E-CARES Research Project:
Understanding Complex Legacy Telecommunication Systems

André Marburger
Aachen University of Technology
Department of Computer Science III
Ahornstr. 55, 52074 Aachen, Germany
marand@i3.informatik.rwth-aachen.de

Dominikus Herzberg
Ericsson Eurolab Deutschland GmbH
CNM – Node Product Unit MSC
Ericsson Allee 1, 52134 Herzogenrath, Germany
Dominikus.Herzberg@eed.ericsson.se

Abstract

There are many reasons for reverse engineering or re-engineering legacy systems. To date, many approaches concerning re-engineering of legacy systems have been made. The majority of these approaches are dealing with systems in the field of business applications. This paper describes the work performed for the E-CARES project so far. This project is concerned with understanding and re-structuring complex legacy telecommunication systems. In contrast to business applications embedded systems, e.g. telecommunication systems, have additional requirements regarding fault tolerance, reliability, availability, and response time. We found that these requirements have a significant impact on the software part of an embedded system. It has different characteristics concerning structuring, inter-program communication, etc. Therefore, an approach is presented that includes usage of “dynamic” information, multi-level abstraction/visualization, and user interaction to improve the understanding of telecommunication systems.

1. Introduction

The importance of embedded systems for our daily life is rapidly increasing. Although, more or less unnoticed, they fulfill the task of controlling the “behavior” of technical systems ranging from drinks’ dispensers to big industrial plants. Embedded real-time systems play a special role. Besides the typical requirements of embedded systems concerning e.g. reliability or fault tolerance, embedded real-time systems also have to fulfill requirements concerning availability and response time. Another property often found in embedded real-time systems is concurrency. One important field of application for embedded real-time systems is in the telecommunications industry. The complexity of these systems is “exploding” while at the same time the software part is becoming more and more important.

In the literature only little work on reverse engineering and re-engineering of embedded systems has been described. Work in this field of activity has concentrated on sequential, untimed systems; mainly business applications. The approaches found there deal e.g. with decomposing monolithic systems (REforDI [6]), decoupling user interface/presentation, application logic, and data handling/database management (RECAST [1]), or with identifying reusable components (COBOL/SRE [15]). In general, different kinds of object/component identification methods are used to fulfill these tasks. The majority of these methods are based on information extracted from analyzing a system’s source code.

In contrast to business applications, telecommunication systems only have a very simple user interface or no user interface at all. But, they combine different applications like call handling, administration of subscriber data, etc. in a single system. Moreover, a telecommunication system already is a distributed system with respect to the basic ideas described e.g. in [6]. The problem is that some of the components of these large distributed systems are becoming too complex to understand and maintain themselves.

Furthermore, we found that the analysis of static relationships of different components of a system’s software is not sufficient with respect to telecommunication systems. In these systems the executable units are self-contained regarding data and execution. At run-time they behave like independent processes. Each unit can have an arbitrary number of incarnations. Many services provided by telecom systems are realized by re-connecting certain units/processes appropriately at runtime. That is, there are many more possible paths of execution through a telecom system than in sequential systems. We believe that this is a consequence of requirements regarding fault tolerance, reliability, availability, response time, and concurrency.
Therefore, to understand a system’s dynamic behavior process incarnation, process handling and process interaction at runtime has to be taken into account explicitly.

The approach presented in this paper combines “classical” reverse engineering with the explicit use of information about the dynamics of an analyzed system. We will focus on the understanding of a special class of legacy embedded systems, telecommunication systems. Furthermore, similar to the Rigi concept [14] our approach comprises multi-level abstraction/visualization and user interaction to improve the understanding of telecommunication systems.

There are similarities in our approach to the one presented by Krikhaar in [8]. Both methods have a theoretical mathematical basis. Krikhaar’s approach is based on relation (partition) algebra while ours is based on graphs and graph theory [17][18]. However, both approaches can be transformed into each other. Furthermore, in both approaches we have identified almost the same sources of information. The most obvious difference is that our approach considers information on a legacy system’s dynamic behavior explicitly. We also preserve the relationship of a piece of information to the corresponding parts of the information’s source. Additionally, our work addresses methodology as well as integrated, interactive, and visualizing tools to support software engineers.

In section 2, we introduce the domain of telecommunication system. Afterwards, we give a short overview of our basic approach. The different aspects of this approach are discussed in the following sections. In section 3, we focus on the extraction of information from source code (code analysis). As the regarded legacy software system is written in a non-standard programming language a short introduction in this language is given as well. The simplification of the information basis that is the result from code analysis is discussed in section 4. Section 5 addresses the need of a possibility to inspect a legacy system on various levels of detail and from different points of view. Our re-engineering method supports the definition of views each visualizing only a certain aspect of an analyzed system. In contrast to other re-engineering methods, our approach uses information on a legacy system’s dynamic behavior. We describe this aspect of our work in section 6. In section 7, we give an overview of further work planned for the next stage of our work. Finally, we summarize our findings in section 8.

2. Overview

The E-CARES¹ research cooperation between Ericsson Eurolab Deutschland GmbH (EED) and the Department of Computer Science III, RWTH Aachen, has been established to improve the re-engineering of complex legacy telecommunication systems. It aims at developing methods, concepts, and tools to support the processes of understanding and restructuring of this special class of embedded systems. The subject of study is Ericsson’s Mobile-service Switching Center (MSC) for GSM-networks called AXE10. Note, GSM (Global System for Mobile Communication) is the current standard for mobile communication systems.

The MSCs are the “heart” of a GSM-Netowrk as shown in the simplified sketch in figure 1. A MSC provides the “services” a person can request by using a mobile phone, e.g. a simple phone call, a phone conference, or a data call, as well as additional infrastructure like authentication. Each MSC is supported by several Base Station Controllers (BSC), each of which controls a set of Base Station Transceivers (BTS). The interconnection of MSCs and the connection to other networks (e.g. public switched telecommunication networks) is provided by gateway MSCs (GMSC). In fact, the MSC is the most complex part of a GSM-network. The AXE10 software system comprises approximately 10 million lines of code spread over circa 1,000 executable units.

The basic E-CARES approach is outlined in figure 2. We obtained three different sources of information. The first one is the source code of the system. It is considered to be the core information as well as the most reliable one. Via code analysis (parsing) a structure document is generated, which is stored as an attributed graph. This structure document is the basis for any other investigation. It is extended stepwise with additional information.

The second source of information is the “method of use”. In Ericsson, there are a lot of design rules defining how to implement certain functionality. These design rules can be considered as a set of micro-patterns. Furthermore, the programmers themselves have “defined” additional rules. Additional domain knowledge can be obtained by interviewing Ericsson’s experts, a group of senior programmers

¹The acronym E-CARES stands for Ericsson Communication ArChitecture for Embedded Systems.
and designers. On the one hand, domain knowledge is used to enrich the structure document. On the other, graph queries and graph transformations are derived. These graph queries and transformations are implemented in our reverse engineering tool to help software architects to inspect a legacy system’s structure and to rebuild its software architecture.

There are two possibilities to obtain “dynamic information”, using an emulator or querying a running AXE10. In both cases, the result is a list of events plus additional information in a temporal order. In the latter case, querying the running system, timing problems occur when the repolled information is too complex. Hence, the emulator is the number one choice to gather information on the system’s dynamic behavior. This “dynamic information” is the third source of information. Traffic cases can be identified, fed into the reverse engineering tool, and simulated there by traversing the structure document. This helps software architects to identify components of a system that take part in a certain traffic case. Therefore, “dynamic information” allows us to analyze a system regarding different run-time conditions.

3. Code Analysis

In the first stage of our work, we concentrated with code analysis (terminology according to [4] and [13]). As the software system running on Ericsson’s AXE10 switching center is written in a non-standard programming language, a short introduction into this language is given in section 3.1. Afterwards the code analysis phase and the creation of a structure document is described in section 3.2.

3.1. Introduction into PLEX

The application part of the AXE10 software is implemented in a non-standard programming language called PLEX (Programming Language for EXchanges), which was developed at Ericsson. PLEX is an asynchronous concurrent real-time language designed for programming of telecommunication systems. This programming language has a “signaling paradigm” as the top execution level. That is, only events can trigger code execution. Events are programmed as signals.

Program systems written in PLEX must be run in an operation system whose main task is the signal handling. Signals and signal handling are very important issues because signals are the overall concept of information interchange in the heterogenous architecture of the MSC. A MSC consists of a mixture of hardware and software units all using signals as the basis for communication among each other.

Using the PLEX language a software system consists of one or more programs. A program is a quantity of code comprising several “functions” (executable parts of a program), which are compiled together and have data in common. Programs have data encapsulation, that is, a program’s data cannot be accessed by other programs. Internally a program is subdivided into a declare sector (declaration of variables etc.), a parameter sector, a program sector (program logic), a data sector (initialization data), and a signal sector (definition of (local) signals).

The program sector is unstructured to a certain extent. It can be subdivided using goto-connected labeled statement sequences and subroutines. Subroutines are well-formed structural units but meaningful statement sequences are hard to identify. This is unsatisfactory in regard to a fine grained analysis of program internals. We know from expert advice, that there are usually different “functions” implemented in the program sector of a single program, but they are not clearly identifiable from the code’s structure. Considering some of Ericsson’s design rules, we decided to define labeled statement sequences and signal entry points as additional structural units for subdividing a program sector. Simply speaking, a labeled statement sequence is defined as the amount of source code between two labels. Most signal entry points (SEP) are located at the very beginning of the program sector to form a kind of interface. Subsequent to an “interface” SEP there is at least a goto-statement. Therefore, a SEP consists at first of the corresponding statement itself. If it is located in the virtual interface part, the amount of source code subsequent to a SEP until the next one is also part of the corresponding structural unit.

The only way to communicate/interact between programs is via signals. There are different signal types to perform asynchronous and synchronous communication,
single signals and combined signals respectively. Single signals can be sent in a buffered mode or directly. That is, a signal is either handled by a multi-level scheduling mechanism or it is directly sent to its destination. The scheduler consists of a set of queues (job buffers) with different execution priorities; this allows the system to prioritize and schedule different kinds of “activities”. Whenever a signal is received by a program, this signal is the entry point to code execution in the program. That is, an incoming signal triggers the execution of a certain “function” within a program; the program reacts to an event. Each of these “functions” consists of a signal entry point, a number of GOTO-connected labeled statement sequences, and at least one terminating exit statement. We found that it is very difficult to decouple these “functions”, because, unfortunately, some labeled statement sequences are used by different “functions”.

3.2. Generating Structure Graphs

To generate a basis for further investigations that is easier to handle than plain source code, we decided to store the information obtained from the source code as an attributed, directed graph. Graphs are a very flexible and well understood data structure that allow complex analysis using comparatively simple graph tests and graph transformations. We will refer to this graph as a structure graph from here on. A simple example of a structure graph is shown in figure 3.

![Structure Graph](image)

**Figure 3. A simple structure graph**

A node of the structure graph represents either an “atomic” unit of a program or a structural unit like a subsystem. An atomic unit identifies a portion of source code that is closed under execution. Some of these basic units represent a single statement (label, exit), others a sequence of statements, e.g. signal entry points, atomic statement sequences, and subroutines.

Relationships between basic units/structural units are modeled as edges of the structure graph. There are edges representing structural relationships (e.g. contains, has), control flow relationships (e.g. calls, goto, signal), or data flow relationships (e.g. reads, manipulates, signal). Besides signal edges, which are part of control flow and data flow, neither data flow edges nor nodes representing data elements or data structures are produced by the parser yet. It is planned to extend the parser/analyzer in the next stage of our work.

The combination of information on structure, control flow, and data flow of an analyzed system within one graph allows us to specify complex graph queries using all three information at once very comfortably. This is a major benefit for the development of a powerful reverse engineering toolkit.

Nodes and edges of the structure graph are attributed. These attributes are used to store different kinds of additional information (e.g. signal or label name). Regarding re-structuring the most important information stored in attributes is the information which node/edge corresponds to which part of the source code. That is, for every node and edge of the graph that is directly related to a certain part of the source code, the filename, starting line, and end-line are stored.

Figure 4 describes the tools and documents involved in the E-CARES reverse engineering approach. The first step towards creation of a structure graph is the parsing of the source code. A combined parser/analyzer reads a source code file and produces a Python [12] script that consists of a sequence of commands. Each command represents a graph transformation to insert nodes and edges into a structure graph. Hence, every script file contains a textual description of the structure graph of a single program.

This script is the input to a reverse engineering tool. It processes the script and extends the actual structure graph accordingly. Unfortunately, we found that it is not possible to determine the receiver of a signal sent by a program while parsing the corresponding source code. The linking of two programs via a signal is performed at runtime. Therefore, as long as the corresponding information is missing, the insertion of signal-edges into a structure graph is performed in a consolidation process subsequent to the main insertion process. For example, the signal edge labeled signal $H$ in figure 3 is inserted by this consolidation process. Furthermore, because of ambiguity of some signals every signal-edge within the structure graph has to be considered as a potential communication between two programs.

The resulting structure graph is the basis for further investigations. It is easier to handle than plain source code. Furthermore, we can use the expressiveness of graph
Figure 4. Tools and documents in E-CARES project

queries and graph transformations for the development of a powerful reverse engineering toolkit. Unfortunately, the structure graph itself is too detailed to be of any benefit to somebody who wants to understand a telecommunication system. Considering the whole AXE10 software system, it would contain some hundred of thousands of nodes and edges. Moreover, it still contains information that is only needed to create the structure graph. Once the creation is completed, these additional nodes and edges become needless. We tackled this problem on two different levels; (1) by simplifying the graph structure and, (2) by defining a set of views to adjust the detail level on user level. The simplification of the structure graph reduces the amount of information in the graph. Whereas, views are a means to adjust the displayed amount and type of information in a reverse engineering toolkit.

4. Simplifying the Structure Graph

The first step to reduce the complexity of the structure graph is to identify needless information. The second step is to define suitable graph transformations to delete redundant graph elements or to reorganize subgraphs. Changes to the structure graph must preserve the connection to the corresponding parts of the source code. Otherwise, we will lose the ability of transforming the source code in a later restructuring phase. That is, if a node/edge directly represents a certain portion of source code this has to be a one-to-one relation; no two elements of the basic structure graph are allowed to reference the same part of the source code.

At this point, we involved the expert group supporting the E-CARES project. This group consists of some senior system designers from Ericsson. An analysis of a structure graph representing a small part of the AXE10 software system and the subsequent discussion lead to the following results:

1. Nodes representing a label can be deleted if the corresponding labeled statement sequence is named accordingly and the incoming goto-edges are connected to this statement sequence.

2. Nodes representing a label and goto-edges that are connected to a subroutine-node can be deleted. According to a PLEX design rule a goto passing subroutine borders is not allowed. Hence, these label-nodes and goto-edges are needless implementation details.

3. Goto-edges connecting a node representing a statement sequence with its label-node can be deleted. These edges just represent some kind of malformed loop implementation or conditional clause.

4. Statement sequences that are terminated by an unconditional exit statement at their very end can be transformed to a new node representing a terminating statement sequence.

5. Statement sequences referenced by only one other statement sequence can be combined to form a bigger statement sequence if the source code references are stored as an ordered list (execution order) using an attribute.

6. A goto-connected chain of labeled statement sequences where each node is only reachable starting with the same signal entry point, can be collapsed to form a new node representing a “function”.

Using these simplification rules the example graph in figure 3 could be reduced to the graph in figure 5. The function labeled $G,X$ is created by applying rules 1, 4, and 6; function $H,Y,Z$ is created by applying rules 1, 4, 5, and 6. In this optimal case, the graph is reduced by 58 percent regarding the number of nodes and edges. We still lack a complex case study but in general we expect an average reduction of the structure graph of 30 percent. The implementation of some of the simplification rules mentioned above within the analyzer will, furthermore, lead to a more efficient reverse engineering process.

The structure graph obtained by applying the simplification rules is still too complex to be displayed at user level simultaneously. For the purpose of visualization, we have defined a set of views, which are sufficient to provide a first facility to switch to different levels of detail.
5. Defining Views

The possibility to inspect a system’s structure on different levels of detail is of enormous importance. Furthermore, to inspect a legacy system from different points of view, a software architect requires a facility to adjust the amount and type of information displayed accordingly. Therefore, we at first decided to implement three detail levels that are inspired by the coarse grained structuring of a PLEX software system. The most coarse grained level is the subsystem level. Each node visualized on this level represents a subsystem. Edges between two nodes express that there is communication between those subsystems. A subsystem consists of a number of programs. Thus, the next level is the program level. On program level, each node represents a program of the system. Edges have the same meaning as in the subsystem view. The most detailed level of visualization is the base level. On base level, the structure graph itself is displayed.

The communication edges needed on subsystem and on program level to depict inter-node communication cannot be obtained directly by parsing the source code. Therefore, a set of consolidation functions was implemented that, among others, automatically insert a communication edge into the program view if there is at least one signal sent from program A to program B. If A and B belong to different subsystems a communication edge between the corresponding subsystem nodes is inserted into the subsystem view as well.

The different views are implemented using a filtering technique. When a user opens a view in the reverse engineering tool the corresponding filter is invoked. Each filter is a specialized static graph unparsar implemented to realize a certain view. It queries the underlying graph storage according to its filtering rules. The returning information from the graph storage is then delivered to the view window for visualization purpose. This procedure is sketched in figure 6.

We found that switching between different predefined views is still not sufficient. For example, considering program level there are still more than one thousand nodes plus corresponding communication edges displayed simultaneously. To overcome this deficit we have combined the three views to allow interactive refinement of a visualized portion of the structure graph.

The default view is the subsystem view. To get more details, the user simply tags a subsystem node(s) of interest and activates a step-into function. Another window will be opened on program level. This window visualizes the internals of the tagged subsystem(s); it simply displays the corresponding programs and communication edges. Tagging at least one node in this program view and again activating a step-into function, the internals of the tagged node(s) will be displayed in a third window.

At this point, we sought expert assistance to identify further improvements concerning views. It turned out that interactive adjustment of the displayed amount of nodes and edges is useful to be able to concentrate on a restricted issue. Hence, the view handling mechanism has been extended with interactive filtering facilities. On the one hand, the user is now able to define customized filters that affect all views. On the other hand, there are functions that allow fine grained adjustment of the displayed information by hiding tagged nodes and edges on demand.

The facilities to inspect the structure of a PLEX software system on different levels of detail are very powerful. They support a software architect in understanding a complex legacy software system by providing predefined views, customized filters, and on-demand adjustment of views. However, case studies using only a small amount of the AXE10 system demonstrated, that static information like the structure graph is not sufficient to understand a telecommunications system.
6. Dynamic Information

Telecommunication systems are highly dynamic systems. For example, switching systems handle thousands of calls at the same time. They have to adjust themselves to different situations very fast, that is, they have to be very flexible. We found that these requirements have an impact on structure and implementation of this kind of embedded systems. The systems functionality is spread over a large number of small units, programs in the case of the AXE10. Each unit realizes a special part of the functionality. Accordingly, a service is realized by connecting the programs needed at runtime. In many cases, a single program is used to realize different services. In the program view this is indicated by highly interconnected graphs (cf. figure 7).

Figure 7. Highly interconnected “call” graph

To understand a telecommunication system and to derive a system’s architecture, static information received by code analysis and additional information extracted from design documents etc. are not sufficient. To recreate a reasonable architecture (design recovery phase) further information regarding the dynamic behavior of a system is needed to identify parts realizing certain functionality. This contrasts to reverse engineering and re-engineering of business applications as described e.g. in [6], [2], and [19].

In the context of the E-CARES project, there are two possibilities to get runtime information; tracing a running AXE10 or tracing the emulator machine. Both systems are running the same (PLEX) software system. In the case of an operating AXE10, the complexity of tracing is reduced to avoid the system becoming unstable. Tracing jobs have the highest priority. If there are too many of them, the system gets into timing problems. There is no such problem for the emulator. The emulator works in a virtual time mode that allows complex tracings. Consequently, we decided to use tracings of the emulator to produce information on the dynamic behavior of Ericsson’s switching system.

Figure 8. Obtaining “dynamic information”

Tracing of the emulator is initiated by defining which kind of information and which part of the system is of interest. Once the machine is started, different call scenarios can be stimulated using the testing equipment connected to the emulator. The tracing information is continuously written to a text file (see figure 8). Currently, the following information is obtained using the emulator:

- signal (name, identification, type)
- sender of a signal (program)
- receiver of a signal (program)
- data transferred with a signal
- signal priority

This information is sufficient to “simulate” the dynamic behavior of the AXE10 in a first case study. Each trace file contains the runtime information obtained from the emulator for a special call scenario. The trace files are well-structured and easy to process automatically using some simple text processing functions or a small parser. We expect the tracing information processed to become more complex by and by. Hence, we decided to implement a small parser using jlex/jay [9][10] to be more flexible. The parsing is triggered by the reverse engineering prototype. That is, a trace file is parsed stepwise on demand.
A “simulation” of the dynamic behavior starts with the selection of a call scenario (trace file). The current view\(^2\) is cleared afterwards. Depending on the current view (subsystem view or program view) the starting node – subsystem or program – is shown and highlighted. In this first version the user is now able to stepwisely “execute” the selected trace. The amount of information extracted from a trace file in order to simulate the next step depends on the active view:

- In the subsystem view, the current trace file is parsed until the first signal is found that leaves the currently active subsystem (currently highlighted node in subsystem view). The receiving subsystem is shown and highlighted and the communication edge between sending and receiving subsystem as well.
- In the program view, the current trace file is parsed until the next signal information is found. The receiving program is shown and highlighted as well as the communication edge between the sending and receiving program.

When both a subsystem view and a program view are open, tracing is “simulated” in both windows simultaneously. Accordingly, the step range is determined by the view in which the next step is activated.

Sometimes it is necessary to have an idea which subunits of a program are involved in the processing of an incoming signal. Unfortunately, there is no corresponding information available from tracing a running switching system. Therefore, we implemented a heuristic that tries to work out which subunits of a program are visited on reception of a certain signal and displays the corresponding amount of the structure graph in a program details view.

This heuristic is implemented in terms of graph tests. It simply identifies all possible paths between a specified signal entry point and all reachable outgoing signals. An outgoing signal is defined to be reachable if there is a path of control flow edges in the structure graph starting with the specificed signal entry point and ending with a corresponding signal edge.

After a simulation is finished, the displayed graphs represent a kind of simple multi-level collaboration diagram with respect to UML \[16\]. If the order of edges is preserved by labeling them appropriately, the result would be a full collaboration diagram. In addition, every trace file can easily be transformed into a sequence diagram.

The tracing “simulation” facility described in this section is a first step towards integrating information on the dynamics of an analyzed system. Furthermore, it is useful to reduce the amount of information displayed on the user level. In combination with the view concept described in section 5, it enables the user to concentrate on certain aspects of an analyzed system and to adjust the view’s focus accordingly. Case studies and workshops with experts are planned to evaluate this concept in practice.

7. Further Work

Currently, we are working on the consolidation of the interactive “call scenario simulation”. During the next stage of our work, we intend to concentrate on the identification of areas of functionality. To achieve this, we intend to apply some of the approaches described in e.g. \[3\], \[5\], \[7\], and \[11\]. In addition, we will also investigate further methods to analyze the dynamics of telecommunication systems. We believe that there is a possibility of identifying program link chains that realize a certain functionality at runtime by analyzing special constructs in the source code in combination with certain design rules.

In parallel, we intend to develop a suitable notation for architectures in the telecommunications domain. The concepts of this (graphical) architecture description language are supposed to inclose entities and relationships a telecom engineer needs to model a telecommunication system. Ericsson’s experts believe that modeling languages, which are currently available are not sufficiently reflecting their needs. The definition of an architecture description language has some impact on our reverse engineering approach. We have to work out abstraction techniques that allow us to identify the defined architectural entities in source code.

In addition, we are planning to improve the process of system understanding and restructuring by integrating software metrics into our approach. Currently, a sub-project running in Finland is concerned with software metrics.

8. Conclusion

The E-CARES research project is concerned with the understanding and restructuring of complex legacy telecommunication systems. The subject of study is Ericsson’s AXE10, a mobile service switching center for GSM networks. So far, we have concentrated on the understanding aspects. The work started with a code analysis phase. The resulting structure graph is the basis for further investigations. Although the structure graph is more abstract than plain code, it is still too complex to be of any use when visualized at user level simultaneously.

The second phase dealt with the definition of views that allow the inspection of a systems structure on different levels of abstraction and from different points of view. We decided that a user of a reverse engineering tool should be able to interactively adjust the detail level and the information displayed. Therefore, the reverse engineering methods to analyze the dynamics of telecommunication systems. We believe that there is a possibility of identifying program link chains that realize a certain functionality at runtime by analyzing special constructs in the source code in combination with certain design rules.
prototype implemented so far, contains three predefined views: subsystem view (standard view), program view, and program details’ view. Furthermore, the information displayed in a view can be adjusted on demand. Views support a software engineer in understanding static and structural properties of an analyzed system. These properties include a system’s decomposition in different types of components as well as inter-component control flow and data flow.

In the case of a telecommunication system, the information on the structure of an analyzed system is still not sufficient to understand the system. Unlike the analysis of business application, additional information on the runtime behavior and the runtime communication structure of a telecommunication system is needed. In the current third phase, we extended our approach accordingly and implemented a facility to inspect a system with respect to a certain call scenario. An AXE10-emulator is used to gather information on a system’s runtime behavior regarding certain traffic cases. Afterwards, this information is used to simulate the traffic cases within the reverse engineering environment. This facility enables a software architect to identify components involved in the realization of a certain traffic case at runtime. Furthermore, he can search for execution patterns that occur regularly. Execution patterns on the other hand can help to identify high level concepts within a system.

In general, we can conclude that the characteristics of a telecommunication system impact the structuring and implementation of these kinds of systems. Domain knowledge acquisition involving the interviewing of experts is very important to define appropriate reverse engineering methods. Graphs, graph queries, and graph transformations have turned out to be a powerful means to reverse engineer complex legacy telecommunication systems. However, our work is still in an early phase and we found that the understanding of telecommunication systems is a difficult task. A more complex case study is needed to prove our concepts. Additionally, we are planning to integrate abstraction techniques to support coarse grained redesign.

References