UDC 004.43

Program Comprehension for Domain-Specific Languages[*]

Maria João Varanda Pereira[1], Marjan Mernik[2], Daniela da Cruz[3] and Pedro Rangel Henriques[3]

[1]Polytechnic Institute of Bragança
Campus de Sta. Apolónia, Apartado 134 – 5301-857, Bragança, Portugal
mjoao@ipb.pt
[2]University of Maribor
Faculty of Electrical Engineering and Computer Science
Smetanova ul. 17, 2000 Maribor, Slovenia
marjan.mernik@uni-mb.si
[3]University of Minho - Department of Computer Science,
Campus de Gualtar, 4715-057, Braga, Portugal
{danieladacruz,prh}@di.uminho.pt

Abstract. In the past, we have been looking for program comprehension tools that are able to interconnect operational and behavioral views, aiming at aiding the software analyst to relate problem and program domains in order to reach a full understanding of software systems. In this paper we are concerned with Program Comprehension issues applied to Domain Specific Languages (DSLs). We are now willing to understand how techniques and tools for the comprehension of traditional programming languages fit in the understanding of DSLs. Being the language tailored for the description of problems in a specific domain, we believe that specific visualizations (at a higher abstraction level, closer to the problem level) could and should be defined to enhance the comprehension of the descriptions in that particular domain.

Keywords: program understanding, problem comprehension, DSLs.

[*] This work is supported by the grant for Slovenia-Portugal Cooperation in Science and Technology, SLO-P-11/01-04: Program Comprehension for Domain-Specific Languages

1. Introduction

Domain-Specific Languages (DSLs) ([1–10]) are languages tailored to specific application domains and offer users more appropriate notations and abstractions. By definition, DSLs are more expressive and easier to use than General-Purpose Languages (GPLs) for the domain in question, with corresponding gains in productivity, and reduced maintenance costs. Some specific goals of DSLs such as:

- to make programming more accessible to end-users;
- to improve correctness of the written programs, and
- to improve the program developing time,

seem to follow on implicitly from the DSL definition. But, have these claims really been proved in practice? All the above claims have a common denominator in the assertion that *DSL programs are easier to comprehend*.

Therefore, in the project *Program Comprehension for DSLs* we have the following objectives:

- to measure how much easier it is to use (learn, develop, evolve) and understand DSLs when compared to GPLs,
- to ascertain whether if existing program comprehension approaches, techniques, or even tools are applicable to DSLs, and
- to allow the enhancement of DSL program comprehension tools by enabling user-centric visualization.

Program Comprehension (PC) [11, 12] is a hard cognitive task that involves constructing a mental model of the program and trying to reconstruct the thoughts of the original programmer. This process becomes easier when concrete representations are automatically produced, revealing different aspects of the program's structure and behavior. Hence, *program visualization and program animations are important aids* for accomplishing this task. Even more important, is the ability to create visual representations that allow the programmer to interconnect the execution of program statements with the effect produced by them; thus allowing visualization of the relation between problem and program domains.

We discuss in the paper how this generic assertion—which is the basis for PC in the context of traditional programming languages—can be more adequately exploited within the context of DSLs.

The paper is organized as follows. In Section 2, the Cognitive Dimensions Framework (CDF), used to assess visual languages, is briefly introduced and its application to study the usability of DSLs is discussed. Existing approaches and techniques for the Program Comprehension of GPLs are shortly revisited in Section 3, where we also discuss their reuse in building specific tools to help in the comprehension of DSL programs. A description of user-centric visualization (the concept and a possible realization) follows the Section 4. Finally, two examples that illustrate our user-centric visualization idea are included in Section 5. The paper is concluded in Section 6.

2. Using and Understanding DSLs

Cognitive Theory ([13–16]) provides some guidelines on how to measure a human's ability to program. *Cognitive Dimensions Framework* [17] (or CDF for short) provides cognitively-relevant aspects which can be used to determine how easy it is to *learn* the language, *develop*, *evolve*, and comprehend a program.

These cognitive dimensions, included in the referred CDF, are:

– Closeness of mapping - languages should be task-specific,
– Viscosity - revisions should be painless,
– Hidden dependencies - the consequences of changes should be clear,
– Hard mental operations - enigmatic constructions should not be allowed,
– Imposed guess-ahead - user should not be obliged to premature commitment,
– Secondary notation - languages should allow for encompassing additional information,
– Visibility - search trails should be short,
– Consistency - user expectations should not be broken,
– Diffuseness - language should not be too verbose,
– Error-proneness - notation should inherently catch mistakes avoiding errors,
– Progressive evaluation - the user should get immediate feedback,
– Role expressiveness - the relationships among components should be clearly seen,
– Abstraction gradient - languages should allow different abstraction levels.

These cognitive dimensions framework has been used to assess the usability of visual programming languages [17–19], while no such study exists for DSLs. One of the main goals of the proposed project is to study how easy it is to use DSLs and comprehend the respective programs by preparing a set of questions guided by those dimensions. Those questions will then be used to compare DSL programs with the equivalent GPL ones.

Below are some speculations concerning the gain afforded by the use of DSLs which will be proved.

Closeness of mapping refers to how wide the semantic gap is between the problem and solution spaces. It was shown in the study [20] that plenty of low-level primitives, which are often purely syntactical, is one of the biggest cognitive barriers for end-user programmers. In this regard DSLs can outperform GPLs. On the other hand, an experienced programmer comprehends program not just as a series of statements but as a structure of components working together. In other words, a programmer needs to understand operations, and data structures, as well as program structure. Therefore, it is important to study whether end-users have problems when composing components together within DSL programs. They might understand primitives well, but have difficulties putting pieces together.

Viscosity refers to how much effort is needed to perform small changes. It is somehow surprising that visual programming languages have high viscosity yet the opposite is true for textual languages. This is due to spatial relationships in visual languages, where even a small change requires to rearrange several components. Since many DSLs are textual, we expect them to perform well in this dimension too; the high abstraction level and usual simplicity of those languages should also have a positive influence in this dimension, inducing low viscosity.

Hidden dependencies refer to the interactions among program components (short and long-range) that are not immediately visible. Changing one part might have an undesirable effect on the other parts of the program. Textual languages often suffer from severe hidden dependency problems (e.g., side effects, aliasing, ...). An open question is: *can hidden dependencies be avoided in DSLs by proper design*?

Hard mental operations refer to points in the program where the programmer needs to think hard in order to understand it or even needs additional tools. Another open question deserving further investigation is: *can DSL be free of such hard mental operations*?

Imposed guess-ahead refers to situations where the programmer is forced to make a decision before he has the information he needs. This often happens when there are a lot of internal dependencies, when constraints on the ordering exist, or when inappropriate notation is used.

Secondary notations refers to the possibility of using other mechanisms (e.g., grouping, positioning, commenting, interleaving code of a different language) to convey important information about the code. Some studies [17] show that textual languages allow a substantial amount of secondary notation; and what about DSLs: do they usually allow the use of secondary notations?

Visibility refers to code which can be directly accessible without additional cognitive work. The simple measure would be the number of steps to make a given component visible. Textual languages usually have better visibility than visual languages. This is true if programs are relatively short. However, it is necessary to research whether this is the case for DSLs.

Consistency refers to the ability to infer the rest of the language from current incomplete knowledge of it. This much depends on proper language design rather than on differences among GPLs and DSLs. Anyway, DSL has fewer concepts and such language property might be easier to achieve.

Diffuseness refers to the number of symbols needed to express the meaning. By definition, DSLs use existing domain notation which should be at an appropriate level of verbosity, so it is expected that they exhibit low diffuseness.

Error proneness refers to the capability of a language to induce ’careless mistakes’. GPLs, due to their extension and intrinsic complexity, are usually error-prone. We conjecture that this problem can be overcome in the context of DSLs due to the narrow domain they are designed for; usually a DSL is much smaller and simpler, so mistakes are more unlikely to happen. This intuition also requires deep study to be proved.

Progressive evaluation refers to the ability to test an incomplete program. No general statement can be made about this ability concerning DSLs as it depends completely on the domain and mainly on the philosophy underlying the language design. So we can say that this is another open item for further investigation.

Role expressiveness refers to the ability to see how each component of a program relates to the whole. The high role expressiveness can be more easily achieved in DSLs due to domain specifics and shorter programs.

Abstraction gradient refers to the minimum and maximum levels of abstraction. DSLs might suffer from the problem that raising abstraction level to the point where end-users are unable to handle them, since hidden dependency might be bigger.

Some of these cognitive dimensions (e.g., hidden dependencies, hard mental operations, secondary notation, visibility, and role expressiveness) can be enhanced by DSL program visualization and program comprehension tools. This topic is discussed in the rest of the paper.

3. Comprehension Tools for DSLs

The second objective of the work under discussion is to identify the precise needs in terms of information and visualization for DSL program comprehending, in order to know if the existing approaches, techniques and tools for the comprehension of GLP programs can be reused. Of course, this investigation will lead to the development of aid tools. Just as happens with program understanding tools, the tools for Domain Specific Program Comprehension (DsPCTools) have to extract and display static or dynamic data about a program, in order to help the analyst understand its structure and behavior.

In the context of the research described here, the first task is to identify information that would be useful for comprehension and that must be extracted from the source program. This stage is specific and should be worked out from the beginning.

Then, we search for suitable approaches (methods and techniques) to extract, and store this information. According to our background on program comprehension ([21–24]), we are convinced that existing PC techniques can be used for DSLs.

We have some experience with two different approaches: on one hand, we have developed an animator that does not modify the source program and uses abstract interpretation techniques, aimed at an easy and systematic adaptation to cope with different programming languages; on the other hand, for the development of other PC tools we have applied a technique called *program instrumentation* that modifies the source code (inserting *inspector functions*) in order to collect dynamic information at runtime.

In the first case, the source program is not compiled and so: variables are not converted into memory locations; algebraic operations are not transformed into register operations involving value-transfers among memory addresses; control flow into jumps to code addresses; and input/output into read/write operations on files. Instead, we work with *abstractions of program concerns*—such as *assignment, algebraic operations, conditions for controlling the execution flow, input/output*, etc.); then we interpret those abstractions (no assembly code is executed).

Concerning the second approach, we have expertise in weaving *inspectors* in the source program to catch and record the functions that are actually called during execution and their concrete parameters (or in a Web context, the program units that are interpreted by the server, or the links really visited).

The development of both approaches—abstract interpretation and code instrumentation—relies completely on traditional grammar-oriented techniques for compiler writing and implementation. We use **Translation Grammars** or **Attribute Grammars** [25] to specify the tools, and resort to **Compiler Generators** for automatically producing the code of the target processors. As DSLs processing is also completely supported by grammars technology we sustain the statement above that PC techniques are reusable in this specific context.

Techniques to visualize and navigate over the information so far collected—which constitutes the third step in this work—may also be inherited from generic PC approaches. This intuition comes directly from the above referred evidence—the same internal representation is usable for both contexts.

What should then be tuned specifically for each domain is the visual representations to be employed by the visualizers in order to make the perception easier and clearer. When conceiving visual representations to display the static or dynamic data extracted from programs written in GLPs, it is impossible to choose icons or drawings which are too expressive for the sake of generality; moreover and given such a broad range of application areas, it is fairly difficult to find systematic and generic ways to graphically represent the problem domain adequately. In contrast, we hope that, working with DSLs, we can obtain total profit from the inherent speciality, in order to look for expressive and adequate visual representations for each domain.

Concerning the implementation of such a strategy, we suggest to follow Alma's approach [26]. Alma is a *system for program visualization and animation* that deals easily with different programming languages and allows for the construction of more appropriate visualizations for each domain. The purpose of this tool is to help the programmer to inspect *data* and *control flow* for a given program (*static view* of the algorithm realized by the program — **visualization**), and to understand its behavior (*dynamic view* of the algorithm — **animation**).

The core of such tool is language independent; it is similar to a compiler's Back-End (BE) that takes as input an abstract representation—as intermediate representation, between the FE and the BE, we use a *Decorated*

Abstract Syntax Tree (DAST)—and implements the visualizer and the animator components in a systematic way. This is achieved by means of two rule bases, one for the visualization of tree nodes, and another for tree rewriting.

To process a concrete programming language, Alma is customized by providing a dedicated Front-End (FE) that converts the input programs into internal abstract representation.

Concerning the characteristics of each particular DSL, we are aware that we need to study as many cases as possible to understand whether the *specific language concepts and constructions* require definition and inclusion in our internal representation of new abstraction, or even adaptation of their operational semantics.

However for all the cases worked out until now, the abstractions provided by the original DAST were sufficient. The next section introduces how to apply ALMA's approach in the implementation of a user-centric program comprehension tool.

4. User-centric Comprehension Tools for DSLs

As previously stated, there are many different DSLs (focused on different targets and following different styles). DSLs can have a more procedural (imperative) style or follow a more declarative one. In the procedural case, those languages describing data and operations over data; can be considered as very similar to the GPLs. Declarative DSLs usually describe high-level specifications, data or activity models, etc.; in this case, it makes no sense at all to analyze the descriptions written in that DSL from an operational point of view, because typically they do not have an associated execution model.

It means that any direct influence of the language itself during the comprehension process needs further investigation. In this section we discuss how to explore domain specific property in order to enhance the visual representation to be used by comprehension tools.

As previously stated, the semantic gap between the problem and program domains is much smaller in the DSLs context. Program and problem comprehension can be achieved easily because it is easier to visualize a conceptual mapping between both. Within the problem domain level, visualizations deeply depend on this domain. The big challenge in this direction relies precisely on the fact that a DSL has special characteristics that imply a deeper study concerning the kind of visualizations that are more appropriate for each case.

So, it would also be useful to construct visualization tools where end-users, not language designer or developer, can easily specify their own visualization (End-user programming: [27–29]).

In this section, we expose one solution which is based on visualization/animation system Alma [30]. The main idea is to build a graphical editor. The graphical editor will enable the end-user to associate each node of the DAST with a geometric figure (a square, circle, etc), or an image; and also, it will enable the end-user to associate each node with an external (end-user defined) drawing function. This will permit the creation of specific drawings parameterized to fit well in each particular DSL. The external function will be called using the attributes available in the DAST nodes to tune the picture to each concrete situation, as illustrated in the next Section for the Robot example (see Section 5.1) - a parameterized external function is necessary to show the Robot movements in the room.

We can include that functionality, keeping the tree visualizer engine generic and unchanged; and also the animator system, based on a tree rewriting engine, is kept unchanged.

This approach is quite easy to implement and will grant to the visualizer/animator system, customized for a concrete DSL, effective improvement and better quality as an aid tool for understanding specifications/programs written in that specific language.

5. Illustrating User-centric Comprehension Tools for DSLs

In this section, and aiming at illustrating the ideas proposed, we introduce two DSLs and show the visualizations that should be created by generic DsPCTools enhanced with the user-centric approach, in order to refine problem domain visualization.

Controlling a Robot, a first example of DSL

In this section we take, as an example, a program that controls the movements of a cleaning robot. Let us assume that *Roby* is a small robot whose mission is to clean a rectangular area; a grid is used for quick referencing the robot's position (line 0 is the top, and column 0 is the leftmost). Roby can move straight-ahead up, down, right and left, a given number of steps (one step corresponds to one grid square).

To control *Roby*, we use a simple DSL that basically allows us to choose the direction and length of each straight movement in order to, sequentially, compose its activity. The program below is written in the robot control language (RCL); after setting the start position as the upper left corner (the square with coordinates 0,0), we define its cleaning path as 3 steps down, 7 steps right, 2 steps up, and 4 more steps left, before stopping:

```
xi= 0
yi= 0
DOWN 3
RIGHT 7
UP 2
LEFT 4
```

By aiming at making a clear distinction between the abstraction levels of the operational (at program domain level) and behavioral (at problem domain level) views, and willing to clarify how each one contributes for the program understanding, the purpose of this example is to produce two different views from the same input program.

Fig. 1 is a screen-shot obtained after executing the last statement in the program; it corresponds to the complete animation scenario, exhibiting the final state. This is an operational view.

Another possible animation is shown in Fig. 2; notice that in this case only the last visualization is shown (the path, or the intermediate robot positions are kept). This one is more abstract and shows the effect produced by the program over the robot.

To produce this behavioral view, the visualization doesn't show anymore, variables and operations; instead, they are now concerned with displaying the external objects controlled by the program.

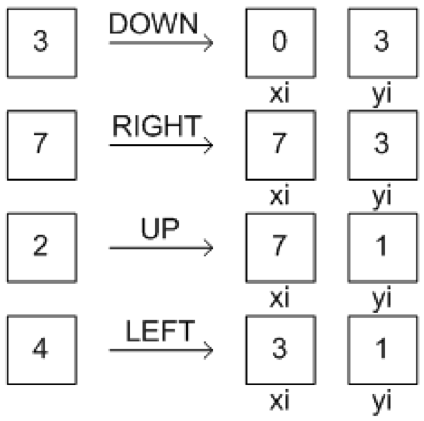


Fig. 1. Robot Operational Animation

The interesting, and perhaps difficult, point is to understand what the relevant attributes are. In this example, it is clear that what we need to draw the robot in each cleaning position.

The robot example is a typical case where it is more useful to inspect the object's evolution (behavioral view) than the program behind it (operational view). However both play an important role in the program comprehension process. In our opinion, visualization of these two views makes possible the relationship between the two different domains and follows Brooks theory [11] of a complete mental representation of a program. The DAST shown in Figure 3 is obtained from the program listed above, using a map between the robot language and DAST nodes. The assignment statements (in the robot control language) were represented by *ASSIGN* nodes, and the movement statements (defining the direction and number of steps) were represented by LST nodes.

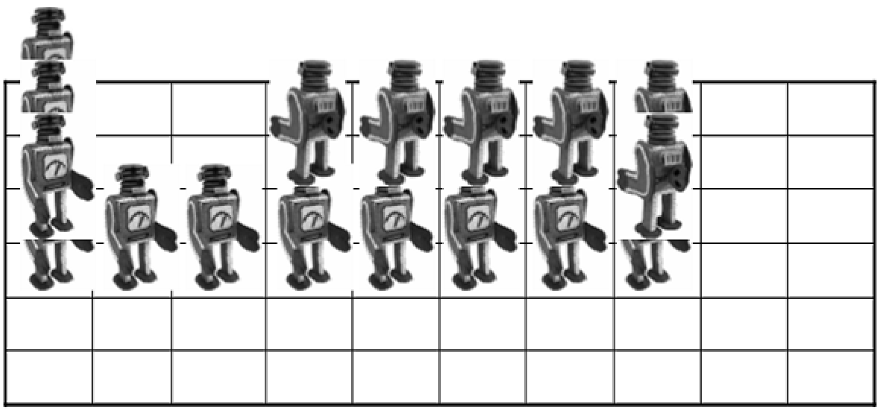


Fig. 2. Robot Animation

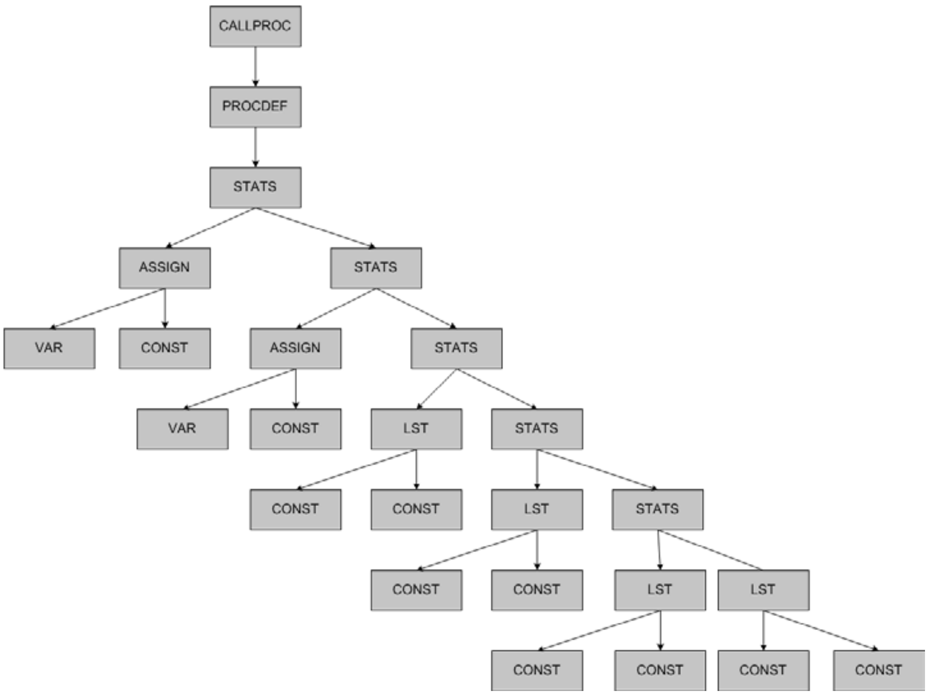
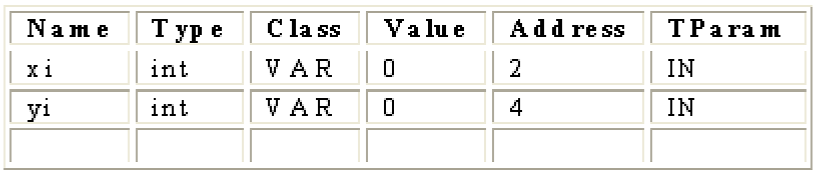


Fig. 3. DAST generated for the robot example

Then we can assign rewrite and visualization rules to the nodes of the DAST in order to create a visual representation, and animation; these rules make use of the semantic information stored in the Identifier table shown in Fig. 4.



Name	Type	Class	Value	Address	TParam
xi	int	VAR	0	2	IN
yi	int	VAR	0	4	IN

Fig. 4. Identifier Table containing the Name, Type, Class, Value and memory Address

of the program variables and the parameter kind (TParam={IN, OUT, INOUT})

The rewrite rules used in this example evaluate the robot coordinates after the execution of each movement statement, looking up, once again, on the Identifier Table. The written form of each rewrite rule is as follows:

```
rew_rule(idProd)= <t: tree-pattern>,
            (cond: condition),
            <newProdId: idProd : newtree: tree-pattern>,
            {eval: attribute_evaluation}

<tree-pattern>=<root, child_1, ..., child_n>
```

In this template, `cond` is a boolean expression (by default, evaluates to `true`) and `eval` is a sequence of statements to compute the new attribute values (the default action is skip). Below is the rewrite rule associated with a production, named `plst`, of RCL grammar:

```
rew_rule(plst)=<l:LST, a:CONST, b:CONST>,
               (getvalue(b)!=NULL),
               <l:LST, a:CONST, b:CONST>,
               {x=getTableVal(xi);
                y=getTableVal(yi);
                calculate(x,y,getValue(a),getValue(b))
                putTableVal(xi,x);
                putTableVal(yi,y);}
```

The written form of each visualization rule is as follows:

```
vis_rule(idProd)= <t: tree-pattern>,
                  (cond: condition),
                  {dp: drawing_procedure}
```

In this template, `cond` is a `boolean` expression (by default, evaluates to true) and `dp` is a sequence of one or more calls to elementary drawing procedures.

A set of visualization rules, like the one listed below, originates the picture shown in Fig. 1. The purpose of such a set of rules is to display the robot command, its parameter, and the new values for `xi` and `yi`.

```
vis_rule(plst)=   <l:LST, a:CONST, b: CONST>,
                  (),
                  {drawRect(a.name,a.value);
                  draw_arrow(b.name);
                  drawRect("xi",getTableVal(xi));
                  drawRect("yi",getTableVal(yi));}
```

We can modify the abstraction level of the visualization produced, associating new visual rules (drawing different kinds of pictures) to different nodes. The idea is to create new visualization rules in order to associate more abstract drawings to upper tree nodes (nodes corresponding to high level grammar symbols). Concerning this example (Roby cleaning task) we want to use new visualization rules to be able to inspect the object behavior (Roby walking along the cleaning area), instead of the behavior of the control program.

By using the same rewrite rule (above defined to evaluate Roby's coordinates), and a new visualization rule (listed below and now associated with the RCL grammar named `pprocdef`), that draws the robot in different positions, we will obtain the picture shown (overlapped) in Fig. 2.

```
vis_rule(pprocdef)= <a:PROCDEF, b:STATS>,
                    (),
                    {drawrobot(getValue(xi), getValue(yi))}
```

The function `drawrobot` draws the robot in those coordinates. This function could also use other parameters in order to put different robot images depending on its direction as we can see in Fig. 2.

FDL - Feature Description Language

As a second example, we chose FDL, a *Feature Description Language* introduced in [31] aiming at the description of objects in knowledge domains. The following sentence is an example of a FDL description:

```
car: all(carBody,Transmission,Engine,HorsePower,pullsTrailer?)
Transmission: one_of (automatic, manual)
Engine: more_of (electric, gasoline)
HorsePower: one_of (lowPower, mediumPower, highPower)
```

The specification above describes a car in terms of its parts. A car is composed by other features. Each of these features can be atomic or composite. This DSL has a set of operators: `all`, `one_of`, `more_`of and `?` for optional features. This FDL specification will be translated to an internal representation. Then, a set of visualization rules will be applied in order to generate dual views.

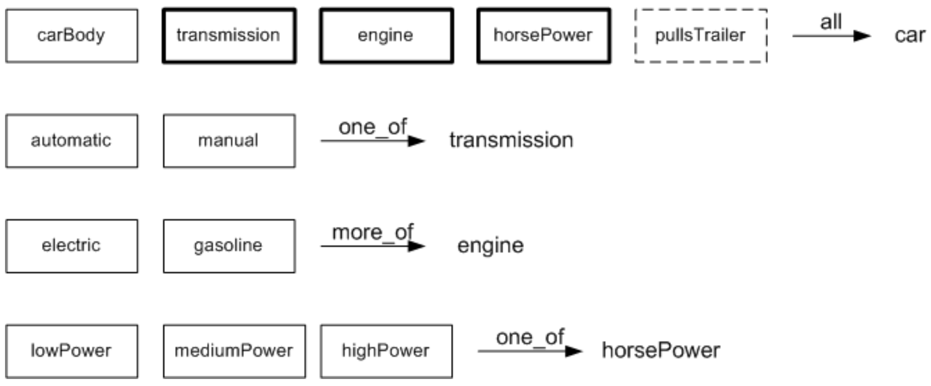


Fig. 5. FDL operational view

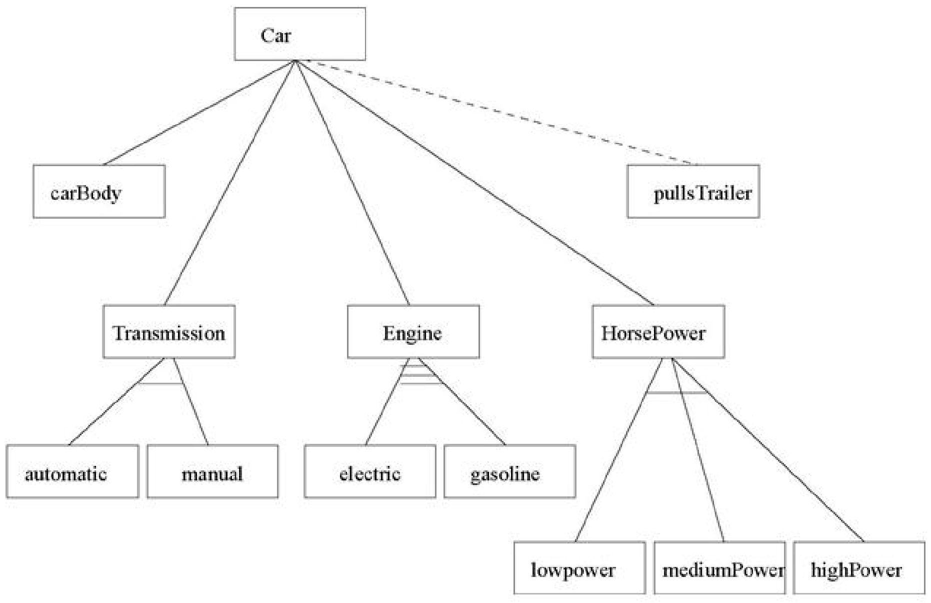


Figure 6. FDL diagram

Fig. 5 shows the *operational view*. This kind of view can be constructed using visualization rules associated to lower level nodes of the DAST.

The *behavioral view* is shown the Fig. 6. Here we use a visualization rule associated with the root of the syntax tree. The main idea is to represent the object defined by the specification visualizing its behavior and checking the attribute values in the identifier table. In this case, a FDL diagram can be generated emphasizing the relationship between entities.

6. Conclusions

This paper introduces the three main research directions of our ongoing bilateral (Portugal/Slovenia) project on Program Comprehension for DSLs (named DSLpc).

We started the paper by briefly revisiting the definition of DSL, and analyzing its actual impact. We stated that this concept implies that a DSL program should be more concise, natural and clearer than the equivalent solution (to solve the same problem) expressed in a GPL. This perspective leads directly to our first concern in this project: to understand and measure how easy it is to use (learn, develop, and evolve) and comprehend programs written in DSLs; this task, similar to a language usability assessment, is hard but it should be done. We will carry out that study using direct observation, and questionnaires to measure the user comprehension of DSL and GPL descriptions (this requires the preparation, application and analysis of appropriate inquiries).

The three main components of a Program Comprehension tool were revisited.

Then we affirm that standard approaches to deal with GPLs can be reused with DSLs. Our second goal in this project is precisely concerned with proving the statement above. This second task will also identify the particular information needs in the context of DSL comprehension.

The third direction of our research will focus in the enhancement of DSL program comprehension tools, by enabling user-centric visualization. We exposed a concrete solution improving Alma, a visualization/animation system, with extra functionality to allow the user to specify for each particular DSL, the visual representation he wants to apply.

7. References

1. Mernik, M., Heering, J., Sloane, T.: When and how to develop domain-specific languages. ACM Computing Surveys 37(4) (2005) 316 -344
2. Kosar, T., Lopez, P.M., Barrientos, P.A., Mernik, M.: A preliminary study on various implementation approaches of domain-specific language. Inf. Softw. Technol. 50(5) (April 2008) 390–405
3. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. SIGPLAN Not. 35(6) (June 2000) 26–36
4. Wile, D.: Lessons learned from real DSL experiments. Sci. Comput. Program. 51(3) (2004) 265–290
5. Wile, D.: Supporting the DSL spectrum. J. Comput. Inform. Techn. 9(4) (2001) 263–287
6. Bentley, J.: Little languages. Communications of the ACM 29(8) (1986) 711–721

7. Kamin, S.N.: Research on domain-specific embedded languages and program generators. Electronic Notes in Theoretical Computer Science 14 (1998)
8. Thibault, S.: Domain-specific languages: Conception, implementation and application. PhD thesis, University of Rennes (1998)
9. Klint, P., Laemmel, R., Verhoef, C.: Towards an engineering discipline for grammarware. ACM Transactions on Software Engineering and Methodology 14(3) (2005) 331–380
10. Spinellis, D.: Notable design patterns for domain-specific languages. Journal of Systems and Software 56(1) (2001) 91–99
11. Brooks, R.: Using a behavioral theory of program comprehension in software engineering. In: ICSE '78: Proceedings of the 3rd international conference on Software engineering, Piscataway, NJ, USA, IEEE Press (1978) 196–201
12. Storey, M.A.: Theories, methods and tools in program comprehension: Past, present and future. In: 13th International Workshop on Program Comprehension (IPWC'05). (2005)
13. Pane, J.F., Myers, B.A., Miller, L.B.: Using HCI techniques to design a more usable programming system. (2002) 198–206
14. Blackwell, A.F.: Ten years of cognitive dimensions in visual languages and computing: Guest editor's introduction to special issue. J. Vis. Lang. Comput. 17(4) (2006) 285–287
15. Petre, M.: Cognitive dimensions "beyond the notation". Journal of Visual Languages and Computing 17(4) (2006) 292–301
16. Clarke, S., Becker, C.: Using the cognitive dimensions framework to measure the usability of a class library. In: Proceedings of the First Joint Conference of EASE PPIG (PPIG 15) (2003)
17. Green, T., Petre, M.: Usability analysis of visual programming environments: a 'cognitive dimensions' framework. Journal of Visual Languages and Computing 7(2) (1996) 131–174
18. Yang, S., Burnett, M., DeKoven, E., Zloof, M.: Representation design benchmarks: a design-time aid for VPL navigable static representations. Journal of Visual Languages and Computing 8(5/6) (1997) 563–599
19. Burnett, M.: Visual programming. Encyclopedia of Electrical and Electronics Engineering (1999)
20. Lewis, C., Olson, G.: Can principles of cognition lower the barriers to programming? In: 2nd workshop on Empirical Studies of Programmers. (1987)
21. da Cruz, D., Henriques, P.R., Pereira, M.J.V.: Strategies for program inspection and visualization. In: CSE'08 - International Scientific Conference on Computer Science and Engineering, High Tatras, Slovakia (September 2008)
22. Sim, S.E., Storey, M.A.: A structured demonstration of program comprehension tools. In: Seventh Working Conference on Reverse Engineering (WCRE'00), Brisbane, Australia (November 2000)
23. Ko, A.J., Uttl, B.: Individual differences in program comprehension strategies in unfamiliar programming systems. In: 11th IEEE International

Workshop on Program Comprehension (IWPC'03), pages 175–184, Portland, Oregon,USA (May 2003)
24. Maletic, J.I., Marcus, A.: Supporting program comprehension using semantic and structural information. In: 16th IEEE International Conference on Automated Software Engineering (ASE2001), San Diego - USA, IEEE (November 2001) 107– 114
25. Knuth, D.E.: The genesis of attribute grammars. In: WAGA: Proceedings of the International conference on Attribute grammars and their applications, New York, NY, USA, Springer-Verlag New York, Inc. (1990) 1–12
26. da Cruz, D., Henriques, P.R., Pereira, M.J.V.: Constructing program animations using a pattern-based approach. ComSIS – Computer Science an Information Systems Journal, Special Issue on Advances in Programming Languages 4(2) (Dec2007) 97–114 ISSN: 1820-0214.
27. Nardi, B.A.: A small matter of programming: perspectives on end user computing. MIT Press (1993)
28. M. Burnett, C.C., Rothermel, G.: End-user software engineering. Communications of the ACM 48(9) (2005) 53–58
29. Sutcliffe, A., Mehandjiev, N.: End-user development: Tools that empower users to create their own software solutions. Communications of the ACM 47(9) (2004) 31–32
30. Pereira, M.J.V., Henriques, P.: Visualization / animation of programs in Alma: obtaining different results. In: VMSE2003 - Symposium on Visual and Multimedia Software Engineering (HCC'03), New Zealand, IEEE (October 2003) 260–262
31. van Deursen, A., Klint, P.: Domain-specific language design requires feature descriptions. Journal of Computing and Information Technology 10(1) (2002) 1–17

Maria João Varanda Pereira received the M.Sc. and Ph.D. degrees in computer science from the University of Minho in 1996 and 2003 respectively. She is currently an Adjunct Professor at the Polytechnic Institute of Bragança in the Informatics and Communications Department. Her research interests include programming languages, compilers, grammar-based systems, visual languages, program comprehension, animation systems and domain specific languages.

She was responsible for PCVIA (Program Comprehension by Visual Inspection and Animation), a FCT funded national research project. She is involved in bilateral cooperation projects with Slovenia since 2000.

Marjan Mernik received the M.Sc. and Ph.D. degrees in computer science from the University of Maribor in 1994 and 1998 respectively. He is currently a professor at the University of Maribor, Faculty of Electrical Engineering and Computer Science. He is also an adjunct professor at the University of Alabama at Birmingham, Department of Computer and Information Sciences.

His research interests include programming languages, compilers, grammar-based systems, grammatical inference, and evolutionary computations. He is a member of the IEEE, ACM and EAPLS.

Daniela da Cruz received a degree in "Mathematics and Computer Science", at University of Minho, and now she is starting a Ph.D. degree in "Computer Science" also at University of Minho, under the MAPi doctoral program. She joined the research and teaching team of "gEPL, the Language Processing group" in 2005. She is teaching assistant in different courses in the area of Compilers and Formal Development of Language Processors; and Programming Languages and Paradigms (Procedural, Logic, and OO). As a researcher of gEPL, Daniela is working with the development of compilers based on attribute grammars and automatic generation tools. She developed a compiler and a virtual machine for the LISS language (an imperative and powerful programming language conceived at UM).
She was also involved in the PCVIA (Program Comprehension by Visual Inspection and Animation), a FCT funded national research project; in that context, Daniela worked in the implementation of "Alma", a program visualizer and animator tool for program understanding. She is now enrolled in a new bilateral cooperation project with Slovenia under the subject "Program Comprehension for Domain Specific Languages".

Pedro Rangel Henriques got a degree in "Electrotechnical/Electronics Engineering", at FEUP (Oporto University), and finished a Ph.D. thesis in "Formal Languages and Attribute Grammars" at University of Minho. In 1981 he joined the Computer Science Department of University of Minho, where he is a teacher/researcher. Since 1995 he is the coordinator of the "Language Processing group". He teaches many different courses under the broader area of programming: Programming Languages and Paradigms (Procedural, Logic, Functional and OO); Compilers and Formal Development of Language Processors; etc. He is co-author of the "XML & XSL: da teoria à prática" book, publish by FCA in 2002. Pedro Rangel Henriques has supervised M.Sc. (16) and Ph.D. (14) thesis, and more than 100 graduating trainingships/projects, in the areas of: language processing (textual and visual), and structured document processing; program animation and program comprehension; knowledge discovery from databases, data-mining, and data-cleaning. He also was responsible for several applicational projects (in the interface university/external-community, industry), mainly in the area of information systems (databases and web oriented). From 2002 until 2004 he was the Head of the Department, and at moment he is the President of APPIA, the Portuguese Association for Artificial Intelligence.

Received: July 16, 2008; Accepted: November 17, 2008.