Call Graph of FORTRAN Captured by GXL

Martin Chlumecký

Abstract: The most traditional reverse engineering tools focus on an abstraction and an analysis of source code. An appropriate data structure is necessary for an analysing legacy source code and it enables transfer of useful information about reversing systems. This article aims to answer, how we can store a structure of legacy systems that can be used for an analysis. A structure of a program can be captured by a directed call graph. The author introduces a few existing approaches how to keep the graph in an appropriate data structure. Further, the author describes its use and its features and discusses issues. The structure of legacy FORTRAN programs is captured by this structure that is analysed and transformed into an object oriented form later. Among others the paper aims to create a metamodel of the FORTRAN language and to use GXL for its capturing. Further, the paper describes software for a work with reversing system and its graphic visualization. At the end, the author summarizes how a resultant structure can be used for an analysing of a legacy system.

Keywords: graph exchange language, reverse engineering, legacy software, FORTRAN, directed graph, visualization, call graph

JEL classificaton: O33, C88, I29

1 Introduction

Waters and Chikofsky stated that "year after year the lion's share of effort goes into modifying and extending preexisting systems, about which we know very little" [1]. Today, it is well known that massive computer programs are complex and very difficult to maintain, especially legacy software that include millions of source code lines. Understanding and modifying code in legacy software is very challenging because it is time consuming and costly. The documentation is usually out of date because designers have not maintained it. But Gerardo Canfora has described in paper [2] an importance of software maintenance. Nowadays, requests for modifying and extending legacy software are still required.

A large amount of software can be used for analysis legacy software, whether graphic or text interface. An input of software for analysis is a legacy program which is stored in an appropriate data structure. The problem is follows: a large part of the software uses a different form of input data. If we need to use more than one program, we have to convert an input and an output data into a required format. Fortunately, few standards for portability are developed in recent times. A source code of program can be represented by a direct graph, where each node represents a class or a method and an edge represents an association or a call of a subroutine. Dennis Strein has described in paper [3]

a structure of metamodel for program analysis. He used GXL for capture of observed program and GXL has been shown as a powerful tool.

In paper [3] is described only a metamodel which captured object-oriented programs. However, for the purpose of reverse engineering we need a different model. Primarily, it must capture i.e. "call graph" of individual subroutines.

This paper deals with a study of GXL in reverse engineering, more specifically, captured a structure of legacy procedural software and its use for analysis.

2 Reverse engineering

Reverse engineering is defined [13] as the process of analysing a subject system to identify the system's components and their interrelationship and create representations of the system in another form or at a higher level of abstraction. Reverse engineering involves extracting design artefacts and building abstractions that are less dependent on an implementation.

There are several methods how to use the reverse engineering:

1) Manually rewrite the existing source code.

2) Use an automatic language transformer.

3) Redesign and re-implement original software.

The first method fundamentally means to rewrite completely the legacy software to a target language. This approach does not need tools. However, it is usable only for small legacy software. This method produces many errors because people usually make mistakes. Further, it is very time-consuming and has a low efficiency.

The second method is not time-consuming because it uses supporting software. This approach transforms a legacy source code to a target language. It is fast and with minimal number of errors. This way of generating a new source code is very confused. The possibility of intervention, enlargement and modification is minimal.

The third method eliminates the main disadvantages of the two previous methods. The method starts by analysing requirements of legacy software. It continues by redesign of the software architecture and ends by an implementation of new software. There are many advantages of this method, especially a possibility to fix known bugs, enlargement about a new functionality and an opportunity to create a documentation of new architecture.

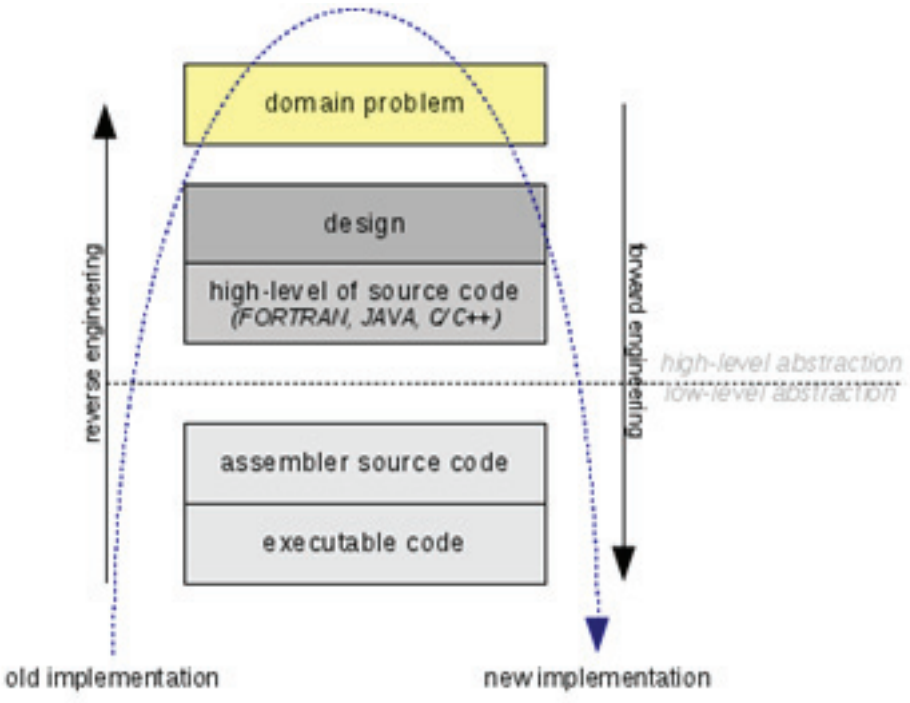

Figure 1. The levels of reverse engineering

The reverse engineering is a wide discipline of the software engineering. It aims to several levels. The figure 1 shows the levels of the reverse engineering. This paper describes re-engineering of the FORTRAN source code. It is occupied by a higher-level of abstraction.

2.1 Call graph

A call graph of a legacy program is a flow-graph CG=(FP,E,m), where: m ∪ FP is the set of nodes, FP is the set of functions and subroutine and m is the main program. E is the set of edges and describes the activation relation on (FP ∪ m)×FP. A call graph can be transformed into a hierarchy by using analysis techniques [16].

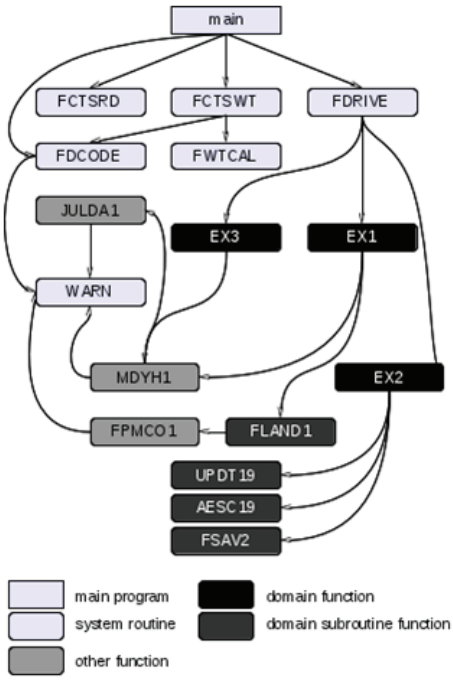

Figure 2. Example of call graph

The figure 2 shows an example of a call graph that was generated manually. The call graph is composed of nodes and edges and captures the flow control of legacy software. The nodes represent subroutines and the edges represent a calling from higher subroutines.

3 Presumption and our approach

There is described the architecture of a program can be stored in a directed graph which can be captured by GXL language in the introduction. A basic characteristic of this language is described (see below). Further, there is a necessity to create a metamodel of the FORTRAN language which is necessary for a mapping to GXL.

3.1 Graph eXchange Language (GXL)

Graph eXchange Language is an XML-based standard exchange format for sharing data between tools. Formally, GXL represents typed, attributed, directed and ordered graphs which are extended to represent hypergraphs and hierarchical graphs. An advantage of GXL is that it can be used to exchange instance graphs together with their corresponding schema information in a uniform format, i.e. using a common XML Document Type Definition (DTD). GXL has been ratified by reengineering and graph transformation research communities [4] as a standard exchange format in software reengineering at the seminar „Interoperability of Reengineering Tools“ in January 2001 [5].

3.1.1 Requirements for the GXL standard

GXL is graph-based, because graphs have an understandable mathematical foundation. Requirements are based on the fact that GXL can be used to represent data from a range of domains with disparate structures. GXL requirements are as follows:

1) Works for several purposes (e.g. various levels of abstraction).

2) Works for multi-million lines of code (e.g. 3 to 10 MLOC).

3) Maps between entities/relationships and source code statements.

4) Is incremental (e.g. one subsystem at a time can be added).

5) Is universal.

6) Is extensible.

The requirement no. 2 is an essentially scalability. It is an important feature for legacy software. It can be quite large, so fine-grained representations, such as an abstract syntax tree, can have millions of nodes and edges. The requirements no. 3 and no. 4 are derived from the software analysis problem domain.

3.1.2 Exchanging Graphs

Owing to their algorithmic and mathematical power, graphs are frequent data structure in software engineering. Different graph models e. g. directed or undirected graphs, node and edge attributed, trees etc. are applied in many tools for analysis [6].

GXL supports graphs representing a program. The graph can have directed or undirected edges, typed nodes and edges, attributes attached to nodes and edges, and ordered edges i.e. the edges incident with a particular vertex have a persistent ordering.

he structure of GXL is as follows: The root of a GXL document is enclosed in <gxl> tags. The GXL document can refer to a DTD regulation. Nodes and edges of a graph are illustrated by <node> a <edge> elements. Identification of a node or an edge is made by their attributes. The information about the orientation of edge is stored in from and to attributes within <edge> tags. Node and edge types are represented by links pointing to the corresponding schema information. The link is enclosed in <type> tags. GXL includes many more tags [6].

3.1.3 Interoperability by using DTD

To support interoperability of graph based tools uses DTD. Since GXL is an XML sublanguage. It implies the GXL graph model had to be transcribed into an XML document type definition.

The GXL DTD has only 18 XML elements. The GXL DTD begins by specifying predefined extension points for customizing GXL. These lines can be used to add sub elements or attributes to their corresponding graph elements. The rest of the DTD gives the syntax for graph components, attributes and references to schema information.

The DTD document provides a possibility to define a transformation rule which is used for a migration of one model into another.

4 The metamodel of FORTRAN

If we wanted to use GXL to catch the structure of a procedural source code, we need to create a metamodel of the FORTRAN language. This means to determine which information should be used from the source code.

The structure of procedural programs is divided into two levels. The first one, which is higher level, captures a structure of subroutines, its arguments and its calling by "call graph". The second one - lower level, captures an inner structure of subroutines. Primarily, it describes loops and the work with global variables.

In the first phase of analysis we need to capture the higher level. This structure has been described by Suman Roychoudhury in paper [8]. He described the aspect of the FORTRAN metamodel. However, for our purposes we need a different and preferable structure.

The figure no. 1 shows basic building blocks of the FORTRAN language of the higher level. A first building block is program. In a legacy project it must be only once.

Next elements of the FORTRAN hierarchy are methods which types are subroutine and function. Each of these methods is composed from no less than one Block data. The methods can call each other or recursively. The order of the calling of function and subroutine is very important. Information about the calling methods is stored in a description of an association that represents the direction of the calling.

The block data represents a sequence of instructions. For example, a command of an assigning, jump etc. But it does not include the instructions for calling of function and subroutine.

The structure of the methods is defined as a sorted set of block data, subroutines and functions. The ordering of the set determines the order of the execution of the commands. The description of the lower level is not a purpose of this paper.

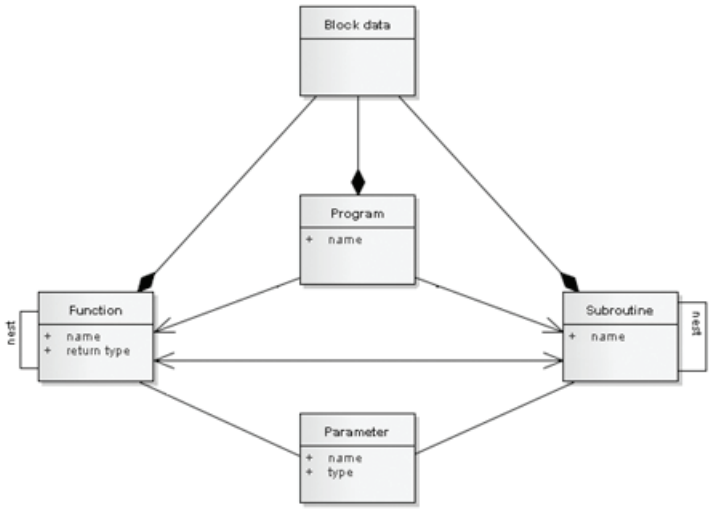

Figure 3. Metamodel of FORTRAN

4.1 The FORTRAN GXL metamodel

Now we can deduce a GXL model from the figure 1 above. A conversion of the source code of FOTRAN into the GXL model is straightforward. The classes on the figure 1 represent elements that are occurred in GXL. Their association means edges of a directed graph. An orientation of the edge says who is "called" and who is "calling". An order of the calling is determined by an attribute of a corresponding edge.

A mapping of elements of the source code shows the table 1. All edges in the graph of the higher level represent the calling of subroutines or functions. The direction of edges keeps GXL attributes form and to in the element edge. The order of the calling methods is ensured by the attribute order, which is an integer type. The nodes which represent functions and subroutines may contain attr elements. It stores a data type and a name of a corresponding argument. The order of these arguments in the GXL model responds with the source code which GXL captures.

syntax of FORTRAN	GXL element
`program $pname`	`<gxl name="$pname">`

`subroutine $sname($arg1)`	`<node name="$sname"`
	`rtype="nil">`
	`<attr name="$arg1" type="int">`
	`</node>`

`$rtype function $fname($arg1)`	`<node name="$fname"`
	`rtype="$rtype">`
	`<attr name="$arg1" type="int">`
	`</node>`

`subroutine $sname()`
`A = 1`
`call $sname1`
`B = 2`
`call $sname2`
`end`
	`<edge order="1"`
	`from="$sname"`
	`to="$sname1" />`
	
	`<edge order="2"`
	`from="$sname"`
	`to="$sname2" />`

Table 1. Mapping of FORTRAN to GXL

4.2 The FORTRAN DTD metamodel

The next step defines a DTD model which guarantees a correct use of GXL elements. DTD for the FORTRAN metamodel is as follows:

```
<!ELEMENT gxl (node* , edge*) >
<!ATTLIST gxl name ID #REQUIRED>

<!ELEMENT node ( attr*) >
<!ATTLIST node name ID  #REQUIRED>
<!ATTLIST node rtype ( int | real | double | bool | complex | common | nil)  #REQUIRED>

<!ELEMENT edge EMPTY >
<!ATTLIST edge order NMTOKEN #REQUIRED>
<!ATTLIST edge from IDREF #REQUIRED>
<!ATTLIST edge to IDREF #REQUIRED>

<!ELEMENT attr EMPTY >
<!ATTLIST attr name NMTOKEN #REQUIRED>
<!ATTLIST attr type ( int | real | double | bool | complex | common | nil) #REQUIRED >
```

The element gxl is the root element of XML document that has a necessary attribute name. It is a name of the analysed program.

The element node represents the method. A name of the calling method is stored in the attribute name. The fact that the method is function or subroutine is determined by the attribute rtype. If the rtype is nil, the method will be subroutine. In all other cases the method is function.

Each element node may contain the elements attr which captures arguments of the described method. The element attr contains mandatory attributes name and type. The attribute name represents a name of the argument of the described method. The attribute type represents a data type of the described argument.

The element edge represents a calling of two subroutines. The attribute from specifies the method which calls the method saved in the attribute to. If the method saved in the attribute from called more than one method then the order of the calling is determined by the attribute order.

4.3 Tool works with GXL

This chapter is a summarized overview of available software tools which works with GXL. GXL was developed to enable the interoperability between software reengineering tools and components, such as code parsers, analysers and visualizers. GXL allows tools especially for parsing, source code extraction, architecture recovery, data flow analysis, source code visualization, object identify, refactoring.

For the purpose of reverse engineering of legacy systems we want tools which would allow: code parsers, data flow analysis and source code visualization.

GXL Code parser for the FORTRAN language is not available. Therefore it is necessary to implement it. This parser will have to follow the DTD schema which is described in the section 3.2.

XIG is an XSLT-based XMI2GXL-Translator. It translates a graph schema defined in the form of UML class diagrams into the internal GXL graph schema format supported by GXL.

Groove is a graph transformation tool set, intended for the simulation and analysis of graph grammars. This involves generating a transition system from a given graph grammar, consisting of all sequences of direct derivations from the grammar‘s initial graph.

FUJUBA is a tool for round-trip software engineering using UML diagrams. It combines UML class diagrams and UML behaviour diagrams to a powerful design and specification language.

4.4 Example

We consider the following source code. We have sets A and B of real numbers. The following program calculates an average, search the maximal and minimal number from the set A union B.

```
      PROGRAM TEST
C
      IMPLICIT NONE
C
      INTEGER ND
      PARAMETER (ND=10)
      REAL A(ND),B(ND),C(ND)
      REAL CSUM,CMAX,CMIN,AVERAGE
C
      DATA  A/1.,2.,3.,4.,5.,6.,7.,8.,9.,10./,B/3*1.,4*2.,3*3./
C
      CALL CPROPS(A,B,C,ND,AVERAGE,CMAX,CMIN)
      CALL WRITEC(C,ND,AVERAGE,CMAX,CMIN)
      STOP
      END

      SUBROUTINE CPROPS( A, B, C, N , CAVG, CMAX, CMIN)
C
      REAL A(*),B(*),C(*)
      REAL CSUM,CMAX,CMIN,CAVG
      INTEGER I,N
C
C     Begin Executable Statements
C
      C=A+B
      CMAX=MAXVAL(C)
      CMIN=MINVAL(C)
      CSUM=SUM(C)
      CAVG=CSUM/N
      RETURN
      END

      SUBROUTINE WRITEC( C, N, CAVG, CMAX, CMIN)
C
      REAL C(*)
      REAL CMAX,CMIN,CAVG
      INTEGER J,N
C
      WRITE(*,*) ' RESULTS FOR FULL C ARRAY'
C
      WRITE(6,2000)CAVG,CMIN,CMAX
 2000 FORMAT(' AVERAGE OF ALL ELEMENTS IN C = ', F8.3,/,
     &       ' MINIMUM OF ALL ELEMENTS IN C = ', F8.3,/,
     &       ' MAXIMUM OF ALL ELEMENTS IN C = ', F8.3)
C
      WRITE(6,2001) (C(J),J=1,N)
 2001 FORMAT(' C = ',1p,/,(8E10.2))
C
      RETURN
      END
```

The UML diagram capturing the call graph of the example:

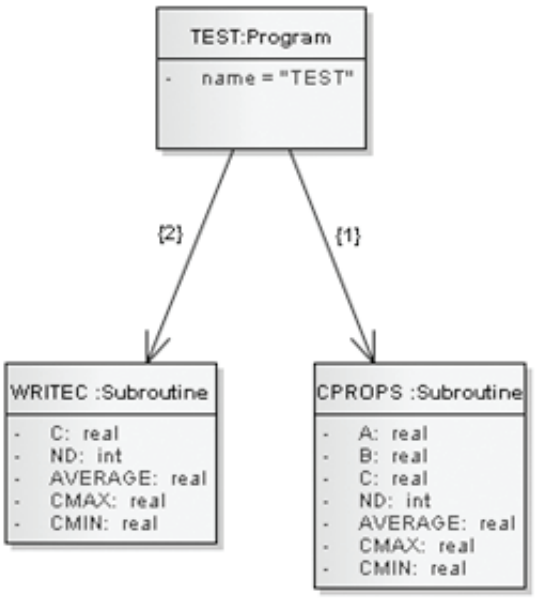

Figure 4. The call graph of the example

The GXL model for the example is follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE gxl SYSTEM "legacyFortranDTD.dtd">
<gxl name="TEST">
    <node name="CPROPS" rtype="nil">
        <attr name="A" type="real" />
        <attr name="B" type="real" />
        <attr name="C" type="real" />
        <attr name="ND" type="int" />
        <attr name="AVERAGE" type="real" />
        <attr name="CMAX" type="real" />
        <attr name="CMIN" type="real" />
    </node>
    <node name="WRITEC" rtype="nil">
        <attr name="C" type="real" />
        <attr name="ND" type="int" />
        <attr name="AVERAGE" type="real" />
        <attr name="CMAX" type="real" />
        <attr name="CMIN" type="real" />
    </node>
    <edge from="TEST" to="CPROPS" order="1" />
    <edge from="TEST" to="WRITEC" order="2" />
</gxl>
```

5 Conclusion and future work

In this paper the author deals with the issue of reversing legacy software. The challenge is to create an appropriate data structure for capturing a structure of legacy systems. More precisely, to create a "call graph" that can be used as an input for software that analyses source codes.

The article summarizes the existing solutions that uses GXL standard. The author has created and described the metamodel of the FORTRAN language. It is extended by the mapping from syntax of FORTRAN into the GXL metamodel that captures a structure of legacy software. In the example in section 3.4 its use is demonstrated.

Next goal is to find a suitable program for processing and visualization a reversed system that is captured by GXL. In the chapter 3.3 the author summarizes a basic overview of programs that work with GXL. Another area of the research is a graph transformation. The graph transformation allows identifying potential objects in a legacy system. These objects can be used as a base to create an object oriented architecture.

References

[1] WATERS, R. G. and CHIKOFSKY, E. 1994. Reverse engineering: progress along many dimensions. In Communications of the ACM, 37, 5, 22-25

[2] CANFORA, G and CIMITILE A, E. 1998. Software Maintenance. In Proc. 7th Int. Conf. Software Engineering and Knowledge Engineering, 478-486

[3] STREIN D., An Extensible Meta-Model for Program Analysis. Omnicore Software, page 1-8, 2006.

[4] HOLT R. C., WINTER A. GXL: Toward a Standard Exchange Format. In Seventh Working Conference on Reverse Engineering. IEEE Computer Society, Los Alamitos. pages 162–171. 2000.

[5] EBERT J., Seminar No. 01041: Interoperability of Reengineering Tools, Schloss Dagstuhl, Germany. pages 21-26. January 2001.

[6] WINTER A. An Overview of the GXL Graph Exchange Language. In Revised Lectures on Software Visualization, London. pages 324-336. 2002.

[7] JÜRGEN E., WINTER A., Graph Based Modeling and Implementation with EER / GRAL. In Proceedings of the 15th International Conference on Conceptual Modeling, London, pages 163-178. 1996.

[8] Roychoudhury S., A Model-Driven Framework for Aspect Weaver Construction. In Tata Research Development and Design Center, Berlin, pages 1-45. 2011.

[9] BALL, T. Software visualization in the large. IEEE Computer, vol. 29, no. 4, page 33-43. 1996.

[10] SNEED H., Object-Oriented COBOL Recycling. Proceedings of the IEEE Working Conference on Reverse Engineering, California, pages 167-178. 1996.

[11] ONG C., Class and Object Extraction from Imperative Code. Journal of Object-Oriented Programming, page 58-68, 1993.

[12] FERRANTE, J., OTTENSTEIN, K., The Program Dependence Graph and its Use in Optimization. In ACM Transactions on Programming Languages and Systems, pages 319-349, 1987.

[13] CHIKOFSKY, E.H. Reverse Engineering and Design Recovery: A taxonomy, IEEE Software Vol. 7, No. 1, IEEE Press, Piscataway, NJ, USA, pp. 13-17.

[14] GRAHAM S., A call graph execution profiler. SIGPLAN Not., New York, pages 120-126, 1982.

[15] HALL W.M., Efficient call graph analysis. ACM Transactions on Programming Languages and Systems, New York, pages 227-242.

[16] HECHT, M.S, Flow Analysis of Computer Program, In Elsevier, Amsterdam, 1997.

Martin Chlumecký, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, Karlovo náměstí 13, 121 35 Praha 2, chlumma1@fel.cvut.cz