Graph-Based Structural Analysis for Telecommunication Systems

André Marburger[1] and Bernhard Westfechtel[2]

[1] DSA Daten- und Systemtechnik GmbH, Pascalstr. 28, D-52076 Aachen
andre.marburger@gmx.de
[2] Applied Computer Science I, University of Bayreuth, D-95440 Bayreuth
bernhard.westfechtel@uni-bayreuth.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. The E-CARES project is dedicated to reverse engineering of complex legacy telecommunication systems by providing graph-based tools. With E-CARES, the software architecture of a telecommunication system is recovered in two steps. In the first step (program analysis), the source code is parsed to build a structure graph which uses the abstractions of the underlying programming language and describes the internals of program units (blocks) as well as their communication via exchange of signals. In the second step, a software architecture description is abstracted from the structure graph. The software architecture is described in ROOM, a real-time object-oriented modeling language for embedded systems design. Both program analysis and architecture recovery are based on graphs and graph transformations. In both steps, domain-specific knowledge — referred to as methods of use — is exploited which refers to the ways how language constructs are used to realize processing concepts of telecommunication systems.

Keywords: Reverse Engineering, Structural Analysis, Telecommunication System, Graph Transformation.

1 Introduction

Reengineering of large and complex software systems has proved a difficult task. According to the "horseshoe model of reengineering" [1,2], 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

G. Engels et al. (Eds.): Nagl Festschrift, LNCS 5765, pp. 363–392, 2010.

COBOL — to an object-based or object-oriented architecture [3,4]. 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 [5].

Reengineering of *process-centered* applications has been addressed less extensively so far [6]. 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, RWTH Aachen University, 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 object 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 urgent need for tool support to improve program evolution and maintenance.

This paper deals with the *reverse engineering environment* developed within the E-CARES project, as presented comprehensively in [8]; see [9] for restructuring. Moreover, we confine the presentation to *structural analysis*, i.e., recovery of the structure of the system under study. For the sake of modularity and reuse, structural analysis is decomposed into two major steps. The first step, which builds a suitable abstraction of the source code, is called *program analysis*. The resulting representation still depends on the underlying programming language (PLEX, a proprietary language which was developed at Ericsson in the 1970s). In contrast, the second step — *architecture recovery* — delivers a representation in a modeling language which is not specific to the programming language any more. For architecture recovery, we selected *ROOM* [10] as the target language, since it was applied extensively to the development of telecommunication systems and — by and large — provides appropriate concepts and abstractions for this domain (see [11] for further extensions to ROOM to improve the modeling of telecommunication systems). However, we could have alternatively used another language, e.g., SDL [12] or the UML component model [13].

To recover meaningful abstractions of the program source code, *domain-specific knowledge* is exploited heavily in structural analysis. In the context of the E-CARES project, this knowledge was summarized under the term *methods of use*: Telecommunication experts at Ericsson use the programming language and the runtime system in systematic ways in order to realize processing concepts

[1] *E*ricsson *C*ommunication *AR*chitecture for *E*mbedded *S*ystems [7].

and paradigms. For example, the AXE-10 system follows a signaling paradigm of processing, where instances of blocks (program units in PLEX) cooperate to operate calls by passing signals along a link chain between the originator of a call and its receiver. The way how the signaling paradigm is mapped onto PLEX results in domain-specific code patterns which are subsequently transformed into design patterns at the architectural level.

The E-CARES reverse engineering environment is *graph-based*, since recovered structures are represented as graphs. Graphs serve as a general, natural, and expressive data model which is frequently used in many other reengineering environments, as well. However, E-CARES differs from other reengineering environments because it relies on *graph transformations*: Large parts of the E-CARES environment are generated from a formal specification based on a graph transformation system. In this way, the effort for developing the reverse engineering environment has been reduced significantly.

The remainder of this paper is structured as follows: Section 2 provides some background knowledge from the telecommunication domain. Section 3 gives a brief overview of the E-CARES environment. Sections 4 and 5 are devoted to program analysis and architecture recovery, respectively. Section 6 is concerned with the realization of the E-CARES environment. Section 7 discusses related work. Finally, Section 8 concludes the paper.

2 Background

2.1 GSM Basics

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 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

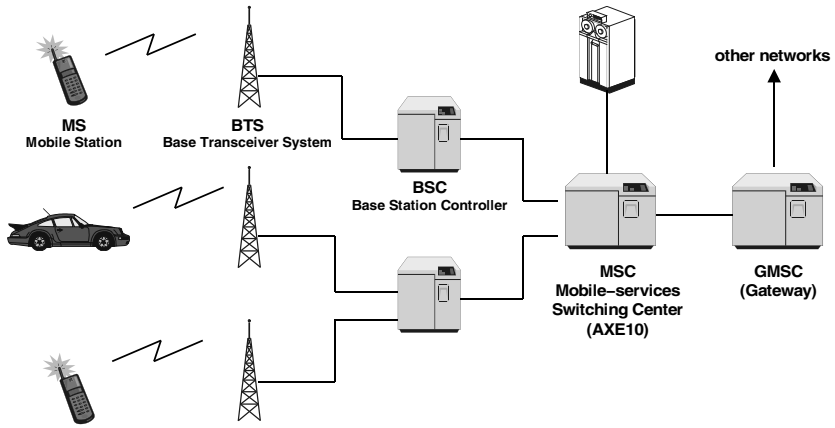


Fig. 1. Simplified sketch of a GSM network

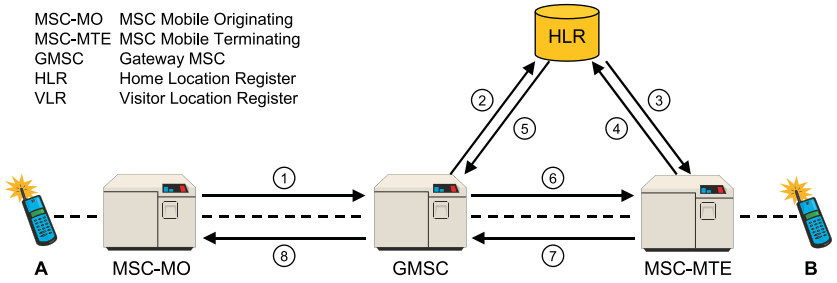


Fig. 2. Mobile originating call

requested routing information to the GMSC. After that, the GMSC (6) sends a call request to the MSC-MTE. The MSC-MTE (7) returns an address complete message (ACM) which (8) is forwarded to the MSC-MO. Now, user data — e. g., the two participants' voices — may be transferred between A and B as indicated by the dashed line in the figure.

2.2 The Programming Language PLEX

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

A PLEX program is composed of a set of *blocks* (designated by the keyword DOCUMENT) which serve as units of compilation. Blocks have data encapsulation, that is, a block's data cannot be accessed by other blocks. PLEX has a COBOL-like syntax with blocks being divided into multiple sectors. Only two of these are of interest within the scope of this paper, namely the *declare sector* for the declaration of local variables, and the *program sector* for the control logic. As an example, Figure 3 shows cutouts of several blocks being involved in the processing of a mobile originating call as illustrated in Figure 2.

The *declare sector* defines variables for elementary data types such as e.g. integers, strings, and symbols, as well as structured data such as arrays and records. Data may be marked as persistent by a corresponding keyword in the declaration. A *record* which is marked as persistent stands for a file storing instances of records of this type. Typically, a record instance holds data for a specific call. For example, the record MSCMO in block MSCMO holds — among other data — the telephone number of the telephone of the B side, the so-called BNUMBER.

Blocks communicate via *signals* which are declared in separate signal description tables (not shown). The SEND statement serves to send a signal. In the case of a *unique signal*, the recipient is not specified as it may be identified by a simple name match of the signal name in the reception statements of other blocks. In the case of a *multiple signal*, the SEND statement contains a reference to a variable which stores the recipient at runtime. Here, the name match alone is ambiguous as it only provides possible pairs of senders and receivers. Data may be transferred along with a signal via the WITH clause. By default, signals realize asynchronous communication (*single signals*). In the case of synchronous communication (*combined signals*), the SEND statement specifies the set of alternative signals which are expected as backward signals (WAIT FOR clause). Single and combined (forward) signals are received by ENTER and RECEIVE statements, respectively. In the latter case, backward signals are sent via RETURN statements and received by RETRIEVE statements.

The control flow of a block is described in its program sector. Execution starts with the reception of a signal and continues until an EXIT statement is reached. Typically, a GOTO statement is executed in response to a signal which transfers the control flow to a *labeled statement sequence*. A DO statement calls a parameter-less *subroutine* which is embraced by BEGIN and END statements. Parameters can be passed only with *local signals* which are processed by the sending block itself. This is achieved via a TRANSFER statement which performs an unconditional jump and optionally passes parameters to the target code fragment (starting with an ENTRANCE statement).

As presented so far, a block is a fairly unstructured program unit. Unfortunately, PLEX does not offer language constructs to decompose blocks into more fine-grained logical units. At a conceptual level, however, such units do exist (and need to be recovered by structural analysis). First, since execution is driven by signals, the code of the program sector may be decomposed into *functions*, i.e., pieces of code being executed in response to some signal. Second,

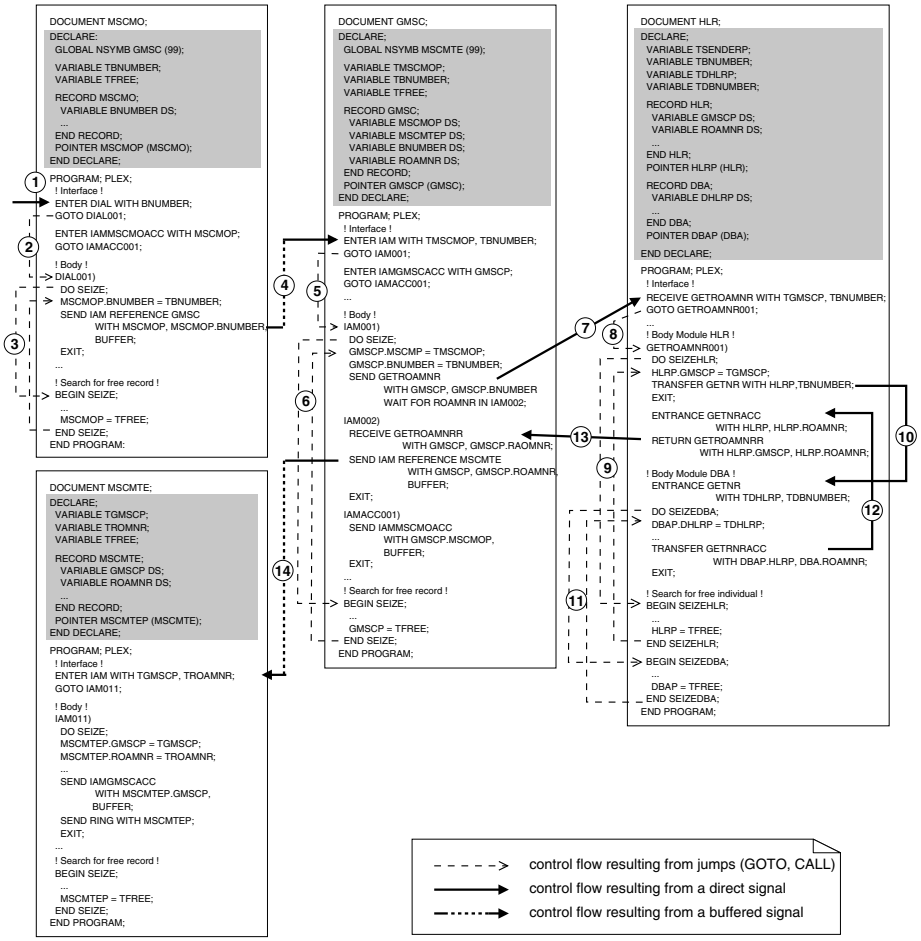


Fig. 3. PLEX blocks and their communication

multiple functions may be aggregated into *modules*, which provide logically coherent "mini-services". For example, the block HLR consists of two modules — the main module HLR and the subordinate module DBA (data base access). The latter serves as a wrapper that provides a standardized interface to a specific physical database to decouple the application from a specific database implementation.

2.3 Execution Model

Program systems written in PLEX must be run on top of an operating system whose main task is signal handling. Single signals are managed in priority queues called *job buffers*. The runtime system selects a signal from a queue and delivers it

to the receiving block. *Direct signals* (for local and synchronous communication) are executed immediately rather than stored in job buffers. Within a block, memory is usually managed statically. If resources are exhausted, further space may be allocated via size alteration events.

At runtime, telecommunication systems handle thousands of different telephone calls at the same time. Each call — as e.g. illustrated in Figure 2 — represents a service request to the telecommunication system. The numerous services are realized by re-combining and re-connecting sets of small services, provided by respective blocks. Thus, a telephone call is realized by numerous mini-services spontaneously linked together. Telecommunication experts call the interconnected services for a single telephone call a *link chain*.

Conceptually, a link chain is composed of a set of interconnected *block instances*, each of which holds data for a specific call. Since PLEX does not provide language support for dynamic instantiation of blocks, block instances have to be managed "manually" by the respective block, i.e., the PLEX programmer has to simulate dynamic instantiation by writing code for allocating and deallocating memory units, etc. If a block consists of multiple modules, instantiation has to be performed at the module level, resulting in module instances. For the sake of simplicity, however, we will use the term "block instance" in the description below.

Figure 3 shows a cutout of the processing of a mobile originating call as an example for the dynamic creation of a link chain. The figure covers part of the mobile originating call scenario of Figure 2, namely those steps in which the blocks MSC-MO, GMSC, and HLR are involved. Processing starts with the reception of a DIAL event (1), which causes a jump to a labeled statement sequence for processing this event (2). An idle MSC-MO record is occupied through the call of the subroutine SEIZE (3). After that, the signal IAM is sent to block GMSC (step (4), which corresponds to step 1 in Figure 2). The recipient in turn jumps to a labeled statement sequence (5) and allocates a record for the link chain (6). Subsequently, it sends a request for a roaming number to block HLR (step 7, corresponding to (2) in Figure 2). After a jump to the processing labeled statement sequence (8) and seizing of a record for the new call (9), a local signal is sent (10) which triggers the allocation of another record in the subordinate module DBA (11). Then, another local signal is sent (12) whose associated data contains the roaming number for the call. The roaming number is passed back to GMSC (step 13, corresponding to (5) in Figure 2), from which it is finally forwarded to MSC-MO (step 14, corresponding to (8) in Figure 2).

3 The E-CARES Environment

Within the E-CARES project, a *reengineering environment* for telecommunication systems has been developed. The system architecture of the E-CARES environment is illustrated in Figure 4. In this paper, we are concerned only with reverse engineering (solid lines). The components displayed with dashed lines deal with restructuring and forward engineering; see [9].

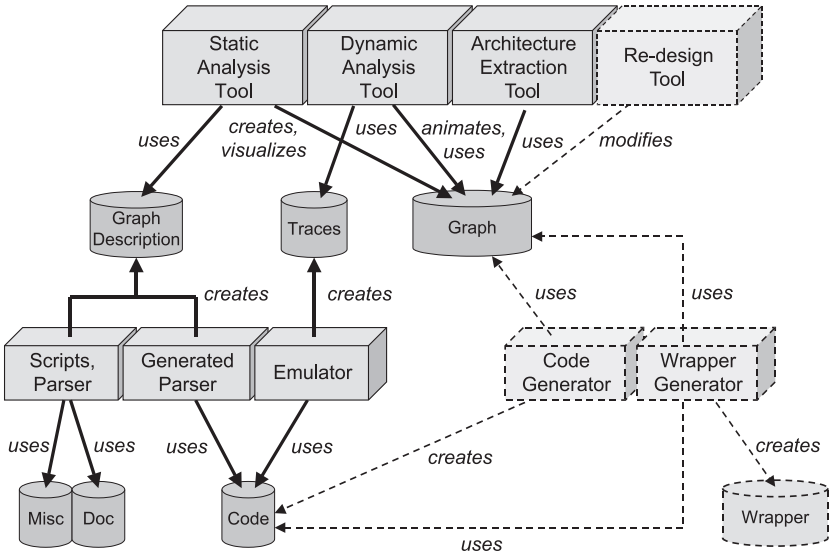


Fig. 4. Architecture of the E-CARES environment

Reverse engineering involves different kinds of analysis: *Structural analysis* — the focus of this paper — refers to the static system structure, while *behavioral analysis* is concerned with the system's 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 (or structure document, respectively), while *dynamic analysis* requires information from program execution. Thus, "static" and "dynamic" refer to the inputs of analysis.

For the static analysis of the structure of a PLEX system, we base our system on three sources of information. The first one is the *source code* of the system. It is considered to be the core information as well as the most reliable one. Via code analysis (parsing) a 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, signal database) and the system *documentation*. As far as the information from these sources is computer processable, we use parsers and scripts to extract additional information. This additional information is stored in structure documents, as well.

The *static analysis tool* processes the graph descriptions of individual blocks and extends the structure graph which represents the overall application by creating corresponding subgraphs. The subgraphs are then 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. 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).

The graph produced by the static analysis tool depends on the programming language (PLEX) in which the source code is written. Therefore, we will use the term *program analysis* (Section 4) to denote this step of processing. The *architecture extraction tool* transforms the structure graph into elements of a modeling language which does no longer use abstractions specific to the programming language. This results in an architecture graph that can be used to review and (re-)document the currently implemented architecture or to perform architectural changes. As modeling language, we selected ROOM; other languages — e.g., SDL or the UML component model — could be supported in an analogous way (see also Section 7 on related work).

Dynamic information originates from using an emulator or querying a running AXE10, resulting in a list of events plus additional information in a temporal order in both cases. 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 software architects 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. In E-CARES, dynamic analysis is performed only for recovering the behavior; therefore, it goes beyond the scope of this paper. For further information on dynamic analysis, the reader is referred to [8,14].

4 Program Analysis

Program analysis builds a structure graph which still depends on the underlying programming language PLEX. However, program analysis does not exploit only the syntactic structure of PLEX programs. In addition, it takes advantage of *methods of use*, i.e., coding conventions and programming patterns which are applied by software developers at Ericsson. These methods of use constitute indispensable expert knowledge which assists in building program representations with appropriate domain-specific abstractions.

4.1 Building the Structure Graph

The *structure graph* is created by parsing diverse sources. The most important information is the source code of PLEX blocks. In addition, global signals are declared in signal description tables. Furthermore, information about the coarse-grained system structure above the level of blocks is given in product configuration files. In the following, we will primarily focus on parsing of PLEX blocks.

The design of the structure graph was driven heavily by the requirements of telecommunication experts at Ericsson. From the very beginning, the experts were more interested in the coarse-grained structure of the system under study rather than in detailed code analysis. Therefore, the structure graph contains

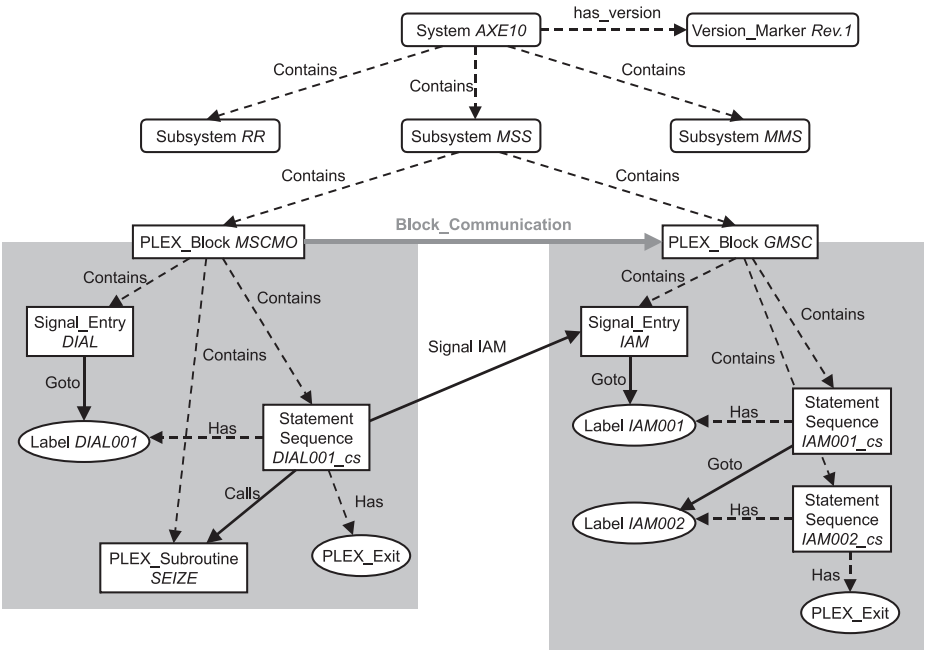


Fig. 5. Example of a structure graph

only the information which is required for the purposes of reverse engineering. In particular, the structure graph has to reveal the interfaces of blocks (processed signals), the actions performed in response to received signals, and the inter-block communication.

For this reason, the structure graph is more compact than an *abstract syntax graph*, which would provide a detailed syntactic representation of the source code. From the structure graph alone, it would not be possible to reconstruct the source code. However, the structure graph contains references to text lines in the source code. Thus, for each element of the structure graph the source code fragment may be retrieved from which this element was derived.

An example of a (cut-out of a) structure graph is presented in Figure 5. Partially, this structure graph is created from the code fragments shown in Figure 3. The nodes represented by rounded rectangles on the top do not correspond to language constructs of PLEX. Rather, they are created by parsing a configuration file which decomposes the AXE10 system into subsystems and blocks (as leaves of the composition hierarchy).

The PLEX parser processes each block independently, resulting in a block graph (shaded regions) which is embedded into its context in a subsequent phase. The figure shows only parts created from the program sector. In addition, a block graph contains further elements created from other sectors, e.g., nodes for variable declarations.

The program sector is represented in such a way that the computational behavior of blocks is reflected. Thus, it contains nodes for *signal entries*, which constitute the export interface of a block. The coding conventions at Ericsson state that a GOTO statement — represented by an edge — is executed in response to the signal reception. Its target is a *label* (nodes displayed as ellipses) which marks the start of some *statement sequence*. Within this sequence, subroutines may be called. Furthermore, a statement sequence may contain an EXIT statement which returns control to the dispatcher of the operating system.

The notion of *labeled statement sequence* constitutes an example of the methods of use mentioned above. A labeled statement sequence does not correspond to a syntactic unit of the PLEX programming language. Rather, a labeled statement sequence is defined as a code fragment starting with some label and ending before the next label. Labeled statement sequences constitute a useful abstraction which reflects the way in which PLEX programs operate: They execute statement sequences in response to signals. The internals of these sequences are not important; therefore, they are not represented in the structure graph.

After building initial subgraphs for each block, several steps of *postprocessing* have to be performed. For example, the *control flow* has to be completed by postprocessing. Like in COBOL, execution of a labeled statement sequence may *fall through* to a consecutive statement sequence if the preceding sequence is not terminated by an EXIT statement. In this case, the labeled statement sequences are connected by a control flow edge. This kind of processing is a prerequisite for the recovery of functions to be discussed in the next subsection.

Furthermore, after initial local analysis blocks are isolated and need to be connected by *binding* global signal sending statements to signal entry points. Local analysis prepares the binding by creating virtual signal entry points acting as temporary target nodes (not shown in Figure 5). Global analysis searches for matching entry points in other blocks, creates inter-block edges for representing the sending of signals, and removes the temporary virtual signal entry points. For example, this binding step inserts an edge from the labeled statement sequence DIAL001 in block MSCMO to the signal entry point IAM in block GMSC. In the case of single signals, the binding is unique. In the case of multiple signals, the actual receiver is determined only at run time; static analysis creates an edge to the entry point of each potential receiver.

In the binding step, it is crucial to take the following method of use into account: In AXE10, each block is responsible for returning control to the operating system within a maximal time slice. For efficiency reasons, there is no interrupt handling built into the operating system. But the processing of an incoming signal may consume more than one time slice. Thus, the processing is divided into multiple phases where each but the last phase sends a CONTINUE signal to the block itself and terminates. This mode of processing is known as *phase division*.

CONTINUE signals have to be excluded from the binding step; otherwise, the structure graph would be cluttered with many erroneous edges (recall that in the case of multiple signals edges are created to all potential receivers). Rather, these signals are handled specially by a phase division analysis. All phases which

are executed in response to a certain signal are connected by control flows. In this way, artificially separated computation fragments are joined together.

After having created connections between signal sending statements and single entry points, the communications are *lifted* to the block level. For example, lifting results in the block communication edge from MSCMO to GMSC. Furthermore, lifting is continued to the subsystem level by connecting subsystems with mutually communicating blocks.

4.2 Recovering Functions and Modules

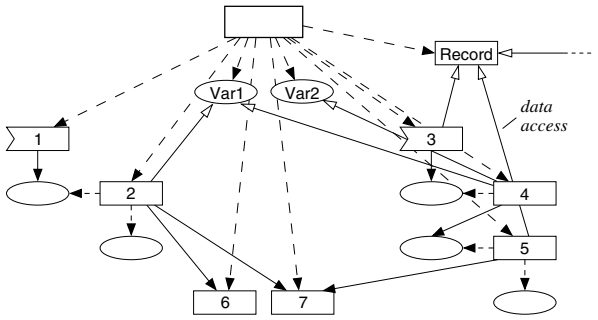
Functions and modules are conceptual abstractions rather than syntactic units of the PLEX programming language. Several methods of use are exploited to recover these abstractions from the structure graph built up so far. Since the recovery of modules is concerned with the grouping of functions, the recovery of functions is described first.

Recovery of Functions. A *function* represents the set of code fragments which are uniquely executed as a reaction to the reception of a certain signal. Consequently, each function contains exactly one signal entry. The remaining nodes of the structure graph that belong to a function are determined through control flow analysis.

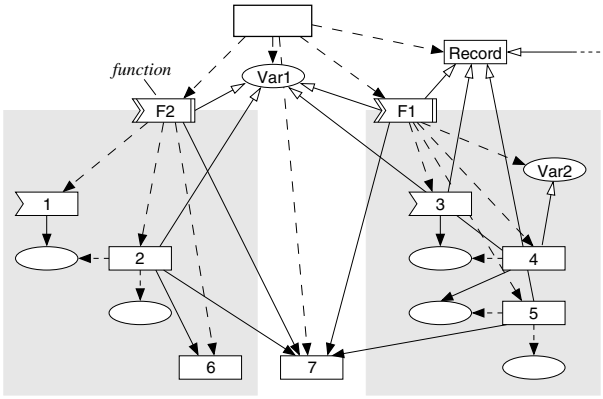
Figure 6 illustrates the recovery of functions (and modules, see below) by an example. In this example, we start with a block structure graph comprising two signal entries (nodes 1 and 3), three labeled statement sequences (nodes 2, 4, and 5), and two subroutines (nodes 6 and 7). Furthermore, some data elements, one record and two atomic data elements (Var1 and Var2), are depicted. The further structuring of the record has been omitted for the sake of brevity. Therefore, the data access edges, represented by arrows with open heads, normally accessing some data element in the record have been redirected to the record node.

For function recovery, all graph elements are determined that can be reached from the given signal entry via a transitive closure of control flow edges. The preceding postprocessing steps ensure that the control flow is represented completely in the structure graph; consider e.g. processing of fall-through execution and phase divisions as explained in the previous subsection. With respect to the example in Figure 6, the transitive closure returns a set consisting of nodes 2, 6, and 7 when starting at signal entry 1. Furthermore, all data elements accessed from these nodes are determined. For example, in the case of signal entry 1 the variable Var1 is accessed. Starting from signal entry 3, we obtain the variables Var1 and Var2 as well as the record.

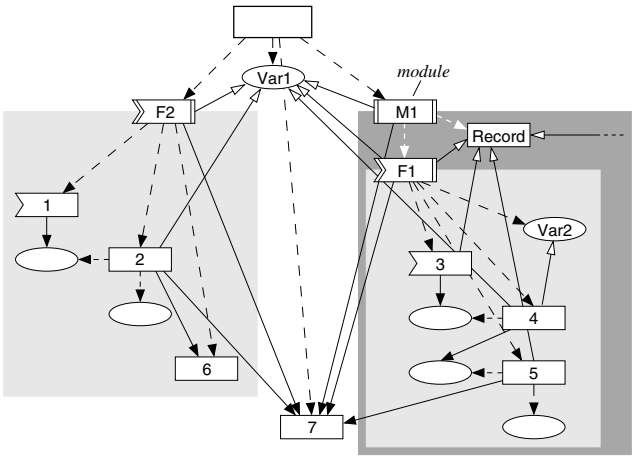
Next, graph elements shared by at least two transitive closures are removed from the respective sets in order to obtain only uniquely assignable graph elements. In Step 2 of Figure 6, all uniquely assignable graph elements are shown in regions with a light gray background. The subroutine 7, the variable Var1, and the record — which is also accessed from another function not displayed in the figure — are located outside these regions because they are shared by multiple



Step 1: Structure graph after embedding, correction, and completion



Step 2: Functions detected and created



Step 3: Module detected and created

Fig. 6. Effects of detection of functions and modules on a structure graph

functions. On the other hand, subroutine 6 and variable Var2 have been identified as local elements.

The graph shown in Step 2 also includes the function nodes (F1 and F2) which have been created to represent the results of the recovery in the structure graph. These nodes are inserted in between the block node and the nodes representing local graph elements. Furthermore, outgoing control flow and data flow edges have been lifted to the function node. In this way, the "imports" of each function are represented.

Recovery of Modules. Recovery of functions serves to identify the *interface* of a block. As already mentioned at the end of Subsection 2.2, blocks may be decomposed into more fine-grained units called *modules*. A module groups data and functions. For example, the block HLR consists of two modules — the main module HLR and the subordinate module DBA (data base access).

As we have explained in Subsection 2.3, telephone calls are handled by building up link chains. A link chain is composed from instances of participating blocks. If a block is decomposed into multiple modules, instantiation has to be performed on the level of modules. For each instance, its data are typically held in a single record.

This domain knowledge is exploited in *module recovery*. In contrast to function recovery, which is based on control flow analysis, module recovery analyzes accesses to data elements. However, we do not consider all data elements declared in a block. Rather, we confine the scope of analysis to records holding instance data. These records are discovered by applying some heuristic rules described below.

A record indicating that there is a module contained in a PLEX block satisfies certain requirements. First of all, instance records are stored records. They are stored in a file and are referenced by a similarly named pointer variable. In addition, instance records have to contain an enumeration type variable defining a list of symbolic state values. According to a coding rule, the name of this variable contains the substring STATE. Therefore, a string match is sufficient for its identification. Another good hint that a record is an instance record is its size. Instance records tend to be much larger than other records in a block.

Since modules are grouped around an instance record, we use a data centered algorithm for the calculation of the module candidate. A well-known algorithm of this class has first been proposed by Liu and Wilde in [15]. An adaption of this algorithm to graphs can be found in [16]. Originally, the idea of the algorithm is to identify *object candidates* in procedural code. The idea bases on the observation that software has already been developed in an object-based manner long before object-oriented design has become popular. Thus, it is assumed that object candidates in a conventional programming language can be identified as a collection of routines, (global) data types and/or (global) data elements. That is, an object candidate is a triple (F, T, D), where F is a set of functions, T a set of data types, and D a set of data elements.

Adapted to the facts of E-CARES, object candidates conform to module candidates. Furthermore, the set of data types will always be empty as there are

no user defined data types in PLEX. Also, the data elements used as an input to the algorithm can be limited to records only. Therefore, the procedure of the module candidature for PLEX code according to Liu and Wilde reads as follows:

1. *For each instance record x, let $P(x)$ be the set of routines which directly use x. Moreover, x should be shared by at least two different routines. Routines can correspond to functions (if present[2]) but also to any other kind of multi-statement structure object inside a block (signal entry, statement sequence, subroutine).*
2. *Considering each $P(x)$ as a node, construct a graph $G = (V, E)$ such that*

$$V = \{\ P(x) \mid x \text{ is shared by at least two routines}\} \quad \text{is a set of nodes,}$$

$$E = \{\ \overline{P(x_1)P(x_2)} \mid P(x_1) \cap P(x_2) \neq \emptyset\} \quad \text{is a set of undirected edges.}$$

3. *Construct a module candidate (F, T, D) from each strongly connected component $G_{cand} = (V_{cand}, E_{cand})$ in G such that*

$$F = \bigcup_{P(x) \in V_{cand}} P(x)$$

$$T = \emptyset$$

$$D = \bigcup_{P(x) \in V_{cand}} \{x\}$$

4. *Complete each module candidate by adding all data elements which are accessed only locally.*

Applied to the graph shown in Step 2 of Figure 6, the algorithm selects the only record as its starting point (assuming that the heuristic rules described above are satisfied). This record is accessed by the function F1 and another function not shown in the figure (say F3). The module candidate comprises the selected record, the functions F1 and F3 (Step 3).

Please note that in general a module may comprise multiple records (e.g., if F3 accesses another record matching the heuristic rules). However, in the case of the AXE10 system each module is typically grouped around a single record.

What would have happened if we had neglected the domain knowledge on the realization of modules and if we had just used the Liu and Wilde algorithm as proposed? In that case, Step 1 of the algorithm would refer to all global data elements and any kind of routines. By consequence, the algorithm would determine two sets $P(\mathsf{Var1}) = \{2, 3, \mathsf{F1}, \mathsf{F2}\}$ and $P(\mathsf{Record}) = \{3, 4, \mathsf{F1}, \mathsf{F3}\}$. Because these two sets have a non-empty intersection, the corresponding module candidate would comprise the contents of both $P(\mathsf{Var1})$ and $P(\mathsf{Record})$. Thus, all elements of the sample graph would be assigned to a single module.

[2] If function recovery has not been performed, the algorithm will work anyway.

5 Architecture Recovery

5.1 Motivation

So far, we have been concerned with the creation of a structure graph from the source code and other auxiliary documents. The structure graph strongly depends on the syntax of the underlying programming language (PLEX). In addition, it contains conceptual abstractions such as functions and modules which do not occur as syntactic units in the PLEX language.

Architecture recovery creates an architectural description of the system under study from the structure graph. The underlying *architecture description language* (ADL) has to meet the following requirements:

- The ADL has to be independent from the programming language(s) in which the system is written. In particular, language independence is important when different parts of the system under study are written in different programming languages. In the case of AXE10, which has originally been written exclusively in PLEX, more and more parts are being added in C++, resulting in a hybrid system.
- The ADL must support domain-specific abstractions and must be accepted widely by telecommunication experts.

The second requirement rules out general-purpose ADLs to which the world of processes communicating by exchanging signals via protocols defined by state machines cannot be mapped adequately. On the other hand, there are still multiple candidate languages meeting the stated requirements. In particular, the domain experts at Ericsson have applied both *SDL* [12] and *ROOM* [10] to the modeling of telecommunication systems. In the end, it was decided to use ROOM as the ADL in the E-CARES project, primarily because it provides clean concepts for modeling components interacting via ports along which messages are sent and received.

However, ROOM was not a clear winner over SDL. Furthermore, the UML component model [13] constitutes another candidate ADL which could be considered in future work[3]. For these reasons, we were aware that the target language for architectural recovery may be changed later on.

Therefore, we decided to decompose structural analysis into two *phases*: program analysis (as presented in Section 4) and architectural recovery. Furthermore, the architecture is represented separately from the structure in an architecture graph (with mappings into the structure graph). This design provides *modularity* of the E-CARES reverse engineering environment. Thus, the mapping to ROOM described below could be replaced easily with a similar mapping to SDL or to the UML components.

5.2 Real-Time Object-Oriented Modeling Language

The *Real-Time Object-Oriented Modeling* language [10] was designed for modeling asynchronous, concurrent, and distributed real-time systems. ROOM is

[3] When architectural recovery was realized in the E-CARES project, the UML 2.0 component model was still in flux.

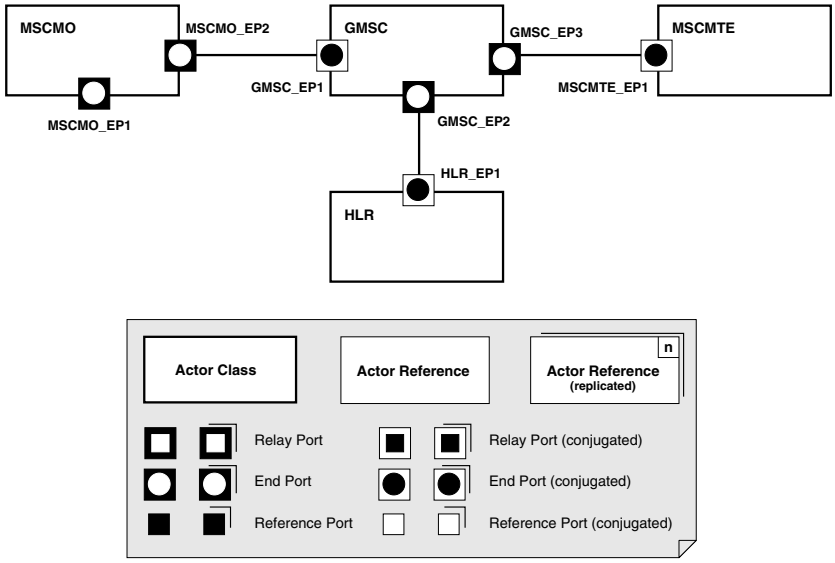


Fig. 7. ROOM diagram for the mobile originating call scenario (simple mapping)

the outcome of the authors' and their colleagues' combined practical experience in developing a variety of real-time systems in different industries including robotics, aerospace, and telecommunications. They claim that especially the development of a large distributed telecommunication system had a significant impact on the ROOM language.

A ROOM model consists of two parts: *structural models* and *behavioral models*. ROOM structure models are a kind of component inter-connection and component refinement diagrams. Behavior is modeled by means of extended state machines. Since this paper is concerned only with structural analysis, we will not discuss behavioral modeling below.

A simple example of a ROOM *structure diagram* is given in Figure 7[4]. Architectural components are represented by *actor classes*. Each actor class is displayed as a rectangle (e.g., MSCMO). The interfaces of actor classes are defined by *ports*, which are shown as small squares (with nested circles) placed on the border of the rectangle of the respective actor class. An actor communicates with its environment by sending and receiving *messages* via these ports. For each port, a *protocol* defines its incoming and outgoing messages. For some protocol p, its *conjugated protocol* is constructed by inverting the direction of messages. Finally, ports are connected by *bindings*. The ports connected by a binding must be *compatible*, the port at one end must be able to receive the messages sent via the port at the other end. This requirement is satisfied if the ports are associated to conjugated protocols.

[4] The legend refers to Figure 8, as well.

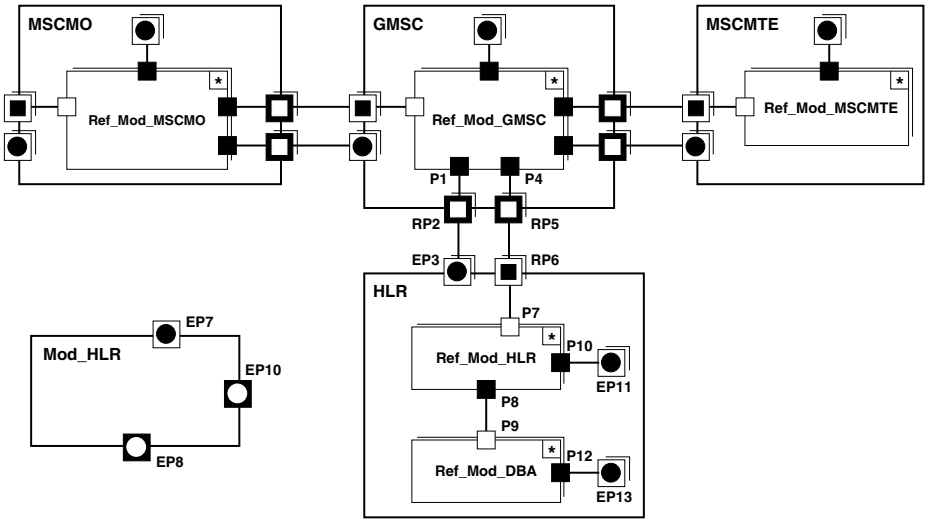


Fig. 8. ROOM diagram for the mobile originating call scenario (improved mapping)

Figure 8 shows a more complex structure diagram which makes use of further language constructs provided by ROOM. Actor classes may be refined into a subdiagram, which is shown within the rectangle for the actor class. Nodes of a subdiagram stand for *actor references*, i.e., references to instances of actor classes. In general, actor classes may be reused in different parts of the overall architecture; their instances are plugged into their respective context by connecting ports with bindings. In the case of double-shaped actor references, multiple instances may be created at run time (*replicated references*). The multiplicity is given in the upper right corner of the rectangle; * denotes an unbounded number of instances.

Furthermore, the diagram illustrates some generalizations and refinements concerning the modeling of ports. Analogously to replicated actor references, *replicated ports* are shown as double-shaped squares. In addition to *external ports*, an actor class may have *internal ports* which do not belong to its interface and therefore are shown inside (but near the border of) the respective rectangle. Furthermore, a distinction is made between class end ports and relay ports. *Class end ports*, which were shown already in Figure 7, are processed by the actor class itself; they are represented by squares with nested circles. In contrast, *relay ports* (nested squares) are used to pass messages upwards or downwards in the hierarchy without processing.

So far, we have introduced merely those elements of the ROOM language which are needed in the context of this paper. Further important language constructs such as inheritance on actor classes and layers are not exploited by the mapping algorithms below. *Inheritance* cannot be applied in architecture recovery since telecommunication systems programmed in PLEX are object-based

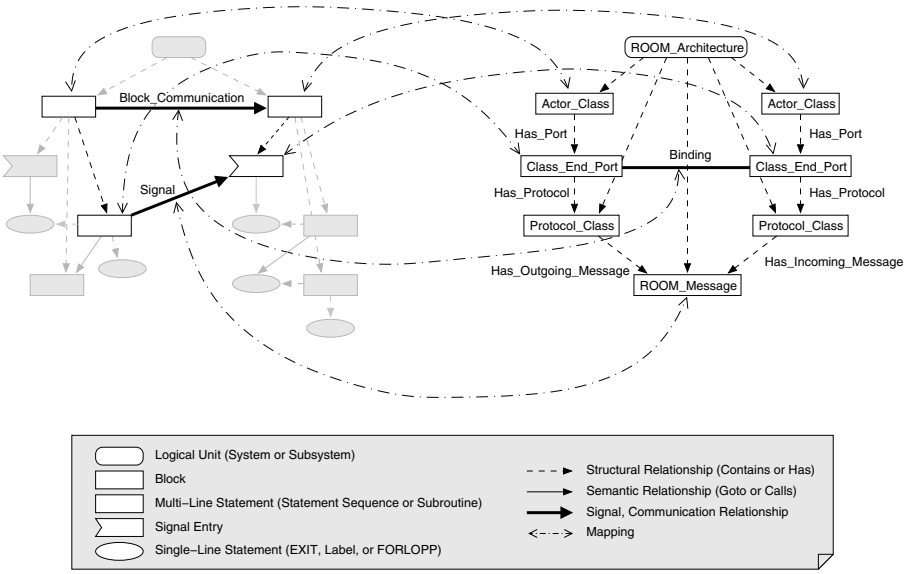


Fig. 9. Example mapping of a simple PLEX structure graph to a ROOM graph

rather than object-oriented. *Layers* are an important concept in telecommunication architectures. Unfortunately, no machine processable sources are available from which the layering of the AXE10 system could be inferred.

5.3 A Simple Mapping Algorithm

In general, the basic idea of the E-CARES approach to architecture recovery is to identify elements in the structure graph that can be considered to correspond to elements of the ROOM language. More precisely, this means that we have to identify elements in the PLEX structure graph that can be thought of as realizations of actors, ports, protocols, messages, bindings, etc. The identified structure graph elements are then mapped to newly created elements of an architecture graph.

First, we have defined a simple approach for mapping elements of the structure graph to ROOM elements (Figure 9) as follows:

1. PLEX blocks are mapped onto actor classes. Blocks are the active entities of a PLEX software system while actor classes are the active elements of a ROOM model.
2. Each block communication is mapped onto a corresponding binding. Before creating the binding, the ports to be connected have to be created. Thus, ports bundle all messages sent to or received from one destination.

3. Signals are mapped onto messages.
4. The protocol class attached to a port is derived from signal entries and signal sending statements, i.e., from the fine-grained communications which are aggregated into a block communication.

With respect to the running example of a mobile originating call and the participating blocks MSCMO, MSCMTE, GMSC, and HLR (see Figures 2 and 3), the architecture diagram resulting from this simple mapping algorithm is depicted in Figure 7. The diagram shows some characteristics typical for the type of ROOM structure diagrams produced by the simple mapping algorithm. Obviously, the diagrams consists of actor classes only. Furthermore, there is always a single communication contract between a pair of communicating actor classes. The ports bound to one of these communication contracts have conjugated protocols in the sense that no port is able to receive messages which cannot be produced by the other port. Finally, the port MSCMO_EP1 symbolizes that there might be ports that have no connection to any communication path. This may indicate dead code (received signals which are not sent) or may result from a constrained scope of program analysis (confined e.g. to some subsystem).

Comparing the results of this first and simple mapping algorithm with the information already provided by the structure graph, the improvement is quite small. The simple algorithm can be considered an *unparsing algorithm* that converts the block communication level of the structure graph into a standardized notation. The only improvement is that interfaces are now clearly visible in terms of ports and that the signals sent and received by a block are associated to different protocols that can be named, described and compared easily.

5.4 An Improved Mapping Algorithm

So far, we have merely performed a syntactic transformation from PLEX into ROOM. In the following, we will present a more elaborate algorithm which exploits *methods of use*. Once again, this mapping demonstrates the importance of domain knowledge beyond the language level.

The simple mapping algorithm presented so far does not consider any decomposition of blocks and actor classes, respectively. But, as already elaborated, most blocks in a PLEX software system contain at least one *module* (which is a conceptual rather than a syntactical unit). In addition, there can be several *instances* of a block or, more precisely, of its modules being the active entities of a PLEX system at runtime (see Subsection 2.3). Furthermore, we have already mentioned that the instances of a block are created, managed, and destroyed by the block itself.

Consequently, the actor classes resulting from architecture recovery should show a decomposition into several actors representing the module objects contained in the corresponding block. In addition, there should be a possibility to access the actors in the decomposition frame of an actor class from its behavior component that represents the control interface for instance management. Taking these considerations into account, our mobile originating call scenario is mapped into the refined structure diagram shown in Figure 8.

The improved mapping algorithm proceeds as follows:

1. Each block is mapped onto a complex actor class. In our example, the actor classes named MSCMO, GMSC, MSCMTE, and HLR are created.
2. Each module is mapped onto an an atomic actor class. In Figure 8, only the actor class Mod_HLR is shown as an example in the lower left corner.
3. Since as many module instances as needed are created at runtime, replicated references with multiplicity * are placed inside in the actor class for the surrounding block. In the example, the actor classes MSCMO, GMSC, and MSCMTE contain references to only one nested actor class because each of the corresponding blocks contains only one module. Since the block HLR contains two modules (one implementing the application logic of the block and one for the database), the corresponding actor class contains references to two nested actor classes.
4. For each block, a class end port is added to the interface of its corresponding actor class. This port represents the management interface of the block. In particular, the management interface handles SEIZE signals requesting the creation of link chains.
5. Similarly, a class end port is created for each actor class representing a module. For example, EP10 represents the interface which is in charge of managing the instances of module Mod_HLR.
6. For connecting to the management ports of nested modules, corresponding internal class end ports are generated within the respective actor classes (e.g., port EP11 in HLR).
7. For each pair of communicating modules (locally or across different blocks), class end ports are created on either side. For example, the port EP7 of actor class Mod_HLR is created for the connection to Mod_GMSC, and EP8 is going to be connected to a corresponding port of Mod_DBA.
8. For local communication among modules of the same block, bindings are created among the ports of actor class references. In this way, the binding between ports P8 and P9 within the actor class HLR is created.
9. If communication is performed between two modules belonging to different blocks, bindings cannot connect references to actor classes for modules directly because direct connections would break encapsulation. Rather, communication is routed via relay ports attached to the actor classes of the respective surrounding blocks. For example, for the communication between Mod_GMSC and Mod_HLR the relay ports RP5 and RP6 are created and connected with each other. Furthermore, they are connected to the ports of the respective actor class references acting as source or sink of the communication (ports P4 and P7, respectively).

Let us briefly sketch how a link chain would be set up at run time: First, the block MSCMO receives a request for setting up a link chain via its management port (lower port on the left-hand edge of the rectangle). Through its internal class end port (close to the upper edge), it delegates the request to its nested module Mod_MSCMO via its management port (located on the upper edge of the rectangle). In response to this request, an instance of this module is created.

To embed this instance, instances of relay ports of the surrounding block are created and connected to the respective ports of the module instance. Via the lower port shown on the right-hand edge, a message is request is sent to the next block in the link chain (GMSC). The reply returns a reference to a new instance of Mod_GMSC, and subsequent communication with this instance is performed via the new instances of the upper relay ports on the left-hand side of MSCMO and the right-hand side of GMSC, respectively.

Altogether, the improved algorithm constructs an architecture diagram which makes heavy use of domain-specific *design patterns*. In this way, the architecture diagram reflects the methods of use employed as design guidelines at Ericsson.

6 Realization

An overview of the E-CARES environment has been given in Section 3. To reduce the effort of implementing the E-CARES environment, we made extensive use of generators and reusable frameworks [17]. Scanners and parsers are generated with the help of JLex and jay, respectively. Graph algorithms are realized in PROGRES [18], a specification language based on programmed graph transformations. From the specification, code is generated which constitutes the application logic of the E-CARES environment. The user interface is implemented with the help of UPGRADE [19], a framework for building interactive tools for visual languages. Project specific extensions to the framework have been realized in Java.

A core part of the E-CARES environment consists of a (large and complex) PROGRES specification of structural and behavioral analysis. In this paper, we have given an informal presentation of structural analysis in E-CARES, mainly focusing on the construction of domain-specific abstractions for telecommunication systems. Due to the lack of space, we cannot elaborate in depth on the PROGRES specification for E-CARES. Rather, the reader is referred to [8,17,20] for detailed information in this area.

In the sequel, we present a small example from *program analysis* which demonstrates the application of PROGRES in the E-CARES environment. The graph transformation rule shown in Figure 10 is used to connect signal sending statements to matching signal entries. This rule is invoked after isolated structure graphs have been built by parsing PLEX blocks separately. For example, the application of this rule is used to generate the connection labeled SignalIAM in the structure graph of Figure 5.

Please note that Figure 5 shows a user-friendly, simplified representation of the structure graph. In particular, the connection SignalIAM is represented internally by a node and two adjacent edges rather than by a single edge. This way of representation is necessary because the underlying data model (attributed graphs) does not allow to attach attributes to edges (which is required in the case of the connection to be created).

The graph transformation rule is supplied with a parameter which determines the node for the signal sending statement to be connected (signalNode). Due to

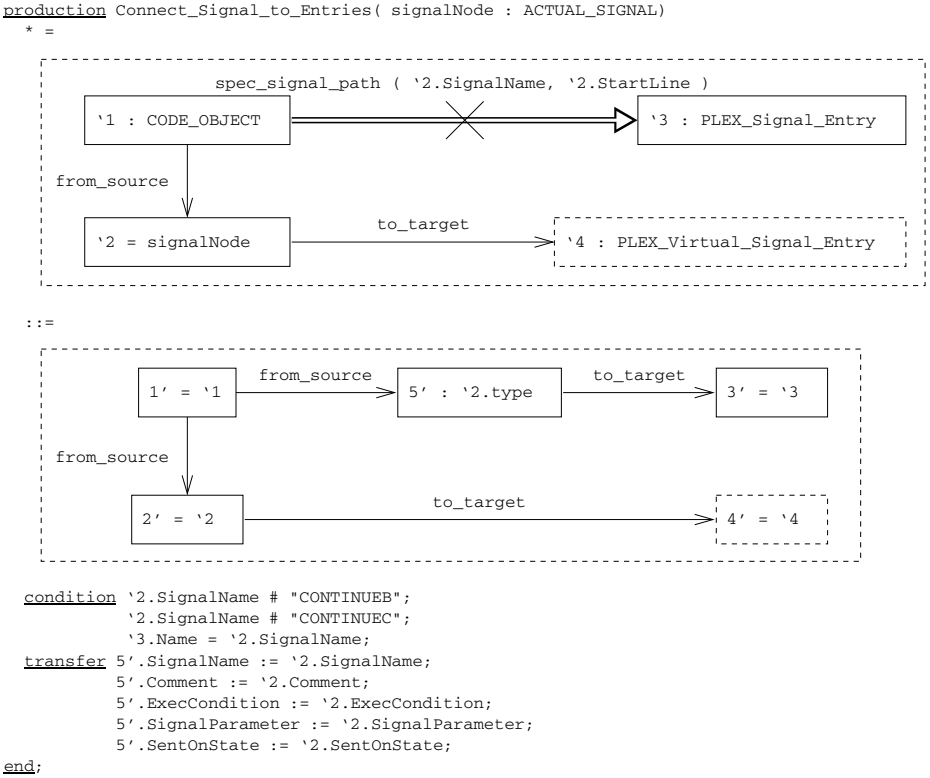


Fig. 10. Graph transformation rule for connecting sent signals to signal entries

the operator * (second line), the rule is applied to all matching subgraphs rather than just once. This is essential for handling multiple signals, which have to be connected to all potential recipients.

On the left-hand side (upper dashed box), which specifies the subgraph to be replaced, node '2 is fixed by the input parameter signalNode. When processing a single block without context information, the signal sending is connected to a virtual signal entry ('4), which acts as a placeholder for the final receiver(s). Both nodes are removed later in a clean-up phase; therefore, they are not contained in the sample graph of Figure 5. Node '1 represents the surrounding code object to which the signal sending statement will be lifted (statement sequence DIAL001_cs in Figure 5). Node '3 represents the signal entry to which the signal sending statement will be connected. Candidate nodes are retrieved by name matching (see the last line in the condition part which compares names attributes). In addition, a connection must not have been established yet, i.e., the (crossed out) path from '1 to '3 must not exist.

The right-hand side (lower dashed box) describes the transformation to be executed when a match of the left-hand side has been found (satisfying also the condition part of the rule). Here, the node 5' and two adjacent edges are created

(corresponding to the connection SignalIAM in Figure 5). The transfer part assigns values to various attributes attached to the node (SignalName, Comment, ...). All other parts of the graph remain unaffected.

It is interesting to note that methods of use have been taken into account in the specification of this graph transformation rule. As already mentioned at the end of Subsection 4.1, telecommunication experts at Ericsson employ the *phase division idiom* if processing of a signal requires more than a single time slice. Thus, CONTINUE signals have to be ignored when connecting single sending statements to single entries. For this reason, the condition part of the graph transformation rule requires that the name of the signal must not be equal to CONTINUEB or CONTINUEC (more than three cycles are never needed).

Architecture recovery does not augment the structure graph. Rather, it builds a new representation (an architecture graph). This allows for a clean separation of program analysis and architecture recovery, and it also permits multiple alternative architectural representations in different ADLs. Technically, architecture recovery is realized by a *triple graph transformation system* which is based on *triple graph grammars* [21,22]. A triple graph grammar is composed of rules operating on three graphs – a source graph (the structure graph), a target graph (the architecture graph) and a correspondence graph storing the mappings between the elements of the source and the target graph. In the original triple graph grammar approach, synchronous rules (extending all graphs simultaneously) are specified first, and directed rules (e.g., source-to-target transformations) are generated from synchronous rules. In our work, we specified directed transformations from the structure graph into the architecture graph immediately without coding the synchronous rules first (only one direction of transformation was needed in the E-CARES reverse engineering environment). For further details, the reader is referred to [8].

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 still strongly dependent on the syntax of the underlying programming language (PLEX), additional conceptual abstractions such as functions and modules have been added. From the structure graph, a programming language independent architectural description of the system under study is created. As telecommunication systems are designed in terms of layers, planes, services, protocols, etc., it has been crucial to choose an ADL which supports domain specific abstractions.

While E-CARES also considers the behavior of the system under study [14], many other reengineering tools such as e.g. Rigi [23] or KOGGE [24] 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

[3,4,25]. Here, recovery of units of data abstraction and migration to an object-oriented software architecture play a crucial role [26]. More recently, reengineering has also been studied for object-oriented programming languages such as C++ and Java. E.g., TogetherJ or Fujaba [27] generate UML class diagrams from source code.

The phases extraction, post-processing, and inspection can be found in a similar form in different reverse or reengineering approaches. A typical example is the *extract-abstract-view* metaphor described in [28], which consists of three phases that occur in terms of activities in our process, too. In this abstract form, the metaphor also serves as a kind of reference architecture for reverse engineering tools. In particular, integrated reverse engineering tools like DALI [29], GUPRO [30], PBS [31], ReForDi [3], Rigi [23,32], Bauhaus [33], ANAL/SoftSpec [28], and SWAGKit [34] follow this reference architecture. The E-CARES reengineering environment does so, too.

Architecture recovery means to bridge the gap between implementation and architecture specification using appropriate reverse engineering techniques and tools. Basically all of these approaches employ some kind of information retrieval facility that extracts information artifacts from the subject system's source code. The techniques used there range from parsers [33] to string pattern matchers like the UNIX-tool grep or ESPaRT [35,36].

In general, we identified two different classes of approaches to solve this problem – approaches using any kind of domain information, e. g. in terms of patterns, and approaches employing more general-purpose complex flow analysis techniques. For example, Fiutem et al extract information on systems, programs or modules by means of architectural recognizers [37] working on the Abstract Syntax Tree (AST).

Another architecture reconstruction method that employs patterns to identify architectural components and connectors is described by Guo et al [38]. Their ARM method first obtains knowledge on the as-designed architecture. Then, this knowledge is used to define queries for potential patterns which are applied automatically to extracted and fused source model views. ARM uses lower-level patterns to build higher-level patterns and supports composite patterns as well. Similar work has been described in [36]. In contrast to the former approach, this approach by Pinzger and Gall also considers the knowledge of systems experts in pattern definition.

If the formulation of suitable patterns is impossible for any reason, approaches using elaborate control and data flow analysis are more suited and produce better results. Such approaches recover architectural components and connectors based on other criteria like *low coupling* and *strong cohesion* [15,16,39]. In this context, strongly cohesive code artifacts are considered to belong to the same component while the relationships between the different groups of cohesive artifacts become connectors of the resulting architecture. A possible technique to identify architectural components is, for example, program slicing and concept lattices as described in [33].

Comparing these approaches with the E-CARES approach, a significant difference to the majority of approaches is that we clearly separate architecture recovery specific activities from activities generally performed in structural or behavioral analysis. To the best of our knowledge, our approach is the only one that considers domain specific modeling languages and notations in architecture recovery and thus provides a flexible approach that allows to consider *different* of such architectural styles in the same tool environment. The separation of general structural and behavioral analysis from architecture recovery allows to share major parts of the existing functionality in different recoveries.

The E-CARES approach does incorporate patterns on different levels as well as control and data flow analysis. In structural analysis, the patterns in E-CARES are domain-specific *code patterns* in most cases rather than high-level or general purpose design patterns. In architecture recovery, design patterns above code level have been used. An approach to introduce high-level connectors in E-CARES as defined by Herzberg can be found in [11]. Instead of utilizing design patterns, we currently combine knowledge on code patterns (idioms) with control-flow analysis and data flow analysis to create abstractions in the structure graph. Then, elements of the structure graph are mapped to corresponding elements of an architectural representation. We agree with the pattern-using community that existing knowledge obtained from experts and (design) documents is mandatory in architecture recovery — but also in program analysis.

8 Conclusion

We have presented the E-CARES approach for reengineering of telecommunication systems. In this paper, the focus has been on the reverse engineering tools of the E-CARES prototype, specifically on structural analysis (program analysis and architecture recovery). In addition to structural analysis, E-CARES supports behavioral analysis to enrich the results of structural analysis with additional information. These reverse engineering tools aid in system understanding but not yet in system restructuring. Recently, the E-CARES prototype has been extended to also cover re-design and code transformation [9]. Multi-language support as well as extensions to the ROOM architecture description language have been elaborated in subprojects.

Methods of use have played an important role in structural analysis. In program analysis, conceptual units such as labeled statement sequences, functions, and modules are recovered which do not correspond to syntactic units of the PLEX language. For example, modules group functions which access a common data record for managing an instance of a block (for serving a specific call). Furthermore, in architectural recovery ROOM diagrams are constructed which incorporate instances of domain-specific design patterns. For example, an actor class for a block is connected to its environment via two ports which handle messages at different levels of operation (ports for creating/deleting block instances and ports for communicating with specific instances, respectively). Altogether, domain-specific knowledge has proved indispensable in constructing domain-specific abstractions which are meaningful to telecommunication experts.

Thus, all tools in E-CARES 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. Though domain knowledge was widely used in extraction logic, the E-CARES prototype is a modular reengineering system such that major parts can be reused when analysing software from other domains.

To date, approximately 2.5 million lines of PLEX code plus several ten thousands of lines of additional documents have successfully 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 system 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 and the architecture extraction have proved their value for system analysis and system understanding. Therefore, 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.

Acknowledgments. The work reported in this paper was performed while both authors were members of the group of Manfred Nagl (Department of Computer Science 3, RWTH Aachen University). Manfred set up the E-CARES project in cooperation with Ericsson. Funding from Ericsson is gratefully acknowledged. Furthermore, we are indebted to all persons at Ericsson who provided support in different ways (Ari Peltonen, Dominikus Herzberg, Martin Blumbach, Jörg Bruß, Dietmar Wenninger, and Andreas Witzel).

References

1. Chikofsky, E.J., Cross II, J.H.: Reverse engineering and design recovery: A taxonomy. IEEE Software 7(1), 13–17 (1990)
2. Kazman, R., Woods, S.G., Carrière, J.: Requirements for integrating software architecture and reengineering models: CORUM II. In: Proceedings 5th Working Conference on Reverse Engineering (WCRE 1998), Hawai, USA, pp. 154–163. IEEE Computer Society Press, Los Alamitos (October 1998)
3. Cremer, K., Marburger, A., Westfechtel, B.: Graph-based tools for re-engineering. Journal of Software Maintenance and Evolution: Research and Practice 14, 257–292 (2002)
4. Markosian, L., Newcomb, P., Brand, R., Burson, S., Kitzmiller, T.: Using an enabling technology to reengineer legacy systems. Communications of the ACM 37(5), 58–70 (1994)
5. Kollmann, R., Selonen, P., Stroulia, E., Systä, T., Zündorf, A.: A study on the current state of the art in tool-supported UML-based static reverse engineering. In: van Deursen, A., Burd, E. (eds.) Proceedings 9th Working Conference on Reverse Engineering (WCRE 2002), Richmond, VA, pp. 22–32. IEEE Computer Society Press, Los Alamitos (November 2002)

6. Wilde, N., Scully, M.: Software reconnaissance: Mapping program features to code. Journal of Software Maintenance: Research and Practice 7(1), 49–62 (1995)
7. Marburger, A., Herzberg, D.: E-CARES research project: Understanding complex legacy telecommunication systems. In: Sousa, P., Ebert, J. (eds.) Proceedings 5th European Conference on Software Maintenance and Reengineering (CSMR 2001), Lisbon, Portugal, pp. 139–147. IEEE Computer Society Press, Los Alamitos (2001)
8. Marburger, A.: Reverse Engineering of Complex Legacy Telecommunication Systems. Informatik. Shaker Verlag, Aachen (2005)
9. Mosler, C.: Graphbasiertes Reengineering von Telekommunikationssystemen. Informatik. Shaker Verlag, Aachen (2009)
10. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object-Oriented Modeling. John Wiley & Sons, Reading (1994)
11. Herzberg, D.: Modeling Telecommunication Systems: From Standards to System Architectures. PhD thesis, RWTH Aachen University, Aachen, Germany (2003)
12. Ellsberger, J., Hogrefe, D., Sarma, A.: SDL - Formal Object-Oriented Language for Communicating Systems. Prentice Hall, Upper Saddle River (1997)
13. Object Management Group Needham, Massachusetts: OMG Unified Modeling Language (OMG UML), Superstructure, V 2.1.2. formal/2007-11-02 edn. (November 2007)
14. Marburger, A., Westfechtel, B.: Tools for understanding the behavior of telecommunication systems. In: Proceedings 25th International Conference on Software Engineering (ICSE 2003), Portland, OR, pp. 430–441. IEEE Computer Society Press, Los Alamitos (2003)
15. Liu, S.S., Wilde, N.: Identifying Objects in a Conventional Procedural Language: An Example of Data Design Recovery. In: Proceedings of the Conference on Software Maintenance (CSM 1990), San Diego, CA, pp. 266–271. IEEE Computer Society Press, Los Alamitos (1990)
16. Canfora, G., Cimitile, A., Munro, M.: An Improved Algorithm for Identifying Objects in Code. Software - Practice and Experience 26(1), 25–48 (1996)
17. Marburger, A., Westfechtel, B.: Graph-based reengineering of telecommunication systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 270–285. Springer, Heidelberg (2002)
18. Schürr, A., Winter, A., Zündorf, A.: The PROGRES approach: Language and environment. In: Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.) Handbook of Graph Grammars and Computing by Graph Transformation. Applications, Languages, and Tools, vol. 2, pp. 487–550. World Scientific, Singapore (1999)
19. Böhlen, B., Jäger, D., Schleicher, A., Westfechtel, B.: UPGRADE: A framework for building graph-based interactive tools. In: Mens, T., Schürr, A., Taentzer, G. (eds.) Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2002), Barcelona, Spain. Electronic Notes of Theoretical Computer Science, vol. 72-2, pp. 149–159 (October 2002)
20. Marburger, A., Westfechtel, B.: Behavioral analysis of telecommunication systems by graph transformations. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 202–219. Springer, Heidelberg (2004)
21. Schürr, A.: Specification of graph translators with triple graph grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
22. Königs, A., Schürr, A.: Tool Integration with Triple Graph Grammars - A Survey. In: Heckel, R. (ed.) Proceedings of the SegraVis School on Foundations of Visual Modelling Techniques. Electronic Notes in Theoretical Computer Science, vol. 148, pp. 113–150. Elsevier Science, Amsterdam (2006)

23. 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 Sciences ACFAS 1994, Montreal, Canada, pp. 41–48 (May 1994)
24. Ebert, J., Süttenbach, R., Uhe, I.: Meta-CASE in practice: A case for KOGGE. In: Olivé, À., Pastor, J.A. (eds.) CAiSE 1997. LNCS, vol. 1250, pp. 203–216. Springer, Heidelberg (1997)
25. van Zuylen, H.J. (ed.): The REDO Compendium: Reverse Engineering for Software Maintenance, p. 405. John Wiley & Sons, Chichester (1993)
26. Canfora, G., Cimitile, A., De Lucia, A., Di Lucca, G.A.: Decomposing Legacy Systems into Objects: An eclectic approach. Information Technology and Software 43(6), 401–412 (2001)
27. Zündorf, A.: Rigorous Object-Oriented Development. Habilitation thesis, University of Paderborn, Paderborn, Germany (2002)
28. Lange, C., Sneed, H.M., Winter, A.: Applying the graph-oriented GUPRO Approach in comparison to a Relational Database based Approach. In: Proceedings of the 9th International Workshop on Program Comprehension IWPC 2001, pp. 209–218. IEEE Computer Society Press, Los Alamitos (2001)
29. Kazman, R., Carrière, S.J.: Playing Detective: Reconstructing Software Architecture from Available Evidence. Journal of Automated Software Engineering 6(2), 107–138 (1999)
30. Ebert, J., Kullbach, B., Riediger, V., Winter, A.: GUPRO – Generic Understanding of Programs: An Overview. Electronic Notes in Theoretical Computer Science 72(2), 10 pages (2002), `http://www.elsevier.nl/locate/entcs/volume72.html`
31. Finnigan, P.J., Holt, R.C., Kalas, I., Kerr, S., Kontogiannis, K., Müller, H.A., Mylopoulos, J., Perelgut, S.G., Stanley, M., Wong, K.: The Software Bookshelf. IBM Systems Journal 36(4), 564–593 (1997)
32. Müller, H.A., Orgun, M.A., Tilley, S.R., Uhl, J.S.: A Reverse Engineering Approach To Subsystem Structure Identification. Journal of Software Maintenance: Research and Practice 5(4), 181–204 (1993)
33. Koschke, R.: Atomic Architectural Component Recovery for Program Understanding and Evolution. Doctoral thesis, Institute of Computer Science. University of Stuttgart, Stuttgart, Germany, p. 414 (2000)
34. Holt, R.C., Malton, A., Davis, I., Bull, I., Trevors, A.: SWAGKit – The Software Architecture Toolkit. Software Architecture Group, University of Waterloo, Canada (2003), `http://swag.uwaterloo.ca/swagkit/`
35. Knor, R., Trausmuth, G., Weidl, J.: Reengineering C/C++ Source Code by Transforming State Machines. In: van der Linden, F.J. (ed.) Development and Evolution of Software Architectures for Product Families. LNCS, vol. 1429, pp. 97–105. Springer, Heidelberg (1998)
36. Pinzger, M., Gall, H.: Pattern-Supported Architecture Recovery. In: Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002), Paris, France, pp. 53–61 (June 2002)
37. Fiutem, R., Tonella, P., Antoniol, G., Merlo, E.: A Cliché-Based Environment to Support Architectural Reverse Engineering. In: Proceedings of the 1996 International Conference on Software Maintenance (ICSM 1996), Monterey, CA, USA, November 4-8, pp. 319–328. IEEE Computer Society Press, Los Alamitos (1996)

38. Guo, G.Y., Atlee, J.M., Kazman, R.: A Software Architecture Reconstruction Method. In: Donohoe, P. (ed.) Proceedings of the First Working IFIP Conference on Software Architecture (WICSA 1999), San Antonio, Texas, USA, February 22-24, pp. 15–34. Kluwer Academic Publishers, Dordrecht (1999)
39. Burd, E., Munro, M., Wezeman, C.: Analysing Large COBOL Programs: the extraction of reusable modules. In: Proceedings of the 1996 International Conference on Software Maintenance ICSM 1996, Monetery, Canada, pp. 238–243. IEEE Computer Society Press, Los Alamitos (November 1996)