

Reengineering a Medical Imaging System Using Graph Transformations

Tobias Röttschke

Technische Universität Darmstadt,
Institut für Datentechnik, Fachgebiet Echtzeitsysteme,
Merckstraße 25,
64283 Darmstadt, Germany
tobias.roetschke@es.tu-darmstadt.de
<http://www.es.tu-darmstadt.de> *

Abstract. This paper describes an evolutionary approach to re-engineer a large medical imaging system using graph-transformations. The solution has been integrated into the ongoing development process within the organizational and cultural constraints of a productive industrial setting. We use graph transformations to model the original and the desired architecture, the mapping between architectural, implementation and design concepts, as well as related consistency rules. Violations are calculated every night and provided as continuous feedback to software architects and developers, so they can modify the system manually according to established procedures. With only a very limited global migration step and moderate changes and extensions to the existing procedures it was possible to improve the software architecture of the system, while new features still could be implemented and released in due time. Although this solution is dedicated to a concrete case, it is a starting point for a more generic approach.

1 Introduction

In its early days, about 25 years ago, the medical imaging technique realized by our system could only provide rather crude three-dimensional images of motionless human tissue. Being less invasive, but more expensive than other imaging techniques, much effort has been spent improving image quality and performance during the years, and nowadays the technique is on the verge of providing highly detailed real-time images of a beating heart under surgery conditions.

Accordingly, the number of possible applications and hence the number of features has increased dramatically. As of today, the system has about 3.5 MLOC, growing at a rate of several 100 KLOC each year. The trust in the stability of the system depends mainly on the organizational and cultural framework established around the development process. However, various technological and economical reasons required a major reorganization of the software, while at the same time,

* This research was performed at Philips Medical Systems in Best, the Netherlands when the author was employee of Philips Research in Eindhoven, the Netherlands.

more and more features had to be added to compete in the market. Among other things like porting the system to another hardware platform with different operating system, introducing a more object-oriented programming language, and integrating more third-party components, the reorganization included the introduction of different architectural concepts. To deal with the increasing complexity in an appropriate way, an overall refactoring of the system according to these concepts has been performed.

When the author became involved, the migration process had already begun. The new¹ architectural concepts called “Building Blocks” had been described in a document written in natural language. Only remotely related to the “Building Blocks” defined in [1], these concepts describe hierarchical units of responsibility, corresponding to the functional organization of the department, the scope of product related specifications, and an encapsulated piece of software. There were ideas about the visibility rules, but there was no formal specification, and the details were still subject to change.

The task of the author was to specify the new concepts using the site-specific terminology in a way that could be understood easily by software architects, designers and developers, and modified at least by software architects. Next to the specification, tool support was required to allow the software architects to monitor the progress of building UML models as part of new design specifications and of refactoring the source code according to the new concepts. Automated code transformations had to be avoided because they would affect virtually all source files at once, circumventing the established quality assurance mechanisms. This was considered a serious risk for the continuity of the ongoing development process.

In this paper, we focus on the evolutionary migration from the existing source code according to an implicit modul concept to the desired situation where the implementation is consistent with newly written design models with explicit architectural concepts. In section 2, we provide an a-posteriori-specification of the original module concepts. Next, we define the desired concepts (section 3) and specify adequate consistency rules (section 4). The separate steps to achieve the desired situation are identified in section 5 and a migration plan is set up. Finally, section 6 describes, how the migration is realized in a way that allows optimal integration into the ongoing development process.

2 Initial Concepts

When analyzing the initial situation and specifying the desired one, we use UML class diagrams [2] such as figure 1 to model the basic concepts and we use story diagrams [3, 4] to define derived concepts and consistency rules. Story diagrams combine graph rewriting rules with activity diagrams, in the sense that activities are specified as graph transformations. Using this visualization rather than a

¹ In this context we use the term “new” as opposed to the old concepts of the system, not at all “new to the world”.

textual notation helps us to discuss and understand the different concepts and identify the relevant issues for the introduction of the new concepts.

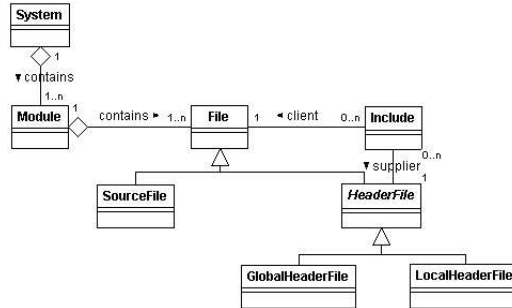


Fig. 1. Initial module concepts

The original architecture is a flat decomposition of the system into *modules*. Each module realizes an isolated piece of functionality. All modules are treated equally and may use functionality implemented by other modules as needed. In the source code, a module is represented by a directory. Modules and directories are mapped on each other using naming conventions according to a coding standard that has been defined as part of the development process.

To organize intra- and inter-module connections, local and global header files are identified by naming conventions. Local header files are only visible in the module they are defined in. To allow the export of functions and data types to other modules, a global scope is defined, which consists of the global header files defined in all modules as a whole. Every module can gain access to this global scope, but by using different include paths for each module, this scope can be restricted to avoid functionally unrelated modules from using each other. All identifiers declared in global header files must be unique to avoid naming conflicts.

To check consistency, we use story diagrams to specify graph queries consisting of several tests. Tests are realized as UML-like activities that do not modify the graph. Figure 2 shows a story diagram consisting of five activities, that defines valid include dependencies between files:

- A start activity, where the execution of the query starts.
- A test, if the *File* and the included *HeaderFile* belong to the same *Module* (valid local include).
- A test, if the included *HeaderFile* is a *GlobalHeaderFile* (valid global include).
- A stop activity returning *false* if the query fails.
- A stop activity returning *true* if the query succeeds.

To interpret the graph patterns of the two main activities, the tool starts with the *this* node, which is bound to the *Include* directive being checked. Next, the

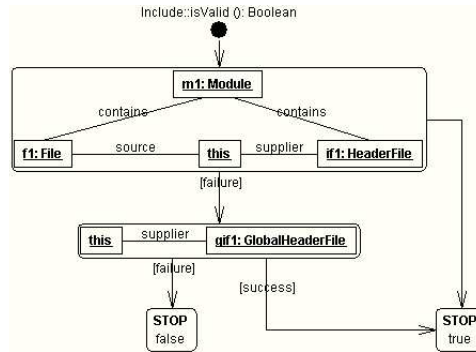


Fig. 2. Initial scoping rule

tool navigates via the associations to adjacent nodes, until it finds a match for the pattern and the activity succeeds. If the pattern cannot be matched, the activity fails. Depending on the result, other activities are executed, until one of the two stop nodes is reached.

Over the years, developing the system using these original module concepts has become increasingly difficult, as functionality has grown tremendously, resulting in an increasing number of modules and interconnections. As a consequence, the system is harder to maintain, and individuals are no longer able to keep an overview over the system as a whole. So it takes more effort to add new features to the system without hampering the work of other developers.

Having more than two hundred software developers working together on a product is virtually impossible, if there is no way to distribute them over smaller teams where they can operate (almost) independently from each other. For example, one team could be concerned with the adoption of a new version of the operating system, without hampering the activities of other teams.

3 Desired situation

The driving idea behind our new module concepts has been *information hiding* [5] and reducing dependencies. Every module should only see resources (data types and functions) that it actually needs. The import scope of every module should be defined explicitly. Available resources should be concentrated on few and unique locations to reduce ambiguity as well as dependencies between modules.

In the desired situation, the architecture consists of an aggregation hierarchy of so-called (building) blocks. Building blocks are architectural units like modules, but they also define an area of responsibility and ownership and a scope for appropriate documentation.

To be able to reason about these building blocks, we have defined additional architectural concepts. We make an explicit distinction between *design concepts* representing the desired system described in design specifications with related

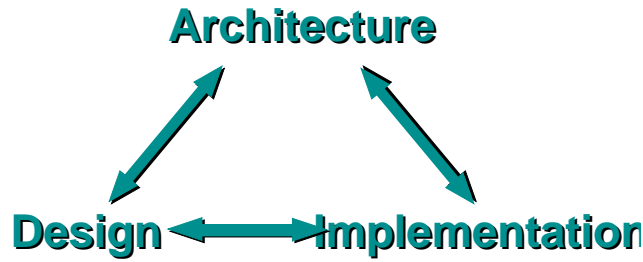


Fig. 3. Consistency between documents

UML models and *implementation concepts* which represent different aspects of the implemented system manifested in the source code. We have defined a *mapping* between these concepts and appropriate *scoping rules* in order to check the consistency between them. The following sections describe the different aspects of our new module concept.

3.1 Design concepts

The basic design concept is a *Block*, as depicted in figure 4. Blocks may *contain* other blocks. All blocks are treated in a way that the same rules can be applied on different system levels and blocks can be moved easily. The system is represented by a root block, which transitively contains all other blocks.

A Block *defines* an arbitrary number of *Interfaces* to export resources to other blocks. The *sees* relationship defines, whether an *Interface* is in the scope of a given *Block*. This relationship is derived from the position of related blocks and interfaces in the hierarchy. Whether a *Block* uses a seen *Interface* correctly, is determined by explicitly defined *Dependencies* between them:

- The *MayUse* dependency indicates that a block may actually import a given interface and is defined explicitly by a software engineer in the UML model. A valid *MayUse* dependency requires an adequate *sees* relationship.
- The *Provides* dependency indicates that a block exports a given interface. The block does not necessarily need to *define* this interface, but may forward an interface from another building block, typically a parent block. A block *defining* an interface does automatically *provide* it as well. Moreover, a block *may use* the interfaces it *provides*.
- The *Implements* dependency indicates that a block implements the resources declared in the interface. The block does not necessarily need to *define* this interface. A block *implementing* an interface does automatically *provide* it as well.

The *sees* relation is composed of two other relations in our model: $sees = sees_{child} \cup sees_{sibling}$. The first relation defines that a block sees all interfaces of

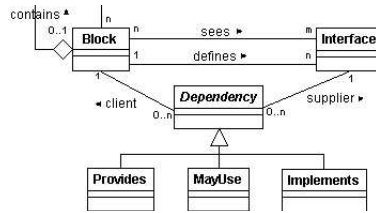
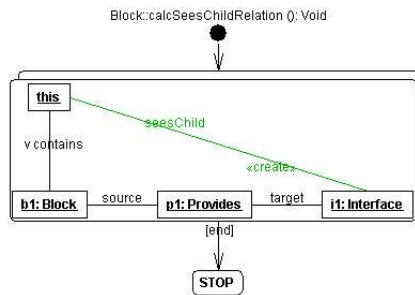


Fig. 4. Design Concepts

its children (figure 5). The second relation defines that a given block B sees all interfaces provided by blocks that are siblings of B or siblings of any of its *ancestors* (figure 6. As opposed to graph queries as in figure 2, these rules modify the model, thus adding derived information to the intrinsic information extracted from the source code and the UML model.

Fig. 5. The *seesChild* relation

The first impression might be, that the definition of the *sees* relation is rather complex, but from all alternatives we have considered, this definition has turned out to match our needs best:

- Maximize information hiding to achieve independent development teams.
- Treat all blocks equally, independent of their position in the aggregation hierarchy.
- Assign correct responsibilities within the organization: the owner of the interface is per definition owner of the top-most block providing it.
- Allow easy relocation of blocks within the hierarchy.

3.2 Implementation concepts

The three major implementation concepts analyzed by our approach are directories, files, and include directives. Several subtypes are distinguished in the class diagram of the model as presented in figure 7.

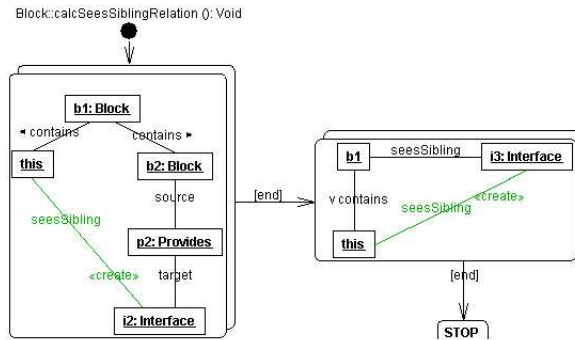


Fig. 6. The *sees_sibling* relation

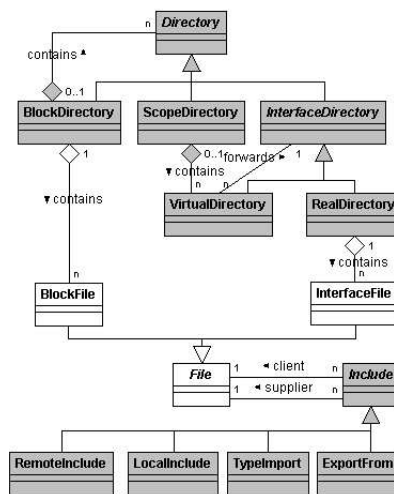


Fig. 7. Implementation concepts

Directories in the source code can either play the role of a *BlockDirectory*, *InterfaceDirectory*, or *ScopeDirectory*. A *BlockDirectory* represents a *Block* and can contain other *BlockDirectories*. Moreover, *BlockDirectories* can contain *InterfaceDirectories* representing provided *Interfaces*. Such an *InterfaceDirectory* can either be a physical sub-directory (modelled as *RealDirectory*) or a link to another *InterfaceDirectory* (modelled as *VirtualDirectories*). Besides, a *BlockDirectory* contains a *ScopeDirectory* with links to *InterfaceDirectories* representing *Interfaces*, which are in the scope of the current *Block*.

Files can be either *BlockFiles* or *InterfaceFiles* depending on whether their parent directory is a *BlockDirectory* or *InterfaceDirectory*, respectively. Include directives can realize either a *RemoteInclude*, *LocalInclude*, *TypeImport* or *ExportFrom* relationship, depending on the class of the source and the target file according to table 1:

	BlockFile	InterfaceFile
BlockFile	LocalInclude	RemoteInclude
InterfaceFile	ExportFrom	TypeImport

Table 1. Include relationships

Different scoping rules apply for these relationships, as explained in section 4.2. But before we can reason about consistency, it is necessary to define the *mapping* between design and implementation concepts.

4 Consistency of design and implementation

4.1 Mapping design onto implementation

The design concepts describe an intended situation that we aim for, whereas the source code contains the implementation of the actual running system. Initially, there will be a huge gap between both kinds of concept. In section 6.4 we show how to monitor the progress of making this gap smaller.

The UML diagram in figure 8 defines our mapping of design onto implementation concepts. Design concepts have a one-to-one relationship with a directory in the source code. Files are owned by a unique block or interface.

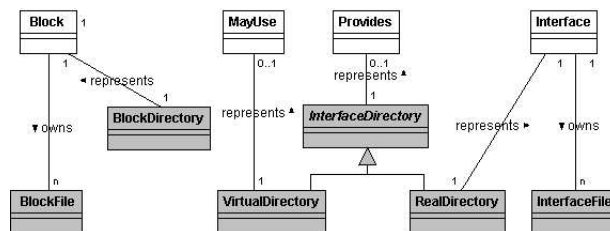


Fig. 8. Mapping the concepts

4.2 Scoping rules

Having defined the mapping between design and implementation concepts, we can check the consistency of our model. This is done by means of 17 so-called scoping rules. Four rules validate the UML model representing the design of the system. Thirteen rules verify the consistency of the implementation with respect to the design.

Figure 9 shows a graph query that defines a rule to verify an include directive in a *BlockFile* referring to an *InterfaceFile*. In section 3.2 we have defined

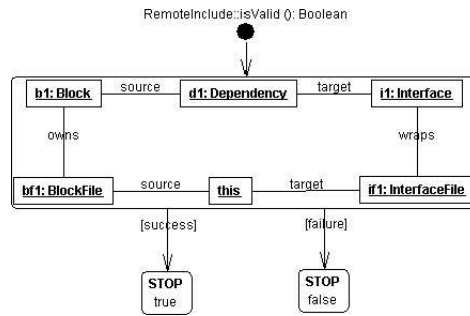


Fig. 9. A scoping rule

this kind of include relationship as *RemoteInclude*. The analyzed instance of *RemoteInclude*, requires a *Dependency* (either *Provides*, *MayUse* or *Implements*) between the *Block* and the *Interface* owning the participating files.

For every object in our model, several of these scoping rules are defined. To check the complete model, we traverse the model in preorder and check the rules appropriate for each visited object.

Having defined our information model and scoping rules to check its correctness, the next step is to apply this model to manage the conceptual reengineering of the system over an extended period of time. The following section will explain our approach.

5 Migration plan for the new software architecture

When we started our reengineering effort, the new building block hierarchy had already been set up. The original modules had been integrated as leaves of the building block hierarchy. By generating makefiles with suitable include paths, the original include directives could be preserved. Thus, no files had to be changed.

The major challenge was to introduce the new visibility concept based on explicitly defined interfaces, without disturbing the ongoing development process. Although the project members had agreed on the general visibility rules, the interfaces and concrete visibility relationships still had to be specified. To implement this specification, hundreds of files had to be moved, thousands of include dependencies had to be relocated and tens of thousands of function calls had to be renamed to reflect the modification with respect to the naming conventions.

Using the normal development procedures including planning, documenting and inspecting each step would be far too complex and time consuming. A fully automated transformation of the complete code base, circumventing the trusted procedures was considered too risky. Besides, specifying all rules in advance, before actually implementing them would provide very late feedback on the effect of the new rules.

As a compromise, we decided to reengineer the top-level building blocks (called subsystems) first and to perform all necessary code transformation. This

should be done considering the actual include dependencies as given and improving them later. Once the subsystems work according to the new architectural concepts, we have enough feedback to reengineer the rest of the system. Because the interfaces of each subsystem hide the internals from the rest of the system, further reengineering can be done locally within single subsystem without effecting the work on other parts of the system.

The following steps have been identified for the first phase of the migration as described above:

1. Identify all header files hit by incoming include dependencies across subsystem borders.
2. Identify all files having outgoing include dependencies across subsystem borders.
3. Define the interfaces of each subsystem.
4. Distribute the header files found in step 1 over the interfaces defined in step 3.
5. Determine the changes to include directives of files in step 2 according to the new distribution of headerfiles (step 4).
6. Define a scope directory (section 6.3 for each building block which serves as include path when building a file of that particular building block.
7. Create the scope directories defined in step 6.
8. Move all global header files to their new location according to step 4 and create a link at the old location so that includes inside the subsystems still work. This includes checking out and in affected directories in the ClearCase archive.
9. Change all files affected according to step 5. This includes checking out and in affected files in the ClearCase archive.

6 Migration tools support

While step 3 and step 4 must be performed manually, all other steps can be automated. Graph rewriting rules defined in FUJABA can help us to perform step 1, step 2, step 5, and step 6 and finally generated Perl scripts to perform the remaining steps.

6.1 Finding includes across subsystem borders

Using our graph model of the system, we could analyze the existing source code and calculate the impact on the system, assuming that we only redirect include dependencies across subsystem borders. We found out, that in total 14160 of the 44624 include dependencies would be modified. In 2386 of the 8186 files, include directives needed to be changed. 521 header files were actually hit and thus had to be distributed manually over the interfaces defined in step 3.

6.2 Calculating include directives

We used the convention `#include "<MODULE>/inc/<HEADERFILE>"` to include headerfiles from other modules in the original architecture. Considering the interfaces of the new architecture, the include directives look like `#include "<BLOCK>/<INTERFACE>/<HEADERFILE>"`. Using the graph model of the system, we can calculate the necessary changes and generate a script to checkout each affected file, modify the include directives and check them in afterwards.

This step circumvents the regular development procedures, but it would be virtually impossible to perform these modifications by hand in a consistent way. However, this transformation is well understood and changes only a single aspect of the system many times. So the risk was considered minimal.

6.3 Calculating scope directories

To implement the visibility rules of the interfaces, we use different so called scope directories for each building block. For each building block, the include path points to a dedicated directory. This directory contains a subdirectory for each building block that may be used. This directory is called "scope directory" and contains links (virtual directories) to directories representing available interfaces. This structure can be modelled by the graph rewriting rule in figure 10. From this model, we generate a script to perform the transformations in the file system.

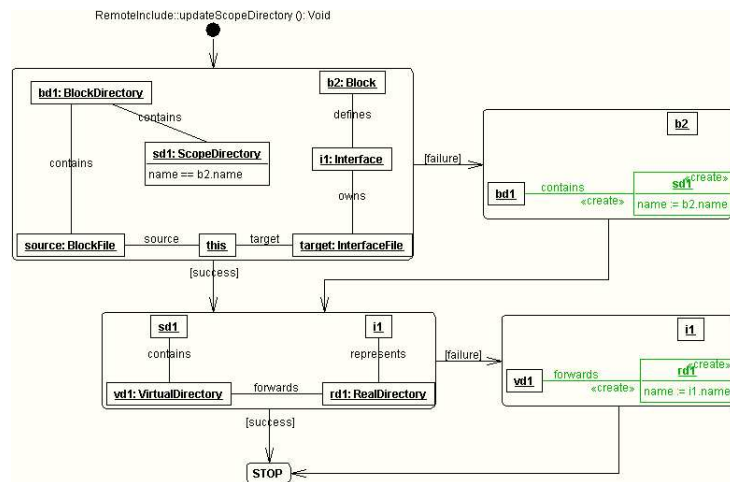


Fig. 10. Updating the scope directory

We tested all the generated scripts on a clone of the repository until it worked flawlessly. Before applying the script to the productive repository, we asked all

developers to check in their modifications before a weekend. During the weekend, we performed the transformation. When the next working week began, everybody started with a consistent version of the new architectural concepts.

6.4 Monitoring reengineering progress

In [6], we discuss trend analysis as a means to provide continuous feedback on the architectural impact of changes to the code. In [7], we outline our overall approach to migrate the existing code base to the new architectural concepts and make it consistent with the new design models.

As described in [7],

7 Outlook and Related work

When considering related work, different aspects can be addressed. First of all, our way of describing architectural concepts can be compared with classic Architecture Description Languages. Next, we compare our work with other tools performing reengineering using graph-transformations. Besides, there are alternative techniques to deal with reengineering. Finally, though our approach is dedicated to a single system, it could be generalized to deal with other systems as well. So we compare our ideas for a more generic approach to evolve large software architectures with existing ones.

7.1 Architectural Concepts

[8]

7.2 Reengineering with Graph Transformations

In [9], a reengineering approach is described, which uses graph transformations to migrate existing COBOL applications into distributed environments. Transformations are formally defined as triple graph grammars and an automated transformation tool is built using the PROGRES [10] environment. This work served as inspiration for our own approach, but because of the constraints of the existing development process model, we choose for a semi-automated tool support. Our tools automate the analysis of the system, but let developers manipulate the system according to the usual development procedures. [11]

7.3 Alternative Reengineering Techniques

7.4 Generic Reengineering Approaches

Columbus [12] is a reengineering tool for C++ based on the GXL [13] reference schema for C++. This is a more generalized approach for analyzing C++ programs than we use in our approach, although it was not yet available when

we started our research. Building our tool based on Columbus would have allowed us to perform more detailed dependency analyses rather than just include dependencies. On the other hand Columbus does not deal explicitly with file systems artifacts, which represent most of our architectural concepts. Nevertheless, analysing only include dependencies was good enough for us to get the job done with the resources available.

8 Conclusions

This paper describes, how graph transformations have been used to support the architectural reorganization of a large medical imaging system. UML-like Graph schemata have been used to separate architectural, design and implementation concepts, which the architectural in an industrial case to define architectural concepts and to specify consistency rules of related design and implementation artefacts. We described our experiences, how a set of architecture analysis tools based on graph rewriting, database and web technology can be successfully applied to support evolutionary architectural improvement of a large medical imaging system. As we have no complete and formal architectural specification at our disposal, we cannot apply formal techniques to analyze the impact of changes a priori (e.g. as proposed by [14]). We rather have to perform our analyses based on the continuously changing source code of the running system and incomplete or possibly outdated design specifications. Using the available information, we keep an up-to-date overview of the system size, structure and consistency on architectural level.

Driven by safety and other quality requirements, we automatically transform the source code only to the extent necessary and perform the remaining reengineering steps manually according to approved development procedures. To support this process, we measure and visualize the progress of the migration using data updated daily.

The idea of applying graph grammars for software engineering tools has already been described by [10] and [15], but we have transferred this idea successfully to a large scale industrial case. Using graph rewriting rules helped us to discuss rather complex visibility and transformations rules with developers and software architects. The ability to generate code from the specifications helped us to try out different variations and keep the tools consistent with the rules we had discussed.

References

1. Linden, F.J.v.d., Müller, J.K.: Creating Architectures with Building Blocks. *IEEE Software* (1995) 51–60
2. Rumbaugh, J., Jacobson, I., Booch, G.: *The Unified Modelling Language Reference Manual*. Addison-Wesley (1999)

3. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A new Graph Grammar Language based on the Unified Modelling Language and Java. In: Workshop on Theory and Application of Graph Transformation (TAGT'98), University-GH Paderborn (1998)
4. Köhler, H., Nickel, U., Niere, J., Zündorf, A.: Using UML as a Visual Programming Language. Technical Report tr-ri-99-205, University of Paderborn, Paderborn, Germany (1999)
5. Parnas, D.L.: A Technique for Software Module Specifications with Examples. *Communications of the ACM* **15** (1972) 330–336
6. Röttschke, T., Krikhaar, R., Havenith, D.: Multi-View Architecture Trend Analysis for Medical Imaging. In: IEEE International Conference on Software Maintenance (ICSM). (2001) 107
7. Röttschke, T., Krikhaar, R.: Architecture Analysis Tools to Support Evolution of Large Industrial Systems. In: IEEE International Conference on Software Maintenance (ICSM). (2002) 182–193
8. Medvidovic, N., Taylor, R.N.: A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering* **1** (2000) 70–93
9. Cremer, K.: Graph-Based Reverse Engineering and Reengineering Tools. In: AGTIVE, Springer (1999) 95–109
10. Schürr, A., Winter, A.J., Zündorf, A.: Developing Tools with the PROGRES Environment. In Nagl, M., ed.: Building Tightly Integrated Software Development Environments: The IPSEN Approach. Volume 1170 of LNCS., Springer (1996) 356–369
11. Mens, T.: Conditional Graph Rewriting as a Domain-Independent Formalism for Software Evolution. In: AGTIVE, Springer (1999) 127–143
12. Ferenc, R., Beszédes, Á., Tarkiainen, M., Gyimóthy, T.: Columbus - Reverse Engineering Tool and Schema for C++. In: IEEE International Conference on Software Maintenance (ICSM). (2002) 172–181
13. Holt, R., Winter, A., Schürr, A.: GXL: Towards a Standard Exchange Format. In: Proc. Working Conference on Reverse Engineering. (2000) 162–171
14. Zhao, J., Yang, H., Xiang, L., Xu, B.: Change Impact Analysis to Support Architectural Evolution. *Software Maintenance and Evolution: Research and Practice* **14** (2002) 317–333
15. Kullbach, B., Winter, A.: Querying as an Enabling Technology in Software Reengineering. In: IEEE Conference on Software Maintenance and Reengineering. (1999) 42–50
16. Jahnke, J., Wadsack, J.: Varlet: Human-Centered Tool Support for Database Reengineering. In: Proceedings Workshop Software Reengineering (WSR'99). (1999)
17. Jahnke, J., Wadsack, J.: Integration of Analysis and Redesign Activities in Information System Reengineering. In: Proc. of the 3rd European Conference on Software Maintenance and Reengineering (CSMR'99), Amsterdam, NL, IEEE Press (1999)
18. Rice, M., Seidman, S.: A Formal Model for Module Interconnection Languages. *IEEE Transactions on Software Engineering* **20** (1994) 88–101
19. Dean, T.R., Lamb, D.A.: A Theory Model Core for Module Interconnection Languages. Technical Report ISSN-0836-0235-94-370, Queen's University, Kingston, Canada (1994)

20. Rumpe, B., Schoenmakers, M., Rademacher, A., Schürr, A.: UML + ROOM as a Standard ADL? In: Proc. 5th International Conference on Engineering of Complex Computer Systems (ICECCS'99), Las Vegas, Nevada (1999) 43–53
21. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In Mayr, E.W., Schmidt, G., Tinhofer, G., eds.: Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94. Volume 903 of LNCS., Springer (1995) 151–163
22. Cremer, K., Marburger, A., Westfechtel, B.: Graph-Based Tools for Reengineering. *Journal of Software Maintenance and Evolution: Research and Practice* **14** (2002) 257–292
23. Marburger, A., Westfechtel, B.: Graph-Based Reengineering of Telecommunication Systems. In: Proc. of the 1st International Conference on Graph Transformation (ICGT 2002). Volume 2505 of LNCS., Springer (2002) 270–285