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.

1 Introduction

Domain-specific languages (DSLs) [1] are languages tailored to specific application domain and offer to users more appropriate notations and abstractions. By definition, DSLs are more expressive and are 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,

* 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.

– to improve correctness of the written programs, and
– to improve the program developing time.

seems to follow implicitly from the DSL definition. But, were these claims really proved in practice? All the above claims have 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 easier is to understand programs written in DSLs,
– to understand if existing program comprehension approaches and techniques or even tools are applicable to DSLs, and
– to allow the enhancement of DSL program comprehension by enabling user-centric visualization

Program Comprehension (PC) [2, 3] is a hard cognitive task that involves the construction of a mental model of the program, 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 structure and behavior. Hence, *program visualization and program animations are important aids* in accomplishing that task.

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

The paper is organized as follows. In Section 2, cognitive dimensions framework is briefly discussed and applied to DSLs. Existing techniques and approaches for program comprehension are briefly introduced in the Section 3, where we also discuss their reuse to build specific tools to help in the understanding of DSL programs. A description of user-centric visualization (the concept and a possible realization) follows in the Section 4. Finally, two examples that illustrate our user-centric visualization idea are included in the Section 5. The paper is concluded in the Section 6.

2 Understanding DSL Programs

Cognitive Theory provides some guidelines how to measure human's ability to program. *Cognitive dimensions framework* [4] provides cognitively-relevant aspects which can be used to determine how easy is to learn the language, developing a program, evolve a program, and comprehend a program. These cognitive dimensions 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 - no enigmatic is allowed,
– Imposed guess-ahead - no premature commitment,
– Secondary notation - allow to encompass additional information,

– Visibility - search trails should be short,
– Consistency - user expectations should not be broken,
– Diffusiness - language should not be too verbose,
– Error-proneness - notation should inherently catch mistakes avoiding errors,
– Progressive evaluation - get immediate feedback,
– Role expressiveness - see the relations among components clearly,
– Abstraction gradient - as simple as possible, but not simpler.

This cognitive dimensions framework has been used to assess the usability of visual programming languages [4–6], while no such study exists for DSLs. Below are some speculations which still need to be proved. This is one of the main goals of the proposed project.

Closeness of mapping refers to how wide is the semantic gap between the problem and solution space. It was shown in the study [7] 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 should outperformed GPLs. On the other hand, experienced programmer comprehends program not just as a series of statements, but as a structure of components working together. In other words, programmer needs to understand operations, data structures as well as program structure. Therefore, it is important to study if end-users have a problems with composing components together in the 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 surprisingly that visual programming languages have high viscosity and opposite is true for textual languages. This is due to spatial relationships in visual languages where even small change require to rearrange several components. Since, many DSLs are textual we expect that they will perform well on this dimension, too.

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

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

Imposed guess-ahead refers to the situation where programmers are forced to make a decision before they have the information they need. This often happened when there are a lot of internal dependencies, when constraints on the ordering exist, or when inappropriate notation is used. Textual languages typically suffer from this problem on structure level, while visual languages on layout level.

Secondary notations refers to the ability that programmers can use other mechanisms (e.g., grouping, positioning, commenting) to convey other important information about the code (e.g., which part belong together, or are most likely

to be changed in the next version, or which part is thoroughly tested). Some studies [4] show that textual languages allow a substantial amount of secondary notation, while in visual languages are rather deficient.

Visibility refers to code which can be directly accessible without additional cognitive work. The simple measure would be the number of steps to make to 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 if this is the case for DSLs.

Consistency refers to ability to infer the rest of the language from current incomplete knowledge of the language. 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.

Diffusiness refers to number of symbols which are needed to express the meaning. By definition, DSLs are using existing domain notation which should be at appropriate level of verbosity.

Error proneness refers to ability 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 harder to happen. This intuition also requires a deep study to be proved.

Progressive evaluation refers to ability to test 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 ability to see how each component of a program relates to the whole. Again, we expect that high role expressiveness can be more easily achieved in DSLs due to domain specifics and shorter programs.

Abstraction gradient refers to minimum and maximum level of abstraction. DSLs might suffer from the problem that raising abstraction level to the point where end-users are not able to handle since hidden dependency might be bigger.

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

3 Program Comprehension for DSL Programs

The second objective of the work under discussion is to identify the precise needs in terms of information and visualization to comprehend DSL programs, in order to know if the existing approaches and techniques for the comprehension of GLP programs can be reused. Of course, this investigation will led the development of aiding 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 to help the analyst to understand its structure and behavior.

In the context of the research here described, the first task will be 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 since the beginning.

Then, we need to search for suitable approaches (methods and techniques) to extract, and store that information. According to our background on program comprehension, we think that existing PC techniques can be used for DSLs.

We have some experience with two different approaches: on one hand, we developed an animator that does not modify the source program and uses *abstract interpretation* techniques, aiming at an easy and systematic adaptation to cope with different programming languages; on the other hand, in the development of other PC tools we have applied a technique called *program instrumentation* that modifies the source code (inserting *inspector functions*) to be able 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 of that, we work with *abstractions of program concerns*—like *assignment*, *algebraic operations*, *conditions* to control 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—completely rely on traditional grammar-oriented techniques for compiler writing and implementation. We use Translation Grammars or Attribute Grammars to specify the tools, and resort to Compiler Generators to automatically produce the code of the desired processors. As DSLs processing is also completely supported on grammars technology we sustain the statement above that PC techniques are reusable in that 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. That intuition comes directly from the evidence above referred—the same internal representation is usable for both contexts.

What should then be tuned specifically for each domain are 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 chose icons or drawings too expressive for the sake of generality; moreover and given the so broad range of application areas, it is fairly difficult to find system-

atic and generic ways to graphically represent adequately the problem domain. On the other way around, we hope that, working with DSLs, we can take total profit of the inherent speciality, to look for **expressive and adequate visual representations for each domain**.

Concerning the implementation of such strategy, we think that we can rely upon the approach followed in Alma [8]—a *system for program visualization and animation* that deals easily with different programming languages and allows the construction of the most 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 specialized providing a dedicated Front-End (FE) that converts the input programs into that internal abstract representation.

Concerning the characteristics of each particular domain specific Language, we are aware that we need to study as many cases as possible to understand if the *specific language concepts and constructions* require the definition and inclusion in our internal representation of new abstraction, or even the adaptation of their operational semantics. However for all the cases worked out until know, we could survive with the abstractions provided by the original DAST.

In next section, we are going to explain how to apply ALMA's approach in the implementation of an user-centric program comprehension tool.

4 User-centric Program Comprehension Tools

As told above, 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 describe data and operations over data; we can consider them very similar to the general purpose programming languages. The 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 execution model associated.

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

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

So, it would be also useful to construct visualization tools where end-users, not language designer or developer, can easily specify their own visualization—problem and person specific.

In this section we propose something that can be done in this direction; to be concrete, we will now take into account our visualization/animation system Alma, above referred. We plan to build a graphical editor two fold: On one hand it will provide to the end-user the chance to associate to each node of the DAST a geometric figure (a square, circle, etc), or an image. One the other hand we provide to the end-user the chance to associate to each node an external (end-user defined) drawing function. This will permit to build 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 will be illustrated in the next Section for the Robot example (see 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; also the animator system, based on a tree rewriting engine, will be kept unchanged.

This proposal does not seem to be difficult to implement and will grant to the visualizer/animator system, customized to a concrete DSL, an effective improvement and a better quality as an aiding tool to understand specifications/programs written in that specific language. We strongly believe that this idea would be an actual contribution in this field.

5 Some Examples of DSPC enhancement

In this section, and aiming to illustrate the ideas proposed, we introduce two DSLs and some show the visualizations that should be generated by the respective DSPCTools.

5.1 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 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 language that basically allows us to choose the direction and length of each straight movement to, sequentially, compose its activity. The program below is written in that robot control language; after setting the start position as the left upper 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
```

Aiming to make a clear distinction between the abstraction levels of the operational and behavioral views, and willing to clarify how each one contributes for the program understanding, the purpose of this example is to produce from the same input program two different views.

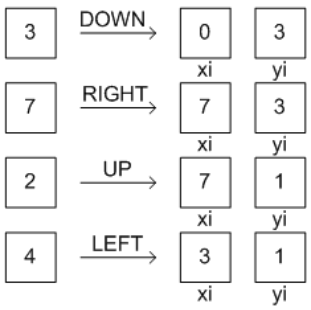

Fig. 1. Robot Operational Animation

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

Other 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 last 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, any more, variables and operations; instead, they are now concerned with the display of the external objects controlled by the program.

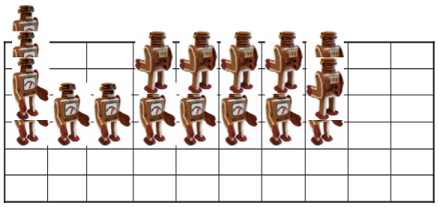

Fig. 2. Robot Animation

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

The robot example is a typical case where it is more useful to inspect the object evolution (behavioral view) than the program behind it (operational view). However both play an important role in program comprehension process. In our opinion, the visualization of these two views make possible the relationship between the two different domains and follows Brooks theory [2] of a complete mental representation of a program.

5.2 FDL - Feature Description Language

As a second example, we chose FDL, a *Feature Description Language* introduced in [9] 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. 3 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. 4. 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.

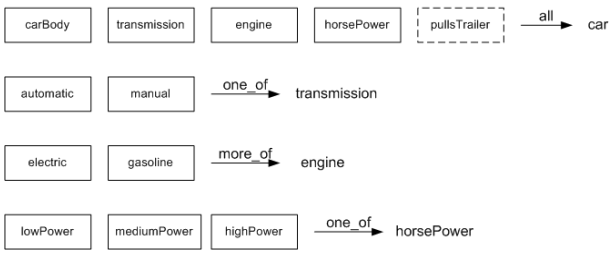

Fig. 3. FDL operational view

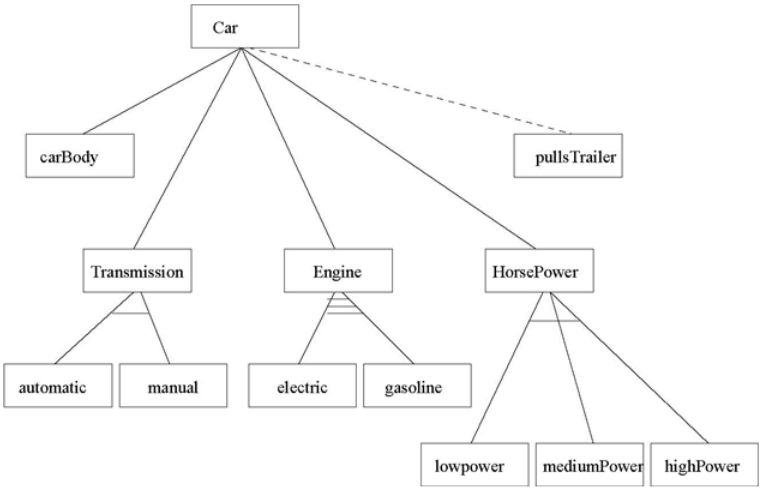

Fig. 4. FDL diagram

6 Conclusion

In this paper we have introduced the three main directions of our new bilateral (Portugal/Slovenia) project for joint research.

We started the paper discussing the definition of Domain Specific Language (DSL), and analyzing its actual impact. We showed that this concept implies that a DSL program should be more natural and clearer than the equivalent solution (to solve the same problem) expressed in a General Purpose Language (GPL). This perspective leads directly to our first concern in this project: to understand and measure how easy is to understand programs written in DSLs; this task, similar to the usability assessment, is not simple but it should be done. We will do that study under controlled and non-controlled programming environments, 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).

After that, we remembered the three main components of a program comprehension tool, and we have affirmed that standard approaches to deal of GPLs could be reused for DSLs. Our second goal in this project is precisely concerned with proving the statement above.

The third direction of our research will focus in the enhancement of DSL program comprehension tools, by enabling user-centric visualization. We made a concrete proposal to improve the program understanding tool 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 .

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. 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
3. Storey, M.A.: Theories, methods and tools in program comprehension: Past,, present and future. In: 13th International Workshop on Program Comprehension (IPWC'05). (2005)
4. Green, T., M.Petre: Usability analysis of visual programming environments: a 'cognitive dimensions' framework. Journal of Visual Languages and Computing **7**(2) (1996) 131–174
5. 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
6. Burnett, M.: Visual programming. Encyclopedia of Electrical and Electronics Engineering (1999)
7. Lewis, C., Olson, G.: Can principles of cognition lower the barriers to programming? In: 2nd workshop on Empirical Studies of Programmers. (1987)

8. 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) (Dec 2007) 97–114 ISSN: 1820-0214.
9. van Deursen, A., Klint, P.: Domain-specific language design requires feature descriptions (2002)