

Architecture Analysis Tools to Support Evolution of Large Industrial Systems

Tobias Röttschke
TU Darmstadt
Merckstraße 25
64283 Darmstadt, Germany
tobiasr@rs.tu-darmstadt.de

René Krikhaar
Philips Medical Systems
Veenpluis 4-8
5684 PC Best, The Netherlands
rene.krikhaar@philips.com

Abstract

This paper describes an architecture analysis tool-set supporting the evolutionary improvement of the software architecture of an existing medical imaging system. The tool-set has been developed in co-operation of Philips Research¹ and Philips Medical Systems. The medical imaging system at hand is a large and software intensive system. Improvements to its architecture will lead to many development and business benefits, provided the improvements can be made efficiently and without major disruptions. The major challenge in supporting these improvements was finding a technical solution that fits well with the organization and processes of the development department. We came up with an approach that enables the migration to be performed in small increments and in line with the existing processes and procedures established to meet requirements for patient safety and other qualities. Our tools provide relevant information about the current status and recent trends in the software architecture, allowing software architects and engineers to monitor the progress of the migration progress and to identify potential problems as early as possible.

Keywords Architecture analysis, trend analysis, software evolution, software reengineering, early feedback, module architecture

1 Introduction

Philips Medical Systems develops large, software-intensive medical imaging systems, comprising hundreds of hardware and software components. To be competitive in the market, the development process must meet several demanding and often conflicting requirements: high patient

¹The first author performed this work as an employee at the Information and Software Technology department of Philips Research, Prof. Holstlaan 4, 5656 AA Eindhoven

safety, superior image quality, optimal patient throughput, but also short time-to-market and competitive performance. Success depends, among other things, on a well-managed software architecture.

Because the customers make high investments when buying medical imaging equipment, they expect them to create value and improve further for at least a decade. To meet these expectations, we develop the required software in an evolutionary software development process, which allows us to adapt our product continuously to additional and changing requirements.

In the course of this evolutionary development process, the architecture needs to change as well. Software architecture concepts and their implementation may become outdated. In our case, decomposing the system in a flat hierarchy of components with globally available interfaces was not good enough to keep the overview with respect to the complex include dependencies inside the system. This paper describes our effort to manage the evolutionary introduction of alternative module concepts in an existing system of more than 3.5 million lines of code.

The main challenge has been to find an approach that fits well into our organization and does not interrupt the development of new features for an extended period of time. Instead of trying to transform the source code and documents, we want to migrate in an incremental and controlled way as part of the normal development process. Adequate tools should facilitate this process by continuously supplying relevant information to monitor and steer the ongoing activities. Existing reengineering tools do not provide the required functionality, so we implemented our own tool set based on scripting languages and the graph rewriting engine of the roundtrip engineering tool Fujaba [3].

Due to the size and the complexity of the existing legacy system, their organizational impact, and the lack of available resources for this task, the introduction process of the new concepts is a matter of several years. Although there is definitely a need to improve the existing concepts, implementing them has no clear added value for the customer. So

the challenge is to realize the changes evolutionary as integral part of the ongoing product development process. We have approached this challenge in five steps:

1. Analyzing and modelling the initial situation and identifying the resulting issues (section 2).
2. Defining the desired situation and specified the rules, which have to apply (section 3).
3. Identifying the separate steps to achieve the desired situation and setting up a migration plan.
4. Introducing the new concepts in the existing implementation as is, thus without improving it. This is the starting point for the migration.
5. Evolutionary refactoring of the implementation to benefit optimally from the new concepts, while monitoring the progress of this reengineering activity constantly.

This paper focuses on the tools we have set up to enable the last step and reflects our research effort of the last two years. The product development department is responsible for actually deploying these tools and executing the migration process.

2 Initial situation

The original architecture of the medical imaging system that we use as a case study, is based on a fairly simple decomposition of the system into *modules*. These are collections of functional related functions and data types. In this section, we introduce these initial concepts, the dependencies between them, and related consistency rules. In subsequent sections, the desired architecture and the migration will be discussed.

When analyzing the initial situation and specifying the desired one, we use UML class diagrams [8] such as figure 1 to model the basic concepts and we use story diagrams [1] 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 visual rather than a textual notation helps us to discuss and understand the different concepts and identify the relevant issues for the introduction of the new concepts.

Each module realizes an isolated piece of functionality. The hierarchy of modules is flat, meaning that all modules are treated equally and may use functionality implemented by other modules as needed. Typically, one software engineer is responsible for a given module. In the source code, a module is represented by a directory. Modules and directories are mapped on each other using naming conventions

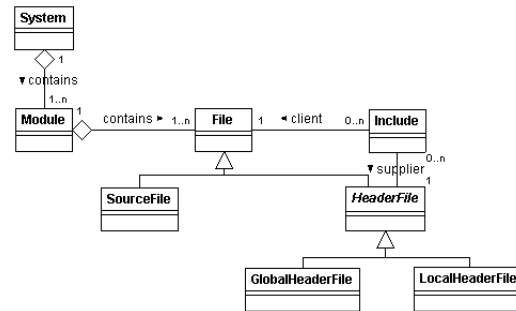


Figure 1. Initial module concepts

according to a coding standard that has been defined as part of the development process.

To organize intra and inter module connections, a distinction is made between local and global header files. Both are identified by naming conventions according to the coding standard. Local header files are only visible in the module they are defined in. Thus, functions and data types defined in local header files of a module form a local scope. To allow export of functions and data types to other modules, a global scope is defined as well, 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. To avoid naming conflicts, all identifiers declared in global header files must be unique.

To check consistency, we first extract the actual directory structure, file names and include directives from the source code and build a graph model of the system according to the graph schema defined by the UML class diagram (figure 1). Then we traverse the graph and check for every single node that the context in the graph matches the patterns defined by all applicable scoping rules like in figure 2.

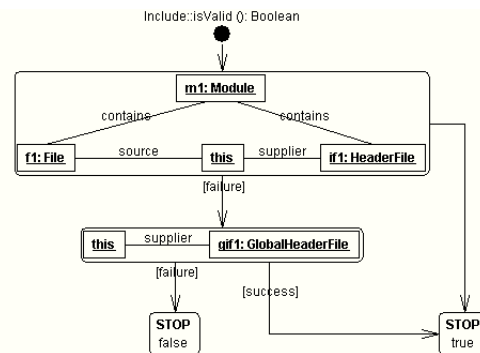


Figure 2. Initial scoping rule

When checking rules, 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 `LocalInclude`).
- A test, if the included `HeaderFile` is a `GlobalHeaderFile` (valid `GlobalInclude`).
- 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 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 according to the result of the graph query.

2.1 Resulting difficulties

Over the years, the system's functionality has increased tremendously. This has resulted in an increasing number of modules and interconnections, adding ever more complexity to both the code and the growing development organization. As a consequence, the system is harder to maintain, fewer individuals manage to keep the overview, and it takes more effort to add new features to the system without hampering the work of other developers.

Using the existing module concepts, the problems raised could not be handled. Having more than a hundred 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.

Although dependencies between teams cannot be entirely removed in practice, it helps to make the interfaces between them explicit and manage the usage of these interfaces in an appropriate way. To help achieving the goals mentioned above, more advanced module concepts were introduced together with rules and tools to monitor the correct use of these concepts. The advanced module concepts differ from parts of the implementation and define an adequate interface concept. When defining our architecture components, we borrowed from component technology as described in [11] and the building block method as described

in [12], but we customized the concepts to make the migration simpler for our developers, who often have a background in physics or electrical engineering rather than in software engineering.

3 Desired situation

The driving idea behind our new module concepts has been *information hiding* [4] 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 dependencies between modules and ambiguity. As a result we hope to decrease code complexity and also limit the organizational complexity required to develop and maintain the system.

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 and related 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

As depicted in figure 3, the basic design concept is a `Block`. `Blocks` contain other blocks, thus forming an aggregation hierarchy. All blocks in the hierarchy should be treated equally, which allows to apply the same rules on different system levels and to move blocks easily in the hierarchy. The root block of the hierarchy is the whole system, 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 explic-

itly by a software engineer. A *MayUse* is valid only if an adequate *sees* relationship exists, meaning that the interface is in the scope of the referring block.

- 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 other building blocks, which is 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.

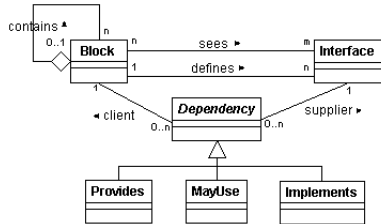


Figure 3. Design Concepts

The *sees* relation is defined merely with the idea of information hiding in mind. The *sees* relation is composed of two other relations expressions in our model:

$$sees = sees_{child} \cup sees_{sibling}$$

These relations are implemented as path expressions in our specification that are materialized during the analysis phase as described in section 4.2. Two examples are shown in figure 4 and figure 5. 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.

In figure 4, the first rule states that a block sees all interfaces of its children. To read this rule, start from the *this* node representing the analyzed block. Next, navigate to the child block *b1* via the *contains* edge. Find a provided interface *i1* by navigating via node *p1* and an attached edges, with together represent a *Provides* dependency between *b1* and *p1*. Note, that the activity has double borders, providing it with *forall* semantics: all possible patterns in the model are matched, and for every match, the green² *sees* edge is added to the model, thus materializing the path expression. As soon as all matches have been found, the story terminates, which is indicated by a transition to a stop node.

²grey in the printed version of this document

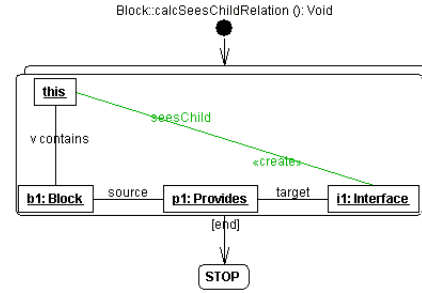


Figure 4. The *sees_{child}* relation

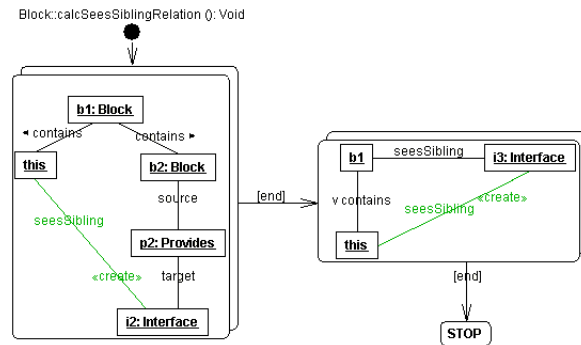


Figure 5. The *sees_{sibling}* relation

The second definition depicted in figure 5 defines that a given block *B* sees all interfaces provided by blocks that are siblings of *B* or siblings of any of its *ancestors*. The story diagram contains two main activities that can be interpreted similar to the detailed explanation above.

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.

Summarizing, the actual usage of interface is restricted by two relationships, *sees* to achieve information hiding and *MayUse* to separate functionality:

$$uses \subseteq MayUse \subseteq sees$$

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 6.

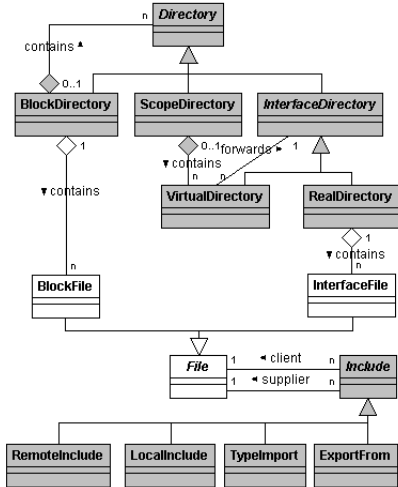


Figure 6. Implementation concepts

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:

Different scoping rules apply for these relationships, as explained in section 3.4. But before we can reason about consistency first, it is necessary to define the mapping between design and implementation concepts.

	BlockFile	InterfaceFile
BlockFile	LocalInclude	RemoteInclude
InterfaceFile	ExportFrom	TypeImport

Table 1. Include relationships

3.3 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 and in section 4 we show how to monitor the progress of making this gap smaller.

The UML diagram in figure 7 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 unique block or interface. For the sake of simplicity, we do not consider Implementation relationships, which would require better parsing techniques than we have currently available.

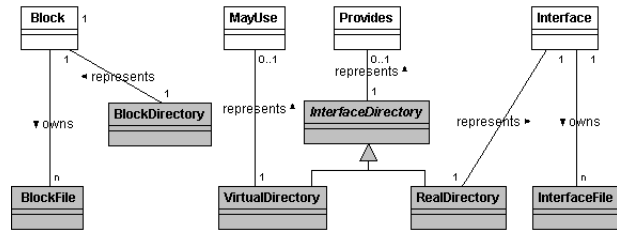


Figure 7. Mapping the concepts

Having defined the mapping between design and implementation concepts, we can check the consistency of our model, which is done by means of so-called scoping rules. These are discussed in the next section.

3.4 Scoping rules

We check a total of 17 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 8 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 this kind of include relationship as RemoteInclude. The analyzed instance of RemoteInclude, represented by the this node, is a relationship between the BlockFile bf1 and the InterfaceFile if1. bf1 is owned by a Block b1 and if1 by an Interface i1. If there is a Dependency d1 between

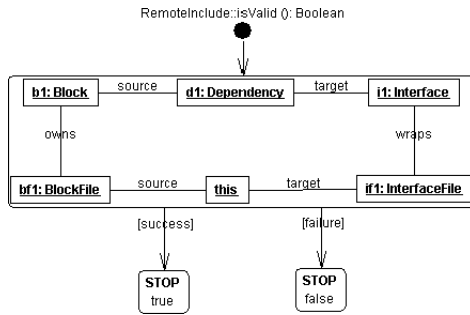


Figure 8. A scoping rule

b1 and i1, which can be either a Provides, MayUse or Implements dependency according to the definition in section 3.1, the complete pattern can be matched and the the query terminates successfully. Otherwise, the query fails.

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.

4 Managing changes

Figure 9 provides an overview of the tool-set we have realized. All components are grouped around a central CMAD³ database containing all collected data, including its history. To examine this data, the software architect uses a web browser to select and request certain information from a web server. Using PHP scripts, the server retrieves the required information and generates tables or diagrams, that are presented to the software architect.

The database is updated once a day with recent from the source code and the (still incomplete) UML model of our system. First, the CMAD tool extracts relevant implementation concepts from the source code and modifies the database accordingly. Next, the IMAN⁴ tool reads the UML model and retrieves relevant design concepts, which are mapped on the actual structure of the source code. Finally, the IMAN tool checks the consistency of design and implementation concepts and stores detected violations in the database. Using the Fujaba tool, the analyzer part of the IMAN tool can be generated from scoping rules (see section 3.4), which are specified once and modified on demand.

³Code and Module Architecture Dashboard

⁴Interface Management

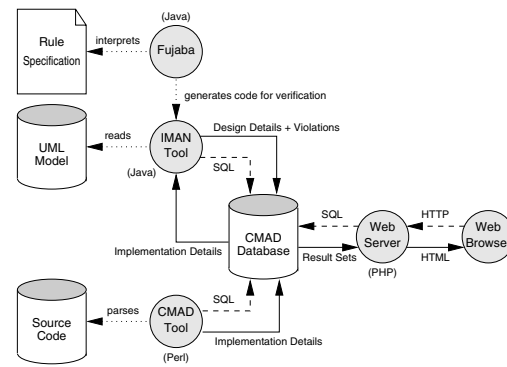


Figure 9. Tool overview

4.1 Extraction phase

Because we make use of relatively few language artefacts, implementation concepts are extracted using PERL scripts rather than a more sophisticated parser technology. The extracted data from the source code is compared to the existing information in the database, which is updated if necessary. However, thanks to the database-centered approach, an alternative parser could easily be integrated.

The updated information should be available during of-ice hours, so our tools update the database overnight. On working days, we consider only files, which have been successfully submitted for integration according our process model since the last run of our tools. In the weekend we analyze the complete source code to ensure the consistency of the database. We consider all modifications that have actually been performed, even if they are not allowed according to the process model.

Design concepts are extracted, by filtering relevant information from the UML model using the Rose Extensibility Interface [6]. Because we consider far less design concepts than implementation concepts, we can afford to analyze the complete UML model every day and update the database.

Figure 10 shows an example⁵ for the provides dependency between the block `Subsystem2` and its interfaces. This diagram is part of a so-called *Interface Specification*, which is defined for every building block. The `mayuse` and `implements` dependencies are defined in similar ways. Note, that this information belongs to the design of the system, which defines the desired situation. Initially, the actual implementation does not match this specification, and we have to extract both source code and UML model to judge the consistency of design and implementation.

⁵Note that we have concealed business relevant information, like names of subsystems and interfaces. Nevertheless, all presented data gives a realistic idea of the figures extracted from the existing system.

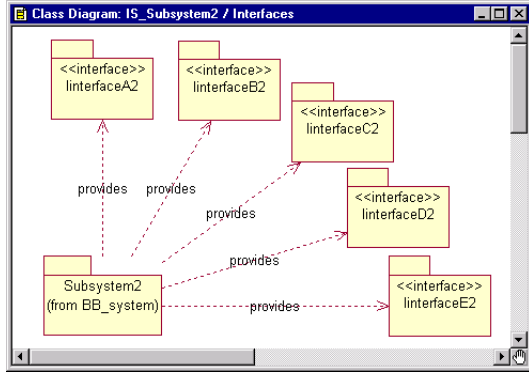


Figure 10. A *provides* relation in Rational Rose

4.2 Analysis phase

The analysis phase starts with an intrinsic graph model of the system as provided by the extraction tool. Now the rules defined by story diagrams in Fujaba can be applied to this model. First, we add derived information to the model like the sees relation described in section 3.1. Then we check the correctness of all model elements according to the defined scoping rules as described in section 3.4.

Specifying scoping rules as visually defined graph rewriting rules has two benefits for us. First of all we can generate the analysis part of our tools using the code generator of the Fujaba tool, allowing us to easily keep the rule specification consistent with the actual implementation. Second, the visual representation makes it easy for us to discuss the rules with developers and software architects.

4.3 Presentation phase

For the presentation of the results we chose to provide information about relevant trends in a single overview and allow further navigation to the desired level of detail. Subsequent sections describe a typical session performed by the software architect to determine the current status and recent progress of the migration process.

First the software architects checks a diagram which provides a quick overview of recent changes of a few major aspects of the architecture as described in section 4.3.1. Next he looks in more detail after the status of the current implementation as discussed in section 4.3.2. After that, the software architect looks up the design status of the desired situation, which is described in section 4.3.3. Finally, he checks the consistency between design and implementation and determines the progress of improving this consistency as discussed in section 4.3.4.

4.3.1 Trend overview

It would make no sense to examine all information produced by our tool every day, because there is far too much information available. On the other hand the software architects are eager to receive relevant feedback as early as possible to react before resources are wasted. For example, platform layers may not depend on application-level layers to allow independent development of the platform as long as the interface is not changed. The number of violations introduced earlier should ideally decrease, but at least not increase. So the software architects want to guard the number of illegal dependencies and be alerted quickly, if things get worse. To satisfy this desire, we came up with the so-called Dashboard View to provide a top-level overview of recent trends in the source code at a glance (figure 11).

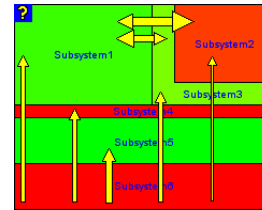


Figure 11. The Dashboard View

This view aggregates implementation details per subsystem and is automatically updated once a day. It presents a summary of relevant changes in the source code during a period that is determined in a configuration file – currently four weeks.

The Dashboard View presents the system as a rectangle partitioned into six square-angled polygons representing the subsystems⁶. The area covered by a polygon is proportional to the code size of the represented subsystem, and currently calculated by hand. The layout of the polygons visualizes the layering of the subsystems, which is not strict. Thus, a block uses automatically any other block drawn below it.

- A polygon *A* is placed *above* another polygon *B*, if *A* may directly use *B*, but *B* should not use *A*.
- A polygon *A* is placed *next* to another polygon *B*, if there is no uni-directional use-relationship between *A* and *B*, i.e. either *A* may use *B* and *B* may use *A* or both may not use each other at all.

The implemented system contains some exceptions from the intended use-relationships, some of them for good reason. Others are illegal uni-directional uses-relationships

⁶Currently, the polygons are calculated by hand, but it would be interesting to find an algorithm to combine partitioning and layering for arbitrary systems

that are explicitly presented in the dashboard view to make aware of violations and help to monitor actual trends over time. These relationships are represented by single-headed arrows from the client to the supplier. The thickness of an arrow grows logarithmical with the number of violations.

Bi-directional use-relationships between different subsystem blur the borders of the subsystems and increase the organizational complexity within the development department. Although these use-relations might be practically inevitable, their number should not grow if possible. The Dashboard View shows bidirectional use-relations as double-headed arrows. The thickness of an grows logarithmical with the number of bi-directional relationships. The size of the arrowheads indicates the ratio of relationships in one and the other direction.

The following color scheme has been chosen to indicate changes in the code archive and pronounce a judgement on the observed trend: neutral yellow represents unchanged entities. Green indicates either a good trend like fewer forbidden use-relationships or an expected trend like increasing code size. Red marks either a bad trend like more forbidden use-relationships or unexpected trends like decreasing code size.

4.3.2 Implementation details

In this section, we discuss the presentation of code statistics, that have been lifted to building block level. These statistics help to detect changes in size and complexity in the implemented system. Abstracting from the implementation details helps to get an initial overview of the system as a whole. Taking this as a starting point, we allow to zoom deeper into the hierarchy to analyze the interesting parts of the system in more detail.

Using this approach, we can limit the amount of information presented to the software architect and help him to identify those parts of the system where developers are currently working on and complications might occur. In our opinion, trend metrics are easier to interpret than absolute metrics. They provide means to focus on recent changes and mask out less active parts of the system as described in [7]. In this way the software architect is able to identify relevant information and act immediately if necessary.

Block implementation statistics The block implementation statistics provide quantitative information about directories in the source code that are interpreted as implemented blocks or interfaces according to the new architectural concepts. Initially, the existing system does only consist of the original modules, which are treated as leaves of the block hierarchy. These are reclustered according to their functionality, resulting in an unbalanced aggregation hierarchy rather the flat original architecture. The increasing number

of building blocks and interfaces helps the software architect to monitor the rearchitecting progress in a certain area.

Block interconnection statistics The block interconnection statistics as depicted in figure 12 present information about include dependencies between building blocks and interfaces. This information is based on include directives in source and header files and lifted to the desired level of abstraction. When analyzing module interconnection statistics, the software architect usually starts with dependencies between top-level blocks. When he notices undesired dependencies, he can navigate downwards through the aggregation hierarchy until he reaches the required level of detail.

Blocks of system	Date	Blocks of system					
		Subsystem1	Subsystem2	Subsystem3	Subsystem4	Subsystem5	Subsystem6
Subsystem1	06-03-2002	11142	347	56	93	105	6137
	05-03-2002	+1					+3
	27-02-2002	+4					+22
	06-02-2002	+11					+12
Subsystem2	06-03-2002	118	7656	18	22	6	2228
	05-03-2002						
	27-02-2002						
	06-02-2002	-4	-6				-28
Subsystem3	06-03-2002	64		2084	7	55	1735
	05-03-2002			+1			+2
	27-02-2002			+1			+3
	06-02-2002			+1			+4

Figure 12. Block interconnections

Initially, there are lots of illegal include dependencies between files and hence blocks and interfaces. The reason is of course, that the scoping rules have been defined after creating the include dependencies. During the refactoring of the module architecture, the number of illegal include dependencies should decrease, but at least not increase. Newly introduced illegal includes dependencies are detected shortly after creating them, so the software architect can react quickly and urge the responsible developer to avoid this dependency.

4.3.3 Design details

Design documents of the system, that exist at the beginning of the migration process, are still written in terms of the original module concepts. It would not be sensible to translate these documents directly to the new building block concepts, because most benefits of the new concepts would be abandoned. So the rework of the documents should be combined with a refactoring step, where the existing mod-

ules are adequately distributed over blocks, eventually after splitting or joining them.

For every block, the provided interfaces must be defined as described in section 3.1. Apart from a textual specification, each design document contains a UML diagram where the relevant blocks and interfaces as well as the dependencies among them are defined. The UML model composed from all those diagrams is the leading definition of design concepts that are analyzed by our tools to check the consistency between design and implementations.

During the migration process, the design documents are reworked top-down. This allows us to decouple subsystems first and thus help the responsible teams to development these subsystems in parallel. Later on, lower level blocks will be defined in a similar way to decouple the activities within the teams responsible for single subsystems.

Block design statistics Figure 13 shows the design statistics of top-level blocks and interfaces as defined in the UML model. The figures demonstrate clearly the effect of our top-down approach: Mostly high-level blocks and interfaces have been defined yet. Because the tools are able to present recent trends and navigate to lower-level building blocks similar to the block interconnection example described in section 4.3.2, the software architect can easily monitor and analyze the progress of the redesign process.

Subsystem	Date	Contained components				Contained interfaces			
		Total	Level2	Level3	Level4	Total	Level1	Level2	Level3
Subsystem1	18-01-2002	5	5			5			
Subsystem2	18-01-2002	5				5			
Subsystem3	18-01-2002	3	3			5	2	3	
Subsystem4	18-01-2002	7	4	2	1	5	5		
Subsystem5	18-01-2002	10	2	6	2	6	6		
Subsystem6	18-01-2002	39	4	5	22	30	23	2	1

Figure 13. Blocks in the Rose model

4.3.4 Violation statistics

To check the consistency more explicitly, we have defined scoping rules as described in section 3.4. In this section, we describe how the results of checking these rules are presented. We distinguish validation rules, which define whether the designed UML model complies in the desired manner to the principle of information hiding, and verification rules, which check the consistency between design and implementation.

As an example, figure 14 shows the aggregated implementation violations for the top-level building blocks. All columns represent verification rules that have been checked on implementation concepts. For instance, the violation

BlockDirectory Not Valid occurs, if the source code contains a `BlockDirectory`, which cannot be mapped on a related `Block` in the UML model. A *RemoteInclude Not Valid* violation occurs, if the `RemoteInclude` relationship is defined between a `Block` and an `Interface` without an appropriate dependency between them (cf. figure 8).

Subsystem	Date	BlockDirectories Not Valid	RealDirectories Not Valid	VirtualDirectories Not Valid	Includes Not Unique	LocalIncludes Not Valid	RemoteIncludes Not Valid	ExportFroms Not Valid	TypeImports Not Valid
Subsystem1	18-01-2002	59	38	481		185	17050		691
Subsystem2	18-01-2002	56	8	95		277	8101		703
Subsystem3	18-01-2002	26	9	102			3937		4
Subsystem4	18-01-2002	26	21	262		103	1524		23
Subsystem5	18-01-2002	43	28	230		130	2356		481
Subsystem6	18-01-2002	111	53	410		630	5113		283

Figure 14. Implementation violations

As we still are very early in the migration process, many of the existing implementation concepts are not correct with respect to the introduced scoping rules. When comparing the *RemoteInclude Not Valid* violations with the number of existing block interconnections in figure 12 virtually all `RemoteInclude` relationships are not valid yet. As the migration proceeds, the number of violations should decrease rapidly. Having this tool available, makes it easy to keep control over the process and monitor the progress.

Our tool supports the work of the software architects as well as of the software engineers. A software architect uses typically the view presented in figure 14, but compares the trend of the statistics similar to figure 12 to monitor the progress of removing the violations. If not satisfied with the result, the software architect can navigate deeper in the block hierarchy for further analysis or click on the number of violations for a certain rule to get a list of all occurrences of this violation. Afterwards, he can discuss the issue with the responsible software engineers.

The software engineer is usually only interested in the violations in the block he is responsible for. After navigating to his block, he can click on the name of the block and gets a list of all violations in his block. By bookmarking this page, he effectively gets a personal to do list, which is updated daily. After discussion with his project leader the software engineer plans what violations to remove.

5 Conclusions

We described our experiences, how a set architecture analysis tools based on graph rewriting, database and web technology can be successfully applied to support an evolutionary improvement of architectural concepts of a large, embedded 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 [13]). 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.

The idea of applying graph grammars for software engineering tools has already been described by [10] and [2], but we have transferred this idea successfully to a large scale industrial case. Driven by safety and other quality requirements, we measure and visualize the progress of the migration process, which is performed manually according to standard procedures rather by automatic transformation.

The tools validate whether the design is well-formed according to the principle of information hiding and verify the consistency between design and implementation. We keep the history of statistical information on violations for the software architect and allow software engineers to analysis concrete occurrences of these violations. In the design of our tools, we focussed on making relevant information easily accessible via intranet and preparing them for further extensions by choosing a simple, database-centric approach.

Ideally, all violations should be removed at the end of the migration process. In practice, this situation will probably never become reality, because there are valid reasons to accept certain violations. As long as a violation is not perceived as a real problem, it can be left in the system. Trying to remove the violations could introduce new bugs and would consume resources that would be needed for more serious matters. In fact, our policy can be compared with other software development processes, for safety critical systems, like the NASA space shuttle program, where problem reports are kept over several releases, but solved only if they are life threatening or mission critical [9].

During one year after the deployment of the first version of the tool-set, we gained a lot of insight about the dynamics of the migration process, but it will go on for at least several more years. Moreover, we could quickly adapt our tools to deal with a number of completely different systems and thus transfer some of our ideas to new domains. So we decided to further improve our tools, to extend their scope, and finally to build up a quality database containing historical information about many relevant aspects of the software of our product. Using this approach, we are confident to

achieve more and more control over our complex, evolutionary development process.

Acknowledgements The author wants to thank William van der Sterren and Auke Jilderda for reviewing this paper. Wim Decroix did an excellent job in designing and implementing a major part of the tools. This work has been performed in the Eureka $\Sigma!$ 2023 programme ITEA 99005, ESAPS and Ip00004, CAFÉ.

References

- [1] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph grammar language based on the unified modelling language and java. In *Proc. 6th Workshop on Theory and Application of Graph Transformation (TAGT'98)*. University-GH Paderborn, Nov 1998.
- [2] B. Kullbach and A. Winter. Querying as an enabling technology in software reengineering. In *Proc. Conference on Software Maintenance and Reengineering*, 1999.
- [3] U. Nickel, J. Niere, and A. Zündorf. Tool demonstration: The fujaba environment. In *Proc. of the 22th International Conference on Software Engineering (ICSE), Limerick, Ireland*, pages 742–745. ACM Press, 2000.
- [4] D. L. Parnas. A Technique for Software Module Specifications with Examples. *Communications of the ACM*, 15:330–336, 1972.
- [5] A. Postma. A Method for Module Architecture Verification and its Application on a Large Component-Based System. *Information and Software Technology, Elsevier Science*, 2002. to appear.
- [6] Rational Software Corporation. *Using the Rose Extensibility Interface*, 2001. http://www.rational.com/docs/v2002/Rose_REI_guide.pdf.
- [7] T. Rösche, R. Krikhaar, and D. Havenith. Multi-View architecture trend analysis for medical imaging. In *Proc. IEEE International Conference on Software Maintenance*, page 107. IEEE, Nov 2001.
- [8] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1999.
- [9] N. F. Schneidewind. Investigation of the risk to software reliability and maintainability of requirement changes. In *Proc. IEEE International Conference on Software Maintenance*, pages 127–136. IEEE, Nov 2001.
- [10] A. Schürr, A. J. Winter, and A. Zündorf. Developing Tools with the PROGRES Environment. In M. Nagl, editor, *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, volume LNCS 1170, pages 356–369. Springer, 1996.
- [11] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley/ACM Press, 1998.
- [12] F. J. van der Linden and J. K. Müller. Creating architectures with building blocks. *IEEE Software*, Nov 1995.
- [13] J. Zhao. Change Impact Analysis for Architectural Evolution. In *Proc. Workshop on Formal Foundations for Software Evolution*, Lisboa, Portugal, Mar 2001.