INFORMATICA, 1997, Vol. 8, No. 1, 153–175

PROGRAM ANALYSIS AND VISUALISATION: TOWARDS A DECLARATIVE APPROACH

Diana SIDARKEVIČIŪTĖ

Department of Teleinformatics, Royal Institute of Technology
Electrum 204, 164 40 Kista, Sweden
E-mail: diana@it.kth.se

Abstract. The aim of *program analysis and visualisation (PA&V)* is to help the programmer understand a program by means of graphical presentations of different aspects of the program. Program analysis and visualisation systems can be classified according to the specification method of visualisation, e.g., in what way can the user of the system specify his own visualisers. In the article three specification methods (predefinition, annotation and declaration) are discussed and some example systems are presented. Particular attention is paid to the declarative specification method, thus, in addition, knowledge-based program analysers are discussed. Increased understandability and modifiability are argued to be the main advantages of declarative PA&V systems.

The general discussion is continued by a short presentation of a case study, where the declarative and synthesisable visualisation in the NUT system is discussed.

Key words: program analysis, program understanding, program visualisation, declarative visualisation, knowledge-based program analysis

1. Introduction

Program analysis and understanding. The aim of *program code analysis* is to help the programmer understand the functionality of a program. What is then *program understanding*? The definition of program understanding constitutes itself a research topic. We agree with the informal definition provided in the paper Biggerstaff *et al.* (1994): "a person understands a program when able to explain the program, its structure, its behaviour, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the programs".

Program understanding has been called a challenge of the 90's (Corbi, 1989). This statement is in no need of justification. A huge amount of legacy code is in

use which is difficult to maintain, difficult to change and even more difficult, or even impossible, to discard. Documentation is usually out-of-date, inconsistent or incomplete. Working in such conditions "programmers have become part historian, part detective, and part clairvoyant" (Corbi, 1989). It follows that any tool, which facilitates program analysis and understanding is valuable.

In software engineering research, *program understanding* is, according to Johnson (1994), understood in two different ways: "some use it to refer to automated techniques that determine the intended function of a software system from source code. Others use it to refer to tools that help people understand the design of a piece of software but may not be capable of analyzing the code themselves".

We assume that only intensive research and experimentation can transform the current assistant-tools into fully automated "decision-makers". Consequently we shall only discuss the tools and techniques of program analysis aimed at helping the programmer with program understanding.

Program visualisation. Traditionally in software engineering tools, program understanding has been enhanced by means of graphical presentations illustrating divers aspects of a program. The construction of a graphical presentation of a program is called *program visualisation*. The term *software visualisation* is also used instead of program visualisation, although we argue that software visualisation covers much more. In addition to program visualisation it may also include visualisations of requirement specifications, information of configurations, history of corrections and similar.

Program visualisation, as discussed in (Shu, 1988), covers pretty-printing of source programs, visualisation through diagrams, multiple views of a program and its execution states, algorithm animation. It is interesting to observe that most program visualisation systems introduce their own graphical notations as well as methods for the presentation of the textual code. It is rather difficult to distill a common notation or technique. This is, of course, in part due to the fact that the systems have quite different goals. But it also indicates that program visualisation is still in the experimental stage.

The distinction between program analysis and program visualisation tools is not clear-cut. On one hand, program visualisation tools usually work with a fixed program model (for example, an abstract syntax tree) and place the main emphasis on efficient and appropriate graphical presentations of the information

from the model; whereas program analysis tools offer alternative, advanced program models (such as, for example, connectionist models). On the other hand, every program visualisation tool contains at least one analysis feature, and program analysis tools usually include graphical presentations of program models. As the distinction is highly subjective and, in many cases, cannot be deduced from articles, we will henceforward refer to program analysis and program visualisation tools as: program analysis and visualisation tools.

Program analysis and visualisation. *Program analysis and visualisation (PA&V)* research investigates the ways of combining the features of program analysis and program visualisation tools. Ideally, PA&V tools should offer the possibility to specify different program models and to present these models graphically in different (also specifiable) ways. Neither of the systems presented in the paper fully satisfy these requirements. In practice, PA&V tools do not cover such a wide range of features and either program analysis or program visualisation is limited.

The organisation of the paper. The paper discusses current trends in the development of tools for program analysis and visualisation. These trends are revealed in an overview of a set of existing PA&V tools (Part 2) where the systems are classified according to the specification method of visualisation. Three common methods are discussed: predefinition, annotation and declaration. For each of these methods, a description, example systems and notes on limitations and advantages are given. The overview of PA&V tools is continued (Part 3) by a presentation of knowledge-based program analysers. Then the general discussion is "mapped" to a case study (Part 4). In particular, we discuss the development and usage of declarative and synthesisable program visualisers in the NUT system (the NUT system itself and the language are also briefly presented in the article). An illustration of the construction of a declarative visualiser as well as its work are provided informally, through an example.

2. Specification methods of PA&V tools. Researchers contributing to the PA&V field offer different classifications of systems being developed. Price *et al.* (1993) use such classification criteria like *scope*, *form*, *content*, *method*, *interaction* and *effectiveness*, whereas Roman and Cox (1993) consider *scope*, *abstraction*, *interface*, *presentation* and *specification method*. In this article we adopt the last criteria mentioned – specification method. Thus, we ask the

question: Can the user of a particular PA&V system specify his own visualisers? If yes, then how? As in the taxonomy of Roman and Cox (1993), we distinguish three main specification methods: predefinition, annotation and declaration.

2.1. Predefinition

2.1.1. Method description. Tools with a predefined method for PA&V hide inside a "black box" all knowledge employed in the visualisation process. The user can neither construct his own views nor modify them and is obliged to employ the predefined graphical notation.

2.1.2. Examples: code viewers. The most common PA&V tools are *code viewers* – tools, which offer the user a fixed set of graphical presentations of an input program. In a series of articles (Koskinen *et al.*, 1994; Linos and Courtois, 1994; Wilde and Huitt, 1992) on object-oriented program maintenance, a book on visual object-oriented programming (Burnett *et al.*, 1995) (articles (Citrin *et al.*, 1995; Chang *et al.*, 1995; Grundy *et al.*, 1995) in particular) a rich set of views is offered. These include the following (the list could definitely be lengthened by consulting more articles and books):

- control flow graphs;
- data flow graphs;
- backward and forward slicers (showing the minimal subset of the code that affects a set of variables and showing the minimal subset of the code affected by a set of variables);
- dicers (showing the subset of the code that can be executed when a given assertion is true);
- definition/usage graphs of program variables;
- call graphs;
- module dependence graphs;
- class hierarchies (inheritance, containment) in OO programs;
- tracing chains of polymorphic functions;
- symbols' (program tokens) lookups;
- deadcode views;
- program layers;
- results of simple queries;
- domain-specific execution visualisers.

The construction of some views is also a feature in many CASE tools which

support reengineering. For example, in Rational Rose (a product of Rational Software Corp.) inheritance and aggregation hierarchies can be shown from the user's C++ program. Similarly, but more configurable, the Graphical Designer (a product of Advanced Software Technologies, Inc.) constructs a variety of views of C and C++ programs.

Why are there so many and so different views? The existence of code views is based on the idea of *program dependencies*. A program dependency can be described as a triple $(Point1InCode, Point2InCode, Link)$. (An example could be $(Class1, Class2, Inheritance)$.) On the other hand, during the development of the area of software engineering, various graph-based presentations of software were offered, beginning with simple program block-charts and continuing to the present day object-oriented diagrams. The graphs are also described by a triple $(Node1, Node2, Link)$. Combining various program dependencies and views of graphs has caused the emergence of quite a varied set of program views. In essence, the process of viewing a program as a graph includes the extraction of instances of a program dependency, the storage of these instances, and the retrieval (for a query) or mapping to a graph (for a graphical view). Chen *et al.* (1990) point to the need of having a concise conceptual model (for example, the entity-relationship model), which defines the software objects and relationships at a selected level of abstraction. But in many of the aforementioned articles this need is not addressed.

2.1.3. Method advantages and limitations. The main advantage of the above mentioned systems is performance. As the construction of views is predefined, then specialised, optimised algorithms can be applied. Very often program visualisation is but one feature among others, nicely integrated with other subsystems (like forward and reverse engineering features are integrated in many CASE tools).

Users of these systems face different kinds of problems. First, they are often offered a narrow set of views in one system. Users may have their own, highly individual "mental maps" of programs. The potential user of program visualisation tools is, most probably, a programmer himself and capable of specifying his own visualisers. It could be argued that the user should be provided with the option to specify his own visualisers, considering that PA&V is still a hot research topic and that researchers are still far from having defined the complete set of program views.

Second, the semantics of graphical symbols used are described in a very informal way. If we can assume that a program dependency can be explained informally or understood intuitively (which is not always true), then we cannot rely on an informal description of the mapping from a program dependency to a graphical view. Suppose, for example, that two classes in a graph are joined with a line presenting link "uses". Do we take into account calling a class from an implementation of a method of another class or not? This and similar points might not be clear. Evidently, the user would like to open the "black box" or, in other words, he needs access to internal representations of visualisers.

2.2. Annotation

2.2.1. Method description. The annotation method is mainly applied in algorithm animation. Here, the user develops animation procedures and marks (or annotates) an input program text with calls to these procedures. Procedures' parameters are used for data passing.

2.2.2. Examples: program animators. In the Balsa-II system (Brown, 1988), the animation of an algorithm involves three steps. First, the program is split into three components: the algorithm itself, various input generators that provide data for the algorithm and different views. Second, the components are implemented. Components have parameters through which the data is exchanged. The implementation of new views or input generators involves the reuse of existing components from the library. Third, views and input generators which can be used with each algorithm are identified and named. The main effort of a Balsa-II programmer is spent in annotating the algorithm being animated. This is quite understandable, as the identification of the essential operations in the algorithm is by no means a trivial task itself. To the eager reader we suggest to take a look at the article (Brown, 1988), where the different steps in the construction of the animation are presented (as well as attractive snapshots of animations).

The Tango system (Stasko, 1990) is based on a framework which includes three components. To produce an animation, the user must 1) annotate the program with algorithm operations (or calls to animation procedures – as named above), 2) write animation actions and 3) specify the mapping from algorithm operations to animation scenes. Keeping mapping and animation procedures

separated gives a significant advantage in flexibility terms as the user can change the animation simply by editing the mapping file.

2.2.3. Method advantages and limitations. The key advantage of the annotation method is that the user is permitted to provide his own definition of what should be animated and how. The user can himself define appropriate events in program execution as well as the way these events should be presented graphically.

The possibility to write your own animation procedures can be considered a disadvantage as well, because it consumes additional work. Here, libraries of animation procedures facilitate the process and ease the workload. But then instead, libraries must be well understood themselves, which is not a trivial task in imperative programming. One more disadvantage of the annotation method is the need to modify the program code.

2.3. Declaration

2.3.1. Method description. PA&V tools which apply declarative approaches differ as significantly from each other as different are the methods which can be typed declarative. Typically, the user is provided with an environment in wich he can specify his own visualisers in a given declarative language. As shown in Fig. 1, the process of writing your own visualiser includes, essentially, the specification of program and view models and of a mapping between these models. The extraction of a program model from an input program and the presentation of the view graphically can also be specified by the user or else done automatically by the system. Additional models (like a user model or similar) can easily be added in the same way as, for example, a new view model.

2.3.2. Examples: declarative visualisers. The usage of the declarative approach in PA&V systems ranges from the introduction of simple declarative mappings to the employment of declarative languages tailored to the specifics of PA&V.

Declarative mappings. In the aforementioned TANGO system the control file serves as storage place for the declarative specifications. Here the names of algorithm operations and animation scenes and mappings between them can be listed. The mappings have simple form: *algorithm operations* $\longrightarrow$ *animation scenes*.

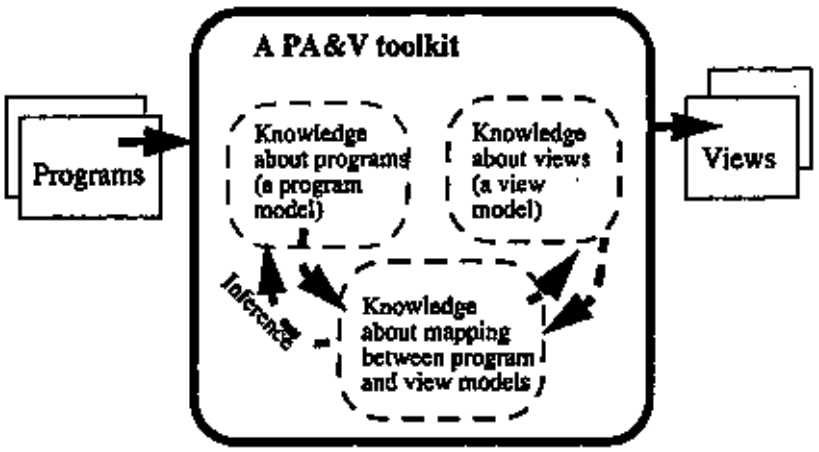

Fig. 1. The general architecture of a declarative PA&V toolkit.

In the reflexion model approach (Murthy *et al.*, 1995) for software analysis, a reflexion model is introduced in addition to a source model, a high-level model and a mapping. Although the authors do not purport to follow a declarative approach, they actually use a declarative language for the specification of maps.

(Selfridge and Heineman, 1994) Interactive Code Understanding Environment (ICUE) takes the information about a C program stored in a database and provides the user with a graphical query-formation facility as well as the environment for manipulating object graphs (the graphical representations of the results of queries).

Declarative languages. In the Pavane system (Roman *et al.*, 1992) the underlying visualisation model is declarative in the sense that visualisation is treated as a mapping from program states to a three-dimensional world of geometric objects. All mappings are represented by rules. Rules can be added, deleted or modified during visualisation. The specification of the visualisation in Pavane requires the user to formally specify the state of programs. This forces the user to work more on the conceptualisation of program behaviour, which although being a time-consuming requirement also gives a fundamental benefit – a deeper understanding of the nature of computations and their graphical representations.

In the SPE/Cerno system (Grundy *et al.*, 1995), users are free to choose both the contents and layout of views. The construction of new display abstractions involves specialising and creating new abstractor classes (written in Snart), while new display visualisations can be developed using the icon layout language. Both Snart (an object-oriented extension to PROLOG) and the icon layout language are declarative.

The key technological idea in the (Kotik and Markosian, 1992) approach is code representation as an annotated abstract syntax tree in an object-oriented database. This approach differs essentially from code viewers (discussed above) as it also provides a high-level language, the Refine language, which allows the user to operate on the abstract syntax tree. For example, one can define one's own analysis functions. The same language is used for the specification of graphical views of the results of analysis functions. This is implemented in Refine Language Tools (a product of Reasoning Systems, Inc.), where the initial set of graphical views can be extended with views written by the user in the Refine language.

In the SoftSpy system (discussed in more detail in Part 4), the user is given full liberty to specify his own visualisers in the NUT language. A specification, as well as a request for computation, are translated into logical language, a proof is performed and, if successful, a visualiser is synthesised. The user is also provided with an environment, which has facilities for NUT language processing, graphics management and other.

2.3.3. Method advantages and limitations. In the declarative approach the user has to abstract (or conceptualise) the construction of a program's view and to record the abstraction (or a conceptual model) in a given declarative language. The conceptualisation is always time and effort-consuming work. But this conceptualisation is in any case performed by the user when trying to understand a program. And so, the main role of a declarative PA&V system is to provide an environment, where the user can operate with the conceptual models he produced: to record, reuse or modify them.

An explicit representation is particularly important in the process of PA&V, as in this case, at least two distinct conceptual models are involved: a model of the program and another of its view. The mapping between different models is declarative by nature, and, can thus, more naturally be represented in a declarative language.

In addition to being more understandable, visualisers written in a declarative language are easier to modify. Easy addition, deletion or change of atomic units of knowledge (like rules, definitions of domain entities etc.) is a feature inherited from knowledge-based systems (knowledge-based systems are chosen as implementation environments for many declarative visualisers).

The main disadvantage (which declarative PA&V systems inherited from knowledge-based systems) is a low speed of execution. The good side of the coin is that speed measures of many knowledge-based methods have been extensively investigated and optimisations are known. In addition, the language used is usually adapted to the visualisation problem and simplified.

3. Knowledge-based program analysers. Kozaczynski *et al.* (1992); Johnson (1994) discuss the general organisation of typical knowledge-based program analysers. This organisation usually includes the parsing of a code, typically generating an abstract syntax tree representation, stored in a knowledge base. The knowledge base also includes representations of programming knowledge or, more precisely, common programming patterns and techniques, variously called *design schemas, programming cliches* and *programming plans*. The analyser matches the programming patterns with the code to infer that higher-level specification concepts are present in the code. The user of a knowledge-based code analyser is provided with the possibility to modify the knowledge-base (programming patterns) as well as to use inferential services by asking questions.

Various representations of programming knowledge and system models as well as inferential features influenced the development of different knowledge-based software analysers:

- Wills (1992) studies a graph parsing approach to automating program recognition in which programs are represented as attributed dataflow graphs and a library of cliches is encoded as an attributed grammar. A graph parsing algorithm is used to recognise cliches in the code.
- Quilici (1994) represents programming plans as data structures containing two parts: a plan definition, which lists the attributes of the plan that are filled in when instances of the plan are created, and a plan recognition rule, which lists the components of a plan and the constraints on those components. An instance of the plan is recognised when all its components have been recognised without violating the constraints.

In addition, each programming plan also includes indices, specialisation constraints, and a list of implied plans. The algorithm employed makes use of indices in order to suggest general candidate plans to match top-down against the code, specialisations to refine these general plans once they are recognised, and implications to recognise other, related plans without doing further matching.

- the LaSSIE system (Devanbu *et al.*, 1991) provides two types of inference: subsumption and rules. The knowledge base has descriptions of the objects and operations in the domain, the processes, layers and messages in the architecture, and the functions, variables and files associated with the code.

- GEN++ (Devanbu *et al.*, 1992), a code analysis tool generator for C++, is implemented by attaching the portable parse tree querying mechanism to the Cfront compiler. GENOA is an applications' generator that produces arbitrary analysers from specifications. The GENOA language has special iteration operators that are tuned for expressing simple, polynomial time analysis programs. The GENOA specification language uses the vocabulary of abstract syntax trees.

- Kozaczynski *et al.* (1992); Harandi and Ning (1990) use an object-oriented environment to implement the concept recognition system. All language and abstract concepts are represented internally as objects of a knowledge base. Plans are also objects and have methods associated with them for recognising concept instances. These instances are found by pattern matching, which is a unification of abstract syntax trees of the attribute values.

- In the DESIRE system (Biggerstaff *et al.*, 1994), a domain model knowledge-base is built as a semantic/connectionist hybrid network and a connectionist-based inference engine is employed.

The section below is devoted to a case study. We discuss the results in the development of toolkits for program analysis in the NUT system. In particular, we point out both merits and deficiencies of a toolkit (for the presentation of predefined graphical views of a code). We then reason about the considerable improvements of this toolkit when shifting to a declarative approach.

4. PA&V in the NUT system

4.1. A general introduction to the NUT system. NUT is a system of object-oriented programming with features of automatic program synthesis (Tyugy, 1991). The NUT programming language rests on two paradigms: procedural object-oriented programming and the automatic synthesis of programs from declarative specifications. The latter is a technique for automatic construction of programs for unprogrammed procedures out of their specifications and of the programs and specifications of programmed procedures. Here a procedure's specification embodies its external view (states the names of its input and output parameters). The automatic synthesis of programs, as practised in NUT, is based on proof search in intuitionistic propositional logic (a more detailed description of the NUT system and the NUT language can be found in (Uustalu *et al.*, 1994)).

The feature of the NUT language of being both an object-oriented programming language and a declarative language, lead us to the idea of carrying out various PA&V experiments. That is, starting with the development of code viewers for object-oriented programs we then moved on to the investigation of declarative analysis of the same code. The NUT system is well suited to this purpose as there is no need to change language and environment when switching to a new (declarative) technique.

In the following two subsections, we discuss the results of our experimentation in PA&V in the NUT system: predefined and declarative approaches. The predefined approach presents a toolkit for creating graphical views of NUT programs. Discussion on the declarative approach includes informal and brief introduction to the problem-oriented language, logical language, proof or inferencing issues as well as an example – once again the toolkit for creating graphical views of NUT programs. The programs selected for analysis were written in the NUT language.

4.2. A toolkit for creating OMT-based views of a program with the predefined specification method. A toolkit (Sidorkevičiūtė *et al.*, 1995) for the automatic visualisation of object-oriented software modules (or packages as they are called in the NUT system) was developed. The OMT (Rumbaugh *et al.*, 1991) graphical notation was selected, because it includes notations for the representation of static, dynamic and functional aspects of a system. OMT graphical icons are simple to draw, adapt and modify. In addition, many pro-

grammers possess knowledge about OMT.

For the presentation of the static structure, three graphical symbols from the OMT Object Model were borrowed and adapted. First, the class icon shows the name, attributes and methods of a particular class. Inherited attributes and methods are also shown. Second, the class hierarchy is visualised in a vertical tree (as it is in the OMT Object Model). Third, the aggregation tree is illustrated with a horizontal tree by using the icon for aggregation association of OMT OM. Other associations are not shown. The dynamic aspects of a code are displayed through the visualisation of a synthesised algorithm. For the visualisation of functional dependencies and dataflow, the OMT Functional Model was chosen and slightly modified. The NUT system provides program synthesis on higher-order functional constraints networks (HOFCN) – which have their own graphical notation. The graphical notation of HOFCN and the modified graphical notation of the OMT Functional Model were combined. Thus, in the functional model the data flow between the methods of a class is shown. Class methods (including equivalences and equations) are considered to be processes of the OMT Functional Model.

Some snapshots of the views constructed by the toolkit can be found on www on the address:

`http://www.it.kth.se/edu/gru/KBPVT/projects/softspy.html`.

A number of experiments were carried out. The purpose of the experiments was to estimate how much the suggested visualisation can help in understanding the program and evaluating the design. Observations were made like follows. Inheritance trees help in acquiring a general view of the static structure of a package: how many classes are employed and how many attributes and methods are used in their definitions, whether the names chosen are self-explanatory, etc. We are able to detect empty or too big classes. Aggregation trees provide a clue for discovering the "main" actors of a package. These are the classes which usually have more aggregated classes and are normally the most general classes of the design of a given problem. One can go further from this point by investigating functional models of these "actors". A functional model of a class helps the user to trace the computation of class attributes.

Some deficiencies were also detected when using this toolkit. First, the correspondence between parts of the program and graphical symbols was described very informally, and so a considerable amount of time had to be spent

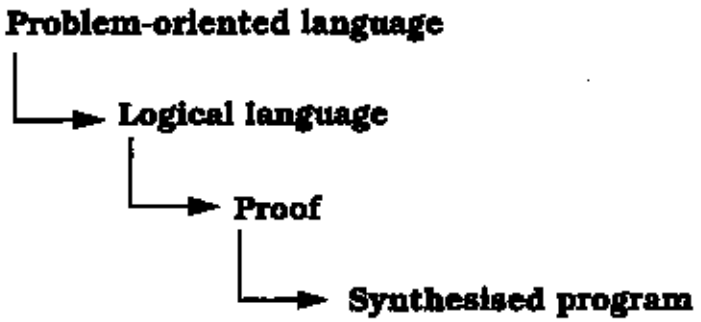

Fig. 2. The relation between languages, proof and program in the NUT system.

in order to get the meaning of pictures ("What does this symbol stand for in the code?"). Second, a very narrow set of views was offered ("Why can I not create my own view?"). For example if the user is interested only in coupling between classes or, more simple, to have just class names in inheritance trees, he can not in a flexible way specify the view he wishes to have.

The exploration of the declarative features of the NUT language seemed to deal with both problems: it would allow the user to explicitly specify any kind of internal representations and mappings involved while the structual synthesis of programs would deal with assembling the visualiser from the specifications. Thus we switched to the declarative approach in order to further extend the functionality of the code analysis toolkit.

4.3. A declarative approach. The key idea behind our declarative approach discussed is the usage of the NUT language for the representation of knowledge about a program and its various views. As shown in Fig. 2, the user starts by specifying his visualiser in a problem-oriented language (the NUT language). Then this specification is automatically mapped into a logical language in which a proof for the request is performed. If the proof succeeds, the program (or a visualiser) is synthesised. The rest of this section will be devoted to illustrating each of the steps in the process of constructing a visualiser. In order not to burden the reader with theoretical and technical details, the illustration is provided informally, with the help of an example.

An example: an OMT-based visualiser. As in Section 4.2 we again discuss an OMT-based graphical presentation of a NUT program. We redevelop our toolkit in the declarative manner as discussed just above and presented in Fig. 1.

We get three system-subparts: a representation of a program model, a representation of OM (Object Model of OMT) and a representation of the mapping of a model to OM. Fig. 3 gives a snapshot of these subparts. The representation of the model includes the classes *Package, Class, Object* (two classes are shown on the left side of Fig. 3). Each of the classes has a method *Extract...* which defines how particular attributes of the class could be computed in the particular package. The representation of OM consists of the classes *OM, OMClass, OMInhLink, OMAggrLink* (two classes are shown on the right side of Fig. 3). The classes describe OM diagrams in the NUT language and may contain methods *Draw...* for the construction of a drawing. The mapping between *Package* and *OM* is represented by two classes: *PackageToOM* and *ClassToOMClass* (shown in the middle of Fig. 3). In the class *PackageToOM*, the specification of the method *ComputeClasses* declares that if in the class *ClassToOMClass* from *Class, OMClass* can be computed then, from *Package.Classes, OM.Classes* can be computed. The class *ClassToOMClass* explicitly defines the mapping between the class in the program model and the class in OM.

The synthesis of a visualiser can be requested by the goal *obj.compute (Drawing)* – here *obj* is any object of a class *OM*.

Problem-oriented language. Classes in the NUT language are used as the main entities of model representation. Classes act as computational frames as they are enriched with computability axioms (marked with 1 in Fig. 3), which contain information about the computability of class components. A class can also have an image (for example, an image of a *OMClass* is a rectangle with one input and one output ports and a parameter for a name).

The NUT language is tailored to PA&V problem, by extending the standard function libraries with three new libraries: a library for the extraction of information from a program, another for the graphical layout and a third for passing data to visualisation in MatLab (a product of the MathWorks, Inc.). The libraries are linked dynamically.

The extraction of information from the program is supported by a set of reflective functions (marked with 3 in Fig. 3), such as *getclasses, getvar, getrel*, etc. The set of available functions covers the extraction of all entities and

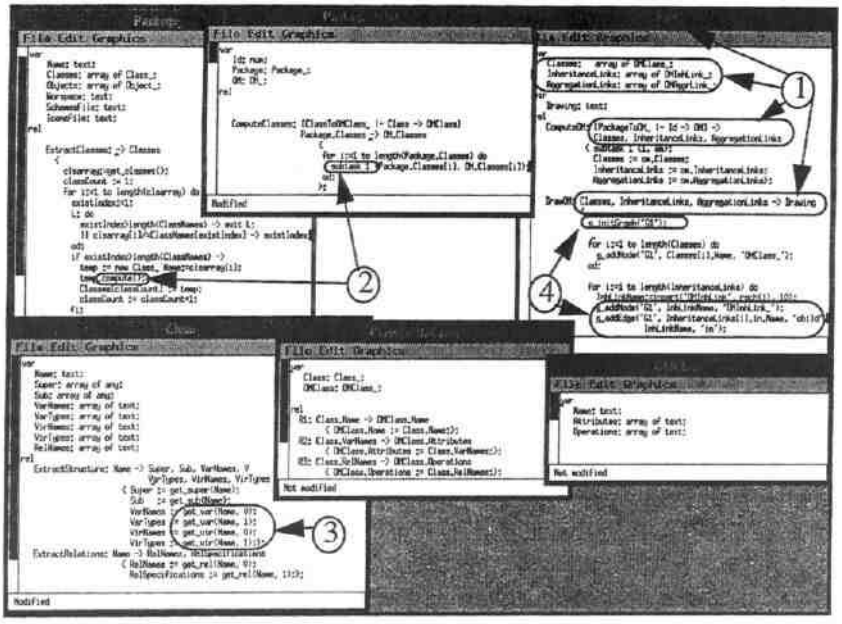

Fig. 3. Parts of the specification of an OMT-based visualiser in the NUT language: 1) specification of the OMT-based view; 2) explicit invocation of inference; 3) extraction of information directly from the program code; 4) passing of information for graphical layout.

relations according to the ontology (or model) of the program. If the user analyses a non-NUT program, he can write extraction functions in the NUT language or invoke programs written in other languages. The existence of reflective functions in NUT facilitates our program analysis task considerably. The program model is easy to build in terms of these functions.

The functions of an independent graphical layout generate a drawing from simple graph specifications. We solved the task of automatic layout of the diagrams as an instance of the general graph drawing problem (Kuusik *et al.*, 1996). We adapted algorithms addressing directed acyclic graphs, which perform, first, a level assignment of nodes by tracing their connections, and then apply some heuristics to reduce edge crossings and bends. The layout algorithm is encapsulated in a separate, self-contained graph layout subsystem under NUT. Functions of this subsystem (marked with 4 in Fig. 3) allow one to construct a

graph in a declarative way by adding edges and nodes to the graph, to request a layout calculation on the constructed graph, and finally, to store the layout as a scheme (diagram representation) of some existing class. The user can view the automatically generated diagram by requesting the NUT graphics subsystem to show the scheme of that class.

Visualisation of data in the MatLab is performed by the use of ready made classes such as *Matrix*, *MatlabLow*, *MatlabAnim*. The link to the Matlab is transparent for the user, e.g., the user specifies in the NUT language the visualisation in the MatLab.

Logical language. A logical justification for the NUT declarative language and the main reasoning procedure – the structural synthesis of programs – is provided in (Uustalu, 1996). The explanation is given in terms of a simple intuitionistic normal modal logic as the observation is made that "classification and computability statements are object-relative in object-oriented synthesis in the same way as propositions are world-relative in normal modal logics – objects and worlds are implicit in the language of each". Thus, objects are treated as worlds and component relations between objects as accessibility relations between worlds.

EXAMPLE 1. The classes `PackageToOM` and `ClassToOMClass` definitions given in Fig. 3 are translated into the following axioms (we shorten `ClassToOMClass` to `CTOMC`, `PackageToOM` to `PTOM`, `OMClass` to `OMC`, `Package` to `P`, `Classes` to `Cls` and `Class` to `Cl`):

$$\texttt{PTOM} \supset \langle \texttt{P} \rangle \texttt{P} \tag{1}$$

$$\texttt{PTOM} \supset \langle \texttt{OM} \rangle \texttt{OM} \tag{2}$$

$$\texttt{PTOM} \supset [*](\texttt{CTOMC} \supset ((\texttt{Cl})\texttt{r} \supset \langle \texttt{OMC} \rangle \texttt{r}))\langle \texttt{P} \rangle \langle \texttt{Cls} \rangle \texttt{r} \supset \langle \texttt{OM} \rangle \langle \texttt{Cls} \rangle \texttt{r} \tag{3}$$

$$\texttt{CTOMC} \supset \langle \texttt{Cl} \rangle \texttt{Cl} \tag{4}$$

$$\texttt{CTOMC} \supset \langle \texttt{OMC} \rangle \texttt{OMC} \tag{5}$$

$$\texttt{CTOMC} \supset (\langle \texttt{Cl} \rangle \langle \texttt{Name} \rangle \texttt{r} \supset \langle \texttt{OMC} \rangle \langle \texttt{Name} \rangle \texttt{r}) \tag{6}$$

$$\texttt{CTOMC} \supset (\langle \texttt{Cl} \rangle \langle \texttt{VarNames} \rangle \texttt{r} \supset \langle \texttt{OMC} \rangle \langle \texttt{Attributes} \rangle \texttt{r}) \tag{7}$$

$$\texttt{CTOMC} \supset (\langle \texttt{Cl} \rangle \langle \texttt{RelNames} \rangle \texttt{r} \supset \langle \texttt{OMC} \rangle \langle \texttt{Operations} \rangle \texttt{r}) \tag{8}$$

Here r stands for computability, $\langle$`ClassComponentName`$\rangle$ and $[*]$ denote accessibility relations. For example, axiom (6) is interpreted as follows: the

world (or object) w of the class CTOMC implies that if there exists such a world w', which is accessible from w via relations Cl and Name and is computable, then there exists such a world w'', which is accessible from w via relations OMC and Name and is computable. *End of Example 1.*

Proof. The inferencing carried out by the NUT system is called *provable realizability* (Uustalu, 1995). Its main goal is to prove the computability or non-computability of an object or its component. If computability can be proven, then an algorithm (or a program) for its computation is synthesised. A logical justification of the computability inferencing is also provided in (Uustalu, 1996) and not discussed here. Rather, an informal illustration of the inferencing procedure is provided by an example.

EXAMPLE 2. If we consider the classes discussed in Example 1, the goal given to the system could be: given an object w of the class PTOM with computed component Package.Classes, find an algorithm for computing its component OM.Classes. This amounts to proving the inference (here the abbreviation of class names is the same as in Example 1): $\dfrac{w : \mathtt{PTOM} \quad w : \langle \mathtt{P} \rangle \langle \mathtt{Cls} \rangle \mathrm{r}}{w : \langle \mathtt{OM} \rangle \langle \mathtt{Cls} \rangle \mathrm{r}}$.

The derivation (based on the rules presented in (Uustalu, 1996) is the following:

$$\dfrac{\dfrac{\overline{(3)}\ w : \mathtt{PTOM}}{w : [*](\mathtt{CTOM} \supset (\langle \mathtt{Cl} \rangle \mathrm{r} \supset \langle \mathtt{OMC} \rangle \mathrm{r}))} \quad \dfrac{\dfrac{\Theta}{w' : \langle \mathtt{OMC} \rangle \mathrm{r}}}{w : [*](\mathtt{CTOM} \supset (\langle \mathtt{Cl} \rangle \mathrm{r} \supset \langle \mathtt{OMC} \rangle \mathrm{r}))} \quad w : \langle \mathtt{P} \rangle \langle \mathtt{Cls} \rangle \mathrm{r}}{w : \langle \mathtt{OM} \rangle \langle \mathtt{Cls} \rangle \mathrm{r}}$$

Here Θ stands for the proof $\dfrac{w' : \langle \mathtt{Cl} \rangle \mathrm{r} \quad w' : \mathtt{CTOMC}}{w' : \langle \mathtt{OMC} \rangle \mathrm{r}}$.

This amounts to proving the inference

$$\dfrac{w' : \langle \mathtt{Cl} \rangle \mathrm{r} \quad w' : \mathtt{CTOMC}}{w' : \langle \mathtt{OMC} \rangle \langle \mathtt{Name} \rangle \mathrm{r} \wedge \langle \mathtt{OMC} \rangle \langle \mathtt{Attributes} \rangle \mathrm{r} \wedge \langle \mathtt{OMC} \rangle \langle \mathtt{Operations} \rangle \mathrm{r}}$$

because OMC has the components Name, Attributes and Operations.

The proof of $\dfrac{w' : \langle \mathtt{Cl} \rangle \mathrm{r} \quad w' : \mathtt{CTOMC}}{w' : \langle \mathtt{OMC} \rangle \langle \mathtt{Name} \rangle \mathrm{r}}$ is the following:

$$\dfrac{\dfrac{\overline{(6)} \quad w' : \mathtt{CTOMC}}{w' : \langle \mathtt{Cl} \rangle \langle \mathtt{Name} \rangle \mathrm{r} \supset \langle \mathtt{OMC} \rangle \langle \mathtt{Name} \rangle \mathrm{r}} \quad w' : \langle \mathtt{Cl} \rangle \langle \mathrm{Name} \rangle \mathrm{r}}{w' : \langle \mathtt{OMC} \rangle \langle \mathtt{Name} \rangle \mathrm{r}}$$

Here w' : ⟨Cl⟩⟨Name⟩r follows from w' : ⟨Cl⟩r, as Name is a component of Cl.

In the same way ⟨OMC⟩⟨Attributes⟩r from ⟨Cl⟩⟨VarNames⟩r and ⟨OMC⟩⟨Operations⟩r from ⟨Cl⟩⟨RelNames⟩r are proved.

The proof is completed. *End of Example 2.*

Synthesised programs. If the proof succeeds, a program for computing the requested component is synthesised.

Let us return to our example of an OMT-based visualiser, where the request to compute *Drawing* is given. In the case when classes do not possess enough information for the computability of *Drawing*, the unsolvable problem will be reported. In the positive case, the program will also be synthesised.

Fig. 4 presents the results of the work of the synthesised visualiser. As input program a visualiser's program was chosen.

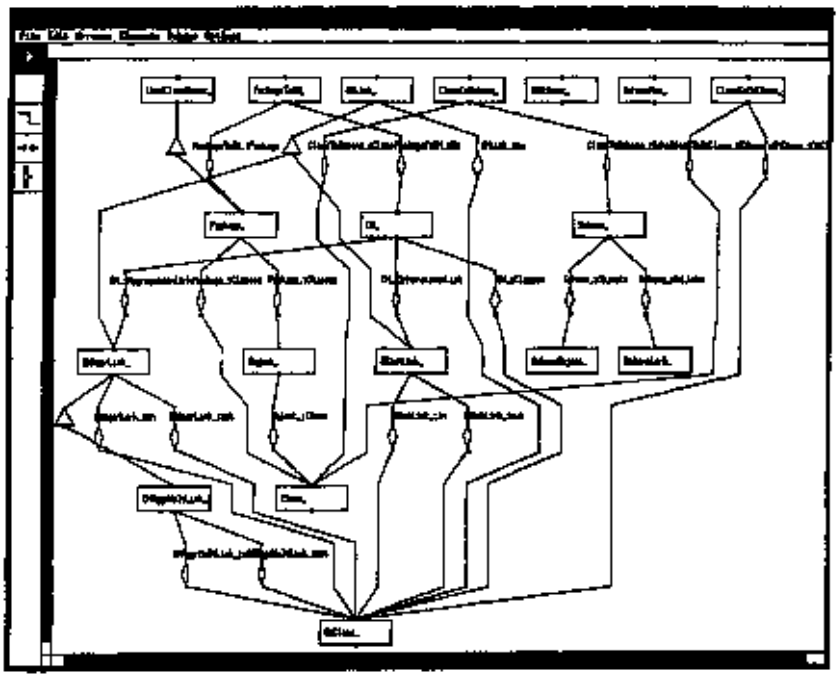

Fig. 4. Part of the view constructed by an OMT-based visualiser from a NUT program.

The advantages of a declarative approach. First, the user can easily modify his viewer. For example, if one does not want aggregation links presented in the OM view, it is sufficient just to take away *AggregationLinks* from the method *DrawOM* in the class *OM*. By invoking inference procedures, the user can check whether a modified specification still possesses the same component's computability features as the old one. Second, let us consider the different representations – one of the program model and another of the OMT-based view. The terminology used by programming language authors and OMT authors is employed in each respective case. For example, the attributes of a class are called *Variables* and *Virtuals* in the model of NUT programs (as in the NUT language documentation); and they are called *Attributes* in the OMT-based view presentation (so are they called by OMT developers). This brings us one step closer to naturalness and user-friendliness. Moreover it simplifies the introduction of changes. Third, one can easily discover, that replacing the OMT-based viewer with another viewer is not a complicated task. It involves the development of the NUT language representations of a new view and a mapping from a program model into this new view.

5. Concluding remarks. We discussed the tendency towards the declarative approach in program analysis and visualisation. Declarative program analysis and visualisation tools considerably extend the functionality of traditional PA&V tools. The main achievement is improved modifiability and extensibility of visualisers. This is due to the reason that explicit declarative specifications are easier to understand and, consequently, to modify. New program models as well as new analysers (for example, metrics tools or evaluators) can be added by adding new specifications.

We also presented a case study: the research in program code analysis in the NUT system. We argued that viewing a program code in several predefined ways is not sufficient for program understanding. This was motivated by discussing the results of applying a program visualisation toolkit in the NUT system. We redeveloped our toolkit in a declarative manner. We used the NUT declarative language for recording knowledge about PA&V. Provable realisability was the main inferencing procedure.

An important issue which has not been thoroughly investigated, and which forms the basis for future work, is the elaboration of a problem-oriented language, e.g., a language for the specification of visualisers. As shown in the

paper, a declarative language is suitable to PA&V. But what additional features (for example, what standard libraries or suitable language constructs) the language should have – remains to be investigated.

REFERENCES

Biggerstaff, T. J., B.G. Mitbander, and D.E. Webster (1994). Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5), 72–82.

Brown, M.H. (1988). Exploring algorithms using Balsa-II. *Computer*, 21(5), 14–36.

Burnett, M., A. Goldberg and T. Lewis (Eds.) (1995). *Visual Object-Oriented Programming: Concepts and Environments*. Manning Publications Co., Greenwich CT.

Chang, B.-W., D. Ungar and R. B.Smith (1995). Getting close to objects. In M. Burnett, A. Goldberg and T. Lewis (Eds.), *Visual Object-Oriented Programming: Concepts and Environments*, Manning Publications Co., Greenwich CT. pp. 185–190.

Chen, Y.-F., M.Y. Nishimoto and C. Ramamoorthy (1990). The C information abstraction system. *IEEE Transactions on Software Engineering*.

Citrin, W., M. Doherty and B. Zorn (1995). The design of a completely visual OOP language. In M. Burnett, A. Goldberg and T. Lewis (Eds.), *Visual Object-Oriented Programming: Concepts and Environments*. Manning Publications Co., Greenwich CT. pp. 67–93.

Corbi, T.A. (1989). Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2), 294–306.

Devanbu, P., B. Ballard, R. Brachman and P. Selfridge (1991). Lassie: a knowledge-based software information system. In M. Lowry and R. McCartney (Eds.), *Automating Software Design*. AAAI Press/The MIT Press, Menlo Park, CA. pp. 25–38.

Devanbu, P.T. (1992). Genoa – a customizable, language- and front-end independent code analyzer. In *Proceedings of the Fourteenth International Conference of Software Engineering (ICSE'92)*. Melbourne pp. 307–319.

Grundy, J., J. Hosking, S. Fenwick and W. Mugridge (1995). Connecting the pieces. In M. Burnett, A. Goldberg and T. Lewis (Eds.), *Visual Object-Oriented Programming: Concepts and Environments*, Manning Publications Co., Greenwich CT pp. 229–252.

Harandi, M.T., and J.Q. Ning (1990). Knowledge-based program analysis. *IEEE Software*, 74–81.

Johnson, W. (1994). Knowledge-based software enginnering. In A. Kent and J. G. Williams (Eds.), *Encyclopedia of Computer Science and Technology*, Vol. 30. Marcel Dekker, Inc., New York. pp. 173–225.

Koskinen, J., J. Paakki and A. Salminen (1994). Program text as hypertext: Using program dependencies for transient linking. In *Proc. of the 6th Int. Conference on*

Software Engineering and Knowledge Engineering, SEKE'94.

Kotik, G., and L. Markosian (1992). Knowledge-based software reengineering tools. In *Proceedings of the Seventh Knowledge-based Software Engineering Conference.* IEEE Computer Society Press, Los Alamitos, CA.

Koraczynski, W., J. Ning and T. Sarver (1992). Program concet recognition. In *Proceedings of the Seventh Knowledge-based Software Engineering Conference.* IEEE Computer Society Presss, Los Alamitos, CA. pp. 216–225.

Kunsik, A., D. Sidarkevičiūtė and E. Tyugu (1996). *Knowledge Level Software Visualisation.* Technical Report TRITA-IT 96:09, Dept. of Teleinformatics, KTH, Sweden.

Linos, P. K., and V. Courtois (1994). A tool for understanding object-oriented program dependencies. In *Proc. of the 3rd IEEE Workshop on Program Comprehension.*

Murphy, G., D. Notkin and K. Sullivan (1995). Software reflexion models: Bridging the gap between source and high-level models. In *The Third ACM Symposium on the Foundations of Software Engineering (FSE'95).*

Price, B., R. Baecker and I. Small (1993). A principled taxonomy of sofware visualization. *Journal of Visual Languages and Computing,* 4(3), 211–266.

Quilici, A. (1994). A memory-based approach to recognizing programming plans. *Communications of the ACM,* 37(5), 84–93.

Roman, G.-C., K.C. Cox, C. Wilcox and J.Y. Plun (1992). Pavane: A system for declarative visualization of concurrent computations. *Journal of Visual Languages and Computing,* 3, 161–193.

Roman, G.-C., and K.C. Cox (1993). A taxonomy of program visualization systems. *IEEE Computer,* 11–24.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy and W. Lorensen (1991). *Object-oriented Modeling and Design.* Prentice Hall International, Inc.

Selfridge, P. G., and G.T. Heineman. (1994). Graphical support for code-level software understanding. In *The Ninth Knowledge-based Software Engineering Conference.* IEEE Computer Society Press. pp. 117–124.

Shu, N. (1988). *Visual Programming.* Van Nostrand Reinhold Company, New York.

Sidarkevičiūtė, D., M. Addibpour and E. Tyugu (1995). *Experimental Software Analysis in the NUT System.* Technical Report TRITA-IT 95:16, Dept. of Teleinformatics, KTH, Sweden.

Stasko, J.T. (1990). Tango: A framework and system for algorithm animation. *Computer,* 23(9), 27–39.

Tyugu, E. (1991). Three new generation software environments. *Communications of the ACM,* 34(6), 46–59.

Uustalu, T. (1995). *Aspects of Structural Synthesis of Programs.* Lic. Thesis TRITA-IT R 95:09, Dept. of Teleinformatics, KTH.

Uustalu, T. (1996). *A Modal Justification for Structural Synthesis of Programs in NUT.* Technical Report TRITA-IT 96:06, Dept. of Teleinformatics, KTH, Sweden.

Uustalu, T., U. Kopra, V. Kotkas, M. Matskin and E. Tyugu (1994). *The NUT language report*. Technical Report TRITA-IT R 94:14, Dept. of Teleinformatics, KTH.

Wilde, N., and R. Huitt (1992). Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, 18(12), 1038–1044.

Wills, L.M. (1992). *Automated Program Recognition by Graph Parsing*. PhD thesis, MIT.

Received December 1996

D. Sidarkevičiūtė started her graduate studies at the Faculty of Mathematics, Vilnius University, Lithuania. Currently she is a doctoral student at the Department of Teleinformatics of the Royal Institute of Technology, Sweden. Her research interests include declarative program analysis and visualisation.

PROGRAMŲ ANALIZĖS IR VIZUALIZAVIMO REIKALAVIMŲ SPECIFIKAVIMAS DEKLARATYVIĄJA KALBA

Diana SIDARKEVIČIŪTĖ

Naudojant programų analizės ir vizualizavimo priemones, programos pavaizduojamos grafiškai. Tai daroma, siekiant palengvinti jų suvokimą. Programų analizės ir klasifikavimo priemonės klasifikuojamos pagal tai, kaip yra aprašomi programos vizualizavimo reikalavimai. Reikalavimai gali būti sufomuluoti iš anksto ir realizuoti, kuriant vizualizavimo priemonę, formuluojami, anotuojant vizualizuojamą programą arba formuluojami specialiai tam tikslui skirta deklaratyviąja kalba. Straipsnyje atlikta šių metodų lyginamoji analizė. Ypatingas dėmesys skirtas deklaratyviajam vizualizavimo reikalavimų formulavimo metodui, pateikti konkretūs jo taikymo pavyzdžiai.