JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. 2002; **14**:257–292 *Prepared using* smrauth.cls *[Version: 2001/03/16 v1.01]*

Research

Graph-based tools for re-engineering

Katja Cremer[1], André Marburger[2], Bernhard Westfechtel[2,*]

[1] *Ericsson Eurolab Deutschland GmbH, Ericsson Allee 1, D-52134 Herzogenrath*
[2] *Informatik III, RWTH Aachen, D-52056 Aachen*

SUMMARY

Maintenance of legacy systems is a challenging task. Often, only the source code is still available, while design or requirements documents have been lost or have not been kept up-to-date with the actual implementation. In particular, this applies to many business applications which are run on a mainframe computer and are written in COBOL. Many companies are confronted with the difficult task of migrating these systems to a client/server architecture with clients running on PCs and servers running on the mainframe. REforDI (REengineering for DIstribution) is a graph-based environment supporting this task. REforDI provides integrated code analysis, re-design, and code transformation for COBOL applications. To prepare the application for distribution, REforDI assists in the transition to an object-based architecture, according to which the source code is subsequently transformed into Object COBOL. Internally, REforDI makes heavy use of generators to reduce the implementation effort and thus to enhance adaptability. In particular, graph-based tools for re-engineering are generated from a formal specification which is based on programmed graph transformations.
Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS: reverse engineering, re-engineering, graph transformations, COBOL, legacy systems

1. INTRODUCTION

It is widely known that about 50–80 % of the costs of a software system have to be attributed to *maintenance*. In particular, poor system structure and documentation contribute to these numbers. To perform changes to a software system is inherently difficult if only the source code is available. While strict and disciplined application of software engineering principles could have reduced these problems, this insight does not help if actual software development has not proceeded that way. Therefore, there is an urgent need for *re-engineering tools* which assist in program understanding and restructuring.

* Correspondence to: Bernhard Westfechtel, Informatik III, RWTH Aachen, D-52056 Aachen, Germany, `bernhard@i3.informatik.rwth-aachen.de`

Received 7 December 2001
Revised 27 May 2002

While modern programming languages such as Java and C++ are becoming more and more widespread, a considerable amount of code being operational today is still written in FORTRAN and COBOL. The latter has been used extensively for programming business applications in banks, insurance companies, administrations, etc. Originally, these applications were programmed for a centralized environment consisting of a mainframe computer connected with dumb terminals. Later on, companies were faced with the challenge of taking advantage of more modern equipment such as PCs, laptops, etc. operating in a distributed environment. However, migration to a *client/server architecture* is far from easy because units of distribution are hard to identify due to missing separation of concerns.

The *REforDI* environment (*RE*engineering *for DI*stribution) [1, 2, 3] assists with this task by providing integrated tools for code analysis, re-design, and source code transformation. REforDI is fed with the source code of an application written in COBOL 85 (or COBOL 77). The source code is parsed and analyzed, resulting in a graph representation aiding program understanding. Then, a re-design is performed to identify units of distribution. This results in an object-based architecture, according to which the source code is transformed (making use of classes, which are available in Object COBOL). In this way, the application is prepared for distribution, which can be performed with the help of CORBA (see [4] for the latter step).

To realize the REforDI environment, we used *formal specifications* and *generators* extensively. In particular, source code transformations — which were used to transform the original application into Object COBOL — were specified with the help of tree transformation rules. Similarly, we specified the manipulation of graph representations with the help of graph transformations. These graph transformations are used for constructing an internal representation of the source program and for mapping the source program to an object-based architecture. From these specifications, a large amount of code was generated. In this way, implementation effort was reduced significantly such that only small portions of the overall environment had to be hand-coded.

The contributions of REforDI may be summarized as follows:

Re-engineering methodology. REforDI puts a strong emphasis on software architecture. In order to understand a legacy system, it is essential to recover and document its software architecture. Only then may the system be re-designed and re-implemented successfully. Transformations performed directly on the source code level are helpful to some extent. However, they do not improve program understanding, which we consider crucial to reduce maintenance efforts in the long run.

Functionality. REforDI is a highly interactive environment. In particular, re-design is supported through a set of algorithms whose application may be controlled by the user. Moreover, the user may adjust the outcomes of these algorithms manually. We believe that there is no re-design algorithm performing 'best' in all situations. Rather, re-design is a non-deterministic, creative process which should be supported by flexible and interactive tools rather than by automatic processors.

User interface. REforDI offers a graphical user interface which visualizes the software system under study at different levels of abstraction. We consider graphical visualizations as an important aid to support program understanding and re-design.

Realization. Hand-coding of re-engineering tools is a painstaking business. Therefore, we made extensive use of reusable frameworks, formal specifications, and generators to reduce the

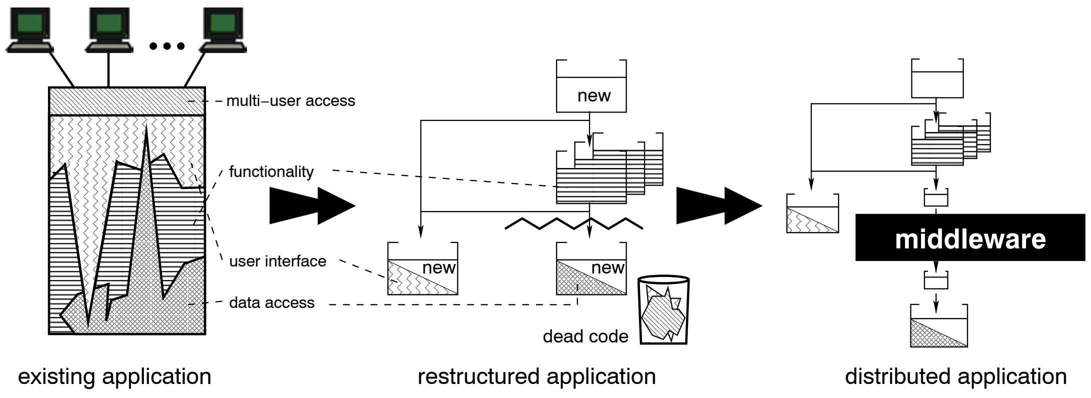


Figure 1. Transition from mainframe applications to distributed systems.

implementation effort. Here, the contributions of REforDI refer to the application of known specification techniques in the re-engineering domain; it was not our intent to develop new specification languages or to extend existing ones.

Therefore, the following kinds of readers may benefit from the material presented in this paper: *researchers* who are interested in re-engineering methodology, *tool users* who are interested in functionality and the user interface, and *tool developers* who are interested in the realization approach followed in REforDI.

The rest of this paper is structured as follows: Section 2 provides an overview of the REforDI environment. Sections 3–5 are devoted to code analysis, re-design, and source code transformation, respectively. Section 6 describes a case study demonstrating the capabilities of the REforDI environment. Section 7 compares related work. Section 8 summarizes some lessons we have learned from our work on REforDI. Finally, Section 9 concludes the paper.

2. REforDI: RE-ENGINEERING FOR DISTRIBUTION

2.1. Background

The research reported in this paper was conducted in a joint project with two German companies: Aachen-Münchener Versicherungen is a large insurance company; GEZ (Gebühreneinzugszentrale) is responsible for collecting radio and TV usage charges. Both companies run large and old COBOL applications on mainframe computers. For various reasons, it is desirable to make use of PCs, laptops, etc. operating in a distributed environment. For example, mobile work is to be supported (e.g., insurance agents serving their customers), operations are to be performed locally to reduce the load on the central computer, etc.

Usually, it is not economically feasible to rewrite large applications from scratch. Rather, they are migrated to a client/server architecture. The left-hand side of Figure 1 shows the existing application consisting of user interface, application logic, and database accesses. Typically, these parts are mingled together rather than separated cleanly. To prepare the application for distribution, it has to be restructured to some extent (separation of concerns). This involves the design of an object-based architecture as well as changes at the code level (both implementation of new functions and removal of redundant functions). Subsequently, the application is divided into clients and server, making use of middleware technology such as CORBA (distributed objects). Here, objects serve as units of distribution. The cut between clients and server can be performed at different levels. In any case, user interface code will be moved to the clients, and database accesses will be handled by the server. The distribution of the application logic depends on the specific requirements and constraints to be met.

Within the REforDI project, two PhD theses were completed. One of them [3] deals with the first step of Figure 1: restructuring the application such that it is prepared for distribution. The other one [5] is concerned with the second step: distribution of the application with the help of transformations performed at the architectural level. This paper focuses on the results of the first PhD thesis. Thus, we discuss how an application may be prepared for distribution by migrating it to an object-based architecture. However, we will not describe how objects may be distributed in the second step. Please note that restructuring could also be applied for other reasons, e.g., to obtain a modular architecture which may be maintained more easily. As a consequence, restructuring may be performed independently from and without a subsequent distribution step.

2.2. The REforDI approach

An overview of the REforDI approach is presented in Figure 2. The re-engineering process as supported by REforDI consists of three steps which are described below.

The first step, called *code analysis*, is based almost exclusively on the source code, which is written in COBOL 85 (or COBOL 77). In addition, it relies on external information concerning the decomposition of the application into subsystems, which cannot be obtained from the source code itself. Code analysis proceeds completely automatically. It extracts information which can be inferred reliably from the source code. The result of code analysis is a *system structure graph* which represents the application at different levels of granularity. The system structure graph represents the coarse-grained organization of the application, which is composed of subsystems, programs, divisions, sections, and paragraphs. In addition, it covers fine-grained information, concerning, e.g., data accesses of procedures. Finally, the nodes of the system structure graph also contain references to the fragments of the source code which they represent. Please note that the information gathered in the system structure graph is dictated by the needs of subsequent re-engineering steps, namely re-design and re-implementation.

The second step, called *re-design*, is concerned with the construction of an object-based architecture. Re-design can be performed either interactively, or the user may choose among multiple re-design algorithms which automate this step. While code analysis closely sticks to the source code, re-design identifies meaningful abstractions which can be grouped into logical modules. Of course, this step is highly non-deterministic. Many algorithms have been proposed for this purpose, a few of which were implemented in the REforDI environment. Re-design yields an object-based architecture which is represented in an *architecture graph*. Thus, the architecture graph is located at a higher level of

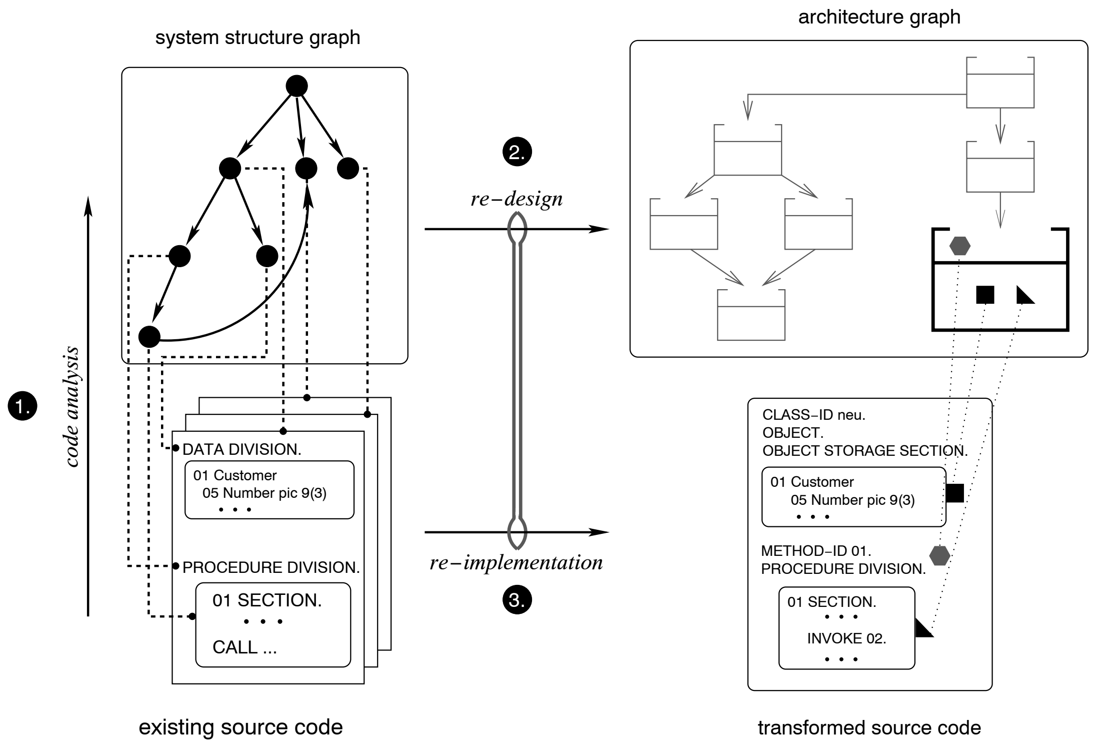


Figure 2. The REforDI approach.

abstraction (and granularity) than the system structure graph: the former contains logical modules, while the latter is composed of physical program units.

In addition to the structure graph and the architecture graph, a *correspondence graph* is created (which is not shown in Figure 2 to keep the figure clear). The correspondence graph maintains relationships between elements of the structure graph and the architecture graph, respectively. In contrast to code analysis (which is automated), re-design proceeds interactively, and m:n relationships may be established. For example, multiple programs may be aggregated into a single class (n:1); conversely, a single program may be mapped onto multiple classes (1:n). This is illustrated in Figure 11, which will be described in a more detailed way in Section 4.4. It is crucial to maintain the relationships between structure graph and architecture graph for the following reasons. Firstly, the relationships document design decisions which should be recorded for traceability. Secondly, they are used for re-implementation in order to establish the connection to the source code fragments. Thirdly, they are employed for incremental re-design when the source code has changed.

The third and last step, called *re-implementation*, deals with the transformation of the source code. By and large, this step works automatically. The architecture created by re-design is traversed and transformed into code fragments which are combined with old fragments from the existing application. Re-implementation is performed with the links from the structure graph into the source code and the relationships between architecture graph and structure graph. Like re-design, re-implementation raises the abstraction level inasmuch as the application code is migrated to Object COBOL. That is, the restructured application makes logical units of abstraction (classes) explicit that were contained only implicitly in the original application.

The REforDI approach diverges in several ways from the conceptual framework introduced by Chikofsky and Cross (which is frequently referenced in the re-engineering community). In [6], among others the following terms are defined. *Re-documentation* 'is the creation or revision of a semantically equivalent representation within the same abstraction level'. It is applied at the code level and is concerned, e.g., with pretty printing, creation of diagrams (call graphs, control and data flow graphs), etc. *Design recovery* 'is a subset of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations of the subject system to identify meaningful higher level abstractions beyond those obtained directly by examining the system itself'. Finally, *re-structuring* 'is the transformation from one representation to another at the same abstraction level, while preserving the subject's external behavior (functionality and semantics)'. Re-structuring can be classified further according to the level of abstraction on which it is performed (re-design and re-implementation).

In REforDI, we do not distinguish between design recovery and re-design: our notion of re-design covers both. First, our eventual goal is to migrate the target application rather than to merely understand it. Second, we believe that design recovery alone has only limited benefits when the identified logical abstractions are not explicitly represented in the source code (which requires re-implementation, which is coupled to re-design). As a consequence, re-design in REforDI raises the level of abstraction, while re-design according to Chikofsky and Cross retains it. Similarly, re-implementation in REforDI is not a strictly horizontal transition since it involves the migration to a language supporting data abstraction. Finally, please note that code analysis corresponds to re-documentation rather than to design recovery: the system structure graph represents the source code (at different levels of granularity), but does not represent logical abstractions which are not explicitly contained in the source code.

2.3. Realization

To reduce the implementation effort, we employed reusable components and generators extensively. Figure 3 gives an overview of our realization approach. Below, we explain in general terms which components and generators we used in our implementation. How they were actually used in the REforDI environment, will be detailed in the following sections.

At the source code level, we apply the *TXL* toolkit [7][†]. With the help of TXL, parsers, transformation tools, and unparsers may be generated. A parser is generated from a context-free grammar and creates a parse tree which can be manipulated with the help of tree transformation rules. The modified parse tree is transformed into text with the help of an unparser.

[†]In our implementation, we used an earlier version of TXL which was available for free. The current version of TXL is commercial. Note that only tools generated with TXL may be purchased (rather than the generator itself).

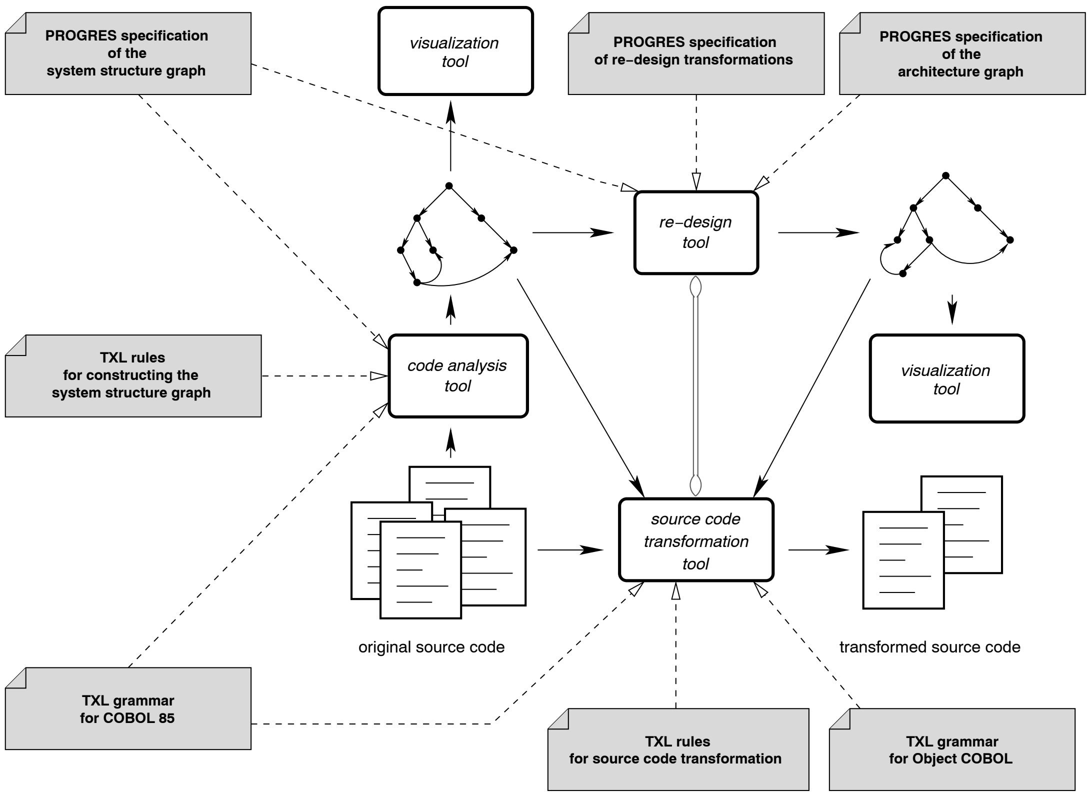

PROGRES specification of the system structure graph
visualization tool
PROGRES specification of re–design transformations
PROGRES specification of the architecture graph
re–design tool
TXL rules for constructing the system structure graph
code analysis tool
visualization tool
source code transformation tool
original source code
transformed source code
TXL grammar for COBOL 85
TXL rules for source code transformation
TXL grammar for Object COBOL

Figure 3. Realization with the help of generators.

At the design level, REforDI employs graphs rather than trees. Here, we make use of the *PROGRES* environment [8, 9]. PROGRES is a specification language which is based on attributed graphs. In PROGRES, both graph structures and graph operations may be specified at a high level of abstraction. The PROGRES environment offers tools for editing, analyzing, and interpreting specifications, as well as a compiler which generates C code.

TXL and PROGRES are used to generate the internal processing components of the REforDI environment. The user interface is based on the *ECU* framework [10] which supports the development of graphical, interactive tools on top of the code generated by the PROGRES compiler. In particular, ECU offers reusable visualization tools that can be adapted with little effort.

Due to the application of reusable components and generators, we were able to reduce the amount of hand written code to a nearly negligible size (less than 1000 lines of C code). In contrast, most effort went into the TXL and PROGRES specifications. The TXL specifications comprise about 8000 lines. The PROGRES specifications cover about 120 pages (both text and graphics), corresponding to a text dump of about 9000 lines. More than 100000 lines of code were generated by the PROGRES compiler.

3. CODE ANALYSIS

3.1. Parsing and information extraction with TXL

Below, we describe how TXL is applied to code analysis. We present the main ideas concisely at an informal level (this also holds for Section 5). Our main focus lies on the application of graph technology, which will be discussed more deeply and formally in Section 3.2 and in particular in Section 4.

Parsing and information extraction works as illustrated in Figure 4. The original source code is fed into a parser which is generated from a TXL grammar for COBOL 85. The parser creates a parse tree which is transformed subsequently. We employ tree transformations to augment the parse tree with extracted information which is used to build the system structure graph. The shaded region highlights a subtree which represents a fact for the analyzed portion of the source code. In our example, the fact says that the paragraph `INITIALIZE-CALCULATOR` is contained in the section `START` of the program `CALCULATION` and carries the label `10`. After the parse tree has been augmented with facts, the unparser generates text for the complete parse tree. A simple text processing tool extracts the facts from the overall text (all lines starting with `$`). Finally, for each fact a corresponding graph transformation is executed. In our example, a node is created which represents the paragraph and carries attributes for the name and the start label. The node is also embedded into its context (i.e. it is connected with the surrounding section node), which is not shown in the figure.

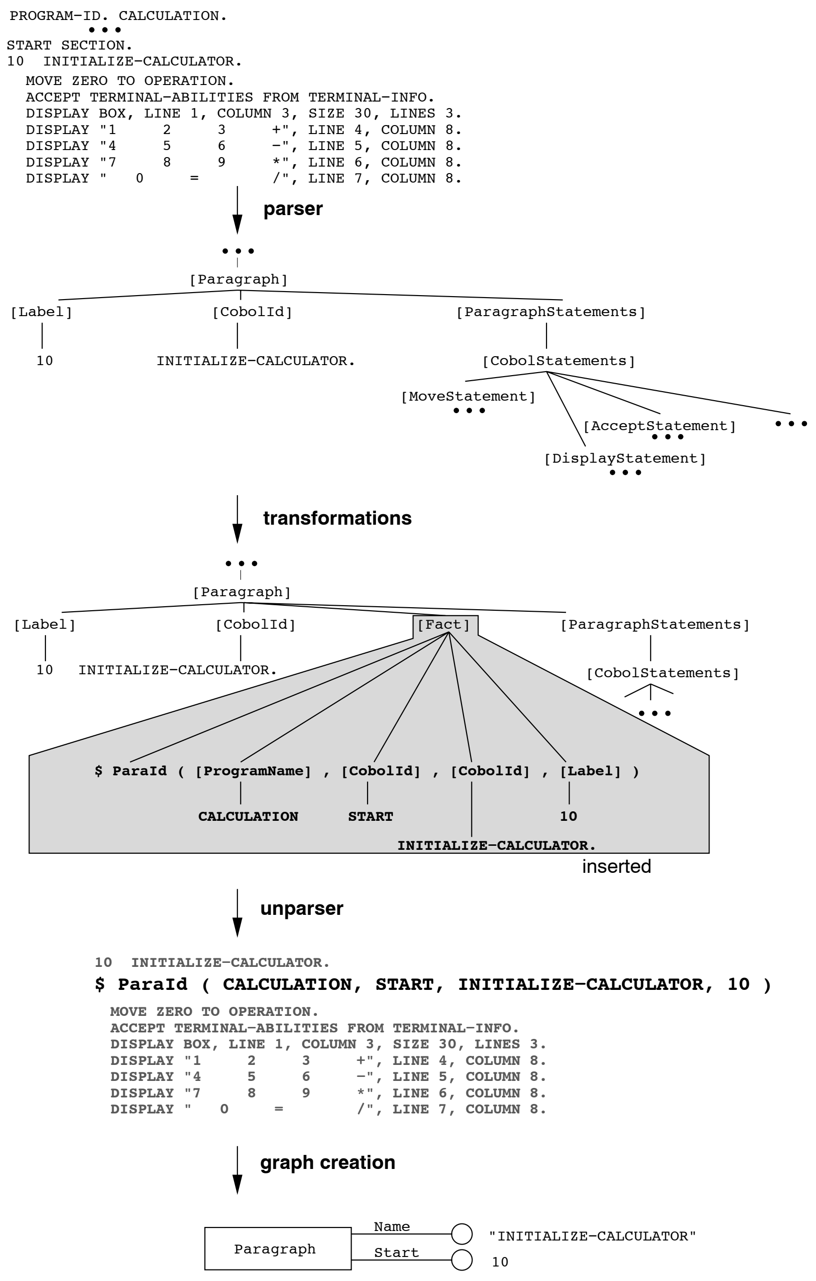
3.2. The system structure graph

The system structure graph represents all information that is extracted from the source code for program understanding and subsequent re-design. In the following, we explain how system structure graphs are defined in PROGRES.

PROGRES is a specification language which is based on *attributed graphs*. An attributed graph consists of typed nodes and edges. The type of a node defines the attributes which are attached to that node. Edges are binary, directed relationships which do not carry attributes. The type of an edge determines the permissible types of source and target nodes, as well as the cardinality of the binary relationship. Since edges do not carry attributes, attributed relationships cannot be represented directly as edges. Rather, they have to be simulated by nodes and adjacent edges.

In PROGRES, the elements of attributed graphs are defined in a *graph schema*, which plays the same role as a database schema in a database management system. The graph schema defines types of nodes, edges, and attributes. For the sake of brevity, we will not go into graph schemas in this paper (see [3]).

The system structure graph represents the coarse structure of the application in terms of subsystems, programs, sections, etc. However, it is not confined to the coarse-grained level; rather, it is refined down to the level of individual COBOL statements (`CALL`, `READ`, `PERFORM`, etc.). All nodes of the system structure graph contain references to the respective source code fragments. From the system structure



```
PROGRAM-ID. CALCULATION.
        • • •
START SECTION.
10  INITIALIZE-CALCULATOR.
  MOVE ZERO TO OPERATION.
  ACCEPT TERMINAL-ABILITIES FROM TERMINAL-INFO.
  DISPLAY BOX, LINE 1, COLUMN 3, SIZE 30, LINES 3.
  DISPLAY "1     2     3     +", LINE 4, COLUMN 8.
  DISPLAY "4     5     6     -", LINE 5, COLUMN 8.
  DISPLAY "7     8     9     *", LINE 6, COLUMN 8.
  DISPLAY "   0     =        /", LINE 7, COLUMN 8.
```

```
10  INITIALIZE-CALCULATOR.
$ ParaId ( CALCULATION, START, INITIALIZE-CALCULATOR, 10 )
  MOVE ZERO TO OPERATION.
  ACCEPT TERMINAL-ABILITIES FROM TERMINAL-INFO.
  DISPLAY BOX, LINE 1, COLUMN 3, SIZE 30, LINES 3.
  DISPLAY "1     2     3     +", LINE 4, COLUMN 8.
  DISPLAY "4     5     6     -", LINE 5, COLUMN 8.
  DISPLAY "7     8     9     *", LINE 6, COLUMN 8.
  DISPLAY "   0     =        /", LINE 7, COLUMN 8.
```

Figure 4. Parsing and information extraction with TXL.

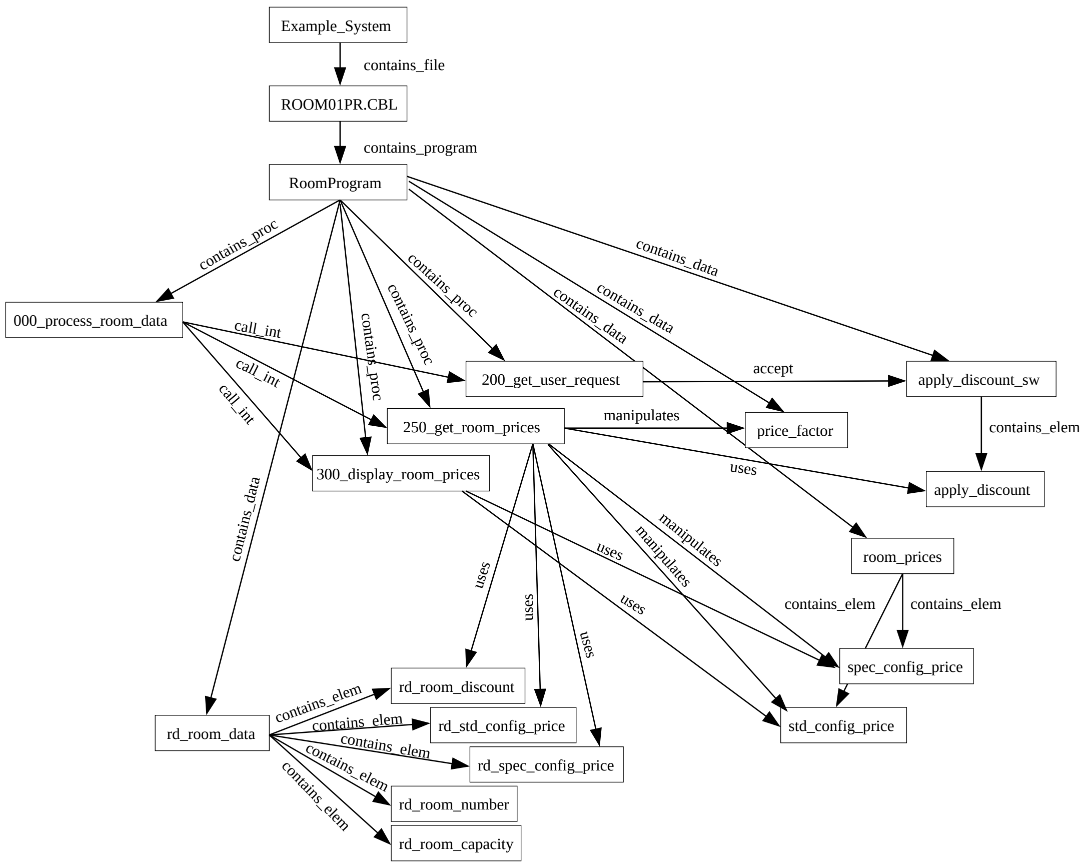
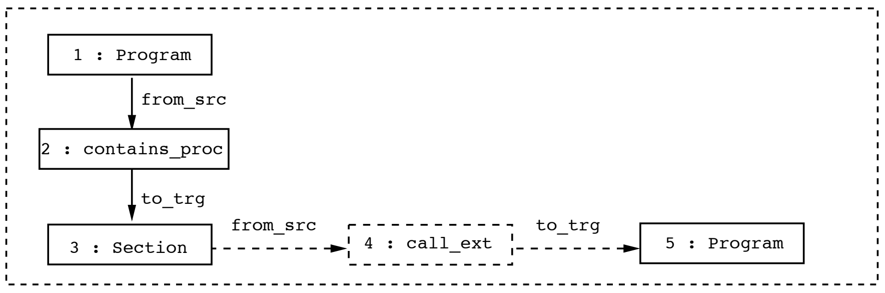


Figure 5. Example of a graph representation of a COBOL program.

graph, different views may be generated that may be visualized at the user interface. Furthermore, different re-design algorithms make use of different parts of the system structure graph.

Figure 5 shows an example of a system structure graph. The figure is simplified inasmuch as node types are omitted and only name attributes are shown (inside the boxes representing the nodes). Moreover, relationships are represented by arrows even if they have to be represented by nodes and adjacent edges internally. The sample graph has been constructed from a part of a room reservation system. The program represented here calculates the price for some room which has already been selected. This program, called `RoomProgram`, is contained in the file `ROOM01PR.CBL`. Its main procedure `process_room_data` calls three auxiliary procedures. `get_user_request` reads a user input for deciding whether a discount should be offered to this customer. `get_room_prices` reads the `apply_discount` variable to calculate a price factor, reads the room data, and calculates two

```
production insert_call (program_id, src_section, trg_program : string; line : integer) =
```



```
    condition 1.Name = program_id; 3.Name = src_section; 5.Name = trg_program;
    transfer 4.Start := line; 4.End := line; 4.File := 1.File;
end;
```

Figure 6. Example of a graph transformation.

prices (standard and special rate), which are finally output by `display_room_prices`. Please note that data records are represented by nodes that are connected to the respective nodes for their elements.

PROGRES offers *graph transformation rules* — also called *productions* — for specifying changes of attributed graphs at a high level of abstraction. A production consists of a left-hand side — the graph pattern to be replaced — and a right-hand side — the replacing graph pattern. In addition, it may have some optional parts such as input and output parameters, conditions (on node attributes), and transfer rules (for assigning new values to node attributes).

An example of a production is given in Figure 6. This production is executed when the TXL processor has created a fact for an external program call. To save space, we have merged the left-hand side and the right-hand side into a single diagram. Solid nodes and edges occur both on the left-hand side and on the right-hand side; i.e. they are replaced identically. Dashed nodes and edges represent new graph elements occurring only on the right-hand side. Each node is identified by a unique number (`1`, `2`, ...); furthermore, it is assigned a node type (`Program`, `Section`, ...). In contrast to Figure 5, attributed relationships are shown as they are actually represented internally (as edge-node-edge patterns).

The production is supplied with the names of a program and a section from where the external call is invoked, the name of the called program, and the line number of the `CALL` statement. In the left-hand side, nodes `1`, `3`, and `5` represent the calling program, the calling section, and the target program, respectively. These nodes are identified through their name attributes (see condition part below the graph pattern). Furthermore, the left-hand side checks whether the section supplied as second parameter is actually contained in the program supplied as first parameter. The composition relationship is represented by node `2`. If all of these conditions are fulfilled, the node `4` is created which represents the call relationship. Furthermore, the attributes `Start`, `End`, and `File` are assigned such that the source code of the `CALL` statement may be retrieved.

4. RE-DESIGN

In this section, we address the challenging problem of identifying logical abstractions and designing an object-based architecture according to which the source is modified in the next step (re-implementation). Our goal is to transform the system structure graph into a corresponding architecture graph. Section 4.1 describes the target of the transformation process (the architecture graph). Section 4.2 introduces the theoretical background of our approach to re-design. Section 4.3 is devoted to manual (interactive) re-design. Finally, Section 4.4 describes some algorithms which automate the re-design process.

4.1. The architecture graph

The architecture graph is based on an *architecture description language* (ADL) which incorporates concepts commonly found in object-oriented modeling languages. In this ADL, a software system may be decomposed recursively into subsystems, each of which consists of a set of modules. Each module is composed of a set of interfaces and classes (implementations of interfaces). An interface defines a set of methods with their parameters, and a class contains method implementations. Both classes and interfaces may import from other interfaces. Furthermore, inheritance hierarchies may be built up for both interfaces and classes (i.e. interface inheritance is distinguished from implementation inheritance).

The architecture graph represents a software architecture defined in the ADL as an attributed graph. In the context of this paper, a detailed understanding of the underlying ADL is not required (see [3]). Moreover, please note that not all features of the ADL are exploited in REforDI. In particular, we do not use inheritance since we restrict our ambition to an object-based rather than an object-oriented architecture.

4.2. Theoretical background

The creation of the architecture graph from the system structure graph is an instance of a more general problem: the coupling of graphs of different types among which non-trivial relationships have to be maintained. It is crucial to represent these relationships explicitly. In the case of re-design, they record the decisions which humans or tools have made concerning the mapping of the original system structure to a software architecture.

Therefore, we insert a third graph in between the source and the target graph. The *correspondence graph* consists of nodes representing mapping decisions. In general, a mapping is established between subgraphs of source and target graph, respectively. A correspondence node is connected to the nodes of these subgraphs by correspondence edges. In addition, correspondence nodes may be mutually connected by edges representing dependencies between mapping decisions.

In order to reconstruct an object-based software architecture from the source program, we have to group the code elements into classes. That is, we have to figure out which declarations and statements form a logical unit of encapsulation. In general, relationships may be many-to-many. For example, a program may be split into multiple classes, or sections from multiple programs may be combined into one class. There is no unique solution to the re-design problem; in contrast, a great variety of re-design algorithms has been proposed [11]. As we have already pointed out in Section 2.2, re-design requires user interaction. Even if an automatic algorithm is run to produce an architecture, we have to

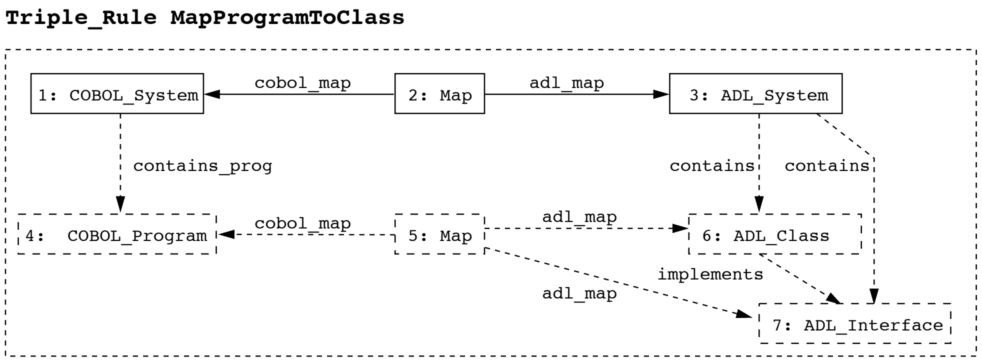


Figure 7. Example of a triple rule.

consider the result as a proposal which the user may modify subsequently to fit his needs. For these reasons, it is crucial to record the relationships between code elements and architectural elements in the correspondence graph.

In REforDI, the correspondence graph is used in multiple ways. During the re-design process, it records those mapping decisions that have already been performed. In this way, it can be determined which elements of the system structure graph still have to be transformed and which constraints have to be taken into account for the remaining mapping decisions. After re-design, the correspondence graph is used for documenting the mapping decisions (e.g., by visualizing the relationships between COBOL procedures and ADL methods). Finally, the correspondence graph is used for re-implementation, which is basically concerned with traversing the architecture graph and collecting and combining code fragments of the original application (see Section 5).

To specify graph correspondences, we make use of *triple graph grammars* [12]. A triple graph grammar refers to three graphs: a source graph, a target graph, and a correspondence graph. A *triple rule* is applied to these graphs simultaneously. It couples productions on the source and the target graph and establishes relationships between them in the correspondence graph.

A simple example is given in Figure 7[‡]. This rule specifies the 1:1 mapping of a program to a class. The left, middle, and right parts deal with the source, correspondence, and target graph, respectively. To apply this rule, the root nodes of the system structure graph and the architecture graph (nodes 1 and 3, respectively) must have been connected by the correspondence node 2, i.e., they must have been mapped onto each other[§]. The rule creates a program node 4 in the system structure graph, nodes for an interface and a class implementing that interface in the architecture graph (nodes 7 and 6, respectively), a mapping node 5 in the correspondence graph, and edges to the corresponding nodes.

[‡]In this informal example, we have taken the liberty to represent relationships as edges which are actually represented as nodes (this also applies to the next figure).
[§]Concerning the left-hand and right-hand side, we use the same notation as in Figure 6.

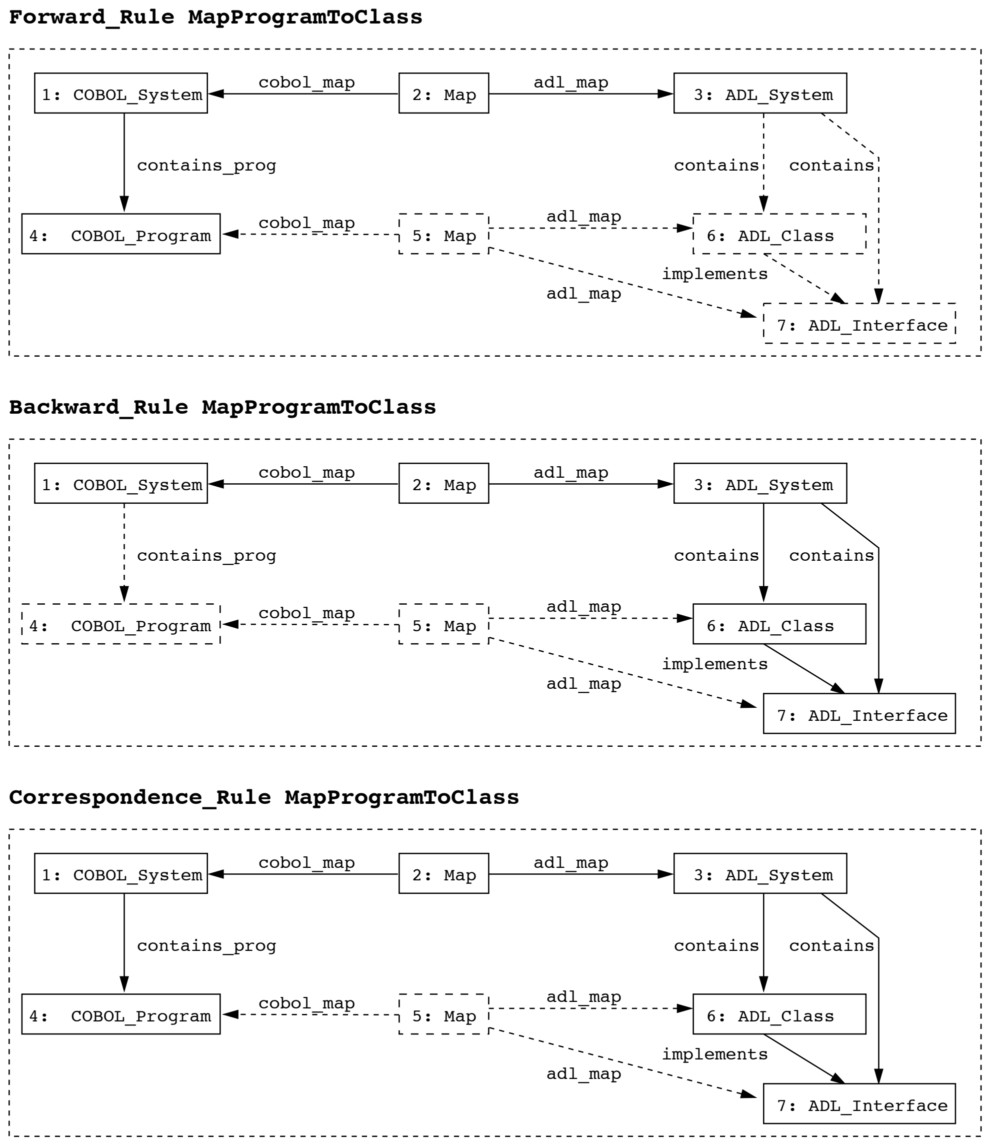


Figure 8. Examples of derived rules.

A triple rule specifies the synchronous construction of all participating graphs. Therefore, we call such a rule *synchronous*. However, in many situations source, target, and correspondence graph are not built up simultaneously. Figure 8 shows three rules that can be derived from the single triple rule of Figure 7.

- A *forward rule* assumes that the source graph has already been constructed, and augments the target graph and the correspondence graph accordingly. The forward rule is constructed from the synchronous rule by converting created nodes of the source graph to given nodes (node `4` in the example). With the help of forward rules, we may specify transformers from the source graph into the target graph.
- A *backward rule* works in the opposite direction. Analogously, all nodes of the target graph are considered as given (nodes `6` and `7` in the example). With the help of backward rules, we may specify transformers from the target graph into the source graph.
- Finally, a *correspondence rule* assumes that both source and target graph have already been built up, and establishes correspondences a posteriori. In the example, only node `5` is created, while nodes `4`, `6`, and `7` must already be present. With the help of correspondence rules, we may specify *consisteny analyzers* which establish correspondences and check consistency between source graph and target graph.

Thus, the general idea behind triple graph grammars is the following: First, we specify correspondences in a declarative way with the help of synchronous triple rules. From a single synchronous rule, we may derive multiple rules (forward, backward, and correspondence rule) which drive tools for directed transformations and consistency analysis.

In REforDI, however, the generality of this approach is not fully exploited. Rather, the architecture graph is typically constructed from the system structure graph with the help of forward rules. Only those will be discussed below. In addition, the user may first design a new architecture and then establish mappings a posteriori with the help of correspondence rules. Since we do not intend to modify the original source, backward rules are not supported.

4.3. Manual re-design

In this section, we discuss *base operations* for mapping the system structure graph onto the architecture graph. These operations define which elements may be mapped onto each other. They constitute a base layer of operations which are intentionally designed to provide much freedom to the re-designer. In particular, the base layer hardly constrains the grouping of COBOL elements into ADL elements. For example, no attempts have been made to enforce rules such as ‘related procedures must be mapped onto methods of the same class’. These constraints depend on mapping decisions performed in the re-design algorithms to be introduced in Section 4.4.

Thus, the base layer plays the following roles.

- The user may perform a *manual re-design* by invoking base operations interactively. While manual re-design provides a maximum level of flexibility, it suffers from a low level of guidance and support. As a consequence, manual re-design of very large COBOL programs is hardly feasible because of the large number of user interactions.

Table I. Correspondences between COBOL and ADL elements.

COBOL element	*ADL element*
program	class
procedure (section, paragraph, sentence)	method
data element	class attribute, method parameter, method variable
call (external, internal)	method invocation

- The user may select one of the *re-design algorithms* to be described in Section 4.4. These algorithms follow different approaches to structuring the software architecture, but all of them rest on the same layer of base operations. In this way, the user benefits from more high-level support, but the re-design algorithms may perform ‘wrong’ mapping decisions.
- After having run a re-design algorithm, the user may still improve its outcome interactively by invoking base operations as required. We consider this usage mode much more realistic than a completely manual re-design.

Table I provides an overview of the mappings supported by the base layer.

- At the most coarse-grained level, mappings may be established between programs and classes (and interfaces, see Figures 7 and 8). In general, relationships are m:n, i.e. a single program may be mapped onto multiple classes and vice versa.
- The procedure division of a COBOL program may be mapped on different levels of granularity (section, paragraph, or sentence). In any case, the target of a mapping is a class method. Again, m:n mappings may be created.
- A data element may be mapped onto a class attribute, a method parameter, or a local variable of a method.
- Finally, both external and internal calls are mapped onto method invocations since the ADL does not distinguish between them.

To illustrate both the functionality of the base layer and our specification method, two examples of transformation rules are given below. Figure 9 shows a forward rule for mapping a procedure onto a method. The procedure to be transformed and the class for the new method are supplied as parameters, which are used to match nodes `6` and `8`, respectively. The rule is applicable only after an initial rule has been executed to map the overall COBOL system into an ADL system (nodes `1`–`3`). That is, rule applications are ordered implicitly according to their left-hand sides. Nodes `4`, `5`, and `7` represent the surrounding context of the supplied parameters within the COBOL and the ADL system, respectively. Double arrows denote paths, i.e. sequences of edges, whose definitions (by path expressions) we omit. For example, the path from node `1` to `4` means that the program containing the procedure to be transformed must be contained in the overall COBOL system. Analogously, the class to which the new method is to be added must be contained in the ADL system. When the required context exists, the rule is applied, resulting in the creation of a new method node `10`, a mapping node `11`, and node `9`

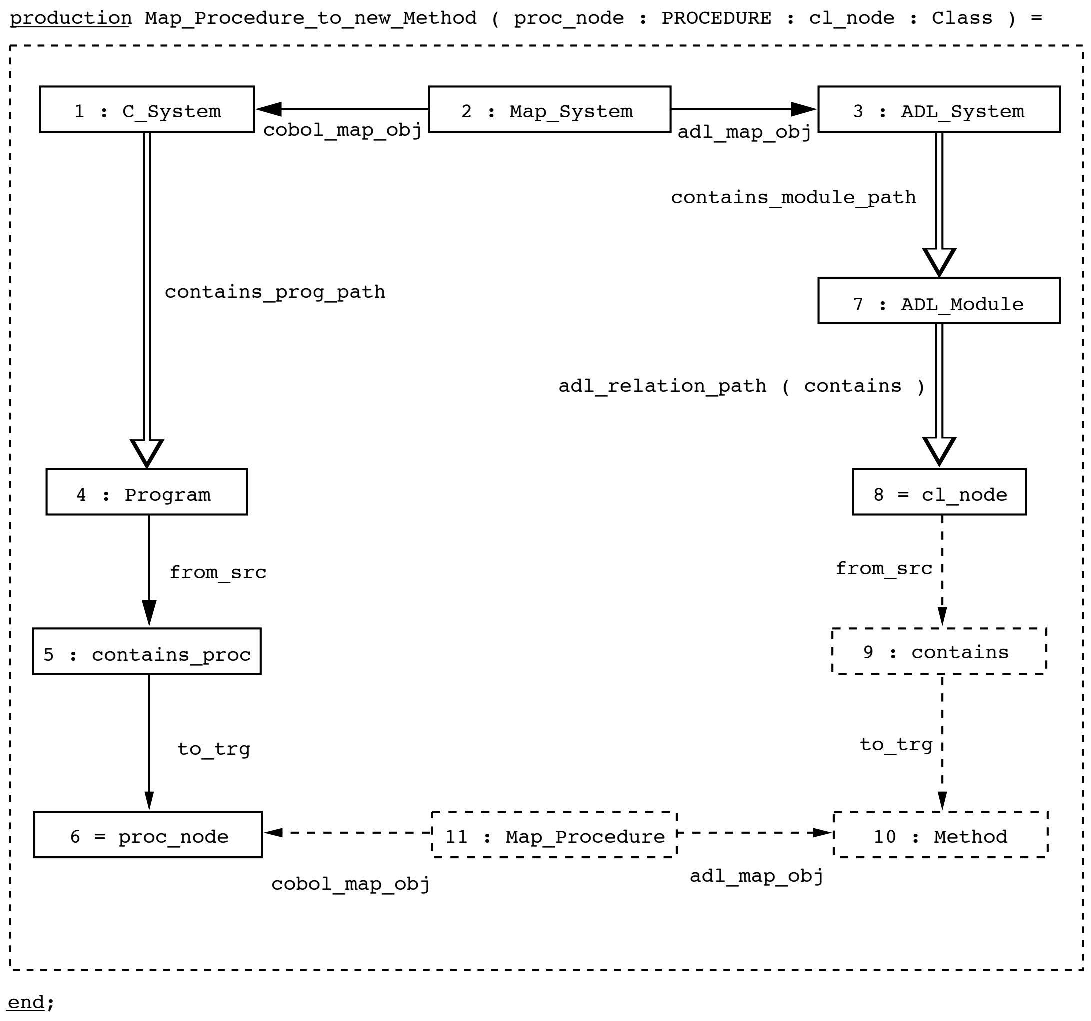


Figure 9. Forward rule for mapping a procedure to a new method.

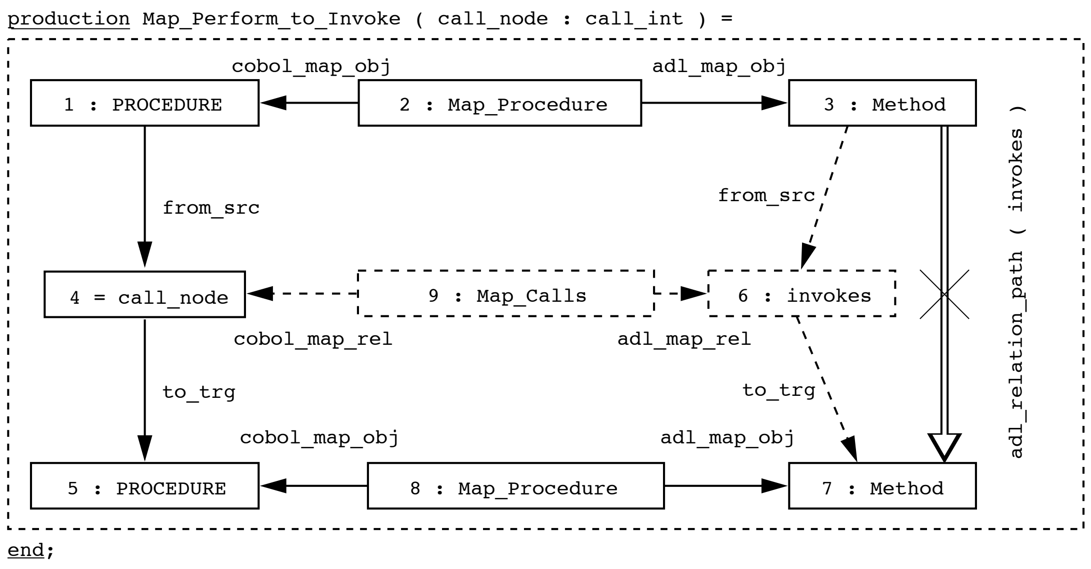


Figure 10. Forward rule for mapping an internal call to a method invocation.

for attaching the new method to the class node 8. Please note that this rule does not enforce any further constraints on the grouping of (methods created from) procedures into classes. These constraints are dealt with in the re-design algorithms to be presented in Section 4.4.

In Figure 10, an internal call (a PERFORM statement) is mapped onto a method invocation. The call to be transformed is supplied as a parameter. The rule assumes that both the calling procedure (node 1) and the called procedure (node 5) have been transformed into methods before (nodes 3 and 7, respectively). The call itself (node 4) is represented as a relationship between calling and called procedure. The call is transformed into an invocation between the corresponding methods (node 6), and the correspondence between the call and the method invocation is recorded with the help of node 9 and its adjacent edges.

Please note the following difference between a call node and a method invocation node: while a call node represents a single call statement, a method invocation node between m_1 and m_2 merely says that m_2 is called somewhere in m_1. As a consequence, multiple calls to the same procedure are mapped onto a single method invocation node. The forward rule displayed in Figure 10 is used for the first call statement (checking that no duplicate method invocations are created[¶]). Further call statements are handled by a similarly working correspondence rule which merely creates a mapping node.

[¶]This is achieved by a negative path between 3 and 7. That is, the rule is applicable only when the methods 3 and 7 are not connected yet by a method invocation.

The transformation rules of Figures 9 and 10 differ in the following respect: mapping a procedure to a method is inherently non-deterministic. Therefore, it requires either user interaction or a heuristic implemented in a re-design algorithm. In contrast, mapping an internal call is deterministic after the calling and the called procedure have been transformed. Invoking transformations of the latter kind may be automated even if the user does not rely on one of the (heuristic) re-design algorithms to be described below.

4.4. Re-design algorithms

On top of these base operations, several re-design algorithms have been implemented which have been proposed in the literature, e.g. in [13, 14, 15]. Since a manual re-design using the base operations is too extensive for large legacy systems, algorithms can support a re-engineering expert in finding candidates for components of a target architecture. In a second step, the re-engineer can interactively use the base operations or tune algorithm parameters to refine the detected components. This procedure accelerates re-design and improves quality by expert intervention according to domain knowledge.

The aim of the approach presented here is to transfer a large legacy COBOL application into an object-based application written in Object COBOL to prepare the application for distribution. Considering large legacy systems, fine-grained re-design algorithms even decomposing program statements are not feasible. Using algorithms of this kind, re-design would be too time-consuming and the resulting data structures would be too complex and not manageable for human interaction. Therefore, we decided to implement algorithms that are more coarse-grained, e.g. by assuming that procedures are atomic elements of re-design. The following algorithms have been implemented.

- The first algorithm implemented performs a complete encapsulation of COBOL programs. COBOL 85 programs are syntactically transformed into Object COBOL classes. It is a simple approach which however may be completely sufficient for distribution (apart from that, no architectural reorganization). Sometimes it is even the only possibility to take full advantage of a legacy system controlled by new system components. This algorithm and the following one were triggered by [15].
- The second approach maps programs to classes with multiple methods. Again, a program is mapped onto a class, but in addition paragraphs/sections are mapped onto exported methods. This is a more fine-grained transformation, which allows one to access the functionality of a class through multiple methods.
- Third, a data-oriented approach has been implemented for programs that have been developed in a data-centered fashion. This algorithm analyzes the data accesses of sections/paragraphs of a program. It forms clusters of sections/paragraphs and data elements based on that analysis and creates corresponding classes. One program is mapped onto multiple classes.
- Since a data-oriented approach produces poor results with some kind of legacy programs, an algorithm basing on functional decomposition has been implemented as well. This algorithm bases on ideas presented in [13], [16] and [17]. It analyzes the call graph of a COBOL program and groups nodes with a single external entry point. For each group, a class is generated. Depending on this structuring, the corresponding data definitions are divided among these classes. Again, one program is mapped onto multiple classes.

To indicate how such algorithms can be specified in PROGRES in a compact manner, the algorithm for the data-oriented approach is explained briefly below[‖]. The data-oriented approach is based on the idea of an object finder (global data) presented by Liu and Wilde in [14]. Similar approaches are described in [19] and [20]. In this approach, for each data element of a program, all routines are collected that reference the data element regarded. The resulting unions are compared pair-wise and possibly united again until all sets of routines are mutually disjoint. In combination with the referenced data elements, each of these sets is considered a so-called object candidate. Furthermore, each object candidate constitutes a separate class.

A (simplified) example is given in Figure 11. Since both `Section 1` and `Section 2` reference the data element `WS_Data 3`, they are inserted into the same cluster. Together with the union of the respective sets of data elements (`WS_Data 1`, ..., `WS_Data 4`), these procedures are mapped onto `Class 1`. Analogously, `Section 3` and its referenced data elements are mapped onto `Class 2`. Please note that the procedures of both object candidates reference disjoint sets of data elements.

The core of the re-design algorithm, the candidature criterion, is described in Figure 12. For coding algorithms, PROGRES provides *transactions*[**]. Like a production, a transaction may have parameters; here, these refer either to single nodes or sets of nodes (cardinality `[0:n]`). Furthermore, local variables may be declared in a *use clause*. In the body of the transaction, statements such as assignments, calls of productions, transactions, etc. are arranged with the help of control structures; here, a *choose statement* is used for branching the control flow.

In the transaction `Produce_Next_Candidate`, `data_node` refers to the next data element to be considered. This data element has to be processed only if it is contained in the set `data_to_process` supplied as another parameter. In this case, all related data elements (`candidate_data`) may be determined in a single step with the help of a *path expression* (step 1). Starting from `data_node`, the path `data_closure_path` performs a transitive closure over references to data elements in both directions (data element $\rightarrow$ procedure and vice versa). The corresponding candidate procedures are calculated in an analogous way (step 2[††]). After that, the transaction `Map_Candidate_Object` is called which creates the corresponding class in the architecture graph (step 3). Finally, the set of processed data elements (`data_to_process`) is returned as an output parameter.

Figure 11 illustrates the application of this transaction to a sample structure graph. Starting from `WS_Data_1`, the candidate data set is obtained in step 1. The corresponding procedures are determined in step 2. The resulting class and the mappings in the correspondence graph are created in step 3. `Class_2` is obtained analogously. Please note that the algorithm also takes procedure calls into account and maps them to method invocations (call `Section 1` $\rightarrow$ `Section 3`).

Applying the data-oriented approach presented above, we faced several problems. First, the algorithm does not create sufficiently small object candidates. Rather it merges different object candidates into a single one. A reason for this behavior can be found in data elements commonly used among different object candidates, e.g., error flags or the like. These data elements where introduced for implementation reasons but do not necessarily belong to an object candidate. To avoid unintentional

[‖]A detailed description of all four algorithms including remarks on their COBOL-specific adaptations is given in [18].
[**]PROGRES transactions share some features with database transactions, e.g., atomicity.
[††]Due to the lack of space, the definitions of path expressions are omitted.

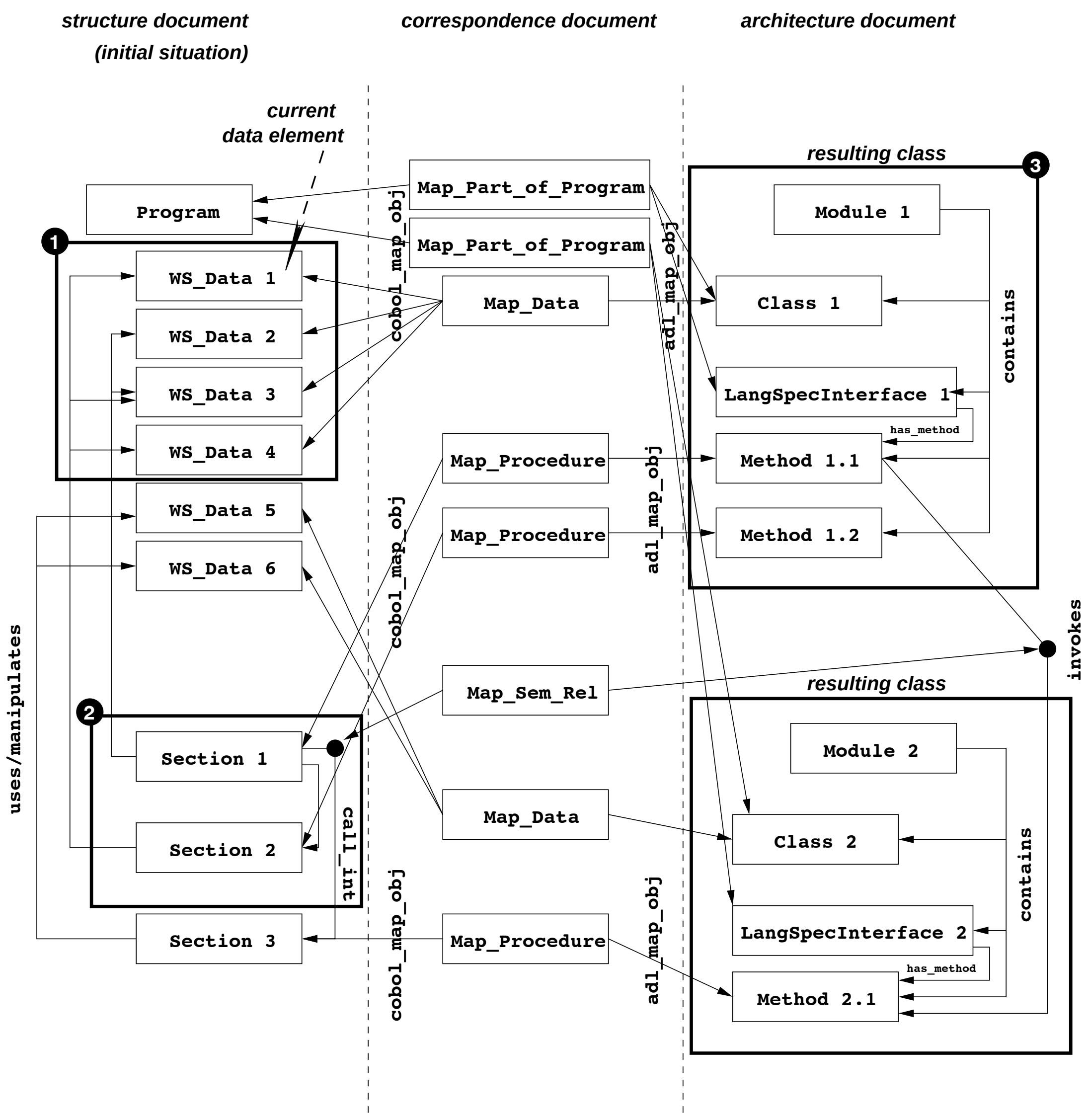


Figure 11. Example for the re-design algorithm according to Liu and Wilde.

```
transaction Produce_Next_Candidate ( prg_node : PROGRAM
                                     data_node : DATA;
                                     data_to_process : DATA [0:n];
                                     out data_processed : DATA [0:n])
=
use candidate_data : DATA [0:n] := empty;
    candidate_procs : PROCEDURE [0:n] := empty
do
  choose
    when (data_node in data_to_process)
    then
       candidate_data := data_node.=data_closure_path=>                      ❶
       & candidate_procs := data_node.=proc_closure_path=>                   ❷
       & Map_Candidate_Object(prg_node,candidate_procs,candidate_data)❸
       & data_processed := candidate_data
  else
    (* data_node is already part of an object candidate *)
    data_processed := empty
  end
end;
```

Figure 12. Re-design algorithm according to Liu and Wilde.

merging of different object candidates, the user may manually exclude data elements and procedures from processing by marking them as irrelevant.

Second, a general prerequisite for the application of this data-oriented algorithm is that all routines inside a source program are called explicitly and there is no (implicit) sequential control flow (known as *fall-through*). Otherwise, the algorithm will possibly decompose the source program into different classes without recognizing that there are interdependencies of routines mapped onto methods of different classes (see [18], p. 134 for a solution to this problem). Consequently, COBOL programs having an implicit sequential control flow can only be mapped onto an object-based architecture utilizing the complete encapsulation approach mentioned at the beginning of this section.

Third, the generic re-design algorithm has to be adapted to the respective programming language. In particular, the peculiarities of data declarations have to be taken into account. For example, in COBOL data elements are often organized in records. Nevertheless, every data element is still accessible without referencing the surrounding record. Therefore, in our implementation of the Liu and Wilde approach we redirect every access to a data element to the root data node that represents the surrounding record. As a result, the structuring of data into different records is not destroyed during re-design and records are transferred to a class completely.

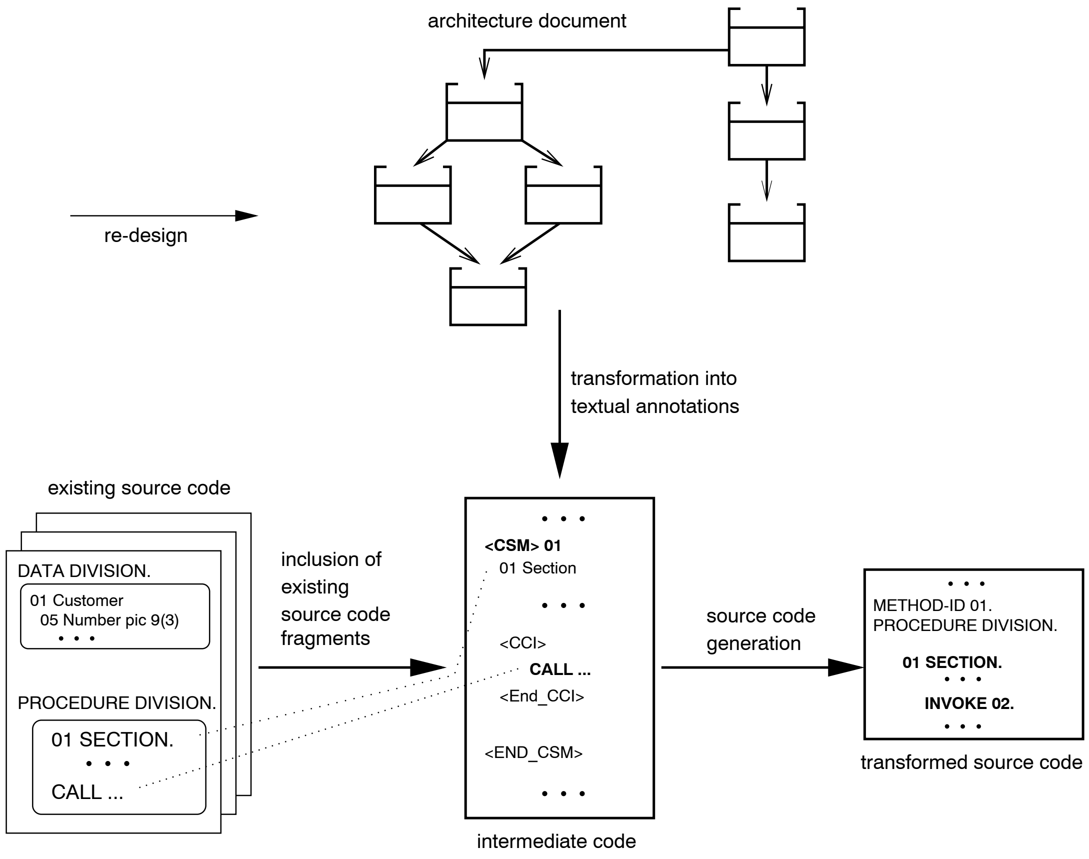


Figure 13. Source code transformation.

5. SOURCE CODE TRANSFORMATION

In the previous sections, the main focus was on the description of the re-design process and its tool support. The architecture is a very important aspect of a system, but also the adaptation of source code according to a changed description is a time consuming task. In this section, the *source code transformation tool* is described which supports the adaptation of the source code in order to achieve consistency with the architecture document.

In the top part of Figure 13, the architecture document is shown as a result of the re-design process. This document describes the entities and relationships of a system on a coarse-grained level. This description has to be transformed into corresponding source code fragments. Furthermore, there are existing source code parts (shown on the left-hand side of Figure 13) which have to be included into the transformed architectural entities. The source code transformation tool supports these tasks as follows.

First, the architectural entities and relationships are transformed into textual annotations, which are combined with existing source code fragments. To this end, we make use of the relationships between the architecture and the system structure graph being kept in the correspondence graph, and the attributes in the system structure graph pointing to source code fragments. In this way, we create an *intermediate code* consisting of textual annotations and already existing source code fragments. The intermediate code is the base for the last step (shown on the right-hand side in Figure 13) — the actual source code creation. In this step the textual annotations are replaced by concrete source code constructs.

The benefit of this two level process is the independence between the architecture and the programming language. The first step — creation of intermediate code — is performed at the architectural level. Generation of intermediate code merely involves the collection of existing source code fragments whose syntax and semantics is not considered. In contrast, the second step operates only on the source code level and is independent from the re-designed software architecture.

The transformation of architectural elements into textual annotations, which is realized in PROGRES, is based on a graph traversal of the architecture graph. Every visited graph element is transformed into a corresponding text fragment. For example, a `Class` node is represented by a `ConvertProgramToClass` annotation. This annotation expresses that a new class has to be introduced by means of an according source code fragment, and that the newly created class is to be ‘filled’ with already existing source code. The reused source code parts are likewise included into the intermediate code during the graph traversal.

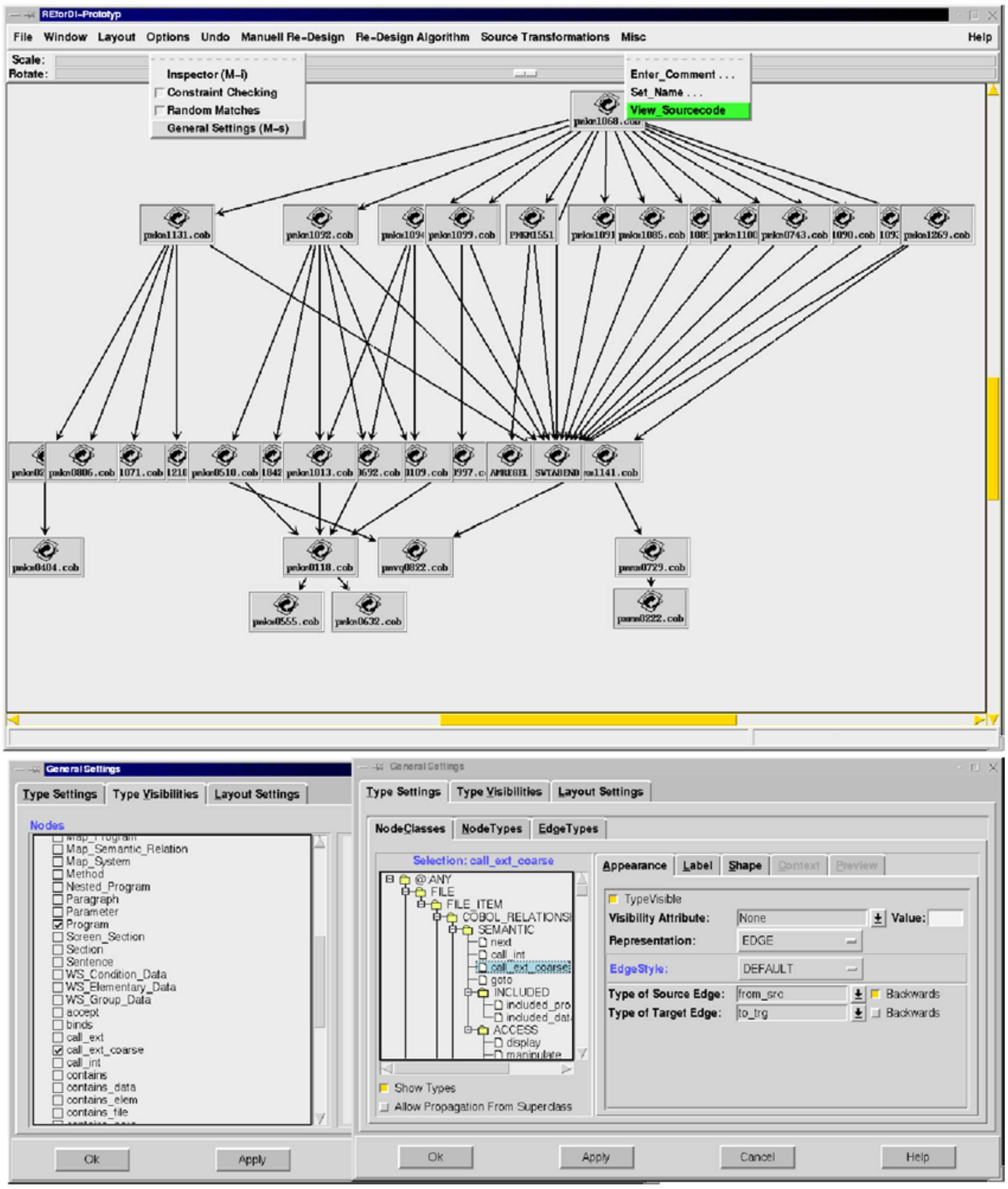
The creation of the intermediate code is closely coupled with the results of the re-design. The last step, namely the creation of concrete source code, is completely independent from the re-design. Again we use TXL to support this last step (for the use of TXL in code analysis, see Section 3.1). The syntax of the intermediate code is described by a TXL grammar, the transformation rules are responsible for replacing textual annotations with source code constructs.

The source code transformation tool may also handle new code that is written in the course of the re-engineering process. The re-engineering expert may selectively discard old code and replace it with a new implementation. Therefore, the source code transformation tool also includes code fragments attached to architectural elements into the result of the transformation process.

The transformation tool automates re-implementation to a great extent. However, the resulting code may have to be reworked for various reasons. In particular, code written anew may contain syntactic or semantic errors. Furthermore, it may happen that the code to be generated is not determined in a unique way. For example, if multiple procedures are merged into a single method, it is not clear in which order their bodies should be executed. In this case, the transformation tool would arrange the code fragments in some textual order and create comments for alerting the re-engineering expert of a potential problem that may have to be fixed manually.

6. CASE STUDY

In this section, we demonstrate the application of REforDI to a legacy system owned by one of the project partners. The application considered here is used for the administration of insurance agents. It maintains personal data, as well as data concerning the geographical region assigned to an agent, the



Nodes: ☑ Program, ☑ call_ext_coarse

☑ TypeVisible
Visibility Attribute: None Value:
Representation: EDGE
EdgeStyle: DEFAULT
Type of Source Edge: from_src ☑ Backwards
Type of Target Edge: to_trg ☐ Backwards

☑ Show Types
☐ Allow Propagation From Superclass

Figure 14. Visualization of the call graph.

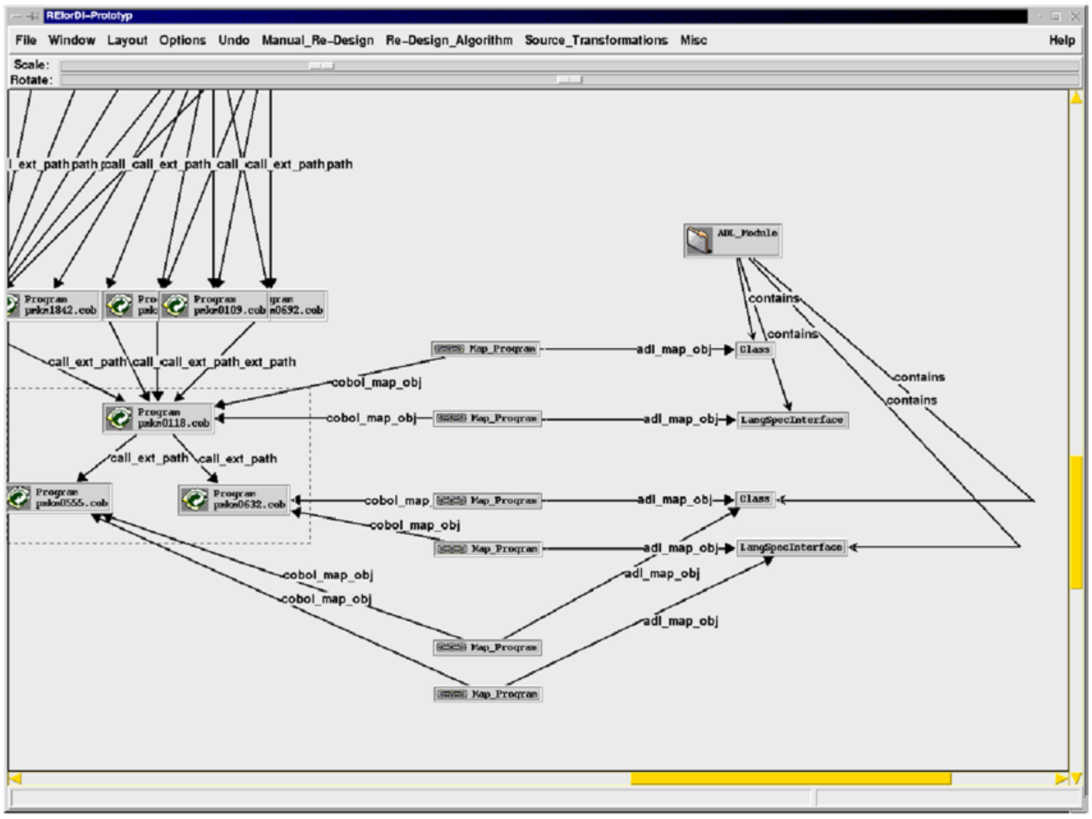


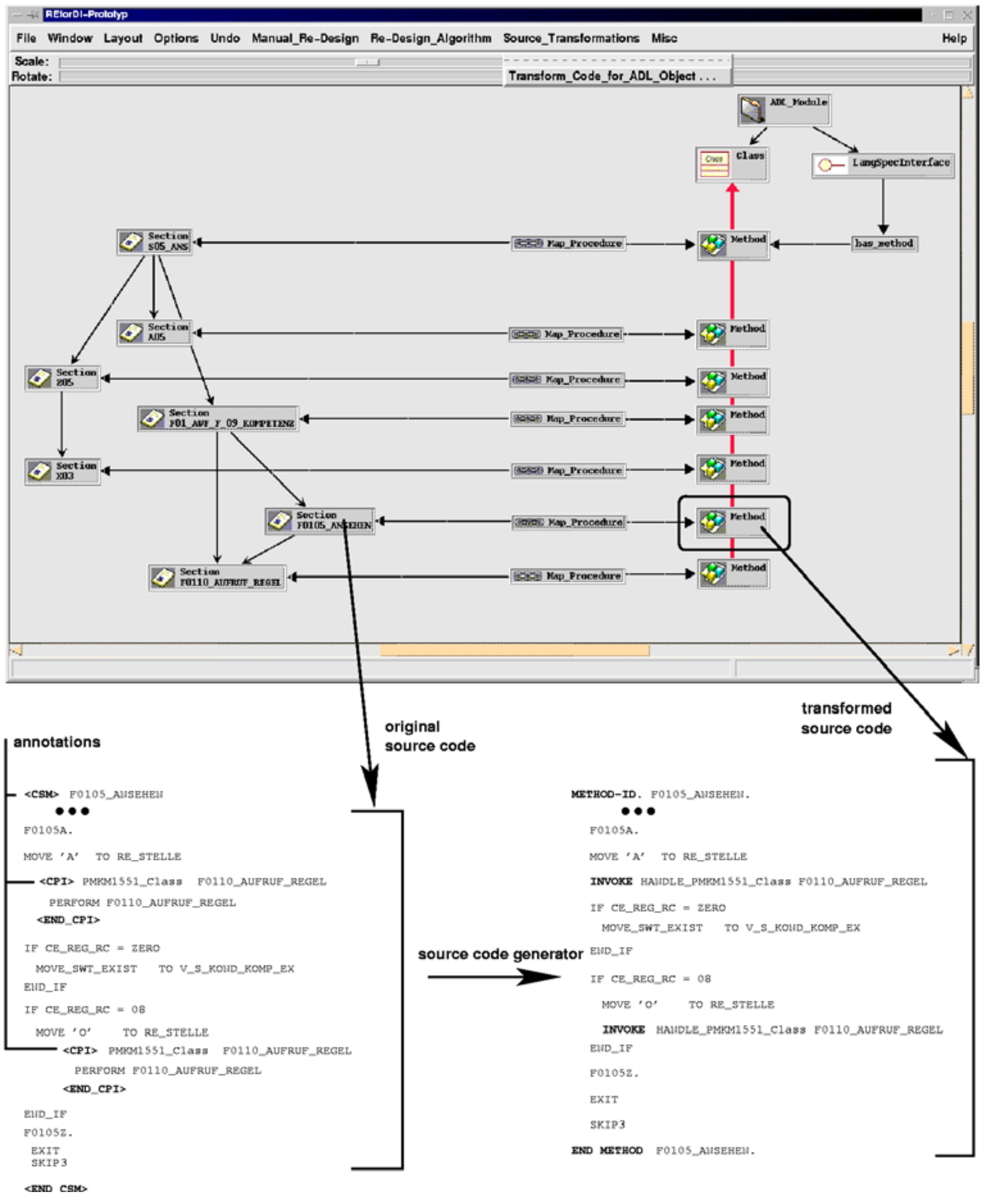
Figure 15. Manual re-design.

contracts for which (s)he is responsible, etc. The application is written in COBOL 85 and consists of 135 files which comprise about 45000 lines of code.

Unfortunately, we are not authorized to publish details of this case study because the material is confidential — a problem which occurs frequently in collaboration with industry and which must be lived with. Therefore, the following description is rather short.

The first step of the re-engineering process — code analysis — is performed automatically. The source code is parsed and transformed into the system structure graph, which may be displayed in multiple ways. Filters may be defined for node and edge types, layout algorithms may be selected and parameterized, etc. REforDI provides graphical tools which are based on the ECU framework [10]. ECU essentially provides a user interface to database code that is generated from the PROGRES specification. This user interface can be obtained with minimal effort by configuring graphical views.

For example, the call graph of the application is shown in Figure 14. Nodes and edges represent programs and external calls, respectively. A variant of the well-known Sugiyama algorithm is used to



```
<CSM> F0105_ANSEHEN
    •••
F0105A.

MOVE 'A'  TO RE_STELLE
  <CPI> PMKM1551_Class  F0110_AUFRUF_REGEL
    PERFORM F0110_AUFRUF_REGEL
  <END_CPI>

IF CE_REG_RC = ZERO
  MOVE_SWT_EXIST   TO V_S_KOND_KOMP_EX
END_IF
IF CE_REG_RC = 08
  MOVE 'O'    TO RE_STELLE
      <CPI> PMKM1551_Class  F0110_AUFRUF_REGEL
        PERFORM F0110_AUFRUF_REGEL
      <END_CPI>

END_IF
F0105Z.
 EXIT
 SKIP3

<END CSM>
```

```
METHOD-ID. F0105_ANSEHEN.
      •••
   F0105A.

   MOVE 'A'  TO RE_STELLE

   INVOKE HANDLE_PMKM1551_Class F0110_AUFRUF_REGEL

   IF CE_REG_RC = ZERO
     MOVE_SWT_EXIST   TO V_S_KOND_KOMP_EX
   END_IF

   IF CE_REG_RC = 08

     MOVE 'O'    TO RE_STELLE

     INVOKE HANDLE_PMKM1551_Class F0110_AUFRUF_REGEL
   END_IF

   F0105Z.

   EXIT

   SKIP3

END METHOD  F0105_ANSEHEN.
```

Figure 16. Section-to-method mapping: architecture and source code transformation.

generate the layout of the call graph. If the generated layout is not satisfactory, the user can still adjust it manually.

The windows shown at the bottom serve to configure the graphical view. The list on the left defines the visibilities of nodes based on their types. In the case of the call graph, only nodes of the types `Program` and `call_ext_coarse` are shown. The latter node type is used to represent program calls internally. The window on the right shows an edge-node-edge filter which defines the mapping of an edge-node-edge triple to an arrow appearing on the screen. That is, what the user perceives as an edge is actually represented as a node in the internal system structure graph.

As the next step of the re-engineering process, the application is re-designed. Re-design may be carried out either manually, or it may be automated with the help of re-design algorithms. Figure 15 shows an example of manual re-design. A human expert identified those parts of the application which access the underlying database (see the dashed box on the left). Then, he decided to group these parts into a module of the software architecture. This module, which is displayed on the right, contains two classes with respective interfaces. Please note that the expert did not map the programs 1:1 onto classes. Rather, two of them were merged into a single class. This illustrates the degrees of freedom which is offered to the re-designer in the case of manual re-design.

In contrast, the upper part of Figure 16 shows the result of applying one of the re-design algorithms mentioned at the beginning of Section 4.4. Here, a program is mapped onto a class, and each of its sections is mapped onto a method. The lower part of this figure serves to illustrate the final step of the re-design process, namely source code transformation. First, the source code is transformed into intermediate code consisting of annotations which are fleshed out with source code fragments written in COBOL 85. After that, the intermediate code is transformed into the target programming language (Object COBOL). For example, `PERFORM` statements are converted into `INVOKE` statements.

7. RELATED WORK

Below, we compare REforDI to related approaches both on a conceptual level (Section 7.1) and on a tool level (Section 7.2).

7.1. Concepts

REforDI addresses the re-engineering of large legacy applications written in COBOL. More specifically, we are dealing with the migration of mainframe applications to a client/server architecture. This task is supported by offering tools for code analysis, re-design, and source code transformation, the latter of which is used to transform the application from COBOL 85 to Object COBOL. In this way, the application is prepared for distribution.

As we have already pointed out in Section 2.1, the work described in this paper does not cover the final step in the migration process, namely the decomposition of the application into client and server components. This problem was studied in a companion PhD thesis [4, 5]. Based on the architecture graph, cut lines are identified by annotating architectural elements. Subsequently, the architecture is transformed by instantiating design patterns for distribution. Finally, corresponding transformations are performed on the source code level, resulting in an application making use of CORBA. This involves the generation of wrappers, which is also addressed, e.g., in [21].

In the re-engineering process, we consider it crucial to perform abstraction such that program understanding is raised to the architectural level. Thus, it is not sufficient to operate on the source code level alone, as it is done, e.g., in [22]. Source-to-source transformations are useful for some purpose, e.g., for getting rid of a legacy programming language which is no longer supported. However, they merely recode the application in another programming language. Usually, the resulting code is at least as difficult to understand as the original code [23]. For client/server migration, it is necessary to understand at least

- how the application is structured on a coarse-grained level,
- which parts are to be moved to the client and to the server, respectively,
- how the server code — which usually is not restructured — is to be wrapped, and
- which functions are performed by the client code that needs to be replaced.

Many design recovery or re-design algorithms have been proposed in the literature and have partially been implemented in tools for re-engineering [24, 25, 26, 27, 28, 15]. As observed in [11], there is no ‘best’ algorithm. Rather, all of them are heuristics which perform reasonably well under some assumptions and fail when these assumptions are not met. In REforDI, we have implemented (and adapted) a set of algorithms known from the literature. The human expert may select the algorithm considered most suitable for the problem at hand. Moreover, (s)he may control where to apply the selected algorithm. As a consequence, different algorithms may be applied at different locations. For example, the human expert may apply simple 1:1 transformations (e.g., mapping of a program to a class exporting a single method) to those parts of the application which are not going to be restructured. For those parts to be restructured, we offer algorithms working at a rather coarse-grained level (procedures rather than statements). Otherwise, restructuring of large applications may prove too time-consuming.

7.2. Tools

We consider the use of formal specifications and generators an essential contribution of our work. As stated in Section 2.3, implementation of REforDI required only a minimal amount of hand-coding. For operating at the source code level, REforDI makes use of parsers and tree transformers generated from TXL specifications. In contrast, graphs are employed at the architectural level. Operations on the system structure graph and the architecture graph are specified with the help of graph transformation rules at a high level of abstraction. More specifically, we are using triple graph grammars for specifying the re-design process. Below, we compare REforDI to a set of other re-engineering tools which have made important contributions to research carried out in this field.

REDO [29, 30, 31] is a research project which was funded by the ESPRIT II programme and was carried out by a European consortium of industrial and academic partners 1989–1992. REDO addresses the re-engineering of legacy systems written in COBOL and FORTRAN. The reverse engineering process is divided into three steps. In the first step (Clean), the application is translated into UNIFORM, a neutral intermediate language with a reduced set of language constructs to simplify further program analysis. In the second step (Specify), objects and functions are identified in the UNIFORM code (design recovery). In the third step (Simplify), the resulting program specifications are simplified and rearranged until they become more recognizable in terms of known concepts. As a net result, executable specifications are created which are written in Z++, an object-oriented extension of Z.

Subsequent modification of the reverse engineered application is considered a forward engineering process, where changes are performed at the specification level and are transformed automatically into changes at the code level. REforDI follows a less ambitious approach, which strives for a structural architectural description rather than for an executable specification. In addition, we do not assume that maintenance can be performed at the level of executable specifications, having code generated silently in the background.

Software Refinery [32, 33] is a commercial environment for building re-engineering tools developed by Reasoning Systems. Software Refinery provides a generic infrastructure for re-engineering tools consisting of a parser generator, an object-oriented database system which is used to store programs as abstract syntax trees, a GUI toolkit for constructing graphical user interfaces, and a workbench providing reusable components for re-engineering tools (e.g., components dealing with call graphs, structure charts, etc.). With the help of Software Refinery, language-specific re-engineering tools have been built, including, e.g., REFINE/COBOL for COBOL programs. These tools may still be customized through an application programming interface (API). Restructuring of programs is specified in a high-level language which is also called REFINE and is based on tree transformation rules. In contrast to REforDI, transformation rules are applied immediately at the source code level (source-to-source transformations, see also Section 7.1). Moreover, REforDI differs from REFINE inasmuch as it relies on graph rather than tree transformation rules.

Rigi [34, 35, 36] is an interactive toolkit with a graphical workbench which can be used for re-engineering purposes. Rigi regards software systems as an accumulation of artifacts. On the one hand, artifacts are software components like subsystems, procedures or interfaces. On the other hand, relationships like control flow and data flow are considered artifacts too. Rigi aims at identifying artifacts in source code and representing the results to the user in an appropriate way. The procedure is very similar to REforDI: in a first phase, components are identified, filed in a graph structure that is stored in a database, and then presented to the user. In a second phase, abstractions are derived in an automatic or semi-automatic manner that are more easy to manage. While the first phase is dependent on the underlying programming language of the source system, language-independent algorithms are used in the second phase. In Rigi, more effort has been put into graph visualization than in REforDI. On the other hand, Rigi is not based on a high-level specification language. Rather, graphs are accessed through a procedural interface, which makes coding of graph algorithms more painstaking than in REforDI.

The *Bauhaus* project [17] is a research collaboration between the Institute for Computer Science of the University of Stuttgart (Germany) and the Fraunhofer Institute of Experimental Software Engineering in Kaiserslautern (Germany). The aim of this project is, as stated in [37], to find methods and techniques for architecture recovery, and to explore languages to describe recovered architectures. The project focuses on automatic and semi-automatic component recovery from source code. Diverse views on the software architecture are built using Rigi (see above). Like Rigi, Bauhaus is not based on formal specifications. Furthermore, it differs from REforDI inasmuch as it addresses mainly programs written in C and is restricted to component recovery (i.e. no re-design and source code transformation).

The *GUPRO* project [38, 39, 40], which has been carried out at the University of Koblenz (Germany), addresses the development of a generic environment for program understanding. GUPRO is based on the meta CASE tool KOGGE [41]. Internally, programs are represented as graphs. GUPRO offers parsers for several languages, including COBOL and C. Like in Rigi, the first phase of processing depends on the programming language, while the second phase (analysis of the graph built up by the parser) is language independent. Different kinds of analyses may be specified with the help of a graph query language. In contrast to REforDI, GUPRO focuses only on design recovery, i.e. the application under study is not modified. Moreover, GUPRO offers a textual user interface, while REforDI provides graphical tools.

DB-MAIN, developed at the University of Namur (Belgium), denotes both a meta CASE tool [42] and tools for data reverse engineering [43] which have been built on top of it. The meta CASE tool is based on an ER-like data model and provides a procedural scripting language for writing specific tools. The reverse engineering tools address the recovery of a conceptual schema of a database (a problem which we have not attacked in REforDI). Recovery is performed by a transformation process which is fed with a physical schema and repeatedly replaces concrete elements with more abstract ones. In addition to the schema, other sources of information such as programs, populated databases, etc. are exploited. The transformation process is not automatic; rather, it is driven by user decisions. In contrast to REforDI, DB-MAIN is based on evolutionary transformations. That is, the source of the transformation is modified step by step, while a separate representation is built up from scratch in REforDI.

VARLET [44, 45, 46], which originates from the University of Paderborn (Germany), addresses a similar problem as DB-MAIN. In VARLET, a relational schema is transformed into an object-oriented one. Like REforDI, VARLET is based on triple graph grammars. Moreover, VARLET tools have been specified and implemented with the help of PROGRES. REforDI differs from VARLET inasmuch as it addresses a different problem (client/server migration of legacy systems written in COBOL rather than data reverse engineering). Moreover, VARLET decomposes the transformation process into two steps. In the first step, an initial transformation is performed automatically. In the second step, the initial object-oriented schema is improved through the application of transformation rules which operate only on the object-oriented schema (but retain and update the links to the relational schema). In contrast, there is no such decomposition in REforDI. The intelligence of the transformation resides in the forward rules whose application may be controlled interactively.

8. LESSONS LEARNED

Re-engineering methodology. The re-engineering methodology we use puts a strong emphasis on software architecture. This idea turned out to be essential for the identification of meaningful objects and subsystems, either by making manual decisions or by ‘correcting’ the results produced by an algorithm. In addition, the recovered and documented software architecture became the main basis of discussions between domain experts, reengineers, and tool developers.

The case study showed that a re-design regarding complex legacy systems should take place in several phases. We propose the following procedure.

- Execute a coarse grained re-design in the first phase by decomposing the system into several major subsystems. Every identified subsystem is then wrapped and provided with an explicit interface (similar to algorithms 1 and 2).
- Now subdivide all subsystems into meaningful modules each comprising several classes. Repeat this step by further subdividing the modules if necessary to get modules of reasonable size.

- In a second phase, apply more fine-grained re-design algorithms to modules or smaller subsystems and correct their results interactively if necessary.

This approach reduces the complexity of the analyzed systems before re-design starts on a more fine-grained level. Due to studies that code transformation is a very difficult task [23], more and more re-engineering projects decide to use wrapping approaches and to leave the original source code untouched as far as possible [47]. That is, they stop at the first or second item in the list above at least in some parts of the application.

Functionality. The case study showed that neither pure interactive nor pure algorithm driven re-engineering of complex legacy systems is feasible. If the legacy system is too large, interactive, human-centered re-engineering is too time-consuming and fault-prone to be efficient. Algorithms on the other hand do not perform well in any case. The quality of their output is very much dependent on whether the appropriate re-design algorithm has been applied to a part of a legacy system. Even if an algorithm performs well on one part of a system it might deliver bad results on another part of the same system. Anyhow, in most cases there will be the need for interactive adjustment of an algorithms' scope or interactive corrections of its results, respectively. Therefore, we recommend to have both an interaction facility and a number of different re-design algorithms in order to obtain the best results possible.

User interface. Very soon we recognized that due to the limited perceptivity of human beings it is very important to be able to reduce and extend the amount of visualized information on demand. Though possible in general, the REforDI environment suffered from the fact, that the overall flexibility of graphical views was not sufficient in any case. Furthermore, in addition to graph views it is desirable to have different kinds of visualizations of the collected information like tree, list, or textual views.

Realization. In general, the realization approach was successfully put into practice in the REforDI project. But, we identified some aspects that could be improved. Firstly, the parsing process using TXL (see Section 3) is awkward. Classical parsing techniques would aim at the desired target more directly. Secondly, the use of TXL for code transformation purposes leads to a loss of modularity in the transformation process. As a result from incremental code generation, the underlying grammar mixes up source and target language. An approach that maintains modularity and separation of concerns would be desirable in this case.

In our opinion, the extensive use of reusable frameworks, formal specifications, and generators are a key concept to efficiently building flexible and powerful re-engineering tools. PROGRES is an expressive programming language that provides specifications on a highly abstract level. In combination with code generation for tool implementation purposes, PROGRES allows to rapidly build and extend tool prototypes. But, the structuring elements for programming in the large and layering in PROGRES specifications are not satisfyingly solved, yet. When the tools were developed, there were no means for large-scale structuring which results in confusingly complex specifications. Today, there are some means for structuring which have not been tested extensively yet [48]. Therefore, the number of reusable parts in the specification is limited to a certain extent. Furthermore, the concept of graph grammars and the PROGRES language might be a bit difficult to learn for newcomers.

Dynamic behavior. Finally, the dynamic aspects of a system (runtime behavior etc.) are not considered in the REforDI project. But, we found that in order to understand a system this information is as important as information on the static parts of a system (structure, (possible) control and data flow). At least the knowledge on a systems' dynamics is needed to be able to perform meaningful re-design by identifying meaningful modules. The importance of dynamic information is also recognized in modern modeling languages like UML [49] and Room [50]. In these languages you have several possibilities to model behavior. Therefore, in a modern re-engineering tool the dynamic information should be addressed as well.

The results from the REforDI project heavily influenced our current re-engineering project, the E-CARES research project [51]. In this project we address, among others, multi-level abstraction and visualization, flexible views and dynamic information (traces, state machines). Furthermore, we experiment with layered specifications, modularity, and multi-language support.

9. CONCLUSION

We have presented REforDI, an environment for reengineering COBOL programs. REforDI provides integrated code analysis, re-design, and re-implementation. At the user interface, it offers graphical views on code abstractions and the software architecture. Internally, REforDI makes heavy use of formal specifications and generators to reduce the implementation effort.

In particular, the experiences gained in applying graph technology to reengineering have been quite positive. This motivated us to explore this technology further in another reengineering project addressing an application domain which has been considered only rarely in the reengineering literature: *embedded systems*. So far, research in reengineering has focused mainly on sequential, un-timed systems, in particular in the area of business applications. The approaches found there aim, e.g., at decomposing monolithic systems (REforDI, described in this paper), decoupling user interface/presentation, application logic, and data handling/database management (RECAST [28]), or at identifying reusable components (COBOL/SRE [15]).

In contrast, the *E-CARES* (*E*ricsson *C*ommunication *AR*chitecture for *E*mbedded *S*ystems) research cooperation [51] between Ericsson Eurolab Deutschland GmbH (EED) and the Department of Computer Science III, RWTH Aachen, aims at improving the reengineering of complex legacy telecommunication systems by developing methods, concepts, and tools to support the processes of understanding and restructuring of this special class of embedded systems. The work in this project is based on the ideas and results of the REforDI approach, at the same time refining and improving the approach, extending it by adding new functionality, and applying it to a different domain – telecommunication systems. Here, it is crucial to support the understanding of dynamic behavior, as we have already argued above.

Acknowledgements

The work described in this paper was partially supported by the Northrhine-Westfalian Ministry of Education and Research (BMBF/MWF) under the project *Domain-Specific Software Engineering in Education*, which was carried out in cooperation with two German companies: Aachener und Münchener Informatik-Service AG (AMI) and Gebühreneinzugszentrale (GEZ).

REFERENCES

1. Cremer K. A tool supporting the re-design of legacy applications. Nesi P, Lehner F (eds.): *Proceedings 2nd Euromicro Conference on Software Maintenance and Reengineering CSMR 1998*. IEEE Computer Society Press: Los Alamitos CA, 1998; 142–148.
2. Cremer K. Graph-based reverse engineering and reengineering tools. Nagl M, Schürr A, Münch M (eds.): *Proceedings International Workshop on Applications of Graph Transformations with Industrial Relevance AGTIVE 1999*, LNCS 1779. Springer: Heidelberg, Germany, 1999; 95–109.
3. Cremer K. *Graphbasierte Werkzeuge Werkzeuge zum Reverse Engineering und Reengineering*. Deutscher Universitätsverlag: Wiesbaden, Germany, 1999; 220 pp.
4. Radermacher A. Support for design patterns through graph transformation rules. Nagl M, Schürr A, Münch M (eds.): *Proceedings International Workshop on Applications of Graph Transformations with Industrial Relevance AGTIVE 1999*, LNCS 1779. Springer: Heidelberg, Germany, 1999; 111–126.
5. Radermacher A. *Tool Support for the Distribution of Object-Based Applications*. PhD thesis, Aachen University of Technology: Aachen, Germany, 2000; 191 pp.
6. Chikofsky EJ, Cross II JH. Reverse engineering and design recovery: A taxonomy. *IEEE Software* 1990; **7**(1):13–17.
7. Cordy JR, Halpern-Hamu CD, Promislow E. TXL: A rapid prototyping system for programming language dialects. *Computer Languages* 1991; **16**(1):97–107.
8. Schürr A, Winter AJ, Zündorf A. Graph grammar engineering with PROGRES. Schäfer W, Botella P (eds.): *Proceedings 5th European Software Engineering Conference ESEC 1995*, LNCS 989. Springer: Heidelberg, Germany, 1995; 219–234.
9. Schürr A, Winter AJ, Zündorf A. The PROGRES approach: Language and environment. Ehrig H, Engels G, Kreowski HJ, Rozenberg G (eds.): *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, volume 2. World Scientific: Singapore, 1999; 487–550.
10. Jäger D. Generating tools from graph-based specifications. *Information and Software Technology* 2000; **42**(2):129–139.
11. Canfora G, Cimitile A, De Lucia A, Di Lucca G. Decomposing legacy systems into objects: An eclectic approach. *Information and Software Technology* 2001; **43**(6):401–412.
12. Schürr A. Specification of graph translators with triple graph grammars. Mayr EW, Schmidt G, Tinhofer G (eds.): *Proceedings 20th Workshop on Graph-Theoretic Concepts in Computer Science WG 1994*, LNCS 903. Springer: Heidelberg, Germany, 1994; 151–163.
13. Canfora G, Fasolino AR, Tortorella M. Towards reengineering in reuse reengineering processes. *Proceedings International Conference on Software Maintenance ICSM 1995*. IEEE Computer Society Press: Los Alamitos CA, 1995; 147–156.
14. Liu SS, Wilde N. Identifying objects in a conventional procedural language: An example of data design recovery. *Proceedings Conference on Software Maintenance CSM 1990*. IEEE Computer Society Press: Los Alamitos CA, 1990; 266–271.
15. Ning JQ, Engberts A, Kozaczynski W. Recovering reusable components from legacy systems by program segmentation. *Proceedings Working Conference on Reverse Engineering WCRE 1993*. IEEE Computer Society Press: Los Alamitos CA, 1993; 64–72.
16. Cimitile G, Visaggio G. Software salvaging and the call dominance tree. *The Journal of Systems and Software* 1995; **28**(2):117–127.
17. Girard JF, Koschke R. Finding components in a hierarchy of modules: A step towards architectural understanding. *Proceedings International Conference on Software Maintenance ICSM 1997*. IEEE Computer Society Press: Los Alamitos CA, 1997; 58–65.
18. Marburger A. Reengineering von prozeduralen zu objektbasierten Software-Systemen. Master's thesis, Aachen University of Technology: Aachen, Germany, 1999; 404 pp.
19. Canfora G, Cimitile A, Munro M. An improved algorithm for identifying objects in code. *Software - Practice and Experience* 1996; **26**(1):25–48.
20. Dunn MF, Knight JC. Automating the detection of reusable parts in existing software. *Proceedings 15th International Conference on Software Engineering ICSE 1993*. IEEE Computer Society Press: Los Alamitos CA, 1993; 381–390.
21. Sneed HM. Generation of stateless components from procedural programs for reuse in a distributed system. Ebert J, Verhoef C (eds.): *Proceedings 4th European Conference on Software Maintenance and Reengineering CSMR 2000*. IEEE Computer Society Press: Los Alamitos CA, 2000; 183–188.
22. do Prado Leite JCS, Sant'Anna M, do Prado AF. Porting COBOL programs using a transformational approach. *Journal of Software Maintenance: Research and Practice* 1997; **9**(1):3–31.
23. Terekov AA, Verhoef C. The realities of language conversion. *IEEE Software* 2000; **17**(6):111–124.
24. Canfora G, Cimitile A, De Lucia A, Di Lucca G. Decomposing legacy programs: a first step towards migrating to client-server platforms. *The Journal of Systems and Software* 2000; **54**(2):99–110.
25. Sellink A, Sneed H, Verhoef C. Restructuring of COBOL/CICS legacy systems. Nesi P, Verhoef C (eds.): *Proceedings 3rd European Conference on Software Maintenance and Reengineering CSMR 1999*. IEEE Computer Society Press: Los

Alamitos CA, 1999; 72–82.

26. Taschwer M, Rauner-Reithmayer D, Mittermeir R. Generating objects from C code – features of the CORET tool-set. Nesi P, Verhoef C (eds.): *Proceedings 3rd European Conference on Software Maintenance and Reengineering CSMR 1999*. IEEE Computer Society Press: Los Alamitos CA, 1999; 91–100.
27. Brito e Abreu F, Pereira G, Sousa P. A coupling-guided cluster analysis approach to reengineer the modularity of object-oriented systems. Ebert J, Verhoef C (eds.): *Proceedings 4th European Conference on Software Maintenance and Reengineering CSMR 2000*. IEEE Computer Society Press: Los Alamitos CA, 2000; 13–22.
28. Burd E, Munro M, Wezeman C. Analysing large COBOL programs: the extraction of reusable modules. *Proceedings International Conference on Software Maintenance ICSM 1996*. IEEE Computer Society Press: Los Alamitos CA, 1996; 238–243.
29. van Zuylen HJ (ed.). *The REDO Compendium: Reverse Engineering for Software Maintenance*. John Wiley & Sons: Chichester, UK, 1993; 405 pp.
30. Breuer P, Lano K. Creating specifications from code: Reverse-engineering techniques. *Journal of Software Maintenance: Research and Practice* 1991; **3**(3):145–162.
31. Bowen J, Breuer P, Lano K. Formal specifications in software maintenance: From code to Z++ and back again. *Information and Software Technology* 1993; **35**(11/12):679–690.
32. Markosian L, Newcomb P, Brand R, Burson S, Kitzmiller T. Using an enabling technology to reengineer legacy systems. *Communications of the ACM* 1994; **37**(5):58–70.
33. Newcomb P, Markosian L. Automating the modularization of large COBOL programs: Application of an enabling technology for reengineering. *Proceedings Working Conference on Reverse Engineering WCRE 1993*. IEEE Computer Society Press: Los Alamitos CA, 1993; 222–230.
34. Tilley SR, Wong K, Storey MAD, Müller HA. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering* 1994; **4**(4):501–520.
35. Müller HA, Orgun MA, Tilley SR, Uhl JS. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice* 1993; **5**(4):181–204.
36. Storey MAD, Müller HA, Wong K. Manipulating and documenting software structures. Eades P, Zhang K (eds.): *Software Visualisation*, volume 7 of *Book Series on Software Engineering and Knowledge Engineering*. World Scientific: Singapore, 1996; 244–263.
37. Koschke R. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Institute of Computer Science, University of Stuttgart: Stuttgart, Germany, 2000; 414 pp.
38. Kullbach B, Winter A, Dahm P, Ebert J. Program comprehension in multi-language systems. *Proceedings 5th Working Conference on Reverse Engineering WCRE 1998*. IEEE Computer Society Press: Los Alamitos CA, 1998; 135–143.
39. Kamp M. Managing a multi-file, multi-language software repository for program comprehension tools - a generic approach. *Proceedings 6th International Workshop on Program Comprehension*. IEEE Computer Society Press: Los Alamitos CA, 1998; 64–71.
40. Kullbach B, Winter A. Querying as an enabling technology in software reengineering. Nesi P, Verhoef C (eds.): *Proceedings 3rd European Conference on Software Maintenance and Reengineering CSMR 1999*. IEEE Computer Society Press: Los Alamitos CA, 1999; 42–50.
41. Ebert J, Süttenbach R, Uhe I. Meta-CASE in practice: A case for KOGGE. Olive A, Pastor JA (eds.): *Proceedings 9th International Conference on Advanced Information Systems Engineering CAiSE 1997*, LNCS 1250. Springer: Heidelberg, Germany, 1997; 203–216.
42. Englebert V, Hainaut JL. DB-MAIN: A next generation meta-CASE. *Information Systems* 1999; **24**(2):99–112.
43. Hainaut JL, Englebert V, Henrard J, Hick JM, Roland D. Database reverse engineering: From requirements to CARE tools. *Automated Software Engineering* 1996; **3**(1):9–45.
44. Jahnke J, Zündorf A. Applying graph transformations to database re-engineering. Ehrig H, Engels G, Kreowski HJ, Rozenberg G (eds.): *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, volume 2. World Scientific: Singapore, 1999; 267–286.
45. Jahnke JH. *Managing Uncertainty and Inconsistency in Database Reengineering Processes*. PhD thesis, University of Paderborn, Department of Mathematics and Computer Science: Paderborn, Germany, 1999; 256 pp.
46. Jahnke J, Wadsack J. The Varlet analyst: Employing imperfect knowledge in database reverse engineering tools. *Proceedings 3rd International Workshop on Intelligent Software Engineering*. ICSE 2000 Workshop 14, 2000; 59–69.
47. Sneed HM. Migration prozeduraler Anwendungssysteme in eine objektorientierte Architektur. *Softwaretechnik-Trends* 2001; **21**(2):13–18.
48. Schürr A, Winter AJ. UML Packages for PROgrammed Graph REwriting Systems. Ehrig H, Engels G, Kreowski HJ, Rozenberg G (eds.): *Proceedings 6th International Workshop on Theory and Application of Graph Transformations TAGT 1998*, LNCS 1764. Springer: Heidelberg, Germany, 1998; 396–409.
49. Booch G, Rumbaugh J, Jacobson I. *The Unified Modeling Language User Guide*. Addison Wesley: Reading, MA, 1999; 482 pp.

50. Selic B, Gullekson G, Ward PT. *Real-Time Object-Oriented Modeling*. John Wiley & Sons: Reading, MA, 1994; 525 pp.
51. Marburger A, Herzberg D. E-CARES research project: Understanding complex legacy telecommunication systems. Sousa P, Ebert J (eds.): *Proceedings 5th European Conference on Software Maintenance and Reengineering CSMR 2001*. IEEE Computer Society Press: Los Alamitos CA, 2001; 139–147.

AUTHOR'S BIOGRAPHIES

Katja Cremer received her diploma degree in computer science in 1995 and her doctoral degree in 1999, both from Aachen University of Technology. She continued as a reseacher in the Collaborative Research Center IMPROVE. Then she joined Ericsson Eurolab Germany. Since 2000 she has been employed as a system designer for test methods and tools. Her interests are software engineering, reverse and re-engineering tools as well as software architectures for testing systems. In 1999 she received the German Software Engineering Award for the results of her doctoral thesis.

André Marburger received his diploma degree in Computer Science in 1999 from Aachen University of Technology, where he has been working as a research assistant at the Department of Computer Science III since then. Additionally, he is a candidate for doctoral degree (PhD) at this university. He has been the main designer and implementor of the re-design and the source code transformation of the REforDI prototype. Currently, his research focuses on the E-CARES research project. He is interested in software engineering environments, object-oriented modeling, embedded systems, prototyping frameworks, re-engineering and software architectures.

Bernhard Westfechtel received his diploma degree in 1983 from University of Erlangen-Nuernberg and his doctoral degree (PhD) in 1991 from Aachen University of Technology, where he has been working as a senior researcher since then. He is interested in software engineering environments, software configuration management, process modeling, workflow management, object-oriented modeling, engineering/product data management, database systems for engineering applications, re-engineering, and software architectures. In 1999, he received the habilitation degree from Aachen University of Technology and published a book on models and tools for managing development processes.