Using Feature Modeling for Program Comprehension and Software Architecture Recovery

Ilian Pashov, Matthias Riebisch

Technical University Ilmenau, Max-Planck-Ring 14, P.O. Box 100565

98684 Ilmenau, Germany

{idpashov|matthias.riebisch}@tu-ilmenau.de

Abstract: The available evidence in a legacy software system, which can help in its understanding and recovery of its architecture are not always sufficient. Very often the system's documentation is poor and outdated. One may argue that the most reliable resource of information is the system's source code. Nevertheless a significant knowledge about the problem domain is required in order to facilitate the extraction of the system's useful architectural information.

In this approach feature modeling is introduced as an additional step in a system's architectural recovery process. Feature modeling structures the system's functionality and supports reverse engineering by detecting the relations between source code elements and requirements. Tracing these relations may lead to a better understanding of the program's behavior and the recovery of various architectural elements. In this way, by providing a mapping between source code and features, the system's feature model supports program comprehension and architectural recovery.

The approach is developed as first part of a migration methodology towards a component-based architecture of legacy systems. Recovered information about features and architecture is collected in a repository to enable a refactoring as next step. The approach is currently applied in a large project for reengineering of an industrial Image Processing System.

Keywords: Feature modeling, software refactoring, architectural recovery, program comprehension, design recovery, legacy systems, reengineering, reverse engineering, traceability

1 Introduction

Due to the rapid development of the software technology during the last decades and the increased demand for software products, a large number of software systems has been developed. Many of them have a long life cycle and contain a lot of company "know-how". On the other hand, the continuously changing needs of the business environment require those systems to be always up to date with the latest technologies and to evolve during their life cycle. Evolving often means refactoring or migrating to a new approach, for example component based systems or product lines. However, to succeed in such a step, a full understanding of the software system is required leading to the need for its architecture and design decisions to be recovered.

Due to the long life cycle of these systems, in most of the cases it is impossible to keep the same development team. A lot of knowledge about the system is lost along with the developers. Often the system's documentation is out of date and insufficient. This means that additional help is required for the architectural recovery of the system from the reverse engineers and the reverse engineering methods. Their task is to extract and reconstruct the system's design, based solely on the available information.

It is true that the most reliable information resource for the reverse engineer is the system's source code, although this proves most of the times to be insufficient. In order to succeed, the reverse engineer needs to combine the information gained from the source code with knowledge about the problem and programming domain. In the case of large systems a well-defined recovery process is needed so as to minimize the risk of the whole process.

This paper presents an approach to program comprehension and software architecture recovery of legacy systems based on the use of feature diagrams and feature modeling.

In general, the approach elaborates the idea of combining system domain knowledge and program understanding. It uses feature modeling as a way of expressing domain knowledge when at the same time bridges between system experts, users and reverse engineers. Finally, it serves as a means of generation and verification of hypotheses.

This approach defines an architecture recovery process initiated at the Feature Modeling level. In this respect, Feature Diagrams provide an orientation for the establishment of architectural element hypotheses. A mechanism for the verification of those hypotheses

based on cross-references is presented, along with different types of feature models, respectively corresponding to design and decisions objectives.

A case study taken from an ongoing industrial project illustrates the applicability of this new technique.

2 State of the art

The presented approach is based on several software engineering methodologies, both in the field of forward and reverse engineering.

2.1 Program comprehension

Most of program comprehension methodologies are based on source code analysis. Program understanding and problem domain knowledge are connected for the first time in [Brooks et al. 1983]. More precisely, this paper presents the establishment of a top-down hypothesis, according to which, program comprehension leads to the construction of a mental model for successive knowledge domains. Each of these domains consists of objects, properties, relations and operations. The succession of domains may include the problem domain, the domain of a mathematical model of the problem, the algorithm domain, the programming language domain, and so on. One must also achieve comprehension of the relationships that exist between adjacent domains.

In [Letovsky et al. 1986] the ideas of Brooks are elaborated. According to this paper, the task of understanding a program is one of uncovering the intention "behind" the code. Intentions are described as goals. Techniques for realizing goals in a particular implementation are called plans. Plans are similar to algorithms, but they may involve non-contiguous elements and may be combined in ways not usually considered for algorithms. For example, two plans involving loops may be combined into a solution using a single loop implementing two distinctive goals.

The authors in [Rajlich et al. 1994] take the development of top-down program understanding techniques one step further. According to their approach, the programmer creates a chain of hypotheses along with subsidiary hypotheses. These are consequently verified in the code. Similar hypotheses are used here.

These three approaches apply only a top-down procedure, thus missing low level dependencies, which usually are visible only within source code.

A recent discussion about the role of domain knowledge in program understanding can be found in [Rugaber et al. 2000]. The author shows how a model of an application's domain is able to serve as a supplement to programming-language-based analysis methods and tools. A domain model contains knowledge of domain boundaries, terminology and possible architectures. This knowledge can help an analyst set expectations for program content. Moreover, a domain model can provide information on how domain concepts are related. The author has described a number of ways for presenting the domain knowledge. In this paper, they are extended by feature modeling to build a bridge to later architectural development.

2.2 Architecture recovery

The software architecture represents the problem solution on a higher abstraction level than the source code, thus reducing the complexity of software systems. This makes architecture recovery the most important task in reverse engineering. Additionally, a well defined process definition, as well as appropriate tool support is required.

The workbench Dali ([Kazman et al. 1997]) helps the reverse engineer in extracting, manipulating, and interpreting architectural information. Dali defines a framework for architecture recovery, a process and the needed tool support. The Architecture Tradeoff Analysis Method (ATAM) [Kazman et al. 1998b] developed by the Software Engineering Institute (SEI) focuses into analyzing software systems with respect to specific quality attributes. ATAM also provides techniques to reconstruct an architecture from a system's implementation. Even if contributing to architecture recovery techniques, Dali and ATAM do not exploit domain knowledge.

The authors in [Finnigan et al. 1997] introduce the concepts of a reverse engineering environment called "software bookshelf" as a means to capture, organize and manage information in legacy software systems. They distinguish three roles directly involved in the construction, population and use of such a bookshelf: the builder, the librarian and the patron. From these perspectives they describe requirements for the bookshelf as well as a generic architecture and a prototype implementation. This approach supports re-documentation of the legacy systems very well, however no overview over domain knowledge is provided. This has to be added to enable architecture recovery and later refactoring.

Storey in [Storey et al. 1997] presents the reverse engineering tool, Rigi. The Rigi system provides two contrasting approaches for presenting software structures in its graph editor. The first approach displays the structures through multiple, individual windows. while the second one (Simple Hierarchical Multi-Perspective (SHriMP) views) employs fisheye views of nested graphs. Rigi is an open environment and could be

extended in order to further support architecture recovery. Extensions of Rigi for a tool-based feature oriented architectural recovery could support the approach presented here.

Riva in [Riva et al. 2002] presents an extension of Rigi with support for Message Sequence Charts (MSC). This paper discusses also the idea of feature oriented reverse engineering. However there is no concept for presentation of features and for analyzing dependencies between relations among features and architectural elements.

2.3 Features and feature modeling

Feature modeling was introduced originally by the Feature-Oriented Domain Analysis (FODA) methodology [Kang et al. 1990] for structuring domain properties from the customers' point of view. In addition, feature models show relations between capabilities and consists of a hierarchy of "contains" and "requires" relations between features. Each feature can be optional or mandatory for a set of systems within a domain. Additional relations between features across the hierarchy can be described, i.e. alternatives or mutual exclusion. Figure 1 shows an example of a simple feature model.

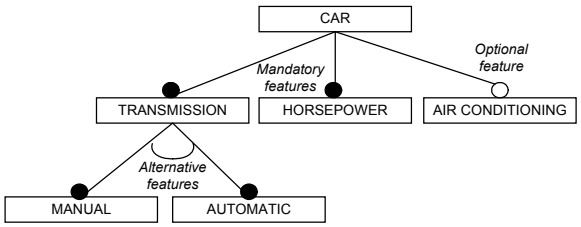

Figure 1. Car feature model [Kang et al. 1990]

In their work [Czarnecki et al. 2000] features are used for modeling the commonality and the variability in a domain model. Feature diagrams are extended by assigning constraints and grouping relations to features. In [Riebisch et al. 2002] these relations are extended and some ambiguities are removed.

The ideas of modeling and expressing relations presented in FODA are further developed in the Feature-Oriented Reuse Method (FORM) [Kang et al. 1998]. FORM extends FODA to the software design and implementation phases and describes how the feature model is used to develop domain architectures and components for reuse. In FORM the feature space is split into different views, depending on the interest one might have in system development. However, it is difficult to use the FORM feature views because their separation is not defined precisely enough. Furthermore, reverse engineering needs a more general separation of the feature spaces.

2.4 Architectural modeling

Architectural modeling represents the framework for constructing an application. An architectural model is the high-level design of the application. It defines the application's basic building blocks. It also defines the basic partitioning and interconnections necessary for constructing the application. The architectural model serves as a frame for organizing architectural element hypotheses during an architecture recovery process.

Two of the fundamental works on architectural modeling are the "4+1 view" model proposed by Kruchten [Kruchten et al. 1995] and the "4 views" architectural model proposed by Hofmeister, Nord & Soni [Hofmeister et al. 2000].

The "4+1 View" model suggests organizing the architectural descriptions in five different categories called views: logical view, process view, physical view and development view. The fifth view, namely user's view, contains scenarios and use cases and is used for defining requirements and for validating the previous four. The model separates static and dynamic aspects of a software architecture. The solutions to functional requirements are concerned mainly in the logical view. The process view focuses on dynamic aspects of the model and also describing runtime behavior. The physical view shows the solutions primarily to non-functional requirements and maps software to hardware. The development view focuses on the actual software module organization and on the software development environment. It also focuses on requirements related to the ease of development, software management, reuse or commonality, and to the constraints imposed by the toolset or the programming language. The "4+1 views" architectural model has become very popular during the last decade, especially for new development.

The "4 views" architectural model also proposes separate descriptions of the different architectural parts. The four views presented are: conceptual view, module view, execution view and code view. The conceptual view describes the system in terms of its major design elements and the relations between them. The module view presents the decomposition of the system and the partitioning of modules into layers. The code view is the organization of the source code into object code, libraries and binaries, then in turn into versions files and directories. The mapping from software to hardware and distribution of the software components is the task of the execution view.

Both models have their advantages and disadvantages with the "4 views" architectural model addressing the case of "mixed" software systems: build on both object and non-object oriented technology, in a more efficient

way. ”Mixed” software systems are the common case in software legacy systems. For this reason further in this paper, the “4 views” model is used as the basic architectural model in the architecture recovery process.

3 New approach

The new approach presented in this paper elaborates on the idea of applying problem domain knowledge to program comprehension. It combines top-down and bottom-up activities for architectural reconstruction.

Architectural element hypotheses are generated based on domain knowledge, which is presented in a top-down view by feature models. The verification of these hypotheses is based on bottom-up tracing procedures.

Finally, feature models are introduced as central modules for bridging the gap between requirements and architecture.

The architecture recovery process consists of 4 major activities (Figure 2): requirements and domain analysis, legacy architecture analysis, architecture recovery by a hypotheses-verification procedure, and scenario driven dynamic analysis. The hypotheses- verification procedure consists of hypotheses establishment and their verification. Section 3.3 explains these major activities in more detail. Figure 3 will show the whole process at a detailed level.

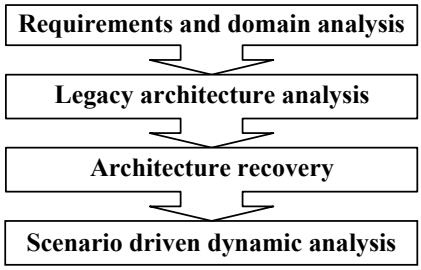

Figure 2. - Feature oriented recovery process – major activities

The architecture recovery process provides information for a later architectural refactoring. Therefore both an architectural description with different views and test cases are elaborated here.

3.1 Design Objectives and Design Decisions feature models

According to [Kuusela et al. 2000] two types of requirements are involved in the life cycle of software systems: *design objectives* and *design decisions*. Both of these should be taken into consideration for the architecture recovery process. This approach achieves this through splitting the features of the system in two spaces and through establishing the corresponding feature models.

All properties of the system related to the functional requirements are called Design Objectives features. They are presented within a Design Objectives feature model (Figure 6), similar to the one described by FODA and Czarnecki and Eisenecker. The features are presented in a tree structure with “decomposed-to” and “requires” relations. Additional relations across the hierarchy could be of the type “excludes”. As a result of the domain analysis each feature can be mandatory or optional. A concrete software system in that domain corresponds to a set of the model’s features.

The design decisions reflect the solution domain of the requirements analysis and capture the intension “behind” the designers’ decisions. They may be presented with a Design Decisions feature model (Figure 7). The nodes in the hierarchy represent solutions. They may be structures, design patterns, third party components or architectural decisions.

While the Design Objectives feature model describes the problem space, its Decisions counterpart describes the solution space.

3.2 Architectural model

Architecture recovery requires an appropriate architectural model as a bases for the resulting work. As already mentioned, this approach uses the “4 views” architectural model as a base model for the recovered software architecture. Focus is placed mainly upon the conceptual, module and code views. The execution view is not yet considered and will be part of future work. The conceptual and the module views could be split into several layers to conform to the complexity of the software systems.

Separating a system into views embodies a significant and crucial part of the work. However this activity follows the principles of conventional architectural design as described for example together with the “4 views” model [Hofmeister et al. 2000].

3.3 Feature oriented architecture recovery process

Figure 3 shows the whole recovery process as an UML activity diagram with a distinction between three levels of development: requirements, architecture, and implementation. It shows the sequence of activities for collecting architectural information in an iterative manner. The major activities contributing to the architectural description are “Static architectural description” and “Dynamic architectural description”.

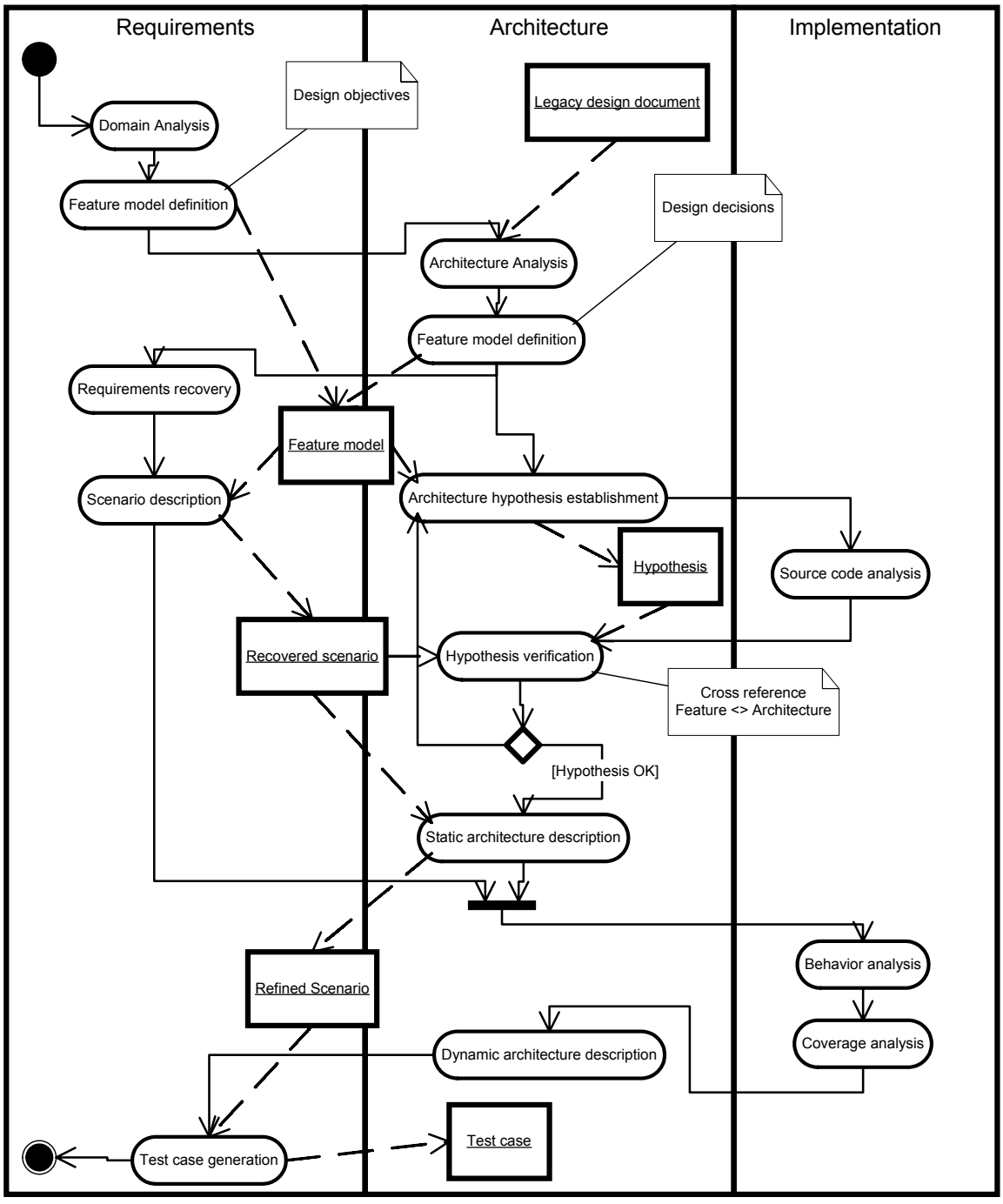

Figure 3. - Feature oriented architecture recovery process - activity diagram

3.3.1 Requirements and domain analysis

In this step both the requirements of the domain and the requirements of the legacy system are analyzed. The usual activities of domain analysis and requirements analysis are applied:

- study of documents, forms, and guidelines
- interviews with users
- interviews with experts for user support and the system's maintenance.

The step results in a Design Objectives feature model and a refined requirements description. In the feature model, at this stage a distinction between mandatory and

optional features is not yet considered. While the requirements are not yet verified by a code analysis, this model serves as a stock for later hypotheses. If possible, the feature hierarchy is established similar to a well-structured functional decomposition of the system's architecture, thus easing the hypotheses definition.

The requirements are refined by the use of case descriptions and scenarios. These are later used during the hypothesis assessment for verification. Finally, the assessed results are used for further refinement both of behavioral description and test cases. Requirements and Design Objectives feature model are contained in a repository with references between them.

3.3.2 Legacy architecture analysis

The architecture analysis begins with the studying of existing documents describing the legacy system's architecture. The documents are studied with comparison to requirements derived from the previous step. Knowledge about the domain's reference architectures of the and design patterns is included in the analysis process as a later source of hypotheses. During later hypothesis' verification possibly outdated information is identified.

The results of this step are collected in a Design Decisions feature model. Similar to Design Objectives this feature model serves as a hypothesis stock for architectural and design elements. It is stored in the repository, together with links to the referenced documents.

Both Design Objectives and Design Decisions feature models are created iteratively over several steps. After each step the feature models are refined according to the remarks of the system experts (users, developers and problem domain experts). According to the progress of refinement both feature models become complete more and more.

3.3.3 Hypothesis-Verification Procedure

This step performs an incremental recovery of the system's architecture through establishment and verification hypotheses. A hypothesis describes a supposed relationship among a feature and an architectural element. At this stage architectural elements can be of different types, for example component, interface, class, method, subsystem or communication protocol element. The hypotheses are collected in cross-reference tables (Tables 2 and 3) with references to elements in the architectural description and to source code. The legacy source code is analyzed in a conventional way: data structures, functional structures and the control flow between modules serve as a source of information. The results of the analysis refine a hypothesis through architectural diagrams. If possible the scenarios are also refined.

The assessment of a hypothesis may lead to new hypotheses, which in turn are added to the cross-reference tables and which are adding elements to the architectural description. A hypothesis verification can fail due to an invalid feature-architectural element relationship or because of a non-present feature. In the latter case the feature is marked as "not implemented in the particular system", the architectural element is unlinked.

After completing this step all verified hypotheses are marked inside the cross-reference tables. The description of the architectural elements is refined and composed to a more complete architectural description. The description is built in the 4 architectural views.

3.3.4 Scenario driven dynamic analysis

In order to complete the architectural description, the scenarios from the first step are applied to analyze the dynamic aspects of the system and to complete the code analysis. In this stage, existing tools and methods are used: coverage analyzers and profilers. The analysis is used to refine or to correct the conceptual, module or code view of the architecture. Furthermore, concurrency aspects may be described and execution scenarios may be further refined.

The gathered system's behavioral information may be used for the construction of test cases, needed for later refactoring. Test cases usually contain behavioral information at a higher level of detail than a requirements specification.

3.3.5 "Features to Architectural Elements" cross referencing

The verification of the established architectural elements hypotheses is performed though tracing their relations to features (Figures 4 and 5). The verification is performed using two cross reference tables: „Features to Architectural Elements" (Table 2) and „Architectural Elements to Features" (Table 3). The rows of the first table are assigned to features while its columns carry architectural hypotheses. The crossed cells list the architectural elements related to the feature in the row. The second table lists all features in columns and architectural elements hypotheses in rows. A confirmation mark is given in each crossed cell where the feature in the column corresponds to the architectural element in the row.

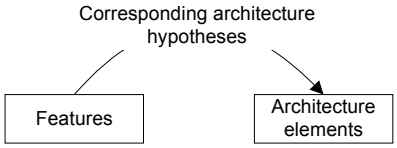

Figure 4 – Tracing the relations features to architectural elements

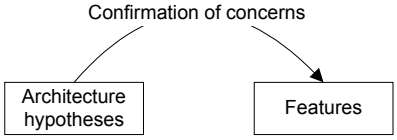

Figure 5 – Tracing the relations architectural hypotheses to features

The results of cross-referencing represents the verification of architectural elements hypotheses. If there is no feature corresponding to a hypothesis then the hypothesis is considered to be false. A feature without any corresponding architectural element could be obsolete, invalid or optional. If none of these is true, then this feature should enforce the establishment of a new, apparently missing hypothesis.

The cross-reference tables also show the relations between features and code. From a more abstract view, they serve bridging the semantic gap between the system's requirements and implementation. In this way they give support for navigation during source code analysis. References from features to source code or configuration is shown by cell entries of the "Feature to Architectural Elements" table.

For cases where a more detailed source code analysis is required, features are traced to source code in the code map.

3.3.6 „Features to Source Code" cross referencing

The „Features to Source Code" cross-reference table, called code map, describes the relations between features and concerned source code elements (such as global variables, constants, procedures, functions or classes) as well as the locations of these elements in the source code. The table rows list the features of interest while the columns list the concerned source code elements. The crossed cells contain pointers to source code locations. Table 1 presents an example code map with file names and line numbers as locations. A tool can provide these references as hyperlinks to ease navigation.

By references at this level a more detailed tracing is possible. In more complex systems a global code map would become very large. Therefore, the code maps are built only for those parts, where results of former references are not satisfying, and where a more detailed analysis is required. For a reverse engineer, code maps enable navigation and in-depth exploration of critical source code portions. Depending on the source code's quality this is necessary only for 5 .. 10 % of a system.

4 Tools

The information gathered during reverse engineering shall later be used for refactoring an analyzed system towards a component-based architecture. Therefore, recovered architectural facts are collected in a repository together with requirements and feature models. While performing the described hypothesize and verification activities in an iterative manner, the information in the repository has to completed step by step. Tools have to support developers in merging and unifying the puzzle of discovered facts and in getting an overview. For the activities of the described approach, tools have to fulfill the following requirements:

- To provide a graphical representation for architectural models using UML
- To offer reverse engineering facilities, producing UML models out of source code
- To be capable of handling incomplete and inconsistent information
- To analyze models for completeness and consistency
- To manage references (i.e. hyperlinks) between model elements
- To maintain hypotheses in cross-reference tables, with hyperlinks to model elements and with attributes for the verification status
- Possibility of notes for every item
- To offer manipulating functions for models similar to those of refactoring tools, i.e. [Boger et al. 2002]
- To support fuzzy search for comparison and navigation, for example by a keyword search

Currently, there are no tools available fulfilling these requirements. However, there are some CASE tools in the market with rich configuration possibilities like MetaCASE tools, or with an open plug-in interface, i.e. the ArgoUML successor Poseidon [Poseidon].

Currently, the approach is supported by a configuration of several tools and products. Relational databases are used for maintaining cross-reference lists. They are connected to drawing tools and editors via tool interfaces and XML. For feature modeling the tool AmiEddi [AmiEddi] is used, which was extended by an XML interface.

Table 1. Code map, the „Features to Source Code“ cross reference table

Source code element / Feature	**Variable A**	**Procedure B**	**Class M**	**Type Z**
Feature 1	Impl_1.cpp / L15/	impl_5.h/ L120/		Impl_3.cpp /L15,L36/
Feature 2		impl_3.cpp /L55/	impl_1.h/L10,L70/	Impl_1.h/L45/
Feature N	Impl_2.cpp / L70/			Impl_1.h/L15,L25/

Table 2. Code map, the „Features to Source Code“ cross reference table

Source code element / Feature	**Variable A**	**Procedure B**	**Class M**	**Type Z**
Feature 1	Impl_1.cpp / L15/	impl_5.h/ L120/		Impl_3.cpp /L15,L36/
Feature 2		impl_3.cpp /L55/	impl_1.h/L10,L70/	Impl_1.h/L45/
Feature N	Impl_2.cpp / L70/			Impl_1.h/L15,L25/

5 Case study

The feature based architecture recovery method presented in this approach was applied for an industrial image processing system. The system was built in C and C++; newer parts have an object-oriented structure. It has evolved to its current state over decades. The paper examines a part of the system. Two main subsystems, were selected from the original system, called Image Provider (IP) and Image Store (IS). The Image Provider acts as a gateway between image producing hardware and image processing software. The Image Store stores images and provides them on request for further processing. Due to the development history the functionality of both subsystems is partly overlapping.

5.1 Feature model of the studied system

Initially, a system's feature model was constructed following the described approach. The Design Objectives feature model (see Fig 6) was built using various requirements documents and with support of product managers of earlier versions. The establishment of the Design Decisions feature model (Fig 7) was performed with support of the development team using various documents about architecture. At this stage, this feature model is very similar to the existing structure of the source code.

5.2 Establishment of architectural element hypotheses

This step results in a set of diagrams, which present the established architectural elements hypotheses. The diagrams are grouped according to the architectural model structure.

In our case, the selected parts of the system do not cover architectural elements for all architectural layers. Elements concerning the architectural views *code view*, *external interfaces view* and *internal structure view* were recovered. Explanations and example diagrams are to be found in the following subsections.

5.2.1 Code view

As defined in the "4 views" model, the code view describes the structure and the relations of the source code modules. The example (Figure 8) shows an assumption about the relations of the packages building the Image Store subsystem. It is referenced by the 5th and 6th row of the cross reference table (Table 2).

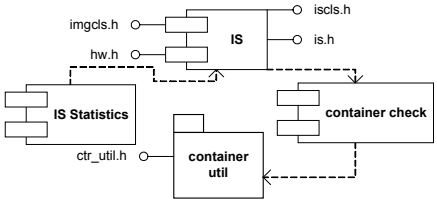

Figure 6. Architectural element hypotheses from Image Store Code view – "IS Modules"

According to the verification result this detail could become a part of the architectural model or it could be abandoned.

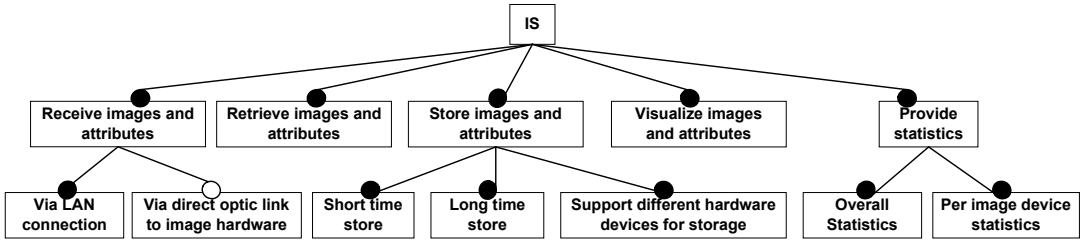

Figure 7. Design Objectives feature model diagram for the Image Store subsystem

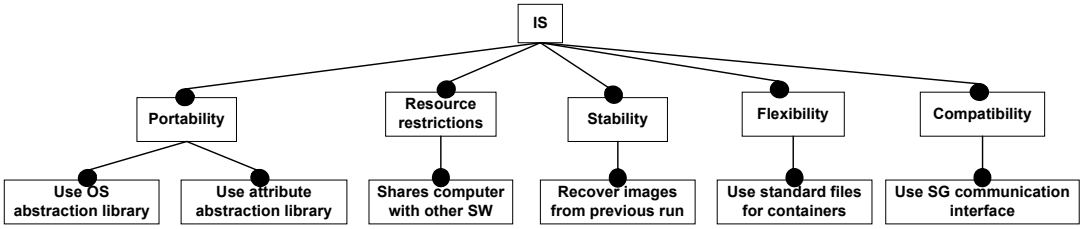

Figure 8. Design Decisions feature model diagram for the Image Store subsystem

5.2.2 External interfaces view

Figure 9 gives an example of an hypothesis about the Image Store environment and the used communication between interfaces. The connection is named "SG"; it is referenced by the last row of Table 2. More details about the data at this connection are represented by a class diagram (Fig 13).

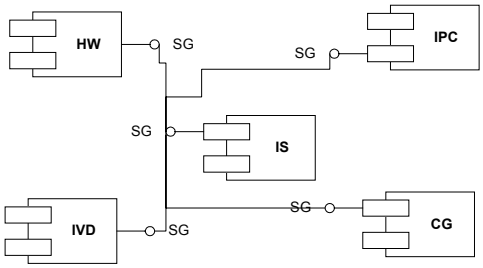

Figure 9. Architectural element hypotheses from external interfaces view – "SG Connection"

5.2.3 Internal structure view

Figures 10-13 show example diagrams of the established architectural hypotheses on modules and their dependencies. This could vary from a simple class as a more detailed piece of information (Fig 10, referenced by Table 2, first row, and Table 3, 7th row) to a class hierarchy Storable Items (Fig. 11) as detailed information to row 3 of Table 2 and row 4 of Table 3.

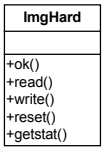

Figure 10. - Architectural element hypotheses from Internal structure view – "IMG hardware communication"

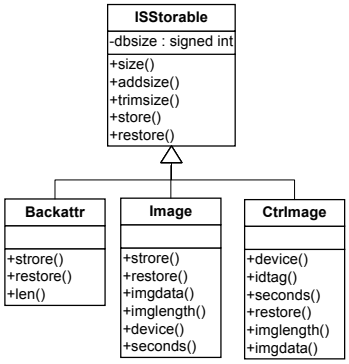

Figure 11. - Architectural element hypotheses from Internal structure view – "Storable Items"

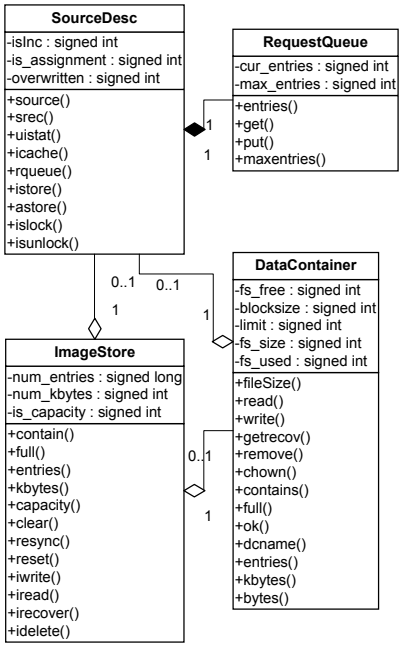

Figure 12. - Architectural element hypotheses from Internal structure view –" Image Containers"

Fig 12 shows a hypothesis concerning static relations between Image Container classes, in this case aggregation and composition relationships. All "other" information in this diagram like member variables and methods are not part of the hypothesis but have been added by a code analyzer.

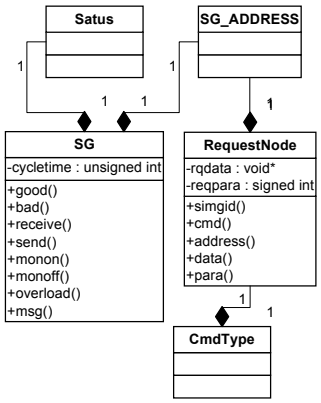

Figure 13. - Architectural element hypotheses from Internal structure view – "SG Communication"

The "SG Communication library" feature from the IS Design Decisions feature model (Figure 7) is directly related to the architecture of the system. For example, all components shown in the external interfaces diagram (Figure 9) support the SG interface, which implements this feature. Another example is visible on the class diagram from the internal structure view (Figure 11). The "ISStorable" class is a part of the implementation of the "Stores Images and attributes" feature from the Design Objectives feature model (Figure 6).

5.3 Cross referencing and hypotheses verification

In this step the architectural elements hypotheses were verified and refined. The verification is made using the "feature to architectural elements" and "feature to source code" cross-references - Tables 2 and 3.

The tables presented above illustrate some interesting cases. The "Visualize images and attributes" feature (Table 2, middle row) has no corresponding architectural elements, but was established according to a requirement.

Table 3. „Features to Architectural Elements" cross-reference for Image Server

Architecture Element / Feature	Internal Structure	External Interfaces	Code	Configuration
Receive images and attributes	IMG Hardware communication, SG Communication	HW Connection	IS Modules -IS	Configuration modules
Retrieve images and attributes	-	-	IS Modules –IS	Configuration modules
Store images and attributes	Storable Items, Image Containers	-	IS Modules –IS	Configuration modules
Visualizes images and attributes	-	-	-	-
Provide Statistics	IS Statistics	-	IS Modules –IS Statistics	
Per image device statistics			IS Modules –IS Statistics	
Use SG Communication interface	SG Communication	SG Connection	IS Modules –IS	Configuration modules

Table 4. IS Internal Structure „Architectural Elements to Features" cross-reference

Feature / Architecture Element	Receive images and attributes	Retrieve images and attributes	Store images and attributes	Provide Statistics	Use SG Communication interface	Visualizes images and attributes	Per image device statistics
IS Modules	☑	☑	☑	☑	☑		
Configuration Modules	☑	☑	☑	☑	☑		
SG Connection	☑						
Storable Items			☑				
Image Containers			☑				
SG Communication					☑		
IMG Hardware communication	☑						
Offline analyses							

This feature led to a hypothesis which could be invalid or could be an optional feature not implemented by the particular system; in this case the third case is valid. The "Per image device statistics" feature (Table 3, most right column) has no corresponding architectural elements. This implies, that a hypothesis might be missing or there is a reference to that hypothesis in the concept view. The "Offline analyses" hypothesis (Table 3, last row) has no corresponding features, which makes it invalid; therefore it is abandoned.

Verification steps are performed for all architectural hypotheses. The tools provide analyses for completeness and consistency of the established models as well as for complete verification of the hypotheses. Every detected inconsistency forces a further refinement or adjustment of architectural models.

The final result consists of a set of structured UML diagrams, presenting the recovered architecture.

6 Future work

Up to now the presented approach mainly deals with the analysis and recovery of static software architectures. The dynamic aspects of the architecture reconstruction are considered as future work. Furthermore, research is to be directed to the following areas:

- Elaboration and more strict representation of the presented feature models.
- Scaling the process for large systems by decision support by introducing metrics.
- Dealing with the cases of exiguous domain knowledge.
- Recovery of the execution view of the architecture.
- Development of a mechanism for the maintenance of the collected information during the architecture recovery process.
- Development of supporting tools for the automation of cross-reference table manipulation as a CASE tool extension .

7 Conclusions

The presented architecture recovery approach argues that the problem domain knowledge plays a significant role in the processes of program understanding and architecture recovery. Feature models act as a bridge between requirements and architecture. Feature modeling is used as a way to express that knowledge in an appropriate way. The application of the approach in industrial projects showed that feature models provide a base for program understanding, an important factor for a successful architecture recovery.

To succeed, an architecture recovery process has to consider design objectives as well as design decisions. As shown both are expressed using feature models.

The introduction of the "design objectives" and "design decisions" feature diagrams together with cross-reference tables for collecting hypotheses, enables capturing of more architectural information during the recovery process, such as specific design decisions and the related architectural elements. By describing it as hypotheses, this information is prevented from being lost.

However, the approach described is somewhat restricted due to the required manually performed operations. Tools support reverse engineers in stepwise verification and by providing consistency checks.

To be successful, an architecture recovery has to combine knowledge from different areas of software engineering. It requires an integration of knowledge derived by requirement engineering, program comprehension and reverse engineering along with architectural principles. As introduced in this paper, feature modeling provides a communication mechanism for the exchange of the implicit knowledge spread throughout all mentioned areas.

8 Acknowledgements

This work is carried out in cooperation with the Postal Automation division of the Siemens Dematic AG. We would also like to give our appreciation to Detlef Streitferdt for the constructive critique about the approach. We wish to thank Periklis Sochos for his hints for improving the paper.

9 References

[AmiEddi] AmiEddi 1.3 – Feature Model Editing tool. Available online at http://www.generative-programming.org

[Bengtsson et al. 1998] Bengtsson, P., Bosch, J.: Scenario-Based Software Architecture Reengineering, Proc. 5th International Conference on Software Reuse (ICSR5), pp.308-317, IEEE Computer Society Press, Victoria, B.C, Canada, June 1998.

[Boger et al. 2002] Boger, M.; Fragemann, P.; Sturm, T.: Refactoring Browser for UML. In: Unland, R. et al.: Proceedings Netobjectdays 2002, Erfurt, October 6-10, 2002. LNCS. Springer, 2002. available online at: http://www.netobjectdays.org/pdf/02/papers/node/0376.pdf

[Brooks et al. 1978] Brooks, R.: Using a Behavioral Theory of Program Comprehension in Software

Engineering, Proc. 3rd Int. Conf. on Software Eng. New York: IEEE, 1978

[Brooks et al. 1982] Brooks, R.: A Theoretical Analysis of the Role of Document. in the Comprehension of Computer Programs. Proc. Conf. on Human Factors in Computer Systems. New York: ACM, 1982

[Brooks et al. 1983] Brooks, R.: Towards a Theory of the Comprehension of Computer Programs. Intl. J. Man-Machine Studies 18, 6 (June 1983)

[Canfora et al. 1994] Canfora, G., De Lucia, A., di Lucca, G. Fasolino, A.: Recovering the Architectural Design for Software Comprehension, Proc. IEEE Third Workshop on Program Comprehension, Washington, DC, November 1994.

[Clayton et al. 1997] Clayton, R., Rugaber, S., Taylor, L., & Wills, L.: A Case Study of Domain-based Program Understanding. 5th Workshop on Program Comprehension, Dearborn, Michigan, 1997

[Clayton et al. 1998] Clayton, R., Rugaber, S., & Wills, L.: On the Knowledge Required to Understand a Program. The Fifth IEEE Working Conference on Reverse Engineering'98, Honolulu, Hawaii, October 1998

[Corbi et al. 1989] Corbi, T. A.: Program Understanding: Challenge for the 1990's. IBM Systems J. 28, 1989

[Czarnecki et al. 2000] Czarnecki, K., Eisenecker, U.W.: Generative Programming. Addison Wesley, Reading, MA, 2000.

[Finnigan et al. 1997] Finnigan, P., Holt, R., Kalas, I., Kerr, S., Kontogiannis, K., Müller, H., Mylopoulos, J., Perelgut, S., Stanley, M., Wong, K.,.: The Software Bookshelf, IBM Systems Journal, Vol. 36, No. 4, pp. 564-593, November 1997.

[Garlan et al. 2000] Garlan, D., Software Architecture: a Roadmap, The Future of Software Engineering, ACM Press, pp.91-101, 2000

[Hofmeister et al. 2000] Hofmeister, C., Nord, R., Soni, D.: Applied Software Architecture. Addison Wesley, 2000.

[Kang et al. 1990] Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1990.

[Kang et al. 1998] Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E., Huh. M.: FORM: A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering, 5:143--168, 1998

[Kazman et al. 1997] Kazman, R., Carrière, S.: Playing Detective: Reconstructing Software Architecture from Available Evidence. Software Engineering Institute, Carnegie Mellon University, CMU/SEI-97-TR-010/ESC-TR-97-010

[Kazman et al. 1998a] Kazman, R., Carriere, S. J.: View Extraction and View Fusion in Architectural Understanding, Proc. 5th International Conference on Software Reuse (ICSR5), pp.290-299, IEEE Computer Society Press, Victoria, B.C, Canada, June 1998.

[Kazman et al. 1998b] Kazman, R., Klein, M., Barbacci, M., Lipson, H., Longstaff, T., Carriere, S. J.: The Architecture Tradeoff Analysis Method, Proc. Fourth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS98), pp.68-78, Montery, USA, August 1998.

[Kruchten 1995] Kruchten, P. B.: The 4+1 View Model of Architecture. IEEE Software, 12(6): 42-50, 1995

[Kuusela et al. 2000] Kuusela, J., Savolainen, J.: Requirements Engineering for Product Families. ICSE 2000, Proc. 22nd Int. Conf. on Software Eng., Limerick Ireland. ACM, 2000

[Letovsky et al. 1986] Letovsky, S., Soloway E.: Delocalized Plans and Program Comprehension. IEEE Software 3, 3 (May 1986)

[Poseidon] Poseidon for UML - CASE Tool. Gentleware AG. http://www.gentleware.de/products/

[Rajlich et al. 1994] Rajlich, V., J. Doran, Gudla R.: Layered Explanations of Software: A Methodology for Program Comprehension, Third Workshop on Program Comprehension, WPC'93, Washington, D.C., pp. 46-52, Nov. 1994

[Riebisch et al. 2002] Riebisch, M., Böllert, K., Streitferdt, D., Philippow, I.: Extending Feature Diagrams with UML Multiplicities. 6th Conference on Integrated Design & Process Technology, Pasadena, California, USA. June 23 – 30, 2002

[Riva et al. 2002] Riva, C., Rodriguez, J. V.: Combining Static and Dynamic Views for architecture reconstruction. Proc. of the Sixth European Conference on Software Maintenance and Reengineering, Budapest, March 2002

[Rugaber et al. 2000] Rugaber, S.: The use of domain knowledge in program understanding. Annals of Software Engineering, 2000

[Storey et al. 1997] Storey, M. D., Fracchia F.D., Müller H. A.: Rigi: A Visualization Environment for Reverse Engineering**.** Proceedings of the International Conference on Software Engineering (ICSE'97), Boston, U.S.A., May 17-23, 1997.

[Tilley et al. 1998] Tilley, S. R.: A Reverse-Engineering Environment Framework. Software Engineering Institute, Carnegie Mellon University. (CMU/SEI-98-TR-005), 1998