Tools for Understanding the Behavior of Telecommunication Systems

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

Bernhard Westfechtel
Aachen University of Technology
Department of Computer Science III
Ahornstraße 55, 52074 Aachen, Germany
westfechtel@cs.rwth-aachen.de

Abstract

Many methods and tools for the reengineering of software systems have been developed so far. However, the domain-specific requirements of telecommunication systems have not been addressed sufficiently. These systems are designed in a process- rather than in a data-centered way. Furthermore, analyzing and visualizing dynamic behavior is a key to system understanding. In this paper, we report on tools for the reengineering of telecommunication systems which we have developed in close cooperation with an industrial partner. These tools are based on a variety of techniques for understanding behavior such as visualization of link chains, recovery of state diagrams from the source code, and visualization of traces by different kinds of diagrams. Tool support has been developed step by step in response to the requirements and questions stated by telecommunication experts at Ericsson Eurolab Germany.

1. Introduction

Reengineering of large and complex software systems has proved a difficult task. According to the "horseshoe model of reengineering" [4, 10], reengineering is divided into three phases. *Reverse engineering* is concerned with step-wise abstraction from the source code and system comprehension. In the *restructuring* phase, changes are performed on different levels of abstraction. Finally, *forward engineering* introduces new parts of the system (from the requirements down to the source code level).

For reengineering, many methods and tools have been developed. To a large extent, however, previous work has been *data-centered* since it focuses on structuring the data maintained by an application. In particular, numerous approaches have addressed the migration of legacy business applications — written, e.g., in COBOL — to an object-based or object-oriented architecture [5, 15, 23]. This task requires the grouping of data and functions into classes with corresponding attributes and methods. Another stream of research has dealt with programming languages such as C++ and Java which already provide language support for object-oriented programming [17].

Reengineering of *process-centered* applications has been addressed less extensively so far [25]. For example, a telecommunication system is composed of a set of distributed communicating processes which are instantiated dynamically for handling calls requested by its users. Such a system is designed in terms of services provided by entities which communicate according to protocols. Understanding a telecommunication system requires the recovery of this conceptual world from the actual source code and other sources of information.

The *E-CARES*[1] research cooperation between Ericsson Eurolab Deutschland GmbH (EED) and Department of Computer Science III, Aachen University of Technology, has been established to develop methods, concepts, and tools for the reengineering of complex legacy telecommunication systems. E-CARES has been driven strongly by the requirements of software engineers who are involved in the design and implementation of GSM networks for mobile communication. The subject of study is Ericsson's Mobile-service Switching Center (MSC) for GSM networks called AXE10. The AXE10 software system comprises approximately 10 million lines of code spread over about 1,000 executable units, and has an estimated lifetime of about 40 years. Thus, there is an urgend need for tool support to improve program evolution and maintenance.

In E-CARES, a prototypical environment — called *E-CARES prototype* [14] — is being developed which addresses the reengineering of telecommunication systems. So far, tool support covers only reverse engineering, i.e., the first phase of reengineering. While *structural analysis* is covered as well, we put strong emphasis on *behavioral analysis* since the structure alone is not very expressive in the case of a telecommunication system.

[1] *E*ricsson *C*ommunication *AR*chitecture for *E*mbedded *S*ystems [13]

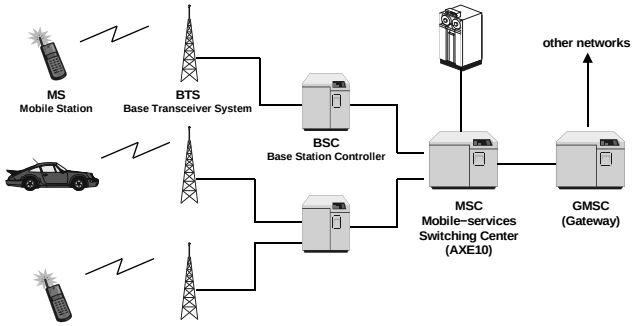

Figure 1. Simplified sketch of a GSM network

The rest of this paper is structured as follows: Section 2 motivates our research and explains its background. Section 3 gives an overview of the E-CARES prototype. Section 4 briefly describes structural analysis. Section 5, the core part of this paper, presents a variety of techniques which support the understanding of behavior. Section 6 introduces metrics — for both structure and behavior — which further improve the understanding of telecommunication systems. Section 7 discusses related work, and Section 8 concludes this paper.

2. Background

The *mobile-service switching centers* are the heart of a GSM network (Figure 1). An 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. An MSC consists of a mixture of hardware (e.g., switching boards) and software units. In our research we focus on the software part of this embedded system.

Figure 2 illustrates how a *mobile originating call* is handled in the MSC. The figure displays logical rather than physical components according to the GSM standard; different logical components may be mapped onto the same physical component. The mobile originating MSC (MSC-MO) for the A side (1) passes an initial address message (IAM) to a GMSC which (2) sends a request for routing information to the home location register (HLR). The HLR looks up the mobile terminating MSC (MSC-MTE) and (3) sends a request for the roaming number. The MSC-MTE

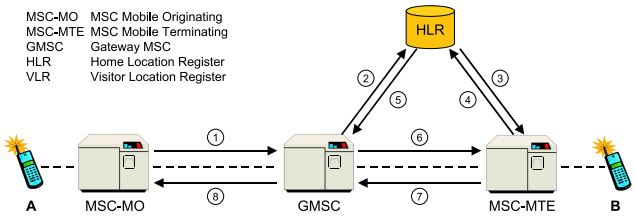

Figure 2. Mobile originating call

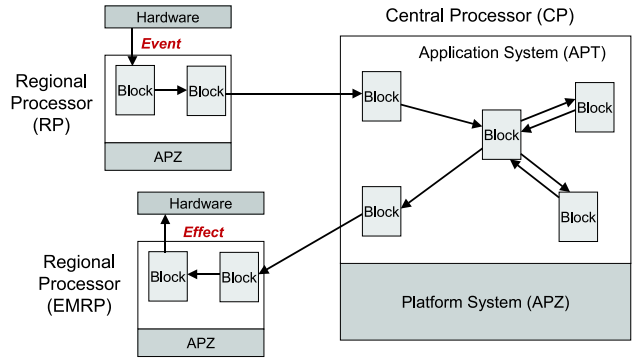

Figure 3. System architecture

assigns a roaming number to be used for addressing during the call and stores it in its visitor location register (VLR, not shown). Then, it (4) passes the roaming number back to the HLR which (5) sends the requested routing information to the GMSC. After that, the GMSC (6) sends a call request to the MSC-MTE. The MSC (7) returns an address complete message (ACM) which (8) is forwarded to the MSC-MO. Now, user data may be transferred between A and B.

Ericsson's implementation of the MSC is called *AXE10*, whose *system architecture* is shown in Figure 3. Each MSC has a central processor which is connected to a set of regional processors for controlling various hardware devices by sensors and actors. The core of the processing is performed on the central processor. The AXE10 software is composed of blocks which constitute units of functionality and communicate by exchanging signals (see below). On each processor, a runtime system (called APZ) is installed which controls the execution of all blocks executing on this processor. An event raised by some hardware device is passed from the regional processor to the block handling this event on the central processor. In response to the event, an effect may be triggered on another hardware device.

The executable units of the AXE10 software system are implemented in Ericsson's in-house programming language *PLEX* (*P*rogramming *L*anguage for *EX*changes), which was developed in about 1970 and has been extended since then. PLEX is an asynchronous concurrent real-time language designed for programming of telecommunication systems.

The programming language has a *signaling paradigm* as the top execution level. That is, only events can trigger code execution. Events are programmed as signals.

A PLEX program is composed of a set of *blocks* which are compiled independently. Each block consists of a number of sectors for data declarations, program statements, etc. (see Figure 8, which will be explained more thoroughly in Section 5). Although PLEX does not support any further structuring within these sectors, we have identified some additional structuring through coding conventions in the program sector. At the beginning of the program sector all signal reception statements (*signal entries*) of a block are coded. After these signal entry points, a number of *labeled statement sequences* follows. The bottom part of the program sector consists of *subroutines*.

The control flow inside a program sector is provided by `goto` and `call` statements. The `goto` statement is used to jump to a label of a labeled statement sequence. Subroutines are accessed by means of `call` statements. Both `goto` and `call` statements are parameter-less. That is, they affect only the control flow, but not the data flow.

Inter-block communication and data transport is provided by different kinds of *signals*. As every block has data encapsulation, signals are able to carry data. Therefore, signals may affect both the control flow and the data flow.

At runtime, every block can create several *instances* (processes). This again is not a feature of the PLEX programming language but achieved by means of implementation tricks and coding conventions. Therefore, these instances are managed by the block and not by the runtime environment.

3. E-CARES prototype

In the E-CARES project, we design and implement tools for reengineering of telecommunication systems and apply them to the AXE10 system developed at Ericsson. The basic architecture of the E-CARES prototype is outlined in Figure 4. The solid parts indicate the current state of realization, the dashed parts refer to further extensions.

Below, it is crucial to distinguish between the following kinds of analysis: *Structural analysis* refers to the static system structure, while *behavioral analysis* is concerned with its dynamic behavior. Thus, the attributes “structural” and “behavioral” denote the outputs of analysis. In contrast, *static analysis* denotes any analysis which can be performed on the source code, while *dynamic analysis* requires information from program execution. Thus, “static” and “dynamic” refer to the inputs of analysis. In particular, behavior can be analyzed both statically and dynamically.

We obtained three sources of information for the static analysis of the structure of a PLEX system. The first one is the *source code* of the system. It is considered to be the

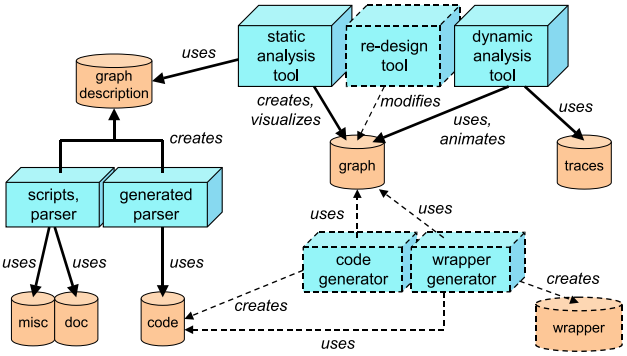

Figure 4. Prototype architecture

core information as well as the most reliable one. Through code analysis (parsing) a number of structure documents is generated from the source code, one for each block. These structure documents form a kind of textual graph description. The second and the third source of information are *miscellaneous documents* (e.g., product hierarchy description) and the system *documentation*. As far as the information from these sources is computer processable, we use parsers and scripts to extract additional information, which is stored in structure documents, as well.

The *static analysis tool* processes the graph descriptions for individual blocks and creates corresponding subgraphs of the structure graph representing the overall application. The subgraphs are connected by performing global analyses in order to bind signal send statements to signal entry points. Moreover, the subgraphs for each block are reduced by performing simplifying graph transformations [13]. The static analysis tool also creates views of the system at different levels of abstraction. In addition to structure, static analysis is concerned with behavior (e.g., extraction of state machines or of potential link chains from the source code).

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. Such a list constitutes a *trace* which is fed into the *dynamic analysis tool*. Interleaved with trace simulation, dynamic analysis creates a graph of interconnected block instances that is connected to the static structure graph. This helps telecommunication experts to identify components of a system that take part in a certain traffic case. At the user interface, traces are visualized by collaboration and sequence diagrams.

Both the static and the dynamic analysis tool calculate *metrics* which were designed to improve the understanding of both structure and behavior. These metrics are visualized e.g. in the underlying structure graph (see Section 6).

The dashed parts of Figure 4 represent planned extensions of the current prototype. The *re-design tool* will be

used to map structure graph elements to elements of a modeling language (e.g., ROOM [20] or SDL [8]). This will result in an architecture graph that can be used to perform architectural changes to the AXE10 system. The *code generator* will generate PLEX code according to changes in the structure graph and/or the architecture graph. The *wrapper generator* will enable reuse of existing parts of the AXE10 system written in PLEX in a future switching system that is written in a different programming language, e.g., C++.

To reduce the effort of implementing the E-CARES prototype, we make extensive use of generators and reusable frameworks [14]. Scanners and parsers are generated with the help of JLex and jay, respectively. Graph algorithms are written in PROGRES [19], a specification language based on programmed graph transformations. From the specification, code is generated which constitutes the application logic of the E-CARES prototype. The user interface is implemented with the help of UPGRADE [1], a framework for building interactive tools for visual languages.

4. Structural analysis

The static structure of a PLEX program is represented internally by a *structure graph*, a small (and simplified) example of which is shown in Figure 5[2]. In the example, there is a subsystem which contains two blocks A and B. The subgraphs for these blocks are created by the PLEX parser. The subgraph for a block — the block structure graph — contains nodes for signal entry points, labels, (contiguous) statement sequences, subroutines, exit statements, etc. Thus, the block structure graph shows which signals may be processed by the block, which statement sequences are executed to process these signals, which subroutines are used for processing, etc. In addition, the block structure graph initially contains nodes representing outgoing signals. Subsequently, a global analysis is carried out to bind outgoing signals to receiving blocks based on name identity. In our example, a signal H is sent in the statement sequence X of block A. This signal is bound to the entry point of block B. From the signal edges between statement sequences and signal entry points, more coarse-grained communication edges may be derived (between blocks and eventually between subsystems).

Externally (at the user interface), the structure graph is represented by multiple *views* [13]. The product hierarchy is displayed in a *tree view*. Furthermore, there is a variety of *graphical views* which display the structure graph at different levels of abstraction (internals of a block, block communication within a subsystem, communication between subsystems). For example, Figure 13 shows block communications within a subsystem of AXE10 (see Section 6 for further explanations). The user may select among a set of different, customizable layout algorithms to arrange graphical representations in a meaningful way. He may also collapse and expand sets of nodes to adjust the level of detail. Graph elements representing code fragments are connected to the respective source code regions, which may be displayed on demand in *text views*.

[2]For the time being, please ignore all graph elements for link chains (lc), which will be explained in Section 5.

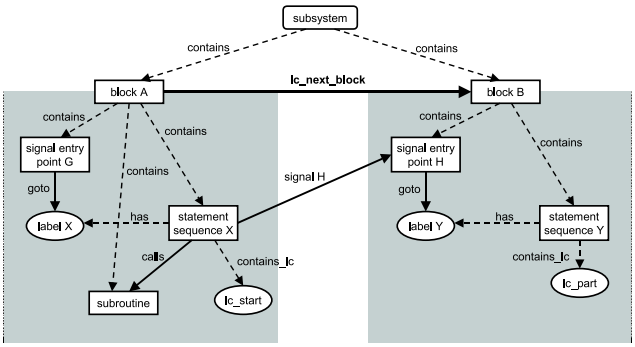

Figure 5. Cut-out of a structure graph

5. Behavioral analysis

5.1. Motivation

As stated in Section 1, we found that the static system structure is not very expressive in the case of telecommunication systems. These highly dynamic, flexible, and reactive systems handle thousands of different calls at the same time. The numerous services provided by a telecommunication system are realized by re-combining and re-connecting sets of small (stand alone) processes, block instances in our case, at runtime. Each of these *block instances* realizes a certain kind of (internal) mini-service. Some of the blocks can even create instances for different mini-services dependent on the role they have to play in a certain scenario.

Therefore, structural analysis as described in Section 4 is not sufficient to understand telecommunication systems. For example, the structure graph does not contain any information on how many instances of a single block are used to set up a simple phone call. In Figure 6, a so-called *link chain* for the GSM-layer 3 part of a simple mobile originating call is sketched. Link chains describe how block instances are combined at runtime to realize a certain service. Each node represents a block instance. An edge between two nodes indicates signal interchange between these blocks. Each link chain consists of a main part (directed edges) and side-links for supplementary services (authentication, charging, etc.). The directed edges between elements of the main link chain indicate its establishment from left to right; communication is bidirectional in all cases. In correspondence to Figure

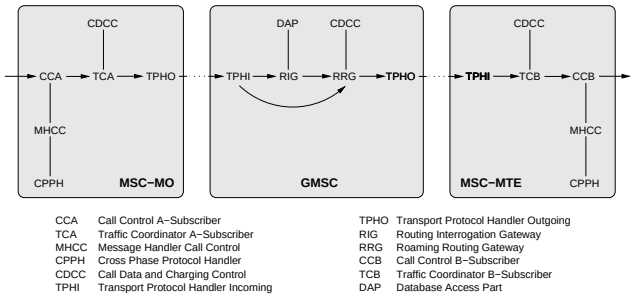

Figure 6. Simplified link chain for mobile originating call at GSM-layer 3

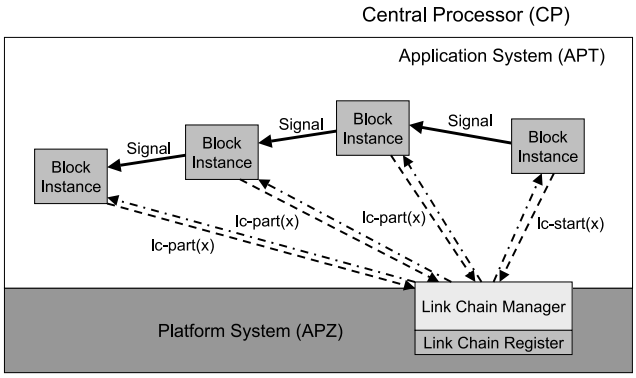

Figure 7. Link chain handling

2, the link chain in Figure 6 is divided into the three parts MSC-MO, GMSC, and MSC-MTE.

This simple example shows that there are, e.g., three instances of the charging access (CDCC), two message handler instances (MHCC), and two instances of the CPPH protocol handler block needed to setup a mobile originating call. This kind of behavioral information is very important to telecommunication engineers. Sections 5.2 and 5.4 describe how information on link chains can be derived via static and dynamic analysis, respectively.

Furthermore, each block in Figure 6 implements at least one *state machine*. State machines are a common modeling means in the design of telecommunication systems. Therefore, telecommunication experts are interested in having a good knowledge about the state machines implemented in a block and their operation at runtime. Extraction of state machines from source code is discussed in Section 5.3.

5.2. Static link chain analysis

In case of the AXE10, link chains can be obtained from static analysis by making use of an *error recovery* mechanism implemented in the system. The mechanism allows to kill processes specific to an erroneous link chain if normal error recovery fails. Each block instance that is the initiator of a new link chain first notifies the *Link Chain Manager* (LCM) of the APZ via an lc_start message (Figure 7). The LCM creates a new *Link Chain Identifier* (LID), stores it in the *Link Chain Register*, and returns it to the initiating block instance. Now the LID will be implicitly attached to every signal the initiating block instance sends to any other block instance. If another block instance is requested, it notifies its participation in a link chain by sending an lc_part message to the LCM which in turn will respond by sending an acknowledgment containing the currently valid LID from the Link Chain Register. The LID will now be implicitly attached to the second block's signals as well etc.

The lc_* messages to the platform system are represented by corresponding PLEX statements in the source code. Thus, these statements are detected and inserted into the structure graph during structural analysis (see Figure 5). The static link chain analysis searches for the occurrence of an lc_start statement in a given block's structure graph and marks the surrounding statement sequence. All signals sent after the link chain initiation notification by this statement sequence do transmit an LID[3]. Thus, these signals are traversed to the receiving blocks. Next, the block structure graph of each receiving block is traversed starting from a signal entry point which one of the signal edges is connected to. If an lc_part statement is reachable from this signal entry, an lc_next_block edge is inserted into the structure graph between sending and receiving block.

In Figure 5, block A contains an lc_start statement in statement sequence X. Furthermore, signal H triggers the execution of statement sequence Y in block B which contains an lc_part statement. As a result, a new edge of type lc_next_block is inserted into the structure graph to indicate that A and B can take part in the same link chain.

After having performed static link chain analysis on the structure graph, it is possible to visualize which blocks *potentially* take part in the same link chain and which block triggers the participation of which other blocks. Indeed, if the static link chain analysis is performed for the whole structure graph, the result is the union of *all* link chains that can occur at runtime. Nevertheless, this information is very useful as most link chains appear to be constructed from smaller, invariant link chains. Additionally, the scope of a static link chain analysis can be limited by selecting a starting block. In this case, only blocks are considered that are reachable from the starting block via a path of signals. Here, link chain analysis also accepts lc_part statements as initiators of a 'new' link chain because an intermediate block of a real link chain might have been selected as starting point. Furthermore, certain classes of telecommunication services

[3]To be precise, the lc_start statement must not be part of a conditional statement to make sure that subsequently sent signals do transmit an LID. Otherwise, only signals in the same conditional context do so.

(e.g., voice calls) always have a kind of basic common link chain. Smaller chains representing different supplementary services are just linked in on demand. In combination with system traces (see Section 5.5), it is possible to identify the basic common link chains.

5.3. State machine extraction

The information extracted from the source code and other sources of information, its representation, and its visualization have to meet the demands of both the reengineer and the system experts involved. In particular, telecommunication systems are often planned and modeled using *state machines*. That is, system experts think in terms of state machines and protocols. Thus, the design process is process- and behavior-centered rather than data-centered, and blocks are just implementations of one or more state machines. Though PLEX does not provide any language elements for state machine implementation, the knowledge of some (informal) design rules at Ericsson and additional manual analysis of a couple of PLEX blocks allowed to develop an algorithm to detect state machines in and extract them from a block's source code or structure graph, respectively.

The example code in Figure 8 represents the main patterns used to implement state machines in PLEX. Every block contains declarations of a stored (DS flag) data record (BlockTRecord), a file to store several of these records, and a pointer (BlockTPointer) to the record currently used in execution. Each of these records represents a block instance. A block implements a state machine if its stored record contains a symbol variable of a string-valued enumeration type whose name contains the substring STATE. Additionally, the set of possible values of this state variable must enclose the values IDLE and SEIZED. Furthermore, each signal entry (ENTER <signal name>) is immediately followed by a case statement whose condition queries the current value of the state variable. Each branch of this case statement triggers different executions in the block (instance). Accordingly, a state change is represented by an assignment to the state variable. A design rule determines that a block must change its state at most once per execution cycle. No state change corresponds to a feedback loop in the underlying state machine. Furthermore, after a state change and a possibly subsequent signal sending statement, a block has to terminate immediately by means of an EXIT statement.

Hence, using this information we can conclude from the example in Figure 8 that block T owns a state variable named STATE. This state variable can have – among others – the values IDLE, SEIZED, SETUP, and DISC. When receiving (consuming) SignalA in states IDLE or SEIZED, the block will change to state SETUP and send SignalB. This results in the lower three states and two transitions on the right

```
...
 ! Declare Sector !
DECLARE
  ...
  RECORD BlockTRecord;
    ...
    SYMBOL VARIABLE STATE = (IDLE, SEIZED, SETUP, DISC, ...) DS;
    ...
  END RECORD;
  POINTER BlockTPointer(BlockTRecord);
  ...
END DECLARE;

 ! Program Sector !
PROGRAM;
  ! Begin Interface !
  ENTER SignalA WITH SignalDataA;
     CASE BlockTPointer:STATE IS
        WHEN IDLE, SEIZED DO
           GOTO SequenceY;
        OTHERWISE DO
           GOTO UnexpectedState;
     ESAC;

  ENTER SignalC WITH SignalDataC;
     CASE BlockTPointer:STATE IS
        WHEN DISC DO
           GOTO SequenceZ;
        OTHERWISE DO
           GOTO UnexpectedState;
     ESAC;
  ...
   ! End Interface !

  SequenceY)
     ...
     BlockTPointer:STATE = SETUP;
     SEND SignalB WITH SomeData;
     EXIT;

  SequenceZ)
     ...
     BlockTPointer:STATE = IDLE;
     SEND SignalD WITH AgainSomeData;
     EXIT;
  ...
END PROGRAM;
...
```

Figure 8. Example code with relevant statements for state machine extraction

side of Figure 9. That is, a cut-out of the block's state machine has been extracted. The rest of the state machine can be extracted stepwise by analyzing the source code for each signal entry as originator again and again. But compared to a graph-based algorithm this procedure is inefficient for large systems like the AXE10.

The *graph-based state machine extraction algorithm* traverses a block's structure graph trying to find a path from a signal entry to a statement sequence that contains an assignment to the state variable. In each run, the starting states from the conditional clauses of the case statement and the name of the incoming signal are stored. If a state assignment is found, a state transition from the starting state to the new state is inserted into the block's state diagram. The state transition is annotated with the incoming and the outgoing signal. Additional information that is necessary

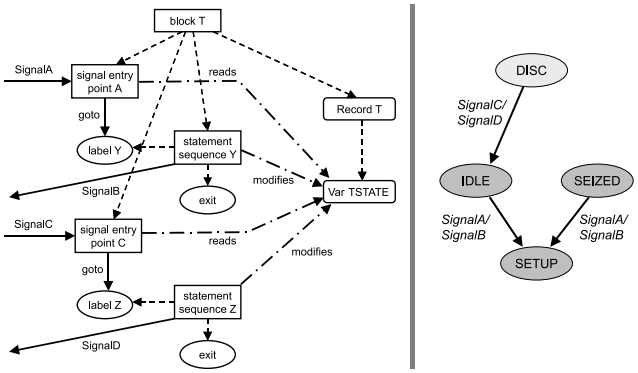

Figure 9. State machine extraction on structure graph

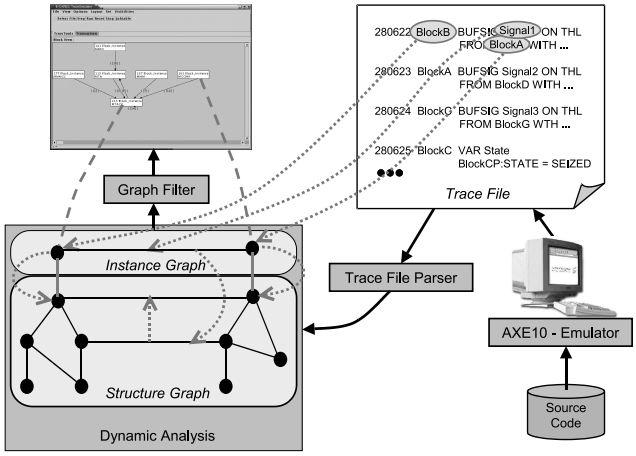

Figure 10. Obtaining dynamic information

for the state machine extraction and that is not represented by elements of the structure graph (e.g., conditional statements) is stored in attributes of nodes and edges. Therefore, all information that is needed for the extraction is obtained in a single run of the PLEX parser.

In the sample graph of Figure 9 corresponding to the source code in Figure 8, the transition from DISC to IDLE can be obtained by starting with signal entry C. The conditions for the branches of the subsequent case statement are stored in the corresponding GOTO edges. On state DISC, the execution of block T is continued at statement sequence Z which modifies the state variable STATE, sends signal SignalD, and terminates its execution.

The result of the state machine extraction is a state machine of a block that might be too large. That is, it may contain too many transitions. A reason for this anomaly is given by nested if-clauses that query the current value of different variables. At runtime only a small amount of all possible pairs of values of these variables might be valid. That is, only a subset of all possible execution paths through this part of the source code might be possible in reality. The identification of those state transitions that are valid is possible in combination with trace information (see Section 6).

Another problem arises from the fact that blocks can have several state machines inside – one for every miniservice they implement. The algorithm is designed to extract all state machines in parallel in a single pass.

5.4. Tracing

As stated in Section 3, there are two possibilities to obtain dynamic information: using an emulator or querying a running AXE10. In the latter case, the complexity of the information traced has to be reduced to avoid that the system gets into timing problems and becomes instable. Besides error handling, tracing jobs have the highest priority in a running AXE10. Therefore, intense tracing can cause abnormal behavior, e.g., through an increased number of missed time constraints and the resulting system recovery actions. Furthermore, tracing an AXE10 that is in normal operation – handling thousands of calls at the same time – results in trace files containing information on all of these calls. This huge amount of information makes it difficult to focus on a special call scenario, which is one of the goals of tracing. If real-time issues are not regarded, there is no need to query a running switching system. Consequently, we decided to use the AXE10 emulator. However, the procedure described below and the dynamic analysis tool are the same for both sources of dynamic information.

First, the tracing facility of the emulator is configured. The configuration determines which information (e.g., signal interchange, data access) is to be captured in a trace file and which parts of the system should be observed (e.g., several blocks in a certain subsystem). Currently, we are interested in the following information: signals, sending and receiving block (instance) of a signal, data transferred with a signal, signal priority, and assignments to certain (state) variables. After the configuration has finished, the running emulator starts to capture all actions whose tracing has been switched on (Figure 10). Now the testing equipment can be used to set up a call scenario of interest, e.g., a mobile originating call, and trigger the execution of telecommunication services implemented in the AXE10.

The dynamic analysis tool and the associated trace file parser read a selected trace file stepwise – action by action – on demand. Every action is analyzed to identify the participating entities (block instances, signals, etc.) of the instance graph. The instance graph is then modified accordingly – new elements are inserted and existing ones are updated. Some modifications are also propagated into the static structure graph. For example, every block instance is connected

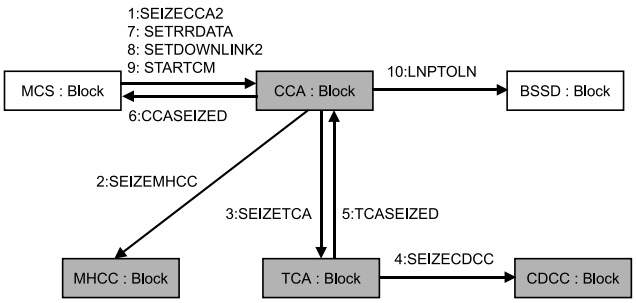

Figure 11. Example of a collaboration diagram

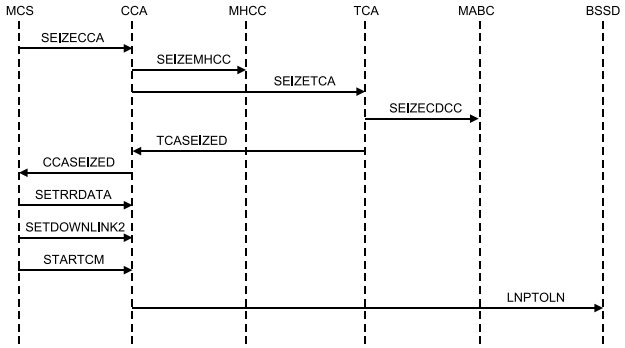

Figure 12. Example of a sequence diagram

to its corresponding block in the structure graph and the activation of a block instance results in the activation of the corresponding block.

During their daily work, telecommunication experts need different views on the traffic cases they analyze. Sometimes they just need a complete overview, sometimes they focus on the inter-work of two block instances, and sometimes the development of a call over time is their main interest. Therefore, dynamic analysis provides three different *working modes*: In *step mode*, the dynamic analysis process is interactive. That is, the reengineer triggers the analysis of each action in the selected trace file manually. In *run mode*, the analysis tool processes the whole trace in one run. This mode has been implemented to create complete diagrams for re-documentation purposes. In *animation mode*, the actions in a trace file are processed successively in adjustable time intervals. The animation mode is used to produce a slow motion replay of the actions that took place in the systems software with respect to a certain call scenario. For convenience, it is possible to switch between the three working modes at any time.

All changes to the instance graph are directly visualized in the dynamic analysis tool. Here, a reengineer is able to refine and adjust the amount of information displayed by means of a complex graph filter interactively. Furthermore, he can choose between two kinds of diagrams that are commonly used in the telecommunications domain: *collaboration diagrams* and *sequence diagrams* similar to those provided in the UML [2]. Collaboration diagrams can be used in all three working modes of the analysis tool whereas sequence diagrams are mainly used for re-documentation purposes. For example, the collaboration diagram in Figure 11 and the sequence diagram in Figure 12 show the same trace at the same stage. The different grey scales of the block instances in Figure 11 illustrate that the block instances belong to different subsystems.

The dynamic analysis tool has proved its value for system analysis and system understanding in several tests and discussions with the expert group from Ericsson supporting our research. However, in complex traces, huge amounts of information are obtained from the emulator. As a result, numerous elements are inserted into the instance graph that have to be visualized to the user at once. In step mode and even worse in animation mode, the user often gets confused by the quantity of information after some time and might even lose his track. There are two complementing possibilities to support the user in understanding complex call scenarios: limiting the observation scope of a trace and extending the visualization with aging.

The *limitation* of the *observation scope* is necessary anyhow to suppress noise in traces that results from periodic jobs like access statistics in order to focus on special aspects. Limitation means that observation is only activated for selected blocks, signals, and variables of a traced system in the emulator configuration. All other blocks etc. are ignored; e.g., tracing could be limited to a certain subsystem. But the block instances that take part in a certain call scenario are always spread over various parts of the AXE10. Hence, there are e.g. signals coming from the ignored parts to observed blocks and vice versa. This leads to missing links (signals) in the control flow of a trace, in the instance graph, and in the diagrams produced by the dynamic analysis tool. Therefore, a heuristic has been implemented in the dynamic analysis tool that inserts *synthetic signals* during trace analysis to keep traces connected. For example, the heuristic marks signals sent to unobserved block instances (*out signal action*), tracks the following signal actions for a corresponding *in signal action*, and inserts an appropriate synthetic incoming signal if it is missing[4]. For this purpose, the dump of the emulator configuration that corresponds to the current trace file is used to extract information on observed entities. This is necessary to keep the diagrams correct. Similarly, if another block is activated out of turn after a signal has been sent to the ignored part of a system, there

[4]For each signal a pair of signal actions must be captured for a trace file to be correct – an outgoing signal action at the sending block instance and an incoming signal action at the receiving block instance. But, signal actions are only captured for observed blocks.

must have been communication in the ignored part which has not been captured. This circumstance can also be represented by an appropriate synthetic signal during dynamic analysis that summarizes this unknown communication.

In addition, the user is supported during trace analysis through the *aging* of the *representation elements* in the visualization of the instance graph. Consider a collaboration diagram view, which represents the current state of a call's link chain. Some elements in this diagram remain inactive while others participate continuously in the current part of the trace. But this difference in the amount of participation is not noticed easily in complex traces. Therefore, all elements in the collaboration diagram which have not participated recently in a current trace are faded out continuously. If an element participates again, it is re-displayed immediately. This procedure guarantees that the visualization focuses on elements that determine the current behavior. Keeping in mind that a call link chain develops over time and that it creates temporary side links that are never released explicitly for efficiency reasons, visualization aging is a good means to analyze the evolution of a call over time.

5.5. Combining analyses

So far, the direct analysis of a system's runtime behavior has been addressed. But there are additional applications for a combination of analyses. In Section 5.2 we have mentioned that link chains for a certain class of calls (e.g., voice calls) have a common *basic link chain* that is extended by *side links* representing different supplementary services (e.g., charging). Traces can be used to identify basic link chains and supplementary side links. The procedure is similar to feature analysis as described in [7].

Always, several traces of the system are needed. To identify the basic link chain of a class of calls, a number of traces of different voice calls is needed. All these traces are now processed by the dynamic analysis tool to produce corresponding instance graphs. Next, the logical `and` operation over all instance graphs is used to create an instance graph that represents the intersection of all instance graphs that result from trace analysis. This intersection graph corresponds to the common basic link chain of all these traces.

To find out whether a side link inherently belongs to a given type of call (e.g., a mobile originating call), additional traces are needed that belong to other types of calls. Now, all elements that are part of the other calls can be subtracted from the instance graph of the first trace which results in a set of elements that inherently belong to this kind of call.

The state machines statically extracted from source code can be reused in the dynamic analysis of a systems runtime behavior. Remember that state changes were captured in trace files as well. In combination with a trace it is possible

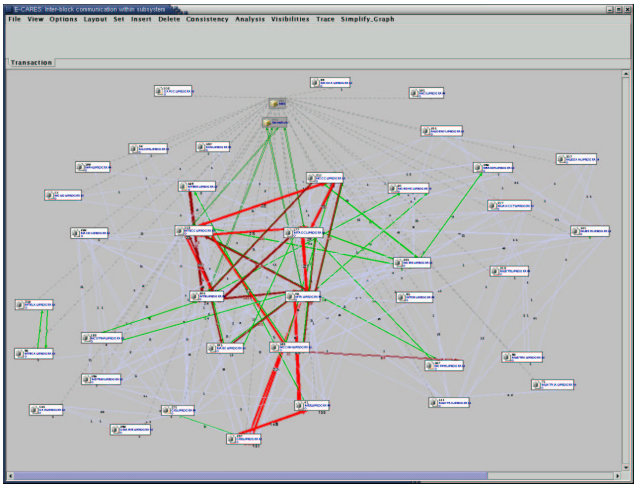

Figure 13. Static communication weights

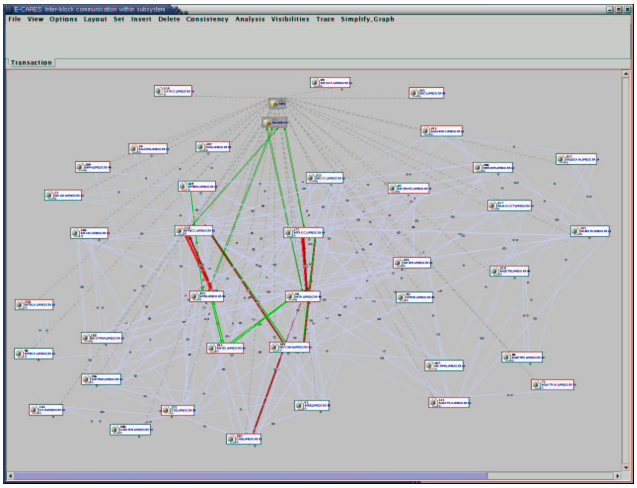

Figure 14. Dynamic communication weights

to animate the state machines of a marked set of blocks and visualize them to the telecommunication expert.

6. Metrics

As presented so far, the E-CARES prototype provides qualitative information on the structure and behavior of a telecommunication system. In addition, telecommunication experts are interested in quantitative information. The *metrics* introduced in this section further improve the understanding of a telecommunication system. Usually, metrics are used for assessment (e.g., of the complexity of a system), but so far we have not used them for this purpose.

Metrics are used to assign *weights* to graph elements. These weights are defined with the help of both static and dynamic information. They are attached to both structural

and behavioral graphs. At the user interface, weights are visualized with the help of different representation attributes, e.g., colors, size of boxes, or thickness of edges.

As an example, Figure 13 shows a view on the structure graph which displays communications at the block level. *Static weights* are assigned to communication edges based on the number of types of signals sent from the source to the destination. Colors and thickness of edges are used to represent communication weights. The thick lines at the center of the diagram indicate where the main traffic occurs. Telecommunication experts at Ericsson considered this information useful to distinguish "communication highways" from "secondary roads". Similarly, weights may be attached to blocks based on the code size so that is easier to understand where the main functionality is implemented (not shown in the figure).

Static weights provide general information on potential executions. However, the telecommunication expert may be interested more specifically in what happens in a specific trace (or a set of traces). To this end, we offer *dynamic weights*. For example, each communication edge is weighted by the number of signals sent along this edge in a specific trace. Again, we may display these weights in the structure graph. By comparing Figure 14 to Figure 13, the user may recognize the specific behavior in a certain trace. It is also possible to hide all elements which do not participate at all in the trace under study. In this way, the user may compare expected and actual behavior. Alternatively, weights may be displayed in a collaboration diagram, which provides a more detailed view on the execution since it shows individual block instances.

As a final example, let us briefly discuss the assignment of weights to elements of state machines. In Subsection 5.3, we mentioned that static analysis may produce a state diagram which is too large. To detect potentially dead elements, state transitions are weighted according to the number of occurrences in some trace (or in a set of traces). State transitions with weight 0 may be obsolete.

7. Related work

The E-CARES prototype is a reengineering environment designed for telecommunication systems. In particular, it is based on domain-specific architectural concepts. A telecommunication system is modeled as a set of active components which communicate by sending and receiving signals. Thus, modeling is process-centered. Since the static system structure is not very expressive, analysis of behavior plays a crucial role in E-CARES.

In contrast, many other reengineering tools such as e.g. Rigi [16] or KOGGE [6] primarily focus on the static system structure. Moreover, they are typically data-centered; consider e.g. tools for the reengineering of COBOL programs as described in [5, 15, 23]. Here, recovery of units of data abstraction and migration to an object-oriented software architecture play a crucial role [3]. More recently, reengineering has also been studied for object-oriented programming languages such as C++ and Java. E.g., TogetherJ or Fujaba [17] generate class diagrams from source code.

Reengineering of telecommunication systems follows different goals. Telecommunication systems are designed in terms of layers, planes, services, protocols, etc. Behavior is described with the help of state machines, message sequence charts, link chains, etc. Thus, reengineering tools are required to provide views on the system which closely correspond to system descriptions given in standards, e.g., GSM. Telecommunication experts require views on the system which match their conceptual abstractions.

In this paper, we focus on tools and techniques for behavioral analysis offered by the E-CARES prototype. Below, we compare these to other approaches.

In the literature, a lot of work is described on the extraction of dynamic information from object-oriented systems. The motivation for the need of dynamic information is quite similar to ours: In object-oriented systems the runtime behavior of a system is neither predictable nor understandable by just performing different kinds of static analysis on the system's source code. Source code analysis does not provide enough information to understand a software system completely because there are components and relations that only exist during its runtime.

There are different methods described how to extract or gather dynamic information. In the majority of these methods, the code of the target system is instrumented so that a trace is produced of the components executed in each test case. The level of instrumentation ranges from very low level instrumentation where the components of a trace are branches in the target system control flow [25] to the instrumentation of class or method entry and exit points [22] or inter-process messages [12, 24], respectively. Low level instrumentation of a production system brings with it a significant performance penalty since trace information is generated each time the system passes through an conditional statement like `if`, `switch`, or any other decision points. In general, for a large real-time system like the AXE10 switching system low level instrumentation would be too difficult. In particular, if we ran a low level instrumented system, we would experience severe timing problems within the system. This would result in unusable traces.

Another approach that can be found nowadays is the utilization of debugger and profiler logs to gather dynamic information [7, 9, 21]. In the end, debugger and profiler just provide a more convenient, more efficient but also more indirect way of instrumenting source code or byte code, respectively. Considering real-time systems, if too much information is queried this could result in unusable traces as

well. Fortunately, we can use an emulator to execute the AXE10 source code as is – no instrumentation is necessary. The output of tracing information is a configurable built in functionality of this emulator. Additionally, the virtual time mode of the emulator guarantees that complex test cases are possible without running into timing problems.

There are different proposals for the representation of dynamic information to a reengineer. In most cases, either collaboration diagrams or different variations of sequence diagrams are used [21]. Others [9, 18] use different kinds of non-standard graph representations. In rare cases, there are even more elaborate representations like the piano-roll representation in [12]. We have chosen to implement both a collaboration diagram view and a sequence diagram view because these kind of representations are very common to telecommunication engineers. Furthermore, we have implemented different inspection modes for traces (manual step-by-step inspection, slow-motion replay with aging, and batch processing for re-documentation purposes). This provides a reengineer with maximum flexibility in the processing and utilization of dynamic information.

When comparing the E-CARES approach of dynamic information retrieval to other approaches, we found that the majority of approaches collect this information from a running system. There are only a few approaches [11] that derive behavioral information via static analysis. In E-CARES both static and dynamic analysis are utilized to retrieve information on a systems' behavior. Furthermore, information retrieved from static and dynamic analysis can be combined to overcome each others' deficits: E.g., for state machine extraction, code coverage is guaranteed via static analysis while the precise dynamic information can be used to search for potentially invalid transitions. That is, information can be exchanged between static and dynamic views.

In the literature, the post-processing of dynamic information ranges from slicing of program structures, feature location in code [7], and state machine extraction [22] to re-documentation of the current implementation. The E-CARES prototype currently comprises state machine extraction and re-documentation; the implementation of program slicing (which is related to architecture extraction in our case) and feature location is still under construction.

8. Conclusion

We have presented tools for understanding the behavior of telecommunication systems. These tools were developed in close cooperation with telecommunication experts from Ericsson. We followed an evolutionary approach to tool development, i.e., functionality was added incrementally in response to the requirements stated by the telecommunication experts. In this way, we took a step towards an environment which is based on domain-specific concepts.

So far, approximately 500,000 lines of PLEX code plus several ten thousands of lines of additional documents have been processed, analyzed, visualized, and inspected on different levels of abstraction. Understandably, the exact and detailed results are confidential and cannot be discussed here. According to the telecommunication experts, the E-CARES prototype allows to visualize the AXE10 software systems in terms of their daily use, e.g., block dependencies, state diagrams, link chains, and sequence diagrams. For that, no tools have been available so far. In particular, the dynamic analysis tool has proved its value for system analysis and system understanding. Therefore, we are convinced that analyzing and visualizing the dynamic behavior of telecommunication systems is key to system understanding. Furthermore, we believe that only the combination of static and dynamic as well as structural and behavioral analysis – integrated in an interactive reengineering framework – allows to obtain best possible results.

The current E-CARES prototype is domain dependent to a certain extent. In particular, the structure graph is tightly bound to the PLEX programming language, i.e., nodes of the structure graph correspond to PLEX constructs such as blocks, signal declarations, signal send statements, etc. At Ericsson, other programming languages, e.g., C and C++, are used, as well. To support multi-language systems, we have started to divide the structure graph into programming language dependent and language independent aspects. The final goal is a flexible and extensible reengineering system that consists of a core system which is extended with specific functionality in form of plug-in modules. For this purpose, we are currently defining general interfaces between the different parts of the system, e.g., between parsers and static analysis tool, etc. The visualization of extracted information in form of different views, e.g., structural views, sequence diagrams, collaboration diagrams, and state machines, are independent of the current subject of study. Even the underlying analyses can be readily applied to other real-time systems, provided that the needed information is available and storable in terms of our structure graph.

Additionally, we need a neutral representation at the architecture level. Therefore, we are currently extending the E-CARES prototype such that it builds a ROOM model from analyzed programs. Others, like SDL or the component model of UML 2.0 will be explored in the future.

Finally, we will extend the E-CARES prototype to cover re-design. That is, we intend to support the restructuring and extension of existing systems in addition to reverse engineering, which has been our focus so far.

Acknowledgments The E-CARES project is funded by Ericsson Eurolab Deutschland GmbH (EED); support is gratefully acknowledged. We would like to thank the fol-

lowing members of EED for their constant active support, critical feedback, and patience: Martin Blumbach, Jörg Bruss, Axel Jeske, Per Ljungberg, Ari Peltonen, Andreas Thülig, Dietmar Wenninger, and Andreas Witzel.

References

[1] B. Böhlen, D. Jäger, A. Schleicher, and B. Westfechtel. UPGRADE: Building interactive tools for visual languages. In *Proceedings of the 6th World Multiconference on Systemics, Cybernetics, and Informatics (SCI 2002)*, volume I (Information Systems Development I), pages 17–22, Orlando, Florida, July 2002.

[2] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley: Reading, MA, 1999.

[3] G. Canfora, A. Cimitile, A. De Lucia, and G. Di Lucca. Decomposing legacy systems into objects: An eclectic approach. *Information and Software Technology*, 43(6):401–412, 2001.

[4] E. J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[5] K. Cremer, A. Marburger, and B. Westfechtel. Graph-based tools for re-engineering. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(4):257–292, Aug. 2002.

[6] J. Ebert, R. Süttenbach, and I. Uhe. Meta-CASE in practice: A case for KOGGE. In *Proceedings 9th International Conference on Advanced Information Systems Engineering CAiSE 1997*, LNCS 1250, pages 203–216, Barcelona, Spain, June 1997.

[7] T. Eisenbarth, R. Koschke, and D. Simon. Aiding Program Comprehension by Static and Dynamic Feature Analysis. In *Proceedings International Conference on Software Maintenance ICSM 2001*, pages 602–611, Florence, Italy, Nov. 2001.

[8] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL - Formal Object-oriented Language for Communicating Systems*. Prentice Hall, 1997.

[9] J. Gargiulo and S. Mancoridis. Gadget: A Tool for Extracting the Dynamic Structure of Java Programs. In *Proceedings International Conference on Software Engineering and Knowledge Engineering SEKE 2001*, Buenos Aires, Argentina, June 2001.

[10] R. Kazman, S. G. Woods, and J. Carrière. Requirements for integrating software architecture and reengineering models: CORUM II. In *Working Conference on Reverse Engineering*, pages 154–163, Hawai, USA, Oct 1998.

[11] R. Kollmann and M. Gogolla. Capturing Dynamic Program Behavior with UML Collaboration Diagrams. In *Proceedings European Conference on Software Maintenance and Reengineering CSMR 2001*, pages 58–67, Lisbon, Portugal, Mar. 2001.

[12] K. Lukoit, N. Wilde, S. Stowell, and T. Hennessey. TraceGraph: Immediate Visual Location of Software Features. In *Proceedings International Conference on Software Maintenance ICSM 2000*, pages 33–39, San Jose, California, USA, Oct. 2000.

[13] A. Marburger and D. Herzberg. E-CARES research project: Understanding complex legacy telecommunication systems. In *Proceedings 5th European Conference on Software Maintenance and Reengineering CSMR 2001*, pages 139–147, Lisbon, Portugal, 2001.

[14] A. Marburger and B. Westfechtel. Graph-based reengineering of telecommunication systems. In *Proceedings of the International Conference on Graph Transformations ICGT 2002*, LNCS 2505, pages 270–285, Barcelona, Spain, Oct. 2002.

[15] L. Markosian, P. Newcomb, R. Brand, S. Burson, and T. Kitzmiller. Using an enabling technology to reengineer legacy systems. *Communications of the ACM*, 37(5):58–70, 1994.

[16] H. A. Müller, K. Wong, and S. R. Tilley. Understanding software systems using reverse engineering technology. In *The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences ACFAS 1994*, pages 41–48, Montreal, Canada, May 1994.

[17] U. Nickel, J. Niere, and A. Zündorf. Tool demonstration: The Fujaba environment. In *Proceedings of the 22nd International Conference on Software Engineering ICSE 2000*, pages 742–745, Limerick, Ireland, Nov. 2000.

[18] T. Richner and S. Ducasse. Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information. In *Proceedings International Conference on Software Maintenance ICSM 1999*, pages 13–22, Oxford, England, Sept. 1999.

[19] A. Schürr, A. J. Winter, and A. Zündorf. Graph grammar engineering with PROGRES. In *Proceedings 5th European Software Engineering Conference ESEC 1995*, LNCS 989, pages 219–234, Barcelona, Spain, Sept. 1995.

[20] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons: Reading, MA, 1994.

[21] T. Systä. On the Relationships between Static and Dynamic Models in Reverse Engineering Java Software. In *Proceedings 6th Working Conference on Reverse Engineering WCRE 1999*, pages 304–313, Atlanta, Georgia, USA, Oct. 1999.

[22] T. Systä and K. Koskimies. Extracting state diagrams from legacy systems. In *Object-Oriented Technology ECOOP'97*, LNCS 1357, Jyväskylä, Finland, 1997.

[23] H. J. van Zuylen, editor. *The REDO Compendium: Reverse Engineering for Software Maintenance*. John Wiley & Sons: Chichester, UK, 1993.

[24] N. Wilde, C. Casey, J. Vandeville, G. Trio, and Hotz Dirk. Reverse engineering of software threads: A design recovery technique for large multi-process systems. *Journal of Systems and Software*, 43(1):11–17, 1998.

[25] N. Wilde and M. Scully. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.