Electronic Communications of the EASST
Volume 1 (2006)

Proceedings of the
Third International Workshop on Graph Based Tools
(GraBaTs 2006)

RePLEX: A Model-Based Reengineering Tool for PLEX Telecommunication Systems

Christian Fuss, Christof Mosler, Marcel Pettau

12 pages

Guest Editors: Albert Zündorf, Daniel Varró
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

RePLEX: A Model-Based Reengineering Tool for PLEX Telecommunication Systems

Christian Fuss, Christof Mosler, Marcel Pettau

[fuss|mosler|pettau]@i3.informatik.rwth-aachen.de
http://www.se.rwth-aachen.de
Department of Computer Science 3 (Software Engineering)
RWTH Aachen University, Germany

Abstract: Maintenance of complex legacy software systems is a challenging task. In the first place, maintenance requires understanding the system. Reverse engineering and reengineering tools, which make the design of the current system available on-line and which support planning and performing changes to the system, are urgently needed. We present a new tool for reengineering telecommunication systems, recovering the current architecture, and extracting state machines reflecting the system behavior. The tool is based on a structure graph of the architecture and allows architectural modifications with according code changes. The modifications are specified as graph transformations using FUJABA enabling the generation of a Java prototype, which is accessible via a GUI based on the Graphical Editor Framework (GEF) plug-in for the Eclipse workbench.

Keywords: Model-driven Development, Graph-based Reengineering Tools

1 Introduction

The reverse engineering and restructuring of large and complex software systems is a difficult task. E-CARES[1] is a research project at RWTH Aachen University, Department of Computer Science 3 in cooperation with Ericsson Eurolab Deutschland GmbH (EED), to study methods and tools for reengineering of complex legacy systems implemented in PLEX. The programming language PLEX [Wen99] was developed in the 70s at Ericsson and is still used within the company for developing telecommunication infrastructure. It is an asynchronous concurrent real-time language using the signaling paradigm, i.e. only incoming signals can trigger code execution. The current system under study is Ericsson's AXE10, a mobile-service switching center (MSC) comprising more than ten million lines of code.

Generally speaking, the reengineering process can be divided into three phases. In the reverse engineering phase, engineers analyse the legacy systems to improve understanding and gain more information about its current state, often by representing the software structure on a high abstraction level (e.g. as an architecture graph). During the restructuring phase, engineers perform re-design transformations to improve the software architecture. Finally, the source code has to be modified according to the changes performed on design level.

[1] E-CARES is the acronym for Ericsson Communication ARchitecture for Embedded Systems

For representation and analysis of the legacy systems, we follow a graph-based approach, i.e. all underlying structures are graphs and editing operations are specified by graph transformation rules. Our goal was to provide an interactive reengineering environment, which should allow a flexible and easy addition of new functionality (e.g. new analyses and transformations).

This paper is structured as follows: our approach and motivation for using graph-based tools for software reengineering are described in section 2. In section 3, we present more details about the realization of the RePLEX reengineering tool as a plug-in for the Eclipse IDE [Ecl06] using the FUJABA tool suite. We focus on the graph-based aspects of the tool, and explain the parser and source code transformations only very briefly. In section 4, a description and walk-through from user perspective is presented. The paper closes with a summary and an outlook.

2 Approach

2.1 Reengineering Process

The reengineering process comprises three phases: reverse engineering, re-design and source code transformation. All of them are supported by our tool environment, shown in Figure 1. In the reverse engineering phase, we use different sources of information. The most important and reliable one is the source code of the PLEX system. We parse the source code and create a textual *structure document* describing the system structure, comprising its communication, control flow, and data flow. Furthermore, we use some other sources of information, e.g. *signal lists*[2], and add it to the structure document.

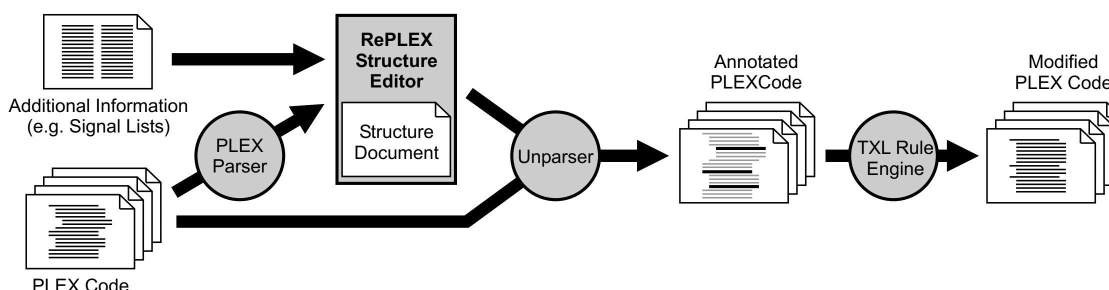

Figure 1: RePLEX Tool Environment with source, intermediate and target documents.

After parsing, the *RePLEX Structure Editor* instantiates a graph from the structure document. On this graph, the user can perform various types of analyses by using different visualization and query techniques. In addition, different metrics can be used to obtain more quantitative characterizations of the analyzed system. These features support the reverse engineering of the analysed systems.

The tool supports the re-design phase by offering some complex algorithms for suggesting how to improve the legacy software. The user can then interactively adapt the suggested re-

[2] A signal list is a textual file, which provides the names of the blocks to which outgoing signals are sent during runtime. In PLEX, signal receivers are often initialized dynamically. By considering the signal list, we are able to exclude any signal edges from the graph which are potentially possible, but never actually used.

design transformations taking the semantics of the software into account. During this process, the graph structure and attributes of the nodes can change. Of course, the tool supports manual re-design of the structure graph by providing basic editing functionalities as well.

After the re-design, the tool propagates modifications from the design level to the implementation level. First, an *unparser* writes information about the modified software structure into a textual file combining the original source code and annotations indicating what text transformations must be preformed. In the last step, the new source code is generated by a TXL-based tool processing the annotations.

2.2 Graph-based Reengineering

As shown, the reengineering process comprises graph-based and source code-based steps. Representing software architectures by graphs is a natural choice, because in this manner not only the hierarchical structures but also all other relations between code artifacts can be represented directly. Therefore using visual and graph-based languages for building corresponding reengineering tools is obviously advantageous. On the other hand, of course also the source code must be processed.

By using both abstraction levels in our process, we can clearly divide the process into phases which are performed automatically and manually. The parsing and unparsing of the source code can be done automatically. During the re-design phase, the interactive process can be performed in a much more convenient and effective way on the graph-level. At the same time, we are exclusively interested in architectural aspects. Hence, we analyze and restructure the software on a relatively high abstraction level without taking into account every single statement. In our graph model, we consider only larger code parts, such as subroutines, signal entries, and variable declarations, and go into detail only when necessary, for example when dealing with statements sending signals. Furthermore, the separation of the graph and the source code levels should allow an easy integration of new programming languages into the RePLEX tool suite. While the model extensions should comprise mainly the algorithmic aspects required for the re-design process, the language specific parsers and unparsers will comprise most of the details needed for the actual source code transformation.

2.3 Related Work

There exist several graph-based reengineering tools. The comparison with other projects following a graph-based approach, such as Rigi [MWT94], Bauhaus [Kos00], and GUPRO [EKRW02], shows that most of these tools lack the support provided by a high-level specification language. Hence, graph transformations cannot be specified in a declarative way. These projects also concentrate on reverse engineering and do not support software restructuring.

The approach in [MVDJ05] shows how refactorings for object-oriented software can be defined by using graph rewrite rules using FUJABA and AGG [Tae99] for tool validation. *AGG* is a general tool environment for algebraic graph transformation following the interpretative approach. The AGG environment consists of a graphical user interface and an interpreter, which can be used for the specification and prototypical implementation of Java applications with complex graph-structured data. The paper at hand presents a very similar approach but aims at the

reengineering of programs written in a different kind of programming language. As we consider the software on a higher architectural level, without going into a detailed analysis of every single statement, the studied re-design transformations are also different.

The FUJABA Tool Suite RE [FUJ05] is a collection of reengineering tools and plug-ins. It allows the parsing of Java source code and supports different kinds of static and dynamic analyses, such as recognition of design patterns and anti-patterns [NSW+02].

3 Realization

Based on knowledge from a former prototype implemented with PROGRES we started developing the new tool RePLEX[3] within a graduate course for computer science students at RWTH Aachen University. The new prototype focuses on the forward portion of the reengineering process offering features for typical re-design tasks on PLEX code, whereas the old one focused on the reverse portion with abstracting from source code level and a plentitude of analyses.

In the following sections, we describe some aspects of the realization in detail, particularly the specification of re-design features with FUJABA, construction of the editor interface as Eclipse plug-in with GEF and text-transformation parsers and unparsers with the help of jay and TXL.

3.1 Parsing PLEX Code

The main unit of a PLEX program is a block corresponding to one source file. Many source files make up a system, which might be divided into subsystems containing further subsystems or blocks. The structure of a system and its subsystems can be derived from the directory structure of the source code. All nodes in the structure graph comprise different types of attributes describing where the corresponding software parts can be found and what their characteristics are.

Major parts of the RePLEX prototype are based on formal specifications from which they are generated. On the source code level, scanners, parsers, and unparsers are generated automatically. The structure of the PLEX system below block-level is determined by the pre-existing PLEX-parser on jay basis. This parser was generated automatically by use of the lexical analyzer generator *jlex* [Ber00] and the parser generator *jay* [SK06], which are Java correlatives of the GNU lex/yacc compiler-compilers.

During parsing, many details from the PLEX code are abstracted away, putting only structurally relevant information into the structure graph, e.g. subroutines, statement sequences, signal entries, and data objects. The primary relation of elements is the contains-relation, taken directly from the nesting of the abstract syntax tree, but additionally references from the PLEX code are introduced as own edges into the structure graph, e.g. goto, calls, from_source, to_target edges.

The parser outputs a description of the abstracted structure graph as Python script or XML file, which can then be read by the structure editor.

[3] The acronym RePLEX combines the two words Reengineering and PLEX.

3.2 Specifying Re-design Features with FUJABA

As outlined in Section 2.2, a graph-based approach is well-fitted for the interactive re-design of the structure graph of PLEX systems. There are several tools for performing the desired graph transformations. Some come from the class of environments for specifying visual notations like DIAGEN [Min02], or MetaEdit+ [KLR96]. Others are general-purpose graph transformation environments like PROGRES [SWZ99], AGG [Tae99], or FUJABA [FUJ99]. There were many reasons to choose FUJABA as realization environment: it has a strong reengineering background [FUJ05], enabling the integration with tools from related projects. From its roots, FUJABA is closely related to PROGRES, which served as environment for the specification of our old prototype and it allows easy integration with Java code, particularly through the possibility to adapt the code generation with little effort. Finally, specifying with FUJABA is very easy to learn for novices like the students involved in the project, due to its resemblance of UML class and activity diagrams.

A FUJABA specification is two-parted and consists of a meta-model of the notation, defined as UML class diagram, and so-called story diagrams [FNTZ98] describing the operations on the elements, in our case re-design operations for the according PLEX structure elements. From this specification, the Java code is generated with the FUJABA plug-in CodeGen2.

Meta Model

Figure 2 depicts the core of our meta model, directly reflecting the structure extracted by the parser as described in Section 3.1. Every element in the graph is derived from *GraphObject*, managing the unique IDs of all graph elements. GraphObject is a specialization of *ObservableObject*, providing the event-handling for the Eclipse/GEF user interface (see Section 3.3 for details).

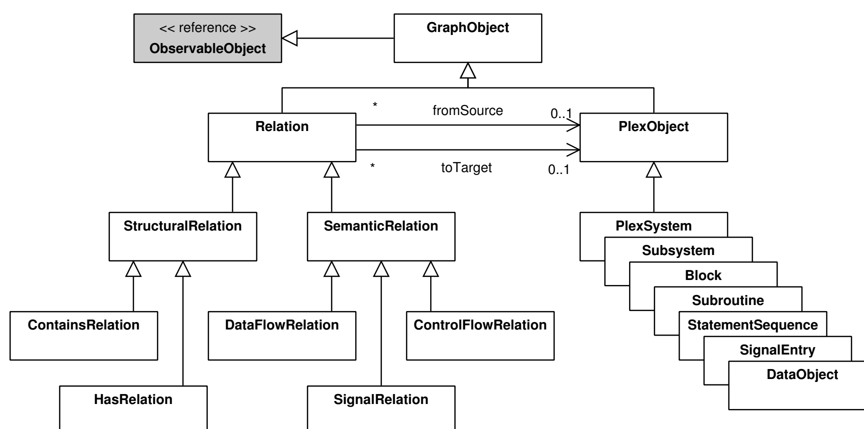

Figure 2: Core of the structure graph model, reflecting the structure graph produced by the PLEX parser.

Relations are directed and attributed; they are modeled as edge-node-edge constructs with

fromSource and *toTarget* associations. The contains and has relations are structural relations. Relations derived from references within the PLEX code fall into the class of semantic relations which is specialized into *DataFlowRelation* (e.g. readsVariable, writesVariable), *ControlFlowRelation* (e.g. goto, call), and *SignalRelation* (e.g. local signal, global inter-block signal).

Story Diagrams

The FUJABA story diagrams, used to describe the re-design transformations, are a combination of UML activity and collaboration diagrams. Story diagrams can be read as activity diagrams with a particular activity type called the *story-activity*, which enables describing interaction of objects during the operational sequence of the program or the time-flow of program execution. In terms of methods of classes, it means that every method in a class is described by a story diagram. For the control flow of a method, elements of UML activity diagrams are used, e.g. start and end points, transitions, and control structures. Within the control flow, so called story patterns are used, modeling the transformation of the object structure. Each method consists of a start point, several story patterns, and at least one end point, all connected by transitions. FUJABA supports negative application conditions, restrictions, optional objects, and set-valued objects within story patterns.

Figure 3 shows an exemplary story diagram implementing the method *mergeSubSystems* of the class *SubSystem*. Each *SubSystem* object offers the method *mergeSubSystems* which moves all contained blocks to another subsystem, given as a parameter, and then destroys itself. In the left part of the story diagram, we search for *containsRelation* objects connecting blocks to the current subsystem. For each relation we find, the *fromSource* edge is moved to the second subsystem[4]. The right part of the diagram shows the destruction of the current *SubSystem* object and its *containsRelation* object to the *PlexSystem* node.

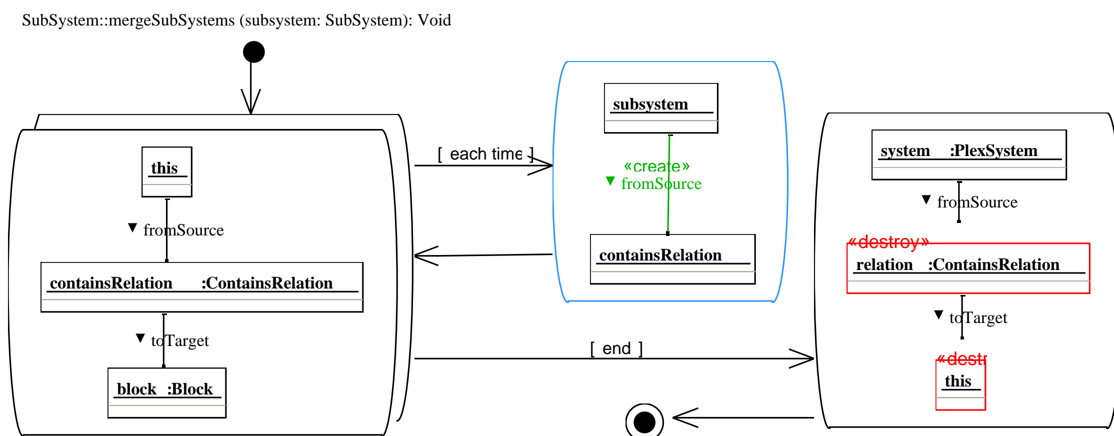

Figure 3: Story diagram for the merge method of the *SubSystem* class

[4] Since the *fromSource* relation has a 0..1 cardinality, the old edge is destroyed implicitly and thus not marked with ≪destroy≫

3.3 Implementing the Editor GUI with GEF

The core features of the RePLEX prototype were generated from the FUJABA documents. Yet, the graphical user interface had to be implemented by hand. We chose to integrate the tool into the Eclipse IDE [Ecl06] and used the Graphical Editing Framework (GEF) [GEF06] for realization.

GEF is a Model-View-Controller (MVC) framework that can be plugged into the Eclipse IDE to implement graphical editors for underlying models easily. GEF offers rich support for the controller part of MVC, dictating a standard GEF architecture. For the view part, it depends on the lightweight graphic system Draw2D, also available as an Eclipse plug-in. The underlying model can be chosen arbitrarily, it merely has to offer event notification so that the controllers can register with the model entities.

Since standard generated FUJABA classes do not support event notification, we had to adapt the code generation using the FUJABA plug-in CodeGen2 [GSR], which allows modification of the templates used to generate the code. We used the observer pattern and implemented a general observable class for the housekeeping of notification queues etc. All generated Java classes for the model entities are then derived from the observable class and the code generation templates insert notifications into all modifiers.

The current implementation of the model representation with GEF deploys only two parameterizable representations for model entities: one for nodes and one for edges (actually, these are edge-node-edge constructs, cf. Figure 2). The view parts for these are parameterized by the type of the associated entity. The functionality offered for a node can also be parameterized by the type of the entity. This is possible through the GEF mechanism of *Requests* and *EditPolicies*, because EditPolicies contain a mapping of Requests to editing commands. The mapping can be implemented to pick the commands dynamically depending on the elements type.

The different choices of parameterization have yet been coded into the prototype by hand, but with the UPGRADE framework [BJSW02] for PROGRES prototypes, it has proven viable to generate a complete parameterizable GUI from the specification of a graph transformation prototype. This should also be possible for FUJABA prototypes in the future.

3.4 Unparsing and Code Transformations

After re-design, the tool propagates the improvements on structural level to the implementation level. The information about the improved software architecture is stored in the structure graph. This graph only forms an instantiated representation of the system, it does not contain all the information required for the generation of new source code. Each graph node representing data of a control structure stores its original file name and its line numbers but not the actual source code. Therefore, to obtain the changed program we resort to the original source code files and enhance them by adding information extracted from the modified graph, describing how the particular parts should be transformed.

The syntax of this code and the corresponding transformations are defined in the rule-based programming language TXL [CHP91]. The TXL transformation system parses the enhanced source code files (see Figure 5) and performs the transformations on the created abstract syntax tree. A TXL rule can define, for instance, transformations between different types of signals and

signal entries (e.g. local and global). In the case of moving blocks to other subsystems, only some comments are modified and the new source code is copied to the corresponding subsystem directory.

During the transformation process, TXL parses the original source code files which is a huge advantage of using this tool. As not all information required for generating the new source code are stored in the graph model, more details must be extracted from the source code. We should remark that TXL could also be used as a parser for extracting the structure document in the reverse engineering phase. But as a jay-based parser for PLEX was already available from former projects, we decided to use the pre-existing one. More detailed information about the TXL-based tool can be found in [Mos06].

4 Using the RePLEX Tool

The RePLEX tool offers complex functionality, specified with the high-level mechanism offered by FUJABA, that can easily be accessed through the GUI of an Eclipse plug-in. Figure 4 shows the RePLEX GUI integrated into the Eclipse IDE. In this perspective it consists of a *Navigator View* Ⓐ, *Strucure Graph Editor* Ⓑ, *Tool Bar* Ⓒ and *Editing Palette* Ⓓ as well as *Property View* Ⓔ and *Code View* Ⓕ.

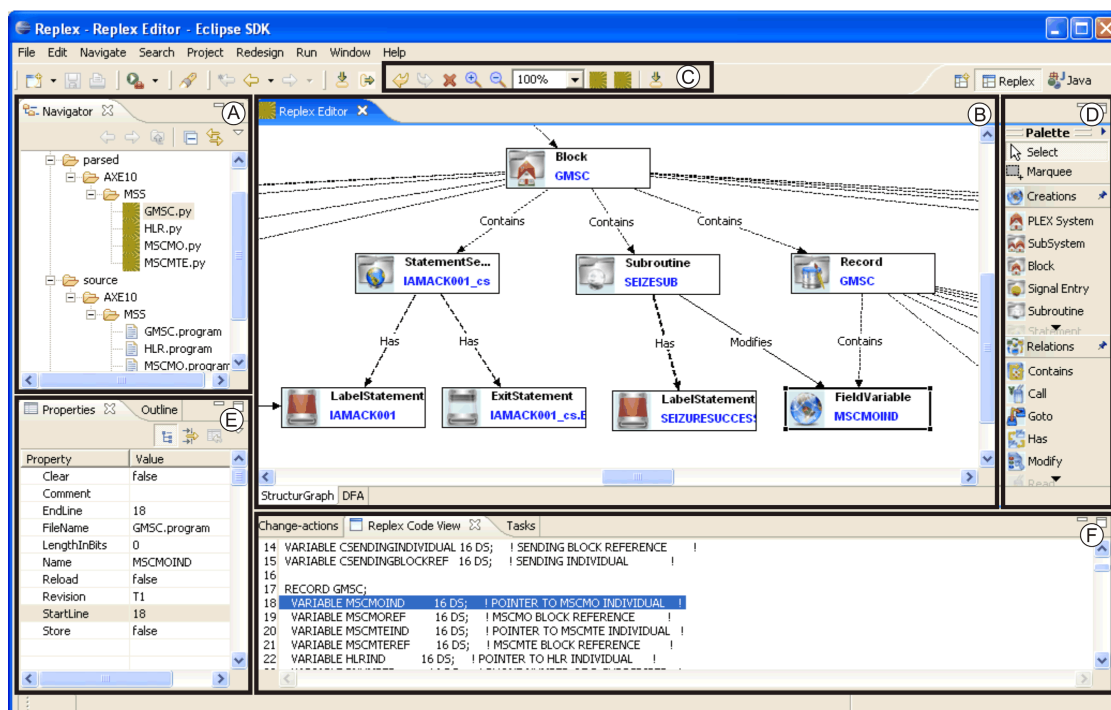

Figure 4: The RePLEX tool

In the *Navigator View* Ⓐ one can see the directory structure of the RePLEX project after the PLEX code has been parsed. It contains a separate directory for PLEX code and parsed structure

documents (one for each PLEX block file). The project contains a third directory for modified PLEX code after generating PLEX code from a modified structure graph.

By choosing a structure document in the Navigator View, the corresponding structure graph is shown in the *Structure Graph Editor* Ⓑ. In the figure, you can see a cutout of the structure graph of the GMSC block. The graph consists of the block node, and nodes for different program parts of the block. To make the analysis of a structure graph as convenient as possible, the *Tool Bar* Ⓒ offers many functions for viewing, e.g. zooming, outline, and layout algorithms. All these features allow effective reverse engineering of the analysed systems.

The *Editing Palette* Ⓓ offers simple functionality for the modification of structure graphs. In particular, one can manually create, destroy, and modify single nodes and edges. Using the selection tool, one can also get more information on the selected element from the *Property View* Ⓔ, which shows all attributes of an element, and the *Code View* Ⓕ, where the corresponding lines in the PLEX source code are highlighted.

More complex reengineering and re-design analyses and transformations are available from the according entries in the menu bar at the top of the window. E.g. from the *Redesign* menu one can trigger the inclusion of the *signal list* (cf. p. 2) as additional information into the structure graph. For each exchanged signal between block nodes, an edge is generated which is labeled with numbers indicating how many signals are exchanged between these blocks. This allows to abstract from block content and to regard only the block external signal flow, which is helpful when a large amount of blocks is considered.

The tool also offers different kinds of re-designs, such as merging of two or more subsystems, splitting of a subsystem into two subsystems, and moving one or more blocks to another subsystem. Similar re-designs for the contents of single blocks exist as well. The algorithms are based on different clustering techniques and effectively support the process of improving the system architecture. Each re-design is separated into two steps. First, the graph is analyzed and a re-design transformation suggested which still can be manipulated by the user after considering the semantics of the program. In the second step, the actual graph transformation is performed.

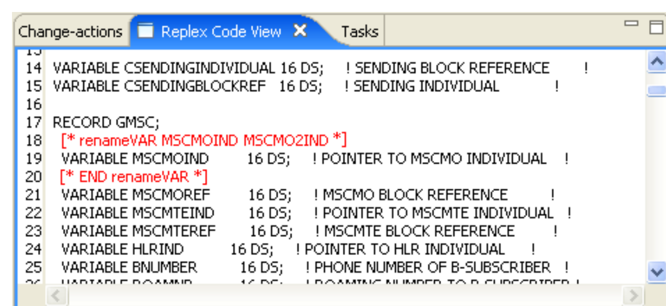

Figure 5: The annotated PLEX source code

After modifying the structure graph, the changes must be propagated to implementation level. From the current graph structure and the node attributes, the tool can derive changes, which must be performed on source code level. As described in section 2, we first generate an annotated version of the PLEX code by adding information about the required changes. Figure 5 shows how the information is stored in the code. In this example, we performed a rename operation. We

rename the variable *MSCMOIND* from Figure 4 to *MSCMO2IND*. Each modification is enclosed by two lines: the first line defines the modification type and the parameters, the second marks the end of the source code area to be considered during the transformation. The TXL-specific command lines are surrounded by *[* *]*. These files are the input for the TXL transformation tool, which generates the new PLEX code, containing all re-design modifications.

Figure 6 shows another feature of the RePLEX tool: a state machine extracted from the PLEX code of the GMSC block. In the telecommunication industry, the behavior of software components is often modeled in terms of state machines. Each state machine is realized in one PLEX block; by convention, their behavior is simulated by the enumeration variable STATE that changes its value every time a signal arrives or leaves the block. By analyzing access to this variable, we can derive a state machine from the structure graph to allow further analysis in the reverse engineering process. Marburger [Mar04] describes the algorithm in detail.

State machines are visualized in an own view, the *DFA tab*. This view can be opened by selecting a block and using the *DFAExtractor* from the *Tool Bar*.

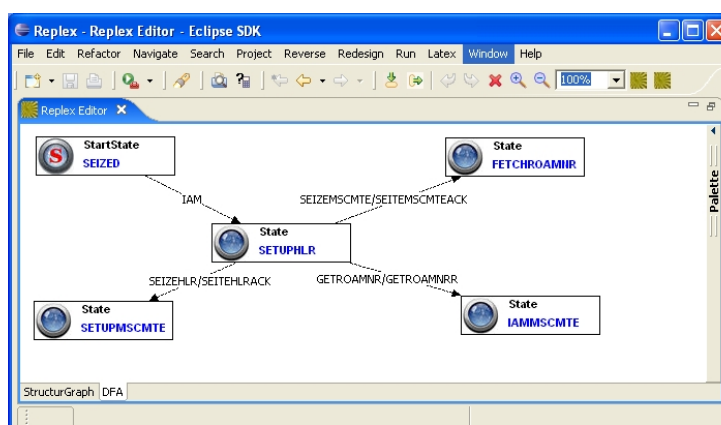

Figure 6: The state machine for block GMSC

This overview showed only a small part of the currently implemented and future functionality, partly already realized in the former PROGRES prototype [Mos06].

5 Conclusion & Outlook

In this paper, we presented a graph-based tool for reengineering telecommunication systems. We explained the underlying object-oriented model and how we use graph transformation rules with FUJABA to specify the tool's functionality. The tool supports the reverse engineering and restructuring process by recovering the actual architecture and propagating the modifications from the architecture level back to PLEX source code. From the FUJABA specification a Java prototype is generated, which is accessible via a GUI based on the Graphical Editor Framework plug-in for the Eclipse workbench.

From our experience with the formal specification of functionality with high level specification languages, we can draw the conclusion that languages like FUJABA or PROGRES allow even

novice developers to realize a complex tool within short time. After first orientation in FUJABA, the basic specification took two students roughly two days. Further complex features could be added within hours. The implementation of the GUI with GEF was straight forward and fairly easy, but quite some hand-coding was necessary. This took ten students about eight workdays. We believe that a parameterizable GUI could be generated from the FUJABA specification in the future.

Further work concerns the development of more re-design transformations, partly already implemented for the PROGRES-based prototype. In the context of the E-CARES project, especially re-design modifications improving the real-time performance of PLEX systems are interesting. On the other hand, our approach for the TXL-based source code transformation must be generalized. The tool is already able to generate new PLEX code for the given set of corresponding graph modifications, but we are still missing a prove that our approach can handle all possible architecture modifications.

Bibliography

[Ber00] E. Berk. JLex: A lexical analyzer generator for Java(TM). Department of Computer Science, Princeton University, Sept. 2000. http://www.cs.princeton.edu/ appel/modern/java/JLex/current/manual.html.

[BJSW02] B. Böhlen, D. Jäger, A. Schleicher, B. Westfechtel. UPGRADE: A Framework for Building Graph-Based Interactive Tools. *Electr. Notes Theor. Comput. Sci.* 72(2), 2002.

[CHP91] J. R. Cordy, C. D. Halpern-Hamu, E. Promislow. TXL: A Rapid Prototyping System for Programming Language Dialects. *Computer Languages* 16(1):97–107, Jan. 1991.

[Ecl06] Eclipse Consortium. Eclipse. 2006. http://www.eclipse.org.

[EKRW02] J. Ebert, B. Kullbach, V. Riediger, A. Winter. GUPRO – Generic Understanding of Programs: An Overview. *Electronic Notes in Theoretical Computer Science* 72(2), 2002. URL: http://www.elsevier.nl/locate/entcs/volume72.html.

[FNTZ98] T. Fischer, J. Niere, L. Torunski, A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In Engels and Rozenberg (eds.), *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*. LNCS 1764, pp. 296–309. Springer, Nov. 1998.

[FUJ99] FUJABA – From UML to Java and Back Again. 1999. http://www.fujaba.de/.

[FUJ05] FUJABA Tool Suite RE. 2005. http://wwwcs.uni-paderborn.de/cs/fujaba/projects/reengineering/.

[GEF06] GEF – Graphical Editing Framework. 2006. http://www.eclipse.org/gef/.

[GSR] L. Geiger, C. Schneider, C. Record. Template- and Modelbased Code Generation for MDA-Tools. 3rd International Fujaba Days 2005, Paderborn, Germany.

[KLR96] S. Kelly, K. Lyytinen, M. Rossi. MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In Constantopoulos et al. (eds.), *CAiSE*. Lecture Notes in Computer Science 1080, pp. 1–21. Springer, 1996.

[Kos00] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. Doctoral thesis, Institute of Computer Science, University of Stuttgart: Stuttgart, Germany, Stuttgart, Germany, 2000. 414 pp.

[Mar04] A. Marburger. *Reverse Engineering of Complex Legacy Telecommunication Systems*. Shaker Verlag, Aachen, Germany, 2004. ISBN 3-8322-4154-X.

[Min02] M. Minas. Specifying Graph-like Diagrams with DIAGEN. *Electr. Notes Theor. Comput. Sci.* 72(2), 2002.

[Mos06] C. Mosler. E-CARES Project: Reengineering of Telecommunication Systems. In Lmmel et al. (eds.), *Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)*. LNCS 4143, pp. 437–448. Springer, Braga, Portugal, 2006.

[MVDJ05] T. Mens, N. Van Eetvelde, S. Demeyer, D. Janssens. Formalizing refactorings with graph transformations. *Journal on Software Maintenance and Evolution: Research and Practice*, pp. 247–276, 2005.

[MWT94] H. A. Müller, K. Wong, S. R. Tilley. Understanding Software Systems Using Reverse Engineering Technology. In *The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences ACFAS 1994*. Pp. 41–48. Montreal, Canada, May 1994.

[NSW^{+}02] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, J. Welsh. Towards Pattern-Based Design Recovery. In *Proc. of the 24th International Conference on Software Engineering (ICSE), Orlando, Florida, USA*. Pp. 338–348. ACM Press, May 2002.

[SK06] A.-T. Schreiner, B. Kühl. jay – a yacc for Java. homepage, 2006. URL: http://www.informatik.uni-osnabrueck.de/alumni/bernd/jay/.

[SWZ99] A. Schürr, A. J. Winter, A. Zündorf. The PROGRES Approach: Language and Environment. In Ehrig et al. (eds.), *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*. Volume 2, pp. 487–550. World Scientific: Singapore, 1999.

[Tae99] G. Taentzer. AGG: A tool environment for algebraic graph transformation. In *Proceedings AGTIVE 99*. LNCS 1779, pp. 481–488. Springer: Heidelberg, Germany, Kerkrade, Netherlands, 1999.

[Wen99] J. Wennersten. PLEX-C Language Description. Ericsson Telecom AB, 1999. EN/LZB 101 1903 R4B.