On project-specific languages and their application in reengineering

Dmitry Boulychev, Dmitry Koznov, Andrey A. Terekhov
St. Petersburg State University, LANIT-TERCOM
198504, Russia, St. Petersburg, Bibliotechnaya sq., 2

Abstract

We propose an approach for tuning reengineering tools to particular projects. This approach is based on the informal knowledge of the system, consisting of specific usages of the programming language. We illustrate this process with examples from an industrial project on PL/I to Java conversion.

1 Introduction

This article is based on the experience accumulated by the authors during implementation and application of an automated reengineering tool RescueWare. Development of this tool has been going on for the last few years [1]. The idea of RescueWare is to provide the complete functionality for reengineering of legacy systems:

- Framework for understanding legacy systems
- Knowledge mining including program slicing, creating data dictionaries for the system, redocumentation etc.
- Generation of programs in modern languages from the legacy code

Initially RescueWare Workbench was a simple language converter, but during deployment it has become clear that compiler technologies alone were insufficient, because any quality reengineering process requires an active human participation. Therefore, some legacy understanding and knowledge mining tools were gradually added to the initial set of tools. Currently RescueWare is a fully-fledged interactive reengineering technology.

There followed industrial legacy reengineering projects, which confirmed the validity of our assumptions. However, in most projects we encountered difficulties, because our tools were unable to extract the sufficient amount of information from legacy sources.

Human developmers always extract more subtle background information on the legacy system, than the automated data mining processes. For example, semantics of variables, naming conventions, comments, structure of source files and other similar data which change from project to project. This informal knowledge does not affect the execution of programs, so it goes unobserved during compilation or analysis with traditional methods. But in reengineering this kind of knowledge is often more useful than the exact semantics of language constructions. This is why the standard compiler approaches may be inappropriate for reengineering.

In this paper we consider one type of this informal knowledge represented by recurring usages of programming language constructs in a project. We argue that most large projects contain repetitive usages of some patterns, and these patterns can be defined, understood (i.e., attributed with their meaning) and used as single language constructs in a new project-specific language.

To illustrate one possible application of this approach, let us consider the following scenario which often happens in language conversion projects. Suppose we examine a legacy system written in an old programming language, planned for conversion into a more modern language. Such systems are almost always heavily influenced by the limitations of the language in which they were written. The legacy language has perhaps a restricted pool of available expressions, so that much code is spent on the features not directly supported by the language. As a result, the system contains plenty of such auxiliary code. It can be concentrated in a single kernel-like module or scattered throughout the system.

When it comes to conversion, it is likely that the target language has in-built features eliminating the need for most auxiliary code. For instance, Java or Visual Basic both have a richer set of string handling functions than C does, so a considerable simplification of code must be expected in conversions from C to either of these languages. To achieve this goal, the converter must analyse the code containing `char*` variables and look for patterns that can be replaced with the corresponding String methods in the target languages.

Note that we are always looking for patterns performing functions absent in the source language (but present in the target one). Referring to the terminology introduced in [2], here we are dealing with conversion of constructs that were emulated in the legacy language to the native constructs in the target language. Abstractly speaking, we can consider these emulated constructs of the legacy language as native constructs of some higher-level virtual language. It is clear that this virtual language reflects the inner workings of the legacy system much better than the original one and that it can be used in reengineering process.

In this paper we describe an effort to define this virtual *project-specific language* in the reengineering project conducted by our team. The goal of the project was to convert a legacy application of approximately 70,000 lines of code from PL/I to Java. An approach stated in this article was developed to enhance the quality of the generated code. In the conclusion, we describe our vision of the "ideal" process of reengineering, where the informal knowledge of the engineers is taken into account during further transformations of the legacy system.

1.1 Related Works

It is often desirable to write the transformation rules in a formal notation for use in other projects. There exist a few equivalently suitable notations — program plans [3], transformation rules [4] and clichés [5]. Each of these methods employs some kind of pattern-matching for application of the stored rules to a piece of code [6].

The patterns are usually stored in an intermediate language. An alternative to intermediate languages, *native patterns*, were proposed in [7] as a formalism for recording patterns in terms of the source language. The authors reason that it is more convenient and clear notation than any specially designed intermediate representation. In our opinion this approach is somewhat restrictive, because it does not allow for the abstractions on a higher level than those present in the source language.

We propose a project-specific language in this paper, which is an extension of the native patterns idea. Our idea is not only to define syntactic patterns commonly used in the source programs, but also to assign meanings to them and view them as language entities in their own right. As far as we know, this approach has not been applied in practice before.

Papers describing methods of extracting the informal knowledge from programs (see [8, 9, 10, 11]) rarely discuss how this knowledge could be used for legacy code transformation. This may be due to the fact that the informal knowledge and its representation change from system to system, and therefore automated application of this knowledge makes sense only for the the duration of one project.

The issue of more efficient search patterns in the source system for the purposes of transformation has been already raised in article [12] and in a later book dedicated to Programmer's Apprentice project [13]. The problem is that there may not be more generalized templates for extraction of higher-level abstractions from the legacy code. In this paper we propose not to look for the patterns viable for the whole language, but only for used in one particular project.

1.2 Acknowledgements

We would like to thank Chris Verhoef (Free University of Amsterdam) and the anonymous reviewers of the preliminary version of this paper for their comments and advice. We would also like to thank Karina Terekhova (Jipeq eSourcing Oy, Finland) for her invaluable help during improvement of the final version of this paper.

1.3 Organization of the paper

Section 2 discusses what is informal knowledge and how it is used. Section 3 describe in some detail the different representations of project-specific language and its application to language conversion. Section 4 is the discussion. The paper ends with conclusions and an appendix.

2 Formal semantics and informal knowledge

High-level programming languages and compilers were introduced to close the semantic gap between human language and computer assembler language. All redundant information, that does not affect the execution of the program, is ignored by the compiler at translation. Therefore, we can say that *formal information* consists of language elements affecting the operational semantics, and *informal information* represents all other elements of the program which serve to enhance the human understanding of the code.

The informal part of a program is important not only during dev elopment, but also for later maintenance and possible reengineering. Alterations of the program after its initial completion, suc h as adding corrections during the maintenance period or addition of new programmers to the maintenance team usually require renewal of program understanding.

We shall focus our attention on a specific type of informal knowledge, namely, the language constructions used in a special way in a project. Let us start with a small example in PL/I that has been adapted from an industrial application. The program presented below simulates support for object–oriented constructions by means of PL/I preprocessor. This idea may seem a bit artificial, but there are several programming languages built as preprocessor layers ov er another language (for example, Ratfor [14] and Objective C [15]), and sometimes this approach is used in real legacy applications to implement the mechanisms lac king in the source language.

```
%include maclib;

p: proc options (main);

 CLASS (point);
  PROPERTY (x) AS (fixed bin);
  PROPERTY (y) AS (fixed bin);
 ENDCLASS;

 MEMBER(get_x) OF(point) RETURNS(fixed bin);
   return (THIS.x);
 ENDMEMBER;

 MEMBER(get_y) OF(point) RETURNS(fixed bin);
   return (THIS.y);
 ENDMEMBER;

 MEMBER(set_x) OF(point) TAKES(value);
   dcl value fixed bin;
   THIS.x = value;
 ENDMEMBER;

 MEMBER(set_y) OF(point) TAKES(value);
   dcl value fixed bin;
   THIS.y = value;
 ENDMEMBER;

 dcl pnt like point;

 INVOKE(set_x) FROM (pnt) PASSING (1);;
 INVOKE(set_y) FROM (pnt) PASSING (2);;

 dcl (x1, y1) fixed bin;

 x1 = INVOKE(get_x) FROM (pnt);;
 y1 = INVOKE(get_y) FROM (pnt);;

 put list ('(', x1, ', ', y1, ')');

end;
```

In this example the names of macros and their parameters are typed in capital letters. The example starts with inclusion of file `maclib`, which implements all macros used in this program (see also Appendix). Then we define class `point`, which contains t wo coordinates `x` and `y`, and also has special access methods for these variables. Finally, we define variable of type `point` and use functions `set_x` and `set_y` with some parameters.

It is clear that the usual reengineering tools do not cope v ery well with this kind of syn tax in a source programs. In the w orst case, all peculiarities of syntax will be removed at the preliminary stage of preprocessing. Even in the best case, there will be no w ay to capture this knowledge about the project and to exploit it.

We can formulate several problems that hamper our efforts to create a general scheme of legacy systems analysis:

- Source programs are usually written in a subset of the language. This subset is sometimes hard to define formally, but possible to identify using simple heuristics. Here w e understand "subset" not as a fixed set of constructions, but rather as a special way of their usage.

- Most legacy systems even tually lose some importan t information (missing files, utilization of unspecified compiler environment etc.), so that the formal analysis is not always possible.

- Programs can contain fragments written in a differen t programming language, which cannot be analysed in terms of syntax and semantics of the source language.

So one might say that eac h project is written in its own language, consisting of peculiarities of the source language, program and development environments, style conventions etc. This leads us to the idea of analysing the project not in terms of the source language, but in terms specific to the project itself.

3 Iterative extraction of language specific to the project

Iterativ e customization of native patterns requires as input a system written in a legacy language and a reengineering tool, which can parse it and build an in termediate representation of it. First step is to start analysing the language constructions. Any prospective candidates found are added to the project-specific language, so that each consecutive iteration increases the number of language constructions and thus enhances the accuracy of analysis. The process of language extension stops when no new constructions are added.

Extraction of a project-specific language leads to creation of a "bigger" language constructions on the basis of the source programming language. We can describe this process as replacement of some constructions with a new ones, equivalent to them from the application's point of view, and truncation of all unused parts of the initial programming language.

Our first task is to formulate syntax, semantics and pragmatics of this project-specific language in a way, which will be the most conv enient for further processing. T o do this, w e ha ve to decide where w e will store the information about the informal kno wledge that is extracted during analysis of the legacy system. Since we are going to use this information only during reengineering process, we do not need to describe it as y et another programming language. F or our purposes, it is quite sufficient to work with the internal representation of the program, so we decided to store all the information about specifics of the project directly in our reengineering tool, customizing it for large reengineering projects. In our opinion, this approach is economically justified because of the huge size of most legacy systems and the fact that most reengineering projects are starting with updates to the existing migration tools anyway (for instance, to support the exact dialect of the legacy language used in the system).

There are other approaches that employ quite the con trary approach, i.e. that store all information about the project in its source code. F or instance, see [16] for description of scaffolding technique that could be used to describe such extensions to the language.

3.1 Legacy understanding and tool customization

The knowledge about the project-specific language can appear in the system in at least tw o different forms.

In the first case w e hav e *syntactic extensions* to the language. We already quoted extensions defined as macros, but there are other forms as well. For instance, consider the follo wing fragment tak en from a real PL/I application:

```
DCL ISPLINK ENTRY EXTERNAL
    OPTIONS(ASM,INTER,RETCODE);
DCL TABEL CHAR(08);

CALL ISPLINK ('TBBOTTOM', TABEL);
...

CALL ISPLINK ('CONTROL ',
    'ERRORS  ','RETURN  ');
...

CALL ISPLINK ('TBSORT  ',
    TABEL, '(OPKRNR,N,D)');
...
```

Here w e are dealing with a typical "open-ended" application where extraction of knowledge in terms of implementation language makes no sense, because assembler function ISPLINK used in this program could be regarded as a "magic wand" and its semantics could not be extracted in terms of PL/I. In the mean time, it is clear that this function simply extends source language with additional functionality. Hence these function calls hav e completely different meaning compared to usual PL/I function calls, even though from the point of view of implementation language there is practically no difference betw een them.

Syntactic extensions are usually local and quite w ell-defined, so it is possible to represent them as additional language constructions that are orthogonal to the legacy language.

In other more complicated cases informal kno wledge is spread ov er different constructions of the language and do not yield to a simple syntactic description (we call this t ype of informal knowledge *semantic extensions*). In particular, they could be expressed in a specific organization of control flow, rules of transferring data from one construction to another etc. Naturally , these idioms could not be represented in a purely sin tactic form (or at least, such representation would be clumsy and ov erly complicated).

Let us illustrate this with the following example:

```
p: proc;
...
IF I<1 OR I>HBOUND(B)
THEN GOTO ERR_HANDLER;
```

```
...
IF SQLCODE ~= 0 & SQLCODE ~= 100
THEN GOTO END_DEL;
...

END_DEL:
    CALL B_1001_PROCESS_ERROR;

ERR_HANDLER:
    /* some code to handle
       OutOfBounds exception */
    ...
end p;
```

In this example we have an implementation of exception handling that uses GOTO statements. Quick examination of this and other programs showed that in this application all error handling routines are concentrated at the end of the corresponding procedures; note that calls to separate error handling routines can be also employed. None of the programs handles their exception handling "inline" and thus we conclude that the application is written in more or less structured way.In fact, this approach to error handling is very typical for older languages: several languages, such as COBOL, PL/I and BASIC, contained special constructions such as `On Error GoTo` for this functionality, though their usage was usually limited to some predefined situations.

Now we can take advantage of this knowledge about the project by adding it to the tool. Unfortunately, we cannot write it down as a syntax rule except in the case of `On Error GoTo` construction, because generally our transformation of the program will be non-local. The operations that should be performed with procedures that contain the error handling are the following ones. We should remove the error checks after each SQL statement and checks for index validity before accesing arrays. Instead we should add new constructions named ExceptionToCheck to the beginning of procedure and also another new construction, ExceptionToHandle, to the end of procedure. These constructions would employ several parameters that describe the type of exception and possible conditions of its appearance for the former and what should be done to handle the exception for the latter.

It is clear that this kind of transformation requires a mechanism that is stronger than simple syntax rule. So we have to use two different forms for storing the extracted project-specific language. In the first case we can simply add new rules describing syntactic extensions to the grammar of the language. One might say that in this case we have the project-specific language in its explicit form. In the second case we represent the new idioms in the form of rewriting rules that are applied to the intermediate representation of the program. This can be viewed as an implicit representation of the project-specific language. Naturally, the latter approach is more general but also more laborious.

In both cases, the process of formalization of the project-specific language is inherently manual, because it is not possible to predict in advance which constructions of the source language will constitute new idioms. So the process of "enriching" the language inevitably depends on human engineers finding and writing down the special rules for a given legacy system.

Nevertheless, we regard the process of updating the reengineering tool to the project-specific language as rather simple, especially compared to the complexity of creating the initial reengineering tool and compared to the complexity of a typical reengineering project. Moreover, the updating process is fairly technical and requires only the knowledge of typical techniques in compiler construction and reengineering.

For instance, the syntactic extensions are added to the parser by means of adding new generalized constructions to the language grammar. This is similar to the process of updating the parser in order to support a new dialect of the language. Usually, one of the most difficult problems in this process is resolution of conflicts that appear during addition of new syntactic rules to the grammar. But in our case this was not an issue, since our PL/I parser was written using recursive descent method This design decision was made because PL/I contains several features that could not be properly supported by YACC:

- Some PL/I constructs require potentially infinite lookahead, which is not provided by YACC;
- Another problematic feature of PL/I is that it allows using keywords as identifiers.

This simplification aside, there exists a number of syntax description and modification mechanisms that could be used for quick implementation of new features in a project-specific language. For instance, paper [17] proposes a flexible scheme that enables easy updates to the parsers of legacy languages. This scheme was originally devised as a method of supporting new dialects of the language, but it could be applied for our purposes as well. Term rewriting could be also regarded as quite technical are by now, though detailed discussion of this method is beyond the scope of this

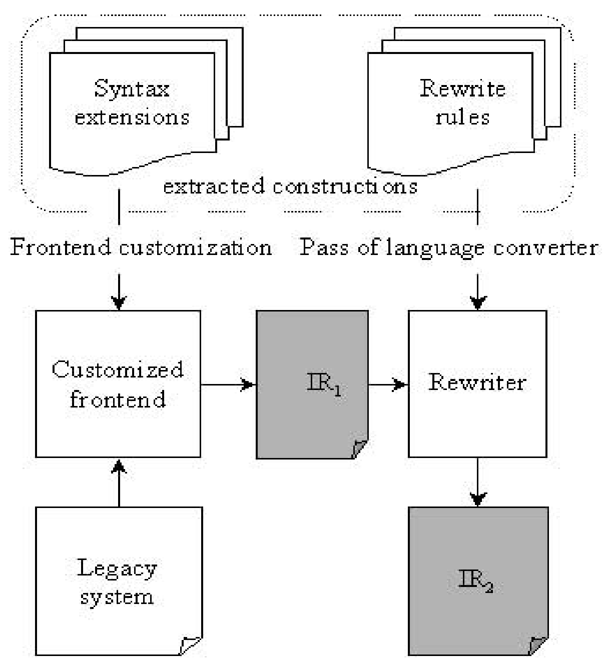

Figure 1: Sc heme of language extraction

article (we refer readers to the work [18] for the thorough account of this topic).

3.2 Updated scheme of the language extraction process

Now we can summarize the process of language extraction in a more detailed scheme (see Figure 1).

Creation of project-specific language consists of sev eral stages. The first stage is front-end customization, where w e extend the grammar b y adding new syntax rules. Application of this cusomized fron tend yields the initial in ternal representation (IR_1), where all idioms that can be expressed syntactically are already reduced. Then this internal representation goes through term rewriting stage, which completes description of the project-specific language. The result of the term rewriting, denoted as IR_2, is the final in ternal representation of the project in its own language. All further works on legacy understanding or language conv ersion should be based on this intermediate representation.

Note that theoretically we can express all syntactic extensions at the term rewriting stage and thus completely remov e the parser customization phase, but in our opinion this would only complicate the process.

3.3 Customized generation of target languages

Defining project language and storing this kno wledge in the reengineering tool is not really helpful unless w ecan use it. In this section w ewill sho who w knowledge about project-specific language can be used during con versionof the legacy system to new programming languages.

It is w ell known that the conv ersion process is difficult and even contradictory because of the differences betw een source and target programming languages [2]. One of the first tasks during language con version is to define conv ersion strategy for every construction of the source language. Usually this list of projections is fixed once and for all, but this makes it impossible to use our knowledge of conv en tionsadopted in this project, because our language con verter can operate only in terms of the native constructions of the source language. In other words, in traditional approach we are always low eringthe lev el of the program to the level of the source language.

Instead of this we can try to customize our target language generation based on features of the project-specific language. It is quite possible that we will find better equivalen ts for them in the target language, especially if the target language has richer set of native language constructions than the source one (for instance, it can support objects, complex data types or simply contain more useful functions in its standard environment).

T o illustrate our reasoning we will return to the first example. Suppose that during legacy understanding w emanaged to determine that the project language is object–oriented and we reflected this knowledge in our parser. In this case the result of transformation to Java will hav e the look and feel of native Jav a code:

```
class Point {
  private int x, y;

  public int get_x()          { return x;  }
  public int get_y()          { return y;  }
  public void set_x(int value) { x = value; }
  public void set_y(int value) { y = value; }
}

class p {
  public static void main(String args[]) {
    Point pnt = new Point();
    pnt.set_x(1);
    pnt.set_y(2);
    System.out.print("(" + pnt.get_x()
      + "," + pnt.get_y() + ")");
  }
}
```

In the meantime transliteration of the source program to Java will end up with the code, in which the meaning of the program will be completely obscured by inessential details.

Similar results could be achieved with error handling example. General idea of conversion for this example could be represented in the following skeleton of the program:

```
class p {
  public static void main(String[] args) {
    try {
     ...
     /* statements such as IF I<1 ...
        and IF SQLCODE <> 0 ...
        simply disappeared */
    }
    catch (SQLException e) { ... }
    catch (ArrayIndexOutOfBoundsException a)
      { ... }
  }
}
```

As a result of customization of our reengineering tool, we generate programs that manage to reflect idioms that were emulated in the original legacy systems and are natively supported in the target language. If we know in advance that legacy system is to be converted to some other language, we can specially look for patterns in the original programs that could be conveniently represented in the target language. But of course, we should not limit ourselves only to this scenario.

4 Discussion

The biggest concern about applicability of project–specific languages in reengineering is the correlation between the cost of extracting the project-specific language and the benefits that it promises. While it is not possible to make any trustworthy predictions on the basis of one application, we believe that notable advantages in the quality of the generated code should clearly outbalance the cost of updating the tool. However, we have to admit that the assumption that is implicitly made in our approach — that it is possible to update the reengineering tool itself during legacy transformation — is quite strong and may not be an option in a lot of reengineering projects.

For this and other reasons we do not expect an approach described in this article to be effective in all cases. On the contrary, if the application to be reengineered is small, then the cost of customizing the tools would become too high to consider this possibility. But then again, the most difficult problem is usually the sheer size of the legacy system that requires reengineering, and for project–specific languages the bigger is usually the better, because the time spent on updating the tools can be more than compensated for during subsequent application of these tools.

For instance, just the before submission of this paper we had an opportunity to assess applicability of our approach on an example of large legacy system that consisted of nearly two million lines of code in PL/I. In-depth study of this system showed that the majority of the code in this application was actually generated by preprocessing of some templates. Naturally, this observation has changed the entire method of attacking the problem. It is clear that in this case the original templates for preprocessing can provide much more information than the results of their expansion. Once we determine the structure of these templates, we can instantly add them to the project–specific language of this system and apply these definitions during subsequent language conversion.

Another objection to proposed approach is that in industrial system written by large team without strict discipline of programming we may fail to find repeatable and widespread templates. And indeed, most of us have encountered a lot of examples of "spaghetti" code where the templates suitable for inclusion into project language were scattered all over the source code.

This problem is really difficult, but in our opinion it can be successfully dealt with. We can start with preliminary restructuring of the code, which is needed for this type of legacy systems anyway. Also, even in these unstructured systems we can find useful patterns — only it is more likely that they will be more complicated and would not look like simple syntactic extensions.

As a side remark, we would like to point out that any reengineering technique would be difficult to apply to these unstructured systems. In fact, all reengineering techniques are heavily favoring good programming style: even program restructuring that theoretically should work well with any kind of input can produce better or worse results depending on the initial program.

Finally, we think that there is a lot of room for improvement in the area of tool support for the process of language extraction. For example, it would be good to have special tools for validation of our hypotheses about constructions that are under consideration. In our example from section 2 we supposed that we can interpret some variables as an internal data of some object, and functions as members of that object. It would be useful to be able to check whether object–oriented semantics is really adequately supported.

Ideally, verification of our hypotheses should be (partially) automated, even though formal verification of hypothesis is usually quite expensive [19]. Currently this stage is supported only by existing legacy understanding functionality, such as diagramming and program navigation tools.

5 Conclusion

In this article we presented an approach that exploits informal knowledge about the legacy system. We believe that this approach could be useful during reengineering process, and so the "ideal" reengineering tool must support the interactive process of understanding the legacy system and allow customization for any given legacy system.

We proposed the notion of project–specific languages and demonstrated the possibilities of its application in software reengineering. Our approach was validated in a real–life project.

6 Appendix

In this appendix we would like to show construction of object–oriented extension of PL/I using its preprocessor. We go into these details only to prove technical feasibility of this approach; this implementation is not essential for the purposes of the paper.

```
%PROPERTY: proc (NAME, AS)
  statement returns (char);

  dcl (NAME, AS) char;

  return ('2 ' || NAME || ' ' || AS || ',');
%end;

%MEMBER: proc (NAME, OF, TAKES, RETURNS)
  statement returns (char);

  dcl (NAME, OF, TAKES, RETURNS) char;
  dcl parmset  builtin;
  dcl parmlist char;

  parmlist = '_this';

  if parmset(takes) & takes ^= ''
  then parmlist = parmlist || ', ' || TAKES;

  return
  (
    NAME || ': proc (' || parmlist
    || ') returns (' || RETURNS || ');'
    || 'dcl _this pointer, _data like '
    || OF || ' based (_this); '
  );
%end;

%ENDMEMBER: proc statement returns (char);
   return ('end;');
%end;

%CLASS: proc (NAME) statement returns (char);
   dcl NAME char, classname char;
   return ('dcl 1 ' || NAME || ',');
%end;

%ENDCLASS: proc statement returns (char);
  return (';');
%end;

%INVOKE: proc (NAME, FROM, PASSING)
  statement returns (char);
  dcl (NAME, FROM, PASSING) char;

  if PASSING ^= ''
  then PASSING = ', ' || PASSING;

  return ('CALL ' || NAME || '(ADDR('
  || FROM || ')' || PASSING || ')');
%end;

%dcl THIS char;
%THIS='_this->_data ';

%act PROPERTY, CLASS, ENDCLASS, MEMBER,
  ENDMEMBER, THIS, INVOKE norescan;
```

References

[1] A.N. Terekhov, L.A. Erlikh, A.A. Terekhov "History and Architecture of RescueWare Project" // In "Automated Software Reengineering", A. N. Terekhov, A. A. Terekhov (eds.), St. Petersburg, 2000, pp. 7–19 (in Russian)

[2] A.A. Terekhov, C. Verhoef "The Realities of Language Conversions", IEEE Software, November/December 2000, Vol. 17, No. 6, pp. 111–124.

[3] A. Quilici "A Memory-Based Approach to Recognizing Programming Plans", Communications of the ACM, Vol. 37, No. 5, pp. 84–93, 1994.

[4] Z.-Y. Liu, M. Ballantyne, L. Seward, "An Assistant for Re-Engineering Legacy Systems", In Proceedings of the 6th Conference on Innovative Applications of Artificial Intelligence, pp. 95-102, 1994.

[5] C. Rich, L. M. Wills "Recognizing a Program's Design: A Graph-Parsing Approach", IEEE Software, Vol. 7, No. 1, pp. 82–89, January 1990.

[6] A. Quilici, S. Woods, Y. Zhang "Program Plan Matching: Experiments with a Constraint-Based Approach", Science of Computer Programming, Vol. 36, No. 2–3, pp. 285–302, 2000.

[7] A. Sellink, C. Verhoef "Native Patterns", In Proceedings of the 5th IEEE Working Conference on Reverse Engineering, Honolulu, Hawaii, USA, pp. 89–103. 1998.

[8] W. Kozaczynski, J. Q. Ning, A. Engberts "Program Concept Recognition and Transformation", IEEE Transactions on Software Engineering, Vol. 18, No. 12, December 1992, pp. 1065–1075.

[9] W. Kozaczynski, J. Q. Ning "Automated Program Understanding by Concept Recognition", Automated Software Engineering Journal, Vol. 1, No. 1, 1994, pp. 61–78.

[10] T.J. Biggerstaff, B.G. Mitbander, D.E. Webster "Program Understanding and the Concept Assignment Problem", Communications of the ACM, May 1994, pp. 72–82.

[11] L. H. Etzkorn, C. G. Davis "Automatically Identifying Reusable OO Legacy Code", Computer, October 1997, pp. 66–71.

[12] R. C. Waters "Program Translation via Abstraction and Reimplementation", IEEE Transactions on Software Engineering, Vol. SE-14, No. 8, August 1988, pp. 1207–1228.

[13] C. Rich, R. C. Waters "Programmer's Apprentice", ACM Press, 1990. 238 pp.

[14] B. Kernighan, P. J. Plauger "Software Tools", Addison-Wesley, Reading, MA. 1976. 338 pp.

[15] B. Cox "Object-Oriented Programming: An Evolutionary Approach", Addison-Wesley, Reading, MA. 1976.

[16] A. Sellink, C. Verhoef "Scaffolding for Software Renovation", In Proceedings of the 4th Conference on Software Maintenance and Reengineering, Zurich, Switzerland, 2000, pp. 151–160.

[17] M. van der Brand, A. Sellink, C. Verhoef "Current Parsing Techniques in Software Renovation Considered Harmful", In Proceedings of the International Workshop on Program Comprehension, Ischia, Italy 1998, pp. 108–117.

[18] J. W. Klop "Term rewriting systems", In Handbook of Logic in Computer Science, Vol. II, Oxford University Press, 1992, pp. 1–116.

[19] B. W. Weide, W. D. Heyrn "Reverse Engineering of Legacy Code Exposed", In Proceedings of the International Conference on Software Engineering, Seattle, WA, 1995, pp. 327–331.