Proceedings of the International Multiconference on Computer Science and Information Technology pp. 699–706
ISBN 978-83-60810-22-4
ISSN 1896-7094

Applying Program Comprehension Techniques to Karel Robot Programs

Nuno Oliveira*, Pedro Rangel Henriques*, Daniela da Cruz*, Maria João Varanda Pereira†, Marjan Mernik‡, Tomaž Kosar‡ and Matej Črepinšek‡

* University of Minho - Department of Computer Science,
Campus de Gualtar, 4715-057, Braga, Portugal
Email: {nunooliveira, prh, danieladacruz}@di.uminho.pt
† Polytechnic Institute of Bragança
Campus de Sta. Apolónia, Apartado 134 - 5301-857, Bragança, Portugal
Email: mjoao@ipb.pt
‡ University of Maribor, Faculty of Electrical Engineering and Computer Science,
Smetanova 17, 2000 Maribor, Slovenia
Email: {marjan.mernik, tomaz.kosar, matej.crepinsek}@uni-mb.si

***Abstract*—In the context of program understanding, a challenge research topic[1] is to learn how techniques and tools for the comprehension of General-Purpose Languages (GPLs) can be used or adjusted to the understanding of Domain-Specific Languages (DSLs). Being DSLs tailored for the description of problems within a specific domain, it becomes easier to improve these tools with specific visualizations (at a higher abstraction level, closer to the problem level) in order to understand the DSLs programs.**

In this paper, comprehension techniques will be applied to Karel language. This will allow us to explore the creation of problem domain visualizations for this language and to combine both problem and program domains in order to reach a full understanding of Karel programs.

I. Introduction

IN THIS paper, we explore the use of program comprehension techniques to understand Domain-Specific Programs (DSPs). By DSP [1], [2], [3] we mean programs written in a Domain-Specific Language (DSL), which in its turn, is designed to program specific tasks in a fixed problem domain. To program in DSLs means to use a specific vocabulary, structures, and higher level components. Moreover, to implement this kind of languages, specific tools can be constructed and they are customized for a problem domain. These facts make the programming task easier in this specific context, but difficult to understand by people that are out of the subject.

We are convinced that we can apply traditional Program Comprehension Techniques to DSLs [4], [5], and we can go further constructing visualizations closer to the problem domain. This is possible because, from a DSL program, we can easily infer information about the problem to be solved.

[1]This work is part of a bilateral cooperation project (Portugal/Slovenia) supported by FCT, Departamento das Relações Europeias, Bilaterais e Multilaterais, and Slovenian Research Agency (grant No. BI-PT/08-09-008).

The construction of program comprehension tools can be based on the formal definition of the language and, in our case, their development relies completely on traditional grammar-oriented techniques. Using the grammar, we can generate automatically textual or visual editors, to create and handle programs in that language. In a similar way, we can also generate parsers (generally speaking, language processors) to extract from the source code static and dynamic information to create visualizations helpful to understand it.

In the context of General-Purpose Languages (GPLs) the information about the problem to be solved, collectable from the code, is neither sufficient to infer the object that is controlled, nor the *problem domain* — perceiving what kind of control can be programmed. Under those circumstances, it is necessary to resort to other kind of resources like annotations, comments, user manuals, implementation reports, and so forth. However, from the definition of DSL comes out that, when dealing with such languages, we know the objects operated by the programs; thus it is possible to construct problem domain visualizations changing the object states according to dynamic data extracted from the source code.

In this case we have information about the object, the problem domain (the operations over the object), and the program domain (the instructions that modifies the object state). The mapping of these views improves the efficiency of program comprehension tools.

The outline of this paper is as follows: in Section II the related work about PC techniques and tools are presented; the application of these techniques to DSLs will be described in Section III; along Section IV we present the processes of extracting, visualizing, and synchronizing the information of different domains for Karel Language [6]; finally the conclusion of the paper is in Section V.

II. Program Comprehension, Techniques and Tools

Program Comprehension (PC) [7], [8] is a hard cognitive task that involves constructing 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 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.

Program Comprehension plays an important role in the area of software maintenance, as it is a complex and expensive task. Thus, the need for software engineering tools that facilitate the process of understanding computer programs is compelling. In this context, the main goal of a Program Comprehension Tool (PCT) is to ease the process of understanding the structure and functionality of a program. In this field of PC, many tools were developed along the last 20 years. Imagix 4D [9], CodeSurfer [10], Shrimp [11], CANTO [12], CodeCrawler [13] and Bauhaus [14] are only a few tools among many others. All the tools comply with the referred objective by: providing one or more known mental models for program comprehension; maintaining a repository of structural and/or behavioral information about a program; providing a presentation model for visualizing information about programs in various ways; providing mechanisms for navigating from one kind of representation to another; and so forth.

According to our background on program comprehension, we are convinced that existing PC techniques can be used for DSLs. We have some experience with two different approaches [15]: a non-invasive approach (the source code does not change) and an invasive approach (it changes the source code).

Concerning the first one, we have developed an animator, Alma [16], 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. Concerning the second one, we have applied it in the development of two other tools, CEAR [17] and WAV [18], a technique called program instrumentation that modifies the source code (inserting inspector functions) in order to collect dynamic information at runtime. In Alma, the source program is not compiled. Variables are not converted into memory locations, algebraic operations are not transformed into register operations involving value transfers among memory addresses, and control flow is not implemented as jumps to code addresses. Instead, we work with abstractions of program concerns (such as assignment, algebraic operations, conditions for controlling the execution flow, input/output, and so forth) and interpret them (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 (in the context of web applications, the program units that are interpreted by the server, or the links really visited).

The development of both approaches—abstract interpretation and code instrumentation—rely completely on traditional grammar-oriented techniques for compiler writing and implementation. We use Translation Grammars or Attribute Grammars [19] to specify the tools, and resort to Compiler Generators for automatically produce the code of the target processors.

III. Our Approach to Apply PC Techniques in DSLs

Although there are several approaches that we could follow to implement our ideas, due to our acquaintanceship with Alma, we have decided to adhere to its philosophy. In this context we extended it to deal with the ideas expounded.

Alma [16] is a system for program visualization and animation that deals easily with different programming languages and allows 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 Front-End (FE) and the BE, we use a Decorated Abstract Syntax Tree (DAST) and implement 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 one for tree rewriting. To process a concrete programming language, Alma is customized by providing a dedicated FE that converts the input programs into the internal abstract representation.

Besides that reconfiguration of Alma, to cope with different input languages, at present we propose another evolution of Alma to Alma^2, a PC tool tuned to cope with a given DSL.

That evolution relies on the use of a second base of visualizing rules, synchronized with the first one and with the tree rewriting system. This new set of visualizing rules is adapted to each DSL and is responsible for producing the problem domain view.

Concerning the characteristics of each particular DSL, a set of animation rules must be defined and the inclusion in our internal representation of new

abstractions or even adaptation of their operational semantics must be done. This will prepare the tool for the final user that just have to insert a source program in order to get the visualizations. On the other hand, since each DSL has special characteristics, we need to perform a deeper study concerning the kind of visualizations that are more appropriate for each case.

In our research project, several DSLs will be studied but we have started the work with Karel Programming Language, and this paper is devoted to report the outcomes so far attained.

IV. Comprehending Karel Programs

In the previous section we gave an overview of the approach we conceive for the development of a program comprehension tool for DSLs. In this section, we show how we use Alma[2] to help on the comprehension of programs written in Karel Language.

Karel Language [6], is a DSL to control a robot, called *Karel*[2]. As the language also has the academic purpose of teaching the bases of imperative programming, 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 in the ground.

To sum up, the robot only understands a few basic instructions, hence, its controlling language is simple as can be noticed in Listing 1, where a version of Karel language grammar is presented[3]. Notice, though, that the language only specifies the robot actions, and it does not concern the modeling of the world where the robot *lives*.

Listing 1. Formal Definition of Karel Language

```
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 AS
7                   statement
8  statement      → block | iteration
9                   | loop | conditional
10                  | instruction
11 block          → BEGIN statement* END
12 iteration      → ITERATE number TIMES statement
13 loop           → WHILE condition DO statement
14 conditional    → IF condition THEN statement
15                  (ELSE statement)?
16 instruction    → TURNON | MOVE | TURNLEFT
17                  | PICKBEEPER | PUTBEEPER
18                  | TURNOFF | identifier
19 condition      → FRONT–IS–CLEAR | FRONT–IS–BLOCKED
20                  | LEFT–IS–CLEAR | LEFT–IS–BLOCKED
21                  | RIGHT–IS–CLEAR | RIGHT–IS–BLOCKED
22                  | BACK–IS–CLEAR | BACK–IS–BLOCKED
23                  | NEXT–TO–A–BEEPER
24                  | NOT–NEXT–TO–A–BEEPER
25                  | ANY–BEEPERS–IN–BEEPER–BAG
26                  | NO–BEEPERS–IN–BEEPER–BAG
27                  | FACING–NORTH | NOT–FACING–NORTH
28                  | FACING–SOUTH | NOT–FACING–SOUTH
29                  | FACING–EAST | NOT–FACING–EAST
30                  | FACING–WEST | NOT–FACING–WEST
31 identifier     → [a–z] ( [a–z] | [0–9]+ )*
32 number         → [0–9]+
```

[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

A. Knowledge Analysis

Regarding the language description and its formal definition, we can infer some knowledge about the program and problem domains, and we also can create a set of connections between them, to ease the comprehension of the target program.

In Karel language, looking to its description it is not difficult to conclude that it is used to control some kind of robot. From the formal definition, we suspect how to control that robot. In other cases, some extra documentation should be consulted.

However, as long as this *machine* has no brain to think on its movements, we can infer that there is an internal state that is changed by the sequence of operations allowed by the controlling language. This means that the language does not control the robot directly, instead it controls its internal state. This is what really happens at the program level. But persons, who try to understand the programs in this language, may be interested not only in what happens internally, at the robot's state, but also in what are the effects produced, externally, in the robot.

Alma[2] purpose is precisely that: to give a joint view of what is done at program level, and what are the repercussions at problem (real-world) level. So, we have to define the visualization of both domains. Program level visualization requires the creation of the program's state (the robot's internal state) and the definition of the *interpretation tree*[4]. Problem level visualization requires the definition of images that depict situation on that domain; it needs also the creation of connections with language operations and constructions. These connections will make possible a synchronized visualization of both domains, enabling an inspection of what are the program actions that produce the effect (movement) on the robot. Finally, it requires the construction of the animation, resorting to the images and the mappings created.

Again from the language description and its formal definition, we can infer concepts that define the robot's state. Table I shows these concepts, to which we call variables.

Also from the grammar of Karel Language and the description of the domain, incremented with the empirical knowledge about the controlled object (the robot), we can infer the situations (poses of the robot) illustrated in Figure 1. This figure is composed of five

[4]By interpretation tree we mean an attribute valued (decorated) abstract syntax tree that is a static/dynamic semantic representation of the input program, either in a imperative or declarative language. Usually in the literature it is named *execution tree*.

TABLE I
KAREL'S INTERNAL STATE VARIABLES

VARIABLE	DESCRIPTION
posX	Stores the X-axe value of the robot's position.
posY	Stores the Y-axe value of the robot's position.
angle	Stores the angle of the robot's direction.
beepers	Stores the number of objects the robot has in its bag.

images. Each one represent an upper view of the robot in a different situation: 1) the robot is turned off (red light in its back); 2) the robot is turned on (green light in its back); 3) the robot rotated left; 4) the robot picked an object and 5) the robot dropped an object. These images would be combined with each other to perform animations directed by the operations at the program level. This is an issue that will be addressed in Section IV-C.

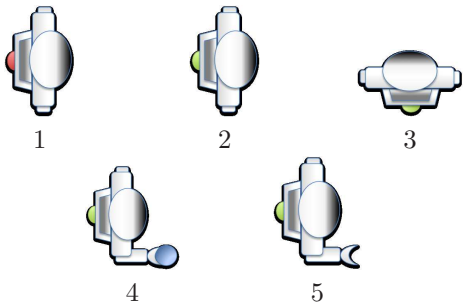

Fig. 1. Problem Domain Situations: Karel Possible Poses

In the next Section IV-B, we will center attentions in the definition of the program domain visualization.

B. Visualizing the Program Domain

When writing a program with Karel Language, the user does not need to be worried about the definition of a state for the robot, because such state should already be defined by the compiler/processor. The delegation of these *tasks* (and other semantic definitions) to the compiler, is a very common practice when dealing with DSLs. As found in [20], the same does not happen with GPLs, where the state of the program is defined in the program itself. This way, as the BE of `Alma`^{2} is an interface for the processing of a language, we must define the operational semantics of Karel Language.

We create an `Alma`^{2} FE for Karel Language, in order to convert Karel programs into `Alma`^{2} tree-based abstract representation. LISA system [21], based on attribute grammars, is used to construct the FE for Karel. We start by implementing this FE by defining the variables and nodes that will be able to describe the robot's internal state. Then, as a second step, for each Karel instruction, we create DAST nodes, resorting to `Alma`^{2} notation. The functions and objects of `Alma`^{2} are implemented in Java. LISA is used to synthesize an attribute that will store the complete DAST of a program. This DAST, representing the internal and abstract structure of a program, is built resorting to the constructs defined in `Alma`^{2}. In some extent, `Alma`^{2} can be seen as an domain-specific embedded language [22].

Listing 2. Definition of the Robot's State (Fragment)

```
1
2  CToken tX = new CToken("posX", ...);
3  CToken tY = new CToken("posY", ...);
4  CToken tA = new CToken("angle", ...);
5  CToken tB = new CToken("beepers", ...);
6      (...)
7
8  public Alma.CAlmaNode initState(){
9      CConstNode c0 = new Alma.CConstNode(0);
10     CAlmaNode nX = new AssignNode(tX, c0);
11     CAlmaNode nY = new CAssignNode(tY, c0);
12     CDeclNode x =
13        new CDeclNode(tX, "integer", null, nX)
               ;
14     CDeclNode y =
15        new CDeclNode(tY, "integer", null, nY)
               ;
16     CAlmaNode decl1=new Alma.CStmtsNode(x, y)
           ;
17
18     (...)
19 }
```

The code fragment in Listing 2 shows how we declare the variables that determine the state of the robot, and initialized the position. The idea is to $i)$ define global tokens (line 2 to 5) that would be used in whole grammar to refer to the variables `posX`, `posY`, `angle` and `beepers`, respectively; $ii)$ then we create an auxiliary function, `initState`, that builds the nodes of a branch with the variable's declaration and initialization (lines 8 to 19). The fragment, in Listing 2, builds a tree equivalent to the tree that would represent a piece of an imperative language program like:

$$integer\ posX = 0,\ posY = 0;$$

The tree resultant from the auxiliary function presented in Listing 2 is prepended to the reminder of the tree synthesized when processing a program in Karel Language. In Listing 3 we show another fragment of the Karel Language processor. This time, we illustrate the construction of the tree representation and semantics behind the command PICKBEEPER.

The command PICKBEEPER hides, in its abstraction, a small operation that modifies the state of the robot, namely, it increments the number of beepers. So, when including this command in a program, at interpretation phase we must consider the program as having one more statement equivalent to:

$$beepers = beepers + 1;$$

As the other instructions in Karel Language have a similar abstraction level, the interpreter of each one requires a similar approach, adding statements to

Listing 3. Definition of the Program Domain Visualization (Fragment)

```
1
2  rule Instruction_PickBeeper {
3    INSTRUCTION ::= #Pickbeeper compute {
4    INSTRUCTION.tree =
5              new AssignNode(
6                    tB,
7                    new COperNode(
8                          new CVarNode(tB),
9                          new CConstNode(1), "+"
10                   )
11             );
12   };
13 }
```

change the state. For instance, the instruction TURNLEFT, changes the value of the variable `angle` in the following way:

$$angle = (angle + 90)\%360$$

A concrete illustration of a sub-tree from the program's DAST, can be seen ahead in this document, in Figure 3.

C. Visualizing the Problem Domain

As stated before, to build the visualization of the problem domain, the first step is to create connections between problem and program concepts — with them we would be able to see which parts of the program affect the produced output (at problem domain); and, as a last step, to define the animation of images (depicting situations of the problem) according to the mappings created.

Since the problem domain underlying the Karel Language was known, we are able to infer the chief concepts characterizing the problem domain (see Table II, first column); from the program domain we identify the main operations (see Table II, second column).

TABLE II
MAPPING PROGRAM AND PROBLEM CONCEPTS

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

With the contents of this table we are able to look back to the processor we were creating with `Alma`^{2}, and finish it by adding the visualization of the problem domain.

In `Alma`^{2}, the visualization of the problem domain has a central concept, which we call *Actor*. An Actor is an object either controlled by the language or just referenced by it. It is composed of a set of *poses*, which it can stand through the animation process, and an internal state. A language can have more than one Actor associated, in order to be more perceptible the visualization of the problem domain. Besides the Actor, to define the problem visualization, `Alma`^{2} offers a set of *Animation Patterns* that stimulate the internal state of the Actors and provoke their animation. Using the same approach of the last section, visualization rules will be applied to the DAST but, in this case, they are based on the animation patterns which are associated to the nodes.

In our case study, the Karel Language only needs one actor: the robot. The images in Figure 1 illustrate some of the possible poses of the robot. Figure 1 (3) is equal to Figure 1 (2), it only was rotated 90^o to the left, the result of a possible animation.

Listing 4, line 2, shows how we created the Actor for Karel Language. The first argument of the constructor is the name of the images that would serve as poses for the Actor. The second argument is the definition of the Actor's state.

This Actor is combined with animation patterns to define the animation of each instruction on the language. We use the knowledge in Table II to guide the creation of the concrete mappings with `Alma`^{2}. In Listing 4 we present two fragments of code that define the animation for the commands TURNLEFT (lines 5 to 17) and PICKBEEPER (lines 19 to 29).

In both cases we append animation patterns to the same kind of tree node: `AnimAssignNode`. These nodes behave exactly the same as the `AssignNode` used in Listing 3, but they have a new attribute that defines the animation. In the first fragment of the code, in Listing 4, we associated the pattern *Rotate*. The code means that the Actor, `robot`, will perform a rotation over the value stored in variable `angle` of its state, and will use the second pose in the set of poses[5]. For the second fragment we used the pattern *Identity*. The code means that the animation of the Actor, `robot`, is only to change its poses from the fourth pose to the third, and from the latter to the second pose in its set of poses.

With all of the animations defined and appended to the tree nodes, the `Alma`^{2} FE for Karel Language is complete. Figure 2 shows some of the results of animating the problem domain of a program, inside the `Alma`^{2}'s environment.

D. Visualizing the Interconnection Between Domains

Finally, with Figure 3, we show the complete synchronization of all the visualization perpectives.

The working window of `Alma`^{2} is divided into four parts that show different perspectives of the program being interpreted. In the upper left corner, the Identifier Table (IT), representing the internal state of the controlled object, is displayed. Also on the left but below the IT, appears the source code (the line being interpreted is highlighted). In the upper right corner,

[5]Notice that the indexes to access the poses are zero-based.

Listing 4. Definition of the Problem Domain Visualization (Fragment)

```
1
2  Actor robot = new Actor(new String[] {"Off", "On", "Pick", "Drop"}, setState());
3
4
5  rule Instruction_TurnLeft {
6    INSTRUCTION ::=  #Turnleft compute {
7      INSTRUCTION.tree =
8        new CStmtsNode(
9            new AnimAssignNode(
10                 (...)
11                 new AnimationPattern[] {
12                     new APRotate(robot, new int[] {1}, "angle")
13                 }
14           )
15       );
16   };
17 }
18
19 rule Instruction_PickBeeper {
20   INSTRUCTION ::= #Pickbeeper compute {
21   INSTRUCTION.tree =
22            new AnimAssignNode(
23                (...)
24                new AnimationPattern[] {
25                    new APIdentity(robot, new int[] {3,2,1})
26                }
27            );
28   };
29 }
```

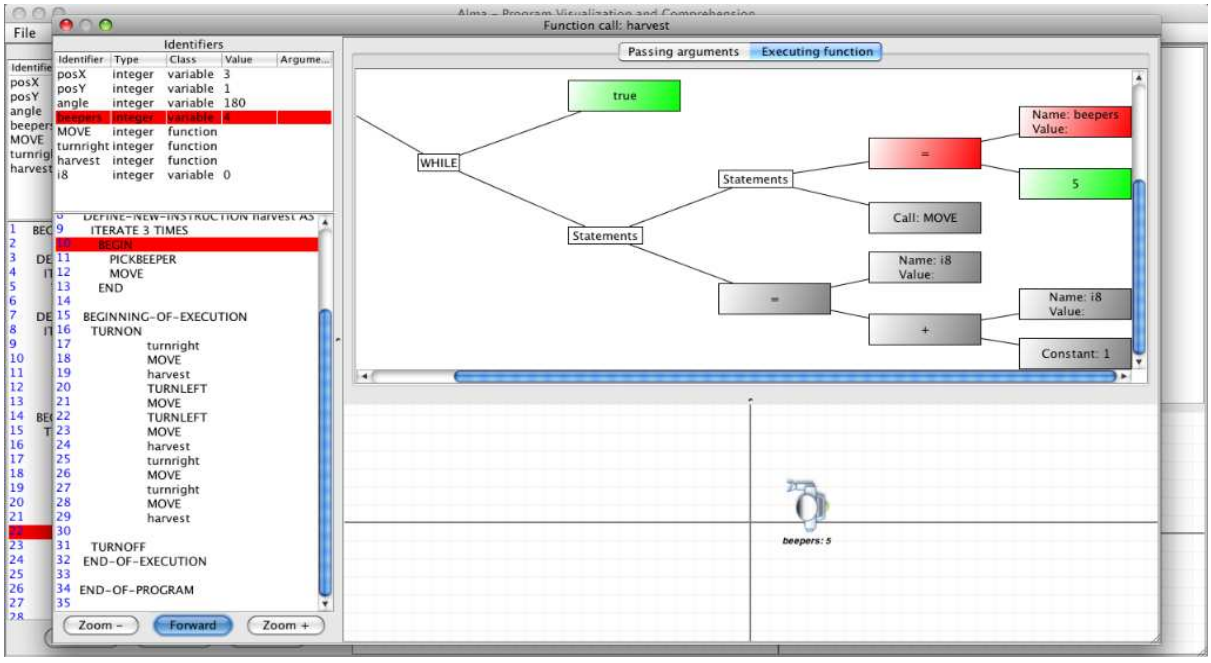

Fig. 3. Synchronization of all Visualization Perspectives

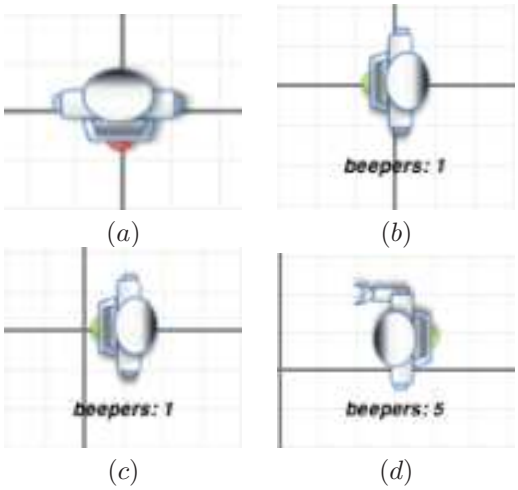

Fig. 2. Problem Domain Visualization. (a) The robot is turned off; (b) The robot turned left for three times; (c) The robot gave a step ahead; (d) The robot is picking an object (we only show the first frame of the animation associated).

the interpretation tree is shown, and below that, the effects of the program execution/interpretation on the objects of the problem domain are displayed.

The views displayed in the four windows are synchronized by Alma^2 engine while performing tree traversals to interpret (and animate) the input program.

The synchronous step-by-step evolution of the information displayed for each view makes visible the cause-effect relation, and grants the envisaged relation between problem and program domains, aiding the analyst to understand the program meaning. It is worthwhile to notice that this feature comes for free due to Alma^2's principle and architecture; it is just needed to develop a FE for the concrete DSL.

V. Conclusion

Karel Programming Language is a Domain-Specific Language designed only to command a robot. Writing a program in Karel Language is an easy task for someone who knows the problem domain owing to the high level of the language constructors and their closeness of mapping to that domain. However the reverse is not true. To understand a program is not an easy task, specially if the person in-charge has no knowledge of the problem domain.

In this paper we propose the use of a traditional non-invasive program comprehension approach to make the understanding of domain-specific programs easier, and more effective. Static information extracted from the source program has been used to create three synchronous views. The Identifier Table (displaying the system state), and Abstract Syntax Tree (decorated with attribute values) are traditional, and provided by many tools; an animation of the program execution is then produced by abstract interpretation over the tree. The third one is novel: it reproduces the effects of program execution on the problem domain. To build that third view, a deeper knowledge of the connections between the language constructors and the concepts in the problem domain is required. When dealing with GPLs this mapping is not evident due to the general purpose character of those programming languages. Therefore it is not common to find PC tools with that capability. Working with DSLs, the closeness between language purpose and a concrete domain, enables to build and offer such a view.

Besides introducing our proposal and displaying a few screenshots from Alma^2 output, we also discussed, from a technical point of view, how the system was implemented.

The main achievements obtained when exercising with Karel Language[6], were:

- the feasibility of re-using Alma, principles and environment.
- the easiness of additionally representing the problem domain and the synchronization of the three views.
- the worth of Alma^2 tool for a faster program comprehension.

In the near future, we will apply the same approach to other case studies dealing with specification languages (that are, indeed, equivalent to declarative programming languages). The aim is to corroborate our working hypothesis, and to generalize the approach.

Concerning the upgrade of Alma^2 in the direction of a customizable tool, we forecast that it would be desirable to allow end-users, not language designer or developer, to easily specify their own visualization.

The chief 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. The external function could be called using the attributes available at DAST nodes, to tune the picture to each concrete situation. 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 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.

References

[1] T. Kosar, P. M. Lopez, P. A. Barrientos, and M. Mernik, "A preliminary study on various implementation approaches of domain-specific language," *Inf. Softw. Technol.*, vol. 50, no. 5, pp. 390–405, April 2008. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2007.04.002

[6]The first case-study of our bilateral project, named *Program Comprehension for Domain-Specific Languages* (DSLpc).

[2] M. J. V. Pereira, M. Mernik, D. da Cruz, and P. R. Henriques, "Program comprehension for domain-specific languages," *ComSIS – Computer Science an Information Systems Journal, Special Issue on Compilers, Related Technologies and Applications*, vol. 5, no. 2, pp. 1–17, December 2008.

[3] M. Mernik, J. Heering, and T. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316 – 344, 2005.

[4] J. I. Maletic and A. Marcus, "Supporting program comprehension using semantic and structural information," in *16th IEEE International Conference on Automated Software Engineering (ASE2001)*. San Diego - USA: IEEE, November 2001, pp. 107–114.

[5] A. J. Ko and B. Uttl, "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.

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

[7] R. Brooks, "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, pp. 196–201.

[8] M.-A. Storey, "Theories, methods and tools in program comprehension: Past, present and future," in *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 181–191.

[9] "Imagix 4d." [Online]. Available: \url{http://www.imagix.com/products/products.html}

[10] P. Anderson and M. Zarins, "The codesurfer software understanding platform," in *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 147–148.

[11] M.-A. Storey, C. Best, J. Michaud, D. Rayside, M. Litoiu, and M. Musen, "Shrimp views: an interactive environment for information visualization and navigation," in *CHI '02: CHI '02 extended abstracts on Human factors in computing systems*. New York, NY, USA: ACM, 2002, pp. 520–521.

[12] G. Antoniol, R. Fiutem, G. Lutteri, P. Tonella, S. Zanfei, and E. Merlo, "Program understanding and maintenance with the canto environment," in *ICSM '97: Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 1997, p. 72.

[13] M. Lanza, S. Ducasse, H. Gall, and M. Pinzger, "Codecrawler: an information visualization tool for program comprehension," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*. New York, NY, USA: ACM, 2005, pp. 672–673.

[14] A. Raza, G. Vogel, and E. Plödereder, "Bauhaus—a tool suite for program analysis and reverse engineering," in *Reliable Software Technologies - Ada-Europe 2006*, 2006, pp. 71–82. [Online]. Available: http://dx.doi.org/10.1007/11767077_6

[15] D. da Cruz, M. Béron, P. R. Henriques, and M. J. V. Pereira, "Strategies for program inspection and visualization," in *CSE'08—International Scientific Conference on Computer Science and Engineering*. High Tatras, Slovakia, September 2008.

[16] D. da Cruz, P. R. Henriques, and M. J. V. Pereira, "Constructing program animations using a pattern-based approach," *ComSIS—Computer Science an Information Systems Journal, Special Issue on Advances in Programming Languages*, vol. 4, no. 2, pp. 97–114, December 2007, ISSN: 1820-0214.

[17] M. Berón, P. R. Henriques, M. J. V. Pereira, and R. Uzal, "Program inspection to interconnect behavioral and operational view for program comprehension," in *York Doctoral Symposium, 2007*. University of York, UK, Oct 2007.

[18] D. da Cruz, R. Fonseca, P. R. Henriques, and M. J. V. Pereira, "How to interconnect operational and behavioral views of web applications," in *ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 263–267.

[19] D. E. Knuth, "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, pp. 1–12.

[20] A. Deursen and P. Klint, "Little languages: little maintenance?" University of Amsterdam, Amsterdam, The Netherlands, Tech. Rep., 1997.

[21] M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer, "LISA: An interactive environment for programming language development," *Compiler Construction*, pp. 1–4, 2002.

[22] P. Hudak, "Building domain-specific embedded languages," *ACM Computing Surveys*, vol. 28, no. 4, pp. 196–202, June 1996. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.6020