Title: ARRANGEMENT AND METHOD FOR EXPLORING SOFTWARE SYSTEMS USING TYPES

Abstract: Computer system (100), including processor (21) and memory (18-20, 22); the memory (18-20, 22) including a software system’s source (201), the software system having a logical structure, the source being defined in a programming language including instructions and variables, the programming language being defined by a grammar for defining a syntactical structure of the instructions; the memory (18-20, 22) further including a first rule-set (203) representing the grammar; the processor (21) being arranged to carry out the following functions: to parse the source (201) into a grammatical sequence by the first rule-set (203); to extract facts (206) from the sequence by comparison with a second rule-set (405) for defining type-definitions and -relations; to infer type-related relationships between the variables from those facts (206); and to execute a query (601) to define a type-related information structure for the software system.
For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.
Arrangement and method for exploring software systems using types

Field of the invention

The present invention relates to a computer system, comprising processing means and memory means connected to said processing means; said memory means comprising a first data file representing a source file of a software system, said software system having a logical structure defining said software system’s functionality, said source file being defined in a programming language comprising a set of instructions, said source file further comprising a set of variables, said programming language being defined by a grammar, said grammar comprising a set of grammatical elements; said grammatical elements defining a syntactical structure of said set of instructions; said memory means further comprising a second data file representing a first rule-set of said grammar of said programming language; the processing means being arranged to carry out the following functions:

- in a first step, to parse said source contained in said first data file into a logically equivalent sequence of grammatical elements by using said first rule-set;
- in a second step, to extract a set of facts from said logically equivalent sequence of grammatical elements by comparing said logically equivalent sequence with a second rule-set which comprises rules for defining type definitions and type relations;
- in a third step to derive and to infer type-related relationships between variables in said set of variables from said set of facts.

Prior art

Software immigrants, employees that are added to an existing software project (a software system) in order to conduct maintenance or development, are faced with the difficult task of understanding an existing software system. This problem is discussed in the article “The ramp-up problem in software projects: A case study of how software immigrants naturalize” by S.E. Sim and R.C. Holt in the proceedings of the 20th Int. Conf. on Software Engineering; ICSE-97, pages 361-370, ACM, 1998. Even the original developers of a system generally have a hard time understanding their own
code as time between development and maintenance goes by. As a consequence, maintenance tasks become difficult, expensive, and error prone.

To reduce these problems, a large research effort is being made to develop tools to assist in program understanding. One line of research focuses on the use of hypertext for program comprehension purposes. Within a hypertext, various layers of abstraction can be integrated, ranging from the system's architecture to the individual statements in the source code. The maintenance engineer can navigate easily between these, using the well-known strategies of top-down and bottom-up program comprehension, as well as the "opportunistic" combination of these.

Such a hypertext can be seen as a (special form of) system documentation. Part of it will be hand-written, especially those sections dealing with domain-specific issues or the system's requirements. However, documentation at the more technical level should be generated automatically whenever possible, in order to keep it up to date and consistent with the software system's sources at all times.

The fundamental problem with documentation generation (and in fact, the key challenge of reverse engineering) is to arrive at non-trivial levels of abstraction, going beyond just cross-referencing information and source code browsing.

For (strongly) typed languages, such as Java, C, and Pascal, using types for program comprehension is relatively straightforward: types are explicit, and can help to determine interfaces, function signatures, permitted values for certain variables, etc. Many of the existing software systems, however, are written in older languages with very weak type systems. In particular, COBOL, the language in which at least 30% of the world's software is written, does not offer the possibility of type definitions, which makes the understanding of such COBOL systems a cumbersome process.


In the article "CHIME: Customisable hyperlink insertion and maintenance engine for software engineering environments", in the proceedings of 21st Int. Conf. on Software Engineering, ICSE-99, pages 473-482, ACM, 1999, P. Devanbu, Y-F. Chen, E. Gansner, H. Müller, and J. Martin present CHIME which is a generator of tools that automatically insert certain links in source code elements.
PAS is a system that can be used to incrementally add partitioned annotations of software, which is discussed by V. Rajlich and S. Varadarajan, “Using the web for software annotations”, Int. Journal of Software Engineering and Knowledge Engineering, 9(1):55-72, 1999.


In “Building documentation generators”, proceedings of the International Conference on Software Maintenance, ICSM'99, pages 40-49. IEEE Computer Society, 1999, A. van Deursen and T. Kuipers present DocGen, a tool for generating hyperlinked visual and textual documentation from COBOL and batch job sources. Distinguishing characteristics of DocGen include extraction based on island grammars rather than full parsing, emphasis on industrial application, and integration of various abstraction layers, ranging from source code up to system architecture.

Many architecture extraction tools (such as Rigi, PBS, Dali, and also DocGen) adopt the extract-query-view approach, i.e., extracting facts from sources, querying a database filled with facts, and presenting these facts in various ways, for example using hypertext.


PBS is presented in the article “Browsing and searching software architectures” by S.E. Sim, C.L.A. Clarke, R.C. Holt, and A. M. Cox in the proceedings of Int. Conf. on Software Maintenance, ICSM’99, pages 381-390, IEEE Computer Society, 1999. PBS, which has been applied mostly to analyse C systems such as Linux, uses Tarski relational algebra for querying.

Dali, published in the article “Playing detective: Reconstructing software architecture from available evidence” by R. Kazman and J. Carrière in Automated Software Engineering, 6:107-138, 1999, emphasizes the need for an open tool set, in which many different tools can be plugged in, when necessary.
Closest in aims to the integration of type analysis and program understanding is Lackwit, a tool for analyzing C programs using type inferencing. This tool is presented in the article “Lackwit: A program understanding tool based on type inference” by R. O'Callahan and D. Jackson, in the proceedings of the 19th International Conference on Software Engineering; ICSE-97, ACM, 1997. Lackwit allows one to ask queries like: “Which functions could directly access the representation of component X of variable Y?”

Other work based on type inferencing includes “physical type checking of C”, a stronger form of type checking for type casts involving pointers to structures, which is discussed by S. Chandra and T. Reps in the article “Physical type checking for C” in the proceedings of the Workshop on Program Analysis for Software Tools and Engineering, PASTE'99, pages 66-75, ACM Press, September 1999, SIGSOFT Software Engineering Notes 24(5).

The analysis of Fortran programs in order to find new type signatures for subroutines is discussed in the paper by N. Williams-Preston, “New type signatures for legacy Fortran subroutines” in the proceedings of the Workshop on Program Analysis for Software Tools and Engineering, PASTE'99, pages 76-85, ACM Press, September 1999, SIGSOFT Software Engineering Notes 24(5).

Type-based analysis of COBOL, for the purpose of year 2000 analysis, is presented in the article “Anno Domini: From type theory to Year 2000 conversion tool” by P.H. Eidorff, F. Henglein, C. Mossin, H. Niss, M.H. Sorensen, and M. Tofte in the proceedings of the 26th Symposium on Principles of Progr. Languages, POPL'99, ACM, 1999, and in the article “Aggregate structure identification and its application to program analysis” by G. Ramalingam, J. Field, and F. Tip, *ibid.:* both provide a type inferencing algorithm that splits aggregate structures into smaller units based on assignments between records that cross (record) field boundaries.


In the article “Understanding COBOL systems using types” in the proceedings of the 7th Int. Workshop on Program Comprehension, IWPC'99, pages 74-83, IEEE Computer Society, 1999, A. van Deursen and L. Moonen described an implementation
of COBOL type inferencing using Tarski relational algebra. Also, a detailed assessment of the benefits of using subtyping to deal with the problem of pollution (i.e., inferring too many type equivalences) is presented in this reference.

COBOL programs consist of a procedure division, containing the executable statements, and a data division, containing declarations for all variables used.

From the perspective of types, COBOL variable declarations suffer from a number of problems. First of all, it is not possible to separate type definitions from variable declarations. Consequently, when two variables for the same record structure are needed, the full record construction needs to be repeated. (In principle the COPY mechanism of COBOL for file inclusion can be used to avoid code duplication here, but in many practical cases, this mechanism is not used.) This not only increases the chances of inconsistencies, it also makes it harder to understand the program, as the maintainer has to check and compare all record fields in order to decide that two records indeed have the same structure.

Furthermore, the absence of type definitions makes it difficult to group variables that are intended to represent the same kind of entities. Clearly, all such variables will share the same physical representation. Unfortunately, the converse does not hold: One cannot conclude that whenever two variables share the same byte representation, they must represent the same kind of entity.

Besides these problems regarding type definitions, COBOL only has limited means to indicate the allowed set of values for a variable (i.e., there are no ranges or enumeration types). Moreover, COBOL uses sections or paragraphs to represent procedures. Neither sections nor paragraphs can have formal parameters, forcing the programmer to use global variables for parameter passing.

In summary, in weakly typed programming languages the information on variable types is hardly available. Moreover, since the restrictions on types are so minimal, usage of variables and types in a software system may have a high complexity and even be cluttered (disordered). Therefore, understanding of such software systems is difficult and error-prone.

Summary of the invention

To overcome the disadvantages of the prior art in programming understanding of weakly typed programming languages, it is an object of the present invention to provide an arrangement and a method for exploring a software system written in a
weakly typed programming language by using types which are inferred from the software system.

The present invention relates to a computer system, comprising processing means and memory means connected to said processing means; said memory means comprising a first data file representing a source file of a software system, said software system having a logical structure defining said software system's functionality, said source file being defined in a programming language comprising a set of instructions, said source file further comprising a set of variables, said programming language being defined by a grammar, said grammar comprising a set of grammatical elements; said grammatical elements defining a syntactical structure of said set of instructions; said memory means further comprising a second data file representing a first rule-set of said grammar of said programming language; the processing means being arranged to carry out the following functions:

- in a first step, to parse said source contained in said first data file into a logically equivalent sequence of grammatical elements by using said first rule-set;
- in a second step, to extract a set of facts from said logically equivalent sequence of grammatical elements by comparing said logically equivalent sequence with a second rule-set which comprises rules for defining type definitions and type relations;
- in a third step to derive and to infer type-related relationships between variables in said set of variables from said set of facts characterised in that the processing means execute a querying procedure to define a type-related information structure for said software system.

Moreover, the present invention relates to a computer system as described above, characterised in that the processing means are further arranged to carry out the following function:

to generate as type-related information for said structure of said software system for each derived and inferred type-related relationship at one least one item selected from a list of items comprising at least:

- a byte representation;
- an enumeration range;
- usage links in said source of said software system;
7

- links to records;
- links to programs;
- links to copybooks;
- a type name;

- a representation structure for visualisation of said types, said subtypes and said supertypes.

Furthermore, the present invention relates to a computer system as described above, characterised in that said query procedure provides rules for abstraction during extraction of said type-related information.

Also, the present invention relates to a computer system as described above, characterised in that said type-related relationships are being defined as equivalences, subtypes and supertypes.

In addition, the present invention relates to a computer system as described above, characterised in that the processing means are further arranged to carry out the following function:

- to merge said equivalences and said subtypes of a type-related relationship between variables in said set of variables into a type-cluster.

The present invention also relates to a computer system as described above, characterised in that the processing means are further arranged to carry out the following function:

- to format said type-related information as hypertext or graph, and
- to present the result in an on-line or off-line mode to a user.

Also, the present invention relates to a computer system as described above, characterised in that the processing means are further arranged to carry out the following function:

- to display said type-relationships visually as a graph displaying said subtype and supertype dependencies between displayed types, and displaying assignments made to variables of said displayed types.

Moreover, the present invention relates to a computer system as described above, characterised in that said type-related information structure for said software system comprises navigational information for navigating said structure of said software system.
Further, the present invention relates to a computer system as described above, characterised in that the processing means are arranged to carry out the following function:

- to add said type-related information for said structure of said software system into documentation generated for said structure of said software system by a documentation generation system.

Furthermore, the present invention relates to a computer system according to any of the preceding claims, characterised in that the processing means are further arranged to carry out the following function:

- to modify documentation generated for said structure of said software system such that type-related information is added to said documentation, said type-related information containing at least one item from a list of items comprising at least:
  - a type signature for programs,
  - a type for columns occurring in database tables,
  - a type for records used for flat files,
  - a type for data entered through on-line screens.

In addition, the present invention relates to a computer system according to any of the preceding claims, characterised in that the processing means are further arranged to carry out the following function:

- to modify documentation generated for said structure of said software system such that type-related dependencies between system elements are added to said documentation, said type-related dependencies containing at least one item from a list of items comprising at least the following items:
  - copybooks,
  - programs,
  - tables,
  - columns,
  - flat files,
  - screens.

Also, the present invention relates to a computer system as described above, characterised in that said type-related information is transmitted over a network.

Moreover, the present invention relates to a method to be carried out by a computer system, comprising processing means and memory means connected to said processing
means; said memory means comprising a first data file representing a source file of a software system, said software system having a logical structure defining said software system’s functionality, said source file being defined in a programming language comprising a set of instructions, said source file further comprising a set of variables, said programming language being defined by a grammar, said grammar comprising a set of grammatical elements; said grammatical elements defining a syntactical structure of said set of instructions; said memory means further comprising a second data file representing a first rule-set of said grammar of said programming language; said method comprising the following functions:

- to parse said source contained in said first data file into a logically equivalent sequence of grammatical elements by using said first rule-set;
- to extract a set of facts from said logically equivalent sequence of grammatical elements by comparing said logically equivalent sequence with a second rule-set which comprises rules for defining type definitions and type relations;
- to derive and to infer type-related relationships between variables in said set of variables from said set of facts;

characterised in that the method comprises a querying procedure to extract from said type-related relationships between variables in said set of variables type-related information for said logical structure of said software system.

Also, the present invention relates to a method to be carried out by a computer system as described above, characterised in that said type-related information for said logical structure of said software system for each derived and inferred type-related relationship comprises a list of items comprising at least:

- a byte representation;
- an enumeration range;
- usage links in said source of said software system;
- links to records;
- links to programs;
- links to copybooks;
- a type name;
- a representation structure for visualisation of said types, said subtypes and said supertypes.
Also, the present invention relates to a method to be carried out by a computer system as described above, characterised in that the method provides rules in said query procedure for abstraction during said extraction of said type-related information.

Moreover, the present invention relates to a method to be carried out by a computer system, as described above, characterised in that said type-related relationships are being defined as equivalences, subtypes, and supertypes.

In addition, the present invention relates to a method to be carried out by a computer system, as described above, characterised in that the method further comprises the following function:

- to merge said equivalences and said subtypes of a type-related relationship between variables in said set of variables into a type-cluster.

Furthermore, the present invention relates to a method to be carried out by a computer system, as described above, characterised in that the method further comprises the following function:

- to format said type-related information as hypertext or graph, and
- to present the result in an on-line or off-line mode to a user.

In addition, the present invention relates to a method as described above, characterised in that the method further comprises the following function:

- to display said type-relationships visually as a graph displaying said subtype and supertype dependencies between displayed types, and displaying assignments made to variables of said displayed types.

Also, the present invention relates to a method to be carried out by a computer system as described above, characterised in that said type-related information structure for said software system comprises navigational information for navigating said structure of said software system.

Moreover, the present invention relates to a method to be carried out by a computer system, as described above, characterised in that the method further comprises the following function:

- to add said type-related information for said logical structure of said software system into a document generated for said logical structure of said software system by a documentation generation system.

In addition, the present invention relates to a method as described above, characterised in that the method further comprises the following function:
to modify documentation generated for said structure of said software system such that type-related information is added to said documentation, said type-related information containing at least one item from a list of items comprising at least:
  - a type signature for programs,
  - a type for columns occurring in database tables,
  - a type for records used for flat files,
  - a type for data entered through on-line screens.

Furthermore, the present invention relates to a method as described above, characterised in that the method further comprises the following function:

- to modify documentation generated for said structure of said software system such that type-related dependencies between system elements are added to said documentation, said type-related dependencies containing at least one item from a list of items comprising at least the following items:
  - copybooks,
  - programs,
  - tables,
  - columns,
  - flat files,
  - screens.

Also, the method of the present invention encompasses the following functionalities. In the present invention, the method to be carried out by a computer system, as described above, is characterised in that the method provides information for maintenance and/or impact analysis of said software system.

Further, the method to be carried out by a computer system, as described above, is characterised in that the method provides information for re-engineering of said software system.

In addition, the method to be carried out by a computer system, as described above, is characterised in that the method provides information for quality assessment of said software system.

Furthermore, the method to be carried out by a computer system, as described above, is characterised in that said type-related information is transmitted over a network.
Also, the present invention relates to a computer program product to be loaded by a computer system, comprising processing means and memory means connected to said processing means; said memory means comprising a first data file representing a source file of a software system, said software system having a logical structure defining said software system's functionality, said source file being defined in a programming language comprising a set of instructions, said source file further comprising a set of variables, said programming language being defined by a grammar, said grammar comprising a set of grammatical elements; said grammatical elements defining a syntactical structure of said set of instructions; said memory means further comprising a second data file representing a first rule-set of said grammar of said programming language; and allowing said computer system to carry out the following functions:

- in a first step, to parse said source contained in said first data file into a logically equivalent sequence of grammatical elements by using said first rule-set;
- in a second step, to extract a set of facts from said logically equivalent sequence of grammatical elements by comparing said logically equivalent sequence with a second rule-set which comprises rules for defining type definitions and type relations;
- in a third step, to derive and to infer type-related relationships between variables in said set of variables from said set of facts;

characterised by the following additional function:

- to execute a querying procedure to define a type-related information structure for said software system.

The present invention also relates to a data carrier provided with a computer program product as described above.

In the present invention, a tool set is disclosed that infers types for COBOL software systems automatically, based on an analysis of the use of variables. This results in types for variables, program parameters, database records, literal values, and so on, which can be used to understand the relationships between programs, copybooks, databases, screens, and so on. Moreover, the present invention addresses the problems involved in integrating inferred types into hypertext-based program understanding tools.
Brief description of diagrams

Below, the invention will be explained with reference to some drawings, which are intended for illustration purposes only and not to limit the scope of protection as defined in the accompanying claims.

Figure 1 shows a general overview of a computer arrangement to illustrate the invention;

Figure 2 shows a schematic block diagram of a system for exploring software systems using types according to this invention;

Figure 3 shows a schematic block diagram of an extraction procedure of the present invention;

Figure 4 shows a schematic block diagram of a type inferencing procedure of the present invention;

Figure 5 shows a schematic block diagram of a querying and presentation procedure, for type information of a software system, as generated according to the present invention;

Figure 6 shows a schematic block diagram of a graphical presentation of type relations within a software system.

Description of preferred embodiment

Figure 1 shows a general overview of a computer system 100 to illustrate the invention, comprising host processor means 21 with peripherals. The host processor means 21 are connected to memory units 18, 19, 22, 23, 24 which store instructions and data, one or more reading units 30 (to read, e.g., floppy disks 17, CD ROM's 20, DVD's, etc.), a keyboard 26 and a mouse 27 as input devices, and as output devices, a monitor 28 and a printer 29. Other input devices, like a trackball and a touchscreen, as well as other output devices may be provided. An input/output (I/O) device 7 is provided for data-communication over a network 1. The I/O device 7 is linked to the network 1. The network 1 may comprise a plurality of interconnected networks, that may be the Public Switched Telephone Network (PSTN), or any other network suitable for data transmission. For instance such an interconnected network may be a Local Area Network (LAN), or a Wide Area Network (WAN). On the network 1, other computer systems (not shown) may be connected in a similar way as computer system 100.
The memory units shown comprise RAM 22, (E)EPROM 23, ROM 24, tape unit 19, and hard disk 18. However, it should be understood that there may be provided more and/or other memory units known to persons skilled in the art. Moreover, one or more of them may be physically located remote from the processor means 21, if required. The processor means 21 are shown as one box, however, they may comprise several processing units functioning in parallel or controlled by one main processor, that may be located remotely from one another, as is known to persons skilled in the art.

Figure 2 shows in a schematic block diagram the architecture of a tool set for a type information explorer system 200 to be carried out by a computer system 100, in accordance with the present invention.

The type information explorer system 200 analyses a software system. The software system is represented by its source code 201. The source code 201 is stored in the memory means 18-20, 22 of the computer system, preferably hard disk 18.

In a first step 202, the processing means 21 execute a parser procedure to parse the source code 201. The parser procedure uses a rule-set containing the grammar of the programming language (e.g., COBOL) 203. The rule-set is stored in the memory means 18-20, 22 of the computer system. The parser procedure 202 produces a collection of abstract syntax trees (ASTs). The ASTs are stored in a data file 204, preferably on hard disk 18.

Subsequently, the results of the parsing procedure (i.e., the ASTs) are passed on to a second step 205. In this step 205, the processing means 21 run a fact extraction procedure to analyse the results from the parsing procedure with regard to type-related facts, and stores the results of the analysis into a database 206 which contains the observed type-facts. The database 206 is stored in the memory means 18-20, 22 of the computer system, preferably hard disk 18. This second step 205 will be explained in more detail in Figure 3.

In a third step 207, the processing means 21 run a type inferencing procedure to process the content of the database 206, i.e., the observed type-facts, to combine and to abstract the observed type-facts. The result of the type inferencing procedure is additionally stored in the database 206. Thus, the inferred type information is stored together with the observed type information. As is known to persons skilled in the art,
the inferred information, may also be stored in a separate relational database (not shown).

The database 206 provides information on the software system at various levels of abstraction. This third step 207 will be explained in more detail in Figure 4.

In a fourth step 209, the processing means 21 execute a querying and presentation procedure to process the data in the database 206 in order to generate type-related information. Subsequently, the processing means 21 run a formatting tool to generate documentation 210 in a user-readable format at the various possible levels of abstraction. The formatting tool generates documentation 210 in a hypertext format, which advantageously enhances the accessibility of the documentation, by providing the capability to browse and to navigate through the documentation, in textual and graphical modes. The documentation 210 may be available to a user in either an on-line or off-line mode. In the on-line mode, the procedures of step 209, can be accessed directly and dynamically, on user demand, thus allowing users to define their own specific queries and visualisations of the results. In the off-line mode, the procedures of step 209, have been executed in order to generate documentation based on an automated querying procedure. The results of step 209 in the off-line mode are collected in a static database, preferably on hard disk 18 or CD-ROM 20. Due to the static nature of the documentation, some limitations in accessibility may be presented to a user. This fourth step 209 will be explained in more detail in Figure 5.

Figure 3 shows a schematic block diagram of a fact extraction procedure 205 to be carried out by a computer system 100. In Figure 3 the fact extraction procedure 205 is shown in more detail.

In a first step 401, the processing means 21 read the data file 204, which contains the ASTs generated in the parser procedure 202.

In step 402, the processing means 21 convert the ASTs into a form, known in the art as an object. As known to persons skilled in the art, this conversion may be to any usable format. Preferably, the object used here is a Java object, which relates to the programming language Java. It is noted that the functionality of such an Java object may also be obtained by alternatively programmed representations as known in the art.

In step 403, the object, containing the ASTs, is analysed by a procedure which is defined such that it represents a specific knowledge, depicted by database 405, which contains "a constructor filter": i.e., the relevant language constructs with regard to type
inferencing as described in the preceding paragraphs (e.g., variable declarations, assignments, relational expressions, call statements, etc.). In this example to illustrate the present invention, the analysis procedure is a Java program which traverses the AST using the knowledge from the constructor filter. (It is to be noted that the Java program can use a separate database 405, but, alternatively, it may use relevant information coded within itself). By defining which specific occurrence of facts is to be detected, the comparison can be tailored to the particular requirements for the analysis.

In the subsequent step 404, the processing means 21 store the matching type-related facts that are found in step 403 (i.e., relevant to its specific analysis), in the database 206.

The procedure ends in step 406.

In the analysis procedure as shown in Figure 3, the specific knowledge for type-related facts, depicted by the constructor filter in database 405, has been derived with reference to syntactical constructs that, for example, are presented in the aforementioned article “Type inference for COBOL systems” by A. van Deursen and L. Moonen.

In the procedure shown in Figure 3, the pattern matching was done using an object and a program written in Java code. As known to persons skilled in the art, an object form and program written in any other programming language may be used as well.

Figure 4 shows a schematic block diagram of a type inferencing procedure 207 to be carried out by computer system 100.

In a first step 501, the processing means 21 retrieve the observed type-related facts from the database 206.

In step 502, the processing means 21 combine the observed type-related facts from the database 206 and, subsequently, infer a number of conclusions regarding type relations. As an example, one of the tools that can be used for inferring type relations is “grok”, a calculator for Tarski relational algebra, as published by R. Holt in the article “Structural manipulations of software architecture using Tarski relational algebra”, the proceedings of the 5th Working Conference on Reverse Engineering, WCRE’98, pages 210-219, IEEE Computer Society, 1998. The processing means 21 process the facts from database 206 using (e.g., Tarski) relational algebra operators for relational composition, for computing the transitive closure of a relation, for computing the
difference between two relations, and so on. This type of algebra is used, for example, to turn the inferred type facts into the required equivalence relation.

Finally, in step 503, the processing means 21 store the inferred facts in the relational database 206 (together with the observed facts). The database 206 is an SQL (Structured Query Language) database, as published by e.g., MySQL.org on http://www.mysql.org/.

For the present invention, type inferencing can be considered as a black box, which analyses a Cobol system and computes types for source code elements present in the Cobol system. As explained above, COBOL by itself is lacking type definitions for such code elements. The details of type inferencing are presented in the article "Type Inference for COBOL Systems" in the proceedings of the fifth Working Conference on Reverse Engineering, WCRE'98, pages 220-230, IEEE Computer Society, 1998.

The effect of type inferencing can be illustrated by an example, using the Cobol fragments listed below:

```
1 / variables containing business data.
2 01 PERSON-RECORD.
3   03 INITIALS PIC X(05).
4   03 NAME PIC X(27).
5   03 STREET PIC X(18).
6 ...
7
8 / variables containing char array of length 40,
9 / as well as several counters.
10 01 TAB000.
11 03 A00-NAME-PART.
12     05 A00-POS PIC X(01) OCCURS 40.
13     03 A00-MAX PIC S9(03) COMP-3 VALUE 40.
14     03 A00-FILLED PIC S9(03) COMP-3 VALUE ZERO.
15
16 ...
17
18 / other counters declared elsewhere.
19 01 N000.
20   03 N100 PIC S9(03) COMP-3 VALUE ZERO.
21   03 N200 PIC S9(03) COMP-3 VALUE ZERO.
22 ...
23
24 / procedure dealing with initials.
```
R210-INITIALS SECTION.
MOVE INITIALS TO A00-NAME-PART.
PERFORM R300-COMPOSE-NAME.

/ procedure dealing with last names.
R230-NAME SECTION.
MOVE NAME TO A00-NAME-PART.
PERFORM R300-COMPOSE-NAME.

/ procedure for computing a result based on the
/ value of the A00-NAME-PART.
/ Uses A00-FILLED, A00-MAX, and N100 for array indexing.
R300-COMPOSE-NAME SECTION.
...

PERFORM UNTIL N100 > A00-MAX
...
IF A00-FILLED = N100
...

The first 23 lines contain variable declarations, coming from a Cobol data division. Variable declarations in COBOL indicate the memory layout of variable, i.e., how many bytes they occupy. Variable declarations are not needed for type inferencing. Lines 24 until 42 contain the program's statements, coming from a Cobol procedure division. Lines starting with a "/" are comment lines.

Type inferencing analyses the use of variables and literal values such as strings and numbers in expressions and statements as occurring in the procedure division. From this, it invents new types for each variable, literal, expression, and so on.

Going from bottom to top in the example, in line 41 variable A00-FILLED is compared to N100. From this, it is concluded that A00-FILLED and N100 must have the same type. Likewise, from line 39, it is inferred that N100 and A00-MAX must have the same type. Combining this requirement with the earlier requirement (from line 41) that N100 and A00-FILLED have the same type, it follows that A00-MAX and A00-FILLED must also belong to the same type. This yields one type containing three different variables: N100, A00-MAX, and A00-FILLED.

Comparing this with the data division, one can observe that this makes sense: the declared picture layouts of these variables (in lines 13, 14, and 20) is indeed the same, which are all numeric data elements. However, it is impossible to infer such
equivalences from just the pictures, as entirely unrelated data structures may share the same physical layout (for example, N200 in line 21).

An assignment example is given in line 31, where variable NAME is assigned to NAME-PART. Here it is inferred that the type of NAME is a subtype of NAME-PART, i.e., NAME-PART can contain at least all the values that NAME can hold. In line 26, another variable, INITIALS, is assigned to NAME-PART as well, giving rise to a second subtype relationship, now between INITIALS and NAME-PART. In this way, INITIALS and NAME share a common supertype (NAME-PART), but there is no direct relationship inferred between them. Looking at the declared physical layout one can see that all three are strings of a different length (in lines 3, 4, and 12). NAME-PART is the largest, capable of accepting values from both INITIALS and NAME.

In fact, NAME-PART is a global variable acting as a formal parameter for the procedure R300-COMPOSE-NAME (COBOL does not support the declaration of parameters for procedures). What is inferred is that the type of the actual parameter is a subtype of the formal parameter. Just deriving equivalences from assignments would lead to so-called pollution: it would give all the actual parameters, in this case the two different concepts “initials” and “first name”, the same type.

The above example only contains the simplest form of type inferencing: finding types for variables based on assignments and comparisons. Additional type relationships can be inferred from:

- arithmetic expressions,
- array indexes,
- assignments or comparisons between structured data such as records,
- redefine clauses,
- database operations,
- file operations,
- copybook inclusions,
- parameter passing in CALL statements.

Observe that the last four result in type relations that have a system-wide scope, i.e., they are not restricted to one program (module). The details of these inferencing steps are covered in "Type Inference for COBOL Systems" in the proceedings of the fifth Working Conference on Reverse Engineering, WCRE'98, pages 220-230, IEEE Computer Society, 1998.
20

Figure 5 shows a schematic block diagram of presentation procedure 209, for type information of a software system, as generated according to the present invention.

In step 601 of the presentation procedure 209, the processing means 21 perform a query on the relational database 206.

In step 602, a documentation procedure, the processing means 21 generate hypertext documentation from the results of the querying procedure. In the on-line version, the hypertext documentation 210 is sent as displayable code to the browser application requesting the information from the querying and presentation procedure 209. In the off-line version, the hypertext documentation 210 is stored on suitable storage media such as hard disk, floppy disk, or CD-ROM.

The procedure ends in step 603.

The querying procedure 601 is capable of retrieving information at various levels of abstraction. Depending on the required level of abstraction, the querying procedure may focus on different aspects of the analysis. To illustrate the present invention, as an example features of a querying procedure for a software system written in COBOL will be shortly discussed.

First, it is noted that in the procedure 209 a subtyping analysis is included to avoid type pollution. In some cases, though, there would be no pollution even if plain equivalences between types would be used. It could even be argued that using subtyping in those cases obscures understanding since it creates additional levels of indirection between types that would otherwise be considered equivalent. Thus, the problem may occur that for some types subtyping is necessary to avoid pollution, whereas for other types subtyping should actually have been type equivalence. In the present invention this problem is solved by including an additional abstraction layer, the type cluster, in the derived hypertext. The contents of a type cluster are collected by querying the relational database 206. A type cluster consists of all types that have an equivalence or subtype relation to each other (effectively regarding the subtyping relation as an equivalence relation). During the exploration of the hypertext-based information at a later stage, a user, who is not interested in the subtyping details of a particular type, can move up to the type cluster level.

Further, in the querying procedure 601, it is needed to determine names for the observed equivalence classes in the source code. Preferably, the name for a type should be descriptive. Therefore, the processing means 21 execute a process in which an effort
is made to distil meaningful names from the variable names involved, by determining
the words occurring in them. Such words can be found by splitting the variable names
based on special characters (\-, \_, etc.) or lexical properties (e.g., caseChange). The
actual splitting should be a parameter of the analysis since it is influenced by the
particular coding style that is used in a system. Candidate names of a given type can
then be based on the frequency of words that occur in names of variable of that type.
Since these names should be as descriptive as possible, one also needs to consider all
combinations of words that occur in variable names. As an example, for a variable
called A00-NAME-PART, not only the words NAME and PART may be relevant for a
user, but also the word NAME-PART.

For programs, it is possible in some cases to derive a textual description
explaining their behaviour based on a comment prologue. Since types are abstractions
that are not directly present in one particular place in the source code, it is not possible
to find meaningful texts explaining types automatically. Therefore, the querying
procedure 601 offers the option to maintainers to add annotations by hand. In practice,
such a feature will be used mostly for types that play a significant role in the system.
Furthermore, there can be a special annotation allowing a maintainer to improve the
name given to a type. In the on-line version, annotations can be added on the fly, and
have immediate effect; in the off-line mode annotations are incorporated after
regeneration of the hypertext documentation.

In the querying procedure 601, the processing means 21 compile type information
on various levels. In the following table, the information presented for a particular type
is listed by type element in combination with the information generated, relevant to
each element.
<table>
<thead>
<tr>
<th>Element</th>
<th>Available information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>Hand-written description of the type</td>
</tr>
<tr>
<td>Byte representation</td>
<td>The picture or record declaration(s) for variables of the type</td>
</tr>
<tr>
<td>Values</td>
<td>All literal values found for the type</td>
</tr>
<tr>
<td>Usage</td>
<td>Links to source code lines where a variable of the type is used</td>
</tr>
<tr>
<td>Parents</td>
<td>Links to records with fields of the type</td>
</tr>
<tr>
<td>Programs</td>
<td>Links to programs that use the type</td>
</tr>
<tr>
<td>Copybooks</td>
<td>Links to copybooks that use the type</td>
</tr>
<tr>
<td>Words</td>
<td>List of domain concepts extracted from names of variables of the type (heuristics based)</td>
</tr>
<tr>
<td>Type name</td>
<td>Suggestion for a name of the type based on these domain concepts</td>
</tr>
<tr>
<td>Type graphs</td>
<td>Visualisation of subtypes and supertypes of the type</td>
</tr>
</tbody>
</table>

The items listed in this table are only shown as examples, other items may be conceivable depending on the specific analysis. It is also to be understood that for other programming languages similar lists can be compiled, although the actual items and information levels may differ.

The information on a type as listed in the table above, contains a plurality of elements, which are discussed briefly in the following.

The declared COBOL pictures of primitive types provide information about the bytes occupied and the intended use (number, character, ...). In most cases, all primitive types in an equivalence class will have the same picture. If the pictures of such types are different, this means that the COBOL code using variables of this type relies on coercion, which may indicate bad programming style or potential programming errors.

If the primitive types of a type are all records, the most common case is that all variables in this type are declared with the same number of fields, each of the same length. In this case, the type inferencing procedure 207 applies a rule of substructure completion which will infer equivalences between these field types. If the field types are of different shape, aggregate structure identification techniques in the type inferencing procedure 207 may be used to find subfields that are small enough to unify the various records in the type. Thus, although the primitive records in the type may be of different shape, one record type is inferred with the smallest necessary fields for the type.
The inferred literals provide information about the sort of values that are permitted for this type. Moreover, they show which literal values are actually used in the system analysed. Since a supertype of the type can hold at least the values of all its subtypes, the literals are also listed in all subtypes of the type.

In addition to structural information about a type, the querying procedure 601 provides data on its usage. The querying procedure includes links to source code lines in which a variable of the type is used, as well to those lines in which a literal of the type is used. Moreover, the querying procedure 601 includes links to the documentation of all programs and copybooks that use the type.

For types used as fields in other records, the querying procedure 601 includes a link to each of the parent records.

An inferred type can be related to other types via subtype (or supertype) relationships. As part of the documentation generated for a type, in the querying procedure 601 the capability is implemented to set up a type graph, i.e., an information structure which can be used by the presentation procedure to display graphically all sub- and supertypes of the inferred type. An inferred type can be related to other types via subtype (or supertype) relationships. As part of the documentation generated for a type, all sub- and supertypes of that given type can be displayed in a type graph.

Figure 6 shows a schematic block diagram of a graphical presentation of type relations within a software system. As an example a type graph of a part of an accounting software system is shown.

The nodes in the graph are types: the text in a node is the name chosen for a type. This name is obtained by picking one of its primitive types as representative. Navigating the structure is possible: clicking on the nodes brings up the page for the type clicked on. The particular type to be analysed is shown in a ellipse. In Figure 6 it has the name “ibq007.feature” 700. In Figure 6 arrows pointing from a node to another node indicate that for each arrow the former node is a subtype of the latter node.

A number of observations can be made from this graph. First of all, the subtype relationship on types closely corresponds to the assignment relationship between variables. Thus, one can read an arrow, indicating a relation between a node and another node, also as: for example, variables of type “cop800.payment” 710 are assigned to variables of type “fib35.payment-old” 707.
Second, within the graph, one can recognise groups of related types: in Figure 6, examples are the three kind types on the right (705, 709, 711), or the four types in the middle (703, 706, 707, 710).

Third, the type selected, “ibq007.feature” 700, happens to be a supertype of several other types. Thus, “ibq007.feature” 700 can accept values of several different subtypes, dealing with various sorts of numbers, such as, for example, country codes, title codes, etc. Such a type with several different subtypes is typically the input parameter of a procedure or program, where each incoming edge corresponds to the subtype of an actual parameter. If no subtypes would be inferred, but equivalences instead, all these types would become the same (via “ibq007.feature”).

Fourth, some types have dashed outgoing (or incoming) arrows. This means that these types have other supertypes (subtypes), which are, however, not sub- or supertypes of the type selected for analysis, “ibq007.feature”. An example is the left most salutation type 701. Its outgoing arrow to “ibq007.feature” means that salutations are moved to features: its dashed outgoing arrow means that salutations are moved elsewhere as well.

Fifth, the type “cop603.num” 712 only has outgoing arrows. This typically means that “cop603.num” 712 is the output parameter of procedure or section. Furthermore, the fact that “cop603.num” 712 has no incoming edges means that there are no assignments from other types into “cop603.num” 712. This can mean one of three things for variables of type “cop603.num”:
1. They never get a value within the programs analysed, but only in external libraries.
2. They do get a value, but only from variables also of type “cop603.num”.
3. They do get a value, yet not as a scalar value, but viewed as an aggregate. This, is in fact the case for type “cop603.num”, which is filled as an array, digit by digit.

In short, type graphs can be used to show a number of interesting properties regarding types and variables. For the case studies conducted, most of the type graphs are reasonably small and understandable. The dashed arrows are an important tool to keep them small: If all dashed arrows would be expanded transitively, the type graph for “ibq007.feature” would become several hundreds nodes larger.

In the documentation procedure 602 of the presentation procedure 209, to generate HTML code based on queries on the database, a dynamic hypertext tool can
be used, such as, for example PHP (PHP Hypertext Processor, available from http://www.php.net/). PHP is an HTML-embedded scripting language, developed for dynamically generating HTML pages. It contains support for a wide range of databases, including MySQL. In the on-line version of a computer arrangement and method according to the present invention, the processing means 21 utilise PHP as a server-side scripting engine to generate HTML code dynamically. In the off-line version, the processing means 21 use PHP at "compile time" to generate static hypertext-based documentation.

To present types in the context of programs and copybooks, the processing means 21 can integrate them with software system documentation that is automatically derived from legacy sources by a documentation generation system, such as DocGen (A. van Deursen, T. Kuipers, "Building documentation generators", Int. Conf. on Software Maintenance, ICSM'99, pp. 40-49, IEEE Computer Society, 1999), during the execution of the documentation procedure 602. Such software system documentation describes the system at various levels of detail, as known to persons skilled in the art. One method of integration is to provide links from variables and literals occurring in the source code to their inferred type pages.

Moreover, in the type inferencing procedure 502, the processing means 21 derive signatures for COBOL modules that are called or can be called by others. Such a signature documents the intended use of a module. It gives the types of the formal parameters, which are derived from the variables declared in the COBOL linkage section. The signatures presented can be used to understand the interfaces of the programs of the software system analysed. In the type-related documentation generated in documentation procedure 602, this not only provides information about the formal parameters: the aforementioned type graph of each of the formal parameters also contains subtypes for all actual parameters used in the software system under analysis.

Further, in the context of programs and copybooks, in the type inferencing procedure 502, the processing means 21 obtain types for the records that are written to or read from persistent data stores such as data files or database tables. In particular, in COBOL systems, such records are likely to hold business-related data. In the type-related documentation generated in documentation procedure 602, the types of these records indicate how such business data is used within individual programs, or across the entire software system analysed.
Also, from the type-related documentation generated in documentation procedure 602, type-dependencies between programs and copybooks can be derived. Clearly, if a program uses a variable declared in a copybook, the program depends on that copybook. A second possibility is that a first copybook containing a section (to be included in the procedure division), uses variables declared in a separate second copybook (to be included in the data division). This leads to an inferred type dependency between the using first copybook and the declaring second copybook.

In the documentation procedure 602, the processing means 21 generate index files to types and programs, listing all words found in types, type names, types used in signatures, types used in persistent data stores, and so on. Moreover, in procedure 602, the processing means 21 generate listings of all programs, tables, and so on with additional type information, such as the type signature which concisely reveals the intended purpose of a program. These index files are included at the top-level, but also at the subsystem, program, type cluster, and copybook level.

The present invention enables people unfamiliar with a given software system to acquire in-depth understanding of many important aspects of a software system, such as:
- the system's key data structures,
- the actual use of these data structures,
- the system's business data, as stored in persistent data stores,
- the actual use of the business data in the program logic,
- business logic in the form of statements manipulating business data,
- program signatures and component interfaces,
- dependencies between programs showing how they share types via copybooks, databases, and calls,
- dependencies between copybooks,
- the system's overall architecture, comprising components as well as their interfaces,
- a detailed view on the role of variables in any particular piece of code.

Normally, either complicated queries or inaccurate textual searches are needed to acquire this understanding. The present invention achieves high accuracy by conducting full type inference. Moreover, it achieves ease of use by relying on a solid navigation structure, which not only hides a number of complicated underlying queries, but which also permits switching smoothly from one representation to another.
The resulting understanding is essential to perform many tasks concerning software systems. An important category of tasks is related to software maintenance, which generally involves 60% of the total cost of deploying a software system. With the present invention, such maintenance tasks can be planned more accurately, and conducted more effectively. Typical maintenance tasks supported by the present invention include:

- modifying key data structures,
- analysing the impact of data structure modifications,
- analysing the impact of functionality modifications,
- simplifying a software system’s external interfaces,
- assessing the success of a modification by comparing the type structure before and after a modification.

Another category of tasks supported by the present invention includes re-engineering the software system. These activities are usually considerable projects, which require a careful planning and effective support while conducting the re-engineering. The present invention supports, for example, planning and carrying out:

- modification of a system’s interfaces such that it can be accessed via new channels, such as, for example, the world-wide-web,
- modernisation of a software system by migrating to object technology,
- modernisation of a software system by migrating to a new database management system,
- integration of a software system with other software system, for example after a company merger or acquisition.

Last but not least, the present invention can be used for quality assessment. It provides insight into the external interfaces of a system, as well as it's internal structure. This is needed, for example, when:

- two companies consider a merger, and want to assess the costs and benefits of unifying their installed software bases,
- a software house specialised in outsourcing considers offering a (fixed-price) contract for conducting software maintenance of a particular software system,
- software maintenance on a particular software system is experienced as too expensive.
Claims

1. Computer system (100), comprising processing means (21) and memory means (18-20, 22) connected to said processing means (21); said memory means (18-20, 22) comprising a first data file (201) representing a source file of a software system, said software system having a logical structure defining said software system’s functionality, said source file being defined in a programming language comprising a set of instructions, said source file further comprising a set of variables, said programming language being defined by a grammar, said grammar comprising a set of grammatical elements; said grammatical elements defining a syntactical structure of said set of instructions; said memory means (18-20, 22) further comprising a second data file (203) representing a first rule-set of said grammar of said programming language; the processing means (21) being arranged to carry out the following functions:
   • in a first step (202), to parse said source contained in said first data file (201) into a logically equivalent sequence of grammatical elements by using said first rule-set (203);
   • in a second step (205), to extract a set of facts (206) from said logically equivalent sequence of grammatical elements by comparing said logically equivalent sequence with a second rule-set (405) which comprises rules for defining type definitions and type relations;
   • in a third step (207) to derive and to infer type-related relationships between variables in said set of variables from said set of facts (206)
the processing means (21) execute a querying procedure (601) to define a type-related information structure for said software system.

2. Computer system (100) according to claim 1, characterised in that the processing means (21) are further arranged to carry out the following function:
   • to generate as type-related information for said structure of said software system for each derived and inferred type-related relationship at least one item selected from a list of items comprising at least:
     - a byte representation;
- an enumeration range;
- usage links in said source of said software system;
- links to records;
- links to programs;
- links to copybooks;
- a type name;
- a representation structure for visualisation of said types, said subtypes and said supertypes.

3. Computer system (100) according to claims 1 or 2, characterised in that said query procedure (601) provides rules for abstraction during extraction of said type-related information.

4. Computer system (100) according to any of the preceding claims, characterised in that said type-related relationships are being defined as equivalences, subtypes and supertypes.

5. Computer system (100) according to any of the preceding claims, characterised in that the processing means (21) are further arranged to carry out the following function:
   • to merge said equivalences and said subtypes of a type-related relationship between variables in said set of variables into a type-cluster.

6. Computer system (100) according to any of the preceding claims, characterised in that the processing means (21) are further arranged to carry out the following function:
   • to format said type-related information as hypertext or graph, and
   • to present the result in an on-line or off-line mode to a user.

7. Computer system (100) according to any of the preceding claims, characterised in that the processing means (21) are further arranged to carry out the following function:
• to display said type-relationships visually as a graph displaying said subtype and supertype dependencies between displayed types, and displaying assignments made to variables of said displayed types.

8. Computer system (100) according to any of the proceeding claims, characterised in that said type-related information structure for said software system comprises navigational information for navigating said structure of said software system.

9. Computer system (100) according to any of the preceding claims, characterised in that the processing means (21) are arranged to carry out the following function:
• to add said type-related information for said structure of said software system into documentation generated for said structure of said software system by a documentation generation system.

10. Computer system (100) according to any of the preceding claims, characterised in that the processing means (21) are further arranged to carry out the following function:
• to modify documentation generated for said structure of said software system such that type-related information is added to said documentation, said type-related information containing at least one item from a list of items comprising at least:
  - a type signature for programs,
  - a type for columns occurring in database tables,
  - a type for records used for flat files,
  - a type for data entered through on-line screens.

11. Computer system (100) according to any of the preceding claims, characterised in that the processing means (21) are further arranged to carry out the following function:
• to modify documentation generated for said structure of said software system such that type-related dependencies between system elements are added to said documentation, said type-related dependencies containing at least one item from a list of items comprising at least the following items:
  - copybooks,
- programs,
- tables,
- columns,
- flat files,
- screens.

12. Computer system (100) according to any of the preceding claims, characterised in that said type-related information is transmitted over a network (1).

13. A method to be carried out by a computer system (100), comprising processing means (21) and memory means (18-20, 22) connected to said processing means (21); said memory means (18-20, 22) comprising a first data file (201) representing a source file of a software system, said software system having a logical structure defining said software system's functionality, said source file being defined in a programming language comprising a set of instructions, said source file further comprising a set of variables, said programming language being defined by a grammar, said grammar comprising a set of grammatical elements; said grammatical elements defining a syntactical structure of said set of instructions; said memory means (18-20, 22) further comprising a second data file (203) representing a first rule-set of said grammar of said programming language; said method comprising the following functions:

- to parse said source contained in said first data file (201) into a logically equivalent sequence of grammatical elements by using said first rule-set (203);
- to extract a set of facts (206) from said logically equivalent sequence of grammatical elements by comparing said logically equivalent sequence with a second rule-set (405) which comprises rules for defining type definitions and type relations;
- to derive and to infer type-related relationships between variables in said set of variables from said set of facts (206);

characterised in that

the method comprises a querying procedure (601) to extract from said type-related relationships between variables in said set of variables type-related information for said logical structure of said software system.
14. Method to be carried out by a computer system (100) according to claim 13, characterised in that said type-related information for said logical structure of said software system for each derived and inferred type-related relationship comprises a list of items comprising at least:

- a byte representation;
- an enumeration range;
- usage links in said source of said software system;
- links to records;
- links to programs;
- links to copybooks;
- a type name;
- a representation structure for visualisation of said types, said subtypes and said supertypes.

15. Method to be carried out by a computer system (100) according to claim 13 or 14, characterised in that the method provides rules in said query procedure (601) for abstraction during said extraction of said type-related information.

16. Method to be carried out by a computer system (100), according to any of the preceding claims 10-12, characterised in that said type-related relationships are being defined as equivalences, subtypes, and supertypes.

17. Method to be carried out by a computer system (100), according to any of the preceding claims 13-16, characterised in that the method further comprises the following function:

- to merge said equivalences and said subtypes of a type-related relationship between variables in said set of variables into a type-cluster.

18. Method to be carried out by a computer system (100), according to any of the preceding claims 13-17, characterised in that the method further comprises the following function:

- to format said type-related information as hypertext or graph, and
- to present the result in an on-line or off-line mode to a user.
19. Method to be carried out by a computer system (100), according to any of the preceding claims 13-18, characterised in that the method further comprises the following function:

- to display said type-relationships visually as a graph displaying said subtype and supertype dependencies between displayed types, and displaying assignments made to variables of said displayed types.

20. Method to be carried out by a computer system (100), according to any of the preceding claims 13-19, characterised in that said type-related information structure for said software system comprises navigational information for navigating said structure of said software system.

21. Method to be carried out by a computer system (100), according to any of the preceding claims 13-20, characterised in that the method further comprises the following function:

- to add said type-related information for said logical structure of said software system into a document generated for said logical structure of said software system by a documentation generation system.

22. Method to be carried out by a computer system (100), according to any of the preceding claims 13-21, characterised in that the method further comprises the following function:

- to modify documentation generated for said structure of said software system such that type-related information is added to said documentation, said type-related information containing at least one item from a list of items comprising at least:

  - a type signature for programs,
  - a type for columns occurring in database tables,
  - a type for records used for flat files,
  - a type for data entered through on-line screens.
23. Method to be carried out by a computer system (100), according to any of the preceding claims 13-22, characterised in that the method further comprises the following function:
• to modify documentation generated for said structure of said software system such that type-related dependencies between system elements are added to said documentation, said type-related dependencies containing at least one item from a list of items comprising at least the following items:
  - copybooks,
  - programs,
  - tables,
  - columns,
  - flat files,
  - screens.

24. Method to be carried out by a computer system (100), according to any of the preceding claims 13-23, characterised in that the method provides information for maintenance and/or impact analysis of said software system.

25. Method to be carried out by a computer system (100), according to claim 13-24, characterised in that the method provides information for reengineering of said software system.

26. Method to be carried out by a computer system (100), according to any of the preceding claims 13-25, characterised in that the method provides information for quality assessment of said software system.

27. Method to be carried out by a computer system (100) according to any of the preceding claims 13-26, characterised in that said type-related information is transmitted over a network (1).
28. Computer program product to be loaded by a computer system (100), comprising processing means (21) and memory means (18-20, 22) connected to said processing means (21); said memory means (18-20, 22) comprising a first data file (201) representing a source file of a software system, said software system having a logical structure defining said software system's functionality, said source file being defined in a programming language comprising a set of instructions, said source file further comprising a set of variables, said programming language being defined by a grammar, said grammar comprising a set of grammatical elements; said grammatical elements defining a syntactical structure of said set of instructions; said memory means (18-20, 22) further comprising a second data file (203) representing a first rule-set of said grammar of said programming language;

and allowing said computer system (100) to carry out the following functions:

- in a first step (202), to parse said source contained in said first data file (201) into a logically equivalent sequence of grammatical elements by using said first rule-set (203);

- in a second step (205), to extract a set of facts (206) from said logically equivalent sequence of grammatical elements by comparing said logically equivalent sequence with a second rule-set (405) which comprises rules for defining type definitions and type relations;

- in a third step (207), to derive and to infer type-related relationships between variables in said set of variables from said set of facts (206); characterised by the following additional function:

  - to execute a querying procedure (601) to define a type-related information structure for said software system.

29. Data carrier provided with a computer program product as claimed in claim 28.
Fig 3

READ PARSE TREE FILE

CONVERT INTO JAVA OBJECT

ANALYSE PARSE TREE OBJECT

STORE ANALYSIS RESULTS IN DATABASE; EMIT TYPE-RELATED FACTS

CONSTRUCTOR FILTER

DATABASE OBSERVED TYPE-RELATED FACTS

EXIT
Fig 4

207

501

RETRIEVE
OBSERVED
TYPE-FACTS

DATABASE:
OBSERVED AND
INFERRED TYPE-
FACTS

502

INFER TYPE
RELATIONS

503

STORE OBSERVED
AND INFERRED
TYPE-FACTS

504

EXIT
Fig 5

QUERY ON RELATIONAL DATABASE

GENERATE (TYPE) DOCUMENTATION

OBSERVED AND INFERRER TYPE-FACTS

HYPERTEXT DOCUMENTATION

EXIT
**INTERNATIONAL SEARCH REPORT**

**A. CLASSIFICATION OF SUBJECT MATTER**

| IPC   | G06F9/44 |

According to International Patent Classification (IPC) or to both national classification and IPC.

**B. FIELDS SEARCHED**

Minimum documentation searched (classification system followed by classification symbols)

| IPC   | G06F |

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched.

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

EPO-Internal, WPI Data, INSPEC

**C. DOCUMENTS CONSIDERED TO BE RELEVANT**

<table>
<thead>
<tr>
<th>Category</th>
<th>Citation of document, with indication, where appropriate, of the relevant passages</th>
<th>Relevant to claim No.</th>
</tr>
</thead>
</table>

Further documents are listed in the continuation of box C.

| X | Patient family members are listed in annex. |

* Special categories of cited documents:

- **A** document defining the general state of the art which is not considered to be of particular relevance
- **E** earlier document but published on or after the international filing date
- **L** document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another document or of other special reason (as specified)
- **O** document referring to an oral disclosure, use, exhibition or other means
- **P** document published prior to the international filing date but later than the priority date claimed

- **F** later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
- **X** document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
- **Y** document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
- **A** document member of the same patent family

Date of the actual completion of the International search: 18 July 2001

Date of mailing of the International search report: 06/08/2001

Name and mailing address of the ISA

European Patent Office, P.O. Box 5818 Patentlaan 2 NL-2280 Hw Rijswijk
Tel (+31-70) 340-2040, Fax: 31 651 epc nl, Fac (+31-70) 540-3016

Authorized officer

Krischer, S

Form PCT/03/A0210 (second sheet) (July 1999)
<table>
<thead>
<tr>
<th>Category</th>
<th>Citation of document, with indication, where appropriate, of the relevant passages</th>
<th>Relevant to claim No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>US 5 778 368 A (HOGAN KEITH ET AL)  7 July 1998 (1998-07-07)  claim 1; figures 4,10</td>
<td>1,6-8, 12,13, 18-20, 27,28</td>
</tr>
</tbody>
</table>
## INTERNATIONAL SEARCH REPORT

<table>
<thead>
<tr>
<th>Patent document cited in search report</th>
<th>Publication date</th>
<th>Patent family member(s)</th>
<th>Publication date</th>
</tr>
</thead>
<tbody>
<tr>
<td>US 5778368 A</td>
<td>07-07-1998</td>
<td>NONE</td>
<td></td>
</tr>
</tbody>
</table>