Behavioral Analysis of Telecommunication Systems by Graph Transformations

André Marburger and Bernhard Westfechtel

RWTH Aachen University,
Department of Computer Science III,
Ahornstrasse 55,
52074 Aachen, Germany
{marand, westfechtel}@cs.rwth-aachen.de
http://www-i3.informatik.rwth-aachen.de

Abstract. The E-CARES project addresses the reengineering of large and complex telecommunication systems. Within this project, graph-based reengineering tools are being developed which support not only the understanding of the static structure of the software system under study. In addition, they support the analysis and visualization of its dynamic behavior. The E-CARES prototype is based on a programmed graph rewriting system from which the underlying application logic is generated. Furthermore, it makes use of a configurable framework for building the user interface. In this paper, we report on our findings regarding feasibility, complexity, and suitability of developing tool support for the behavioral analysis of telecommunication systems by means of graph rewriting systems.

1 Introduction

The E-CARES[1] research cooperation between Ericsson Eurolab Deutschland GmbH (EED) and Department of Computer Science III, RWTH Aachen, has been established to improve the reengineering of complex legacy telecommunication systems. It aims at developing methods, concepts, and tools to support the processes of understanding and restructuring this special class of embedded systems. 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 circa 1,000 executable units.

Maintenance of long-lived, large, and complex telecommunication systems is a challenging task. In the first place, maintenance requires understanding the actual system. While design documents are available at Ericsson which do support understanding, these are informal descriptions which cannot be guaranteed to reflect the actual state of implementation. Therefore, reverse engineering and reengineering tools are urgently needed which make the design of the actual system available on-line and which support planning and performing changes to the system.

According to the "horseshoe model of reengineering" [2], reengineering is divided into three phases. *Reverse engineering* is concerned with step-wise abstraction from

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

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).

In E-CARES, a prototypical reengineering tool is being developed which addresses the needs of the telecommunication experts at Ericsson. Currently, it assumes that the systems under study are written in PLEX, a proprietary programming language that is extensively used at Ericsson. However, we intend to support other programming languages — e.g., C — as well, so that the prototype may handle multi-language 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 [3]. These systems are *process-centered* rather than *data-centered* like legacy business applications. Therefore, tool support focuses particularly on understanding the behavior by visualizing traces, constructing state diagrams, etc.

Internally, telecommunication systems are represented by various kinds of graphs. The structure of these graphs and the effects of graph operations are formally defined in PROGRES [4], a specification language which is based on programmed graph transformations. From the specification, code is generated which constitutes the core part of the application logic of the reverse engineering tool. In addition, the E-CARES prototype includes various parsers and scripts to process textual information, e.g., source code. At the user interface, E-CARES offers different kinds of textual and graphical views which are realized with UPGRADE [5], a framework for building graph-based applications.

The general applicability of graph transformation systems in the reengineering domain and the advantages of using the PROGRES generator mechanism for tool development have already been discussed in [6]. A more detailed comparison of the different tools that contribute to the reengineering framework showed that there are major differences in the role of graph transformations in the different analyses implemented so far. Furthermore, there are significant differences concerning the proportion between graph transformation code and additional code (e.g., in Java) that is needed to implement these analyses. These differences will be discussed on the basis of the three examples for behavioral analysis of telecommunication systems that are introduced in 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.

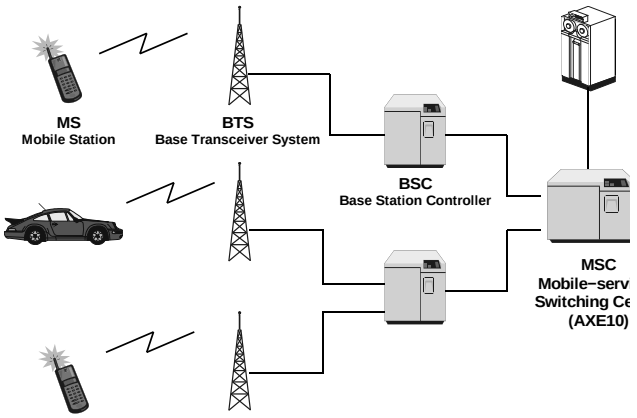


Fig. 1. Simplified sketch of a GSM network

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 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*. 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.

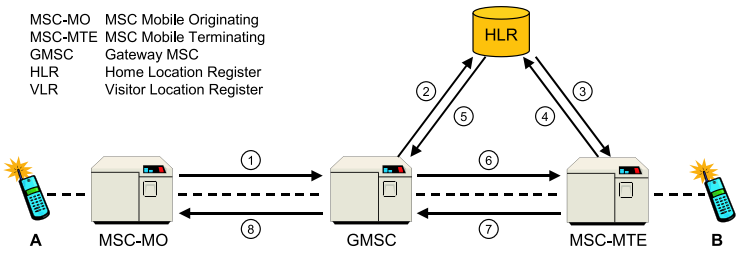


Fig. 2. Mobile originating call

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. 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 3. 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 analyses: *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

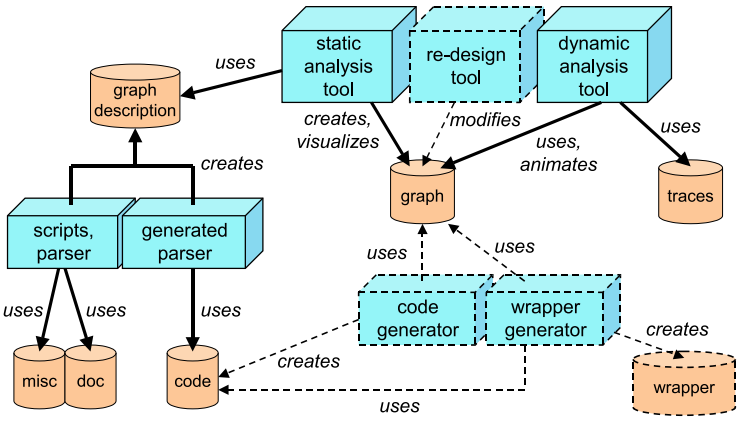


Fig. 3. Prototype architecture

the core information as well as the most reliable one. Through code analysis (parsing) a number of structure documents are 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 of 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 [1]. 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.

The dashed parts of Figure 3 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 [7] 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 [6]. Scanners and parsers are generated with the help of JLex and jay, respectively. Graph algorithms are written in PROGRES [4], 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 [5], 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 4[2]. This graph conforms with the graph scheme in Figure 5 which will be explained in Section 5. 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* [1]. 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). 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*.

5 Behavioral Analysis

The behavioral analysis of a software system either uses and extends the information gathered during structural analysis or processes graphs or graph like descriptions of the runtime systems structure, control flow, and data flow. The former is referred to as *static*

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

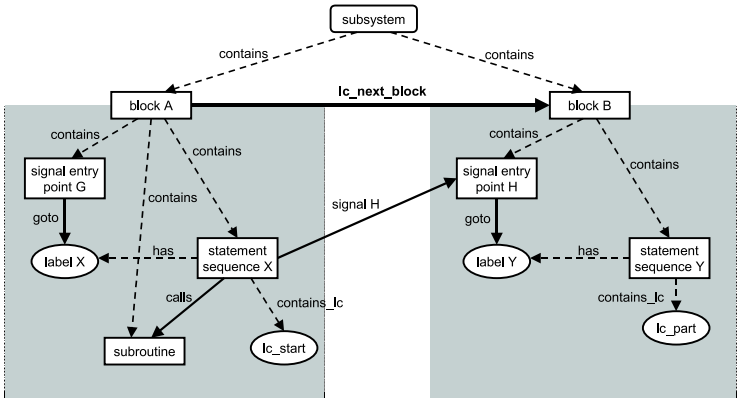


Fig. 4. Cut-out of a structure graph

analysis, the latter is called *dynamic analysis*. During the development of the different facilities for the behavioral analysis we found, that there are significant differences in the way graph transformations and the PROGRES system contribute to these facilities. In the sequel, we will discuss three example methods for behavioral analysis which show different complexity and different characteristics concerning the proportion of the graph transformations part and "externally" defined functionality.

5.1 Graph Scheme

In Figure 5, the basic graph scheme we set up for our reengineering tools is described in a notation similar to UML class diagrams. The node classes `FILE`, `ITEM`, and `ERROR` build the basis of the graph scheme. The node class `ERROR` has been introduced to be able to annotate errors with respect to structure or semantics of the graph or graph elements[3]. Nodes of type `ERROR` are connected to nodes of type `ITEM` via a `has_error` edge.

The two classes `OBJECT` and `RELATIONSHIP`, both derived from `ITEM`, divide the majority of nodes that occur in a structure graph into two groups. All node types derived from `OBJECT` are used to abstract from different kinds of fragments in the source code of an analyzed system or they describe runtime entities, respectively. Node types derived from `RELATIONSHIP` serve to describe relationships between code fragments/runtime entities. Starting from the source object, a relationship is represented by the concatenation of a `from_source` edge, a node of the corresponding subclass of `RELATIONSHIP`, and an edge of type `to_target`. This *edge-node-edge* construct allows to simulate attributed edges which we are lacking. In the remainder, we sometimes use the term *edge* if we refer to such a construct.

The node classes `OBJECT` and `RELATIONSHIP` are further refined to be able to differ between different kinds of code fragments (node class `CODE_OBJECT`), runtime

[3] Sometimes it is necessary to allow intermediate inconsistencies in a graph, e.g. during manual changes to the graph. In other cases, the correctness of a structure graph depends on a user's focus and interest.

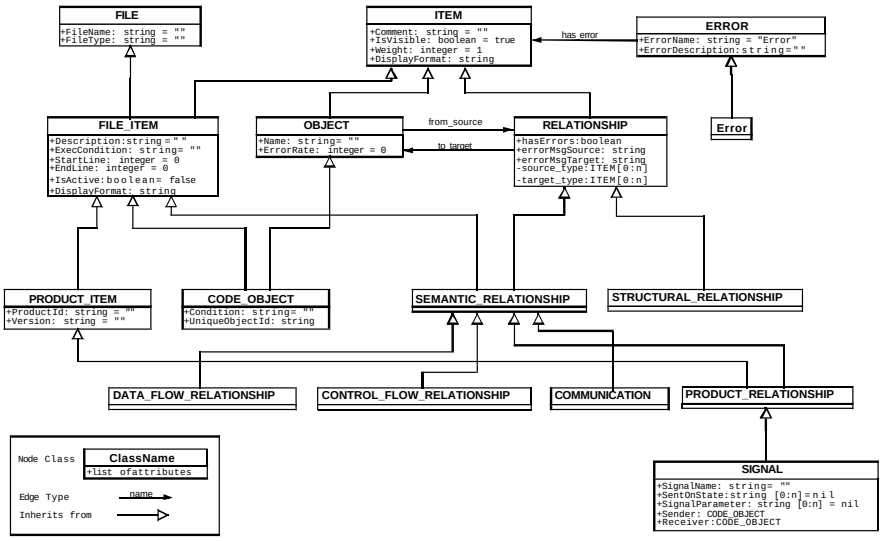


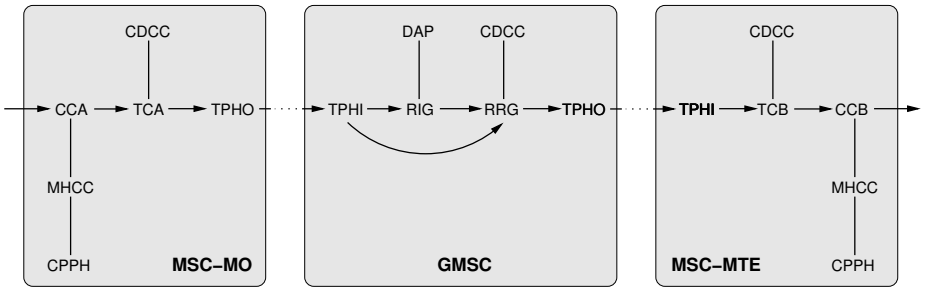
Fig. 5. Graph scheme

entities (node class TRACE_OBJECT), and relationships. For example, the structure graph we use combines structural information (STRUCTURAL_RELATIONSHIP), control flow information (CONTROL_FLOW_RELATIONSHIP), and data flow information (DATA_FLOW_RELATIONSHIP) in a single graph. This allows complex graph transformations and graph queries that utilize all three kinds of information at once. Some examples of these graph transformations are described in the following section.

5.2 Example 1: Static Link Chain Analysis

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



CCA	Call Control A–Subscriber	TPHO	Transport Protocol Handler Outgoing
TCA	Traffic Coordinator A–Subscriber	RIG	Routing Interrogation Gateway
MHCC	Message Handler Call Control	RRG	Roaming Routing Gateway
CPPH	Cross Phase Protocol Handler	CCB	Call Control B–Subscriber
CDCC	Call Data and Charging Control	TCB	Traffic Coordinator B–Subscriber
TPHI	Transport Protocol Handler Incoming	DAP	Database Access Part

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

chain indicate its establishment from left to right; communication is bidirectional in all cases. In correspondence to Figure 2, the link chain in Figure 6 is divided into the three parts MSC-MO, GMSC, and MSC-MTE.

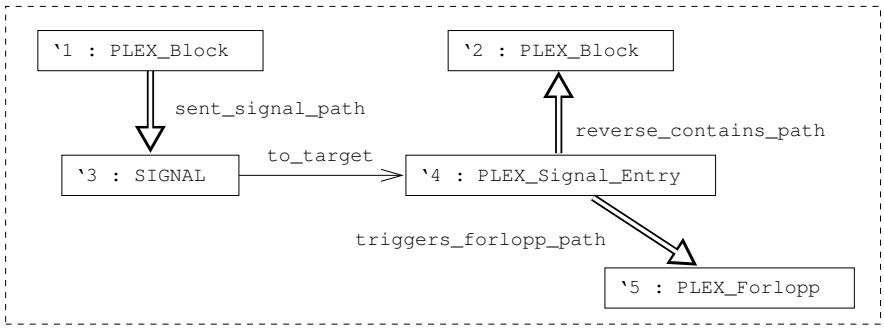
This simple example shows that there are, e.g., three instances of the charging access (CDCC), two message handler instances (MHCC), and two instances of the CPPH protocol handler block needed to setup a mobile originating call. This kind of behavioral information is very important to telecommunication engineers. Therefore, we will now show how information on link chains can be derived via static analysis using a simple graph transformation, a (static) path expression.

Figure 7 shows this static path expression. Paths are simple graph queries that only consist of a left hand side. In case of static paths, there is a side effect that materializes all matches of the left hand side in the form of edges between the source node and the target node of a path. Materialized paths can be visualized in our graph visualization tool. Therefore, we use static paths, e.g., in the analysis part of the specification to gather additional information on a system that is not obtainable via code parsing.

The path expression in Figure 7 is used to detect blocks in the analyzed system that are inter-connected in a so-called *forlopp link chain* at runtime. Normally, the information on forlopp link chains is only available at runtime. But, using this path expression we are able to predict which blocks of the AXE10 software system are able to take part in a certain link chain. The path corresponding expression is defined to start and end at a node of type `Block`. Furthermore, it is defined that node 1 in the left hand side is the starting node of the path. Accordingly, node 2 is the target node of the path. Blocks 1 and 2 take part in the same link chain, if there is a path from block 1 via a signal to a forlopp action in block 2.

Though the task of extracting link chains from the system's source code seemed to be very difficult in the beginning, this simple static path expression is all we need to automatically perform a static link chain analysis on the structure graph. No additional functionality had to be defined and coded outside the PROGRES specification. Furthermore, no user interaction has to take place to start or end the analysis. Even if

```
static path + lc_next_block : PLEX_Block -> PLEX_Block =
  `1 => `2 in
```




```
end;
```

Fig. 7. Static path expression to establish link chains

the structure graph is consecutively extended with new block subgraphs, these will be analyzed automatically as well. Therefore, this static path expression is a good example how graph transformations can help to reduce complex issues significantly.

5.3 Example 2: Static State Machine Extraction

Each block responsible for the instances in Figure 6 implements at least one *state machine*. As a result, each of the block instances has its own 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. Therefore, extraction of state machines from source code is another hot topic in behavioral analysis.

The basic ideas for the corresponding extraction algorithm have been defined in collaboration with telecommunication experts from Ericsson. The algorithm is based on special coding patterns within the source code. We have then transferred a first informal description of this algorithm into the graph transformation domain.

Figure 8 shows the main transaction for the extraction of a single block's state machines from the structure graph. Consequently, the corresponding block is supplied as input to this transaction. The transaction `Extract_State_Machine` is divided into three parts. First, the set of state variables associated with the given block is determined by calling the transaction `Determine_State_Variables`. The existence of at least one variable that meets the criteria for state variables is a necessary precondition to state machine extraction. If no state variable is found, that is, the set `stateVariables` is empty, then the current block does not implement a state machine.

If there is at least one state variable detected, the second part of the extraction transaction determines the export interface of the block, that is, all signal entries that are reachable from outside the block. Each of the signal entries contributing to the export interface represents a "service" offered to "clients" of a block. Next, a new

```
transformation Extract_StateMachine ( block : PLEX_Block)
  [0:1] =

  use readStateVariables, writtenStateVariables : DATA_ELEMENT [0:n];
      stateVariables : DATA_ELEMENT [0:n];
      exportInterface : PLEX_Signal_Entry [0:n];
      diagramRoot : ROOM_State_Diagram;
      fromState : string;
      toStates : string [0:n]
  do
    Determine_State_Variables ( block, out stateVariables )
    &
    choose
      when not empts ( stateVariables )
      then
          exportInterface := (block.=contains_path=>) : PLEX_Signal_Entry
        & New_State_Diagram ( block, block.Name, out diagramRoot )
        & for all signalEntry := elem ( exportInterface )
          do
              readStateVariables := (signalEntry.=reads_path=>)
            & readStateVariables := (readStateVariables and stateVariables)
            & for all stateVariable := elem ( readStateVariables )
              do
                  IdentifyFromState ( signalEntry, stateVariable, out fromState )
                & IdentifyToStates ( signalEntry, out toStates )
                & InsertTransitions ( diagramRoot, fromState, toStates )
              end
          end
    end
  end
end;
```

Fig. 8. Transaction to extract state machines

state diagram is introduced by adding a corresponding root node to the structure graph, and connecting this node to current block node. This is achieved by calling the graph transformation `New_State_Diagram`.

In the third part, each of the signal entries in the block's export interface is processed separately to extract possible state transitions one by one. According to a design rule at Ericsson, every signal execution should at most trigger a single state change. Therefore, simply speaking, each state change within the statement sequences reachable from a signal entry point can be considered to represent a transition from the starting state in the signal entry to the state specified by the state change.

Consequently, the further procedure is as follows: The set of state variables read within the statement sequence represented by a node of type `Signal_Entry` is identified first. For each of these variables the expected starting state is determined by calling the transaction `Identify_From_State`. Next, the state changes of all control flow paths triggered by this starting state are returned by the transaction `Identify_To_States`. Finally, the transaction `Insert_Transitions` adds new state transitions into the state diagram – one for each pair of the state in `fromState` and one element of the set `toStates`.

Though the initial problem seemed to be only slightly more complex (on paper) as the link chain analysis discussed in Section 5.2, the resulting specification is much

more complex. Instead of a single path definition, several transactions are necessary to implement the state machine extraction algorithm. Furthermore, the extraction has to be initiated by the user of the dynamic analysis tool (which is intentionally). Inspected in more detail, the specification of the state machine extraction algorithm cannot take too much advantage of special features of the PROGRES machinery like automatic path calculation or backtracking. Instead, the extraction algorithm utilizes normal control flow elements known from traditional programming languages. This is a matter of the characteristic of the extraction problem. Here, we are not interested in inserting a single edge but in creating a state machine consisting of named nodes and labeled edges. This cannot be achieved by using a static path expression. Instead, operations for finding state nodes, inserting state nodes and transitions, and the like are needed to perform the task. In addition, a number of non-graph-based operations like attribute value analysis at nodes in the structure graph or set operations are involved. As a result, the mixture of different operations — graph-based and non-graph-based — that contribute to the final goal and that have to be executed in a certain controlled order prevent from taking too much advantage of the PROGRES machinery.

5.4 Example 3: Dynamic Trace Analysis

The last example introduces another important issue in the behavioral analysis of telecommunication systems — trace analysis. As already described in Section 3, traces are lists of runtime events in temporal order. Traces are dynamic information, that is, this information cannot be obtained from the source code. Instead, a runtime inspection of the analyzed software system is necessary.

For trace analysis, a trace file is loaded into the trace simulator. This trace file is then analyzed stepwise — event by event. During the processing of the trace file, an instance graph is created that constitutes a new subgraph of the structure graph. This instance graph is used to present collaboration and sequence diagram like views to the user of the trace simulator tool.

In Figure 9, a transaction is presented that illustrates the connection of a trace to the structure graph as described above. It performs a single step of a trace file. That is, the transaction is used to transform a single signal action in a trace file into corresponding graph transformations. A signal action references a sending block instance, a receiving block instance, and a signal. Furthermore, each signal action is part of a specific trace. This information is again supplied as parameters of the transaction.

The transaction `Trace_Step_Block` is divided into three parts. The first part checks whether the counterparts of the two block instances referenced by a signal action are already elements of the current structure graph. If not, the transaction `Add_or_Get_Node` inserts appropriate block nodes into a special part of the structure graph and annotates them as formerly missing. This enables a user on the one hand to use the trace animation tool without having an appropriate structure graph. On the other hand, the separation of formerly missing parts in a special part of the structure graph gives quick access to this kind of inconsistencies and allows controlled corrections.

The second part of the transaction first switches the execution status of the block instance currently working to `suspended`. Next the node that corresponds to sending block instance is either added to or fetched from the current trace graph. The execution

```
transaction + Trace_Step_Block( traceId : string ; senderBlockName : string ;
                                senderInstanceId : string ; signalName : string ;
                                executionNumber : integer ; receiverBlockName : string ;
                                receiverInstanceId : string )
  [0:1] =
  use
    senderBlock, receiverBlock : Block
    (* ... and other local declarations *)
  do
    (* ------------------------ changes to structure graph ------------------------ *)
      Add_or_Get_Node ( Block, senderBlockName & "UPROGRAM", out senderBlock )
    & Add_or_Get_Node ( Block, receiverBlockName & "UPROGRAM", out receiverBlock )
    & Add_or_Extend_Block_Communication ( senderBlock, receiverBlock, signalName,
                                          out communicationEdge )
    & Activate ( senderBlock, receiverBlock, communicationEdge )
    (* -------------------------- changes to trace graph ------------------------- *)
    & Suspend_Working_Instance ( traceId, signalName )
    & Add_or_Get_Block_Instance ( traceId, senderBlock, senderInstanceId,
                                  out senderInstance )
    & Switch_ExecStatus_Of_Instance ( senderInstance, "WaitingOrSuspended" )
    & Add_or_Get_Block_Instance ( traceId, receiverBlock, receiverInstanceId,
                                  out receiverInstance )
    & Switch_ExecStatus_Of_Instance ( receiverInstance, "Working" )
    & Insert_Block_Instance_Communication ( traceId, senderInstance, receiverInstance,
                                            signalName, out instanceCommunicationEdge )
    & executionId := ("[" & string ( executionNumber ) & "] " & signalName)
    & instanceCommunicationEdge.SignalExecutionOrder :=
             (instanceCommunicationEdge.SignalExecutionOrder or executionId )
    (* ------------------------ changes to structure grap ------------------------- *)
    & currentSubsystem := senderBlock.=contained_in_subsystem_path=>
    & Propagate_Activation ( currentSubsystem )
    & receiverSubsystem := receiverBlock.=contained_in_subsystem_path=>
    & Propagate_Activation ( receiverSubsystem )
    & Propagate_Activation ( communicationEdge )
  end
end;
```

Fig. 9. Transaction to support animation of traces

status of this block instance is either set to `waiting` or it is set to `suspended`. The value depends on the kind of signal action processed (combined signal or single signal). The corresponding information is obtained by querying the structure graph. This procedure is repeated for the receiving block instance node. But, its execution status is set to `working`. After the block instance communication edge between the sending and the receiving block instance has been inserted, an execution identifier for the current signal is calculated. This identifier is added to the attribute `SignalExecutionOrder` of the communication edge. Finally, in the third part of the transaction, the information about changes of the execution status of parts of the structure graph and trace graph is propagated from the block level to the more abstract levels (subsystems etc.).

Considering the trace simulation tool as a whole, this transaction represents just a small part of its implementation. The main control part does not even reside within the graph specification. Instead, the contribution of the graph transformation machinery is limited to “simple” database-like activities. The core of the trace processing and analysis is performed within hand-written Java code that is integrated as a module into the UPGRADE framework. One reason for this way of implementing the trace analysis is the difficulty in embedding non-graph-transformational functionality like parsers into PROGRES specifications. But, more striking is the necessity to create very complex temporary data structures while processing a trace file.

In general, it is possible to import additional functionality into PROGRES specifications through external functions. Furthermore, complex data structures (e.g., records) can be represented through equivalent graphs. But, it is more suitable to implement the corresponding functionality, which does not profit from the strengths of a graph machinery, in C, C++, or Java programs (to name just some programming languages).

In the case of UPGRADE and PROGRES, the Java-part can access and manipulate the graph via a special interface, but not vice versa. All transformations specified in the specification can be accessed. That is, the PROGRES graph machinery is the passive part while the UPGRADE framework provides the active part.

Summarizing the findings of the three examples, there is a trade-off in the proportion of graph specification regarding the implementation of the different behavioral analysis algorithms. The more the solution of the initial problem is suited for the graph transformations domain, the higher is the share solved within the specification. But, the more the solution requires direct control, complex data structures, or an increasing amount of user interaction, the more of the realization will take place outside the graph transformation system. But, this is not a disadvantage in general. An appropriate combination of both worlds — graph transformation and traditionally programmed systems — resulted in flexible, manifold, and efficient solutions to our behavioral analysis problems.

6 Related Work

In contrast E-CARES, much work has been performed in business applications written in COLBOL. The corresponding tools such as e.g. Rigi [9] or GUPRO [10] primarily focus on the static system structure. Moreover, they are typically data-centered; consider e.g. [11, 12]. Here, recovery of units of data abstraction and migration to an object-oriented software architecture play a crucial role [13]. More recently, reengineering has also been studied for object-oriented programming languages such as C++ and Java. E.g., TogetherJ or Fujaba [14] 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.

Graphs and graph rewriting systems play an essential role in E-CARES. In the following, we examine reengineering tools from a tool builder's perspective. We compare E-CARES to a set of other graph-based reengineering tools.

Rigi [9] is an interactive toolkit with a graphical workbench which can be used for reengineering. Internally, a software system is represented by a set of entities and relationships. Externally, program understanding is supported by graph visualization techniques. Rigi is also used in the *Bauhaus* project [15], whose aim is to develop methods and techniques for (semi-)automatic architecture recovery, and to explore languages to describe recovered architectures. In contrast to E-CARES, Rigi (and thus Bauhaus) is not based on a high-level specification language. Rather, graphs are accessed through a procedural interface, which makes coding of graph algorithms more painstaking.

The *GUPRO* project [10] is concerned with the development of a generic environment for program understanding. Internally, programs are represented as graphs. GUPRO offers parsers for several languages, including COBOL and C. Different kinds of analyzes may be specified with the help of a graph query language, but graph trans-

formations cannot be specified in a declarative way. Moreover, GUPRO offers a textual user interface, while E-CARES provides graphical tools.

VARLET [16] addresses the problem of database reengineering. In VARLET, a relational schema is transformed into an object-oriented one; triple graph grammars [17] provide the underlying theoretical foundation. VARLET tools have been specified and implemented with the help of PROGRES. E-CARES addresses a different application domain. Up to now, we only address reverse engineering, i.e., the system under study is not modified. For the re-design of telecommunication systems, we intend to use triple graph grammars, as well, as we did in a previous project [18].

FUJABA [14] is a CASE tool for UML which supports round-trip engineering from UML to Java and back again. While reverse engineering of class diagrams is well understood and implemented in a couple of commercial tools (e.g., Together and Rational Rose), Fujaba also recovers collaboration diagrams from Java source code. In Fujaba, collaboration diagrams are not used as examples of computations; rather, they are executable and may be considered as generalizations of graph transformations. The E-CARES prototype is based on a different specification language (PROGRES), addresses PLEX rather than Java programs, and deals with the application domain of telecommunication systems, which has not been considered in Fujaba so far.

7 Conclusion

We have presented the E-CARES prototype for reengineering of telecommunication systems, which is based on a programmed graph rewriting system acting as an operational specification from which code is generated. In this paper, we have provided insight into the behavioral analysis part of this specification. Reengineering is an application domain which is well-suited for graph rewriting. But, only the combination of graph transformation and traditional programming resulted in an appropriate analysis tool. There are other reengineering tools which are also based on graphs, but most of them lack the support provided by a high-level specification language and force the tool developer to perform low-level programming.

So far, the implementation supports only reverse engineering, i.e., it aids in system understanding, but not yet in system restructuring. To date, approximately 2.5 million lines of PLEX code have successfully been parsed and analyzed by means of the E-CARES prototype. Current and future work addresses (among others) the following topics: reverse engineering of state diagrams (testing and improvement), application of metrics, multi-language support (e.g., C or SDL in addition to PLEX), improvement and extension of the ROOM architectural description language support, re-design, and source code transformation.

References

1. Marburger, A., Herzberg, D.: E-CARES research project: Understanding complex legacy telecommunication systems. In: Proc. 5th European Conference on Software Maintenance and Reengineering, IEEE Computer Society Press (2001) 139–147

2. Kazman, R., Woods, S.G., Carrière, J.: Requirements for integrating software architecture and reengineering models: CORUM II. In: Working Conference on Reverse Engineering, IEEE Computer Society Press (1998) 154–163
3. Marburger, A., Westfechtel, B.: Tools for understanding the behavior of telecommunication systems. In: Proceedings 25th International Conference on Software Engineering ICSE 2003, IEEE Computer Society Press (2003) 430–441
4. Schürr, A., Winter, A., Zündorf, A.: The PROGRES approach: Language and environment. [19] 487–550
5. Jäger, D.: Generating tools from graph-based specifications. Information Software and Technology **42** (2000) 129–140
6. Marburger, A., Westfechtel, B.: Graph-based reengineering of telecommunication systems. In: Proceedings international conference on graph transformations ICGT 2002. LNCS 2505, Springer (2002) 270–285
7. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object-Oriented Modeling. Wiley & Sons (1994)
8. Ellsberger, J., Hogrefe, D., Sarma, A.: SDL - Formal Object-oriented Language for Communicating Systems. Prentice Hall (1997)
9. Müller, H.A., Wong, K., Tilley, S.R.: Understanding software systems using reverse engineering technology. In: The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Scienes (ACFAS). (1994)
10. Kullbach, B., Winter, A., Dahm, P., Ebert, J.: Program comprehension in multi-language systems. In: Proc. 4th Working Conference on Reverse Engineering, IEEE Computer Society Press (1998)
11. Markosian, L., Newcomb, P., Brand, R., Burson, S., Kitzmiller, T.: Using an enabling technology to reengineer legacy systems. Communications of the ACM **37** (1994) 58–70
12. van Zuylen, H.J., ed.: The REDO Compendium: Reverse Engineering for Software Maintenance. John Wiley & Sons: Chichester, UK (1993)
13. Canfora, G., Cimitile, A., Lucia, A.D., Lucca, G.D.: Decomposing legacy systems into objects: An eclectic approach. Information and Software Technology **43** (2001) 401–412
14. Zündorf, A.: Rigorous Object-Oriented Development. PhD thesis, University of Paderborn (2002) Habilitation thesis.
15. Koschke, R.: Atomic Architectural Component Recovery for Program Understanding and Evolution. PhD thesis, Institute of Computer Science, University of Stuttgart (2000)
16. Jahnke, J., Zündorf, A.: Applying graph transformations to database re-engineering. [19] 267–286
17. Schürr, A.: Specification of graph translators with triple graph grammars. In: Proceedings WG '94 Workshop on Graph-Theoretic Concepts in Computer Science. LNCS 903, Springer (1994) 151–163
18. Cremer, K., Marburger, A., Westfechtel, B.: Graph-based tools for re-engineering. Journal of Software Maintenance and Evolution: Research and Practice **14** (2002) 257–292
19. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools. Volume 2. World Scientific (1999)