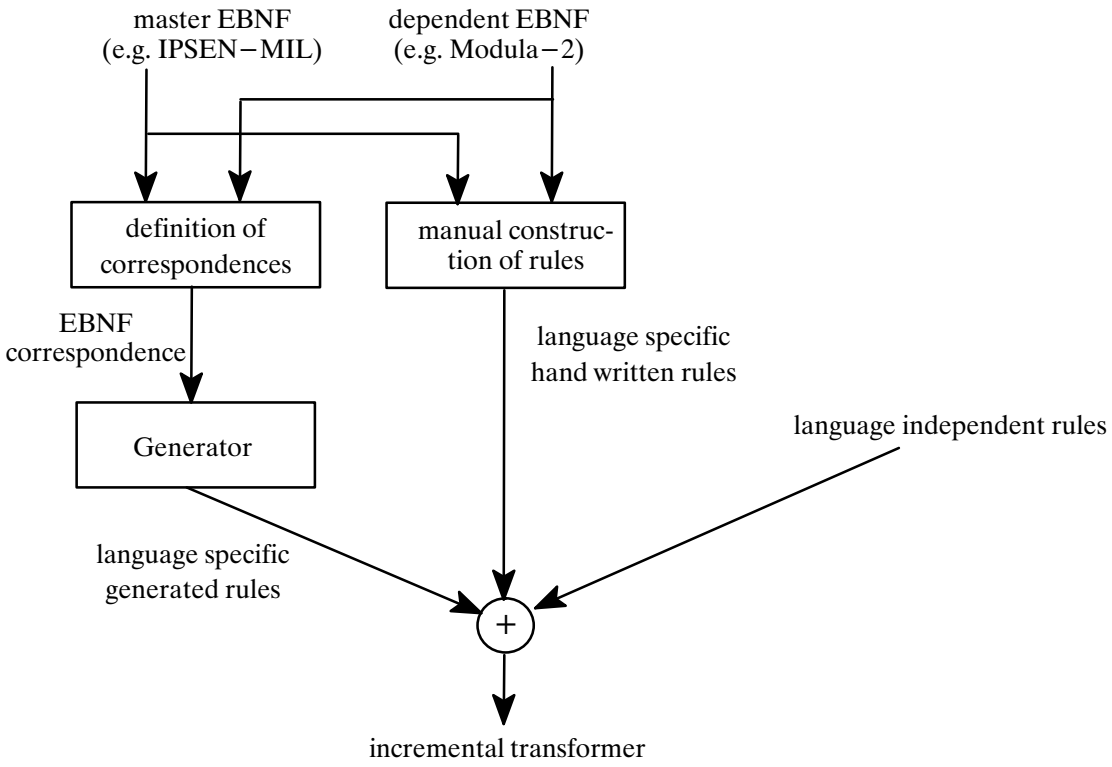

Fig. 6 Semi–automatic construction of an incremental transformer

from node `5` to node `6'`; all other new nodes are not connected to any context nodes.

The reader should note that the examples presented above are taken from a restricted class of transformations – namely **tree–to–tree transformations** – which will be discussed in more detail in the next subsection. However, the power of graph rewrite rules is fully exploited only when dealing with context–sensitive relationships. In particular, context–sensitive relationships play an important role in the transformation of requirements definitions into software architectures which is described in another paper /4/.

3.2 Towards the semi–automatic construction of integration tools

As we already mentioned in subsection 2.1, the actual functionality of an incremental transformer heavily depends on the degree of formalization of and the conceptual distance between source and target language. Therefore, in general it is hard to systematize the construction of an incremental transformer. On the other hand, parts of incremental transformers do exist which are amenable to **automation**. In particular, this applies to tree–to–tree transformations an example of which was presented above. To a varying extent, such transformations are virtually ubiquitous. The generation of module templates is just one example. As another example, consider the generation of a template for a documentation from a software architecture: In our current IPSEN environment, a chapter is generated for each module occurring in the architecture; each chapter consists of sections for the resources which are exported by that module. Therefore, it is worth–wile to consider the automation of such transformations in order to relieve tool builders from some routine aspects of specifying an incremental transformer.

Fig. 6 gives a survey of our approach to the **semi–automatic construction** of an **incremental transformer**. We assume that the languages underlying the master and dependent document are defined by normalized EBNFs. Starting from these descriptions, correspondences between constructs of the master and the dependent language are defined. From these correspondences, language–specific transformation rules may be derived automatically. These rules are combined with language–independent rules which may be reused for each transformer, and with language–specific rules which handle transformations that may not be generated from language correspondences.

Before discussing this approach in a more general way, we present an **example** which demonstrates its benefits. In fig. 7, a correspondence is defined between the productions for function specifications and procedure declarations.

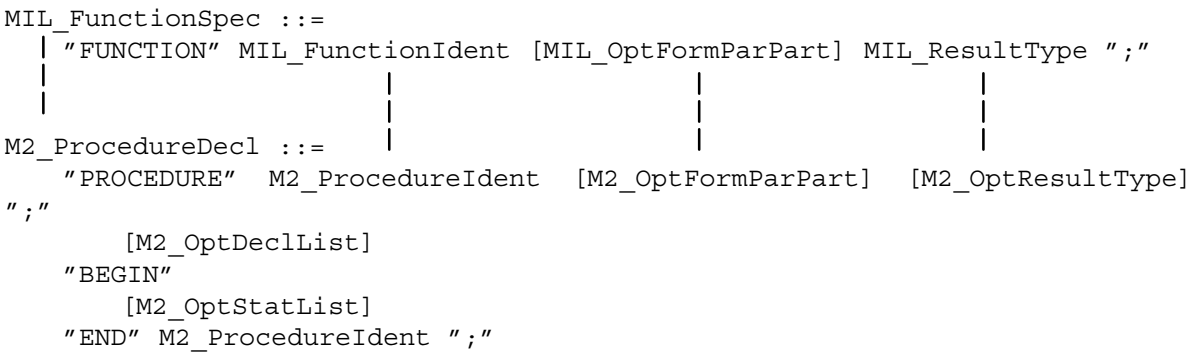

Fig. 7 Example of a production correspondence

Dashed lines denote correspondences between components of the productions. From this production correspondence, the transformation rule of fig. 5 may be generated automatically. This example clearly demonstrates the leverage for the tool builder: It is much easier to establish correspondences between productions than to write the transformation rule of fig. 5.

Constructing an incremental transformer in this way extends our approach to the **semi–automatic construction** of **syntax–aided editors** in a natural way: In IPSEN, the context–free part of a syntax–aided editor (which internally operates on abstract syntax graphs) is generated from a normalized EBNF. Analogously, the context–free part of an incremental transformer is (partially) derived from a description of correspondences between the respective EBNFs.

Let us now describe our approach to the construction of an incremental transformer more closely: The context–free syntax of a textual language is defined by means of a **normalized EBNF grammar** which solely consists of productions of the following types:

- **Alternative productions** which define a selection, e.g.
  ```
  Declaration ::=
      ConstDeclList | VarDeclList |
      TypeDeclList | ProcDecl
  ```
- **List productions** which define list structures, e.g.
  ```
  VarDeclList ::=
      "VAR" {VarDecl} VarDecl
  ```
- **Structure productions** which define record–like structures, e.g.
  ```
  VarDecl ::=
      DeclIdentList ":" TypeDef ";"
  ```
- **Terminal productions** which act as an interface to the lexical syntax for identifiers and literals and (in the context of this paper) are treated as dummy completing productions.

An **EBNF correspondence** between two normalized EBNF grammars M (master) and D (dependent) consists of a set of **production correspondences**

(p_M, p_D, r),

where p_M and p_D are productions from M and D, respectively. A production correspondence establishes a correspondence between the left–hand sides of the productions. Furthermore, components of the right–hand sides of p_M are mapped onto corresponding components of the right–hand side of p_D by means of the partial function r. Correspondences may only be established between symbols of the abstract syntax; keywords and delimiters are ignored.

How production correspondences are interpreted, depends on the features of the integration tool which shall be constructed. So far, the work described here refers to **context–free incremental 1:1 transformations**:

- The incremental transformer only takes context–free relationships into account.
- There is a 1:1 correspondence between generated increments and their masters: each generated increment is associated to one master increment and vice versa.
- The incremental transformer operates deterministically without requiring user interactions. To enable this kind of operation, each production p_M in M may be related to at most one production p_D in D. In particular, this implies the existence of a partial function which maps symbols of M onto symbols of D.
- Essentially, the transformation preserves the topology of the abstract syntax tree; i.e. it does not affect hierarchical relationships between composite and component nodes, and it also does not change the order of list elements.

In practical applications, an incremental transformer rarely performs pure context–free 1:1 transformations. In order to overcome the limitations of this class of transformations, the tool builder has to supply appropriate hand written rules which may specify arbitrarily complex graph transformations (dealing not only with context–free, but – if required – also with context–sensitive relationships).

Since we aim at generating a context free, incremental 1:1 transformer, we have to impose **restrictions** on **production correspondences**. The matrix of fig. 8 shows which combinations of productions are legal. Rows and columns

p_M \ p_D	alternative	list	structure	terminal
alternative	**X**	**X**	**X**	**X**
list	–	**X**	–	–
structure	–	**X**	**X**	–
terminal	–	–	–	**X**

X legal
– illegal

Fig. 8 Legal and Illegal Combinations of Productions

correspond to the master and the dependent EBNF, respectively; crosses denote legal combinations. Each legal combination has a certain meaning with respect to the incremental transformer:

- p_M is an alternative production: If p_D is an alternative production, as well, the correspondence (p_M, p_D) implies that each alternative on the right–hand side of p_M is mapped onto an alternative of the right–hand side of p_D. Otherwise, all alternatives of p_M are mapped onto the same symbol – the left–hand side of p_D.
- p_M is a list production: In this case, we require p_D to be a list production, as well. A correspondence between list productions implies on the instance level that there is an isomorphic mapping between the lists.
- p_M is a structure production. If p_D is a structure production, as well, some of the components of the right–hand side of p_M are mapped onto pair–wise distinct components of the right–hand side of p_D. If p_D is a list production, components are mapped onto list elements.
- p_M is a terminal production. Then, p_D has to be a terminal production, as well. A correspondence between terminal productions implies on the instance level that the dependent lexical string has to match the master string[4].

All other combinations are defined as illegal. For example, a list production must not be combined with a structure production because in general it is not possible to map a homogeneous sequence of variable size onto a heterogeneous sequence of fixed size. Furthermore, a non–alternative production must not be mapped onto an alternative production because this would imply a non–deterministic transformation.

From the production correspondences, language–specific transformation rules may be derived automatically. An example of such a rule was given in fig. 5. Note that a language–specific rule need not be generated if the corresponding transformation is covered by a language–independent rule. For example, no rule has to be generated from a correspondence between terminals because all dependent terminals are kept consistent with their masters by means of the language–independent rule of fig. 4.

4. This restriction could be weakened by allowing the dependent string to be an arbitrary function of the master string (the function would have to be supplied by the tool builder).

4 Relation to other work

In contrast to IPSEN, most structure–oriented software development environments (for textual languages) are based on **abstract syntax trees** rather than on graphs. As already mentioned in the introduction, these environments fall short of providing adequate support for the integration between software documents. Environment generators such as PSG [3] and CPSG [24] produce monolingual program development environments. Systems such as Alma [17], Gandalf [13], Mentor [9] and its successor Centaur [2] support integration between documents written in different languages through gate nodes which carry abstract syntax trees as attributes and thus allow for the transition from one formalism into another. However, it is hard to represent and maintain non–hierarchical links between increments belonging to different documents.

In section 2, we already pointed out some important differences of our approach to general hypertext systems which have been designed to support writing of arbitrary structured texts. Recently, several approaches which apply **hypertext concepts** to **software development environments** have been developed and implemented as prototype systems. Examples of such systems are DIF [6, 12], SODOS [15, 16], and ConceptBase [23]. These systems have in common that they are based on a software life cycle model and support the creation of links between documents belonging to different phases of the software life cycle. Analysis tools check the consistency and completeness of a software system in terms of its structural description. However,

support of integration is confined to the coarse–grained level inasmuch as the interrelationships between the contents of documents are not controlled in any way. In contrast to this, IPSEN provides both active and passive support for fine–grained integration between interdependent documents.

Our approach to the construction of incremental transformers is related to previous work on **source–to–source transformations**. Mostly, source–to–source transformations have been studied in compiler construction [10]. More recently, this topic has also been dealt with in software development environments [11, 22]. While – to the best of our knowledge – all of these approaches are confined to trees, our approach is more general inasmuch as it relies on graph transformations. Another difference is that source–to–source transformers often are batch–oriented inasmuch as they transform complete documents from the source language into the target language. In contrast to this, IPSEN transformers operate incrementally inasmuch as incremental changes to the master document are translated into corresponding changes to the dependent document.

However, **incremental transformation systems** have also been proposed recently [5]. Even these systems are not applicable to the transformation problems which we have been studying: They are only capable of handling dependent documents which are completely created automatically (derived objects). By way of contrast, we deal with templates which have to be filled out (and in some cases may be changed) by the user. Thus, in IPSEN dependent documents simultaneously are source and derived objects.

5 Conclusion

We have presented an approach to supporting the **life cycle integration** between software documents such as requirements definitions, software architectures, module implementations, etc. For formally specifying incremental integration tools, graph rewriting systems are used. We have shown that certain parts of such a graph rewriting system may be generated from a simple description of language correspondences.

Formal specifications were written for most of the integration tools mentioned in subsection 2.2. The approach to the semi–automatic construction of an incremental transformer was developed within the author's dissertation [31] where it was (manually) applied to the integration between software architectures described in the IPSEN–MIL and module implementations written in Modula–2. The experience gained in this experiment was quite promising: only 30 % of the specification consists of hand written, language–specific rules.

All integration tools mentioned above were implemented within the IPSEN software development environment. The **implementations** were hand coded in Modula–2; a generator tool (whose underlying concepts were described in subsection 3.2) is not yet available. The implementations of the integration tools comprise about 12 k loc while the IPSEN environment as a whole consists of about 150 k loc.

Although we have gained some valuable insight into the construction of integration tools, a lot of work still needs to be done. Firstly, our specification language PROGRESS currently does not support hierarchical graphs. So far, the model of a software system as depicted in fig. 2 is not yet completely formally defined. What we need is a concept such as **molecular aggregation** [1].

Molecular aggregation is only one of multiple features which have to be supported by a **module concept** for PROGRESS. So far, a PROGRESS specification is monolithic and may not be divided into modules which are specified separately. When developing a module concept, we have particularly to clarify the notion of data abstraction. For example, a module `I` which contains the specification of an integration tool relies on modules `M` and `D` for the types of the master and dependent documents, respectively. Incremental transformations may be specified in a natural way by means of graph rewrite rules (see fig. 4 and 5) which, however, violate data abstraction. It is still an open question how this conflict may be reconciled.

Acknowledgements

In addition to the author, many other researchers involved in the IPSEN project have significantly contributed to concepts for the life cycle integration between software documents : Manfred Nagl (leader of the IPSEN project), Claus Lewerentz, Andy Schürr, Thorsten Janning, Martin Lefering, ... Furthermore, the author is indebted to Jürgen Börstler, Martin Lefering, Andy Schürr, and Jürgen Theis for helpful and constructive comments on this paper.

References

1. D.S. Batory, A.P. Buchmann : *Molecular Objects, Abstract Data Types, and Data Models : A Framework*, Proceedings of the 10th International Conference on Very Large Data Bases, 172–184 (August 1984)
2. P. Borras et al. : *Centaur : the system*, in [14], 14–24 (November 1988)
3. R. Bahlke, G. Snelting : The PSG System : *From Formal Language Definitions to Interactive Programming Environments*, ACM Transactions on Programming Languages and Systems, vol. 8–4, 547–576 (October 1986)
4. J. Börstler, T. Janning : *Traceability between Requirements and Design*, to appear in : Proceedings COMPSAC '92 (September 1992)

5. A. Carle, L. Pollock : *Modular Specification of Incremental Program Transformation Systems*, Proceedings of the 11th International Conference on Software Engineering, 178–187 (May 1989)
6. S. Choi, W. Scacchi : *Assuring the Correctness of Configured Software Descriptions*, in [32], 66–75 (1989)
7. J. Conklin : *Hypertext : An Introduction and Survey*, IEEE Computer, 17–41 (September 1987)
8. S. Dart, R. Ellison, P. Feiler, N. Habermann : *Software Development Environments*, IEEE Computer, 18–28 (November 1987)
9. V. Donzeau–Gouge, G. Kahn, B. Melese : *Document Structure and Modularity in Mentor*, in: P. Henderson (Ed.): Proceedings of the Symposium on Practical Software Development Environments, SIGPLAN Notices, vol. 19–5, 141–148 (1984)
10. H. Ganzinger, R. Giegerich : *Attribute Coupled Grammars*, Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices, vol. 19, 157–170 (June 1984)
11. D. Garlan, C. Krueger, B. Staudt : *A Structural Approach to the Maintenance of Structure–Oriented Environments*, in : P. Henderson (Ed.): Proceedings of the 2nd Symposium on Practical Software Development Environments, ACM SIGPLAN Notices, vol. 22–1, 160–170 (January 1987)
12. P.K. Garg, W. Scacchi : *A Software Hypertext Environment for Configured Software Descriptions*, in : J. Winkler (Ed.) : Proceedings of the International Workshop on Software Version and Configuration Control, Teubner Verlag, Stuttgart, 326–343 (1988)
13. N. Habermann, D. Notkin : *Gandalf : Software Development Environments*, IEEE Transactions on Software Engineering, vol. 12–2, 1117–1127 (December 1986)
14. P. Henderson (Ed.): *Proceedings of the 3rd Symposium on Practical Software Development Environments*, ACM SIGSOFT Software Engineering Notes, vol. 13–5, (November 1988)
15. E. Horowitz, R. Williamson : *SODOS : A Software Documentation Support Enviroment – Its Definition*, IEEE Transactions on Software Engineering, vol. 12–8, 849–859 (August 1986)
16. E. Horowitz, R. Williamson : *SODOS : A Software Documentation Support Enviroment – Its Use*, IEEE Transactions on Software Engineering, vol. 12–11, 1076–1087 (November 1986)
17. A. van Lamsweerde et al. : Generic Lifecycle Support in the ALMA Environment, IEEE Transactions on Software Engineering, vol. 14–6, 720–741 (June 1988)
18. C. Lewerentz : *Extended Programming in the Large in a Software Development Environment*, in [14], 173–182 (November 1988)
19. M. Nagl : *Software Engineering : Methodical Programming in the Large* (in German), Springer Verlag, Berlin/Heidelberg/New York (1990)
20. M. Nagl : *Characterization of the IPSEN Project*, in : H. Weber (Ed.) : Proceedings of the International Conference on Systems Development Environments & Factories, Pittman, London (1989)
21. R. J. Norman, M. Chen (Ed.) : *Special Issue on Integrated CASE*, IEEE Software (March 1992)
22. K. Normark : *Transformations and Abstract Presentations in a Language Development Environment*, Dissertation, University of Aarhus, Denmark (February 1987)
23. T. Rose, M. Jarke : *A Decision–Based Configuration Process Model*, Proceedings of the 12th International Conference on Software Engineering, 316–325 (March 1990)
24. T. Reps, T. Teitelbaum : *The Synthesizer Generator*, Springer Verlag, Berlin/Heidelberg/NewYork (1988)
25. M.J. Rochkind : *The Source Code Control System*, IEEE Transactions on Software Engineering, vol. 1–4, 364–370 (December 1975)
26. A. Schürr : *Introduction to PROGRESS, an Attribute Graph Grammar Based Specification Language*, in : M. Nagl (Ed.) : Proceedings of the WG '89 Workshop on Graph–Theoretic Concepts in Computer Science, LNCS 411, 151–165 (1989)
27. A. Schürr : *Operational Specification with Programmed Graph Rewriting Systems : Formal Definitions, Applications, and Tools* (in German), Deutscher Universitäts Verlag, Wiesbaden (1991)
28. A. Schürr, A. Zündorf : *Non–Deterministic Control Structures for Graph Rewriting Systems*, in : G.Schmidt, R. Berghammer (Ed.) : Proceedings of the WG '91 Workshop on Graph–Theoretic Concepts in Computer Science, LNCS 570, 48–62 (1991)
29. W.F. Tichy : *RCS – A System for Version Control*, Software : Practice and Experience, vol. 15–7, 637–654 (July 1985)
30. B. Westfechtel : *Revision Control in an Integrated Software Development Environment*, in [32], 96–105 (1989)
31. B. Westfechtel : *Revision and Consistency Control in an Integrated Software Development Environment* (in German), Informatik Fachberichte 280, Springer Verlag, Berlin/Heidelberg/NewYork (1991)
32. J. Winkler (Ed.) : *Proceedings of the 2nd International Workshop on Software Configuration Management*, ACM Software Engineering Notes, vol. 14–7 (November 1989)