Visualization of Domain-Specific Programs' Behavior

Nuno Oliveira
University of Minho
Braga, Portugal
nunooliveira@di.uminho.pt

Maria João Varanda Pereira
Polythecnic Institute of Bragança
Bragança, Portugal
mjoao@ipb.pt

Pedro Rangel Henriques
University of Minho
Braga, Portugal
prh@di.uminho.pt

Daniela da Cruz
University of Minho
Braga, Portugal
danieladacruz@di.uminho.pt

Abstract

Program domain concepts are rather complex and low level for a fast assimilation. On the other hand, problem domain concepts are closer to human's mind, hence they are easier to perceive. Based on Brook's theory, a full comprehension of a program is only achieved if both domains are connected and visualized in synchronization, resulting on an action-effect *visualization.*

Domain-specific languages, as languages tailored for a specific class of problems, raise the abstraction of the program domain concepts and approximate them to the problem domain's. This way, a systematic approach can be used to perform the action-effect *visualization of a program written in a domain-specific language. In this paper, we use a domain-specific language to exemplify how the concepts involved in both domains are visualized and how it is possible to map each problem domain situation (depicted by images) to the program domain operations.*

1 Introduction

The visualization of a program is commonly performed at the program domain level. Such a visualization is made using execution trees, dependency graphs, call graphs and other similar artifacts [6]. But these are approaches somewhat distant from the needs of a person who relies on visualization to understand a program. Human brain needs more than that to be capable of mapping the relations between how the program executes internally (program domain level) and what are the effects produced, externally, by such execution (problem domain level).

A synchronized visualization of these two aspects would give a more interesting and intuitive perspective on the program being visualized. Program domain concepts are at a lower level when compared with the concepts of the problem domain. Such visualization would not set them at the same level, nonetheless it would gather their synergy to provide an *action-effect* point of view.

When dealing with a *Domain-Specific Language* (`DSL`), these domains are closer to each other [2]. This means that the program domain concepts are at a much higher level than in a *General-purpose Programming Language* (`GPL`). Then, it is easier to infer the problem domain from the language definition and to create mappings between the concepts of both domains [5]. The approach for the visualization of *Domain-Specific Programs* (`DSP`)[1], proposed in this paper, is based on the creation of mappings between both domains. We use `Alma` system [1] to construct the visualization of the source code, based on the formal definition of the language and traditional grammar-based techniques; then we extend `Alma` to cope with the problem domain visualization. The latter is, in fact, the association of figures of one or more objects that depict a situation in the problem domain, with the concepts of the program domain.

Looking to the literature, we found many papers on program comprehension tools for `GPL`s [7], and some references to proposals and projects about `DSL` debuggers [3]. However we did not find any work directly related to program comprehension for `DSL`s.

In the reminder of this paper we present an example of a `DSL`, describe its specific problem domain, formalize the language, and build the mapping between the program and problem concepts (Section 2). Then we show some figures about the synchronized visualization of both domains, (Section 3). Finally we draw some conclusions in Section 4.

[1] To programs written in a `DSL`, we call `DSP`.

2 Karel, an Example

Karel programming language was designed by Pattis [4]. It is a language to control a small robot, called *Karel*[2], in a small virtual world. The robot is neither a full-featured nor a sophisticated machine. Besides turning on or off, moving one step ahead, turning left, picking objects from the ground, keeping them in an object bag, and putting them back on the ground, *Karel*, the robot, knows (i) which direction it is facing to; (ii) whether it is blocked by walls or even (iii) whether it sees objects on the ground. The tasks this robot can perform fit inside the boundaries of handling objects from a place to another.

The robot only understands a few basic instructions, hence, the language to control it is simple as can be noticed in Listing 1, where we list a fragment of this DSL[3].

Listing 1. Formal Definition of Karel DSL

```
1
2  start       → BEGINNING–OF–PROGRAM program
3                END–OF–PROGRAM
4  program     → definition* BEGINNING–OF–EXECUTION
5                statement* END–OF–EXECUTION
6  definition  → DEFINE–NEW–INSTRUCTION identifier
7                AS statement
8  statement   → block | iteration | loop
9                | conditional | instruction
10 block       → BEGIN statement* END
11 iteration   → ITERATE number TIMES statement
12 loop        → WHILE condition DO statement
13 conditional → IF condition THEN statement
14               (ELSE statement)?
15 instruction → TURNON | MOVE | TURNLEFT
16               | PICKBEEPER | PUTBEEPER
17               | TURNOFF | identifier
18 condition   → FRONT–IS–CLEAR
19               | ...
20 identifier  → [a–z] ([a–z] | [0–9]+)*
```

2.1 Domains Interconnection

To achieve a visualization of the problem domain, the concepts related to the behavior of the controlled object (the robot in this case) must be intrinsically associated with some parts of the syntax and semantics of the DSL.

To conceptually do such a connection we need to retrieve the most important concepts resident in both domains. In problem domain we can identify the following concepts: $i)$ *turn off*, $ii)$ *turn on*, $iii)$ *turn left*, $iv)$ *step ahead*, $v)$ *pick object* and $vi)$ *drop object*. From the program domain we retrieve: $i)$ TURNON, $ii)$ MOVE, $iii)$ TURNLEFT, $iv)$ PICKBEEPER, $v)$ PUTBEEPER and $vi)$ TURNOFF. As we are dealing with a very abstract DSL the mapping of concepts is trivial. Each row in Table 1 represents a connection between the concepts at both domains.

[2]The robot inherited its name from the inventor of the word and concept *robot*: Karel Čapek, a well-known Czech writer and playwright.

[3]The original grammar is available at http://mormegil.wz.cz/prog/karel/prog_doc.htm

Table 1. Concepts effective connection

PROBLEM DOMAIN	PROGRAM DOMAIN
Turn Off	TURNOFF
Turn On	TURNON
Step Ahead	MOVE
Turn Left	TURNLEFT
Pick Object	PICKBEEPER
Drop Object	PUTBEEPER

2.2 Problem Domain Visualization

At this moment, the connection of domains was done. However no steps on the visualization of the problem domain were given. In our visualization technique, the expert should define a set of images that represent the objects as well as the situations involved in the domain. When these situations require movement, it should be defined animation sequences to represent them.

In the Karel DSL, great part of the situations are depicted by sequences of poses of the controlled object. Figure 1 shows the several poses that the robot can hold.

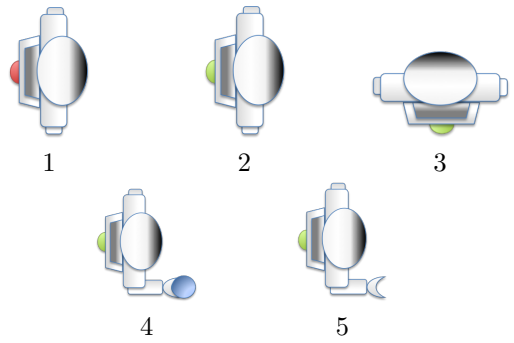

Figure 1. Possible poses for Karel, the Robot

With these *poses* we can define the visualization and animation of the problem domain, by associating the images with the concepts of the problem domain. In Table 2 we present the mapping between the poses (represented by the number of the images in Figure 1) and the concepts at the problem domain level.

Obviously the visualization/animation may require more artifacts than just images. For instance, for the *Step Ahead* concept, we define the sequence: image 2 to image 2; but, in fact, what should change in this sequence is the position of the image. Another example is the concept *Turn Left*: image 3 is the same as image 2, only rotated 90^o to the left. This kind of aspects define the mappings between the images and the DSL's semantics at program domain level.

Table 2. Behavioral Visualization Definition

PROBLEM DOMAIN	POSES SEQUENCE
Turn Off	$2 \rightarrow 1$
Turn On	$1 \rightarrow 2$
Step Ahead	$2 \rightarrow 2$
Turn Left	$2 \rightarrow 3$
Pick Object	$2 \rightarrow 5 \rightarrow 4 \rightarrow 2$
Drop Object	$2 \rightarrow 4 \rightarrow 5 \rightarrow 2$

In the end, there is a threesome connection that involves the problem domain concepts, the images and the program domain concepts.

3 Synchronized Visualization

In the previous section we conceived the behavioral visualization of Karel programs. Under the hood, we used the extended version of `Alma` system to automatically generate operational and behavioral views of the input program[4]. In this section we show some results of the visualization, in `Alma`, of the Harvest Program[5]. Listing 2 shows a fragment of this program.

Listing 2. Sample of the Harvest Program

```
1
2  BEGINNING–OF–PROGRAM
3  DEFINE–NEW–INSTRUCTION turnright AS
4            ITERATE 3 TIMES
5               TURNLEFT
6  DEFINE–NEW–INSTRUCTION harvest AS
7            ITERATE 3 TIMES
8               BEGIN
9                  PICKBEEPER
10                 MOVE
11              END
12 BEGINNING–OF–EXECUTION
13           TURNON
14           turnright
15           MOVE
16           harvest
17           (...)
18 END–OF–EXECUTION
19
20 END–OF–PROGRAM
```

`Alma` gives us, for free, a visualization of the program domain. The views offered are the Interpretation Tree, the Identifier Table and the Source Code. The Interpretation Tree correspondent to the program under study is depicted in Figure 2. By Interpretation Tree we mean a tree that is a static/dynamic semantic representation of the input program, either in an imperative or declarative language. Usually in the literature it is named *execution tree*.

[4]The explanation of how we do this in `Alma` is out of this paper's scope.

[5]See description at: `http://www.cs.mtsu.edu/~untch/karel/functions.html#style`

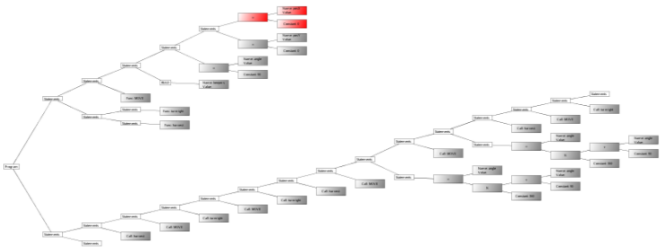

Figure 2. Interpretation Tree View

Figure 3 shows some examples of the behavioral visualization of the Harvest Program in `Alma`. The first image occurs when the program is initialized, the second occurs after executing the `turnright` instruction and the third after the execution of the `move` instruction. Finally, the fourth is one frame of the animation sequence defined for the *pick object* concept, and which occurs in the context of the `harvest` instruction.

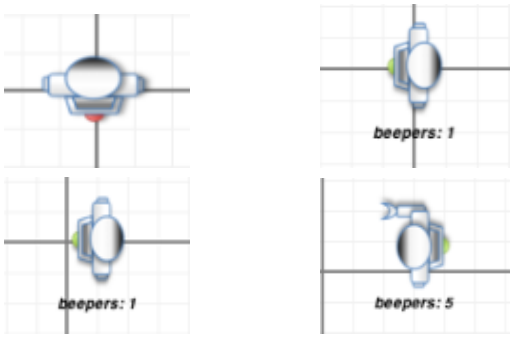

Figure 3. Karel Poses in `Alma`

The tree, the identifier table and the source code views are always synchronized for a better understanding at program level. With the extension made to `Alma`, we have not only those three views synchronized with each other, but also synchronized with the new perspective which depicts, in every step of the program interpretation, the effects that the operations provoke in the real world concepts.

Figure 4 is the confirmation of what we stated in the last paragraph. The four perspectives of the program visualization are synchronized. We can see the program and problem animation together in each step of the program visualization process.

4 Conclusion

In this paper, we briefly described our visualization technique that is still under research by presenting some of the

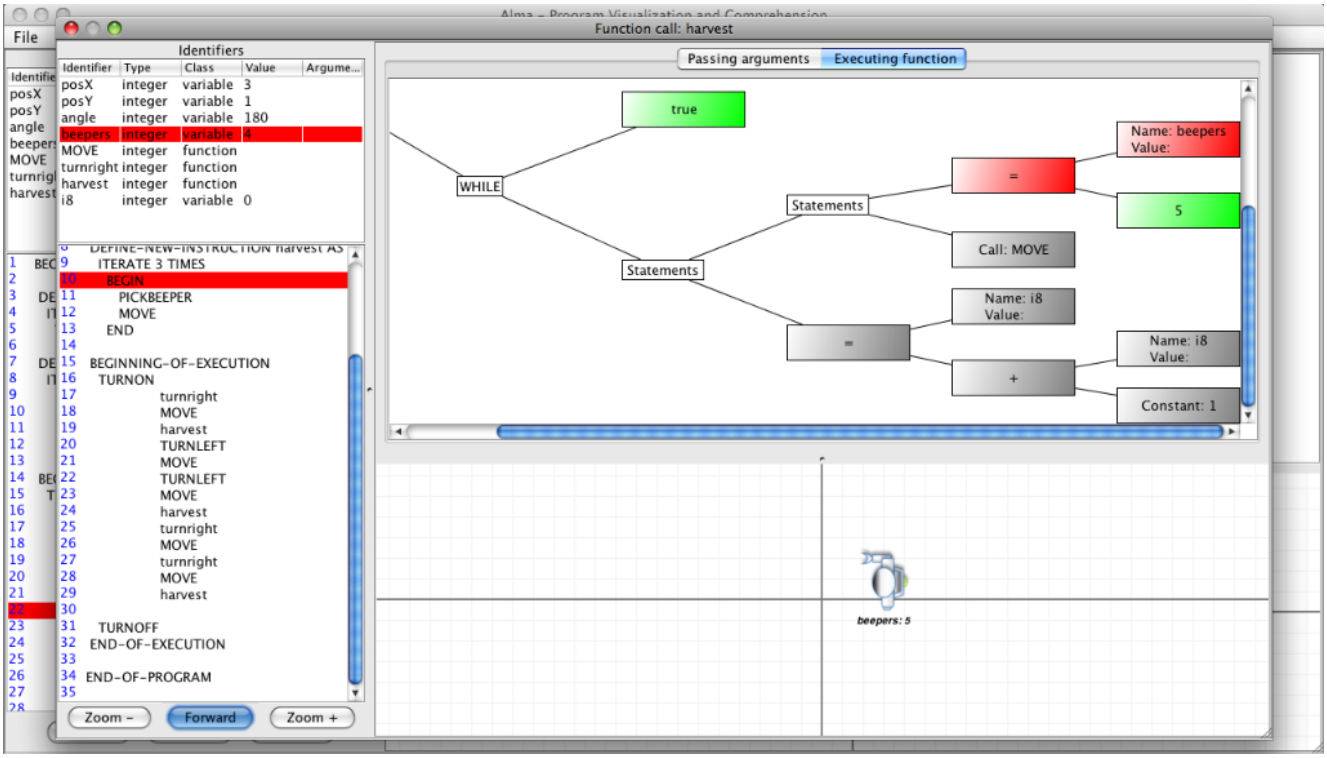

Figure 4. Complete view of the synchronization among the several visualization perspectives

results that were already achieved. From the example reported we learned that: i) the connection between program and problem domains can be achieved effortless when resorting to domain-specific languages; ii) the connection between both domains not only allows a bigger intuition about how the program does, but it also gives a more sophisticated view: the behavior of the program and the changes it provokes in the domain (what the program does); iii) the synchronization of the program and problem visualization resources is a reality and can be attained using traditional grammar-based approaches.

With this experiment, we reinforce the idea that a program is only fully comprehended when the domains are connected and visualized in synchronization. This example was developed using `Alma` system and all visualizations were constructed over an abstract representation of the source program (Karel programs in this case). This fact turns the tool easily adaptable to cope with other languages.

In the future, the consistency of the tool should be tested with more and varied domain-specific languages. These studies should also corroborate the approach we are defending. Also, it would be imperative to assess, with final users, the effectiveness of our belief.

References

[1] D. da Cruz, P. R. Henriques, and M. J. Varanda. Constructing program animations using a pattern-based approach. *ComSIS – Computer Science an Information Systems Journal, Special Issue on Advances in Programming Languages*, 4(2):97–114, December 2007.

[2] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

[3] T. Mészáros and T. Levendovszky. Visual specification of a DSL processor debugger. In *Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling*, pages 67–72, Nashville, USA, 2008.

[4] R. Pattis. *Karel, The Robot: A Gentle Introduction to the Art of Programming*. John Wiley and Sons, Inc., 1st edition, 1981.

[5] M. J. V. Pereira, M. Mernik, D. da Cruz, and P. R. Henriques. Program comprehension for domain-specific languages. In *CoRTA08 — Compilers, Related Technologies and Applications, Bragança, Portugal*, July 2008.

[6] M. Pinzger, K. Grafenhain, P. Knab, and H. C. Gall. A tool for visual understanding of source code dependencies. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 254–259, 2008.

[7] M.-A. Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, 14(3):187–208, 2006.