

Program Comprehension in Multi-Language Systems*

Bernt Kullbach Andreas Winter Peter Dahm Jürgen Ebert
University of Koblenz-Landau
Institute for Software Technology
Rheinau 1, D-56075 Koblenz, Germany
(kullbach|winter|dahm|ebert)@informatik.uni-koblenz.de

Abstract

This paper presents an approach to program comprehension in multi-language systems. Such systems are characterized by a high amount of source codes in various languages for programming, database definition and job control. Coping with those systems requires the references crossing the language boundaries to be analyzed.

Using the EER/GRAL approach to graph-based conceptual modeling, models representing relevant aspects of the language are built and integrated into a common conceptual model. Since conceptual modeling focusses on specific problems, the integrated model presented here is especially tailored to multi-language aspects. Software systems are parsed and represented according to this conceptual model and queried by using a powerful graph query mechanism. This allows multi-language cross references to be easily retrieved.

The multi-language conceptual model and the query facilities have been developed in cooperation with the maintenance programmers at an insurance company within the GUPRO project.

1. Introduction

It is widely agreed that a software system becomes legacy as soon as it is delivered. Thus, software maintenance is becoming increasingly important. Following the worldwide benchmark project [37] only 36.9% percent of work are allocated to new development of software whereas

*This work has been performed within GUPRO (Generic Unit of Program understanding) which is a joint project of the IBM WT, Heidelberg, the University of Koblenz-Landau, Institute for Software Technology, Koblenz, and the Aachener und Münchener Informatik Service GmbH, Hamburg. GUPRO is supported by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie, national initiative on software technology, No. 01 IS 504. Information on GUPRO including technical reports cited in this paper is available at <http://www.uni-koblenz.de/~ist/gupro.en.html>.

52.9% are spent on maintenance, modification and migration.

Program understanding plays an essential role during the phases after software development. Already in the late seventies it has been reported that about 47% of maintenance efforts are spent on program understanding whereas only about 25% are spent on modification [18]. Following other studies, the program understanding effort is said to cover 40% [21], 60% [38] or even up to 90% [40]. As such, program understanding is the key activity during software maintenance providing a large potential to improving the efficiency of software development and maintenance processes.

There are various approaches to program understanding, which follow a common structure: Source codes are translated into a general data structure which is analyzed afterwards. Different choices for representing source code information exist, such as relational databases [30], [5], [20], PROLOG databases [26], [3], object-oriented databases [31], [34], abstract syntax trees [12], [34], [42], [7], LISP images [33] or hybrid knowledge bases [25]. The repository structures are described in terms of textual languages [42], entity-relationship languages [25], [3] or formal algebraic models [34]. Coarse-grained repository definitions are given for a PASCAL-like language [30] and for C [5], [34]. A set of three data structures with different level of granularity is defined in [25].

The way chosen for representing program information implies the analysis mechanism. In the database-driven approaches convenient database query mechanisms are used. ASTLOG [7] defines a PROLOG-based query language for analyzing abstract syntax tree representations. In [25] a database-independent program query language (PQL) is introduced. An algebraic expression-based query language has been implemented within the Software Refinery Toolset [34].

Although these approaches are in general suited to represent multi-language systems, only single-language implementations are known to the authors. These are not much

suiting to support program understanding in multi-language systems which are the usual thing in today's enterprises. Most of these real world software systems consist of several – sometimes many – languages. This especially holds if a system is rather old, surviving through generations of programming languages and hardware architectures.

GUPRO differs from multi-language software repositories like the Software Bookshelf [17] or the Software Information Base [6] in that it focusses on program understanding and not on the management of software projects.

Our work is concerned with the software of the Aachener und Münchener Informatik Systeme GmbH (subsidiary of a German insurance company) which used an MVS system consisting of multiple sources in various programming languages, database definitions and job control languages. Altogether, the system consists of about 25 000 JCL-, 5 700 CSP-, 4 000 COBOL II-, 3 800 Delta COBOL-, 6 000 PL/1-, 1000 Assembler-, 100 REXX sources as well as programs written in languages like APL, SAS or Easytrieve. Furthermore 100 data models with about 3 000 entities and 60 000 attributes have to be considered.

In this paper, we present an approach to program comprehension in multi-language software systems. Section 2 describes how a multi-language model is established and how multi-language source codes can be parsed into an object-based repository. A coarse-grained conceptual model of a subset of the programming language environment as used in the insurance company is introduced. Within section 3 the analysis of this multi-language system is described. Section 4 sketches the realization of the multi-language program understanding system as part of the GUPRO-MetaCARE approach [14]. The paper ends with a conclusion and an outlook in section 5.

2. Representing multi-language systems

There exist various kinds of software maintenance tasks ranging from analyzing single program statements [42] to inspecting the overall architecture of a whole software system [43]. In building tools to support program understanding one has to define which aspects of software are relevant for the maintenance tasks to be approached. This definition can be performed by conceptual modeling techniques which can be derived from knowledge representation (e. g. conceptual graphs [39], $\pi\epsilon\lambda\sigma$ [32]) or can be based on entity-relationship modeling (e. g. [4], NIAM information structure diagrams [41], or even UML class diagrams [36]). Conceptual models are used to focus the attention on the relevant concepts of the software system and their interdependencies with respect to the given maintenance problem.

Within our work the EER/GRAL approach [16] to graph-based conceptual modeling is used. EER/GRAL is based on TGraphs [13] which are a very general class of graphs.

Conceptual modeling is enabled by an extended entity-relationship dialect (EER) and the GRAL constraint language [19]. The approach is based on a formal TGraph-oriented semantics, which is defined in [9]. Together with the GraLab implementation toolkit [10], EER/GRAL provides a seamless approach object-based to modeling and implementation.

In the following it is addressed how a conceptual model for multi-language software systems can be obtained using EER/GRAL and how source codes can be translated into a repository suited to this conceptual model.

2.1. Building a multi-language conceptual model

An important problem with maintaining multi-language systems is to cope with references crossing the language boundaries.

In MVS systems [2] e.g. the connections between sources from different languages are defined by job control procedures. From job control procedures programs are called. Furthermore, mappings between programs and databases are defined with respect to these calls. An application in such systems consists of a number of JCL procedures and a number of correlating programs and databases [23]. For identifying all programs belonging to an application, one has to find all programs being called by the respective JCL procedures. In this context indirect calls from these programs to other programs have to be considered too. Another significant cross referencing problem arises in database migration. Changing from one database system to another requires modifications in all programs accessing the respective data. These programs have to be detected by treating both, JCL procedures and database definitions, in conjunction.

A conceptual model for a multi-language software system capturing such kind of analysis has to include concepts representing the relevant source code aspects (e. g. JCL procedures or programs) as well as the conceptual relationships between them (e. g. calls or *isCalledBy*). Multi-language models have to be composed from single language models by identifying their interconnections.

Now the integration of MVS-JCL [2] and COBOL [24] shall be sketched as an example.

Figure 1 shows a coarse-grained conceptual model for MVS-JCL'. According to this, a *JclProcedure* consists of Steps. JCL **Steps** are to call **Programs**, provide bindings between physical data **Files** and logical file names

¹In the EER-dialect used vertex types are represented by rectangles, edge types are represented by (directed) arcs. A triangle symbol on an arc denotes generalization while aggregation is depicted by a rhomb at the vertex type rectangle. Relationship cardinalities are given by an arrow notation at the participating vertex types.

(*DdStmt*) and specify accesses to hierarchical databases via *Psb* definitions.

In Figure 2 a coarse-grained conceptual model for COBOL is introduced: *CobolPrograms* contain (*isModuleOf*) *CobolProcedures* which can call other *CobolProcedures* or other (not necessarily COBOL) programs (modeled as *callsProc* and *callsProgram* relationship, respectively). COBOL programs may also have access to data files (*accessesFile*). In the COBOL code this access is specified indirectly by a logical file reference (*DdFileRef*). As we will see the connection to a physical file can only be established in conjunction with a JCL procedure. For this reason, the *accessesFile* relationship in figure 2 is marked dashed.

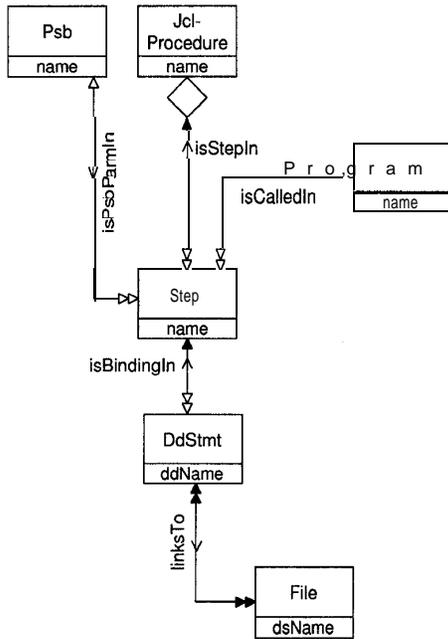


Figure 1. JCL single-language model

Within this multi-language model we are primarily interested in concepts relevant for inter-language cross-references. As such, concepts like internal definitions e. g. the COBOL data division are not represented in the model.

Building a common conceptual model requires the corresponding single-language models to be integrated. In this respect a very general integration strategy is adapted from the EER/GRAL approach to conceptual modeling [14]:

(A) Concepts which are similar are generalized into a com-

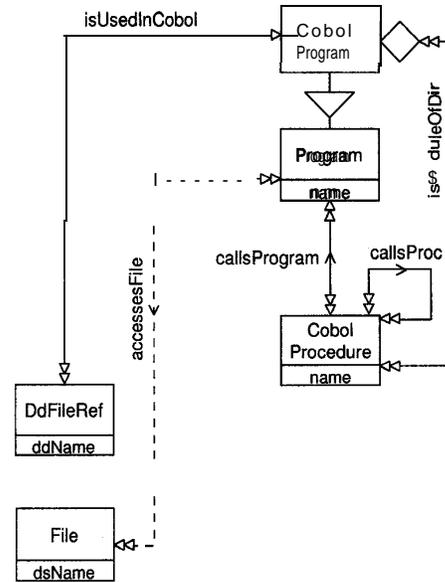


Figure 2. COBOL single-language model

mon super concept. If they denote the same information they are unified.

(B) Concepts from different models, which have interconnections are connected by further conceptual relations.

When integrating the conceptual models for JCL and COBOL presented in figures 1 and 2, two common concepts can be identified: Firstly, a **Program** is called within a JCL **Step** as well as by a **CobolProgram**. Secondly, both models include bindings resp. accesses to **Files**. According to strategy (A), the concepts **Program** and **File** have to be unified from both models.

As an integration of type (B), a conceptual relation *accessesFile* between **Program** and **File** can be established, indicating the access of a **Program** to a **File**. The logical file reference (*DdFileRef*), used in COBOL programs is mapped to a physical file *in* a JCL step. *So, an accessesFile* conceptual relation between a **CobolProgram** and a **File** can only be established, iff the **File** is bound by some *ddStmt* in a call to a COBOL program within a JCL step. Furthermore, the *ddStmt* has to match the name of the logical file reference *DdFileRef* specified in the COBOL program called. This condition is specified by the GRAL constraint which completes the integrated conceptual model shown in **figure 3**

Just as in the case of COBOL, conceptual models of other programming languages can be added to this model, e.g. using an integration of **type** (A) *P11Programs* or

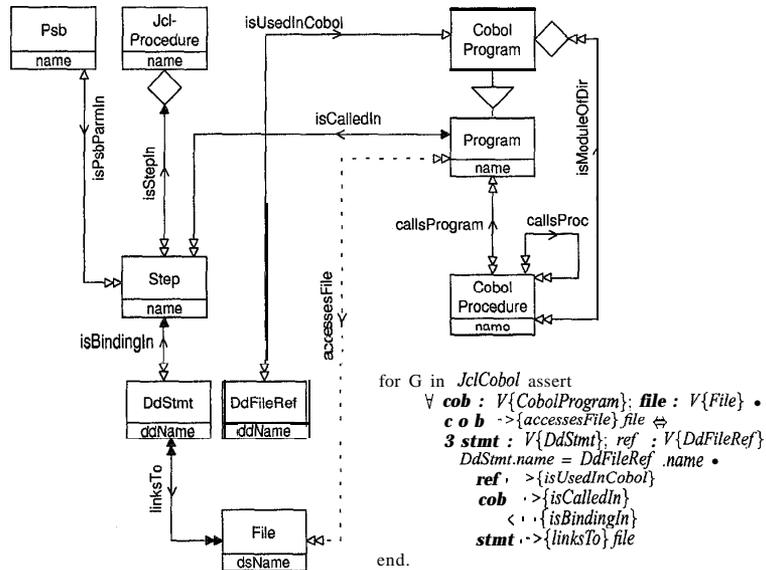


Figure 3. integrated conceptual models for JCL and COBOL (*JclCobol*)

CspApplications etc. can be integrated as additional sub-concepts of Program.

A slightly more complicated situation arises, when databases are added to the given integration. IMS (hierarchical) databases as an example consist of a tree-like structure of Segments. A subtree of this structure is mirrored in PSB specifications which are used by Steps. The JCL step is responsible for connecting the physical segments to programs. Again we have an integration step of type (B) where the access from programs to segments is defined by an additional conceptual relationship (c. f. *accessesSegment* in figure 4²). A respective constraint has to be defined with respect to IMS-DBD, PSB, JCL, and the respective programming language:

for G in Macro assert

```

Vprog : V{Program}; seg : V{Segment} •
  prog . . > {accessesSegment} seg ⇔
  prog . . > {isCalledIn} < . . {isPsbParamIn}
  . . > {usesSegRel}
  ( < . . {isRoot} < . . {isChild}
  < . . {isParent} ) seg

```

end.

²The EER dialect depicts generalization by the usual triangle notation but also allows an alternative notation by nesting object types which is very useful for clustering EER diagrams. Within both notations an abstract generalization is symbolized by hatching.

According to this constraint an *accessesSegment* edge is introduced between all vertices representing a program (Program) and all vertices representing a segment (Segment) iff the segment is child (*isChild*), parent (*isParent*), or root (*isRoot*) of a segment relation which is used by (*usesSegRel*) a program specification block (Psb). This has to be a parameter in (*isPsbParamIn*) a JCL step in which the program is called (*isCalledIn*). □

The integration as presented so far is part of the multi-language model representing the programming environment at Aachener und Miinchener as depicted in figure 4. The model has been developed together with maintenance engineers at the insurance company [11]. Exceeding its use as a basis for the program comprehension tool, the model has been successfully used to communicate about software artifacts. The model is the result of an integration of MVS/JCL, COBOL, PL/I, CSP/ESF, IMS-DBD, PSB, as well as SQL DML and DDL.

According to this model, a file (*SourceFile*) may be viewed as the uppermost concept generalizing the respective concepts of the single languages like *ImsDbdFile*, *Psb*, *JclProcedure*, *CobolFile*. The single-language file concepts relate to other concepts of the same language. E.g. a *CspApplication* is an aggregation of records (*CspRecord*), tables (*CspRecord*), and maps (*CspMap*). The multi-language cross-references are represented by relationships between single-language concepts. E.g. a CSP record represents an SQL DML statement (*SqlDmlStmt*).

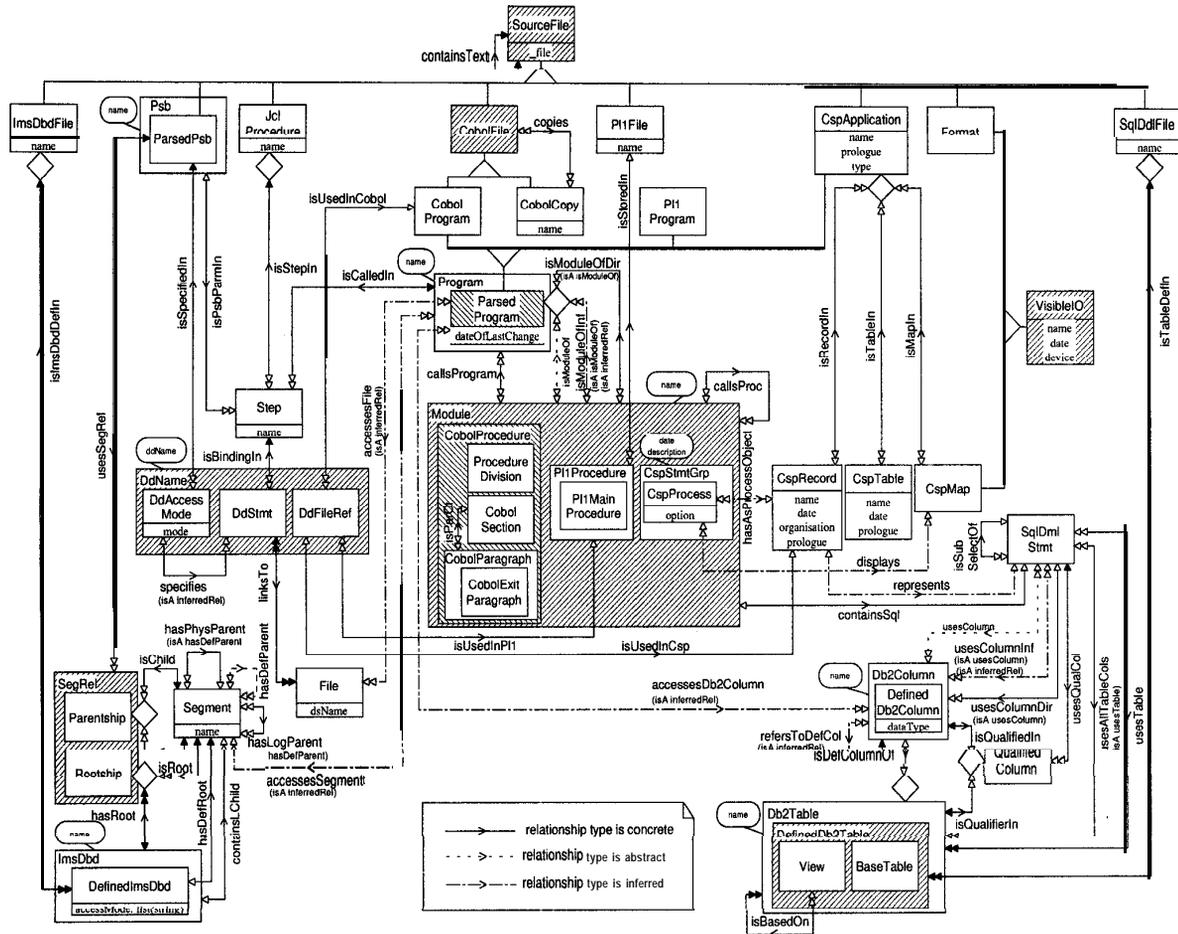


Figure 4. GUPRO multi-language conceptual model (Macro)

2.2. Filling a multi-language repository

The multi-language model defines the repository structure of a program comprehension tool. Following the EER/GRAL-approach the repository is held in an object-based graph representation.

In order to fill this repository, source codes have to be translated into the underlying graph representation. Because the software system consists of an enormous amount of documents, it is not feasible to fill the repository at once. As a consequence, the translation process has to allow an arbitrary parsing order of sources from various languages. Furthermore, the parsing facilities should enable updating (or even removing) repository entries, caused by changes in software components.

These requirements are met, using a four step parsing approach [29]. The first step checks if the document is already

represented in the repository in a former version. If so, it is removed. The document itself is parsed in a second step. In the third step, this representation is integrated into the repository and a fourth step is to ensure the integrity constraints. These parsing steps are controlled by a set of rules, which are derived from the conceptual model.

The demand of parsing source codes from different languages in an arbitrary order also affects the definition of the conceptual model. The model has to assure that incomplete conceptual relationships can be derived when all necessary informations are captured by once parsing the defining sources. In order to delete former code versions, the remove step has to keep the information that has been caused by other source codes parsed at an earlier time in the repository.

For a detailed description of this parsing processes see [29].

3. Querying multi-language systems

In order to retrieve information about a multi-language system, the repository is analyzed by a general, model-independent query mechanism. The query facility suited to the EER/GRAL approach is provided by the graph query language GReQL (Graph REpository Query Language) [28]. GReQL is an expression language offering restricted first order logic being especially suited to querying TGraph structures. GReQL queries may include path expressions as defined by GRAL [19]. This allows complex relationships between the concepts represented in the repository to be included. GReQL comes along with an extensible library of built-in functions and predicates. These include TGraph-specific functions and relations as well as aggregate functions like counting, average computing, and such.

In the following, the GReQL language shall be explained in some detail along with the MVS application example:

To get the general idea of all software modules making up an application, the maintenance programmer has to find out which programs are called by the JCL procedures defining this application. As said before, indirect calls are of interest too. A GReQL query addressing this task looks as follows:

```

FROM      jcl : V{JclProcedure},
          called : V{Program}
WITH      jcl
          <-- {isStepIn}
          <-- {isCalledIn}
          (<-- {isModuleOf}
          --> {callsProgram}) *
          called
REPORT    jcl.name, called.name
END

```

This query embodies the basic GReQL construct which is the FROM-WITH-REPORT expression. The FROM clause introduces variables for the concepts and conceptual relationships to be used. Within the example query these are of type *JclProcedure* and *Program*, respectively. The WITH clause imposes a restriction on the possible values of the variables defined. According to the conceptual model in figure 3 (which is a subpart of the model in figure 4) the call of a program from a JCL procedure is represented by a path connecting an object of type *JclProcedure* (*jcl*) with an object of type *Program* (*called*). In the case of direct calls this path has to follow an *isStepIn* and an *isCalledIn* relation each in opposite direction. This is denoted by the path-expression *jcl <-- {isStepIn} <-- {isCalledIn} called*. For including the indirect calls the path description *<-- {isModuleOf} --> {callsProgram}* has to be inserted. Each *Program* object may contain some modules, reached by an *isModuleOf* relationship in reverse order, which themselves call other *Program* objects. To retrieve

all indirectly called programs the reflexive, transitive closure (denoted by *) over this subpath has to be calculated. The REPORT clause defines how the output of a query is composed. Within our example only the *JclProcedure* and *Program* identifiers are reported. These are specified in the conceptual model as name attributes. The REPORT clause again may contain further FROM-WITH-REPORT expressions resulting in nested queries.

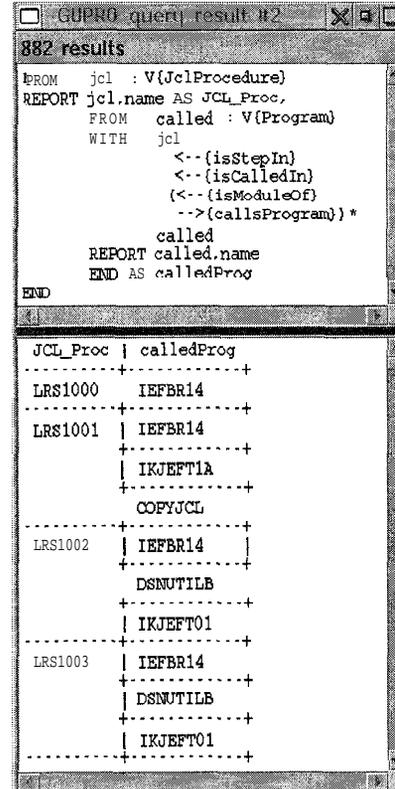


Figure 5. GUPRO Query

Figure 5 shows a query screenshot presenting the same question as a nested GReQL query together with the query result in a tabular form. The output of a FROM-WITH-REPORT expression is always a bag. Within the implementation this bag is interpreted and visualized as a table. □

In order to give a deeper insight into GReQL and its use in querying multi-language systems the IMS-DBD maintenance problem sketched in section 2.1 should be approached.

Example. If one has to replace an IMS database by a relational one, all programs using this database are affected.

Unfortunately, programs are connected to IMS databases only indirectly via JCL and PSB definitions. Starting from an *ImsDbd* object (ims) those Programs (prog) are of interest, which can be reached by a path following a *hasDefRoot* relationship to a *Segment* object which itself is accessed by the Program through *Psb* and Step-objects. The complete path according to the conceptual model in figure 4, is given as follows:

```

ims -->{hasDefRoot}
  (-->{isChild}  -->{isParent}
   -->{isRoot})
  <--{usesSegRel}  -->{isPsbParamIn}
  <--{isCalledIn}
  (<- * {isModuleOf} -->{callProgram})*
prog

```

In the conceptual model of the multi-language system, the access of Programs to *segments* is also represented by *the accessesSegment* relationship defined in the GRAL constraint on page 4. This, of course, can be used in the GReQL query:

```

FROM      ims : V{ImsDbd};
          prog : V{Program}
WITH      ims
          -->{hasDefRoot}
          <--{accessesSegment}
          (<--{isModuleOf}
           -->{callsProgram})*
          prog
REPORT    ims.name, prog.name
END

```

This query yields pairs of IMS databases and those programs which might be affected by modifying the database specification. □

GReQL queries are evaluated by an eval/apply evaluator using an automaton-driven strategy for calculating path expressions efficiently (with respect to the repository content) [1] and including a query-optimizer [35]. To provide non GReQL wizards with querying facility a form-based user interface (with restricted power) has been implemented.

4. The current implementation

The program understanding tool for multi language-systems presented before has been developed as one instance of the GUPRO MetaCARE *toolset* [14].

This *toolset* provides a generic environment which is parameterized by a specification of the actual maintenance problem in order to derive concrete program understanding tool instance. This problem specification is defined by an EER/GRAL conceptual model as the one defined in figure 4. Software is parsed by parsers suiting to this model and represented in an object repository. These parsers are to a large extent generated by the PDL (Parser Description Language) parser generator [8]. PDL extends the Yacc

parser generator [27] by EBNF syntax and by notational support for compiling textual languages into TGraphs. A source code independent GReQL query facility, also working on the underlying TGraph model, is used for static source code analysis. The GUPRO MetaCARE system has been coded in C++ and runs under Sun Solaris and IBM AIX as well as under IBM OS/2.

The *toolset* for multi-language program understanding has been built by parameterizing the GUPRO system with the conceptual model defined in figure 4. Additionally, parsers for each language included in this model have been built as sketched in section 2. Analyzing the multi-language system as described in section 3 is performed by the general GReQL retrieval component. The multi-language GUPRO instance is currently being integrated into the software production and maintenance process at Aachener und Mtinchener.

Further GUPRO instances have been developed e. g. for supporting the understanding of ANSIR85-COBOL [22] and C [15], both on a very fine-grained level of granularity. This GUPRO C instance is used within the Reverse Engineering Demonstration Project [44]. An other GUPRO instance exists for analyzing the conceptual models used in the EER/GRAL-approach. Here GUPRO is applied to understanding the models used in GUPRO themselves.

5. Conclusions and future work

We presented an approach to program understanding in multi-language systems which has been used to represent the programming environment of the Aachener und Mtinchener insurance company. The work is formally based on the EER/GRAL modeling approach. In accordance with this, the single programming languages of a multi-language environment are modeled using an EER dialect together with the GRAL constraint language. Based upon common concepts the resulting models are combined into a single model using a general integration strategy. *Stepwise* parsing technology is used to fill a repository *defined* by the conceptual model with the varying source code information. Program comprehension is enabled on the repository using the GReQL query language. The GReQL queries presented make use of path expressions that exactly represent the cross references between the concepts of different languages in the model. The query-mechanism is used at the Aachener und Mtinchener to easily retrieve multi-language cross references. As such, the tool will provide the maintenance engineers with increased productivity.

In addressing the special needs of the Aachener und Mtinchener, the GReQL querying facility has been extended with a form-based query interface. Meanwhile the query facility has been integrated with a browsing component to let the user combine query and navigation as desired. This

component is also used for presenting query results in terms of source code instead of the tabular representation depicted in figure 5.

GUPRO offers scalability, in that an understanding-in-the-large as well as an understanding-in-the-small can be provided by individually adapting the granularity of representation. Within a coarse-grained model even large systems with MLOCs can be represented because only the relevant source code information is retained. Within a fine-grained model the very low-level relationships can be determined.

The multi-language program comprehension tool is just one instance of the GUPRO MetaCARE environment. In future, additional GUPRO instances will be developed. A fine-grained multi-language tool will be a desired goal. In this context new problems like different kinds of parameter exchange formats have to be encountered. But GUPRO and EER/GRAL seem to offer adequate support to this and upcoming problems.

References

- [1] M. Behling. Ein Interpreter regulärer Pfadausdrücke. Studienarbeit S-525, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, Dezember 1997.
- [2] G. D. Brown. *System 370/390 JCL*. Wiley, New York, 3 edition, 1991.
- [3] G. Canfora, L. Mancini, and M. Tortorella. A Workbench for Program Comprehension during Software Maintenance. In A. Cimitile and H. A. Müller, editors, *Proceedings Fourth Workshop on Program Comprehension, March 29-31 1996, Berlin, Germany*, pages 30-39. IEEE Computer Society Press, Los Alamitos, 1996.
- [4] P. P. Chen. The Entity-Relationship Model -Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9-36, March 1976.
- [5] Y.-F. Chen, M. Y. Nishimoto, and C. V. Ramamoorthy. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, 16(3):325-334, March 1990.
- [6] P. Constantopoulos, M. Jarke, J. Mylopoulos, and Y. Vassiliou. The software information base: A server for reuse. *The VLDB Journal*, 4(1):1-43, Jan. 1995.
- [7] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proceedings of the Conference on Domain-Specific Languages (DSL-97)*, pages 229-242, Berkeley, Oct. 15-17 1997. USENIX Association.
- [8] P. Dahm. PDL Reference. Technical Report (to appear), Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 1998.
- [9] P. Dahm, J. Ebert, A. Franzke, M. Kamp, and A. Winter. TGraphen und EER-Schemata, formale Grundlagen. Projektbericht in Vorbereitung, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 1998.
- [10] P. Dahm, J. Ebert, and C. Litauer. Das EMS-Graphenlabor 3.0. Projektbericht, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 1994.
- [11] P. Dahm, J. Fricke, R. Gimnich, M. Kamp, H. Stasch, E. Tewes, and A. Winter. Anwendungslandschaft der Volksfürsorge. Projektbericht 5/95, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 1995.
- [12] P. T. Devanbu. GENOA - A Customizable, Language and Front-End independent Code Analyzer. *Proc. 14th International Conference on Software Engineering, Melbourne*, pages 307-317, 1992.
- [13] J. Ebert and A. Franzke. A Declarative Approach to Graph Based Modeling. In E. Mayr, G. Schmidt, and G. Tinhofer, editors, *Graphtheoretic Concepts in Computer Science*, number 903 in LNCS, pages 38-50, Berlin, 1995. Springer.
- [14] J. Ebert, R. Gimnich, and A. Winter. Wartungsunterstützung in heterogenen Sprachumgebungen, Ein Überblick zum Projekt GUPRO. In F. Lehner, editor, *Softwarewartung und Reengineering - Erfahrungen und Entwicklungen*, pages 263-275. Gabler, Wiesbaden, 1996.
- [15] J. Ebert, B. Kullbach, and A. Panse. The Extract-Transform-Rewrite Cycle - A Step towards MetaCARE. In P. Nesi and F. Lehner, editors, *Proceedings of the 2nd Euromicro Conference on Software Maintenance & Reengineering*, pages 165-170, Los Alamitos, 1998. IEEE Computer Society. to appear in Proceedings of the 2nd Euromicro Conference on Software Maintenance & Reengineering, CSMR '98.
- [16] J. Ebert, A. Winter, P. Dahm, A. Franzke, and R. Süttenbach. Graph Based Modeling and Implementation with EER/GRAL. In B. Thalheim, editor, *15th International Conference on Conceptual Modeling (ER '96), Proceedings*, number 1157 in LNCS, pages 163-178. Springer, Berlin, 1996.
- [17] P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. A. Müller, J. Mylopoulos, S. G. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564-593, 1997.
- [18] Fjeldstad and Hamlen. Application Program Maintenance Study- Report to Our Respondents. IBM Corporation, DP Marketing Group, 1979.
- [19] A. Franzke. GRAL: A Reference Manual. Fachbericht Informatik 3/97, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, 1997.
- [20] J. E. Grass. Object-oriented design archaeology with CIA++. *Computing Systems*, 5(1):5-67, Winter 1992.
- [21] S. Henninger. Case-Based Knowledge Management Tools for Software Development. *Journal of Automated Software Engineering*, 4(3), July 1997.
- [22] M. Hümmerich. Entwicklung und prototypische Implementation eines konzeptionellen Modelles zum Reverse-Engineering von ANSI85-COBOL-Programmen. Studienarbeit S 380, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, Juni 1995.
- [23] IBM, editor. *MVS/Extended Architecture JCL Reference*. International Business Machines Corporation, North York, Ontario, 5 edition, September 1989.
- [24] IBM, editor. *VS-COBOL II, Application Programming Language Reference*. International Business Machines Corporation, San Jose, 8 edition, March 1993.
- [25] S. Jarzabek. PQL: a language for specifying abstract program views. In W. Schafer and P. Botella, editors, *Proc. 5th*

- European Software Engineering Conf. (ESEC 95)**, volume 989 of *Lecture Notes in Computer Science*, pages 324-341, Sitges, Spain, Sept. 1995. Springer-Verlag, Berlin.
- [26] S. Jarzabek and T. P. Keam. Design of a Generic Reverse Engineering Assistant Tool. In L. Wills, P. Newcomb, and E. Chikofsky, editors, *Proceedings Second Working Conference on Reverse Engineering, July 14-16 1995, Toronto, Ontario, Canada*, pages 61-70. IEEE Computer Society Press, Los Alamitos, Cal., 1995.
- [27] S. C. Johnson. YACC -Yet another compiler - compiler. Computing Science Technical Report No. 32, Bell Laboratories, Murray Hill, N.J., 1975.
- [28] M. Kamp. GReQL - Eine Anfragesprache für das GUPRO-Repository 1.1. Projektbericht 8/96, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, Januar 1996.
- [29] M. Kamp. Managing a Multi-File, Multi-Language Software Repository for Program Comprehension Tools A Generic Approach. In U. D. Carlini and P. K. Linos, editors, *6th International Workshop on Program Comprehension*, pages 64-71, Washington, June 1998. IEEE Computer Society.
- [30] M. A. Linton. Implementing Relational Views of Programs. *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 132-140, May 1984.
- [31] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A Reverse Engineering Approach To Subsystem Structure Identification. *Journal of Software Maintenance: Research and Practice*, 5(4): 181-204, Dec. 1993.
- [32] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarkis. Telos: Representing Knowledge About Information Systems. *ACM Transactions on Information Systems*, 8(4):325-262, October 1990.
- [33] P. Newcomb. Legacy System Cataloging Facility. In L. Wills, P. Newcomb, and E. Chikofsky, editors, *Proceedings Second Working Conference on Reverse Engineering, July 14-16 1995, Toronto, Ontario, Canada*, pages 52-60. IEEE Computer Society Press, Los Alamitos, Cal., 1995.
- [34] S. Paul and A. Prakash. Querying source code using an algebraic query language. In *Proceedings of the International Conference on Software Maintenance 1994*, pages 127-136. IEEE Computer Society Press, Sept. 1994.
- [35] D. Polock. Ein statischer Optimierer für GRAL- und GReQL-Ausdrücke. Diplomarbeit D-414, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, September 1997.
- [36] Unified Modeling Language. Notation Guide. Technical Report Version 1.1 alpha 6 (1.1 c), Rational Software Corporation, Santa Clara, 21 July 1997.
- [37] Rubin Systems Inc. The Worldwide Benchmark Project. Final Report 1997, 1998.
- [38] P. Selfridge. Integrating Code Knowledge with a Software Information System. In L. Hoebel, editor, *Proceedings of the 1990 Knowledge-Based Software Engineering Conference*, pages 183-195, 1990.
- [39] J. Sowa. *Conceptual Structures. Information, Processing in Mind and Machine*. The Systems Programming Series. Addison-Wesley, Reading, 1984.
- [40] T. A. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, SE-10(5):494-497, Sept. 1984.
- [41] G. M. A. Verheijen and J. van Bekkum. NIAM: An Information Analysis Method. In T. W. Olle, H. G. Sol, and A. A. Verrijn-Stuart, editors, *Information Systems Methodologies*, pages 537-589. North Holland, Amsterdam, 1982.
- [42] C. H. Wells, R. Brand, and L. Markosian. Customized Tools for Software Quality Assurance and Reengineering. In L. Wills, P. Newcomb, and E. Chikofsky, editors, *Proceedings Second Working Conference on Reverse Engineering, July 14-16 1995, Toronto, Ontario, Canada*, pages 71-77. IEEE Computer Society Press, Los Alamitos, Cal., 1995.
- [43] K. Wong, S. R. Tilley, H. A. Müller, and M.-A. D. Storey. Structural Redocumentation: A Case Study. *IEEE Software*, pages 46-54, January 1995.
- [44] WorldPath Information Services. Reverse Engineering Demonstration Project. <http://www.worldpath.com/reproject/>, 1998.