



Universidade do Minho
Escola de Engenharia

Nuno Ernesto Salgado Oliveira

**Improving Program Comprehension Tools
for Domain Specific Languages**



Universidade do Minho

Escola de Engenharia

Nuno Ernesto Salgado Oliveira

Improving Program Comprehension Tools for Domain Specific Languages

Master in Informatics

Supervised by:

Professor Doutor Pedro Rangel Henriques

Professora Maria João Varanda Pereira

É AUTORIZADA A REPRODUÇÃO PARCIAL DESTA TESE APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

Universidade do Minho, ___/___/_____

Assinatura: _____

Resumo

Desde o início dos tempos a curiosidade e a necessidade de melhorar a qualidade de vida impeliram o humano a arranjar meios para compreender o que o rodeia com o objectivo de melhorar. À medida que a habilidade de uns foi aumentando, a capacidade de compreensão de outros seguiu-lhe os passos. Desmontar algo físico de modo a compreender as ligações entre as peças e assim perceber como funcionam num todo, é um acto bastante normal dos humanos. Com o advento dos computadores e os programas para ele codificados, o homem sentiu a necessidade de aplicar as mesmas técnicas (desmontar para compreender) ao código desses programas.

Tradicionalmente, a codificação de tais programas é feita usando linguagens genéricas de programação. Desde logo técnicas e artefactos que ajudam na compreensão desses programas (nessas linguagens) foram produzidas para auxiliar o trabalho de engenheiros de software que necessitam de manter ou alterar programas previamente construídos por outros. De um modo geral estas linguagens mais genéricas lidam com conceitos a um nível bastante abaixo daquele que o cérebro humano, facilmente, consegue captar. Previsivelmente, compreender programas neste tipo de linguagens é uma tarefa complexa pois a distância entre os conceitos ao nível do programa e os conceitos ao nível do problema (que o programa aborda) é bastante grande.

Deste modo, tal como no dia-a-dia foram surgindo nichos como a política, a justiça, a informática, etc. onde grupos de palavras são usadas com maior regularidade para facilitar a compreensão entre as pessoas, também na programação foram surgindo linguagens que focam em domínios específicos, aumentando a abstracção em relação ao nível do programa, aproximando este do nível dos conceitos subjacentes ao problema.

Ferramentas e técnicas de compreensão de programas abordam, geralmente, o domínio do programa, tirando pouco partido do domínio do problema. Na presente tese assume-se a hipótese de que será mais fácil compreender um programa quando os domínios do problema e do programa são conhecidos, e entre eles é estabelecida uma ponte de ligação; e parte-se em busca de uma técnica de compreensão de programas para linguagens de domínio específico, baseada em técnicas já conhecidas para linguagens de carácter geral. O objectivo prende-se com aproveitar o conhecimento sobre o domínio do problema e melhorar as ferramentas de compreensão de programas existentes para as linguagens genéricas, de forma a estabelecer ligações entre domínio do programa e domínio do problema. O resultado será mostrar, visualmente, o que acontece internamente ao nível do programa, sincronizadamente com o que acontece externamente ao nível do problema.

Abstract

Since the dawn of times, curiosity and necessity to improve the quality of their life, led humans to find means to understand everything surrounding them, aiming at improving it. Whereas the creating abilities of some was growing, the capacity to comprehend of others follow their steps. Disassembling physical objects to comprehend the connections between the pieces in order to understand how they work together is a common human behavior. With the computers arrival, humans felt the necessity of applying the same techniques (disassemble to comprehend) to their programs.

Traditionally, these programs are written resorting to general-purpose programming languages. Hence, techniques and artifacts, used to aid on program comprehension, were built to facilitate the work of software programmers on maintaining and improving programs that were developed by others. Generally, these generic languages deal with concepts at a level that the human brain can hardly understand. So understanding programs written in this languages is an hard task, because the distance between the concepts at the program level and the concepts at the problem level is too big.

Thus, as in politics, justice, medicine, etc. groups of words are regularly used facilitating the comprehension between people, also in programming, languages that address a specific domain were created. These programming languages raise the abstraction of the program domain, shortening the gap to the concepts of the problem domain.

Tools and techniques for program comprehension commonly address the program domain and they took little advantage of the problem domain. In this master's thesis, the hypothesis that it is easier to comprehend a program when the underlying problem and program domains are known and a bridge between them is established, is assumed. Then, a program comprehension technique for domain specific languages, is conceived, proposed and discussed. The main objective is to take advantage from the large knowledge about the problem domain inherent to the domain specific language, and to improve traditional program comprehension tools that only dealt, until then, with the program domain. This will create connections between both program and problem domains. The final result will show, visually, what happens internally at the program domain level, synchronized with what happens externally, at problem level.

Acknowledgements

“Love and work are the cornerstones of our humanness”

Sigmund Freud

Love and work are the cornerstones of our humanness, said Freud! I could not agree more. Is it possible to do a good work if you are not surrounded by friends who really love and care for you? I am afraid not! So, I relied on my friends and took advantage of their love to accomplish this master’s work. Now it is time to write some words acknowledging them, one by one, expressing my deepest feeling of friendship and thankfulness. Long time I’ve waited for this moment. The possibility to thank someone would be the greatest thing in a person’s life!

First of all I would like to thank *Professor Pedro Henriques* for the magnificent and patient time he devoted supervising me, advising me personally when we were together, and writing or whispering comforting words on the chat or on the phone, when we were apart. Perhaps calling him supervisor or professor is meaningless. . . *friend for life* would suit best and would made a better combination with his friendly beards.

He played a very intensive and important role both in my professional and personal life. I am glad I could become a friend of him, being introduced to his family and other friends, and also for being invited to feel the warmth of his home. Muito obrigado, Rocas!

Second, I would like to thank *Professora Maria João Varanda Pereira*, for all the time she spent traveling all the way from Bragança to Braga to lend me her hand and guide me through the stormy days of this thesis. I would never forget the amazing times she, her husband, Rolando, and daughter, Patrícia, offered me when I spent some days in Bragança, introducing me to nice and unforgettable persons. Muito obrigado, Zinha!

Third, I would like to thank *Daniela da Cruz*, for being by my side whenever I needed. She was my pillar! I can’t classify her help on this thesis because such help provoked an enormous stack-overflow in my memory and, principally, in my heart. I am really glad to have a friend like she is! Muito Obrigado, Ni!

Specially, I want to thank these three individuals for our friendship during the previous three years, where we worked together in an unrealistic but terrific environment.

I would like to thank also the Slovenian team members *Marjan Mernik*, *Matej Črepinšek* and *Tomaž Kosar*, for the constant help and the fruitful discussions in the bilateral project's meetings. *Za vse, moj hvala!*

Me gustaria, también, dar mis reconocimientos a mi grand amigo *Mario Berón* por su constant amistad aunque la distancia, y también por los debates sobre comprensión de programas y la prestación de material en este campo de la investigación. *Gracias amigazo!*

For pointing some bibliographic references to complete and improve the state of the art of this thesis, I would like to thank *Professor Vasco Amaral* and *Professor Ivan Lukovic*.

I must thank also all the participants in the experiment I needed to perform to complete this thesis: *Alice Marques*, *Ana Sofia*, *Bruno Costa*, *Carlos Pereira*, *Dave Moderno*, *Fábio Lima*, *Francisco Cruz*, *Francisco Maia*, *João Granja*, *João Tiago*, *Jorge Abreu*, *Miguel Lopes*, *Miguel Nunes*, *Miguel Pires*, *Nélio Guimarães*, *Ricardo Vilas Boas*, *Rui Pedro*, *Susana Silva*, *Nuno Vasco* and *Zé Luis*. To all of you, my sincere acknowledgments.

I can not forget the group of special friends with whom I share interesting time on the most sacred part of the day (the lunch), and with whom I spent some free time playing football and discussing whatever we desired: *Carlos*, *Granja*, *Fábio*, *Chico*, *Maia*, *Miguel Lopes*, *Miguel Pires*, *João Tiago* and *Giovan*.

During the two years of this master degree I met other incredible persons who really helped me a lot in several moments. Besides those acknowledge before, I really want to thank *Pedro Lemos* for the incredible friendship that we built together; *Bastian Cramer*, for helping me getting to grips with his amazing tool, *DEViL*; *Cristina Pereira*, for the hours she spent translating technical german text into portuguese, specially for me; and *Alberto Simões*, for the conversations in the corridor and the help he gave me on solving several issues concerning my thesis and work around it... I would say that if there is a solution for your problem, he knows the answer! To finish this group of acknowledgments, I really want to thank two other amazing persons: *Professor José João* and *Professor Jorge Sousa Pinto*; not only for the conversations we had, but also for the friendly support they gave me, which is the most valuable value in the world!

A special acknowledgement I would like to express to all my friends living in other countries or distant cities with whom I could only exchange some comforting words, during this degree, by chat, email, mobile phone messages, phone calls, and so on. I will not name all of you, folks, but you know who you are. Friendship and love can always break the distance!

Finalmente quero agradecer aos meus pais, *Arlindo* e *Teresa*, irmã, *Patrícia*, e seu namorado, *Ibrahim*, por todo o apoio que me deram nas horas mais complicadas deste mestrado, e o suporte nos momentos das decisões importantes da vida. Peço desculpa pelas muitas horas de ausência e de silêncio passados em devoção ao trabalho ou pelos momentos de mau humor provocados pelo cansaço ou stress. Obviamente, este agradecimento não é só por estes dois anos; inclui também todos os outros anos de estudos que os antecederam, os quais me guiaram a sucessos a todos os níveis. Sem este constante apoio, nada seria possível. Muito Obrigado Pai, Mãe e Irmã!

Contents

Figures	xii
Tables	xiii
Listings	xv
Acronyms	xvii
1 Introduction	1
1.1 Overview	1
1.2 Problems and Doubts	3
1.3 Goals to Achieve	4
1.4 Outline	6
2 Domain-Specific Languages	9
2.1 Characteristics	11
2.1.1 Cognitive Dimensions	11
2.1.2 Cognitive Dimensions Applied to DSLs	13
2.2 Dichotomies	14
2.2.1 Abstractness and High-level <i>versus</i> Concreteness and Low-level	14
2.2.2 Expressiveness <i>versus</i> Computational Power	15
2.3 Pros and Cons	16
2.3.1 On DSLs Usage Perspective	16
2.3.2 On DSLs Development Perspective	17
2.4 Developing DSLs	18
2.4.1 Complete Language Design Approaches	19
2.4.2 Languages Extension Approaches	21
2.4.3 COTS Products-based Approach	21
2.5 Application Domains	22
2.6 Summary	23
3 Program Comprehension	25
3.1 Fundamental Definitions	28
3.1.1 Program Reader	28
3.1.2 Problem Domain	28
3.1.3 Program Domain	28
3.1.4 Domain Concepts	29

3.1.5	Mental Model	29
3.1.6	Information Extraction	29
3.2	Software Visualization	30
3.2.1	Software Visualization Systems — Categorizations	30
3.2.2	Visual Representations and Their Requirements	31
3.3	Cognitive Models	33
3.3.1	Top-Down Model	33
3.3.2	Bottom-Up Model	34
3.3.3	Hybrid Cognitive Models	35
3.4	Program Comprehension Tools	36
3.4.1	The Tools	36
3.4.2	Tool’s Output Characterization	38
3.5	Domain-Oriented Program Comprehension	40
3.5.1	Approaches for Connecting Domains in DSL Comprehension	40
3.5.2	Tools for DSLPs	42
3.5.3	Discussion	45
3.6	Summary	46
4	An Idea to Comprehend DSLs	47
4.1	Choosing One Suitable Approach	48
4.2	DAST — The Initial Approach	49
4.3	D _{ap} AST — The Improved Approach	51
4.3.1	DAST <i>versus</i> D _{ap} AST	52
4.3.2	Inside Animation Nodes	52
4.3.3	Traversing the D _{ap} AST — Theoretical Approach	54
4.4	Summary — Answering Question One	55
5	Alma²	57
5.1	Developing the System	58
5.1.1	Architecture	59
5.1.2	Internal Structure	61
5.1.3	Traversing the D _{ap} AST — Practical Approach	64
5.2	Using the System	68
5.2.1	The Expert does.	69
5.2.2	The Program Reader does.	74
5.3	Summary — Answering Question Two	75
6	Case Studies	77
6.1	Karel’s Language	77
6.1.1	Problem Domain Definition	78
6.1.2	Language Formal Definition	78
6.1.3	Conception: Mappings, Visualization and Animation	80
6.1.4	Animation Using Alma	84
6.2	The Lavanda Language	87
6.2.1	Problem Domain Definition	88
6.2.2	Language Formal Definition	88
6.2.3	Conception: Mappings, Visualization and Animation	89

6.2.4	Animation Using Alma	92
6.3	Summary	95
7	Approach Assessment	97
7.1	Experiment Preparation	98
7.1.1	Team Know-how	98
7.1.2	Related Work	99
7.1.3	Objective of the Experiment	100
7.1.4	Questionnaire Design	101
7.1.5	Participants Identification	103
7.2	Experiment Execution	104
7.3	Final Results	105
7.3.1	Results and Discussion	105
7.3.2	Threats to Validity	111
7.4	Summary — Theorem Validation	112
8	Conclusion	115
8.1	Discussion and Conclusions	118
8.2	Future Work	119
	Bibliography	121
A	<i>AnimXXXNode</i> Family of Classes	139
B	Experiment Questionnaires	141

Figures

1.1	Method to Proof the Theory	5
2.1	Overview of Compiler Tasks	20
3.1	Reengineering - A Maintenance Methodology	26
3.2	Gap Between Program and Problem Domain	27
3.3	Framework for Debugging DSLs	44
4.1	Answering Question 1	47
4.2	DAST	50
4.3	Processing a DAST	51
4.4	D_{ap} AST	53
4.5	Processing the D_{ap} AST	54
5.1	Answering Question 2	57
5.2	Alma ² Architecture	59
5.3	Alma ² Internal Structure	61
5.4	Animation Pattern Workflow	64
5.5	Process of Drawing Actors	67
5.6	The Alma ² Users	68
5.7	State Initialization	72
5.8	Initial state of Alma ²	74
6.1	Problem Domain associated with the <i>Karel's Language</i>	79
6.2	Sequence to move the robot	82
6.3	Sequence to rotate the robot	83
6.4	Sequence to pick an object	83
6.5	Sequence to drop an object	84
6.6	Label as a Resource for Visualization	84
6.7	Visualization after executed the first <i>turnright</i> instruction	86
6.8	Visualization after executed the first MOVE instruction	86
6.9	Visualization of a frame of the <i>picking object</i> sequence	87
6.10	Problem Domain associated with the <i>Lavanda Language</i>	89
6.11	Initial Visualization of a <i>Lavanda Language</i> especificaiton	93
6.12	Visualization of the contents in the first bag	94
6.13	Problem Domain Visualization of the second Bag	94
6.14	Final Visualization of the <i>Lavanda Language</i> specification	95

7.1	Theorem Proof Schema	98
7.2	Main Structure for the Experiment Tasks	102
7.3	Distribution of Questions	102
7.4	Comprehension Tasks — History and Expertise of the Participants . .	104
7.5	Time Results	106
7.6	Correctness Results	108
7.7	Importance Results	110
8.1	Final Theorem	117
B.1	Participant Identification Form	142
B.2	Questionnaire — Page 1	143
B.3	Questionnaire — Page 2	144
B.4	Questionnaire — Page 3	145
B.5	Questionnaire — Page 4	146
B.6	Questionnaire — Page 5	147

Tables

2.1	Sample of DSLs Application Domains	22
3.1	Information Provided by the PCTools Examined	39
6.1	Connection of concepts to images in <i>Karel's Language</i>	82
6.2	Connection of concepts to images in <i>Lavanda Language</i>	91
7.1	Average of Time Spent in Solving the Questions	106
7.2	Average of Time Spent per Type of Question	107
7.3	Average Correctness of the Questions	108
7.4	Average of Correctness per Type of Question	109
7.5	Average Importance of Problem Domain Visualizations and Views Synchronizations per Question	110
7.6	Summary of the Experiment's Results	113

Listings

5.1	Fundamental Packages to Import	70
5.2	Toy Language Syntax	70
5.3	Example of a Tree Injection Method	71
5.4	Creation of an Animation Node	71
5.5	Semantic Rules for the Robot Movement	73
5.6	Definition of Animation Nodes for one Instruction	73
6.1	Formal Definition of <i>Karel's Language</i>	80
6.2	The code for the Harvest Problem	85
6.3	Formal Definition of the <i>Lavanda Language</i>	88
6.4	Specification in <i>Lavanda Language</i>	92

Acronyms

A

- ADT Abstract Data Type, p. 42.
- AG Attribute Grammar, p. 20.
- API Application Programming Interface, p. 21.
- AST Abstract Syntax Tree, p. 49.

B

- BE Back-End, p. 50.
- BNF Backus-Naur Form, p. 19.
- BORS Behavioral-Operational Relation Strategy, p. 42.

C

- CD Cognitive Dimension, p. 11.
- CDF Cognitive Dimensions of Notations Framework, p. 11.
- CFG Context Free Grammar, p. 70.
- COTS Commercial Off-The-Shelf, p. 21.

D

- D_{ap} AST Decorated with animation patterns Abstract Syntax Tree, p. 52.
- DARE Domain Analysis and Reuse Environment, p. 18.
- DAREp Domain Analysis and Reverse Engineering Project, p. 40.
- DAST Decorated Abstract Syntax Tree, p. 49.
- DDF DSL Debugging Framework, p. 43.
- DSL Domain-Specific Language, p. 1.

- DSL_P Domain-Specific Language based Program, p. 2.
- DSL_{pc} Slovenian-Portuguese Bilateral Project on Program Comprehension for Domain Specific Languages, p. 1.
- DSML Domain-Specific Modeling Language, p. 10.
- DsPCTools Domain-Specific Program Comprehension Tools, p. 3.
- DSSA Domain Specific Software Architectures, p. 18.
- E**
- EBNF Extended Backus-Naur Form, p. 19.
- F**
- FAST Family-Oriented Abstractions, Specifications and Translations, p. 18.
- FE Front-End, p. 50.
- FODA Feature Oriented Domain Analysis, p. 18.
- G**
- GPL General-Purpose Language, p. 2.
- GPLP General-Purpose Language based Program, p. 2.
- I**
- IDE Integrated Development Environment, p. 43.
- IT Identifier Table, p. 31.
- J**
- JVM Java Virtual Machine, p. 63.
- M**
- MDSE Model-Driven Software Engineering, p. 10.
- O**
- ODE Ontology-based Domain Engineering, p. 18.
- ODM Organization Domain Modeling, p. 18.

P

PC Program Comprehension, p. 1.

PCTools Program Comprehension Tools, p. 2.

PR Program Reader, p. 28.

R

RE Regular Expression, p. 80.

S

SV Software Visualization, p. 2.

SVS Simultaneous Visualization Strategy, p. 42.

V

VL Visual Language, p. 69.

VMTS Visual Modeling Transformation System, p. 43.

X

XML Extensible Markup Language, p. 10.

*To Ni, Zinha and Rocas,
for being such incomparable friends.*

Chapter 1

Introduction

We don't see things as they are, we see them as we are.

Anaïs Nin

This document is a dissertation presenting a master's thesis in the area of *Program Comprehension* (PC) applied to *Domain-Specific Languages* (DSL). This master's thesis is a component of the second year of the new Masters Degree (second cycle of Bologna's), that is to held at University of Minho in Braga, Portugal. The subjects discussed in this document sprang out from the *Slovenian-Portuguese Bilateral Project on Program Comprehension for Domain-Specific Languages* (DSLpc)¹.

In this chapter is given an overview of this work's topics in the real world. After a glance on the main pitfalls, which steer the studies in this thesis, are presented the main objectives and the contributes. Also a brief discussion about the methodologies applied in the work behind this dissertation is made.

1.1 Overview

Since the advent of the computer era, every day, some genius brains come up with new ideas and new techniques to solve problems that exist in the world, and can only be solved by a computer program or system. Several problems have been worked out and many others are still there to be figured out. Informatics is not a limited discipline applied to a limited set of domains. Instead, it is everywhere. Businesses, medicine, justice, politics, sports, biology are some of the domains where it is applied to solve emergent problems.

As the techniques and technologies for software development are being constantly improved, rapidly the systems developed (those used to solve the problems) turn into legacy systems.

Legacy systems have been the main propeller for the advances made in the PC area [136]. They need maintenance or even improvements to follow the new requirements imposed by the evolving needs. Maintenance is known to be the most time-consume part of the software life-cycle. In [172] the author, based on [29]

¹Project's home page: www.di.uminho.pt/~gepl/DSL

and [65] informs that 50% to 75% is the average time spent for the tasks involved in the maintenance; and that 47% to 62% of the maintenance time is spent in comprehending the whole system. Many reasons can cause this time expenditure: the complexity of the systems, the non-preparation of the programmers to perform such a task, the non-familiarization of the programmers with the domain for what the system was developed, and so on. But the main reason lies on the fact that in the major part of the cases, the process of comprehending a program is done manually [172].

In order to gain time in the comprehension tasks, tools have been developed. These tools try to systematize and automatize the process of comprehending programs from the simplest program to the most complex system. However, until now, any of them reduces up to 100% the manual labour involved in these tasks.

Many techniques, which can be seen as sub-systems, are explored in *Program Comprehension Tools* (PCTools). *Software Visualization* (SV) systems are an example. They aid in the comprehension task, by showing internal aspects of the program or system, that otherwise, humans would not assimilate so rapidly [98]. These aspects are, traditionally, sets of concepts retrieved from the program domain.

Program domain concepts are more complex to understand, the more generic is the language used to write a program. *General-Purpose Languages* (GPL), like C, C++, C#, Java among others, have this handicap. As they are used to program and solve any problem in any kind of domain, their syntax and semantics are generic and closer to the machine level of understanding than to the human's. Although the abstraction level of these languages have been increased with the born of new programming paradigms, like the Object-Oriented, they still have programming concepts that are hard to comprehend. And these difficulties grow bigger, when the programs coded with such languages — *General-Purpose Language based Programs* (GPLP) — are huge.

However, as in politics, justice, medicine, and so on, groups of specific words are regularly used to ease the comprehension between people, also in informatics, programming languages that address a specific domain were created. These languages, so called DSLs, rise the abstraction of the program domain and shorten the distance to the concepts of the problem to which they were designed. So that, programs written with these languages — *Domain-Specific Language based Programs* (DSLp) — can be easily comprehended by persons aware of the problem domain. But, the same may not be true for persons with lack of knowledge in the domain.

Thus, the study of program comprehension for DSLs is a necessity and a reality. In the last few years some studies were done in order to understand on which type of programming language (GPLs or DSLs) program comprehension strategies are more effective. The intuition suggests that is easier to use a language (in the extent of learning it, developing with it and evolving programs made with it) when domain of its application is known and well defined [153].

The contribute provided, so far, by previous projects like VODA² and PCVIA³, are important for this area of research. In the sequel of them, a part of the DSLpc project, that origins this thesis, aims at studying *Domain-Specific Program Comprehension*

²Project's home page: <http://wiki.di.uminho.pt/twiki/bin/view/Research/Voda>

³Project's home page: <http://wiki.di.uminho.pt/twiki/bin/view/Research/PCVIA>

Tools (DsPCTools), in order to find new techniques that can be applied to DSLs.

1.2 Problems and Doubts

In 1978, *Brooks* [31] drew a theory about the behavior of programmers before the task of comprehending a program. He affirms that, in order to comprehend a program, any person (commonly a programmer) should gather knowledge, from several resources related with the program and from the program itself; and with this knowledge, be capable of creating bridges, addressing several domains of knowledge, between the program and the problem domains. To put this in simpler words, the theory says that a program is better understood if the person who is through the process of comprehension knows the program and problem domains, and is capable of establishing bridges between them.

Until today, this theory is proven to be correct. The development of some tools and techniques for program comprehension, have adopted *Brook's* theory in order to ease the process of program comprehension [131, 157]. But, commonly, they give more attention to program domain aspects, and took little advantage from the problem domain. Many times, the latter is not even addressed. So, despite adhering to the proposed theory, they do not follow it completely. Thus, in general, PCTools do not address the problem domain. This causes a hole in this set of tools, and it should be filled out.

In part, it happens because these tools and techniques work upon GPLs, and the difficulties on retrieving problem domain aspects from GPLs and dealing with them, are known for this kind of languages.

However, when dealing with DSLs, as the problem and program domains are close to each other, it is easier to infer a conceptual connection between them. But as said before, this is true if the person who is being through the process of comprehending a program is acquainted with the problem domain. Otherwise it would be necessary dedicated techniques and tools to help on the comprehension of DSLP [185]. A DSL user has different necessities, compared to a GPL user. He would rather have a visualization of the program at the problem level than at the program level.

As long as PCTools mostly cope with GPLs, throughout the years of research on this area, many techniques tools and approaches (henceforth the set of tools, techniques and approaches will be referred to as PCTools) took shape and were applied to those languages. Concerning this, one question seems to have necessity of being answered:

Question 1. *Can the PC techniques, tailored for GPLs, be applied on DSLs?*

In case of a positive answer, this question arises an important group of other questions. From that group, the following question gains relevance, since it summarizes all the doubts and misbeliefs that hang upon this work:

Question 2. *Is it possible to improve existent PCTools, in order to give the end-user useful and better perspectives to comprehend the program and the problem?*

Both questions can be answered affirmatively. At least, at a first glance. Depending on the PCTools that are trying to be adapted, the conviction is that *yes*,

they can be configured to cope with a new kind of languages, DSLs, in this case. However it is a conviction that should be proved.

Regarding the second question, it is also a conviction that adapted versions of PCTools can help users on the process of comprehending a DSLP, *yes*, but only if a clear investment in order to address the problem domain, besides the program domain, is made. Without it, the users will only absorb the program internals, and those (users) not familiarized with the domain, will remain with almost all the difficulties as if there were not such PCTools.

Since it is not usual to find, in PCTools, visualizations of the effects that the execution of a program provoke in the problem domain, an investment on this direction should be made when adapting the PCTools to cope with DSLs, taking great advantage from the problem domain and its intrinsic connections with the program domain.

1.3 Goals to Achieve

Summing up the previous section, it is known that some PCTools follow Brook's theory, but not completely. The major part of them only address the program domain and work upon generic programming languages. A problem was raised from the fact that there is not effective PCTools to help in DSLP comprehension, addressing both program and problem domains. From here emerged doubts and convictions about the possibility of adapting PCTools to cope with DSLs, that should be proved.

Brook's theory is the basis of the hypothesis that steers the present thesis. This theory is not focused on the development of PCTools, but on the behavior of the programmer when studying a program to comprehend it. Since PCTools aim at helping the programmers to understand programs, the knowledge behind the theory can be taken to guide the behavior of the programmer via a tool. But, in an attempt to bring closer such theory to the development or improvement of PCTools, it seems that it needs an increment. The following theorem represents the new visage of the theory applied to PCTools:

Theorem 1. (Extended Brook's Theory) *A program comprehension tool, will ease the task of comprehending a program when synchronized visualizations of its program and problem domains are provided, since from these visualizations, the user is able to build a conceptual knowledge base that easily bridges both program and problem domains.*

One of the aims of this master's thesis is to prove that the theory just composed is correct. But, to do so, first, it is needed to support the affirmative answers given to the questions raised in the previous section. In Figure 1.1 is shown a scheme of how to proceed with this proof and what are the main resources needed to succeed.

According to this, the studies underlying this thesis are twofold: on the one hand it is made a more theoretical study, and on the other hand a more practical one for proof of concepts. The goals proposed to attain in this work are the following:

- Study techniques, tools and other approaches on program comprehension, to

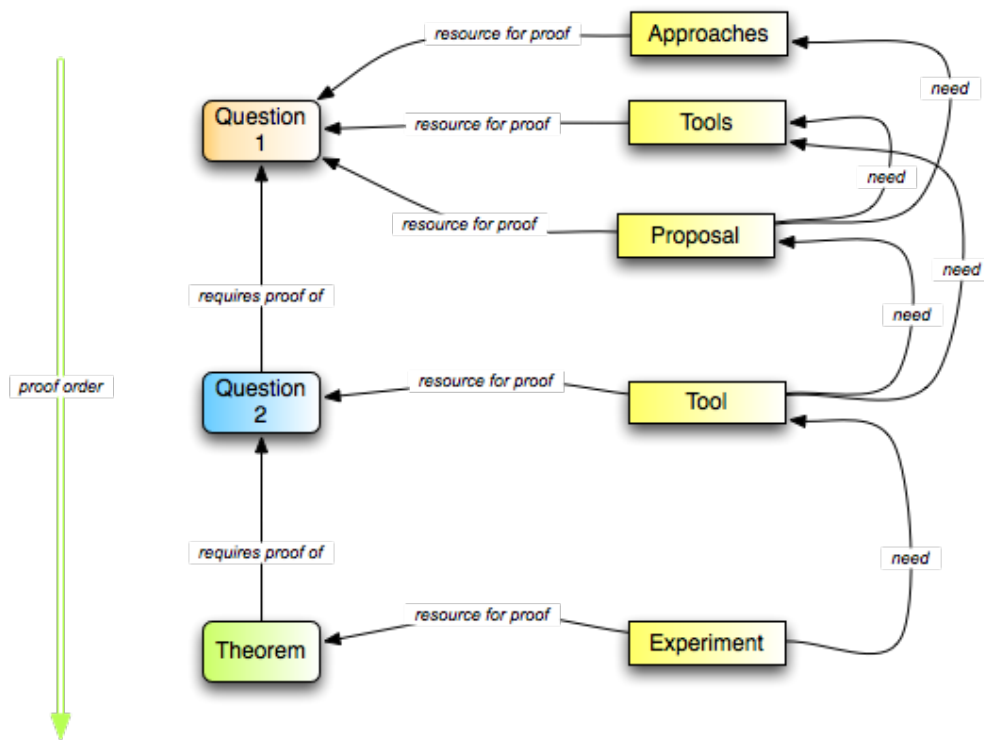


Figure 1.1: Method to Proof the Theory

reason about whether they can or can not be applied to the comprehension of DSLP;

- Study a way of applying them to the development of PCTools and/or their improvement into DsPCTools.
- Propose the new technique which, theoretically, may ease the process of comprehending DSLP;
- Develop the prototype of a program comprehension tool based on an existent one, which follows a technique studied before. The prototype should focus on the creation of the visualization at the problem domain level, and, fundamentally, provide a synchronized visualization of the program and problem domains. This would hold the requirements of the theory stated before.
- Evaluation of the developed prototype, resorting to a non uniform group of possible users. This last objective would be used to corroborate, or not, the hypothesis of this thesis.

This work is expected to improve the way the problem subjacent to a program is shown to the user, in order to accomplish its comprehension. Also, it is intended to be a pioneer work aiming at increasing the number of DsPCTools, so the hole on the set of PCTools will be filled out.

1.4 Outline

After this opening chapter, the document is structured as follows:

In Chapter 2 a deep study on domain-specific languages is provided. The characteristics of these languages, the advantages and disadvantages concerning their use and development, and approaches to improve and make more agile the DSL's implementation process are some of the topics addressed in this study. This chapter would provide a basilar knowledge about these languages as they are the focus of this master thesis.

In Chapter 3 are addressed the main keywords of the research field on program comprehension. Some specific and fundamental definitions that will be used through all the document are presented here. Main focus is given to the topics of software visualization and cognitive models, as they are also central issues on this thesis. A study on the state of the art, concerning the tools for program comprehension is also provided in this chapter. Tools are classified according to important program comprehension aspects, to make evident what is missing in those tools. This, along with a presentation of tools to analyze domain-specific language based programs, enables the reader to understand where the contributions of this thesis stand.

In Chapter 4, a program comprehension tool is chosen from those studied before, in Chapter 3; then the program comprehension technique followed by that tool is carefully studied. Finally a new approach is designed based on the analyzed one. This new approach has the objective of providing visualizations and animations of the problem domain concepts, and also to synchronize them with the program domain visualizations. For that, a powerful intermediate representation of the program, extended with special patterns, is used; and the method to animate this representation, resorting to the patterns, is depicted.

In Chapter 5 the tool chosen before is extended to cope with the new program comprehension approach. The architecture of the resultant tool and internal relation between the several components are presented and explained. Moreover, the main algorithm for the animation of the problem domain and its synchronization with the program visualizations, is explained with some detail. At the end, it is briefly explained how the users should use the tool; two groups of users with different expertise level are taken into account in such a usage explanation.

In Chapter 6 the tool is tested with two different case studies. The case studies follow a set of specific steps determined in the previous section as those necessary to build the visualizations of the domains. Each case study addresses a different domain specific language; the first one is an imperative language, and the second is a declarative. Difficulties and problems on creating these visualizations are addressed for each type of language, however explanations to solve them are given.

In Chapter 7 an experiment to assess the effectiveness and usefulness of the approach is described. The details of the preparation, concerning the previous know-how, the tasks and the questionnaire design, and the details of the conduction, concerning the steps followed to achieve successful results are presented. Tables and charts present the results obtained, and are accompanied by analysis and explanations for those results.

In Chapter 8 the thesis is concluded. The main achievements and contributions

are highlighted, and a space for discussing future work topics is reserved.

Chapter 2

Domain-Specific Languages

A programming language is low level when its programs require attention to the irrelevant.

Alan Perlis

No matter what is done, computers will always understand things in terms of 0's (zeros) and 1's (ones). These *things* they understand are human-made programs. And humans, although capable of doing it, are not proficient in *speaking* that binary language. However, the computer can be taught to translate any language into its preferred idiom. For this reason, humans do not need to go down to binary level. Instead, they can keep the way they talk to computers at a very perceptible level by raising the abstraction level of the language they use.

Programming is no more a computer-oriented task. Several factors play an important role in it. One of the most important is the concern about the future of a software piece, namely, its maintenance. It is not a computer that will maintain the software, but a human. So the target of the code written must not be only the computers, but also the humans.

Thus, to clarify the dialogue between persons and program code, the terms and concepts used in the programming language syntax and semantics should be close to those the person knows about the real world.

Two types of languages are mainly used to write programs for computers: **GPLs** and **DSLs**.

There is not a precise definition for **GPL** [46, 45, 201]. They are created to solve any kind of problem no matter the area or domain this problem fits into. Normally they are the programmer's choice for the communication with the computer. As they are general purpose and widely used, they are adopted by the majority of programmers. The existence of a large community of experts in one **GPL** also plays an important role in such adoption. But issues like the maturity of the language, the availability of well tested and optimized compilers, and the existence of good development tools or environments [84], are the more crucial for the referred preference. But, these languages have drawbacks when regarding other aspects like writing and reading or understanding their programs, commonly addressed as semantic gap.

Concerning the former aspect, they always imply vast programming expertise. Hence, not every human is able to use them. Concerning the latter aspect, **GPLs** are

commonly low level languages. *Low level* language means that the syntactic constructors and underlying semantics of that language address implementation particularities that are closer to the machine's way of working than to the human's way of thinking. Their varied formats and the programming paradigms they follow, which try to raise the abstraction level, are not enough to erase the difficulties observed on comprehending the programs written with these languages.

DSLs can be defined by the following sentence:

A domain-specific language is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain. [196]

This sentence gives a concrete definition of what a DSL is. Although a small aspect can be discussed. It is said that a DSL is a programming language. But *Wile* [204] affirms that this is not necessarily true. He supports his affirmation with the example of musical notation, which is a DSL, but it is not a programming language. On the other hand, it is said that a DSL is an executable specification language. But, again, *Mernik et al.* [123] say that DSLs are not necessarily executable, referring that they can be used just for being processed as happens, for instance with *Extensible Markup Language* (XML) dialects. Therefore, perhaps the better solution is to change the beginning of the definition, and say that a DSL is a computer language; the definition with this change neither generalizes nor specifies the purposes of such a language, but enhances its major characteristic, which is the fact of being tailored for a specific problem domain [197].

As these languages are built for a specific domain, it is easy to absorb the main concepts and features inherent to this domain, and bring them to the language notation [105]. This would result in a notation capable of raising the abstraction level of the programming [41], that is, the notation addresses a higher level of abstractness, enlarging the distance to the machine's way of working but shortening it to the human's way of thinking. Moreover, these languages tend to be small [16, 126] and less complex.

DSLs, as well as GPLs, can be regarded as programming languages, or modeling languages. The latter are called *Domain-Specific Modeling Languages* (DSML) [92], sharing the same global definition given before for DSLs. Although programming languages and modeling languages have not precise boundaries, there are efforts trying to clarify their borders, by defining sets of criteria for their classification [187]. One important point is that DSMLs are used in earlier phases of the development of a software system, namely, the system's designing or modeling phase, in the *Model-Driven Software Engineering* (MDSE) approach. Another aspect is that DSMLs can be used to derive computations, by translating the modeled behavior of the software in an executable source.

Since DSLs are the central issue of this thesis, a further and deeper look into their discussion is given in the remainder of this chapter. First, in Section 2.1, the characteristics of these languages are addressed, in order to enhance the main differences to GPLs, which are discussed in Section 2.2. The characteristics of these

languages are intrinsically related with the advantages and disadvantages that their usage and development present. These pros and cons are discussed in Section 2.3. Finally, it is presented a small overview of DSLs application areas and some examples of these languages, in Section 2.5.

2.1 Characteristics

GPLs are perfectly established in the software development life-cycle. Their characteristics are so widely spread among the software engineers community that are regarded as a natural thing. However, with DSLs is not the same. There is the necessity of enhancing their characteristics in order to defend their usage instead of GPLs. In part, this oughts to the research and studies made on DSLs during the last ten years [204, 123, 105], revealing the importance that these languages are achieving in software engineering. Because of that, the characteristics of DSLs are worthwhile to know and understand.

The following sections present those characteristics and justifies them resorting to the *Cognitive Dimensions* (CD), which are also overviewed in the remainder of this section.

2.1.1 Cognitive Dimensions

CDs [71, 73, 74] are attributes of computer languages and other similar notations. These attributes are comprehended in a framework, known as the *Cognitive Dimensions of Notations Framework* (CDF) [27], which was created to help on the design of new notational systems, by providing available design choices, and means to evaluate the usability of these systems. So, an usual application of the CDF is on the improvement of notations and associated systems, with respect to the users' perspectives [93, 38].

In this section, the CDs are listed and each one is briefly explained, in order to expose an overview about its purpose in the framework.

- Abstraction — is related with the possibility of incrementing the original notation by creating an higher level perception of the terms. Examples are the possibility of defining new macros and new data-structures.
- Closeness of Mapping — is related with the distance between the notation and the terms of the domain that are to be described. That is, the notation entities describe precisely what the user wants to describe.
- Consistency — is related with the capacity of expressing similar semantics resorting to a similar syntax. That implies the existence of patterns in the notation, so they help users on identifying desired information, without breaking expectations.
- Diffuseness — is related with the verbosity of the notation. That is, the number of words and characters necessary to express a term or concept of the domain.

- **Error-Proneness** — is related with the non-protection of the notation against user-made errors. In other words, are the characteristics of the notation that allow users to express erroneous things.
- **Hard Mental Operations** — refer to the great necessity that users have on thinking hard and resorting to tools or other resources in order to understand what the notation means in a precise context.
- **Hidden Dependencies** — refer to connections on the notation entities that are important, but are not fully exposed to the users' eyes. Examples are intricate class hierarchies, HTML links, operations with side-effect, among others.
- **Premature Commitment** — is related with the imposition of an order of doing things. The users are constrained on making decisions even before they have the notion of what would be needed in the future. This dimension can also appear under the name of *Imposed Guess-Ahead*.
- **Progressive Evaluation** — is related with the possibility of checking the state of a description in a notation, before finishing it. The interpretation rather than the execution of programs, as happens with `Haskell`, for instance, allows a partial evaluation of the programs.
- **Provisionality** — is related with the possibility of making provisional actions. It is similar to the *Premature Commitment* dimension, but in the present case, the commitment is not imposed.
- **Role-Expressiveness** — refers to the easiness on inferring the purpose of a notation entity, just by reading the specification in the given notation.
- **Secondary Notation** — is related with the availability of mechanisms for expressing extra information, but not in the syntax of specification. Examples are comments, annotations, colors, and others which give information that sometimes is not possible (or is hard) to infer from the formal syntax only.
- **Viscosity** — refers to the difficulties observed at the time of changing something in a specification. Normally it can be measured using the number of actions a user need to accomplish a task. Intricate dependencies on the notation entities can be the cause to make a notation or system viscous. More on this dimension can be seen in [72].
- **Visibility** — is related with the ability of viewing notation components easily. The existence of this dimension (the same happens with all the others dimensions described above) on a notation, depends on the user perspective and usage of the notation. Many times, visibility on notations can only raise problems when performing a comprehension task on a specification.

The CDF is an interesting and usable approach to assess the usability, and to enhance and explain the characteristics of notations. In the following section (Section 2.1.2), characteristics of DSLs are presented, and their explanation is based on some of the CDs presented above.

2.1.2 Cognitive Dimensions Applied to DSLs

As DSLs are notations, the application of the CDF, in order to study the usability of these languages, is a powerful approach. DSLs are different, but somehow share some characteristics. In this context, it is possible to create a shelf for these languages, and evaluate and explain their common characteristics. In [153], the authors used the CDF to accomplish a speculation-based evaluation of DSLs.

In this section, the intention is to do a similar analysis, since a complete empirical experiment is not in the scope of the present work. However in this case, the aim is to enhance, describe and point reasons for the characteristics of DSLs based on the CDs presented in Section 2.1.1.

Normally, DSLs are *small*. The fact of these languages being tailored to deal with the problems of a specific domain, allows their designers to grab only the essential features and concepts of that domain, and manage them to create a small and restricted notation. When languages are small, the proneness to errors is, *a priori*, also small, so this characteristic diminishes the presence of the *error-proneness* CD. Small notations allow the specification of solutions to solve the problems, instead of programming them, unlike what happens when dealing with the major part of GPLs. So, DSLs are usually more *declarative* than imperative languages [196]. This characteristic enables the presence of the *progressive evaluation* dimension, in the sense that specifications can be interpreted at any time, but this is a topic that depends on the system that cope with the DSL (compiler, processor, and so forth) and not on the notation itself.

Moreover, they are *abstract* [82] and *expressive* in their domain [123]. Actually, the latter is not always true, regarding the following law proposed by *Heering* [78]:

$$\textit{Expressiveness} \times \textit{Domain size} = \textit{Constant}$$

When analyzing and designing a DSL, engineers should be aware of the domain where the language will be applied. The semantics of the domain should be implicit in the language notation [83]. The latter means that abstraction should be brought to the notation of the language; that is, the low level notions of how something is done should be encapsulated by a high level notation, expressing, for each sentence or statement, the precise purposes of their existence and usage. Indubitably, this allows their users to easily create mappings between the syntax of the language and the objects of the problem domain. So, these two characteristics, enable the presence of the *role-expressiveness*, *closeness of mapping* and *diffuseness* CDs. On the other hand, the encapsulation of semantic information in the notation, reduces the *visibility* dimension on the language; and, opposed to the *role-expressiveness* dimension, DSLs would present low indices of *hard mental operations*. Also, the *viscosity* of these languages is supposed to be small, because of DSLs being little and abstract languages. In a naif supposition, it is also valid that the *abstraction* dimension is also present, since DSLs are, *per se*, abstract languages and, some of them, present mechanisms to create even more abstraction. On the other hand, the abstraction would increase the number of *hidden dependencies*.

The concentration on the definition of a notation that would only express concepts of a single application domain, brings the possibility of sharpening edges on the

language, and make it more and more *efficient* on various directions. One of these directions is the efficiency on being read and learned by the domain experts [2, 41]. Domain experts are persons with great knowledge on a given domain, but, normally with any or little expertise on programming. Thus, given the abstractness and the expressiveness of DSLs, they can easily read programs, and learn the languages in order to specify the programs with efficiency, or in another words, with little time spent. The same may not happen if the person interpreting the language is outside of the domain.

Another facet of that efficiency can be observed on the tools that give support to the language. Their processors, for instance, can be improved to offer better results, as the domain is restricted and the knowledge is centralized.

2.2 Dichotomies

In this section are used comparisons with both general-purpose and domain-specific languages, to discuss about some dichotomous characteristics on DSLs. For this discussion, the following pairs of characteristics are taken into account: *i) Concreteness and Low-level vs Abstractness and High-level; ii) Computational Power vs Expressiveness*. The following paragraph gives an overview on the concerns of the characteristic written above.

Abstractness and *Concreteness* address the semantics of the language and how it is exteriorized to the user. *Low-Level* and *High-Level* focus on the level where a programmer must think to be capable of working with the language. *Expressiveness* is related with the easiness of mapping the language notation into the real world objects and other inherent characteristics. *Computational Power* concerns with the different possible ways of defining a computation and the capability of defining several things using the same constructors.

2.2.1 Abstractness and High-level *versus* Concreteness and Low-level

What should be answered when it is asked whether DSLs are abstract or concrete? A language is more abstract the more it hides the operational semantics [137] from the final user. One of the main efforts during the construction of a DSL, is to encapsulate the semantical knowledge in the language notation, in order to have it implicitly, instead of explicitly, as happens with GPLs. For this reason DSLs are more abstract than GPLs, which, by exclusion of parts, are more concrete.

In fact, as *Deursen* and *Klint* observed [195], the knowledge about the domain is concentrated in the language notation, and the knowledge about the implementation (the operational semantics of the language) is delegated to the compilers, processors or other related tools that give support to the language usage.

A low level language is a language that takes its user into a thinking level that is known to be unusual for human beings. That is, the handling of constructors and abstractions of a low level language imply the presence of specialized knowledge beyond the empirical knowledge on an application domain.

On the other hand, high level languages, remove from their notations explicit implementation aspects (abstractness). Users are able, then, to handle the language constructors at a more rational level.

DSLs are usually high level languages, when comparing with GPLs. The abstractness of DSLs raise this level, once the semantics of the language are implicit in the constructors and abstractions that the notation of the language presents. This way, the user's concerns are focused on issues associated with the problem and not with the solution (code, program and other intricacies). The gains on following this philosophy are great; using high level languages (DSLs, namely DSMLs) that allow the focus on the problem and not in the solution, can be profitable at earlier stages of the software life-cycle [76, 87], like the requirements analysis and management [8, 86].

2.2.2 Expressiveness *versus* Computational Power

A language is said to be expressive when its notation helps the user on creating mappings between the program and problem domain concepts, without resorting to documentation of the software pieces, whether it is internal (comments, annotations and so forth) or external (user manuals, implementation reports and so forth). Such characteristic is observable when the language is easy to learn, and their programs are easy to read and understand.

On the other hand, the computational power of a language is related with the possibility of specifying multiple and different computations relying on the same vocabulary and structures, offered by the language. Also it is manifested when the notation allows the definition of similar computations, using different approaches.

DSLs are usually, more expressive than GPLs; on the other hand, GPLs offer a greater computational power (concerning the CDs, DSLs present more *consistency* than GPLs). The major difference between these types of languages is precisely this dichotomy. However, it was not always true. Languages like `Fortran` or `Cobol`, can be regarded, nowadays, as GPLs, because of their computational power, but they were first constructed to fulfill requirements on the mathematics and the business areas, respectively [196]. That is, they were tailored for a single domain of application.

The expressive power of DSLs make of them very objective languages, in the extent that the user would know, easier, what happens when a statement of a program is interpreted or executed. The vocabulary of these languages store knowledge of the application domain, and mask behavioral aspects of this same domain. As they are used to cope with the aspects of a single domain, they do not need extra computational power to extrapolate for other domains. The language constructors must encapsulate precisely the behavior required for the concepts of the domain, with which they are associated.

As final words about this dichotomy, *Ladd* and *Ramming* [108] state that DSLs should not be designed to describe computations, but to express useful facts from which one or more computations can be derived. This express precisely what was said about DSLs throughout the present section.

2.3 Pros and Cons

All the characteristics of DSLs, listed and explained before, lead to a group of advantages and disadvantages.

In this section, an impartial and literature-based overview on this matter is given. In order to proceed, it is important to left clear two perspectives on this discussion. On the one hand, there is the usage perspective, which concerns with the usage of DSLs to produce programs or specifications, and other tasks associated, like comprehension, maintenance or evolution of the specifications. On the other hand, there is the creation perspective, which concerns with the development of new DSLs and tools like processors, compilers or interpreters, and also with the maintenance of the languages and the associated tools.

2.3.1 On DSLs Usage Perspective

The most claimed advantage on using DSLs is the possibility of integrating domain experts in later stages of the software development life-cycle [195, 2, 70]. Normally, domain-experts are very required in the analysis phase of the whole software development process, and their functions end right there. But are not few the times that, a new overview on the conceptual aspects concerning the software in production, is needed. Since the usage of GPLs require good programming skills, the domain-experts, who are not proficient on that area, little work can do on this matter. However, using DSLs they can steer the flow on the programming tasks and they can even give a hand on specifying the programs.

In this context, DSLs are also appropriated to diminish the distances that exist between the conception and implementation phases of the software development process. Also, this leads to the creation of new software development methodologies, meeting the requirements that the domain imposes [2].

It is possible to sub-divide the present perspective into two parts: (i) the implementation phase and (ii) the maintenance phase.

Concerning the former sub-perspective, (i), DSLs ease and increase the speed on the construction of software pieces [195]. *Kiebertz et al.* [94] also defined a set of advantages, that, in some extent, empower the existence of the last addressed advantage: they claim that DSLs enhance the flexibility, productivity and usability. The justification for these advantages is trivial by regarding the characteristics of DSLs presented before. All they are due, mainly, to the expressiveness and abstraction of these languages, which are, indeed, the main characteristics. Also, the usability of these languages can be justified by the fact of being small and easy to read and write.

Further in the implementation steps, DSLs would enhance the testability of the programs or specifications being described. In this context, non-traditional methods like [179] can be followed in order to test these programs.

Regarding the latter sub-perspective, (ii), there are many advantages. The main one is that using DSLs the software maintenance is simplified [195]. Completely related with this one, is the claim that these languages enhance the comprehensibility of programs and specifications [204, 105]; which is not any novelty, because the

easiness of maintaining a piece of software implies the easiness of comprehending the program specifications. To this advantage contributes the easiness on inferring the domain of the language, and on creating mappings between the program and problem domains. Another great aspect of these languages is that, in many cases, they provide self-documentation, what avoids the search for documentation resources that may be unavailable. Together, these three aspects diminish the costs of engineering and reengineering, and increase reliability and repairability on the software constructed with DSLs [81].

DSLs are claimed to be a good approach for software reuse [107]. In this context, not only the pieces of software are reused, but also the knowledge embodied in the language. So, the reutilization is another advantage that is connected to the usage of DSLs.

Until now, only advantages were pointed. However DSLs have also some drawbacks associated, concerning their usage. Some programmers create some resistance to use DSLs, because it may be necessary many DSLs to do the job of a single GPL. This dues to the fact that DSLs are tailored for a specific domain. So the programmers claim that they do not need to learn many DSLs when already know one GPL. The point is, if there is not adherence on a DSL, it would not have any success, and its evolution or work done upon them, like tool support construction, would be tasks without future; on the other hand, the adherence implies teaching costs, which can be significative [195]. Although the latter has been pointed as a disadvantage, it is believed that it is easy to learn a DSL with little effort.

Moreover, this variety reduces the easiness of interoperability with other well established languages [70]. This is obviously a disadvantage, because in real projects, several domains can be addressed and, thus, several DSLs can be used; and even the combination of DSLs with GPLs is a reality. So, without interoperability between them, DSLs can fall into disuse.

Missing tool supporting for monitoring, debugging and other necessary tasks for DSL users is, indubitably, a great disadvantage. Without tool support, the availability of DSL is limited [196], what leads to low number of adherents on such language.

2.3.2 On DSLs Development Perspective

The development of DSLs allow optimization and validation at domain level [33, 13, 122]. The optimization of a language concerns with details of implementation at machine level, e.g. managing the memory allocation. As was stated before, these kind of implementations are done at language development time, and not at programming time, providing abstractions that encapsulate these details. So, at programming time, the user is able to build specifications and optimize them at domain level, because the optimization at machine level, is already done. For instance, in \mathbb{C} , memory allocation is an indispensable task, but it takes the user to cope with details that are, most of the times, beyond their capabilities.

But this raises the majority of the disadvantages on developing DSLs. Firstly, in order to develop a language, the engineer must be expert on both the domain of application and compilers engineering [195]. Most of the times, the language engineers are not proficient on the problem domain, so creating a language is not a

single-man task; domain experts are needed to steer the DSL requirements [103], and language engineers are needed to concretize the requirements into an abstraction capable of coping with the domain concepts.

This fact empowers the consumption of time and money on the several phases of the language development (design, implementation and maintenance) [196]. In fact, developing a language requires knowledge about programming in GPLs, on most part of the cases. The issues related with the maintenance of GPLs are widely known. So, maintaining DSLs and their associated tools, like compilers, processors or interpreters, can be a very complicated and difficult task. Not for so little times, this leads to low number of tool supporting; and when there are tools, they may not follow up the evolutionary trends of the language [70].

The disadvantages on the present perspective, can be attenuated regarding the methodologies used for the development of DSLs. Section 2.4 gives a little survey on some development methodologies that were successfully used to implement DSLs, over the times.

2.4 Developing DSLs

Language engineering is an old discipline perfectly established as a branch of the software engineering. The development of DSLs is just a small part on that branch. Unlike the development of GPLs, which is, normally, based on compiler techniques [1], the development of DSLs follows several methods and techniques, including also the techniques used for creating GPLs [123].

Each development methodology is well supported by tools. Not all the tools were tailored to cope with the development of DSLs, but they are successfully and easily adapted to cope with it.

The following sections address the DSLs development techniques, their associated classifications and the tools used for their crafting. The knowledge present in these sections is not relevant for the work of this master's thesis, thus this small survey does not go deep into details.

In general, five main steps are pointed to be essential on DSL development: decision, analysis, design, implementation and deployment [123, 196]. The decision phase concerns with the analysis of pros and cons about developing or not a new language. The analysis is the phase where domain experts gather knowledge related with the domain, relying on domain analysis methodologies like *Feature Oriented Domain Analysis (FODA)* [90], *Domain Specific Software Architectures (DSSA)* [188], *Organization Domain Modeling (ODM)* [178]. *Domain Analysis and Reuse Environment (DARE)* [67], *Family-Oriented Abstractions, Specifications and Translations (FAST)* [202] and *Ontology-based Domain Engineering (ODE)* [64]. The design step is concerned with the choice and adoption of patterns and conceptual implementation decisions. The implementation phase, is related with the concrete construction of the DSL, using the patterns and implementation approaches adopted in the design phase. The deployment (or distribution) step is when the language is ready to be used, and is made available for general usage.

During these phases, patterns can be used to ease the process of language development. *Spinellis* [181], reinforced later by *Mernik et al.* [123], defined a set of several design patterns. Some of them are listed below with a snippet of their description. For further reading about these patterns, the given references are recommended.

Piggyback The capabilities of existing languages are used as host to the development of new DSLs;

Pipeline DSLs are constructed in a way that the input of one is the output of another;

Lexical processing The languages are designed with the intuit of not using syntax analysis, but only lexical scanning.

Language extension DSLs are constructed in order to add features to existing languages. It can be seen as an extension of the base language;

Language specialization The languages created remove features from existing languages which serve as base;

Source-to-source transformation The source code of a DSL is transformed into the code of an existing language;

Data structure representation DSLs are designed to represent data structures;

System front-end DSLs are constructed to be the front-end of a system, in order to configure and adapt the underlying system.

These patterns are basis for implementation approaches. In [204], three main shelves were identified to contain such approaches. In the list below are presented the containers and the approaches contained in each one. For each implementation approach a small description is given.

2.4.1 Complete Language Design Approaches

The approaches contained in this shelf involve the creation of a language from the zero. This requires the definition of the language syntax (commonly achieved by using *Backus-Naur Form* (BNF) or *Extended Backus-Naur Form* (EBNF) notations), the specification of the semantics, and the translation into target code. *Compilation* and *Interpretation* are the main approaches within this container.

Compilation. It is the most traditional approach to implement a language, no matter the language is a DSL or a GPL. The language constructs are analyzed and synthesized into a target code, that can be machine code or a common GPL language code, that is to be executed by a machine. The common tasks performed by compilers can be seen in Figure 2.1. Besides the tasks, it is also presented auxiliary data-structures and mechanisms, used to communicate between tasks, and assess the correctness of the programs.

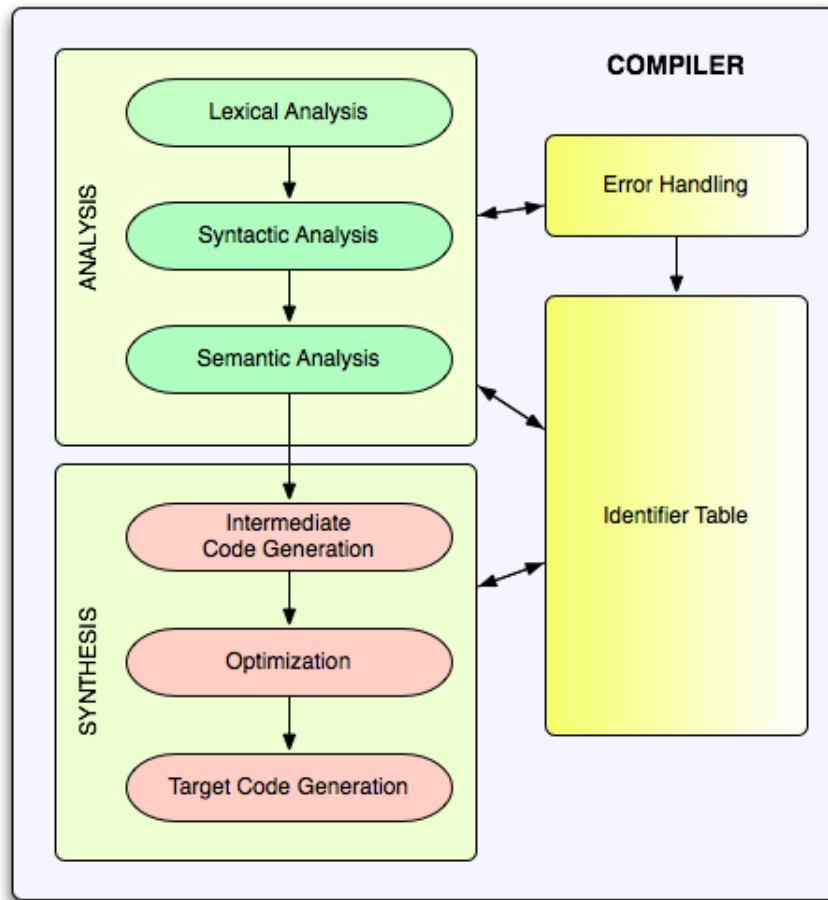


Figure 2.1: Overview of Compiler Tasks

Interpretation. The difference to the compilation approach is that the language constructs are recognized but not executed at machine level; instead they are interpreted.

These implementation approaches are supported by compiler-compilers (tools for compiler generation) that are widely available for many years. The `Lex` [112] and `YACC` [88] system, used for the development of any kind of language, is one of the most popular; but more recent and similar tools (in the extent that are not only focused on the development of DSLs) are available: `JavaCC`¹, `SableCC` [69], based on translation grammars, and `LISA` [125], `AntLR` [146], `JastAdd` [61], based on *Attribute Grammars* (AG) [101], are some examples. On the other hand, there are tools specified on the construction of compilers and interpreters for DSLs. `ASF+SDF` [194], `Khepera` [63], `Kodiyak` [81], `InfoWiz` [134], are some examples of these specialized tools.

The languages created with these approaches are said to be *external languages* [164], because they are independent languages; that is, the DSL constructed on this strategy is completely designed and implemented from the scratch, not relying on any pre-existent language.

¹Home page: <https://javacc.dev.java.net/>

2.4.2 Languages Extension Approaches

In this container are placed the approaches for DSLs implementation using GPLs, or components of these languages, as a start point. In this context, two approaches are considered to be the most used: *Embedding* and *Extensible compiler/interpreter* approaches.

Embedding. In these approaches, the developer does not need to have compilers expertise, because all the work of semantics verification and machine code transformation is delegated to the compiler of the host language. Nevertheless, the syntax of the DSL must be designed, but this time is used the host language syntax to develop the new DSL constructs. The construction of an *Application Programming Interface* (API) for a given application domain, is considered a concretization of this approach.

The advantages on embedding languages are considerable. Starting on the fact that it needs no compiler knowledge, to the fact that the compiler of the host-language is completely reused; from the point that the development of the language needs only knowledge on the domain and on parts of the host language, to the point that, for usage purpose, it is not need to know the underlying language.

The languages implemented with this approach are called internal or embedded languages [82]. Almost any GPL can be used as host-language, but some have more appropriated characteristics and are frequently used: Ruby [44, 47], Python [148], Haskell [82, 155, 154], Java [68, 89], C++ [163] and Boo [164] are some examples of utilization.

A specialization of the embedded languages was introduced by *Fowler* and *Evans* [66], and is called *Fluent Interfaces*. These languages still being classified as embedded, but their construction upon a GPL follows a methodology enabling a more flexibility in the syntax, so writing operations is no more than chaining function calls, what allows a more fluent specification.

Extensible compiler/interpreter. This approach is very similar to the compilation approach presented in Section 2.4.1. The main difference is that an existent compiler of a GPL is extended with domain-specific aspects in order to add domain-specific constructs to the underlying language, rather than creating the compiler from the scratch. This way, the task of creating the language is easier than using the other approach, but attention must be paid when modifying the compiler, in order to not screw up the base language.

2.4.3 COTS Products-based Approach

Commercial Off-The-Shelf (COTS) products, as the name induces, are very accessible. In this context, the implementation of DSLs followed the trends of the crescent usage of such products. The idea on this approach is to specify the language structural aspects in terms of these tools [204].

The most used COTS-based product is the XML. However other products like *Microsoft Access*, *Microsoft Powerpoint* and *spreadsheet representations* can be used as claimed by *Wile* [204]. XML gives a fairly easy to use approach to write new DSLs.

The tools associated with it increment flexibility on creating processors for the language. However, the XML dialects tend to be very verbose, and hard for humans to read.

Summing up this discussion about the implementation of DSLs, *Kosar et al.* [105], in their study on the implementation of DSLs, concluded that the extension of languages by the embedding approach are the most efficient approach to be used when developing DSLs. Moreover they state that a great alternative to this approach is to use the compilation approach, because of the possibility of handling minor aspects and having more control over the language implementation.

2.5 Application Domains

The applicability of DSLs is widely spread through many domains, so, it is not a surprise when it is said that hundreds of languages have been developed until today [96]. In this section is given a sample of those domains and languages tailored for them, following the surveys on this matter [196, 204, 123].

Table 2.1 lists some of the domains to where DSLs were constructed. The first column shows the application domains, and the second associates names of DSLs used on each domain.

Table 2.1: Sample of DSLs Application Domains

APPLICATION DOMAIN	LANGUAGE EXAMPLES
Finances	RISLA
Business	Cobol
Databases	SQL
Web Computing	Mawl, StruQL
Imaging	Envision, PIC
Graphs/Diagrams	DOT
3D Animation	HAL
Communications	PRL, Promela++
Simulation	SHIFT, APOSTL
Robotics	ARCL, Karel
Mathematics	Mathematica, Matlab, Fortran
Compilation	YACC
Software Building	Make
Spreadsheets	Excel
Syntax Specification	BNF, EBNF
Web Development	HTML, CSS
Typesetting	LaTeX
Hardware Design	VHDL
GUI Construction	SWUL, XAML
Data Processing	Hancock

2.6 Summary

The fact of DSL being tailored for a specific domain, gives great advantages and add a reasonable number of characteristics that make them different from GPLs: they are high-level, small and declarative (most of the times), abstract, expressive and efficient. However, in [97], the author concludes that any of the characteristics of DSLs make them different from the the other software languages. Maybe this is a correct statement regarding a specific perspective like their existence purpose, nonetheless, in terms of using them and understanding their programs, they really mark a difference.

These characteristics enable the presence of many advantages like the fact of allowing the insertion of domain experts (usually non-programmer persons) in advanced phases of software development. However their development is not an easy task and requires, in the major part of the times, compiler construction knowledge.

Nevertheless, these difficulties can be soften regarding the variety of implementation approaches that have been successfully used. The embedding approach, where a GPL is used as basis to develop a new DSL, is claimed to be the most flexible and valuable approach.

An important conclusion about these languages is that is easier to infer its application domain, and to identify and create mappings between the program and problem domain concepts. This way, the comprehension of DSLPs is achieved with more ease.

In this chapter, DSLs were introduced and a deep study on them was provided, addressing *(i)* their characteristics; *(ii)* vantages and disadvantages on their usage and development; *(iii)* methodologies of development, including design patterns and implementation approaches and finally *(iv)* application areas and examples of existing DSLs.

Chapter 3

Program Comprehension

Testing leads to failure, and failure leads to understanding.

Burt Rutan

Software is developed through several phases [85,77]. Some may think that developing tasks concern just with the phases of creating the software pieces, but there is more beyond that: after a software piece is completely implemented and tested, its development continues to the maintenance phase. As already introduced before, this is the most expensive and time-consuming phase [29, 65, 207], and much of it is due to the fact that software must be comprehended.

Software maintenance relies on several activities that have the necessity of programs to be comprehended, or are means to achieve that comprehension:

(i) *Reverse Engineering*, is an activity to gather information about the interrelationship of a system's components, and to create a new representation, usually more abstract, of the same system. Its dual is known as *Forward Engineering*.

(ii) *Design Recovery*, is similar to reverse engineering, but it takes advantage of resources like documentation and domain knowledge, to gather the information about design decisions, so it is possible to define an abstract representation of the system under study;

(iii) *Restructuring*, is the activity of modifying the system's representation but preserving its initial abstraction and semantics. A common practice is to change source code structures into more efficient or modern structures;

(iv) *Reengineering*, is a twofold activity: first, an analysis of the system is made (usually relying on reverse engineering) and then, using the knowledge base gathered in the first step, the system is re-implemented in a new form, in order to meet the changes needed. Figure 3.1 shows how reverse engineering and forward engineering activities may be composed, in order to originate a maintenance methodology.

Above were listed only a few activities that are related with PC and to maintenance in general. More on this matter can be seen in [37].

In this context, *Program Comprehension* (PC), defined as:

A discipline of Software Engineering aimed at providing models, methods, techniques and tools, based on specific learning and engineering processes, in order to reach a deep knowledge about a software system. [20].

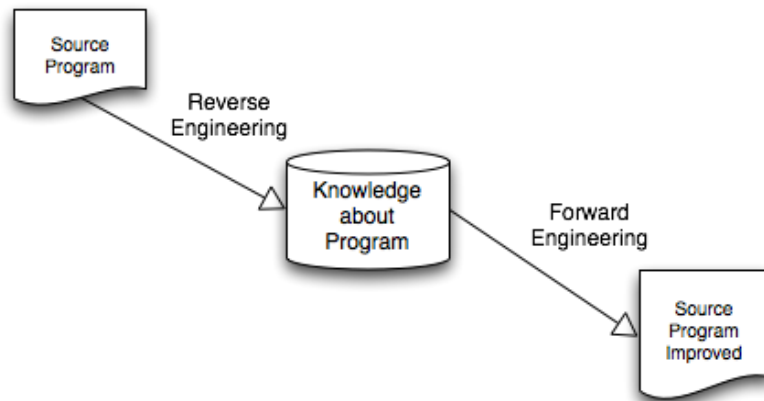


Figure 3.1: Reverse Engineering and Forward Engineering Activities Combined to Implement a Maintenance Methodology — Reengineering

emerged as an useful and interesting research area [199, 136, 138, 185], to leverage the work of maintaining and evolving software. But comprehending code is not an easy task. Some empirical studies have been made in order to gather information about the difficulties on comprehending programs and how to improve it [166, 129, 91]. A great contribute on the identification of difficulties associated with PC is given by *Rugaber* [172]. The author identifies five gaps between different conceptual areas, and states that program comprehension is compromised by the difficulties on bridging those gaps. In an abstract point of view, those gaps can be turned into one single gap: the gap between the problem and the program domains, as shown in Figure 3.2.

The research involved in PC, walks around this gap, trying to fill it, for more than three decades. This problem led researchers to the construction of theories and cognitive models [189, 200, 208, 75], which have been applied in building *PCTools*, to (semi)automatize the program comprehension task. Figure 3.2 presents a common PC process sketch, identifying the domains of knowledge and the usual activities performed in each one. In addition, is marked the direction of the principal theories on program comprehension: *Bottom-up* and *Top-down*, which are addressed later in this chapter.

But before providing information on program comprehension theories and cognitive models (Section 3.3), important and commonly used terms in PC area, are explained, in Section 3.1. In Section 3.2, a general overview on Software visualization, is given. Then, in Section 3.4 some *PCTools* are surveyed, but not with the objective of giving a complete description of these tools; instead the aim is to show what kind of information they present to the users, to enhance the program comprehension. Finally, in Section 3.5 work related with the one discussed in this dissertation, is presented; there are presented approaches and tools that, on the one hand, take advantage of the problem domain to enhance the comprehension of programs, and on the other hand, deal with DSLs.

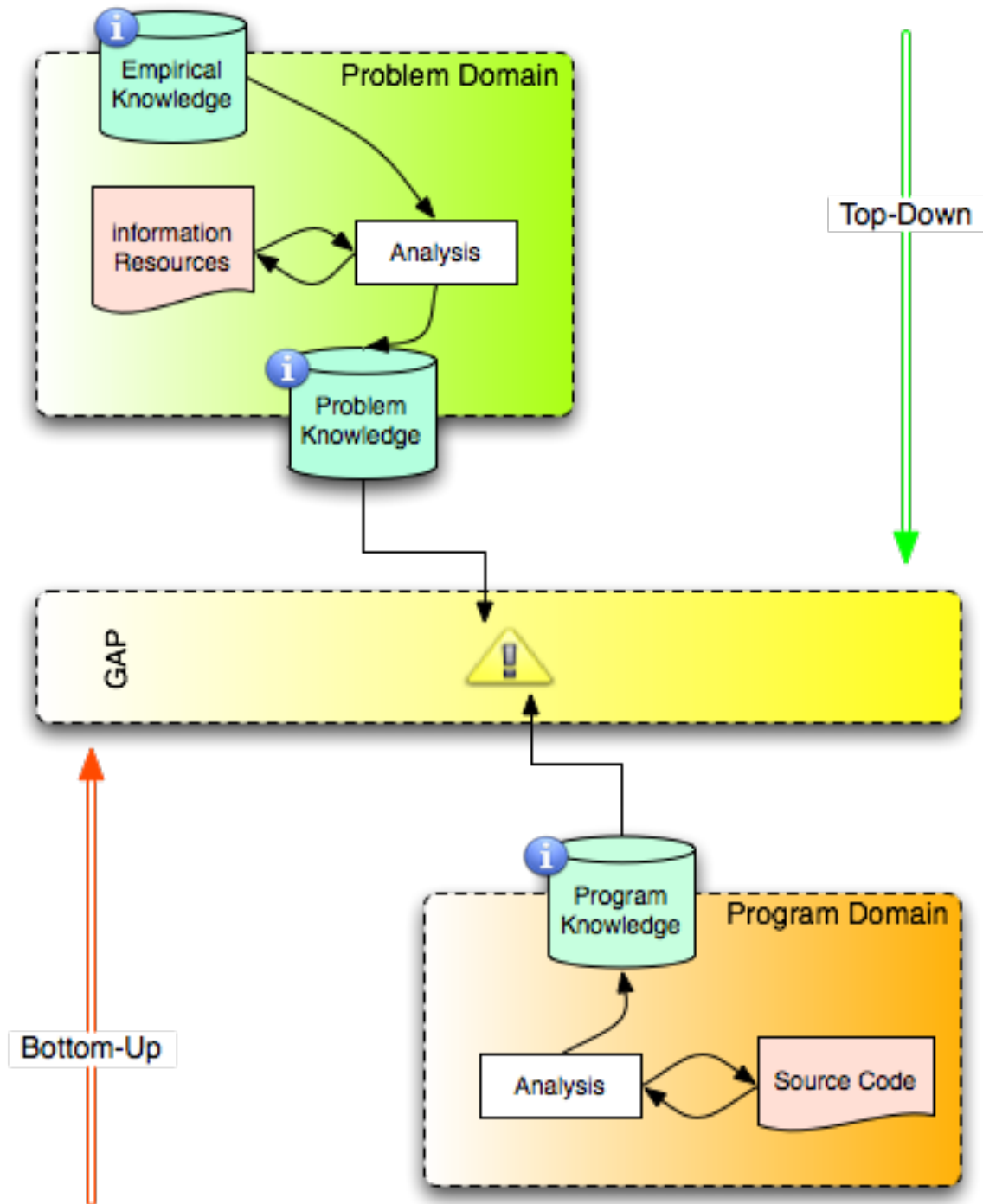


Figure 3.2: Gap Between Program and Problem Domain

3.1 Fundamental Definitions

PC is an area of research, as many others, where specific terms are used to foment the dialogue and establish a comprehension between the researchers. Throughout the previous chapters, some of these terms have been, inevitably, used, but without been given a complete definition or explanation. In order to converge the meanings, some of the more relevant terms to establish a perfect comprehension between the reader and the message being passed in this dissertation, are defined and explained in the following sections.

3.1.1 Program Reader

Although the name is not perfectly established, *Program Reader* (PR), or simply *Reader*, is the most common name used, by some authors [172, 113], to identify the person who is being through the process of understanding a program.

Reverse engineer or *programmer* are also used. But these are narrower terms, when comparing with PR. A programmer is normally connoted as a person who implements a program and little times is seen as someone who tries to understand a program. Reverse engineer may be a better approximation, but it seems, somehow, to infer that it is a task performed by some specialized person — some times it is not.

Thus, henceforth, the term *program reader* (or for short, PR) will be used to refer to the person who performs mappings between the problem and program domains, in order to understand the program as a whole.

3.1.2 Problem Domain

Problem domain, also called *application domain*, is the term used to refer to the knowledge and the information resources on the area where a software system was developed to. In PC, the problem domain is also regarded as the domain of knowledge which “*is concerned with the effect of the program execution, i.e., the final result produced and the impact at the level of the problem to be solved*” [20].

Normally, it is regarded as a set of concepts which are closer to the humans perception. These concepts are usually gathered resorting to domain analysis and ontologies [6, 64, 111], defining and describing the problem area.

3.1.3 Program Domain

Program domain is the term used to refer to the program level of a software system, that is, to the source code of the software programs and to the low level concepts associated with it. This domain “*is concerned with technological issues of the programming language (...), and how the program is executed to produce an output*” [20].

At program domain, the sentences of the program, composed of statements or function calls, are the resources to raise information that helps to comprehend a program, namely its data and control flows.

3.1.4 Domain Concepts

In [165], *Rajlich and Wilde* define *Domain Concepts* or just *Concepts* as

Units of human knowledge that can be processed by human mind (short-term memory) in one instance.

Other authors have a more pragmatic idea about concepts on programming perspective, describing them as equivalent to the objects of the object-oriented programming, however it seems to be a limited approach. Others see them as attributes and subsets of attributes constituting lattices [115]. Another perspective about concepts can be retrieved from [149], where the authors present a concept as a *brick* representing an informational idea, capable of being present in one or several domains.

In PC, the concepts are very important, because it is based on them that the knowledge about a program and the problem is interconnected. However, these interconnections are hard to do, because high level concepts at source code are spread all over the code; thus, many studies and approaches for their localization in the source code have been done for years [26, 114, 53, 121].

3.1.5 Mental Model

Mental Model [43], is a central term in PC research. A good summary on the story of its usage and application in PC or cognitive sciences is provided in [133].

In simple words, a mental model is the internal representation of the program being studied, created by the PR [200]. This means that, somehow, the program reader creates an internal representation (at human mind level) of the connections, dependencies and other dynamic and static aspects of the source program.

3.1.6 Information Extraction

Information Extraction is a discipline focusing on the extraction of relevant information, from the source code or other information resources of a system, in order to present it over several perspectives. Information can be extracted, mainly, by following static or dynamic approaches.

Compiler techniques [1, 9] are traditionally used for static information extraction. Also slicing techniques [191, 25] can be used to retrieve static information from code, in order to know the influence of parts of the program in the whole program or system.

For dynamic information extraction [210], other techniques are used. Code instrumentation [190, 24] is an example. With this technique is possible to know, in run-time, which are the objects, functions, variables and so forth, of the program that are being used or called.

The terms enhanced in this section are the most relevant to establish a perfect symbiosis between the reader and the text. However, other PC-specific terms are used in this dissertation. As the work made upon them is related with the topic of this thesis, they will be explained in more detail in Sections 3.2 and 3.3.

3.2 Software Visualization

Software Visualization (SV) [156, 56] is a discipline of the software engineering that have been evolving along with the trends of the PC. It provides the visual representation of the information present in software systems, allowing a better comprehension of such information [23]. *Price et al.* defined, theoretically, SV as follows:

Software Visualization is the use of the crafts of typography, graphics design, animation and cinematographic with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and the effective use of computer software [161].

and in [36], the authors refer to SV tools as systems used, mainly, to support the *navigation* and *exploration* of the software information resources, to give *insight* about the complexity of information, and to support *communication*, giving a more technical approach to that definition.

3.2.1 Software Visualization Systems — Categorizations

In order to materialize the techniques and approaches researched on the SV area, tools and complete systems have been developed. Some of these tools are *stand-alone*, in the extent that have no other purpose besides the visual representation of the software information. But the major part of these tools are known as pure `PCTools`¹, or are sub-systems of such tools, because the visualization of the information is an important part on the PC approaches. These systems have been categorized regarding their features and functionalities, throughout the years. Taxonomies on this matter have been created, to classify the systems and to guide their developers. Before presenting taxonomies, is worthwhile to know that SV systems can be grouped into *Algorithm Visualization*, *Program Visualization* and *System Visualization*, depending on their scope and applicability. Algorithm visualization refers to the visualization of algorithms and is applied on teaching situations. Program visualization refers to the visualization of programs, regarding the interconnections between the components. System visualization englobes all the other ones [23], because is used to visualize the algorithms, dependencies and functionalities of programs that compound a complex and multi-module software system.

Myers [132] classify software visualization systems using a three-by-two matrix. The possible classifications are (i) Static Code Visualization; (ii) Dynamic Code Visualization; (iii) Static Data Visualization; (iv) Dynamic Data Visualization; (v) Static Algorithm Visualization and (vi) Dynamic Algorithm Visualization.

Price et al. [162, 160] provide a classification framework composed of six major sections: (i) Scope; (ii) Content; (iii) Form; (iv) Method (v) Interaction and (vi) Effectiveness. Each one of these master terms are then subdivided into smaller parts to which the authors called characteristics of SV systems.

The work of *Roman and Kenneth* [170] in this area, results in a taxonomy similar to that of *Price et al.*, referred before. The taxonomy presented by the authors is

¹See Section 3.4 for an overview on these tools.

composed of five main classification terms: *(i)* Scope; *(ii)* Abstraction; *(iii)* Specification Method; *(iv)* Interface and *(v)* Presentation. Again, these terms are subdivided into smaller ones, used to characterize the tools for software visualization.

The works referred in the previous paragraphs are the most influent on this area of SV systems classification. But all these works are only concerned with the visualization of the program domain, giving little importance to the problem domain visualization. Recently, *Berón et al.* [20], presented an extension, by providing a taxonomy for classifying SV tools that address the problem domain. The authors claim that the terms used to classify other SV tools are suitable for this new taxonomy, but they need to be given a new and coherent semantics.

3.2.2 Visual Representations and Their Requirements

The taxonomies are used to classify the systems according to the information that is presented and how it is presented. The visualization of abstract information (for instance, software data structures, algorithms and so forth) is neither easy nor perfectly established [145, 169]. The developers create their own visualization of the software systems and the concepts involved. However they are always restricted to represent that information using 2D or 3D graphics.

The different perspectives, showing static and dynamic information of the software information in SV systems, are called *Views* [24]. The same information resource can be seen over several perspectives, and several approaches can be adopted to do its visual representation. For instance, the source code of a program can be seen using a textual view, but it can also be seen by means of graphs concerning the dependencies between concepts in the program domain [157]. In [23] and [50], the authors refer to a considerable number of possible views to represent the information. Some of them are the module dependency graph, function graph, runtime function graph, type dependency graph and so on. The previous list of perspective approaches resort to graphs. Other important views of a program are the *Identifiers Table* (IT), the source code, itself and the documentation associated with it. All these approaches are based on text and 2D graphics.

Recently, as the improvements on computer graphics are evolving so fast, several works on representing software using three-dimensional approaches have been done. Some authors keep using graphs (but in a 3D approach) to represent the information [168, 40], others are more concerned on giving a new perspective to the code visualization based on metaphors [120]. Virtual reality is also an approach that some authors follow to develop their SV systems [99, 30]; while others are focused on viewing software systems as complex cities [100, 203].

All the 3D approaches seem to be very interesting, but the utility and usability of the associated tools are badly tested, what points that there is no reliable results to affirm that 3D visualizations are better than 2D. At the end, what matters is that the tools should hold the user (the PR) requirements for a better program understanding. *Kienle and Müller* [95] have created a list of such requirements, regarding them as quality and functional requirements. Bellow is given a summary of each one.

- QUALITY REQUIREMENTS

Rendering Scalability The systems' rendering speed should scale up whether the amounts of data to represent is small or great;

Information Scalability The systems should behave equally, whenever the amounts of data are small or great. This aspects concerns the approach followed to represent the data.

Interoperability The systems should provide information capable of being used by other system, whether it is a **SV** system or another kind of system, for instance, measurement systems.

Customizability The systems should be customizable, in order to hold needs that can not be forecasted at developing time.

Interactivity The system should allow users to manipulate the information that is represented, in order to have better understanding of the abstract concepts represented visually.

Usability The systems should be usable. This is a subjective aspect of the systems, in the extent that the opinions may vary. Nonetheless, the tools should be tested against their usability.

Adoptability The systems should provide features to solve needed tasks, otherwise, users will not use the tools, leading to its unsuccess.

- **FUNCTIONAL REQUIREMENTS**

Views The system should provide as many views as possible, in order to praise any kind of stakeholder.

Abstraction The system should give the user the possibility of working and visualize the information on varied levels of abstraction.

Code Proximity The system should provide a raw view of the source code, instead of hiding it, because the source code is always the most important resource to start the exploration of the program.

Automatic Layouts The system should offer the capability of automatize the way the information is displayed in the screen.

Miscellaneous The systems should provide tools and mechanisms that other systems present as are examples the colors, the search filtering and history mechanisms, zoom-in and zoom-out of some views, deletion and edition, and so forth.

Some of the requirements listed above are, in some cases, trivial to offer, as they are generic requirements for software development and several approaches have been proposed. Nonetheless, others are rather complex, and some flaws on **SV** systems come from the absence of such requirements. *Henríquez* [80] fundamentes this statement by claiming that one of the greatest difficulties on the development of **SV** systems is to cope with the scalability problem, i.e., the size of the programs that will be visually explored.

3.3 Cognitive Models

A *Cognitive Model* is a means to create a mental model. That is, a structured way, adhered by the PR, in order to retrieve and gather knowledge from the source program [209]. In [133], the authors refer to a cognitive model as a way of “*understanding how humans know, perceive, make decisions, and construct behavior in a variety of environments*”. From this definition, is possible to see that such term is used not only to describe the process of creating mental representations of a program, but also to explain how, and based on what, the PRs do it.

Cognitive psychology is a term introduced by *Ulric Neisser* [135] and was studied by very well known psychologists, namely *Jean Piaget* or *Saul Sternberg*. The objective of this discipline is to study internal mental processes, including memory, language and problem solving. The study of how persons use cognition to solve problems is in the basis of the cognitive models. In fact, psychological studies on how people learn things are really close to the studies done in program comprehension, thus, the methods and theories that outcome from cognitive psychology can be applied to this area of computer sciences [128].

Throughout the years, many models and theories of cognition to support the program comprehension, have been constructed. The following sections present a fast overview on them.

3.3.1 Top-Down Model

In a top down model of understanding a program, the PR starts from a higher-level domain (top) and goes down to a lower-level domain (bottom). Figure 3.2 identifies the top level of a top-down approach for program comprehension as the level of the problem domain. The process is finished when a mapping of the knowledge about the problem to the knowledge of the program is achieved.

The most influent work on this model was the theory developed by *Brooks* [32], in 1983, being followed by *Soloway and Ehrlich* [180] in 1989.

Brooks' Theory

In his cognitive theory [32], *Brooks* defends that program comprehension is attained when mappings between the problem and program domain are established. The theory is based on hypotheses. That is, the PR creates a primary hypothesis about the program, a very generic one, and goes down to the program domain, discovering the answer. Meanwhile that primary hypothesis is subdivided in a hierarchic structure of hypothesis that would all be answered at the program level.

The hypotheses are corroborated, or not, due to the existence of *beacons* [32] in the code. Beacons are a set of features, marks or structures commonly used to identify concepts in the programs. *Storey et al.* [183] exemplifies that a common beacon is the function `swap` in a sorting program.

Soloway and Ehrlich's Theory

Soloway and Ehrlich's theory [180] is similar to that of *Brooks*. It also follows a top-down approach, however it is based on programming plans and rules of programming discourse. The former are “*generic program fragments that represent stereotypic action sequences in programming*” [180]; and the latter “*capture conventions in programming and governs the composition of plans into programs*” [180].

The authors claim that these are features that only expert programmers have. Thus, their theory is only observed on expert PRs. The theory says that in the beginning, the PR has an overall idea about the functionality of the program, and through a hierarchy of plans and goals, where beacons and rules of programming discourse help on creating the mappings between the higher level (problem domain) and the lower level (program domain).

3.3.2 Bottom-Up Model

As opposed to the top-down, in the bottom-up model, the PR starts from a lower-level domain (bottom) and goes up to a higher-level domain (top). Figure 3.2 identifies the bottom level of a bottom-up approach for program comprehension as the level of the program domain. The process is finished when the knowledge about the program is converted into abstraction that can be mapped to the knowledge at the problem domain.

The most influent works on this model were the theories developed by *Shneiderman and Mayer* [175], in 1979, and *Pennington* [151], in 1987. Another work, although not a theory, is that of *Basili and Mills* [12], in 1982. This one is not addressed in this document.

Shneiderman and Mayer's Theory

Shneiderman and Mayer's theory [175] is based on programming knowledge, which the authors state as being twofold: syntactic and semantic knowledge. These *domains* of knowledge derive into (i) Syntax Knowledge, related with the details of the programming language; (ii) General Semantic Knowledge, related with the knowledge of not domain-specific general concepts in programming; (iii) Task-related Semantic Knowledge, directly concerned with the problem domain knowledge.

The theory tells that the PR, using his own programming knowledge, performs tasks of abstracting source code fragments into chunks (which can be seen, for simplicity, as code fragments [7]). These chunks are gradually revisited, under the several parts of the programming knowledge, until they are mapped into higher level concepts.

Pennington's Theory

Pennington's theory [151] states that the PR starts his comprehension process by collecting information from the source code, in order to create an abstract control-flow, so the sequence of the program's execution is perceived. To this, the author called *program model*. Then, also in a bottom-up process, but requiring problem

domain knowledge, the PR creates a *situation model*, concerned with knowledge on data-flow and functional abstractions of the program.

3.3.3 Hybrid Cognitive Models

Hybrid cognitive models are cognitive models that embody both top-down and bottom-up processes of program comprehension. Several researchers argue that the process of comprehending programs is not as rigid as following a top-down or a bottom-up approach, so they came up with their own models and explanations.

Letovsky [113] (1986), provides an opportunistic view of the process of comprehending programs. His cognition model defines three main components: (i) Knowledge base, which stores the knowledge acquired by the PR during his experience in the area; (ii) Mental model and (iii) Assimilation process, which uses the knowledge base and any other resource (from documentation to source code) to generate and improve the mental model. In this component is where the PR may opt for a bottom-up or top-down approach, taking into account the one that is more valuable to finish the comprehension process.

Littman et al. [117] (1987), claim that strategies for program comprehension process can be systematic, following one of bottom-up or top-down approaches, or *as needed*, that is, depending on the needs of the PR, in a given moment, it can be changed from an approach to another, in an opportunistic manner. They conclude that when adhering to a systematic strategy, the time consumption is higher, but the knowledge gathered is also higher, whilst the *as needed* strategy is the opposite: takes less time to have a comprehension of the program, and the knowledge gathered is focused on fragments of the system, and not as a whole.

Koenemann and Robertson (1991), based on the results from an experiment, claim that “*program comprehension is best understood as a goal-oriented, hypotheses-driven problem solving process*” [102]. The authors also state that PRs follow *as needed* rather than systematic strategies, because readers focus on the parts of the code that are relevant for solving a task. Bottom-up approach is used when hypotheses fail.

von Mayrhauser et al. [198] (1993), collect information from the models presented by *Soloway and Ehrlich* [180] and *Pennington* [151, 150], and build the Integrated Model. This model embodies the program model, situation model, the top-down model (also called domain model) and knowledge base. Depending on the experience of the PR, the integrated model can be started by a top-down approach, if the PR has experience on the domain; or by the construction of the program model, otherwise.

All the theories and models of program comprehension addressed above, are old, regarding their publication dates. However they are so well assessed that convince any researcher on the area. So that, there is not much recent research studies on cognitive models. However, in 2005, *Xu* [208] presented a new theory based on the constructivist learning theories [62, 119] and on *Bloom and Krathwohl’s* taxonomy of cognitive domain [28]; and more recently, in 2009, *Guéhéneuc* [75] proposed a theory of program comprehension, joining both program comprehension and vision science aspects, in order to extend *Brooks’* [32] and *von Mayrhauser’s* [200] models.

3.4 Program Comprehension Tools

Program Comprehension Tools (PCTools) are software systems aiming at helping the PR on its program comprehension task.

PCTools are widely available as free software or commercial products, and a considerable number of studies on their features and characteristics has been done [14, 177, 185, 52, 15], in order to provide both the state of the art, and to offer the PR good bases for choosing the most appropriated tool for the work being realized.

In this section, just a small set of tools is provided. The intent is not to go into a deep description of their features, but to present the software system, by presenting the information they provide, as output.

3.4.1 The Tools

This section presents the state of the art on PCTools. In [185], *Storey* described a considerable number of such tools. As far as it is known, this is the most recent survey addressing general tools for program comprehension. In this context, tools are summarized below, following *Storey*'s work until the year of 2005; and from 2006 to 2009, are presented the newer PCTools developed meanwhile. The tools are listed in an alphabetic order, thus, not regarding their age.

Alborz Alborz² [174], is a general purpose system for reverse engineering, focusing on the recovery and evaluation of a software system's architecture. It uses several techniques for extraction of static and dynamic data, in order to create several views of the software system.

Alma Alma³ [48], is a language-independent system for program animation and visualization. It allows the inspection of code by providing several views, including the static control flow and dynamic animation of the program interpretation.

Bauhaus Bauhaus⁴ [167], is a tool designed to help users on understanding legacy software systems, by deriving and describing the architecture, and by providing several analyses at different abstraction levels. It copes with Ada, C, C++, and Java programming languages.

Cerberus Cerberus [58], is a threefold system taking advantage of information retrieval, execution tracing and prune dependency analysis, to locate concerns (requirements and features, for instance) in the source code of a software system written in Java.

²Alborz — <http://www.cas.mcmaster.ca/~sartipi/Alborz/static/index.htm>

³Alma — <http://epl.di.uminho.pt/~gepl/ALMA/>

⁴Bauhaus — <http://www.iste.uni-stuttgart.de/ps/bauhaus>

CodeCrawler CodeCrawler⁵ [110], is language-independent reverse engineering tool, combining metrics and software visualization for ease the program comprehension task.

DA4Java DA4Java⁶ [157], is a system designed to visualize and analyze Java source code, by providing nested graphs. Users are allowed to use bottom-up or top-down analysis.

Imagix4D Imagix4D⁷, is a system to help users on understanding legacy systems written in C/C++ and Java programming languages. It provides an automatization of understanding program's control flow, dependencies and so on.

JIRiSS JIRiSS (Information Retrieval based Software Search for Java) [158], is a tool used for exploration of software systems. It uses information retrieval approaches to allow the exploration of Java source code.

PICS PICS [21, 18], is a system based on the presentation of several views of software systems written in the C language. It uses a strategy to interconnect both problem and program domain, easing the tasks involved in program comprehension.

Reflexion Model Reflexion Model tool (RMtool)⁸ [131], is a tool designed to provide system summaries (the reflexion models) to help users on understanding the structure of large software systems.

Rigi Rigi system⁹ [130], is a tool to help users on understanding and re-documenting large software systems. It is a multi-plataform system and copes with C/C++, Cobol and Java programming languages.

SeeSoft SeeSoft [10], is a tool designed to provide source code (line-oriented) statistics of a software system.

SHriMP SHriMP (Simple Hierarchical Multi-Perspective)¹⁰ [186, 184], is a system designed for visualization and exploration of software architecture and related information. For visualization and exploration of Java programs, Creole [116], a system derived from SHriMP, can be used.

Not even all the tools presented above are regarded as PCTools, at least according to *Berón*, who claims that a software system is a program comprehension tool if it provides “*support for mental models, techniques for software visualization and strategies to extract and organize the information*” [19]. From the knowledge acquired

⁵CodeCrawler — <http://www.inf.unisi.ch/faculty/lanza/codecrawler.html>

⁶DA4Java — <http://swel1.tudelft.nl/bin/view/MartinPinzger/MartinPinzgerDA4Java>

⁷Imagix4D — <http://www.imagix.com>

⁸RMtool — <http://www.cs.ubc.ca/~murphy/jRMTool/doc/>

⁹Rigi — <http://www.rigi.csc.uvic.ca/index.html>

¹⁰SHriMP — <http://www.thechiselgroup.org/shrimp>

on studying Cerberus, for instance, it does not provide any kind of visualization subsystem.

However they are all a good indicator that work has been done in the development of tools to help the maintenance of legacy systems, by easing the program comprehension tasks.

3.4.2 Tool's Output Characterization

The tools listed in the previous section help their users on understanding programs, by generating several views of the software system. They all use different approaches to cope with information extraction and its presentation. Some of them present the information by using fancy graphs, while others use only textual approaches. However, the discussion in this section, is not on how they present it, but on what kind of output information they provide.

From the analysis made, a small list of dimensions characterizing the information provided by these tools was created. These dimensions are explained in the following points.

Source Code Metrics This aspect encompasses several metrics related with the source code of a program [159], in order to assess its quality. Examples of these metrics englobe the number of lines of code, the age of the code, the cohesion and coupling of classes, Halstead Complexity and McCabe Cyclomatic Complexity.

System Components Dependencies Is related with the structural dependencies at the system's architecture level, i.e., the relations between files, modules and packages.

Code Components Dependencies Is a topic similar to the one defined above, but this time the dependencies are between source code components such functions, attributes, classes, and so on.

Source Code Exploration This aspect englobes the possibility of exploring the code by simple visualization of the source code, or by performing information retrieval or other related techniques for information extraction. Notice that this is better seen as a feature than as an information provided, nevertheless is important to take it into account.

Data Structures Exploration This topic embodies the possibility of exploring data structures present in the source code, regarding their abstract representation.

Static Visualization Is concerned with the presentation of static information retrieved from the source code, via static information extraction. It englobes the presentation of Flow Diagrams, Flow Control, Function Call Graphs and other graphs representation.

Dynamic Visualization Concerns with the presentation of dynamic information extracted from the source code. It englobes the presentation of animation of

execution trees, execution traces, analysis based on the that information, and so forth.

Identifiers/Symbols Table This topic is concerned with presentation of the IT, and dynamic evolution of the values on it.

Operational/Behavioral Relations This aspect is related with the presentation of relations between the program and the problem level information.

Table 3.1 presents a summary of how the tools presented in Section 3.4.1 cope with the information dimensions described above. The columns correspond to the tools, and in each row is marked, with a cross, whether they provide of the several information aspects.

Table 3.1: Information Provided by the PCTools Examined

	Alborz	Alma	Bauhaus	Cerberus	CodeCrawler	DA4Java	Imagix4D	JRiSS	PICS	RMtool	Rigi	SeeSoft	SHriMP
Source Code Metrics			X	X	X		X				X	X	
System Components Dependencies	X		X				X		X	X	X		X
Code Components Dependencies			X		X	X	X						
Source Code Exploration	X	X	X	X		X		X	X		X		X
Data Structures Exploration		X					X		X				
Static Visualization		X	X		X				X		X		X
Dinamic Visualization	X	X							X				
Identifiers/Symbols Table		X					X		X				
Operational/Behavioral Relations			X						X				

At a first glance, Table 3.1 presents a very sparse matrix. This means that the information topics are not covered by the major part of the tools. In fact, the tools present many features and, surely, provide much more information than the indicated in the table. It must be left clear that much of the tools were not tested in order to confirm such information. The table information is, mainly, constructed upon the literature reviewed.

In [80], the author assumes the “*missing-link*” problem as a missing detail in almost every PCTools. This problem consists in the non existence of a *cause-effect* relation between the source code and the visualizations produced. Nonetheless, this problem is not as present as the problem of the absence of connections between the aspects visualized at the program domain, and what is the effect, at the problem domain, produced when executing the program under analysis.

Notice therefore, that almost all the tools referred above, address only aspects related with the program domain, and take little advantage of the problem domain,

left remaining the gap already addressed between the domains. In part, it is due to the fact of dealing with `GPLs`; from the discussion in Chapter 2, it is known that the characteristics of `GPLs` difficult the creation of approaches to abridge these domains. However, from the literature reviewed, there are no general and known `PCTools` that take advantage of `DSLs`' characteristics to fill such a gap.

3.5 Domain-Oriented Program Comprehension

This section presents the state of the art on tools and other works that go further and try to analyze the connection of both problem and program domains.

In this dissertation, the main objective is to improve program comprehension tools to cope with `DSLs`, bridging the problem and program domains. In the present chapter, the state of the art on general `PCTools` already was presented, but any of the tools was concerned with `DSLs`. Other works, considered close to the one inherent to this master thesis can be classified as follow: *(i)* approaches to bring, to `PC`, the problem domain knowledge and connect it with the program domain and *(ii)* tools for `DSLs` that, somehow, help on understanding the programs written using these languages.

So, this section is twofold: First, information about approaches that have been developed to conceptually connect the problem and program domains, providing exploration, navigation and visualization of such relations, is given; and second, some tools and approaches based on `DSLs`, which produce analytical results and visualizations to help the understanding of programs written using these languages, are presented.

3.5.1 Approaches for Connecting Domains in DSL Comprehension

The tasks of finding and creating mappings between problem and program domains, are very complex. Approaches that would reduce this problem are based on gathering information about the programs and confront it with higher-level models, describing the application domain, in order to extract behavioral aspects of the program.

However, in [35], the author goes around this problem by claiming that safety-critical systems can be better understood if they are modeled and programmed using well formalized `DSLs`. The premiss says that, to establish such a link between these domains, it is necessary to find connections between the concepts at the problem level, and the concepts at the programming level. His proposal is the definition of `DSLs` through a systematic and formal design criteria, that englobes the transformation into a standard language — Wide Spectrum Language [34] — which has standard programming constructs, and provides easier reverse-engineering. Although this approach is not a novelty in these days, it enhances the fact that using `DSLs`, it is easier to have program comprehension even without tool support.

The *Domain Analysis and Reverse Engineering Project (DAREp)* [39] is an attempt to enlarge the program comprehension process with the knowledge on the problem domain, trying to fill in the gap between the domains. This approach uses

Synchronized Refinement [171], which is a technique for reverse engineering detecting design decisions from the source code, and mapping it to problem domain concepts. The objective of this approach is that, from the source code and the problem domain description, it produces several outputs, including descriptions of the domain, of the program and of how the code is mapped into the application domain. An intermediate architecture of descriptions, mapping the source code information into high-level models, is the final result, claimed to be useful to understand programs.

A similar approach is presented by *Rugaber*. In [173] he claims that existing tools answer *what* questions but not *why* questions. That is, the tools help the user to understand *what* the programs do, but not *why* the things are doing in that way. Using more familiar words, it means that tools only address the program domain (from where is possible to extract the operational view of the program), and pay little attention to the problem domain and the connections between these domains (from where it is possible to retrieve the information about the behavior of the program). In order to minimize this problem, *Rugaber* proposes a *dowser*, which is a tool framework for domain-oriented program understanding, by exploring the structure of the application domain underneath the software program.

This approach also relies on software documentation and its source code, concentrating the data (conceptual and logical models of the program) into a repository, after its extraction and analysis. This data is converted into information being presented under graphs, images or text, allowing the navigation and exploration of the information.

Some studies done on the area of concept analysis and recovery [26, 121] can also be regarded as approaches for domain-oriented program comprehension. From a considerable list of research projects, the one discussed in [60] gains relevance. There, the authors present a technique that creates mappings between the system's visible behavior and the parts of the source code directly related with that behavior. The approach relies on static and dynamic analysis to retrieve data implicitly connected to the program's features (the high-level concepts), from the source code. Then, resorting to concept analysis, it creates relationships between the program level and the high-level concepts set (problem domain), which can be visualized, explored and navigated as graphs. The *Bauhaus* tool, with help of the *Rigi* system, both presented before, was used to realize this idea. The authors claim that this approach is handy to guide a goal-oriented program understanding.

Anam et al. [4, 5] propose an approach that differs from the others presented before. The authors describe a system that provides domain-oriented explanations of the program's behavior. This system is an improvement of a previous one. That previous system only produced textual explanations of the code. The improvement generated a new version that, based on the textual sentences produced, translates the explanations into visual elements of the application domain. This system only supports *PASCAL* programs, and since *PASCAL* is a *GPL*, the authors concluded that it was impossible to cope with all the domains to where the language can be applied, thus, they designed domain models for a few of them. Nevertheless, they produce domain-oriented imagery for each program statement, depicting the state of the world. So, they are aware of the necessities of initializing the state of the world before each exercise. The description of their approach is not very precise in the

literature. But it seems to be an interesting and very related work to the one underlying this thesis.

A more recent work addressing the problem of domains interconnection for improvement of PC strategies, is presented by *Berón et al.* [19, 21]. They developed a strategy called *Behavioral-Operational Relation Strategy* (**BORS**). The authors defines **BORS** as a “strategy aimed at relating the behavioral and operational views based on problem domain object observation and system abstract data types inspection” [22].

In fact, this strategy is based on the idea that the output of a program is composed of domain objects (also called concepts). The authors assume that each one of these objects can be realized by an *Abstract Data Type* (**ADT**) in the source code. Then, taking advantage of static and dynamic analysis strategies applied on the program’s source code, information concerning these **ADTs** is retrieved. A particularity of **BORS** is that it is performed in two phases: at run-time (*Alive*) and after it (*Post Mortem*). The *Alive* phase enables the creation of, among other artifacts, a so called *FE-Tree*, which is a tree of functions that are actually executed during the run-time of the program. Whereas the *Post Mortem* phase allows the conclusion of the process in two steps: (i) the storage of functions related with **ADTs** to be explained and (ii) the explanation of these functions [23].

In the end, the user, is able to see the relations between the concepts of the domain (expressed as **ADTs**) and the source code. **BORS** is implemented in **PICS** system, introduced in the previous section. Attention should be paid that this strategy relies on code annotation, so it depends on the underlying language.

The same authors present another strategy, called *Simultaneous Visualization Strategy* (**SVS**) for synchronization of program and problem domains [19]. This is a strategy to recover the relations between both program and problem domain, using code instrumentation and a parallel execution of the program’s code and a monitor. The monitor informs which are the objects of the program that produce the output of the program.

3.5.2 Tools for DSLPs

To cope with the most known and used **GPLs**, debuggers and similar tools, which are important and required for the development of more complex systems, were constructed. These tools are not regarded as **PCTools**, nevertheless, they can be used to understand the dynamics of programs. The research on **DSLs** identified the need for construction of similar tools to help the development and consequent understanding of programs using such **DSLs**. In the remainder of this section, are presented works in the area of visualizers and debuggers for **DSLPs**, since there were not found any specific **PCTools** for **DSLs** in the literature.

Debuggers

Debuggers, are essential for development practices. Usually they provide information about the program as if it was really executing. Variables’ values, function calls and its arguments, execution traces, and so on, are some of the aspects that can be retrieved from that fake execution of the program, called debugging. Features provided by debuggers are usually the possibility of setting breakpoints (to pause the

execution), stepping forward, stepping over a statement, and so on. At least, these are some of the features provided by debuggers for GPLs like GDB [182], DDD [212] or those integrated in Eclipse¹¹ or Visual Studio .NET¹².

This section addresses three different approaches for debugging DSLs. These debuggers have, somehow, similarities with the work underlying this master dissertation, because they focus on some aspects of the animation and visualization of programs, rather than just debugging it. Other debuggers for DSLs exist, as is the case of TIDE [193], but are not discussed here.

Gem-Mex, introduced by *Anlauff et al.* [3], in 1998, is above all things, a tool for DSLs development. This *Gem-Mex* tool supports a language description formalism — Montages — for the rapid development of DSLs. From the specification of the language, the tool produces, among many other features, a graphical environment where are included visualizers for program animation and debuggers for the language created. These features allow the visualization of the program's behavior, by means of a static and dynamic animation. The static animation is provided by means of tables and textual descriptions, showing the values of variables and functions. On the other hand, the dynamic animation is brought by means of arrows that point, in the source code, the path that is being executed. Unfortunately, the development strategy of this approach is not available in the literature.

Another approach for a debugger is presented by *Mészáros and Levendovsky* [127]. Actually, it is not a debugger for any kind of DSLs, but for model transformations based on graph rewriting [59], and supported by their *Visual Modeling Transformation System* (VMTS). The authors present the possibility of expressing models in a meta-model, which is then plugged with a description of its dynamic behavior. This description is done by means of visual languages: one used to define an event handler model, and another to define a state machine model. The two models that describe the behavior of the main model, are connected, and the workflow behind this approach is that the state machine transitions trigger and fire events, when there are such events associated to ports, which are the communication window between the components of the main model.

The aim of the debugger presented is to create a trace of the transformations that occurred in the model, and to enable modifications in the model, at run-time. The authors are not concerned with the comprehension of programs (models) but only with the visualization of the transformation process.

Finally, *Wu* [205] describes an interesting system for debugging DSLs, based on grammar technology. The tool presented is a plugin for Eclipse and takes great advantage of the debugging facilities of this *Integrated Development Environment* (IDE). The framework presented by the author (see Figure 3.3), so called *DSL Debugging Framework* (DDF), is twofold. First, the user provides the grammar of the DSL, whether it is imperative, declarative or hybrid, with semantic actions instructing the processor to (i) translate the language into a GPL, Java in this case; and (ii) create mappings between the source code (the DSL) and the translated code (the GPL). The latter is, general words, a mapping between lines of the fragment in the source code and the correspondent in the translated one. For more details on this

¹¹Home Page: <http://www.eclipse.org>

¹²Home Page: <http://msdn.microsoft.com/en-us/vstudio/default.aspx>

mapping see [206].

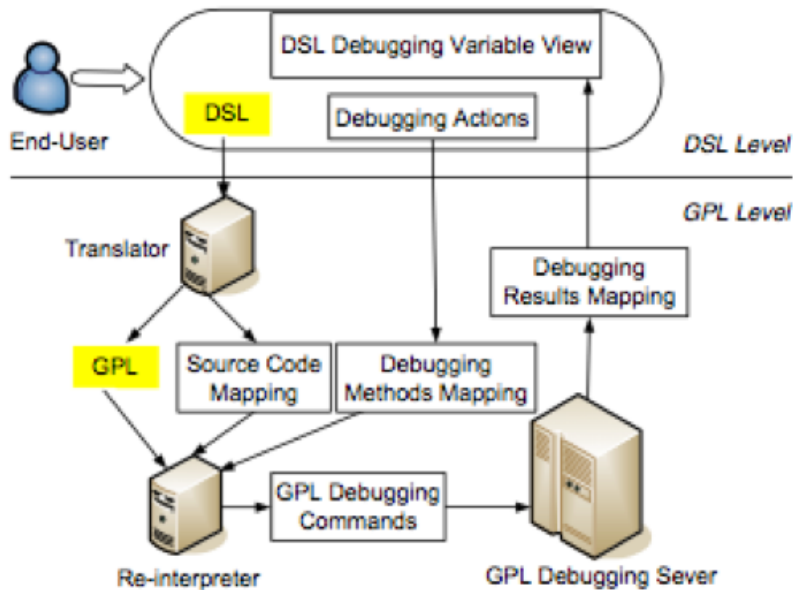


Figure 3.3: Framework for Debugging DSLs

Second the user debugs the program, at DSL level, reusing the Eclipse debugging actions. However, the program which is actually being debugged, is not the original one, but the GPLP one. So, the system relies on the mappings database created to, in a first moment, situate the debugger at GPL level to really extract the information of the program, and in a second moment, to go back and present the result of the debugging action at DSL level.

The main interest of this work is not focused on the debugging actions that are provided for the DSL, because it reuses an existing debugger. The interest, at least for this dissertation, relies on the mappings and translation of the DSLP into the GPLP,

Visualizers

Visualizers are, in the context of this dissertation, tools that help the understanding of a program, by illustrating specific details of that program. Thus, they can be seen as SV systems. In this section, two visualization systems that take advantage of the knowledge on specific domains, are presented, and their approaches are discussed.

Mathematics is one specific area of knowledge where the human brain has more difficulties to grasp. The abstract calculus and functions are not easy to understand by every person. But perhaps with graphical representations, some of these difficulties start to vanish. This was what Peterson [154] thought when developed a system for mathematical aspects visualization. His system is based on a DSL that was firstly embedded in Haskell: Pan; but the author tweaked it to behave as a stand alone language and to improve its syntax, making it more usable. The users

of the system specify, using this language, the program which is based on definitions close to the mathematical notation. They do this in an environment that allows also the introduction of controls to manipulate the final visualization. The final visualization, that is, the effects of the program expressions, are visible upon imported images.

As can be noticed, the visualization is not about the functions, but about the effect of the functions in an image, what is important to understand the function utility. Moreover, the visualizations can be adjusted to cope with the user needs.

That is an interesting work which tries to take advantage of the problem domain to ease the understandability of a mathematical program. However, it does not link the effects of the functions with the functions themselves. In fact this is what happens with the `Dot` language, and the output produced by `Graphviz`, or with the `Java GUI` language and the graphical output produced¹³.

Kolpakov [104], presents `BioUML`¹⁴, a framework for modeling and simulating of biological systems. That framework permits the creation, using graphs, of biological systems that are visualized and edited by means of diagrams. Additionally, it automatically generates models that can be simulated resorting to `MATLAB`¹⁵ capabilities.

This system is another example that using formal specifications (the `MATLAB M-files`) and the knowledge in a specific domain (biologic systems) is possible to visualize, at problem level, the effects produced by that specifications in the biological system. But once again, these simulations only present the results of the specifications, what is not so interesting to understand them. It still missing a link between the execution of the *program* and the effects casted on the problem domain.

3.5.3 Discussion

From all the approaches and tools referred in Sections 3.5.1 and 3.5.2, which are, somehow, representative works on the area of the domain oriented program comprehension, it is possible to draw three main conclusions:

1. These tools and approaches do not fill the gap illustrated in Figure 3.2, between the program and problem domains;
2. Tools are not conceived to provide visualizations of both domains and
3. Some of the tools are only focused on a specific DSL, not coping with all DSLs or, at least, a large number of them and

In fact, there are some approaches trying to fill the gap between the domains, by interconnecting them. The `BORS` strategy (present in the `PICS` tool) is an example of success. But the visualization of the problem domain may be not as higher level as the user would need. However, besides `PICS`, there are no more tools providing this interconnection of domains. So, the gap between them still remaining.

¹³In this case, there are already studies helping on the reverse engineering of the interface to intermediate behavioral models that allow the detection of faults [176].

¹⁴Home Page: <http://www.biouml.net>

¹⁵Home Page: <http://www.mathworks.com>

The tools addressed are not conceived to perform visualization of both domains. In Section 3.5.2, the majority of the presented tools only provide visualization of the problem domain, forgetting about the program domain. This automatically leads to systems not biased for program comprehension.

The solution presented by Wu [205], is a contradiction of the third conclusion. On the one hand, it deals with *all* DSLs, but on the other hand, it just concerns with the program domain, since it is a common debugger, reinforcing the second conclusion.

At the end, the main topic to retain is that tools and approaches for domain-oriented program comprehension are rare; and ,when exist, are not systematic and do not provide features to bridge the gap between program and problem domains.

3.6 Summary

While there is a large work on program comprehension at theoretical, practical levels and at tools implementation level, they do not always meet all the needs of the users, who have to comprehend the programs. The development of tools used in program comprehension, insist on the visualization of software by means of graphs (mainly); and the creation of cognitive models that are, for some times, neglected on the construction of such PCTools.

General PCTools, i.e., those dealing with the most used programming languages (mainly GPLs), gather information from the source code (and a few times, from higher-level resources) and present them over nine identified dimensions: (i) Code Metrics; (ii) System Components Dependencies; (iii) Code Components Dependencies; (iv) Source Code Exploration; (v) Data Structures Exploration; (vi) Static Visualization; (vii) Dynamic Visualization; (viii) Identifiers/Symbols Table and (ix) Operational/Behavioral Relations. The ninth dimension was the one with less relevance in the information provided by a set of tools that were, *roughly*, studied. However this dimension is concerned with the information that should be provided by all the tools to fill in the gap between the program and the problem domains.

In addition, more specific projects and tools also show that exists a big deficit on the domain-oriented program comprehension in what respects the bridge construction between the problem and program domains, and the usage of DSLs to take advantage of the domain knowledge to improve the program comprehension.

In this chapter, an overview of themes and issues on the program comprehension area were introduced. Fundamental definitions were given to establish a more concrete dialogue between the dissertation text and the reader. Software visualization and cognitive model terms were presented with more profundity, because are central aspects not only in the program comprehension, but also in the rest of this dissertation. Then, the state of the art on general PCToolswas studied and the information they provide was analyzed with criticism. Some projects that go further in the domain-oriented program comprehension,were surveyed as related work. The main points were addressed and discussed.

Chapter 4

An Idea to Comprehend DSLs

Give me six hours to chop down a tree and I will spend the first four sharpening the axe.

Abraham Lincoln

Recalling Section 1.3, the main goal proposed for this thesis is to prove the correctness of the extension made to Brook's theory in Theorem 1, which says, briefly, that program comprehension tasks will be easy if tools provide synchronized visualizations of the program and problem domains. To complete this proof, two *minor* answers need to be taken into account, as was depicted in Figure 1.1.

In this chapter, attentions are concentrated, exclusively, on Question 1, which asks *Can the PC techniques, tailored for GPLs, be applied on DSLs?* An answer should be given to that question; to achieve a concrete and well supported answer, a methodology is followed. Figure 4.1 presents the sketch of the methodology.

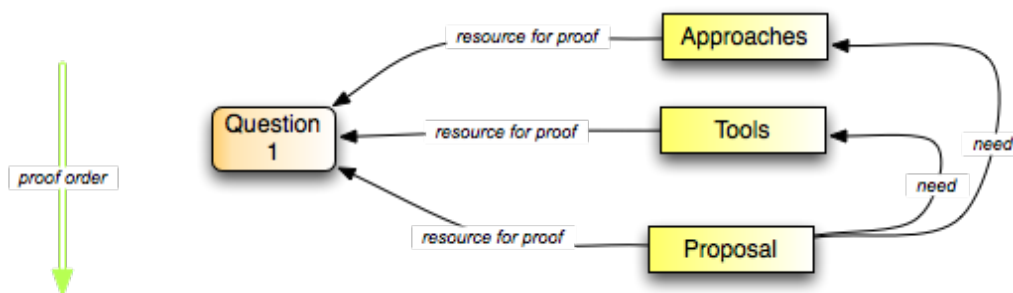


Figure 4.1: Answering Question 1: *Can the PC techniques, tailored for GPLs, be applied on DSLs?*

Figure 4.1 shows that for answering Question 1, three resources must be available: approaches, tools and one proposal. *Approaches* and *tools* refer to the approaches and tools already addressed and analyzed in the previous chapter. The *proposal* is a resource derived from the approaches existent and intends to be the conception of a new approach to understanding DSLPs. Question 1 is automatically answered, when, *and if*, such resource is produced.

In chapter 3, tools and approaches used to extract information and present it, were analyzed. From them, one is chosen to be the base of the *proposal*. This choice is made and justified in Section 4.1. Section 4.2 gives a detailed description of the approach that have been picked up and relates it with the tool which embodies this technique. Finally, the *proposal* is conceived and *abstractly* explained in Section 4.3.

4.1 Choosing One Suitable Approach

The objective of this section is to recall some approaches for program comprehension, that have been discussed before, and to discuss others that have not been presented yet. Finally, one of these approaches is selected to serve as basis for the *proposal* that intends to improve PCTools to cope with DSLs, taking profit of the well defined characteristics of these languages.

The study made in Section 3.5.1 revealed the work done on the creation of approaches for program comprehension that address the application domain of the programs written in GPLs. SVS and BORS [19] techniques could be a good start for an approach that would cope with DSLs, because they address both program and problem domains. However, the associated tool — PICS — only copes with the C language. This is an handicap, because the core of the tool is oriented to a specific language, and the major part of the DSLs are not even closer to that imperative language. Moreover, BORS runs in two separated phases, not being possible a synchronized visualization of the domains; and SVS, despite providing synchronized relations between the domains, heavily depends on C code instrumentation.

From this first attempt of *evaluating* one existing approach, a characteristic should be regarded: the tools that embody the program comprehension technique must not be constrained to a specific programming language, i.e., they must be generic tools¹. This characteristic creates a great filter when applied to the set of tools presented in section 3.4. The major part of these tools are limited to cope with a small number of languages. The theoretical study made, shows that only Alma and CodeCrawler systems are not language-dependent. Thus, the approaches they embody are not dependent on the programming language, as well.

Both CodeCrawler and Alma are tools that depend on other tools. The architecture of CodeCrawler [109] is explicitly built upon the Moose system [57]. In its turn, Alma depends on LISA system [125]. This aspect is also a good characteristic to work as a filter to find a suitable approach that can be applied to DSLs: the more a tool is independent, the easier the approach is to be improved. However, in this case, this characteristic is not decisive to choose one of these two tools, since both depend on *third party* tools. For a more precise decision, next are summarized the approaches followed by each tool.

CodeCrawler uses Moose and associated features as the meta-model where the information of the source code is stored. The meta-model is processed by the core of the system, which serves as bridge to the visualization of the source code of the program under analysis. The system relies on the notion of *graph* to internally

¹The consideration of tools, rather than only the approaches, is due to the fact that a tool is the concrete form of an approach or technique, and is where important characteristics can be easily pointed out.

represent the knowledge that comes from the meta-model, and resorts to traditional graph operations to expound the internals. Summing up the approach underneath `CodeCrawler`, the PR feeds the system with the source code; then `Moose` creates the meta-model, which is then transposed to the internal graph representation of `CodeCrawler` and processed for visualization.

`Alma` uses an abstract representation of the source program, in the form of a tree to store the information at the program level. In a first step (done once per language and not per program) it is used `LISA` (but other compiler-compiler tool could be used) and `AGs` for construction of the generic and abstract representation of the language constructs, and to associate important information to the nodes of the created tree. This tree is called *Decorated Abstract Syntax Tree* (DAST), and is generated every time a program is passed to `Alma`. Then the core of `Alma` traverses the DAST, and resorting to a database of rewriting and visualization rules, the internal tree is modified and the program aspects are visualized.

The approaches used in both the tools are similar. One uses a graph, the other uses a tree for internal representation of the program information. Both of them have a core that processes the intermediate representation of the program to output its visualization aspects. As they are similar approaches, there is not a consensual point for the choice. But it is known that `CodeCrawler` is more adjusted to cope with object-oriented languages, so the DAST approach of `Alma` seems to be a good starting point for the new approach that deals with DSLs and the problem domain of the programs (or generally, the problem domain of the DSL). Also, the acquaintance level to the approaches and the accessibility to the source code of the tool, leads to the selection of the DAST approach.

4.2 DAST — The Initial Approach

An important aspect of an approach for program comprehension is the way the information of the source program is retrieved and stored in an intermediate structure for a later visualization and exploration. This intermediate structure should provide easy and fast access to the information. As `Alma` aims at showing the *soul* of a program (its meaning) resorting to the same core processor, whichever the programming language is, it needs to extract the abstract representation of that program and to store it in a generic intermediate structure. Since it relies on `AG` techniques to retrieve the necessary information of the source program, it is obvious the reutilization of the structures that parsers build during their action: the *Abstract Syntax Tree* (AST) [1]. But an AST does not represent a program in a generic form. Otherwise, it creates an abstract representation of the program for the language processed. Nonetheless, the AST concept can be reused to achieve an abstract representation of any program, written in any programming language. In `Alma` it was achieved by resorting to the DAST concept. Figure 4.2 shows the main differences between the AST and the DAST and how the second can be constructed from the first.

A DAST stores the meaning of the program [152], resorting to patterns. These patterns represent language constructs, which are concepts present in every programming language. They can be seen as semantic patterns. Regarding Figure 4.2, the AST (the leftmost tree) represents the program, resorting to the essential syntac-

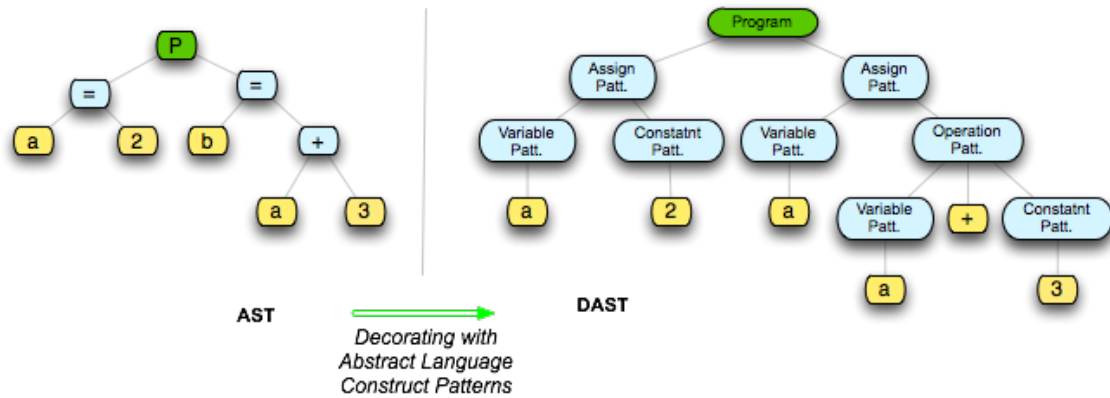


Figure 4.2: DAST - Decorated Abstract Syntax Tree

tic constructs of the language. On the other hand, the DAST (the rightmost tree) also represents the program, but uses the generic semantics of the syntactic constructs, to explain the meaning of the program. For example: $a = 2$ means that the variable a will be assigned the constant 2.

Alma supports patterns for declaration of variables, assignment and conditional expressions, read from input, write to output, arithmetic and logic operations, access to variables, definition of constants, loops, generic statements, declaration of lists, function definition and invocation and finally return expressions [48]. The construction of a DAST with these patterns, as nodes, is briefly addressed in Chapter 5.

Using a DAST as an abstract representation of the source program, brings many advantages to later phases of the visualization. The most important one is that a single core for processing is used to create the same visualization for the most varied programs. The main flow of this approach using the DAST concept starts when the program is passed to Alma, and parsed by the parser built for the underlying programming language. This parser, also called *Front-End* (FE), constructs the DAST of the program, and keep it as an intermediate representation. Then, when the PR starts the interpretation of the program, the processor (called *Back-End* (BE)) traverses the tree, following a top-down strategy, and resorts to databases of pre-defined rules to rewrite the patterns modifying their attributes, and to create the visualization at the program level. Figure 4.3 depicts the expounded process, showing the DAST traversal, the access to the database of rules, and generation of the visualization.

The rules are associated with each one of the patterns, so that, when the *tree walker* traverses the DAST and find non processed leaf nodes, these rules are applied according to the information of the pattern underlying the DAST node. There are two databases of rules: (i) rewriting rules and (ii) visualization rules [79]. The rewriting rules (RR in Figure 4.3), are used to rewrite the nodes of the DAST, performing a semantic modification on it. These rules also actuate at the internal state of the program, by acting as interfaces for storage, modification, and access of the value associated with the variables in the programs. On the other hand, the visualization rules (VR in Figure 4.3) do not actuate over the DAST. Instead, when the pattern in the DAST is being processed, the visualization rule associated with such pattern,

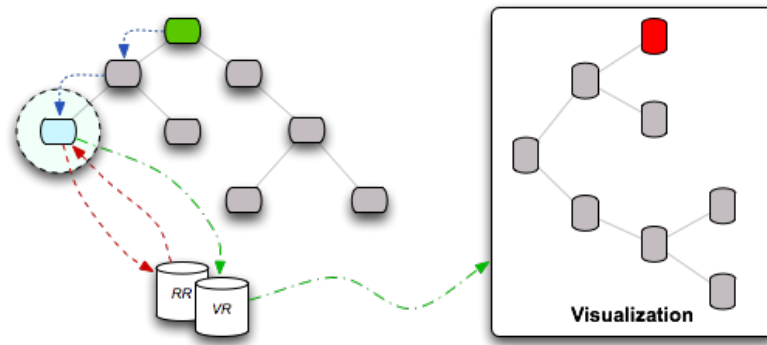


Figure 4.3: Processing a DAST

takes the necessary information from it, and builds or rebuilds the tree correspondent to the dynamic interpretation of the program under analysis, which is one of the views of the program domain provided by Alma.

This approach does not modify the source code of the program, and do not use compilation nor optimization of code for analysis. Then, variables and other identifiers of the program are not stored in memory when the programming is running. Actually, the program is never *running* in Alma, because this system can be seen as a virtual machine that interprets the program, relying on the powerful intermediate structure named DAST. Mistakenly, Alma is being pointed to be a visual debugger. Although it can be seen as one, their authors defend that it is more than a simple debugger [49]. After all, reusing the underlying approach, is possible to enhance the visualization of this approach to cope with the problem domain of DSLs; what is something that a debugger would never be capable to offer. The following section presents that new approach completely derived from the DAST.

4.3 $D_{ap}AST$ — The Improved Approach

In Chapter 2, several characteristics of DSLs were highlighted from the simple fact that these languages are tailored for a specific domain of application. Experts on the application domain of such languages are able to use them even without having programming skills. This happens because the language is so close to the problem domain, that the experts do not need to think about low-level details; instead, they conceive a mental image of what is wanted in the *real world* and easily use the language constructs to concretize it. These points lead to the following conclusions:

- The DSLs' syntactic constructs are very close to the real concepts of the application domain (thanks to the abstractness and expressiveness of these languages), and
- The interconnections between concepts of the application domain, along with their concrete shape, are useful for a mental representation of the problem and its solution.

Regarding the problem from the other side, that is, the side of those who have a DSLP already written but need to understand it, the construction of an image of what and how the program does, would be helpful for its comprehension. Taking into account both points drawn above, it is possible to create such a representative image, by interconnecting the syntactic and semantic constructs of the language, the graphical shape of the concepts in the problem domain, and animating them using dynamic information from the program. The last sentence refers to the starting point of the approach that is introduced along this section and is baptized with the name of *Decorated with animation patterns Abstract Syntax Tree* ($D_{ap}AST$). In some aspects this approach follows a cinema metaphor, which is used to explain some notions that are harder to grasp.

4.3.1 DAST versus $D_{ap}AST$

As said in Section 4.2, the DAST is an abstract representation of the meaning of a program. It is built recurring to generalized semantic patterns, which represent the nodes of the tree. In its turn, a $D_{ap}AST$ is, likewise, an abstract representation of the meaning of a program (also referred as program's semantics). However, there is more inside each node of such a tree.

In a $D_{ap}AST$ the semantic patterns, that decorated the AST to generate a DAST, are incremented with extra information attributes, and are called henceforth *Animation Nodes*. This extra information consists of images to represent the concepts in the problem domain; and animation functions to create the animation of the concepts by relating the images with each other and with the dynamic information extracted from the program interpretation.

The information added to the semantic patterns, constituting the animation nodes, is called *Animation Patterns*. Figure 4.4 shows the $D_{ap}AST$ resulting from the decoration of the DAST with several animation patterns. The nodes in orange (darker ones) are the animation nodes, and the clouds are the representation of the animation patterns. Notice, from Figure 4.4, that a $D_{ap}AST$ is a hybrid tree, in the extent that it is not mandatory that all the nodes have animation patterns associated.

Notice also that a single animation pattern is composed of one image and one animation function. But the animation nodes are not restricted to have only one animation pattern associated; instead they have a list of animation patterns. The animation patterns in this list can repeat the image and the animation function. The way the animation nodes are configured is responsibility of the person who defines the visualization of the problem domain for the DSL. More details on the definition of the visualization are given in Chapter 5.

4.3.2 Inside Animation Nodes

The animation nodes are the pieces that compose a $D_{ap}AST$, and make it different from the DAST presented in Section 4.2. Each animation node is composed of two parts of information: the basic one, constituting the information associated with any DAST node, and the extra part, constituting the information for the visualization and

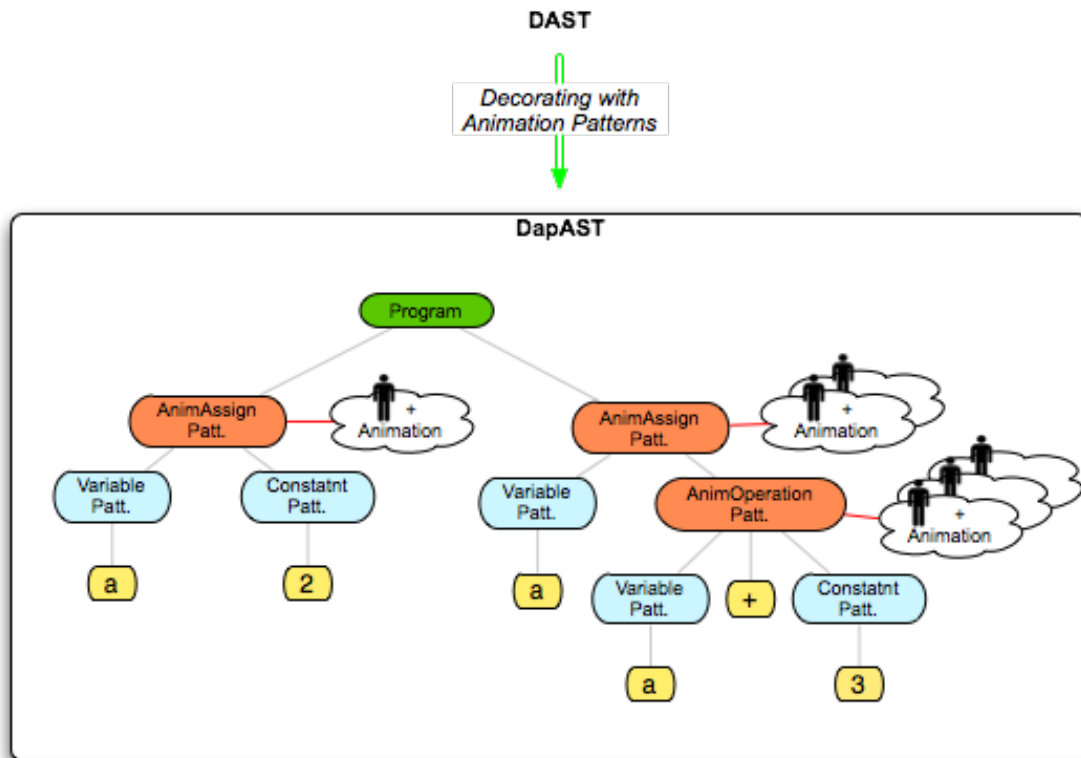


Figure 4.4: $D_{ap}AST$ — Decorated with animation patterns Abstract Syntax Tree

animation of the problem domain. The explanation of the first part can be seen in the literature related with Alma [152, 79, 48, 49].

The second part is the list of animation patterns. Each animation pattern is also composed of two parts: the image and the animation function. Until now these have been the names given to these parts, however to keep close this approach to the cinema metaphor, the image is henceforth called *Actor*, while the animation function remains with the same name.

A real actor is someone or something that interprets the scripts produced by an author, resorting to actions of the body and sounds; and has, as objective, to tell and to explain the idea or moral of the story underlying the script. In this program comprehension approach, an *Actor* represents one concept that is part of the problem domain and is implied in a certain semantic action of the language. Also it can be seen as one of the objects controlled by the DSL. This actor also follows the script of an author (the DSLP) and has the same objective of telling the idea of such DSLP to the person who reads the program. In order to express the moments of its representation, the actor has implicit a set of key figures, named *Poses*. They are addressed by the animation functions, to visually represent the effects of the program in the application domain.

The animation functions can be seen as script lines of the story being played. They transmit to the actors the actions they should perform (e.g. move or rotate) and which are the used poses in a determined semantic situation. A single animation function may order an actor to use a sequence of poses for a more detailed animation

of the problem domain.

These details inside an animation node define some simple concepts with characteristic notation. Knowing the parts played by each one and how they are related with each other, it is important to understand the process of animation being conceived in this chapter. Section 4.3.3 presents an explanation of the $D_{ap}AST$ traversing, which is the essential point of this chapter, because it is where the abstract idea to comprehend DSLs comes close to concretization.

4.3.3 Traversing the $D_{ap}AST$ — Theoretical Approach

The animation of the actors according to the associated animation functions, follow up a given order. The top-down tree traversal used to visualize the program domain from the $DAST$ is applied on the $D_{ap}AST$ to visualize the problem domain. The walker of the $DAST$ is kept untouched, so the program domain visualization is not changed, however extensions are made, enabling the walker to cope with the new animation nodes and the associated animation patterns.

The tree traversal and the execution of the problem domain animation is depicted in Figure 4.5. The scheme followed is equal to that shown in Figure 4.3. The tree

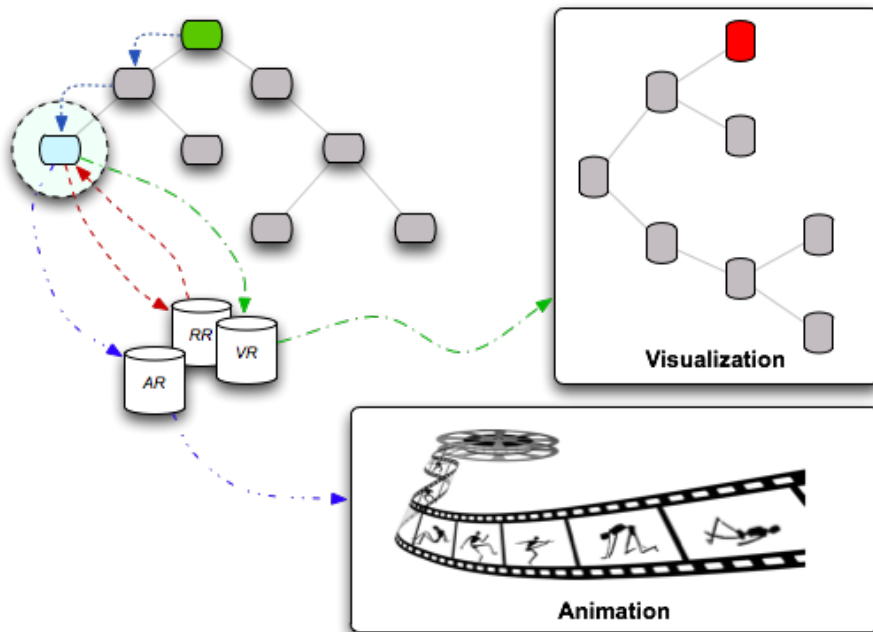


Figure 4.5: Processing the $D_{ap}AST$

walker processes the nodes in a top-down strategy; for each node it accesses the database of rules to rewrite the tree and to build the visualization of the program domain. But in this new scheme, the tree walker is incremented with a new action when processing a node. For each node the walker checks whether it is an animation node or a semantic pattern. In case of being an animation node the walker accesses a new database of rules — *Animatoin Rules* — created on purpose for the animation of the problem domain.

Unlike what happens with the rewriting and visualization rules, the animation rules do not look to the semantic patterns of the nodes, but to the animation patterns that constitute the nodes. Such rules consist of different drawing routines, which are called regarding the animation patterns in the node. Their objective is twofold: first is to apply the animation function to the actors, and second is to draw one or various poses of the actors in the scene of the problem domain animation.

The animation is directed by frames. Theoretically, each frame corresponds to the processing of one animation node; however, in practice, the processing of a node may generate more than one frame, all depends on the number of poses that an actor is *told* to represent by the animation function.

4.4 Summary — Answering Question One

Alma system presents an approach for program comprehension based on the creation of an abstract representation of the program under analysis. This representation is stored in an intermediate structure called DAST, which is similar to an AST but represents the semantics of the program, rather than the syntax. So the nodes of this DAST are generic semantic patterns that are programming language and programming paradigm independent. The technique for visualization of the program comprises a top-down tree traversal and the processing of the nodes resorting to databases of rules.

As the DAST approach supports the major part of the programming language, with small improvements, it would support any DSL and would also be capable of addressing the application domain of these DSLs. So, following a cinema metaphor, was conjectured an extension to that approach. In this novel idea to comprehend DSLPs, the abstract representation of the meaning of the program is also stored in an intermediate structure, called $D_{ap}AST$. This structure differs from the initial, because the tree nodes are now decorated with animation patterns. These animation patterns are used to relate a visual animation of the problem domain with the program's semantic action in a node of the tree. This way, the animation pattern is composed of actors and animation functions to represent concepts of the static problem domain associated with the DSL. The traversal to this structure is similar to the initial one; it only requires the use of a new database of rules to steer the animation of the actors regarding the animation functions, and to draw the frames in the animation scene.

The main objective of this chapter was to answer Question 1, raised in Section 1.2: “*Can the PC techniques, tailored for GPLs, be applied on DSLs?*”. After the discussion presented on the several sections of the present chapter, it is possible to answer affirmatively to that question. In fact, the $D_{ap}AST$ approach conceived in Section 4.3, is based on a technique for GPLs, which was applied to DSLs and improved to cope with their problem domain. Such affirmative answer raises Theorem 2, summing up the chapter.

Theorem 2. *Program Comprehension techniques for GPLs can be applied on DSLs.*

Chapter 5

Alma²

Cherish your visions and your dreams as they are the children of your soul, the blueprints of your ultimate achievements.

Napoleon Hill

Last chapter (Chapter 4) summed up with an important result: Theorem 2, saying that program comprehension approaches usually crafted to deal with GPLs can be applied, with minor changes, to DSLs and take advantage of the problem domain associated with these languages. Despite of its importance, this theorem has no value if it can not be used to achieve other results. So, in the present chapter, Theorem 2 is used to answer Question 2, raised in Section 1.2: *Is it possible to improve existent PCTools, in order to give the end-user useful and better perspectives to comprehend the program and the problem?* Figure 5.1, which is a mutation part of the steering Figure 1.1, depicts the scheme to follow in order to give an answer this question.

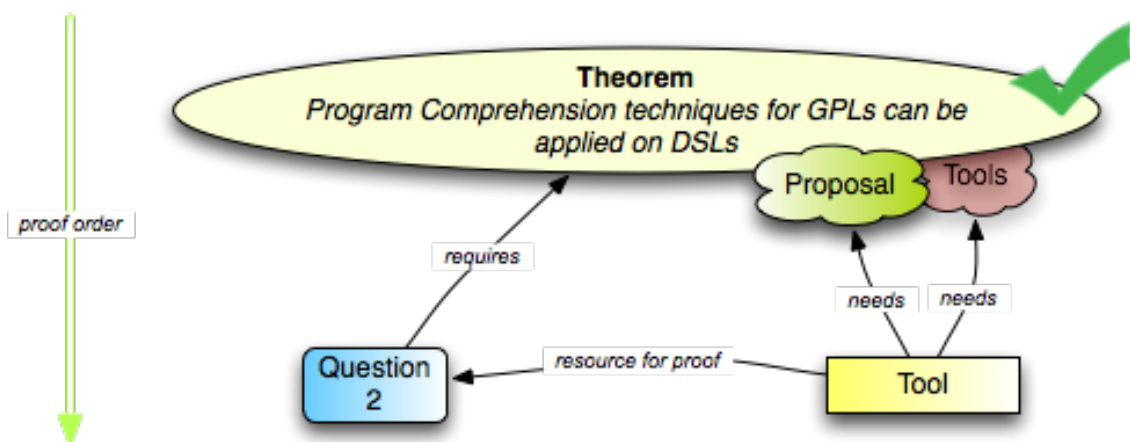


Figure 5.1: Answering Question 2: *Is it possible to improve existent PCTools, in order to give the end-user useful and better perspectives to comprehend the program and the problem?*

Ruled by the result of Theorem 2, along with the approach conceived, $D_{ap}AST$,

and the tool chosen, *Alma*, a proof must be carried out of whether it is or it is not possible to improve program comprehension tools to cope with DSLs and with their problem domain associated. Thus, Question 2's answer is affirmative if a *DsPCTool* can be created based on a *PCTool* — *Alma*, in this case — and on the *D_{ap}AST* approach conceived in Chapter 4.

The question to answer in this chapter has three main aspects that are should be shown. The most important one is, obviously, the improvement of the *PCTool* into the *DsPCTool*; another aspect is the interaction of the different users with the tool; and finally, it is important to describe the new perspectives offered, in order to evaluate their utility regarding the previous version of the tool. So, this chapter is structured over two main sections. In Section 5.1 all the technical aspects related with the development of the tool are expounded. First the overall architecture of the system is shown, then its internal structure, regarding the relations between the different components, is described. A major focus in this section is given to the strategy of traversing the *D_{ap}AST* structure. In Section 5.2, the usage of the system is explained. First, the type of users are identified and characterized; then, for each one is described the main workflow on using the new approach to understand a program. Important focus is given to the construction of a *D_{ap}AST*, complementing the explanation of the approach proposed in Chapter 4.

5.1 Developing the System

Alma is a program comprehension tool with two main focus. One is the understanding of a program, and the other is its teaching purposes by animation of the program flow, and showing internal aspects of the program's state.

An important aspect that must not be forgotten is its *dependency* on *LISA*. The word *dependency* is emphasized because other compiler generator tools could have been used instead; so there is not a strong dependency between *LISA* and *Alma*. However, for explanations purposes, *LISA* will be used, but it should be regarded as just an instance of a compiler generator.

So, in *LISA* is prepared the FE for a given programming language, by using AGs to declare the *DAST* constructs. Once this specification is done, the PR feeds *Alma* with a program in that language, and the processor (the FE generated with *LISA*) parses the program, prepares the internal state of the program and builds its *DAST*. Then, the generic BE of *alma* animates the program domain using the rewriting and visualization rules.

This glimpse on *Alma* helps the reader of this thesis on conceiving a representation of the system's architecture and a way of using it. As *Alma* is the base for the tool described in this section, no more specific details on its architecture and other aspects are given. However, as it can not be neglected, it is kept a parallel between the description of the *DsPCTool* and *Alma* to emphasize the improvements made.

*Alma*² (read: *Alma* two) is the name given to the system originated from the improvement made on *Alma*. The number 2 in superscript style is fully of valid meanings: it means that *Alma*² is a second version of *Alma*; it means that two domains are now addressed (the problem and the program domains); it may mean also the synchronization of these domains; and so many other meanings, that, in

fact, are not important in the discussion of this thesis.

In this section technical aspects concerning the development of Alma^2 are addressed. The architecture, the internal class structure, and some of the algorithms used to traverse the $D_{ap}\text{AST}$ and to draw the actors are the main points.

5.1.1 Architecture

As Alma^2 is based on Alma , the components composing their architectures are almost the same, except for those that implement the extension done. Figure 5.2 presents the architecture of Alma^2 .

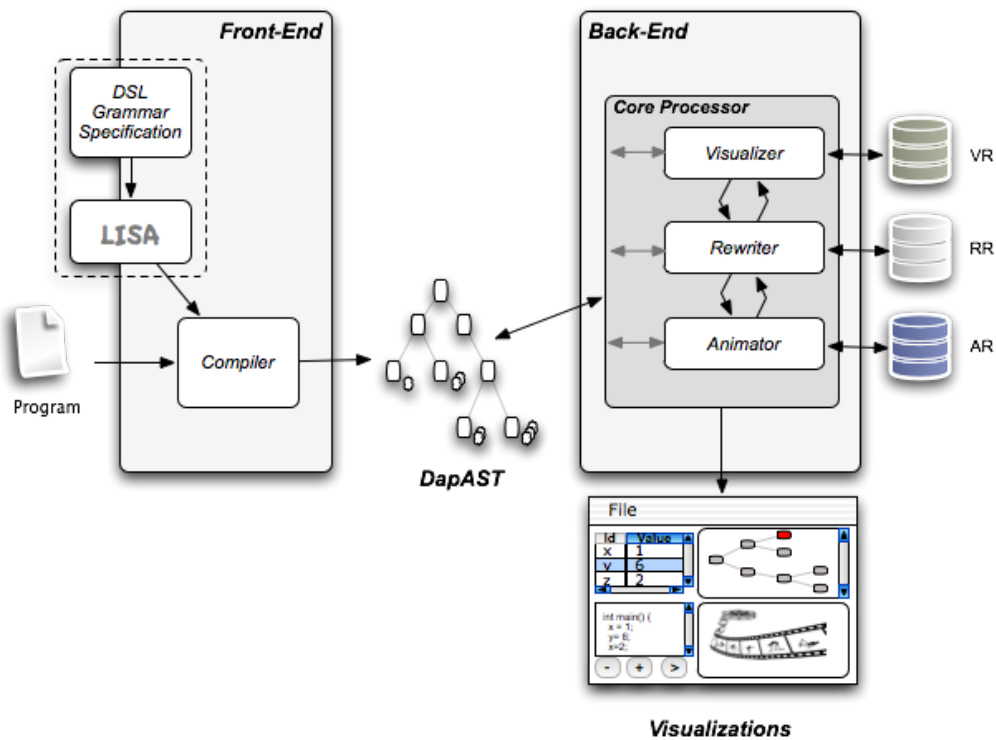


Figure 5.2: Alma^2 Architecture

As can be seen, the system is divided into two main parts: the Front-End and the Back-End. Let's begin by explaining the FE. Its main component is the **Compiler**. It receives the programs and extracts the necessary information to build the $D_{ap}\text{AST}$. The compiler recognizes one single DSL, since it is generated automatically from the grammar specification of such a language, using **LISA**. The **Grammar Specification** and **LISA** are presented inside a dashed box because, in fact, they are not part of the FE; with their inclusion in Figure 5.2 is intended to show what is necessary to create the FE. For each DSL is built a FE, capable of reading the compliant programs to generate the intermediate structure that supports the abstract representation of the program's meaning. Until here, there is nothing different from Alma 's architecture, except, of course, the generation of a $D_{ap}\text{AST}$ instead of a DAST .

The $D_{ap}\text{AST}$ is the communication channel between the FE and the BE, for being the structure where the needed information for the program visualization, animation

and understanding is kept. This component was fully addressed in Section 4.3.

The Back-End is composed of one component called **Core Processor**. From the name of this component, it is easy to understand that it is where the most important actions are executed. This processor *processes* the D_{ap} AST information performing top-down traversals. As was explained before, for each node of the intermediate structure, this processor actuates based on appropriated pre-defined rules. However, it does not do it alone. It delegates responsibilities by exchanging information with other components: the **Visualizer**, the **Rewriter** and the **Animator**. The **Rewriter** component gets the information incoming from the D_{ap} AST node being processed, and transforms it regarding the rewriting rules database (cylinder labeled as RR in Figure 5.2). It performs simple and various calculi to compute the value of a variable, to access a variable's value, and so on. The **Visualizer** also receives information from the D_{ap} AST and from the **Rewriter**. Based on it, it accesses a database of visualization rules (cylinder labeled as VR in Figure 5.2), and creates the visualization of the program domain, by *drawing* a tree of the abstract program's flow. However, other program domain visualization could be constructed, by extending these rules. These two components work equally in **Alma**, of course with the exception that the information comes from a D_{ap} AST. The difference to **Alma**² is in the third component of the **Core Processor**: the **Animator**. This component works similarly to the **Visualizer**. It takes information from the D_{ap} AST and from the **Rewriter**. But instead of being concerned with the semantic information of the D_{ap} AST node under processing, it grabs, essentially, the information related with the Animation Patterns of the node. Then, regarding these patterns, it accesses the animation rules database (cylinder labeled as AR in Figure 5.2) to apply the animation functions to the actors, drawing, then, the actors in the scene.

The **Core Processor**, receives the work done by its components and holds the task of presenting the output visualizations in a *user-friendly* interface. This interface of visualizations corresponds to the main window of **Alma**². Figure 5.2 presents a thumbnail of this window (the box labeled with **Visualizations**). The image is small, but is possible to see four sub-windows representing the four views intended to be necessary to provide important information about the program. The views are listed below with a simple description:

Identifiers Table (left column, first row) Represents the internal state of the program being interpreted. It keeps the variables of the program and the associated values.

Source Code (left column, second row) A text field (not editable) with the source code of the program under analysis. The line being interpreted in each moment is highlighted.

Interpretation Tree (right column, first row) Is the static/dynamic semantic representation of the input program. It is normally called execution tree, but since in **Alma**² the program is never executed, it was decided to call it that way. This tree also represent the program's flow, and uses three colors to indicate that the node is not yet interpreted, the node is under interpretation and the node was interpreted, respectively, grey, red, and green.

Behavior Animation Panel (right column, second row) Is the panel where the scenes with the actors of the problem domain animations are drawn. It represents the effects of the program execution/interpretation in the concepts of the problem domain.

The BE is the part of Alma that suffered more extensions to comply with the improvements. Next section shows the internal structure of the system, regarding the extensions made, and explaining them in practice, in order to complete the information conceived in Chapter 4.

5.1.2 Internal Structure

A simple vision of the internal structure of Alma² is shown in Figure 5.3, by means of a class diagram¹. The internal structure is divided into two separated parts with two distinct utilities. But, before addressing these two parts, is important to inform that everything inside the green box (package) identified with the label **Animation**, embodies all the extensions made to Alma. That is, the original classes of Alma are painted in a darker color and are situated outside the **Animation** package.

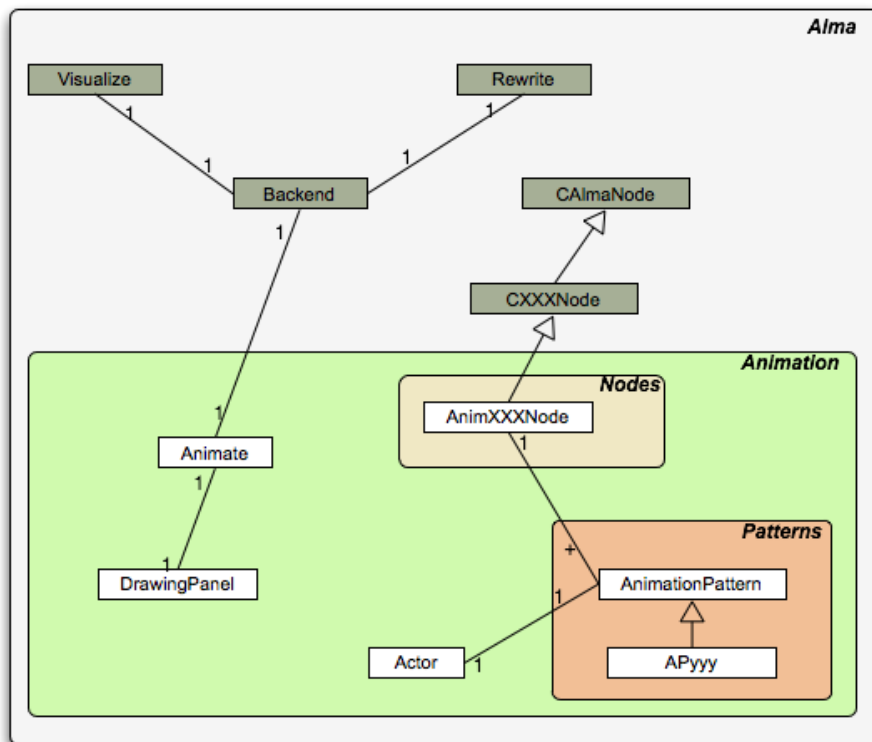


Figure 5.3: Alma² Internal Structure

The first part, comprehending the classes *Backend*, *Visualize*, *Rewrite*, *Animate* and *DrawingPanel*, corresponds to the core processor of the system. The remaining

¹Attention should be paid that much more classes make part of Alma² class diagram; the one presented is kept simple, but with the essential, for an easier understanding.

classes are used to create the D_{ap} AST, as they can be seen as constructs of an embedded language. In this section only the classes of the second part are explained with more detail. The others are addressed in the next section.

The Nodes Classes

In the class diagram presented (Figure 5.3), there are two classes called *CXXXNode* and *AnimXXXNode*. In fact, they do not exist in the system as shown here, instead, they represent a family of classes having the prefixes and suffix right before and after the *XXX*.

The *CXXXNode* classes represent the constructs for the DAST nodes. Each one of the classes stores necessary information about a generic semantic aspect of the program. For instance, the assignment of a variable, is represented by the class *CAssignNode*. But a node of this kind needs information about the variable, and the value to assign. So, the *CAssignNode* class would store that information in attributes, and successively, until being used classes that do not require any attribute representing a tree node. This succession of nodes inside nodes are the principle of the method for constructing a DAST. However, the approach underlying Alma² uses a D_{ap} AST. So, the solution was to extend the major part of the *CXXXNode* classes, to create the already known animation nodes: the *AnimXXXNode* classes. This way, the *AnimXXXNode* nodes behave as a *CXXXNode*, but the information about the animation patterns is also stored inside them.

For instance, the *old CAssignNode* class is now *AnimAssignNode*, and besides the information about the variable and the value to assign, it needs the information about the actors and the animation functions used to animate the problem domain. Notice, however, that both kind of nodes are valid in the construction of the D_{ap} AST, because it is an hybrid tree (remember Section 4.3).

Appendix A list all the classes inside the family of classes *AnimXXXNode*, presenting the attributes they need to fill with information.

The Class Actor

The actor, in the D_{ap} AST approach, represents a concept of the problem domain underlying the DSL. Since the objective of the approach is to perform an animation of the problem domain regarding the effects of the program domain, the concept is represented by one or more images depending on the situations it can appear in the scene.

The information stored in each object of the class *Actor* describes, thus, the concept of the problem domain. Next is listed some of the attributes in this class, along with a small description.

figuresPath Is a list of system paths to the figures representing the poses that the actor can hold;

poses Is a list of buffered images representing the poses that the actor can hold, ready to be used by the animation engine;

usedPoses Is the list of poses the actor uses in a scene. Each pose is a zero based integer, pointing to the position of the image that represent the desired pose in the poses list;

label Regarding the cinema metaphor, it can be seen as the text the actor says in each scene it appears. In other words, the label describes something useful for the understanding of the scene;

state Represents the internal state of the actor. Each actor starts with a position in the plan, (X,Y), and an angle of rotation. These three values start with 0, and are used, internally for drawing purposes. This state can be incremented with other *variables*, reflecting the state of the program.

The utility of this class is further explored next, when the animation patterns are addressed.

The Animation Patterns Classes

The animation patterns are the most important classes in the D_{ap} AST approach. Figure 5.3 presents an abstract class named *AnimationPattern* which is then extended by a family of classes prefixed by *AP*. The animation patterns in this family have the same basic structure, thus, the use of such an abstract class. Also they have a similar behavior, introduced by the implementation of a common method named *animate*.

At this moment, there are available six patterns to produce the animations of the problem domain. The following list presents them:

APMove Is the pattern used to move the actor to any position in the 2D plan.

APRotate Is the pattern used to rotate the actor. It is not limited to 360° . The use of a superior value is processed natively by the *Java Virtual Machine* (JVM), so the effects can not be the desired ones.

APIdentity Is used to keep the actor as it is. This pattern is used to allow the redrawing of an actor in the same position in a sequent scene.

APLabel Is the pattern to set or change the text recited by the actor.

APAggregator This pattern differs from the others because it does not actuates over a single actor, but over a list of them. It is also used to aggregate poses of a same actor through out the scenes, otherwise only one pose would remain for scene. It can be used to show the trace the complete animation of a program.

The APAggregator is a higher-order animation pattern, because besides the aggregation utility, it embodies both the APMove and the APLabel patterns. Moreover, it can be extended to support any other *simple* pattern.

APCleanAggregator This pattern cleans the list of aggregated actors.

From the description given above, it is possible to see an intrinsic relation between the animation patterns and the actors. In fact, regarding Figure 5.3, there is a relationship between these classes. Each animation pattern is constructed with an *Actor*, its used poses and the remaining arguments that define the animation function. The animation function of a pattern is the method *animate*. This method uses the information, passed as argument to its pattern, to actuate accordingly on the actor's internal state. Figure 5.4 explains this process with more detail.

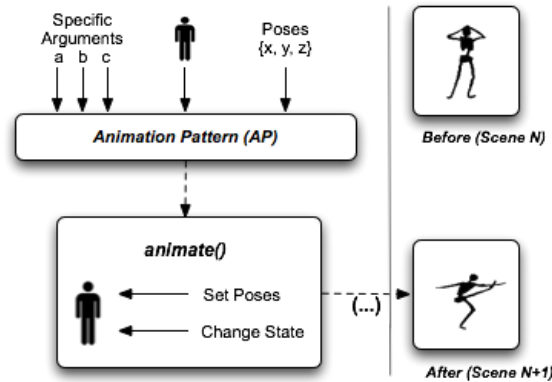


Figure 5.4: Animation Pattern Workflow

The animation patterns can only be applied to one actor. But in an animation node, several animation patterns can be specified and applied to the same actor. In this case the result in the actor's state and drawing is the union of all the animation function effects. When two patterns apply changes to the same variables of the actor's state, the pattern that provokes more significant changes on them, will prevail. For instance, if applying the *APIidentity* and the *APMove* to the same actor, the *APIidentity* is unnecessary because the changes provoked by the *APMove* are bigger.

Notice that all the patterns are processed before drawing the actors in the scene. This way, the actors are drawn once after calculating all the changes in their state. The algorithm of drawing the actors, along with that of traversing the $D_{ap}AST$ is presented in the next section.

5.1.3 Traversing the $D_{ap}AST$ — Practical Approach

Before, in Section 4.3.3, a theoretical approach on how to traverse and process the $D_{ap}AST$ was given. In the present section, the practical approach is explained. As usually happens, the theory and the practice are slightly different, because they tend to address different levels of conceptualization. But the main principles are equal to each one.

The algorithm used to traverse a tree structure, adopting a top-down strategy, is well known and fully documented [213]. The core processor of *Alma* is based on such strategy to process the nodes of the *DAST*. This way, the same strategy to traverse the $D_{ap}AST$, process the nodes and animate the problem domain, according to the animation patterns, was adopted.

In Figure 5.3 is presented the relation between the classes pertaining to the core of **Alma**². The classes *Backend*, *Visualize*, *Rewrite* define the processor of **Alma**. The first class is where the processor engine is located; it is responsible to perform the top-down tree traversal. The other two classes are invoked from this engine and they are responsible to perform the node processing. They can be seen as decision/action processors, because they, based on the information of the node, decide which suitable rule can be applied to the node, and apply it. So, each class has a database of rules for rewriting and visualizing, respectively, inside their structure, as have been explained in earlier sections. Considering Figure 5.2, these classes correspond, simultaneously, to the components inside the **Core Processor**, and the respective rules database.

In **Alma**², the processor was extended with the problem domain animation engine. The classes corresponding to that engine are the *Animate* and the *DrawingPanel*. The first class is connected to the class *Backend*, and follows the principle followed by the classes *Visualize* and *Rewrite*. The second is connected to the first and is used to draw the actors in the scene. Next paragraphs give a detailed description of the functionalities of these classes in the entire process of animation: from the tree traversal and actors preparation to their drawing in the scene.

Class *Animate* — Animation Node Processing Algorithm

When start processing the nodes of the D_{ap} AST the main engine invokes the labour of the classes *Rewrite*, *Visualize* and *Animate*, following this order. A brief explanation on the actions performed by the first two classes was already given. No more details are given, because it is out of this work's scope, and also because they are a simplified version of what happens within the class *Animate*, which is described next.

Before going further on the details about the class *Animate*, it is important to know, more than the attributes of such class, its main methods:

performAnimation() This method is used to traverse the tree, given a starting node, and to grab the animation data, whenever is possible. For processing the nodes' data, *doAnimation()* method is invoked;

doAnimation() This method is used to effectively animate one node by processing the animation patterns associated with the nodes.

These are the two main methods used to transform the data into useful information to be drawn. The first method receives a node, lets call it \mathcal{X} , of the D_{ap} AST. Then it follows a simple decision tree based, essentially, on two types of questions: (i) Is the node Animated? and (ii) Has the node children nodes? To give an answer to question (i), each node was extended with a boolean attribute to store whether it has been processed or not by the animation engine. Then, the node is *animated* or *not animated*. This way, \mathcal{X} is processed if it is *not animated*, but only if it has no children nodes. If \mathcal{X} has children nodes, and they are not animated, \mathcal{X} can not be processed. When \mathcal{X} can not be processed, the same algorithm is applied on its first child, until reach a node able to be animated, following a depth-first tree traversal strategy.

When \mathcal{X} is ready to be animated, then it is invoked method *doAnimation* applied to that node. This follows a simple algorithm based on the six steps listed below:

1. All the animation patterns are extracted from the animation node;
2. The *animate()* message, is sent to each animation pattern, to actuate on the respective actor, changing its state (see Figure 5.4);
3. From the list of animation patterns are extracted all the distinct actors;
4. The distinct actors are sent to the drawing engine;
5. The drawing method is invoked to paint the scene resulting from processing the node, and finally
6. The node is set as animated.

In the case of the animation engine, the animation rules database corresponds to the specific instructions inside the method *animate()*. Unlike the processors for rewriting and visualizing the program domain, this one does not look to the basic information of the D_{ap} AST node, but to the data representing the animation of the problem domain. Except, of course, to see if the node to process is part of a *loop*, *if... else*, *function* or *call* semantic patterns. These four concepts go through the same algorithm, but may perform different actions when processing the nodes.

The drawing of the actors is the last step of the animation nodes processing algorithm. As it is an interesting issue, its discussion is kept separated from the tree traversal algorithm explanation and, is addressed next.

Class *DrawingPanel* — Drawing the Actors

The class *DrawingPanel* is an extension to the known class *JPanel* of the Java API, and is used to support the drawing of the problem domain animation. Class *JPanel* is several times used as canvas to paint and display images. Its method *paintComponent()* enables the painting of the Java Panel component and all its content.

In Alma², the original interface was incremented with a new panel, identified before as *Behavior Animation Panel*. This panel is an instance of the class *DrawingPanel*, which uses the method *paintComponente()* to draw the behavior of the problem domain concepts. This class can be seen as the drawing engine of Alma².

As seen before, there are three important steps until the animation process is complete. The first is the tree traversal, the second is the animation nodes processing, transforming animation data into useful information, and finally, the third is the drawing of that information. The first two steps were already addressed in later sections. The last one, is about to be discussed in the following paragraphs. In fact, there is not rocket science behind the drawing algorithm because, remember, the information to be drawn are actors (standing in one or several poses), and each one of these actors and their poses, are described by images.

When executing the fifth step of the algorithm description for the *doAnimation* method, presented in the last section, the *DrawingPanel*'s method *makeAnimation()* is invoked. If one actor was represented only by one pose in each animation step²,

²One animation step corresponds to the complete processing of one animation node.

this method would not be necessary, because it would call the *paintComponent()* method directly, to effectively draw the actors in the scene. However, one actor can hold multiple poses in each animation step. This means that for each step more than one frame on the *Behavior Animation Panel* should appear. So, the job of *makeAnimation()* is to transform each actor in a pair (ac, p) , where *ac* is the reference for the actor, and *p* is one pose of the actor; and then, paint them following a certain order. Figure 5.5 explains clearly what happens.

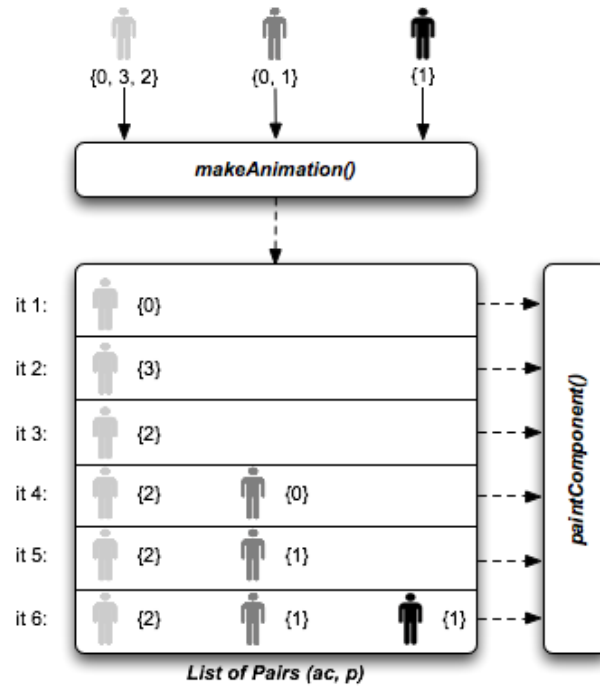


Figure 5.5: Process of Drawing Actors

The *makeAnimation()* method iterates on the list of distinct actors, creates the referred pairs, storing them in a list, and calls the method *paintComponent()* to draw the actors, based on that list of pairs, and on the actors' attributes. Notice that the method *paintComponent()* is called in each iteration inside the method *makeAnimation()*. Regard Figure 5.5 to see that in the first iteration (it 1), the list of pairs is composed of the pair (light-grey actor, pose 0); then, in the second iteration this pair is replaced by the pair with the same actor, but pose 3 and in the third iteration that pair is replaced by the one with the same actor, but pose 2. This is the painting of sequence of poses, creating the effect of an animation. Notice that in iteration four (it 4), the pair (light-gray actor, pose 2) remains, and is added the pair (grey actor, pose 0). As a new actor is being processed, the previous is repainted holding the last pose. This mechanism is repeated until all the distinct actors are processed. The remain steps of the drawing process are basic technical drawing instructions that are out this explanation bounds.

With this last algorithm explanation, the D_{ap} AST processing implementation and all the Alma² system is completely expounded without detailing it with boring code samples. Next section addresses the discussion on how to use the system, with great

focus on the method to create the D_{ap} AST.

5.2 Using the System

As have been told before, $Alma^2$ requires LISA (or a similar compiler generator) to create the FE for a DSL, whose programs can then be analyzed, using the $Alma^2$ interface. So, $Alma^2$ usage is twofold: in a first moment it is needed to prepare the FE, and in a second moment, it is needed to feed the system with DSLPs for analysis. The same way, it is needed two groups of users, which can perfectly intersect, to use the system. Figure 5.6 shows the two kind of users implied by $Alma^2$, and highlights also the intersection of these two groups of users.



Figure 5.6: The $Alma^2$ Users

The two groups of users are identified as *Experts* and *Program Readers*. The following paragraphs give a small description of these users' characteristics.

Program Readers. The PRs, as introduced in Section 3.1, are all the persons interested in understanding or analyzing a program. There is not a set of pre-defined characteristics for these users, because they can be programmers, students, and so on. They can be someone just interested in changing a program, not being necessarily a programmer. They are qualified to use $Alma^2$ interface, and not to prepare the FE.

Experts. The experts are persons with some background in compiler theory and AGs, but with expertise in the problem domain associated with the DSL. Because of their expertise, these users are qualified to work on the preparation of the FE. However, the major part of them can also pertain to the group of PRs allowed to analyze DSLPs.

Now that the possible kind of users are identified, it is important to show how they should perform their tasks. Next sections address the main steps on the execution of such tasks.

5.2.1 The Expert does...

As identified before, the main task executed by the expert, when using *Alma*², is the preparation of the FE for DSLs. This task implies the construction of the AG that would generate the compiler to parse the programs and build the D_{ap} AST. However, to create such a compiler, it is needed to study the application domain, the language, the connections between their concepts and to choose images capable of representing these concepts.

In a work previous to this master's, it was created a *Visual Language* (VL) for attribute grammars [141, 144, 139]. There it was studied a systematic approach for developing domain specific visual languages in general and the environment supporting them. From there, a conclusion was drawn: to conceive a domain-specific visual language, it is needed to have a great knowledge about the application domain in order to find the necessary number of problem domain concepts that would also be intrinsic to the language (program domain) and represented by images. The concepts at the problem domain are mapped into concepts representing the language constructs, and the images of the concepts are set as masks for these constructs. Notice the presence of three entities: (i) problem domain concepts; (ii) program domain concepts and (iii) images representing concepts. The images are interconnecting both domains, so they act as facades for the interconnections between them. This way, the search for appropriate imagery is a core task, but that can only be performed when having identified the interconnections between the domains.

Regarding the lesson learnt with the previous work, a parallel can be traced with the approach to create the FEs for *Alma*². This way, five steps are identified as necessary for the expert's main task:

1. **Domain Analysis.** This task should be the first of all. Here, the expert is encouraged to create a conceptual map or ontology [54], where is described the problem domain by means of concepts and relations between them.
2. **Language Analysis.** As the DSL to which the FE is being constructed, is already defined, the analysis of its main program domain concepts should be made without any troubles. In this case, a list of such concepts with a small description addressing their semantics, would ease the next steps.
3. **Interconnection of Domains.** With the ontology of the problem domain and the list of the program domain concepts, a relation between concepts of levels should be made. This task is made manually, but regarding the recent advances done in the area of concept localization [53, 121] it can be automatized.
4. **Search for Imagery.** This task requires deep knowledge on the problem domain, because the images should depict the concepts and their actions in such a domain.
5. **Creation the Attribute Grammar.** Resorting to LISA(or similar compiler generators), an AG should be defined. In the grammar, the main attribute of the root symbol, should synthesize a D_{ap} AST structure, so the generated compiler for the FE can create a D_{ap} AST when processing the input program.

Next paragraphs show the details on the definition of such AG.

Constructing a D_{ap} AST

Until here, the D_{ap} AST structure was always assumed as already constructed. But no explanation on how to build it was given. As the D_{ap} AST is the most important piece of Alma² system, it must be constructed correctly.

As told before, the D_{ap} AST derives from the DAST. So, the way of constructing them is similar in certain aspects. While in Alma the languages considered follow concrete paradigms like the imperative or the object-oriented, and their constructs are close to a low level generic format, in Alma², the DSLs, although can follow the same paradigms, have very different and higher level constructs. This way the construction of a D_{ap} AST is not as easy as the construction of a DAST, because these structures' constructs regard generic semantic aspects that are much closer to a GPL constructs than to a DSL's. But here enters the analysis made in the fourth previous steps, to diminish the effort of the expert.

The first thing to do is to write (or find already written) the CFG of the DSL, following the LISA syntax. When it is ready, the expert must add the necessary attributes. In the simplest case, only one attribute for each nonterminal is necessary. These attributes must be typed as *CAlmaNode*, except the one connected to the grammar's root symbol; this should be declared as a *CRootNode*.

In order to have a perfect compilation, a few packages should be imported. Listing 5.1 show the fundamental packages to import.

Listing 5.1: Fundamental Packages to Import

```
1 import Alma.*;
2 import Alma.animation.Actor;
3 import Alma.animation.patterns.*;
4 import Alma.animation.nodes.*;
```

After having the skeleton of the AG it is time to start using the D_{ap} AST constructs to define that structure for the DSL. Imagine the language in Listing 5.2 used to move a simple robot in four orthogonal directions.

Listing 5.2: Toy Language Syntax

```
1 robot → moves
2 moves → move moves
3       | ε
4 move  → N
5       | S
6       | E
7       | W
```

This language approaches itself to an imperative paradigm, however it is very high level, compared with imperative GPLs. This simple example addresses, besides others, three important aspects that would be present in great part of the D_{ap} AST creation, which is the main task in the preparation of FEs: (i) the injection of manufactured trees into automatic generated trees and (ii) the usage of animation nodes. (iii) the translation of the DSL into a GPL closer to the D_{ap} AST constructs;

Injecting a Sub-Tree. In this example, the language is used to move a robot in four directions. So, at least, the robot will have a state concerning its position. Lets

imagine such state as a pair (x, y) . This state is needed in every animation step, to know exactly where the robot is. But nowhere in the language is a place defining this state. This way, this state must be injected in the beginning of the $D_{ap}AST$. A possible way to do this is shown in Listing 5.3.

Listing 5.3: Example of a Tree Injection Method

```

1
2 rule program {
3   ROBOT ::= MOVES compute {
4     ROBOT.tree = new CStmtsNode(initState(), MOVES.tree);
5   };
6 }
7
8 (...)
9 method user_Definitions {
10 (...)
11
12 CToken tX = new CToken("posX", 1, 1, 1, false, false, "posX", "posX");
13 CToken tY = new CToken("posY", 1, 1, 1, false, false, "posY", "posY");
14
15 public CAlmaNode initState(){
16   CConstNode c0 = new CConstNode(0);
17   CAlmaNode nX = new CAssignNode(tX, c0);
18   CAlmaNode nY = new CAssignNode(tY, c0);
19
20   CDeclNode x = new CDeclNode(tX, "integer", null, nX);
21   CDeclNode y = new CDeclNode(tY, "integer", null, nY);
22
23   CAlmaNode decl = new CStmtsNode(x, y);
24
25   return decl;
26 }
27 }

```

From line 15 to line 26, the variables x and y are manually declared and initialized. The declaration is effectively performed in lines 20 and 21, using the *CDeclNode* construct. This construct is filled with information containing, besides other, the sub-tree of the assignment (lines 17 and 18). Figure 5.7(a) shows the tree generated correspondent to the return statement in line 25.

In line 4, the attribute *tree* of ROBOT is being assigned the construction of a tree composed of the tree synthesized in the attribute *tree* of the symbol MOVES, of the tree manually created in function *initState()*. Figure 5.7(b) shows the tree originated from the injection of the program's state.

Using Animation Nodes. The creation of the $D_{ap}AST$, is simply done by chaining the trees resulting from the synthesis of the attribute of any nonterminal in the grammar, and assigning it to other attribute of other nonterminal. This is the basic method to build the $D_{ap}AST$. The difficulty lies on the correct application of the semantic patterns, what is overcome with practice. For that small example, only basic semantic patterns were used. The construction of the $D_{ap}AST$ with animation nodes follows the same method; but this time the nodes have data about the animation of the problem domain. Imagine now, that after declaring the y variable, the program should paint a robot. Notice the differences from the declaration of the y variable in Listing 5.3 to the new declaration in Listing 5.4

Listing 5.4: Creation of an Animation Node

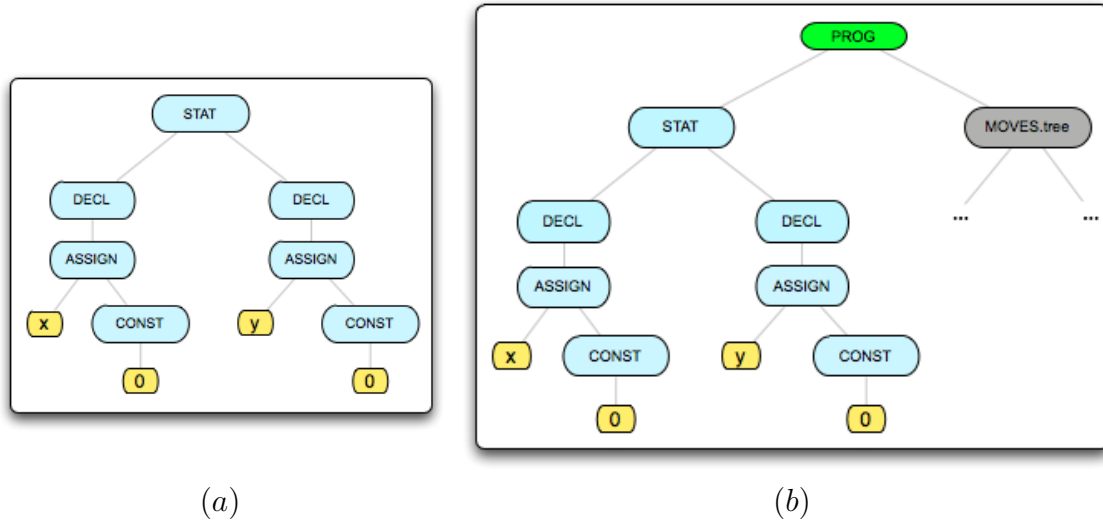


Figure 5.7: State Initialization: (a) The initialization's $D_{ap}AST$ generated; (b) The program's $D_{ap}AST$ generated with the injection of the initialization tree.

```

1
2 Actor robot = new Actor(new String[] {"robot.png"});
3
4 public CAlmaNode initState(){
5     (...)
6     CDeclNode y = new AnimDeclNode(tY, "integer", null, nY,
7         new AnimationPattern[] {
8             new APIdentity(robot, new int[] {0})
9         }
10    );
11    (...)
12 }
13 }
    
```

First, in Listing 5.4, line 2, is declared an Actor. This actor is represented by only one image (one pose), and assumes a default state. Then in lines 6 to 10 is created the animation node. As can be compared, the four arguments given to the *AnimDeclNode* construct, are the same given before. The difference is in the existence of the fifth argument. This fifth parameter is the list of animation patterns of that node. In this example is being used only one animation pattern (*APIdentity*). This pattern refers to the *robot* actor, which will hold its first pose.

Translation. The translation of a DSL into GPL is not a novelty. *Wu* [205] created a debugger for DSLs resorting to the Eclipse's Java debugger. As explained later in Section 3.5.2 the DSL is translated into Java, which is the code effectively debugged. *Wu* also identified three types of DSLs: imperative, declarative and hybrid; and all of them have a possible translation into Java.

In *Alma*², the translation into a GPL is not made in practice. In fact, the translation really made is of the DSL constructs into the $D_{ap}AST$'s. But as the $D_{ap}AST$ constructs are generic and are closer to those used by GPLs, a difficulty arises. But the expert, with knowledge on the domain and on the language can easily overcome it. It is only needed notion about the semantics of the language. For instance, the

language of Listing 5.2, is used to move a robot. The word *move* implies movement, and alteration of the robot's state. The instructions that can be used to move the robot are the initial letters of the four cardinal points: N for North, S for South, E for East and W for West. This means that the robot will move one step forward when reading the instruction N, one step backwards when reading the instruction S, one step to the left and one to the right when processing instructions W and E, respectively. After performed this simple mental exercise, it is now obvious how the state is changed. Regard the semantic rules presented in Listing 5.5.

Listing 5.5: Semantic Rules for the Robot Movement

```

1 initial state:
2     x = 0
3     y = 0
4
5 N → y = y+1
6 S → y = y-1
7 E → x = x+1
8 W → x = x-1

```

Notice that the rules presented in Listing 5.5 are written in a generic mathematical way; then it can be easily translated into the D_{ap} AST semantic patterns and animation nodes. Listing 5.6 shows an example of the tree construction for the N instruction. Notice also the use of animation patterns to define the action of the robot when the N instruction is executed.

Listing 5.6: Definition of Animation Nodes for one Instruction

```

1 rule Instruction_N {
2   MOVE ::= #N compute {
3     MOVE.tree = new AnimAssignNode(
4       tY,
5       new COperNode(
6         new CVarNode(tY),
7         new CConstNode(1),
8         "+"
9       ),
10      new AnimationPattern[] {
11        new APMove(robot, new int[] {0}, "posX", "posY")
12      }
13    );
14 };
15 }

```

The tree defined in Listing 5.6 reflects the semantic operation in the fifth line of Listing 5.5. The root node of this tree is an animation node composed of, besides others, the tree generated by the arithmetic operation construct (*COperNode*), and the list of animation patterns. The latter is composed of only one pattern, the *APMove*, where the actor *robot* is referred to use its first pose, and to assume the position stored in the program's state variables declared in Listing 5.3, lines 12 and 13, as *posX* and *posY*.

With this last paragraphs about the translation of the DSLs into the D_{ap} AST constructs, is complete the explanation of the experts' main task: build the FE for a DSL.

5.2.2 The Program Reader does...

The work of a PR, in *Alma*², is very simplified. It boils down to clicking a button, so the BE performs one animation step, processing one node of the D_{ap} AST. This simplification is important for concentration in what is really important: the program analysis.

This paragraph sums up the *story* of a normal usage of *Alma*². The PR starts *Alma*² prepared with the FE for a determined DSL, and feeds it with a program written in this DSL's syntax. The compiler of the FE creates the D_{ap} AST and the BE creates the first visualizations of the program domain: the interpretation tree, the identifiers table, and the source code text area. The behavior animation panel remains clean until the first animation node is processed. Figure 5.8 shows that situation. The processing of the D_{ap} AST starts when the PR clicks the *forward* button. If the node of the D_{ap} AST under processing is an animation node, the BE creates the visualization of the problem domain in the behavioral animation panel, following the strategy explained before, and the visualization of the program domain in the other panels. Otherwise it only depicts the visualization of the program domain.

So, besides starting *Alma*² and feeding it with a program for analysis, the PR needs to click a button to request the processing of another node of the D_{ap} AST, and for sometimes, is asked to input values in a very easy way.

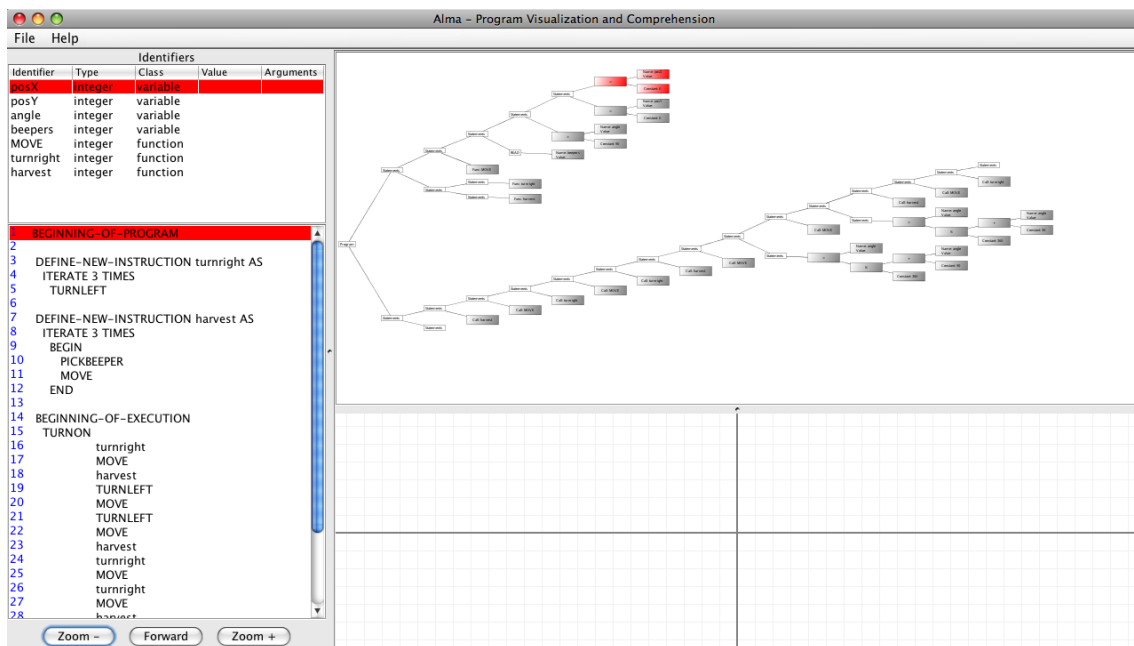


Figure 5.8: Initial state of *Alma*².

In fact, there is not much to add as PR tasks using *Alma*². This system frees the user mind from complications, opening doors to performing the program analysis, following the strategies used to, without losing time with details that cut the normal flow of the analysis.

5.3 Summary — Answering Question Two

Alma^2 is an improvement made to Alma . This improvement embodies the dealing with any DSL and the visualization of the effects of the program in the domain where the language was crafted to. To meet this new functionalities, Alma 's architecture was extended with a new engine for animation of the domain concepts. This engine takes advantage of the D_{ap}AST , which is an intermediate structure representing the semantics of a program and the animation details of the problem domain provoked by each operational semantic action. The animation details are defined in animation patterns, by means of actors and specific animation functions. To process the animation nodes, a twofold algorithm is used: first it handles the data in the animation pattern, transforming it in interesting information; and then, the drawing engine uses that information to draw the frames of the scene.

In order to present the animation of the domain concepts synchronized with the program domain visualization, a new panel on the user interface was added. It was called *Behavior Animation Panel*. This enriches the whole interface, presenting, now, four different views of the same program. One to see the internal values of the program state, other to see the program's source code, another to see the interpretation of the program's semantic and the program's flow, and a new one to see the effects of each semantic action at the problem domain level. This new view is, indubitably, a major help for understanding the program at a more conceptual and higher level: the problem domain level. The synchronization of the four views, offered by the BE 's core processor, helps the users to apply different cognitive models, from the bottom-up to the top-down models.

The improvement made on Alma ends in a new DsPCTool capable of giving more features for program understanding. With this achievement, is possible to give an affirmative response to Question 2. So, a new theorem (Theorem 3) raises from this answer, and it says the following:

Theorem 3. *It is possible to improve existent Program Comprehension tools to enhance the understanding of DSL programs by the end users.*

In this chapter was also addressed the way of defining the FE for the DSLs. The method to define the D_{ap}AST using an AG follows a strategy of chaining animation nodes. This chaining can be prepared for synthesis, or can be prepared manually, for later injection of trees. This is a task produced only by experts in compilers and in the problem domain of the DSL.

An important conclusion is that the D_{ap}AST approach can be applied both to DSLs and GPLs, but in the latter, the output can be different from the desired one. The problem is that in a GPL the problem domain of a system is not known *a priori*, due to the fact that a GPL is used in several domains of application. So, the FE for a given program in a GPL must be adapted or rebuilt according to the problem domain desired. For instance, imagine that a FE is defined for a GPL, using pre-defined images for animation of a meteorologic station program. This would work correctly if the program passed later to Alma^2 is the program for the meteorological station. But in the same GPL, a program for traffic management, can be developed; and the FE built for the meteorological station program, will accept the this second program, but the animation of the problem domain will not be the desired one.

So the conclusion is: for a DSL only one FE needs to be built for all its programs, for a GPL, one FE must be built to accept each program, which is not a desirable situation.

Chapter 6

Case Studies

Studies perfect nature and are perfected still by experience.

Francis Bacon

In this chapter, some case studies are defined. Each one of them is used to support and assess the ideas stated before and to evaluate the improvement made on the program comprehension system, *Alma*. The case studies are based on DSLs, since these kind of programming languages are the main issue of this thesis.

Each case study is defined and described through several well-defined steps, so the issues related with the comprehension of a DSLP are addressed. First, the problem domain associated with the DSL is defined resorting to ontologies; then the language is formally presented. The third step consists on the research for a set of images capable of describing situations at the problem domain of the language. This introduces the conception of the problem domain animation, which is divided into two parts: (i) the static conception and (ii) the dynamic conception.

The static conception, or visualization, addresses the creation of connections between the pieces of the problem domain (the images and the situations) and the pieces of the program domain (the syntactic parts of the language as well as their semantics). The main objective of this conception is to show one (possible) visualization of the problem domain associated with the program domain. The dynamic conception, or animation, consists in describing how the set of visualizations produce an animation.

The results obtained from the usage of *Alma*² will be used to draw conclusions about what have been done and said throughout this master's thesis.

6.1 Karel's Language

Karel Čapek, a well-known Czech writer and playwright, created a dramatic play named R.U.R (*Rossum's Universal Robots*)¹, which was premiered in Prague, in the year of 1921. Although it was a successful play, and being foreseen its adaptation to theaters near 2011, this piece of art left its mark on history for being the first time

¹URL: <http://ebooks.adelaide.edu.au/c/capek/karel/rur>

the word *robot* was used. Because of that, Karel Čapek is pointed as the father of the word and the concept *robot*.

Based on the origins of the word *robot*, Richard Pattis, developed a small programming language named *Karel's Language* [147]. It is a language to control a small robot, called *Karel*, in a small virtual world. This language is used in academic programming courses among several universities, because it eases the task of teaching and learning the basic concepts behind imperative programming: the logic and the structure of programs. Since the language's release, many increments and versions have been made. Nowadays, the language can also be found regarding object-oriented programming aspects [17]; and there are also some simulation tools crafted on purpose for this language. Monty Karel Simulators² are some examples of these tools.

6.1.1 Problem Domain Definition

Looking back to *Karel's* first implementation, the robot is neither a full-featured nor a sophisticated machine. Besides turning itself on or off, moving one step ahead, turning left, picking objects from the ground and putting them back in 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. Due to this limited behavior, it only understands a few basic instructions, and so the language to control the robot is not very complex. However this does not limit the possibilities of using it to solve different and challenging problems.

The tasks this robot can perform fit inside the boundaries of handling objects from a place to another. It seems a very reduced problem domain, but regarding with attention, there are many situations where using robots to change objects from their position is very necessary. Cleaning up a child's room, with toys spread all over the ground is a complex task for a human; waking up early every cold morning to pick up the milk (or the bread) in the outside, is also an hard task; like these, many other jobs can be performed by an automatic system capable of moving around and sensing edges or objects.

The problem domain associated with the *Karel's Language* is depicted in Figure 6.1 by means of an ontology.

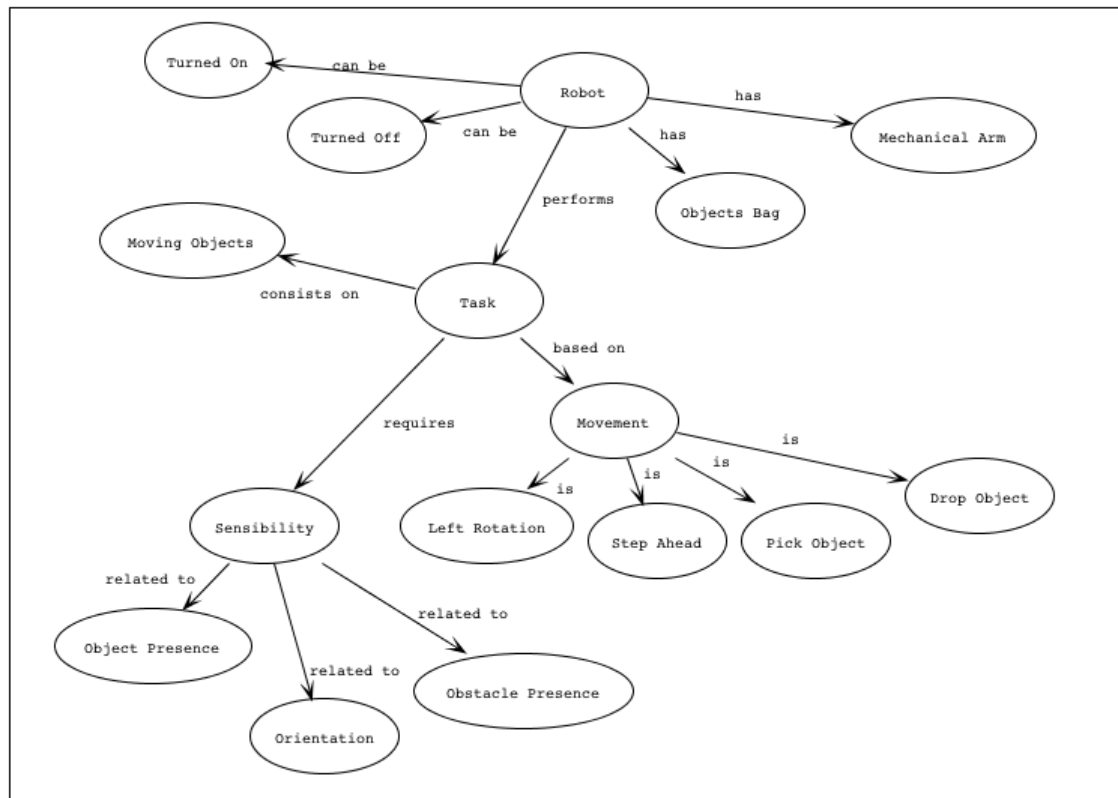
This ontology gives an initial perception of the main concepts related with the domain for which *Karel's Language* was designed. As a DSL, these concepts are very well expressed in the language which is formally presented in the next sub-section.

6.1.2 Language Formal Definition

The definition of the *Karel's Language* is presented in Listing 6.1³. A *Context Free Grammar* (CFG) is used to formally specify it. It uses *Extended Backus-Naur Form* (EBNF) notation for an easier comprehension. The nonterminal symbols appear in small letters, while the terminals are written in capital letters.

²URL: <http://csis.pace.edu/~bergin/MontyKarel/simulatorinfo.htm>

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

Figure 6.1: Problem Domain associated with the *Karel's Language*

Listing 6.1: Formal Definition of *Karel's Language*

1		
2	start	→ BEGINNING-OF-PROGRAM program END-OF-PROGRAM
3	program	→ definition* BEGINNING-OF-EXECUTION statement* END-OF-EXECUTION
4	definition	→ DEFINE-NEW-INSTRUCTION identifier AS statement
5	statement	→ block iteration
6		loop conditional
7		instruction
8	block	→ BEGIN statement* END
9	iteration	→ ITERATE number TIMES statement
10	loop	→ WHILE condition DO statement
11	conditional	→ IF condition THEN statement (ELSE statement)?
12	instruction	→ TURNON MOVE TURNLEFT
13		PICKBEEPER PUTBEEPER
14		TURNOFF identifier
15	condition	→ FRONT-IS-CLEAR FRONT-IS-BLOCKED
16		LEFT-IS-CLEAR LEFT-IS-BLOCKED
17		RIGHT-IS-CLEAR RIGHT-IS-BLOCKED
18		BACK-IS-CLEAR BACK-IS-BLOCKED
19		NEXT-TO-A-BEEPER NOT-NEXT-TO-A-BEEPER
20		ANY-BEEPERS-IN-BEEPER-BAG NO-BEEPERS-IN-BEEPER-BAG
21		FACING-NORTH NOT-FACING-NORTH
22		FACING-SOUTH NOT-FACING-SOUTH
23		FACING-EAST NOT-FACING-EAST
24		FACING-WEST NOT-FACING-WEST
25	identifier	→ [a-z] ([a-z] [0-9]+)*

All the terminal symbols (excluding the *Regular Expression* (RE) associated with the symbol *identifier*) correspond to a keyword, making the language truly verbose. But the presence of all this verbosity in the language is not bad: it diminishes, *per se*, the distance between the problem domain and the associated DSL. Regarding the overall language definition and recalling the problem domain ontology depicted in Figure 6.1, is easy to relate the concepts to the commands which the robot understands, and is also easy to understand what will be the robot's behavior when each instruction occurs.

The language is based on linear instructions that occur sequentially in the program. It presents also some control flow structures like *if...else* or *loop* instructions. This approximates the DSL to the imperative programming languages. However, there are neither variable declarations nor normal expressions (like addition or multiplication).

6.1.3 Conception: Mappings, Visualization and Animation

At this moment, the problem domain and the DSL are described. The main picture that should remain in memory is that *Karel's Language* is used to control a robot to move around, pick objects from a position, and drop them in another place. To achieve a possible visualization of the problem domain, these concepts, related with the behavior of the controlled object, are important. But such higher-level concepts have to be intrinsically related with some parts of the syntax and semantics of the DSL. Only this way it would be possible to visualize a program at problem domain level.

In this section, a threesome relation between problem domain, images and program domain, in order to attain the visualization, is created. But before creating such relation, the more relevant concepts at problem and program domain level are highlighted. In the following list, the concepts at the problem domain level are

described:

- **Turned Off** The controlled object (robot) is turned off;
- **Turned On** The robot is turned on;
- **Left Rotation** The robot makes a rotation to its left side, by a given angle;
- **Step Ahead** The robot gives a step ahead. That is, changes its actual position;
- **Pick Object** The robot picks an object from the ground and stores it in its bag;
- **Drop Object** The robot takes an object from its bag, and drops the same into the ground.







The other concepts presented in the ontology (Figure 6.1) are not directly related with the *physical behavior* of the robot. Sensibility (the sensors of the robot) for instance, constraint the behavior, but do not change the behavioral state of the controlled object. So it is possible to discard these concepts from the visualization matter. The same happens with the program domain concepts. Only those related with an alteration on the robot's state (those capable of provoking the movement of the robot) are taken into account. The following list describes the concepts at the program domain.

- **TURNOFF** The command to make the robot turn off;
- **TURNON** The command to make the robot turn on;
- **TURNLEFT** The command to make the robot perform a rotation to its left side, by a given angle;
- **MOVE** The command to instruct the robot to give a step ahead, on the direction it is turned to;
- **PICKBEEPER** The command to make the robot pick an object from the ground;
- **PUTBEEPER** The command to make the robot drop an object into the ground.

Notice that all the six concepts listed above are, in fact, instructions that can be used to make the robot move around and perform the tasks the programmer desires it to perform. These keywords are very similar to those related with the problem domain, so the distance between the problem domain and the program domain is small. This makes the task of establishing connections between both domains a straightforward and easy job. Table 6.1 presents the connections between the program and problem domain concepts and the main images that depict the visualization of such concepts.

All the images represent the robot actuating in different situations. As they are static images and the concepts represent dynamic situations (except the first

Table 6.1: Connection of concepts to images in *Karel's Language*

PROBLEM DOMAIN CONCEPTS	IMAGES	PROGRAM DOMAIN CONCEPTS
Turned Off		TURNOFF
Turned On		TURNON
Step Ahead		MOVE
Left Rotation		TURNLEFT
Pick Object		PICKBEEPER
Drop Object		PUTBEEPER

two), the effect of presenting the images may be not the desired. So that, the image sequences from Figure 6.2 to 6.5 are used to give a more interesting view of the visualization of the domain.

The first two images in Table 6.1 represent the robot turned off and turned on. This is visually noted by a small red or green led in the back of the robot body.

The third image, connected to the `MOVE` and *Step Ahead* concepts, depicts a movement where the robot changes its position. This simple sequence is shown in Figure 6.2.

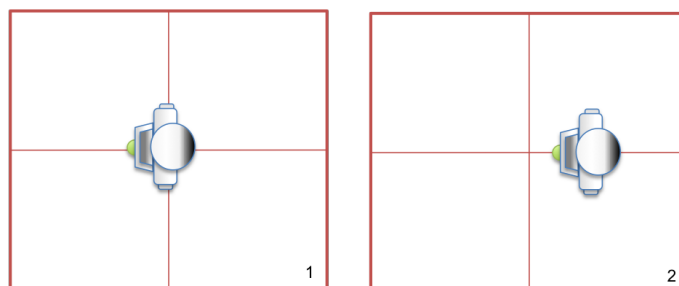


Figure 6.2: Sequence to move the robot

The fourth shows how the robot appears after a rotation to the left by an angle of 90° . In Figure 6.3, the sequence starts with the robot facing *East* (frame 1), and ends with it turned to *North* (frame 2). Obviously, this also happens when the robot has any other orientation rather than *East*.

The fifth image is composed by two figures: the robot and an object in its claw.

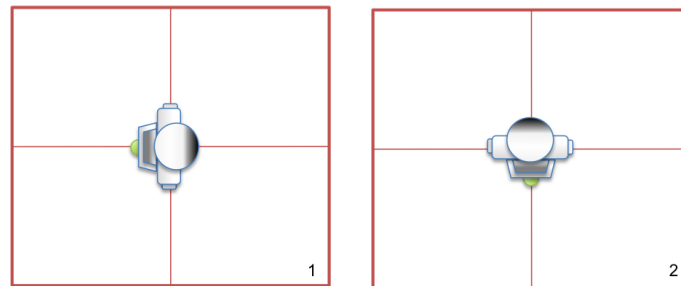


Figure 6.3: Sequence to rotate the robot

This presents how the action of catching an object is depicted. The sequence of images in Figure 6.4 shows it clearly. When picking an object, the robot starts in its normal configuration (frame 1), then it stretches its mechanical arm (frame 2) and catches the object (frame 3). In the end, the robot lows its arm, going back to its normal configuration (frame 4).

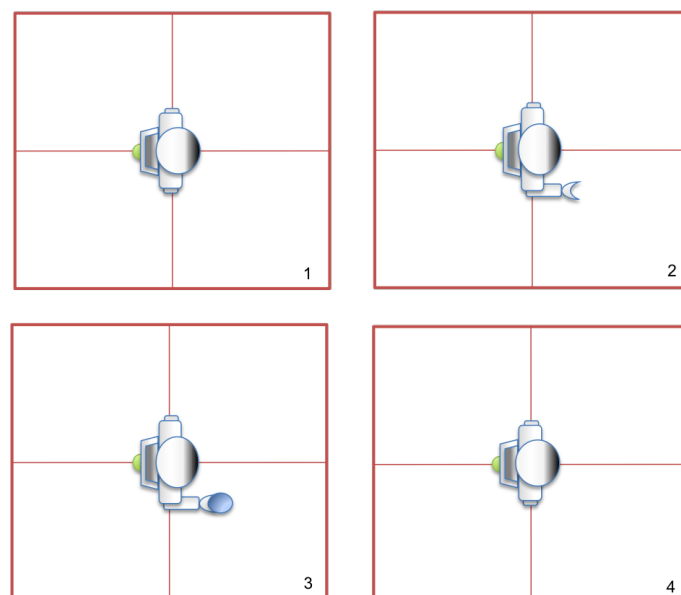


Figure 6.4: Sequence to pick an object

The last image in Table 6.1 is dual from the fifth one: the robot appears with its mechanical arm stretched, but with no object in the claw because it was dropped. The sequence of dropping an object is illustrated in Figure 6.5. Comparing with the sequence in Figure 6.4, frames 2 and 3 changed their position. To drop an object, the robot has the object in its claw, then throws it away, what is represented by a stretched empty mechanical arm (frame 3).

Notice that after picking and dropping an object, the robot stays in its normal configuration. There is not a visualization whether the robot has or not objects in its bag. To solve this problem, other visual elements like textual labels can be added in order to contextualize the visualization and animation. Figure 6.6 presents

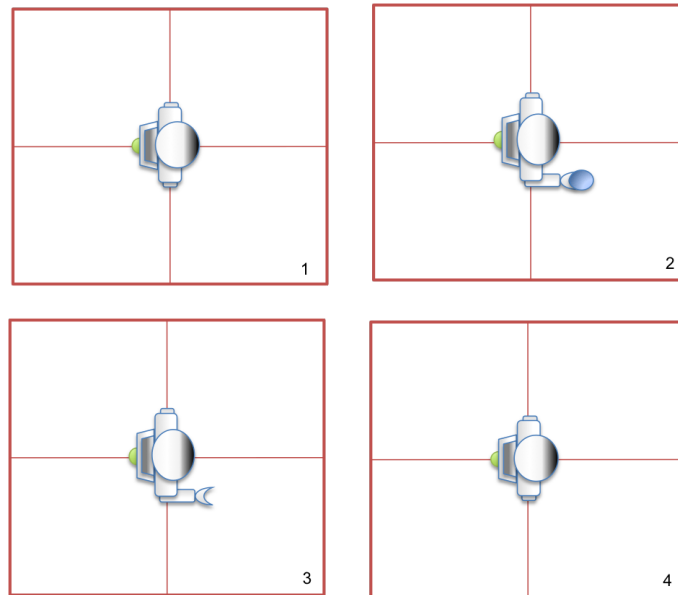


Figure 6.5: Sequence to drop an object

an example of a labeled pose of the robot, where is specified how many objects the robot grabbed so far.

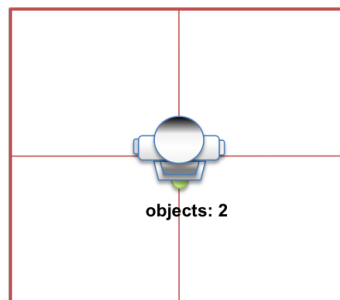


Figure 6.6: Example of a label as a resource element for the problem domain visualization

6.1.4 Animation Using Alma

In this section, a program in *Karel's Language* is automatically animated, resorting to the extended version of Alma.

There are many tasks the robot can perform and which can be coded in *Karel's Language*. The task used here is based on the harvest problem⁴: imagine that the robot is sent to work as a carrot lanyard. Its main job is to harvest a field of carrots. The field has three rows and each row has only three carrots. The robot starts turned off, facing *north*, with the field of carrots at its right side.

⁴URL: <http://www.cs.mtsu.edu/~untch/karel/functions.html#style>

The code correspondent to the described problem is presented in Listing 6.2. The program is very linear. The instructions occur in a sequence and all are executed.

Listing 6.2: The code for the Harvest Problem

```

1
2 BEGINNING-OF-PROGRAM
3
4 DEFINE-NEW-INSTRUCTION turnright AS
5     ITERATE 3 TIMES
6     TURNLEFT
7
8 DEFINE-NEW-INSTRUCTION harvest AS
9     ITERATE 3 TIMES
10    BEGIN
11    PICKBEEPER
12    MOVE
13    END
14
15 BEGINNING-OF-EXECUTION
16    TURNON
17    turnright
18    MOVE
19    harvest
20    TURNLEFT
21    MOVE
22    TURNLEFT
23    MOVE
24    harvest
25    turnright
26    MOVE
27    turnright
28    MOVE
29    harvest
30    TURNOFF
31 END-OF-EXECUTION
32
33 END-OF-PROGRAM

```

Despite of being a small program, its animation is extensive and requires a lot of images to depict all the process. In the present sub-section the objective is not to show the complete animation of the program, instead it is to show how the program can be visualized at the problem domain level, keeping a synchronization with the visualization at the program domain level.

Figure 6.7 shows the visualization produced after the execution of the first *turnright* instruction in the Harvest program presented before; and Figure 6.8 shows the visualization produced after the execution of the sequent instruction (*move*) in the the same program.

The first instruction of the program, *turnright*, commands the robot to turn three times to its left. The second instruction, *move*, makes the robot step one unit into the direction it is facing. Regarding Figure 6.7 (the same aspects can be seen in figure 6.8), the source code viewer highlights the line of the instruction being executed. The execution tree shows that the first instruction of the program was successfully executed (the rightmost green rectangle). Also the Identifiers table shows the values of the variables that compose the internal state of the robot.

Confronting Figure 6.7 with Figure 6.8, can be noticed that the robot changed its position. In the former, the robot is precisely in the center of the panel, and in the latter, the robot is one unit/step ahead. This happened because the *move* instruction, which is the statement executed between those figures, has an operational

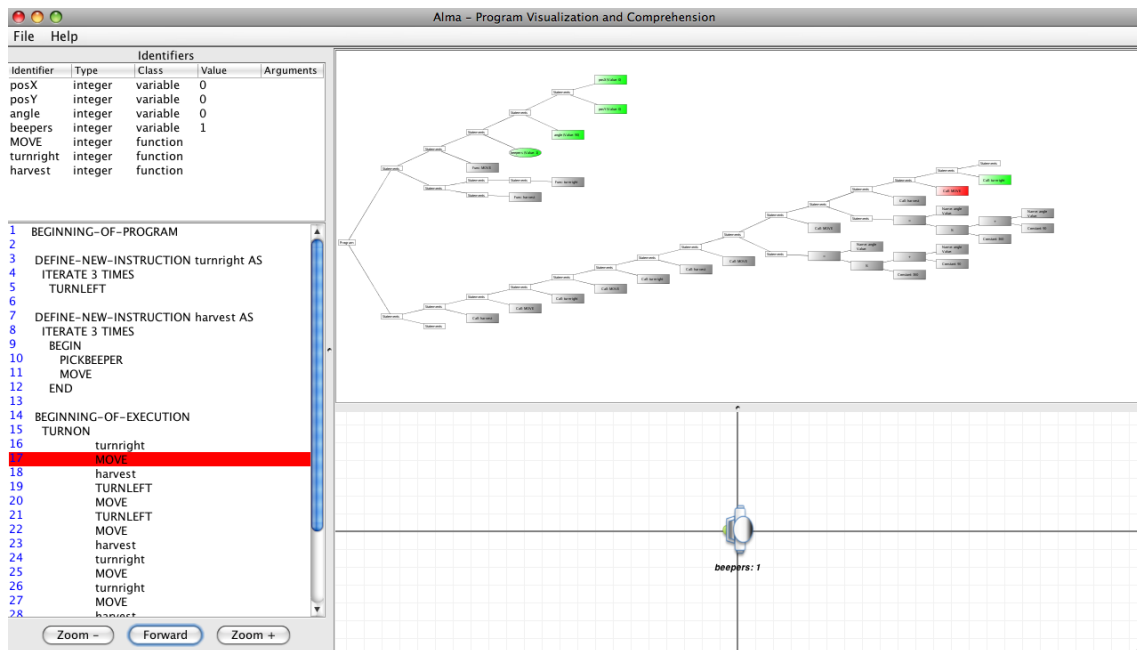


Figure 6.7: Visualization after executed the first *turnright* instruction

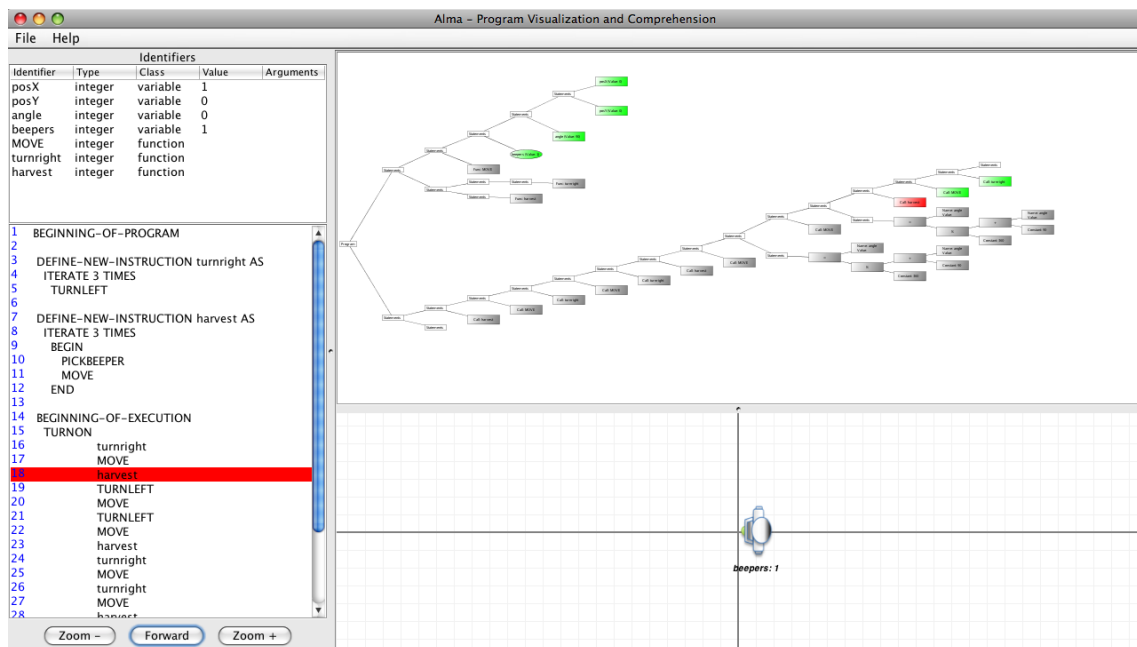


Figure 6.8: Visualization after executed the first *MOVE* instruction

semantics at program level that reflects itself in the problem domain by means of making the robot move into the direction it is facing.

All together, the tree and the source code viewer, the identifiers table and the animation panel are unified into a synchronized visualization and animation of the program and problem domains.

Figure Figure 6.9 depict other situation of the problem domain animation, using the Alma². The animation panel depicts the second frame associated with the sequence of frames designed for the concept of picking an object from the ground (see Figure 6.4). This image appears in the context of executing the user-defined *harvest* instruction, when the robot is harvesting the second row of carrots.

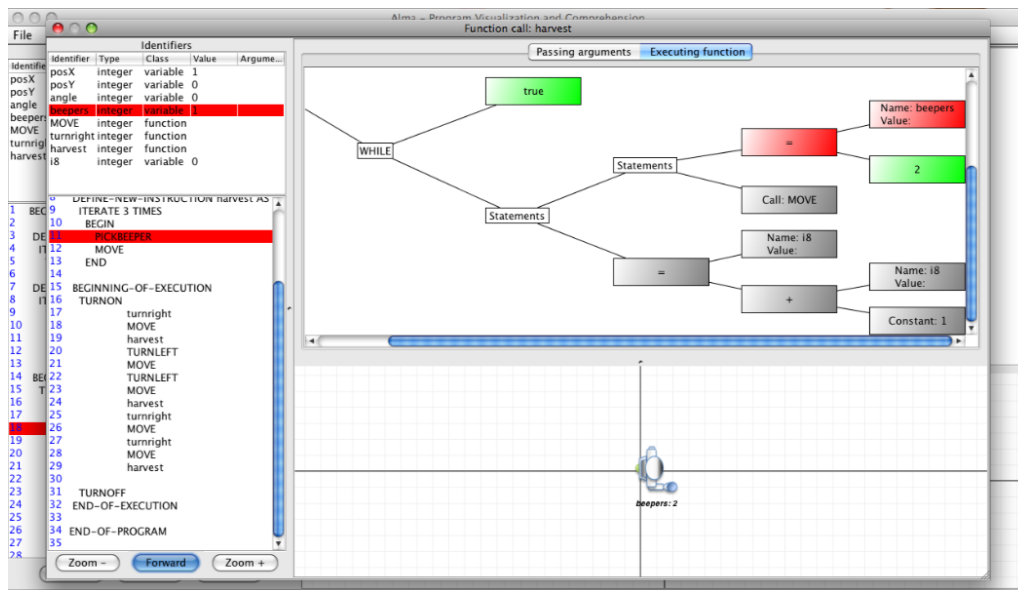


Figure 6.9: Visualization of a frame of the *picking object* sequence

The presented figures show the adquacy of the Alma² for the proposed objectives: the problem domain is visualized, and it is constantly synchronized with the program domain visualization. This synchronization is, no doubts, worthwhile and effective. It shows not only the program being executed but also which are the effects of such execution in the object controlled by it.

6.2 The Lavanda Language

Karel's Language, despite being an high level DSL, has some aspects that are typical in a imperative GPL. So the process of mapping these aspects to the patterns of the Alma² is a simplified task. When dealing with a more declarative or descriptive DSL, the difficulties to create such mapping will be bigger, because the level of concepts are dissimilar. In this section, a descriptive language is used to show that the difficulties identified in this paragraph can be overcome.

The *Lavanda Language* is a simple descriptive DSL. It was designed with the intuit of being the case study for the analysis of some compiler generator tools carried out

by *da Cruz et al.* in [51]. In addition, the language is being used for reviewing other compiler generator tools⁵, and is also used in compiler classes on several exercises. The sentences of *Lavanda Language* are used to describe the contents of laundry bags (bags with clothes or linens to be washed) in a distributed laundry service.

6.2.1 Problem Domain Definition

As introduced before, the *Lavanda Language* is used to describe the contents of bags in a distributed laundry company. This language was not tailored with the purpose of being used by any real laundry company, but it gathers into its design some aspects and concepts used in such a domain.

For the design of the language, a laundry company is constituted by a central building and several small shops, so called collecting points. These collection points, as the name recalls, are used to collect customers' laundry bags and send them to the central building. As bags come from all the shops, it is needed to keep tracking of the map between the collecting point and the customers who ask for the laundry service. These maps were named as *ordering notes* in [51];

A laundry bag may contain body clothes or household linens. The color of these pieces may vary as well as the raw-materials used in their fabrication.

For a visual description of the problem domain underlying the language in question, Figure 6.10 presents an ontology with the more relevant concepts associated with such domain.

The main objective of these ordering notes is to describe the contents of a bag. Altogether, these notes would be used to calculate the number of bags that are received by the central building of the laundry company, and the number of pieces of cloth correspondent to each type of fabrication.

6.2.2 Language Formal Definition

In this section, the formal definition of the *Lavanda Language* is presented. In Listing 6.3 is resorted to a CFG to define the language.

Listing 6.3: Formal Definition of the *Lavanda Language*

```

1 note      → header bags
2 header    → date id
3 bags      → bag+
4 bag       → num id '(' lots ')'
5 lots      → lot (',' lot)*
6 lot       → type num
7 type      → class '-' tint '-' material
8 class     → BODY | HOUSE
9 tint      → WHITE | COLOR
10 material → COTTON | WOOL | FIBER
11 date     → [0-2][0-9]\-[0-9][0-9]\-[0-2][0-9][0-9][0-9]
12 id       → [a-z]+
13 num      → [0-9]+

```

The concepts of the language at the program level are very close to those identified for the problem domain. The sentences of this language are precisely the

⁵The site of these reviews can be found in: <http://epl.di.uminho.pt/~gepl/LP/CompilerGenerators.html>

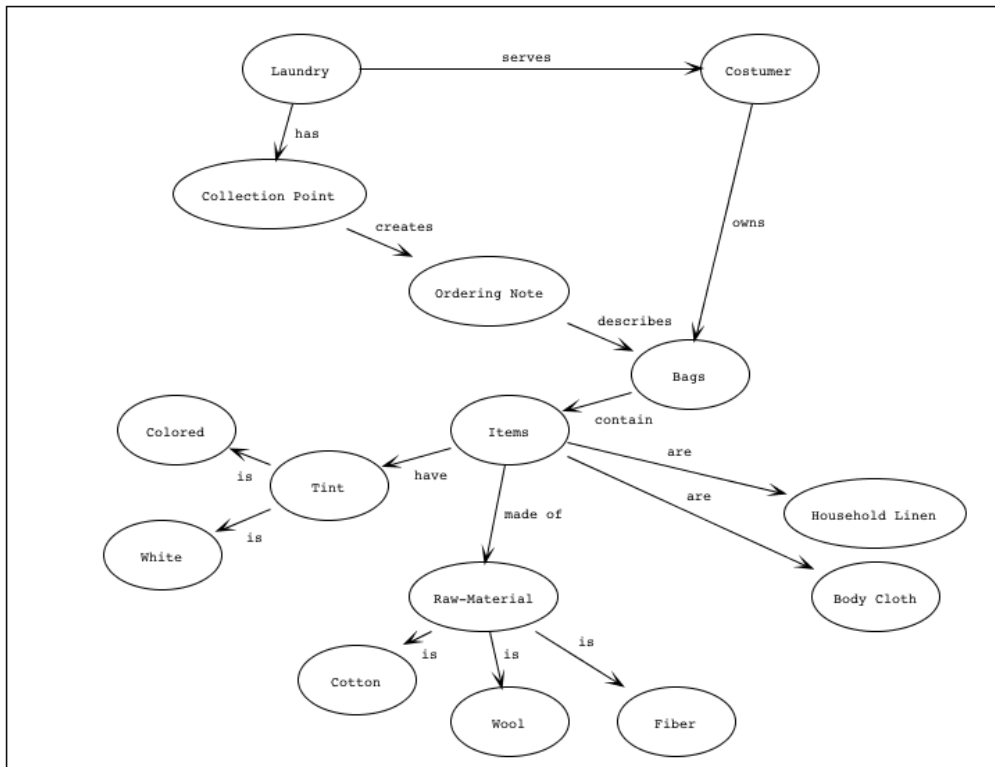


Figure 6.10: Problem Domain associated with the *Lavanda Language*

ordering notes introduced before. Briefly, an *ordering note* defines a set of bags and for each bag is described its content as well as the identification of the customer. The content of each bag is one or more items within a type of fabrication. The type of fabrication, in its turn, is defined by a class (keywords `BODY` and `HOUSE`), a tint (keywords `WHITE` and `COLOR`) and a raw-material (keywords `COTTON`, `WOOL` and `FIBER`).

6.2.3 Conception: Mappings, Visualization and Animation

Summing up, the main idea that should prevail about the *Lavanda Language*, is that it is used to describe the bags and its contents, in order to ease the tracking of the number of customers (number of bags) and the number and types of the clothes that are sent inside each bag.

In a first look, such a descriptive language has no semantics associated, then it has no operational behavior. This is a reality for the major part of the declarative DSLs. While in an imperative DSL is easy to identify one or several controlled objects, in declarative ones, it is not so easy. This is because, in general, they are processed (if so) to retrieve some results, and not to lead an object to perform an activity, as it is the example of the *Karel's Language*. Thus, the semantics of these DSLs, is in the way how those results are calculated; and obviously, these operational calculi are intrinsically related with the concepts of the problem domain presented in Figure 6.10.

The following list identifies and describes the concepts of the problem domain, that are directly related with the expected visualization:

- **Bags** Object used to store the several pieces of cloth from one customer;
- **Items** The pieces of cloth;
- **Colored and White** The tint of the piece of cloth;
- **Cotton, Wool and Fiber** The raw-material of what the piece of cloth is made of;
- **Household Linen** The item is for domestic use;
- **Body Cloth** The item is to be dressed by a person.










The other concepts presented in the ontology (Figure 6.10) can be discarded from the visualization of the problem domain, because they are contextual concepts, and do not interfere directly in the problem visualization. The same happens with the program domain concepts. Some of those that appear in the formal definition of the language, and will appear later in the programs specification, can be discarded, because they do not interfere in the visualization. The following list describes the relevant concepts at the program domain level:

- **bag** Supports the description of an order from a customer;
- **lot** Describes each type of cloth and its quantity, that goes inside the bag;
- **BODY and HOUSE** Define the class of the item in the bag;
- **WHITE and COLOR** Define the color aspect of the items in bag: they can be either colored or white;
- **COTTON, WOOL and FIBER** Define the raw-material of what the item in the lot is made of. It can be made of one of cotton, wool, or fiber.

Differently from what happened in the other case study, in this one the concepts of the problem and program domains do not represent actions to control the object, but words that are used to describe the ordering notes. An ordering note can, then, be seen as the main actor in this language, but is an actor composed of other sub-actors addressed by the language. Table 6.2 shows the easy connection between the concepts at both problem and program domain, and adds the figures that represent those sub-actors in the language.

The images used to represent the bridge between the concepts at both problem and program domain, are very illustrative. The image in the first row of Table 6.2 represents a bag full of stuffs, which are intended to be clothes because of the clothes that remain outside the bag. The image in the second row represents a piece of cloth, by describing (visually) its type, w.r.t. the notion of type of fabrication given before. Thus, it is an image composed of three other images that represent other concepts. As there are 12 possible piece of clothes for this laundry, only one was shown in

Table 6.2: Connection of concepts to images in *Lavanda Language*

PROBLEM DOMAIN CONCEPTS	IMAGES	PROGRAM DOMAIN CONCEPTS
Bags		bag
Items		lot
Colored		COLOR
White		WHITE
Cotton		COTTON
Wool		WOOL
Fiber		FIBER
Household linen		HOUSE
Body		BODY

the image. The third and fourth rows represent the tint of the clothes: a white paper sheet and a colored rainbow. The next three images are used to represent the raw-material of the piece of cloth: a cotton flower, a sheep and a pipe of lines represent, respectively, the cotton, the wool and the fiber. The last two rows present the images for the class concepts: a house and a t-shirt are used to identify the cloth to be used in the house affairs and to be dressed (body clothes), respectively.

The conception of an animation for a declarative language is not a simple task, because it depends on the operational behavior of that language, and much of the times, this operational behavior is not obvious. But for *Lavanda Language*, the problem of counting bags, lots and summarizing them up, in the end, was identified. This determines, automatically, an operational behavior behind the language, and, then, is easier to conceive a behavior for the several concepts on the problem domain.

The animation defined for *Lavanda Language* is mainly composed of three steps:

1. The description of the lots is displayed in the screen during the synthesis of the lots that are stored inside a customer bag;
2. The bags that already are processed are displayed in the screen, during all the animation;
3. After processing a bag, is shown a summary table of the lots that already compose the contents of all the bags.

When a specification is completely interpreted, the final visualization (the final screen) represents the number of bags that were ordered along with the summary table of all the lots that compose each bag.

In the next section, is shown a *Lavanda Language* program and its animation using *Alma*², where are presented figures that corroborate the three stepped animation conceived in the present section.

6.2.4 Animation Using Alma

The programs written in the *Lavanda Language*, are always very similar. The difference between them is in the contents of the bags and in the size of the specification. In this section, a simple program is proposed for an automatic animation using *Alma*². Listing 6.4 presents the code of such a program, where three clients send their clothes to the laundry.

Listing 6.4: Specification in *Lavanda Language*

```
1
2 23-07-2009 Laundry
3   1 Joan
4     (body-color-wool 3,
5      house-color-wool 2,
6      body-white-cotton 5,
7      house-white-fiber 3)
8
9   2 Ulysses
10    (body-color-cotton 1,
11     house-white-fiber 4,
12     house-white-cotton 2)
13
```

```

14 3 Layla
15 (house-color-wool 10)

```

In fact this specification is very simple. As it is considering just three clients, the number of bags is immediately retrieved. But notice that a real laundry have, for sure, more than three customers in a day. On the other hand, the description of contents of the bag, is something not so easy to read, regardless of the number of clients. A more complex handwork would be needed to count the lots of clothes in the bag, and also to distribute this number by the 12 different types of fabrication.

Figures 6.11 to 6.14 illustrate some steps of the global animation for the program presented in Listing 6.4.

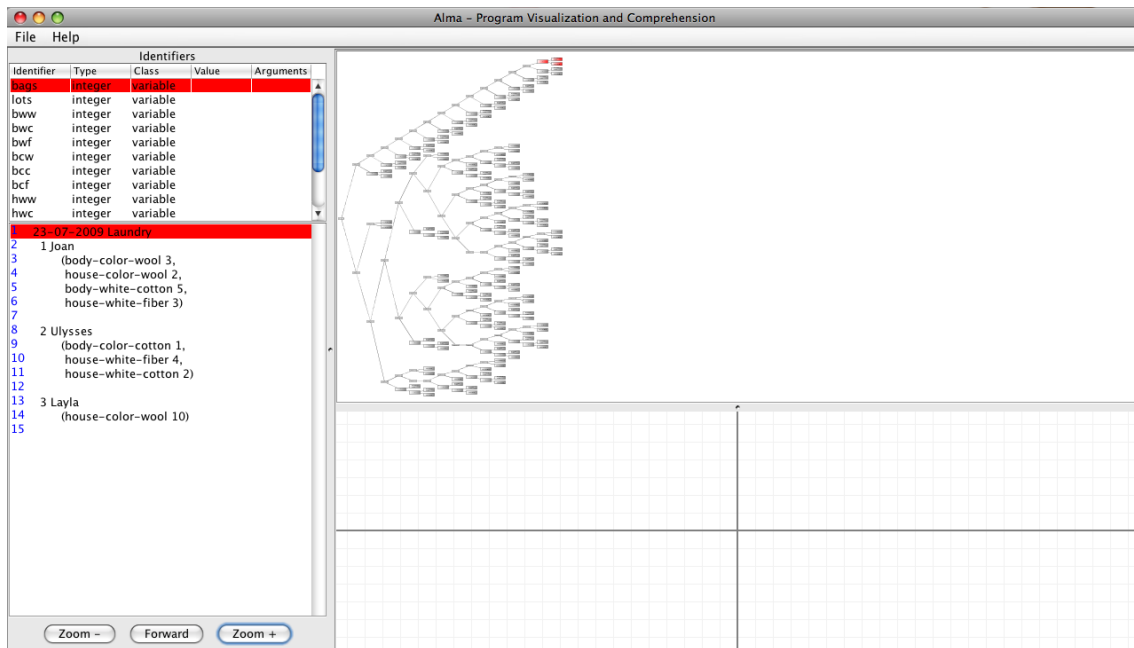


Figure 6.11: Initial Visualization of a *Lavanda Language* specification

In Figure 6.11 is presented the *Alma*² interface with its four characteristic views: the IT, the source code, interpretation tree, and animation. The animation view is still empty, because the interpretation of the program is not yet started. The interpretation tree is zoomed out in order to show the complete tree representing the semantics of the specification given.

Figure 6.12 shows the visualization of the contents in the first bag. Each lot inside the bag appear described by the type of fabrication — represented by the three images in each row in the list of lots — and the number of pieces of cloth that compose the lot — represented by the label below each type of fabrication. The next step on the animation is to calculate the number of bags that have already been processed.

After *closing* each bag, a mark, signaling the number of bags already processed, appears in the upper-right corner of the *Alma*²'s animation view, as can be seen in Figure 6.13. This visualization is composed of the image representing the concept *Bag*, presented before in Table 6.2, and a label printing the value of bags processed.

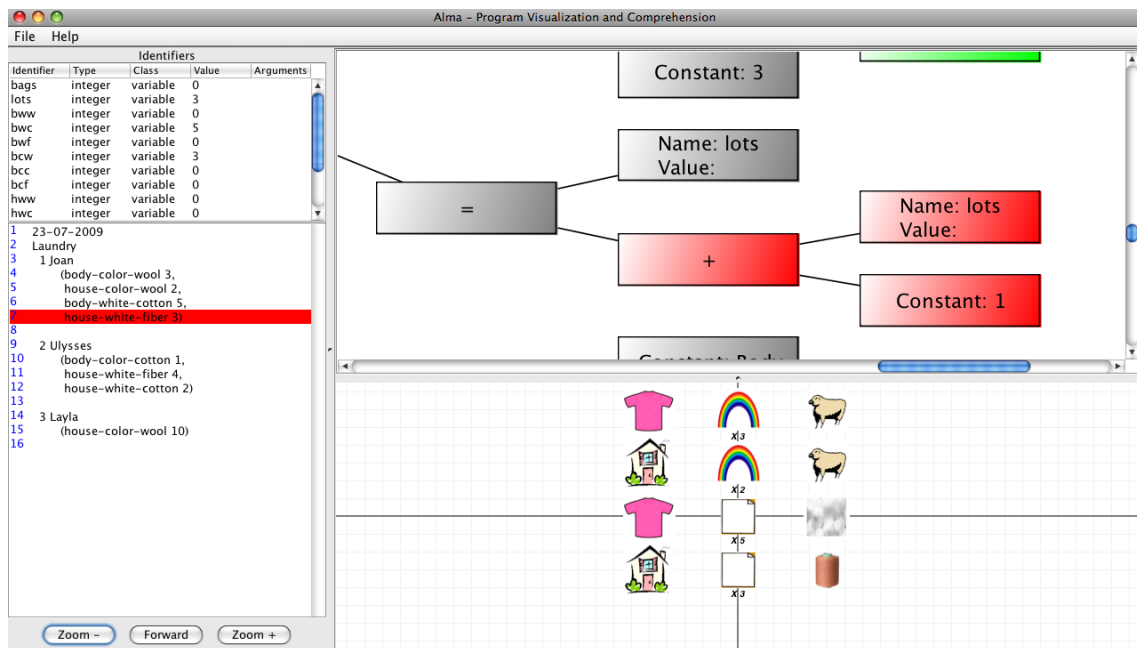


Figure 6.12: Visualization of the contents in the first bag

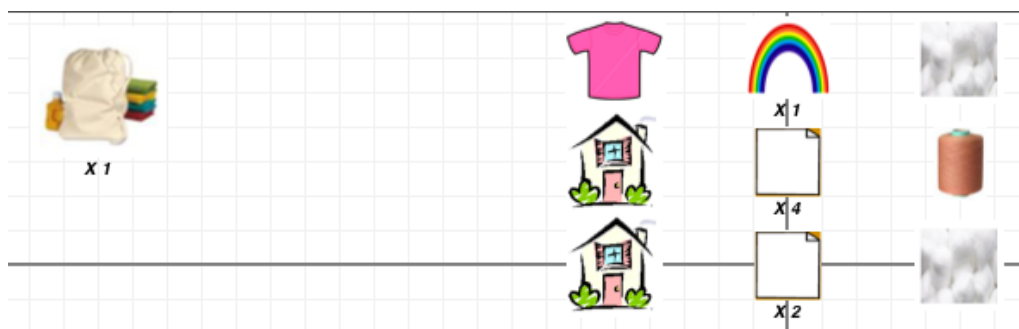


Figure 6.13: Problem Domain Visualization of the second Bag

This composed item of the visualization is always drawn, even when processing the contents of other bags, as is the case of Figure 6.13.

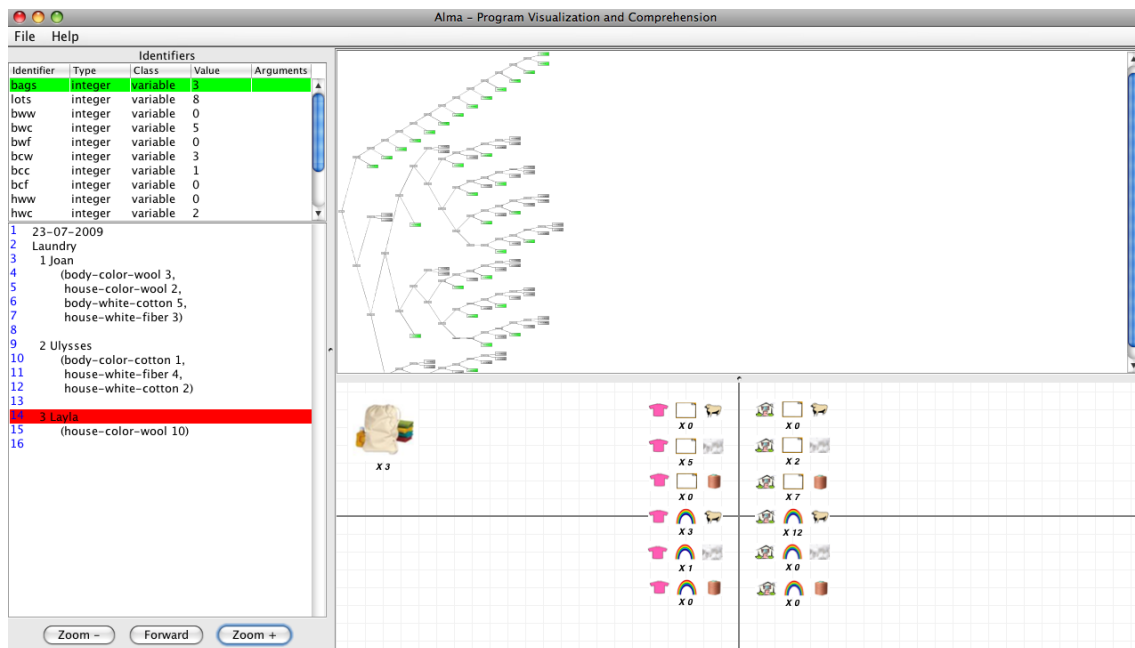


Figure 6.14: Final Visualization of the *Lavanda Language* specification

In the end of the animation, is shown the number of bags processed and the summary table for the lots and type of clothes. Figure 6.14 depicts exactly the final moment in the visualization of the program. From this summary visualization is easy to retrieve the number of bags and clients, and more important is the possibility to get the number of each type of clothes that were sent to wash.

With this case study, is shown how a declarative DSL can be used in *DsPCTools*, in order to offer its comprehension. Using *Alma*², is possible to have a synchronized visualization of both program and problem domains. Although this is not a complex problem domain, the synchronization of the operational and behavioral views is important to understand what, at the program level, leads to a behavior of the concepts, at the problem level.

6.3 Summary

In this chapter were presented two case studies. The first case study was based on an imperative DSL: *Karel's Language*. The second was conceived using a declarative DSL: *Lavanda Language*. Because of the paradigm underlying each language, several facilities or difficulties were found to conceive the visualization and animation of the programs, but in both cases, the goals were achieved in a systematic way.

When dealing with an imperative language that has a well bounded application domain, is easy to identify one or several objects that can be said to be *controlled* by the language. In the case of *Karel's Language*, a robot is such a controlled object.

Then, the important concepts at the program and problem levels are translated by instructions used to command the object in the desired way. An imperative language also defines a concrete operational semantics, what eases the process of constructing an abstract representation of the internal behavior of the program. This, allied to the fact that is easy to determine the connections between the problem and program domains using a DSL, makes possible the synchronization of two views that show the internal and external behavior of the program.

On the other hand, when dealing with a declarative language, it is not so easy to find a concrete object that acts as the *controlled* object, because, in principle, these language were not conceived to control objects. However, as the domain is always well bounded, it is easy to find one or several objects that may play a leading role in the entire domain. Another difficulty associated with these languages is that they have not a precise semantics associated. But the objective on processing these languages is to retrieve results, and not to lead an object to perform an activity. Notice that to retrieve a result, there are needed operations; then, the absence of operational semantics in the language is not a problem anymore. Connecting the concepts at both domains (that differently from the imperative languages, represent words used to describe the specifications) a synchronized visualization of what happens at these domains, is possible to achieve.

Chapter 7

Approach Assessment

The true method of knowledge is experiment.

William Blake

Throughout this dissertation, an approach for DSLP comprehension was conceived and then concretized in a new `DsPCTool`. Both these tasks — conceiving and concretizing — were based on existent approaches and tools crafted to cope with GPLs. The concrete outcomes, the approach (`DapAST`) and the the `DsPCTool` (`Alma2`), were the main resources to demystify two doubts raised in Section 1.2, which originate two theorems. The first theorem (Theorem 2 presented in Chapter 4) says that program comprehension techniques usually tailored to cope with GPLs can also be applied to DSLs, regarding some minor changes. The second theorem (Theorem 3, presented in Chapter 5) assesses the possibility of improving existent `PCTools`, to enhance the understanding of DSLs, by taking advantage of problem domain visualizations.

With the theorems formulated before and the tool developed to assess them, it was objective the formulation of Theorem 1 which, in fact, corresponds to the main goal of this master thesis. Theorem 1, presented in Section 1.3 says, in a very succinct way, that the comprehension of a program is easier when synchronized visualizations of the program and problem domains are provided. But this theorem must be proved correct, so its formulation is valid. Figure 7.1 describes the simple method to prove it. The main resources for such a proof are the tool and the user experiments, since it must confront the understanding of a program by the Program Reader (`PR`) and the tool used to achieve it. The tool must meet the synchronization of domains, required in the formulation of the theorem. Concerning this requirement, in Chapter 6, two case studies were presented, from where it was shown the synchronization of the problem and program domains provided by `Alma2`. So, this requirement is completely assured. The second resource, the experiment, should offer good results, in order to be drawn conclusions, regarding the theorem.

Given this, the content of this chapter reports the three main phases of the experiment involving `Alma2` and several `PRs`. These three phases are the preparation of the experiment, its execution and, finally, the analysis of the results. Section 7.1 describes the preparation of the experiment according to previous experiences and published works on the experiments area. Section 7.2 describes the execution of

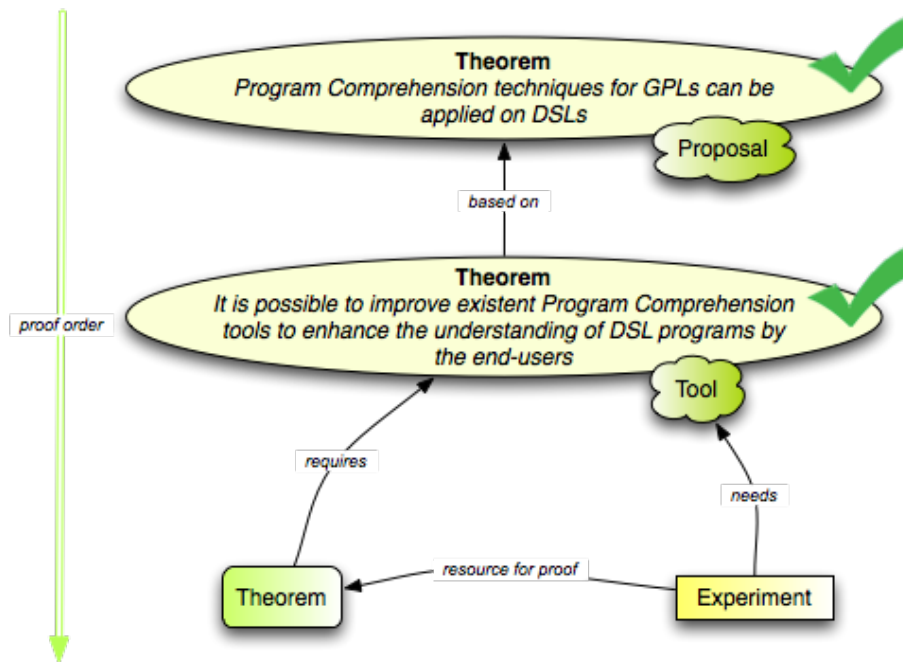


Figure 7.1: Theorem Proof Schema

the experiment. Some online observations are discussed, in order to advance some conclusions. Finally, Section 7.3, presents the statistical results of the experiment. With these results, conclusions are drawn.

7.1 Experiment Preparation

An empirical study, also called experiment, can be prepared and executed following several ways, since there is not a standard. However there are common steps from experiment to experiment. This section presents the team know-how on performing similar empirical studies, discusses some related work on the preparation of experiments, and comes out with concrete results from these studies.

7.1.1 Team Know-how

In the context of DSLpc project, the bilateral cooperation research inside which this master thesis is placed, some empirical studies were proposed and performed to analyze properties of DSLs comparing with GPLs, with respect to three perspectives: learning the language, perceiving the programs and evolving the program [106, 124].

The experiments were supported by two questionnaires for each pair of languages to be compared; the questionnaires were carefully designed and structured.

The experiment involved several students (programmers) from Portugal and Slovenia with different background and expertise on the usage of programming languages. These students were submitted to several of the questionnaires discussed

above with eleven questions (each) evenly distributed by all the three identified perspectives.

During the preparation of these experiments, great importance was devoted to the characterization of the programmers' expertise on programming aspects related to the domain of the experience. Then, several threats to the validity of the results were analyzed to be taken into account when discussing the results. All the process was very well structured, in order to relate, as much as possible, each question with the objective of the experience.

The results of these experiments were conclusive and important to the programming languages community, meaning that experience on this area of empirical studies, was obtained. So, DSLpc team has background on realizing these experiments, what was crucial to prepare the one concerning this master thesis.

7.1.2 Related Work

There are many written reports on research and outcomes of experimental studies [180, 166, 11, 211, 118, 192] performed by experts, from where knowledge can be retrieved to improve the background on this area of experimenting.

Despite the significative number of works cited, there are two major and recent contributions representing the trends and the standardization of the experiments' preparation. The first work was made by *Di Penta et al.* [55]. The authors identify a set of important characteristics that an empirical study should hold to be considered acceptable. These characteristics are on the way of the standardization, because, as the authors claim in their work, "*the standardization of an experiment format would facilitate the replication of experiments and the aggregation of data from multiple experiments*". From the six steps described on their work, three of them seem to be really important: the definition of variables to measure during the experiment, the identification of an appropriate subject population, and finally, the precise description of the system or task under evaluation.

Another important outcome from *Di Penta et al.*'s work is that it must be stated the relation between what is to evaluate and what is used to evaluate, that is the, the relation between the knowledge and the question; it is very important that this remains clear when analyzing the results of the experiment,.

The second work was undertaken by *Cornelissen et al.* [42]. This work follows the main guidelines suggested by *Di Penta*, skipping ones but adding others. In this work, the authors prepare, execute and analyze an experiment on the added value of trace visualization tools for program comprehension. In what concerns the preparation of the experiment, they start by identifying the variables (time and correctness) that will be measured during the experiment; then, they identify the research questions and their initial hypotheses; next, prepare the tasks, and finally identify the subjects of the experiment. They also identify some threats that may compromise the validity of the study.

The latter work follows a complete and interesting preparation of an experiment. As the intention on the present chapter is not to *reinventing the wheel*, blending the ideas coming from the background knowledge, and the steps followed in the latter

work, the experiment underlying this chapter is prepared. Next sections report such preparation.

7.1.3 Objective of the Experiment

Differently from what was done in the experiments reported in section 7.1.1, in the present experiment is objective to see if **Alma**² and its synchronized views of the program and problem domain are capable of giving the user a better experience on understanding a program.

The objective of reaching the understanding is twofold: on the one hand, when a program is understood, the user obtains a certain knowledge about its functionalities and purposes; and on the other hand, when a program is understood, the user is able to modify it in the correct location.

So, to test whether the synchronized views of the program and problem domain in **Alma**² ease the understanding process (in both directions described before), the experiment must compare the usage of **Alma**, which only concerns the program domain, with the usage of **Alma**², which concerns both domains, and present synchronized views of their animation.

The following paragraphs identify the variables, the research questions and the hypotheses of the experiment. These points agree with the strategy of preparing an experiment proposed by the work of *Cornelissen et al.* [42].

Variables. During the experiment it is important to measure two values: the time spent on each question, and the correctness of the questions, as these are the usual directions of an experiment. But for this study, a variable to measure the importance of the incremented value of **Alma**² is essential. So, in this experiment, three variables are taken into account: *time*, *correctness* and *importance*.

Research Questions. In fact, there is only one research question on these experiment: *Does the synchronized visualization of the program and problem domains help users on understanding DSLPs?*

But this question can be divided into three others:

1. *Does **Alma**² decreases the time spent on program comprehension tasks?*
2. *Does **Alma**² augment the correctness of the comprehension tasks?*
3. *Is the problem domain visualization and the views synchronization of **Alma**² important to understand programs?*

A third question was inserted in order to take advantage of the variable *time*. However, to prove Theorem 1, the time spent in each program comprehension task is not relevant.

Hypotheses. The hypotheses of the experiment are associated with the research questions. They are used to confront with the results of the experiment, and then to answer the research questions.

- **H1** — `Alma2` has influence in the time spent on the comprehension tasks, decreasing it.
- **H2** — `Alma2` has influence in the correctness of the comprehension tasks, augmenting it;
- **H3** — The problem domain visualizations and their synchronization with the program domain visualizations offered by `Alma2` are important to the process of comprehension;

7.1.4 Questionnaire Design

Regarding the objective defined in the previous section, the tasks of the experiment must focus on the analysis of the results of the program comprehension process. That is, the participants of the experiment must use `Alma` and `Alma2` to achieve the comprehension of a program, and then, with the knowledge gathered, answer some questions to assess two directions: (*i*) if the program functionalities were perceived, and (*ii*) if the participant is able to evolve the program.

Both directions must be assessed in `Alma` and in `Alma2`. The use of the former system would provide base results that will be confronted against those coming from the usage of the latter system. This confrontation of results answers the research questions drawn in the previous section.

So, the tasks of the experiment must address details on perceiving and evolving programs, but they must also be directed to answer the research questions. Attention should be paid that the tasks must not be biased to give advantage to any of the used tools; otherwise it would distort the final results and conclusions.

Each task of the questionnaire must provide data to fill the three identified variables, which analysis will answer the research questions and enable the drawing of conclusions. The following list describes how each task is composed to gather such information.

- To capture information for the *correctness* variable, a question about a `DSL` is asked. The question addresses either the program purposes or its modification. The correctness of the answer will be used to measure the *correctness* variable.
- To gather information for the *time* variable, the participant is required to stamp the time at the begin and at the end of the task's execution. Once the tasks are executed on a computer, but the answers are written in paper, the time spent on its execution can not be calculated in an automatic way.
- Finally, to gather information for the *importance* variable, the participant is asked to score the importance of the problem domain visualization and its synchronization with the program domain visualization, in the resolution of the task. Notice that this is a subjective answer for each participant, but the overall result reflects the general opinion.

Figure 7.2 depicts the main structure of a task in the questionnaire. Notice however that, from task to task, its content and format may vary a little, depending on the type of the question.

Task 1 ⌚ _____ (Please, stamp the start time)

Q: *This space is for question text*

T: *This space is for tips on Alma and Alma² execution*

A: *This space is for multiple choice or open answers*

Please, classify the help that problem domain animation synchronized with the program domain visualization gave you on the resolution of this task:

★ ____ (Regard classification table above)

⌚ _____ (Please, stamp the finish time)

Figure 7.2: Main Structure for the Experiment Tasks

Now that the structure of the questionnaire is conceived, the next step is to define the questions. But to proceed on this step, it is needed to know, first, which DSLs would be used. In Chapter 6, two languages (*Karel's Language* and *Lavanda Language*) were presented, described and its use on *Alma²* was testified. Their different characteristics (imperative and declarative, respectively) are a good aspect to take into account, so the answers to the research questions and to Theorem 1 are more credible and consistent.

It was decided that the questionnaire would be eight tasks long, evenly divided into two groups: one group for perceiving questions, and another for evolving questions. Two perceiving questions and two evolving questions are associated to each language. Moreover, each one of these questions are to be answered, alternately, using *Alma* and *Alma²*. Figure 7.3 explains, by means of a diagram, the scheme of the questions distribution.

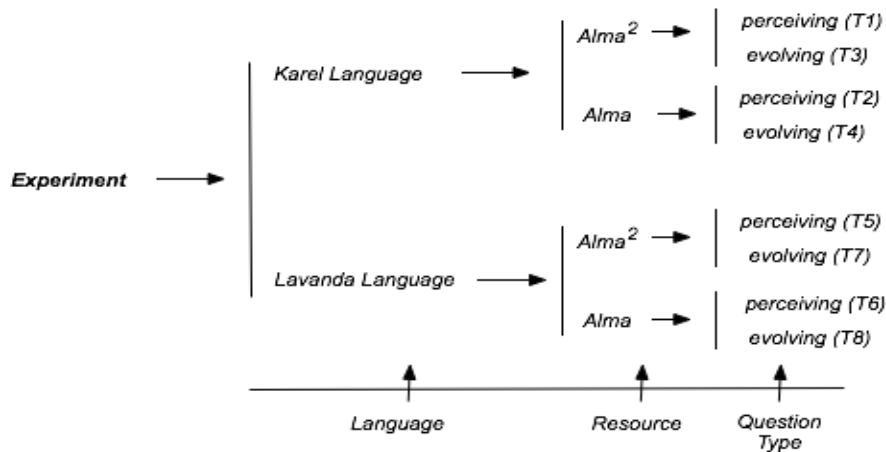


Figure 7.3: Distribution of Questions

The complete questionnaires are presented in Appendix B, from Figure B.2

to B.6. In this section, only the abstract type of the task is shown in the following list:

- **T1** and **T2** — Understand the program to identify functions producing a given output;
- **T3** — Change (adding or removing parts of) a program to cope with new requirements;
- **T4** — Refactoring a program, cleaning unnecessary instructions;
- **T5** and **T6** — Understand the program to calculate part of the final output;
- **T7** — Remove part of the program to achieve a new output and
- **T8** — Add code to the program to achieve a new output.

For this experiment, it is used a type of questions similar to that used in the experiments reported in [106, 124]. The reason to do that, is supported by the fact that in the present experiment it is wanted to conclude about the perception and the evolution of DSLs using a tool for helping the comprehension, while in the past experiment it was objective to study the comprehension of DSLs over the learning, perceiving and evolving directions, without tool support.

7.1.5 Participants Identification

In order to accomplish the experiment, 20 (twenty) persons from the area of informatics were invited. This sample was composed of three females and seventeen males, where one (5%) was *B.Sc.* student, sixteen (80%) were *M.Sc.* students and three (15%) were *Ph.D.* students. The undergraduate student was starting his final year of the *B.Sc.* degree; about 20% of the master students were starting their final year, and the remainder were finishing their master's thesis; all the *Ph.D.* students were starting their third year of studies in that degree. Their areas of specialization were varied and addressed fields like computer networks, distributed computing, bioinformatics, data warehousing, computer graphics and computer interaction.

Before starting to answer the questionnaire, the participants were requested to fill in a simple form to classify their background in performing tasks related with the experiment¹. The information that outcomes from the analysis of the forms' data is important to elaborate an idea about the skills of the participants on (i) analyzing programs to understand them; (ii) using program comprehension tools; (iii) using IDEs with debugging features for software development; (iv) using DSLs and (v) comprehending DSLPs.

Figure 7.4 shows the results of the evaluation made by the participants about their background (history and expertise) on performing the tasks referred in the previous paragraph.

To each task the participants evaluated their history based on the following grades: 1 - Never; 2 - Rarely; 3 - Sometimes; 4 - Several times and 5 - Always; and

¹The participant identification form can be seen in Figure B.1, in Appendix B.

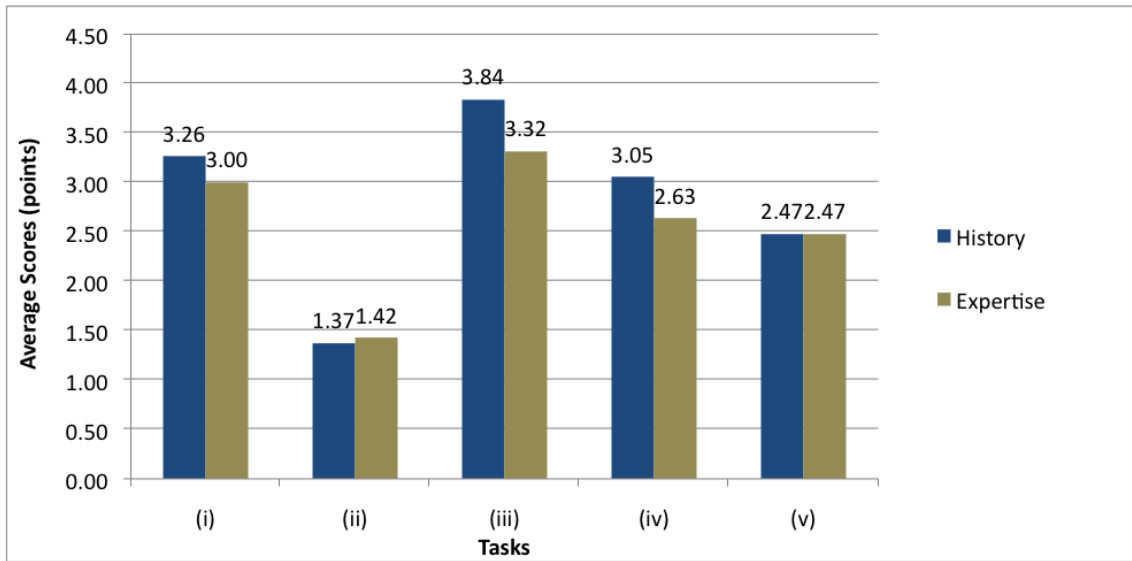


Figure 7.4: Comprehension Tasks — History and Expertise of the Participants

marked their expertise ruled by the following scores: 1 - Null; 2 - Low; 3 - Medium; 4 - High and 5 - Expert;

From the chart presented in Figure 7.4, it is possible to observe that in average, the participants never, or very rarely, used a program comprehension tool. However, they are very used to analyze and understanding programs. This may mean that they follow a non systematic and manual process of program comprehension. On the other hand, they often use IDEs with debugging features, what may mean that they use these tools to help on the comprehension process. Regarding DSLs, the participants are used to use them, but rarely go through a process of comprehending programs written in such languages.

Given this analysis, and knowing that none of the participants have ever had a contact neither with Alma nor with Alma², the results of the experiment will not be influenced by previous bad habits.

7.2 Experiment Execution

The experiment was, initially, thought to be executed in a single session. But due to room space factors and incompatibility of dates, there were scheduled two sessions. Thus, the participants were divided into two groups: in the first session participated 7 persons, and in the second participated 13.

In the overall, each session long about 2 hours. The first 15 minutes of each session was spent to give a small presentation about the experiment and the resources involved. In that presentation each one of the DSLs was briefly explained, presenting their domain and syntax, by means of simple examples. Also, that presentation was used to introduce Alma and Alma², pointing their main differences and their purposes. The remaining time was used to answer the questionnaire.

In order to ease all the questionnaire process, a package was prepared with a

ready-to-use version of `Alma` and `Alma`². Besides it, multi-platform scripts were delivered to execute the systems, hiding some `Java` intricacies. At last, along with these two resources, was delivered a third one, containing the source code of the eight different DSLPs used in each task of the questionnaire.

During the experiment, participants solicited help to clarify the real intent of some questions. Care was taken in the help given, in order not to bias the answer of the respective question. But, in the overall, the participants understood what was asked and filled out the questionnaire without difficulties, adopting some interesting strategies which are worthy to take note and analyze to have a reasoned discussion about the experiment results, in Section 7.3.

As said earlier, the participants used two different systems to answer the questions. As the main difference from `Alma` to `Alma`², is the visualization of the problem domain, it was interesting to see the different strategies adopted by the participants to answer the questions when using both systems. The main observations respecting to this aspect were that when questions have to be answered using `Alma`, principally those concerning *Karel's Language*, the major part of the participants used a draft sheet to draw *world* objects similar to the visualizations offered by `Alma`². So this led to the following conclusion: in every moment, the participants felt the necessity of synchronizing the operation in the source code with the behavior of the controlled objects, by means of problem domain visualizations.

Maybe this conclusion is a little bit biased to the desired outcome. But, together with the analysis of results presented in Section 7.3, will be possible to conclude whether it is tampered or not, as well as drawing new conclusions.

7.3 Final Results

In this section are shown the results of the experiment and a possible interpretation of the output values (Section 7.3.1). In order to have a more reliable discussion about the results, some independent variables, and facts that may have interfered in the results, are identified (Section 7.3.2).

Is worthwhile to say that after the two sessions of the experiment, five experts on language engineering (two from University of Minho, and three from University of Maribor) were asked to go through the questionnaires in order to provide some comments on the questions and on the observed results. So, the discussion of the results below have also the important point of view of these experts.

7.3.1 Results and Discussion

In this part of the section, the results of the experiment are presented by means of charts and tables. For each of the variables defined before (*time*, *correctness* and *importance*) these charts and tables are discussed, taking into account the hypotheses **H1**, **H2** and **H3** defined in Section 7.1.

The *time* variable is scrutinized in first place, because the results of this analysis are important to discuss the results of the second variable being analyzed: the *correctness*. Finally, the results about the *importance* variable are revealed, taking into account the discussions about the *time* and *correctness* variables.

Time

The hypothesis **H1** raised before states that, using **Alma²**, the time spent on performing comprehension tasks decreases. During this section, the hypothesis is studied according to the results obtained.

In Table 7.1 and Figure 7.5 (left) are presented the results of the time spent for each question. The labels Q1 to Q8 are the questions inside the eight tasks presented before and concretely identified through Figures B.2 to B.6.

Table 7.1: Average of Time Spent in Solving the Questions (minutes)

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Average	9,10	8,05	15,35	17,55	3,25	2,75	5,00	3,05
Stdev.	4,70	5,19	5,82	11,59	3,23	2,49	2,99	1,73

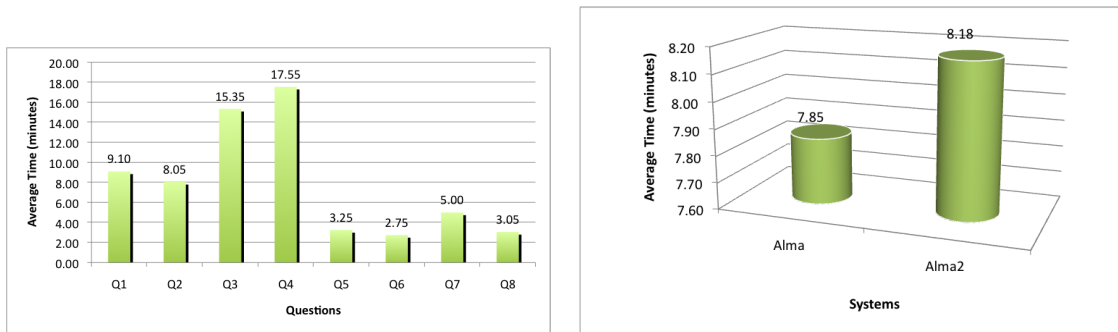


Figure 7.5: Time Results per Question (left) and per System (right)

The odd questions (Q1, Q3, Q5 and Q7) were solved with **Alma²**, and the even ones (Q2, Q4, Q6 and Q8) with **Alma**. As can be seen, Q4, solved with **Alma**, was the question where the participants, in average, spent more time. However, in general, they took more time to answering the questions concerning the use of **Alma²**. Figure 7.5 (right) corroborates it by presenting the bar chart of the comparison between the time spent when answering questions using either **Alma** (7,85 minutes) or **Alma²** (8,18 minutes)².

Despite of not being a great difference in terms of minutes, only about 20 seconds, this is a difference that should be taken into account: *using Alma², the participants spent more 3,98% of their time to solve traditional program comprehension tasks.*

An interesting result that can be mentioned is the difference of time spent when answering questions concerning the type of the language. From the chart in Figure 7.5 (left), it is easy to see that from Q5 to Q8, the participants spent less time than in the first four. Questions 5 to 8 are questions about programs written in *Lavanda Language*, which is a declarative language. This leads to draw the conclusion

²Notice that 7,85 minutes correspond to 7 minutes and 51 seconds, and 8,18 minutes correspond to approximately 8 minutes and 11 seconds.

that performing traditional program comprehension tasks in declarative programs is faster than in imperative ones.

As a last outcome from the time analysis, shown in Table 7.2, is that using Alma the questions concerning the perceiving of the program take less time than using Alma². But the same is not applicable in questions concerning the evolution of a program. In this case, the time spent using Alma² is slightly less than using Alma. Nonetheless, the participants spend twice the time performing evolution based program comprehension tasks, when comparing with their performance on solving perceiving based tasks.

Table 7.2: Average of Time Spent per Type of Question (minutes)

	PERCEIVE	EVOLVE
Alma	5,40	10,30
Alma ²	6,18	10,18
Average	5,79	10,24

So, in conclusion, **H1** is false. This means that the usage of Alma² affects the time on performing program comprehension tasks in the extent that it is needed more to perform the tasks.

The conclusion drawn in the previous paragraph is not the expected one. In fact, using Alma² should have decreased the time needed to perform traditional PC tasks. However, there are some factors contributing to the explanation of these results, that were never thought as possible *threats* to the time spent on the execution of the proposed PC tasks.

The first point is that having an extra view in the layout of Alma² may affect, negatively, the time. In Alma the participants only have three sub-windows to look to; in Alma² they have four, and looking to these four sub-windows may cause an additional overhead, increasing the time spent to perform the tasks. Also related with this point is that the new view may catch the eye of the user, so, on the one hand, they spend extra time *mesmerized* with the problem domain visualizations or, on the other hand, they feel more motivated to create a more accurate mental model, which is a task that takes its time.

The second point is concerned with the difficulty of the questions. When a question is more difficult, the time consumed to answer it is always higher than when it is easier. Although nothing points into that direction, the participants may have felt more difficulties on solving the Alma² questions, then the time spent had increased.

Correctness

The hypothesis **H2** claims that, using Alma², the correctness of answers given to questions concerning traditional PC tasks is augmented. According to the study of the results obtained, and the results of the time, previously hashed, this hypothesis is discussed during the remaining of this section.

Table 7.3 and the bar chart in Figure 7.6 (left) present the average of correctness of the answers given to the eight questions of the questionnaire. Each answer was classified with a value from 0% to 100%, depending on their correctness. So, the average is also presented in terms of percentage.

Table 7.3: Average Correctness of the Questions (%)

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Average	95,00	62,50	35,25	58,10	80,00	55,00	95,00	100,00
Stdev.	22,36	22,21	31,10	38,81	41,04	51,04	22,36	0,00

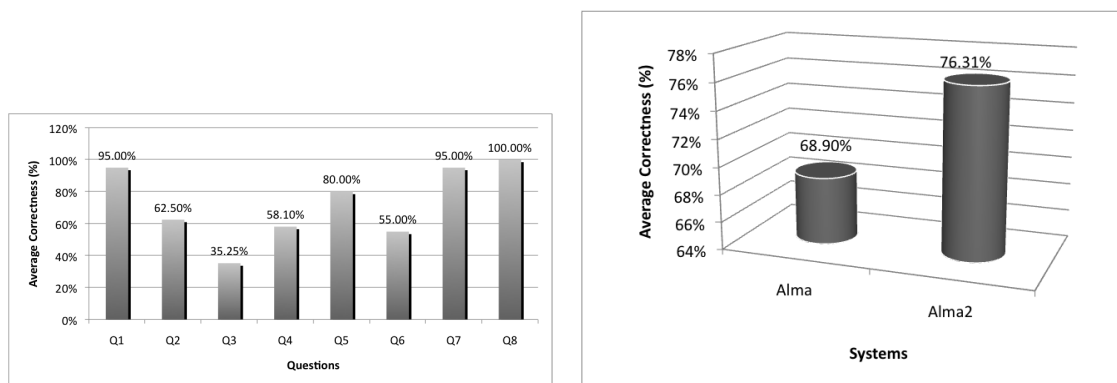


Figure 7.6: Correctness Results per Question (left) and per System (right)

In average, the answers are positive and over the 50% barrier. Only Q3, which was solved using Alma² to analyze the program, had an average of correctness under 50%. On the other hand, Q8, answered using Alma, was guessed by every participants.

However, in general, the participants scored higher in the their answers, when they use Alma² as tool to analyze and explore the program. In Alma the participants scored, in average, 68,90%, and in Alma² they scored 76,31% (see Figure 7.6 (right)). This means that: *using Alma², the participants answer about 10% more correctly the questions, than using Alma.*

From this study, an important result, concerning the type of the language, can be pointed out. Figure 7.6 shows that it is more likely to have good scores when trying to perceive and evolve declarative languages than when doing the same with imperative ones. Just notice the correctness values from the bars respecting Q5 to Q8.

Another outcome concerns the type of the PC task and the comparison between the systems used. Table 7.4 shows such a comparison. It is interesting to see that when using Alma² to perceive a program the correctness of the answers is huge, comparing with Alma. On the other hand, when the questions concerns the evolution of programs, Alma² get worst correctness values than Alma. Nonetheless, in terms of correctness, there is not a great difference between the two types of PC tasks.

Table 7.4: Average of Correctness per Type of Question (%)

	PERCEIVE	EVOLVE
Alma	58,75	79,05
Alma ²	87,50	65,13
Average	73,13	72,09

Because of the three last results, but with more focus on the first one, it is easy to conclude that hypothesis **H2** is true: the usage of Alma² affects the correctness percentage of the answers, augmenting them.

This conclusion confirms exactly what was predicted. However, the results shown do not enhance always the use of Alma². For instance, the percentage of correctness presented for the question 3, is the lowest value in the chart. Several factors could have affected the obtained result: the kind of question is one of them.

Question 3 (defined in Figure B.3 under Task 3) asks the participants to change a *Karel's Language* program's code. As the participants did not know neither the language's syntax nor its semantics, writing such a code, with a score of 100% respecting the correctness, could have been a cumbersome task. In the brief talk, given in the beginning of the experiment sessions, the language was exposed, but maybe it was not sufficient. Still in this case, it is likely that the participants did not use Alma² to confirm whether their solution was correct or not, and perhaps have used the system only to perceive the base program.

The other questions concerning the use of Alma² present higher correctness values on their answers. Much assumptions and theories may be created to discuss the truthfulness of the results. An obvious assumption is that the problem domain visualizations could have helped a lot on performing the tasks. The fact of being provided higher level visualizations leads the user to build a more concrete mental model of the program under study. A second assumption is the continuous synchronization of the several views provided by Alma². The problem and program domain visualizations when synchronized with each other may have helped the participant to adopt a top-down, bottom-up or even to blend both strategies, leading to a better comprehension of the program.

An intermediate conclusion, from this analysis and the analysis made to the *time* variable is that despite the participants spent more time answering the questions using Alma² to analyze and explore the programs, they achieve better results concerning the correctness of the answers. A possible reason for this result may lie on the fact that the participants, relying on problem domain visualizations, tend to examine, with more caution, the programs, which requires more time spent. A thorough analysis leads to a greater comprehension, then, answering the proposed questions is easier and more correct.

Importance

The hypothesis **H3** states that the problem domain visualizations and their synchronization with the program domain visualizations offered by **Alma**² are important to the process of comprehending a program. During the experiment, the participants were asked to score the importance of these aspects on performing the proposed task, based on the following grades: 1 - None; 2 - Negligible; 3 - Medium; 4 - High; 5 - Excellent. The results based on the participants' evaluation are used, through this section, to discuss the hypothesis.

In Table 7.5 and the bar chart in Figure 7.7 (left) is presented the average of the scores given by the participants to each one of the questions, concerning the help of the problem domain visualizations and the synchronization of visualizations. As **Alma** does not provide problem domain visualizations, the users were asked to classify the help that these aspects would have given to solve the question.

Table 7.5: Average Importance of Problem Domain Visualizations and Views Synchronizations per Question (score 1 to 5)

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Average	2,9	2,65	3,9	3,35	3,3	2,55	3,1	2,65
Stdev.	1,21	1,09	1,07	1,23	1,63	1,28	1,21	1,31

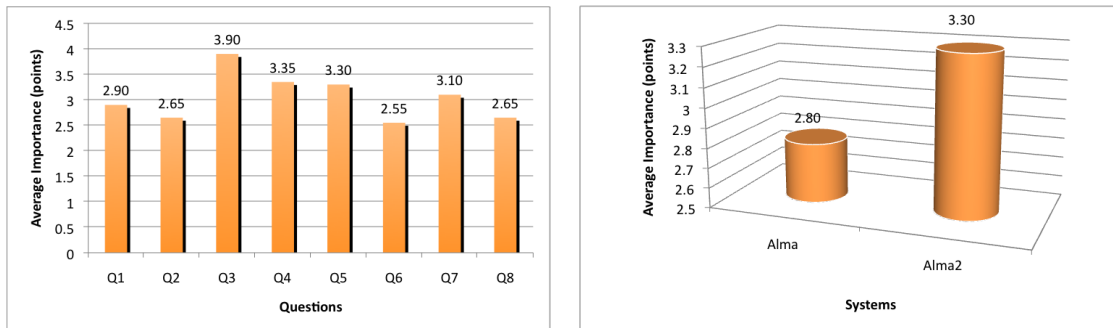


Figure 7.7: Importance Results per Question (left) and per System (right)

Concerning the even questions (Q2, Q4, Q6 and Q8), there occurred a problem which compromises all their scores. The participants, principally those that frequented the first experiment session, did not understand quite well the objective of this evaluation when using **Alma**. So, they scored the help offered by the visualizations that **Alma** provides, and scored it low, instead of scoring whether the problem domain visualizations would or would not help. It is a true fact, because many of the participants, after the session, claimed that if the problem domain visualizations were available in the even questions, it would be a great help. However, in the second session, even after explaining better the propose of these scores in the brief presentation, a small part of the participants have done the same error.

As it is obvious, contemplating such wrong scores to produce the final results, the average is lowered. But as it is not known who were the participants committing this error (the questionnaires were anonymous), it was preferable to keep all the scores contributing to the final average. So, in order to not produce a tampered discussion about the even questions scores, they will not be taken into account.

Lets concentrate then, in the scores of the odd questions (Q1, Q3, Q5 and Q7). Unfortunately these scores are not as higher as expected. However they are all close to each other. Q3 was the question scoring higher. Its score of 3,9 can be rounded to the grade 4, meaning that the help of the problem domain visualizations on that question was *High*. However in average the problem domain visualizations were given the third grade of importance (3,30 — see the bar concerning **Alma**² in Figure 7.7 (right)), meaning that their help is considerable but is not high or extremely important.

Since that is a subjective classification, assumptions or theories to explain these results are not produced.

In conclusion, **H3** is true. The problem domain visualizations offered by **Alma**² are indeed important to comprehend a program. They are not extremely important in the extent that a program can not be comprehended if they are absent. But they hold a considerable importance, not only because of the results of the *importance* variable, but also because of the discussion done about the *correctness* variable.

7.3.2 Threats to Validity

There are many threats that can compromise the results of an empirical experiment. Human threats like the uneven personality of the participants in the experiment, as well as their competence to do the job, or the lack of motivation to perform it. Circumstances threats like the difference on the computers where the experiment is being performed or the noise at the time of the experiment. Author threats like the possibility of the questions being directed to a convenient answer or the imposed difficulty on the questions, and so forth. For a list of threats complementing those given, the reader may want to read [42].

All these threats are valid to the experiment being reported in this chapter. However it is important to list some specific threats that were caught during this experiment. Concerning the participants' characteristics, the following are identified:

- The group of participants was not as heterogenous (there participated more males than females), so, the results may not correspond to the reality of the society;
- The number of participants may have been too low. In this context, the results may as well be not realistic;
- The majority of the participants have background on informatics, and are not used to use **PCTools**. This way, they are accustomed to retrieve information from code, meaning that they may have discarded the visualizations in most of their actions.

- The participants did not understand the propose of questions. This happened when they classified the problem domain visualizations and the views synchronization on *Alma*. The classification they give have compromised an important part of the questionnaire.

Concerning the systems, there are some independent variables that may have conditioned the results of the dependent variables (*time*, *correctness* and *importance*). The following can be identified:

- Four sub-windows in a single window. Although this is not a big problem, it has already been identified, during the discussion of the variable *time*, that the addition of a new view (comparing with *Alma*) may add also a small overhead on the time spent to changing the look from a view to another.
- Specific to analyze programs. As both *Alma* and *Alma*², were created to analyze the programs by exploring it through different views, an overhead on timing could have been added while changing between the context of exploring the program and evolving the same.

Finally, concerning the questionnaire, is possible to rise the following threats:

- The questions may have not comprised the same difficulty. This threat was raised by one of the experts involved in the *off-the-record* experiment. He claimed that some pairs of questions concerning the same type of PC task (perceive or evolve), had not the same difficulty. This means that some questions answered with *Alma*² could have been harder than some answered with *Alma*, and vice-versa.
- The questions could have been biased to achieve the desired results. This is always a possible threat to the validity of the experiment, since the questionnaires were prepared by humans.

These threats must be taken into account to discuss the results, having a more sober-minded perspective about them.

7.4 Summary — Theorem Validation

The objective of this chapter was to prove Theorem 1, which states that the comprehension of a program is easier when synchronized visualizations of the program and problem domains are provided. To prove it, an experiment was conducted, involving *Alma*, *Alma*², and a set of eight questions to retrieve results from the usage of these systems under traditional PC tasks. The experiment followed, as close as possible, the steps adopted by *Cornelissen et al.* in [42].

In this context, three variables (*time*, *correctness* and *importance*) were measured for both *Alma* and *Alma*². This way, a discussion of results could be made by assuming *Alma* results as the base, and comparing them with the results of *Alma*². A summary of these results is presented in Table 7.6. Each three rows of the table show the individual values for the three variables.

Table 7.6: Summary of the Experiment's Results

	MEAN	DIFF	MIN	MAX	STDEV.
TIME	minutes				
Alma	7,85		1	45	8,79
Alma ²	8,18	+3,98%	1	28	6,33
CORRECTNESS	%				
Alma	68,90%		0%	100%	37,96%
Alma ²	76,31%	+9,71%	0%	100%	38,54%
IMPORTANCE	score				
Alma	2,80		1	5	1,35
Alma ²	3,30	+15,15%	1	5	1,25

To help proving the theorem, one research question was raised and divided into three smaller ones.

The first research question asked if, using Alma², the time spent to perform PC tasks is decreased. By the analysis and discussion made before about hypothesis **H1**, and for the results presented in table 7.6, is easy to see that using Alma², the participants were 3,98% slower than when using Alma. The difference is not big, but it can not be neglected. So, the first research question has a negative answer.

The second research question asked if, using Alma², the correctness of the answers in PC tasks would increase. Again, relying on the discussion made for the hypothesis **H2** and the results in Table 7.6, is concluded that there is, indeed, an increased percentage of correct answers when using Alma². The difference between the systems is of 9,71%. This way, the second research question has a positive answer.

Concerning the third research question, it asked if, using Alma², the problem domain visualizations and the synchronization of views play an important role in the comprehension of programs. The analysis made for hypothesis **H3** is conclusive, because the participants classified these aspects of the system as having an average importance, and missed the problem domain visualizations when using Alma. The latter fact is not consistent with the results presented in Table 7.6, because the participants did not understand the purpose of the question, however, during conversation after the sessions they revealed the missing of such higher level visualizations.

The main research question of this experiment asked whether the synchronized visualization of the program and problem domains help users on understanding DSLPs. Given the results, the analysis and the answers of the sub-research questions, a positive answer can be given to this main question. Although it is spent more time performing PC tasks, the answers are more correct and the synchronized visualizations give a great help here.

As this research question can be seen as the question form of Theorem 1, and it has an affirmative answer based on the experiment results, the theorem may be considered proved, and valid.

Chapter 8

Conclusion

It's more fun to arrive a conclusion than to justify it.

Malcolm Forbes

This document is a master degree dissertation in the area of *Program Comprehension* (PC) applied to *Domain-Specific Languages* (DSL). Along the initial chapters it was defined a line to conduct the work to be done; the work already done by other researchers in this area was shown. The central chapters of this dissertation presented the formalization and of a theory, its concretization, and the prove of its influence when used on the real world.

In the present chapter, conclusions about the work done are drawn, and topics for future research are pointed. But first, a small summary about the contents of each chapter, is provided.

Chapter II — Domain Specific Languages In this chapter was presented a deep study on DSLs. A list of characteristics of these languages, advantages and disadvantages on their usage and on their development was provided. Also, techniques to support their creation were presented. From this study, one important conclusion can be drawn: using DSLs it is easier to infer their problem domain, and to create mappings between the concepts of both program and problem domains. This dues essentially to the fact of being high-level and abstract languages. This conclusion was important for the continuation of the work, because it led to the definition of a conducting line for the process of defining problem domain visualizations.

Chapter III — Program Comprehension In this chapter, an introduction to the topics within the PC field was given. Topics like software visualization and cognitive models were addressed to provide an overview on the basis of this thesis. Also, some *Program Comprehension Tools* (PCTools) were reviewed, by analyzing important aspects and features they provide or should provide. From this study, a main conclusion was drawn: in spite of existing many program comprehension tools, the majority of them only address a small set of *General-purpose Programming Languages* (GPL) and do not take advantage of the problem domain and consequently,

do not connect both program and problem domains to enhance the comprehension of programs.

A thorough study of tools for analysis of *Domain-Specific Language based Programs* (DSL_P) was also done. But again, they do not take advantage of the DSLs' characteristics and are mostly debuggers or similar tools and approaches.

Chapter IV — An Idea to Comprehend DSLs In this chapter, an approach for program comprehension to be applied to DSLs was designed. It was chosen a PCTool from those analyzed in the previous chapter, in order to use its program comprehension technique as basis for the construction of an improved approach for DSLs comprehension.

The new approach designed, called *Decorated with animation patterns Abstract Syntax Tree* ($D_{ap}AST$), is based on the abstract tree representation of the program, decorated with semantic and animation patterns. This tree is then traversed and its nodes are processed to create the animation and visualization of both program and problem domains.

The main conclusion of this chapter is that program comprehension techniques for GPLs can also be applied to DSLs, after appropriate changes.

Chapter V — Alma² In this chapter, a description about the implementation of Alma² was provided. Also the tool's architecture and internal structure was presented, and finally, a small tutorial on how to use the system, regarding two types of users, was given.

Alma², is the extension of Alma to produce the problem domain visualizations and the synchronization of visualizations at both domains, using the approach designed in the previous chapter. From the development of this extension an important conclusion was drawn: it is possible to improve existent PCTools to enhance the understanding of DSL programs. This improvement allows the interconnection of domains, and the synchronized visualization of both program and problem domains.

Chapter VI — Case Studies In this chapter, two case studies were defined, described and analyzed. The first one was focused on the use of Alma² to provide the synchronized visualizations of both domains of an imperative DSL. The second case followed the same strategy, but to cope with a declarative DSL. From the case studies, as conclusion, a systematic way to create the connections between the domains can be used, and this approach copes with different types of DSLs.

Chapter VII — Approach Assessment This chapter presented and described the contours of an experiment to assess whether the provision of synchronized visualizations of both program and problem domains are useful for the achievement of program comprehension. The results of this empirical experiment were displayed and analyzed with criticism, raising several conclusions about the approach conceived in this master thesis. The main conclusion drawn in that chapter is depicted in Figure 8.1, in the bottom balloon.

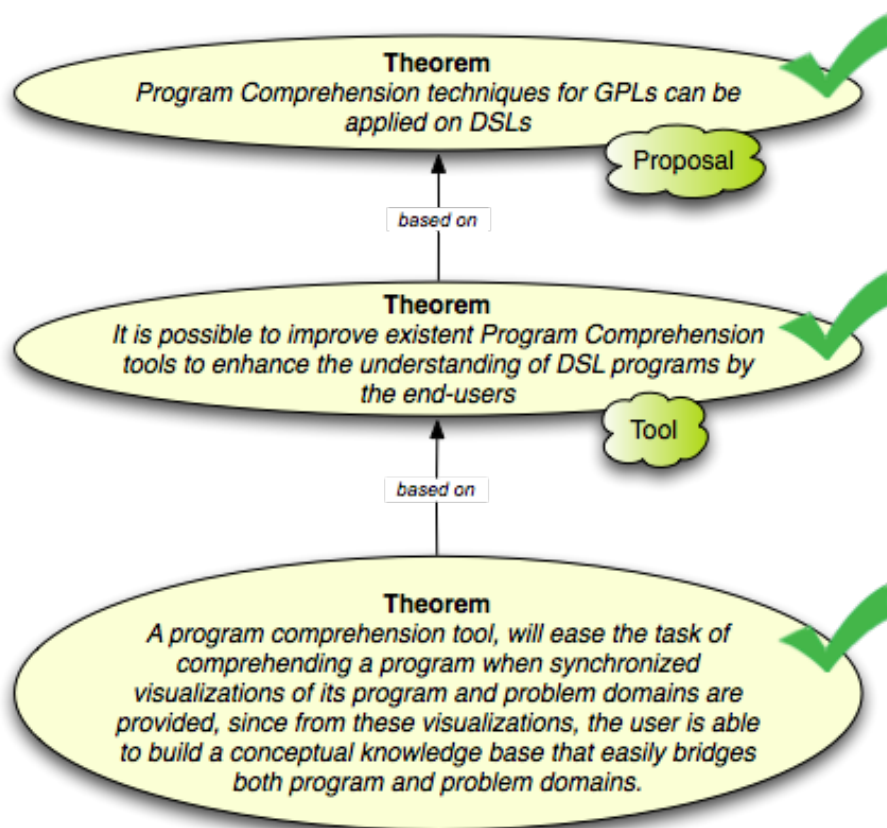


Figure 8.1: Final Theorem

Figure 8.1, summing up the main outcomes from the experimental assessment, shows that all the three objectives of this M.Sc. work were fully reached and supported.

8.1 Discussion and Conclusions

In this thesis, the main goal was to prove that program comprehension is easier when synchronized visualizations of the problem and program domains are provided (c.f. Theorem 1). To achieve this goal, an approach ($D_{ap}AST$) and a program comprehension tool ($Alma^2$) were created based on an existing `PCTool` for GPLs that did not provided the desired characteristics.

The design of the new approach for program comprehension is based on the well known advantages of the DSLs. The easiness on inferring their problem domain and on finding and creating mappings between both program and problem domains *opens doors* to the creation of higher level visualizations and to the synchronization of the visualizations at both domains. As this approach reuses, adapts, and extends another approach used for comprehending programs written in GPLs, it can be concluded (c.f. Theorem 2) that it is possible to find program comprehension techniques for GPLs, that can be applied to DSLs.

In the same way, the development of $Alma^2$, carried out the conclusion that existent `PCTools` can be improved to cope with DSLs, enhancing the comprehension of the programs written in such languages (c.f. Theorem 3), and also, it allowed to conclude that it is possible to bridge the conceptual gap between the program and the problem domains.

By providing the synchronized visualizations of these two domains, the answer to the question *what are the effects of the program in the entities of problem domain?* is given. The user of such a tool is able to create a thorough mental model of the program under analysis, no matter if following a top-down, a bottom-up or even a mixed strategy, but certainly creating a knowledge base following an *action-effect* paradigm.

In this thesis it was not intended to defend the use of DSLs. It was just advocated that `PCTools` can be adapted to easily comprehend DSLs, going along with their high level of abstraction, and taking advantage of the specific domain. Doing this with GPLs is hard, because it is not visible the problem domain, and it is very hard to infer it, or even impossible if not searched on the correct resources.

It is known that programs written in DSLs are easy to comprehend, or at least, are easier to comprehend than programs written in GPLs. However when the size of the programs is bigger, even in DSLs it can become cumbersome to understand. In this context, $Alma^2$ would give a great help. Also, $Alma^2$ has the particularity of being perfectly usable for teaching purposes, giving the students a consistent apprehension of the DSL.

$Alma^2$ is ready to cope with almost all DSL. But as the number of DSLs is great, there must be created a FE, which can be seen as a door to let the programs of a DSL in, and be interpreted using the $Alma^2$'s BE. Of course that an effort is required to set this FE up. It is necessary to have some expertise on compiler techniques and attribute grammars, and it is important to know how to articulate all the semantic

and animation patterns to extract the semantic representation of the program, and prepare the visualizations of the problem domain.

Concerning the initial sentence of the previous paragraph, it is just a speculation and a *belief*, based on the fact that: as long as it is possible to find operational behavior for the language, concepts at both problem and program levels, and images to represent the concepts, an *Alma*² FE for that DSL can be prepared. Bigger and more real DSLs like DOT and XAML are in the scope of *Alma*². Actually they could have been used to support the consistency of the tool in the case studies effectuated; however, as their output is already visual, it was decided to use DSLs with a more abstract output, in order to highlight the relation between the actions on the program and their effects on the problem domain.

But the use of bigger languages may compromise the scalability of the tool. The scalability problem, in this case, is directly related with the size of the language's application domain. If the application domain is big, the number of concepts embodied in the language would also be big, and the same would happen with the relations between the images used to represent these concepts. This is a very well known problem underlying every software visualization systems. This way, the approach followed by *Alma*², thoroughly explained in this dissertation, is prepared to offer partial visualizations of the program. In this approach, it is not required that every node of the D_{ap} AST have animation patterns associated; so, the expert user, when defining the visualizations may customize them, in order to focus only on a part of the programs. This way, the problem of visual scalability may be solved. However, tests concerning the scalability of the tool have not been made yet.

Concerning tests, only the effectiveness and usefulness of the synchronized visualizations of both domains were tested. In this experiment, interesting results were obtained. It was concluded that the time used to comprehend programs is higher when provided visualizations of both domains, but the correctness of the answers is also higher, and it is higher the bigger is the program. This dues to (or it is speculated to ought to) the fact that for small programs, program visualizations are sufficient, nonetheless people spend some time viewing the problem domain animations, to enrich their mental model. Concluding this, visualizations at program level are effective for small programs, and visualizations at problem level are more effective for big ones. The synchronization of both visualizations, eases the process of creating mental models, helping on bridging the gap between both domains. Furthermore, *Alma*² promotes the use of a top-down strategy to comprehend a program, in the extent that the user would start to see the final effects in the problem domain, identifying the several steps of the program's execution in order to reach the piece of code responsible for that action. However it does not discards the possibility of using other strategies, but always reaching an action-effect comprehension of the program.

8.2 Future Work

All the objectives proposed in the first chapter of this dissertation were achieved. Some rough parts of the approach conceived and the prototype built can always be sharpened. For instance, the interaction with the tool can be improved for a better

user experience and new features can be provided, regarding the users needs.

Concerning possible features missing in the tool, they can be provided regarding the results of usability tests not yet performed, but to be done in the future. It is important to left clear that the usability tests have nothing to do with the approach assessment experiment made during this master thesis.

During discussions and principally after exposition of the theme of this thesis in international conferences [143, 140, 142], the comments from the community were very interesting, raising important questions which were envisaged as further research topics. Some of them are listed below:

- **AlmaLAB.** This topic is concerned with the extension of the Alma² interface to define a more customizable one, regarding the possibility of hiding/showing some of the four fixed views provided by the actual tool. Such topic was raised after discussing the results of the experiment, in the extent that would be interesting to repeat the experiments but this time depriving the user from looking to all the visualizations.
- **Automatic mapping of domains.** This is an interesting research topic, because at the moment, the process of creating the visualizations for a given DSL is completely manual, requiring expertise on compiler techniques and attribute grammars.
- **Dynamic Visualization.** This topic concerns with allowing the edition of code inside Alma², rather than just reading it, and with the possibility of creating an interactive visualization of the problem domain concepts, always synchronized with the program domain concepts. The latter means that the user would able to give a *run-time* behavior to the objects of the program domain, according to the semantic operations available.

These are only some of the topics that open new perspectives to Alma² usage and research.

Bibliography

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.
- [2] M. Anlauff, R. Chaussee, P. W. Kutter, and A. Pierantonio. Domain specific languages in software engineering, 1998.
- [3] M. Anlauff, P. W. Kutter, and A. Pierantonio. Montages/gem-mex: A meta visual programming generator. In *VL '98: Proceedings of the IEEE Symposium on Visual Languages*, pages 304–305, Washington, DC, USA, 1998. IEEE Computer Society.
- [4] Fumihiko Anma, Taketoshi Ando, Ryoji Itoh, Tatsuhiko Konishi, and Yukihiro Itoh. The method to visualize the domain-oriented-explanation of program's behaviors. In *ICCE '02: Proceedings of the International Conference on Computers in Education*, pages 910–911, Washington, DC, USA, 2002. IEEE Computer Society.
- [5] Fumihiko Anma, Tomonori Kato, Tatsuhiko Konishi, Yukihiro Itoh, and Toshio Okamoto. An educational system that explains the domain-oriented-explanation of program's behaviors. *Advanced Learning Technologies, IEEE International Conference on*, 0:198–199, 2007.
- [6] G. Arango. Domain analysis: from art form to engineering discipline. *SIGSOFT Softw. Eng. Notes*, 14(3):152–159, May 1989.
- [7] Christoph Aschwanden and Martha Crosby. Code scanning patterns in program comprehension. In *Proceedings of the 39th Hawaii International Conference on System Sciences (HICSS)*, Kauai, Hawaii, 2006.
- [8] Aybüke Aurum and Claes Wohlin. *Engineering and Managing Software Requirements*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
- [9] Roland C. Backhouse. *Syntax of Programming Languages: Theory and Practice*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1979.
- [10] T. Ball and S. G. Eick. Software visualization in the large. *Computer*, 29(4):33–43, 1996.

- [11] V. R. Basili, F. Shull, and F. Lanubile. Building knowledge through families of experiments. *Software Engineering, IEEE Transactions on*, 25(4):456–473, 1999.
- [12] Victor R. Basili and Harlan D. Mills. Understanding and documenting programs. *IEEE Transactions on Software Engineering*, 8(3):270–283, 1982.
- [13] Anindya Basu. *A language-based approach to protocol construction*. PhD thesis, Cornell University, Ithaca, NY, USA, 1998.
- [14] Berndt Bellay and Harald Gall. A comparison of four reverse engineering tools. In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, pages 2+, Washington, DC, USA, 1997. IEEE Computer Society.
- [15] C. Bennett, D. Myers, M. A. Storey, D. M. German, D. Ouellet, M. Salois, and P. Charland. A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams. *J. Softw. Maint. Evol.*, 20(4):291–315, 2008.
- [16] Jon Bentley. Programming pearls: little languages. *Commun. ACM*, 29(8):711–721, August 1986.
- [17] Joseph Bergin, Jim Roberts, Richard Pattis, and Mark Stehlik. *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1996.
- [18] M. Beron, P. Henriques, M. Varanda, and R. Uzal. PICS un sistema de comprensión e inspección de programas. *Congreso Argentino de Ciencias de la Computación CACIC 2007*, 13:462–473, 2007.
- [19] Mario Berón. Program inspection to interconnect the behavioral and operational views for program comprehension. Technical report, National University of San Luis & University of Minho, 2009.
- [20] Mario Berón, Daniela da Cruz, Maria João Varanda Pereira, Pedro R. Henriques, and Roberto Uzal. Evaluation criteria of software visualization systems used for program comprehension. In Universidade de Évora, editor, *Interacção'08 – 3ª Conferência Interação Pessoa-Máquina*, October 2008.
- [21] Mario Berón, Pedro R. Henriques, Pereira, and Roberto Uzal. A system to understand programs written in c language by code annotation. In *European Joint Conference on Theory and Practice of Software (ETAPS 2007)*, pages 1–7, 2007.
- [22] Mario Berón, Pedro R. Henriques, Maria J. V. Pereira, and Roberto Uzal. Static and dynamic strategies to understand c programs by code annotation. In *OpenCert'07, 1st Int. Workshop on Foundations and Techniques for Open Source Software Certification (collocated with ETAPS'07)*, 2007.

- [23] Mario Berón, Pedro R. Henriques, Maria João Varanda Pereira, and Roberto Uzal. Program inspection to interconnect behavioral and operational view for program comprehension. In *York Doctoral Symposium, 2007*. University of York, UK, 2007.
- [24] Mario Berón, Pedro R. Henriques, Maria João Varanda Pereira, Roberto Uzal, and G. Montejano. A language processing tool for program comprehension. In *CACIC'06 - XII Argentine Congress on Computer Science, Universidad Nacional de San Luis, Argentina, 2006*.
- [25] A. Beszédes, T. Gergely, and T. Gyimóthy. Graph-less dynamic dependence-based dynamic slicing algorithms. In *Source Code Analysis and Manipulation, 2006. SCAM '06. Sixth IEEE International Workshop on*, pages 21–30, 2006.
- [26] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering*, pages 482–498, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.
- [27] A. Blackwell, C. Britton, A. Cox, T. R. G. Green, C. Gurr, G. Kadoda, M. Kutar, M. Loomes, C. Nehaniv, M. Petre, C. Roast, C. Roe, A. Wong, and R. Young. Cognitive dimensions of notations: Design tools for cognitive technology. In *Cognitive Technology: Instruments of Mind*, pages 325–341. Springer-Verlag, 2001.
- [28] Benjamin S. Bloom and David R. Krathwohl. *Taxonomy of Educational Objectives, Handbook 1: Cognitive Domain*. Addison Wesley Publishing Company, October 1956.
- [29] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, 1981.
- [30] David Bonyuet, Matt Ma, and Kamal Jaffrey. 3D visualization for software development. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services*, pages 708–715, Washington, DC, USA, 2004. IEEE Computer Society.
- [31] Ruven Brooks. Using a behavioral theory of program comprehension in software engineering. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, pages 196–201, Piscataway, NJ, USA, 1978. IEEE Press.
- [32] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, November 1983.
- [33] D. Bruce. What makes a good domain-specific language? apostle, and its approach to parallel discrete event simulation. In S. Kamin, editor, *DSL'97 - First ACM SIGPLAN Workshop on Domain-Specific Languages, in Association with POPL'97*, Paris, France, January 1997.

- [34] Tim Bull. *Software Maintenance by Program Transformation in a Wide Spectrum Language*. PhD thesis, University of Durham, Durham, England, 1994.
- [35] Tim Bull. Comprehension of safety-critical systems using domain-specific languages. In *Program Comprehension, 1996, Proceedings., Fourth Workshop on*, pages 108–122, 1996.
- [36] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, January 1999.
- [37] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *Software, IEEE*, 7(1):13–17, 1990.
- [38] Steven Clark and Curtis Becker. Using the cognitive dimensions framework to evaluate the usability of a class library. In M. Petre and B. Budgen, editors, *Proc. Joint Conf. EASE & PPIG*, pages 359–366, April 2003.
- [39] Richard Clayton, Spencer Rugaber, Lyman Taylor, and Linda Wills. A case study of domain-based program understanding. In *WPC '97: Proceedings of the 5th International Workshop on Program Comprehension (WPC '97)*, pages 102–110, Washington, DC, USA, 1997. IEEE Computer Society.
- [40] Brendan Cleary and Chris Exton. Chive - a program source visualisation framework. In *IWPC '04: Proceedings of the 12th IEEE International Workshop on Program Comprehension*, pages 268–269, Washington, DC, USA, 2004. IEEE Computer Society.
- [41] Charles Consel, Fabien Latry, Laurent Réveillère, and Pierre. Cointe. A generative programming approach to developing dsl compilers. In R. Gluck and M. Lowry, editors, *Fourth International Conference on Generative Programming and Component Engineering (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 29–46, Tallinn, Estonia, sep 2005. Springer-Verlag.
- [42] B. Cornelissen, A. Zaidman, A. van Deursen, and B. van Rompaey. Trace visualization for program comprehension: A controlled experiment. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pages 100–109, 2009.
- [43] K. Craik. *The Nature of Explanation*. Cambridge University Press, Cambridge, England, 1943.
- [44] Jesús S. Cuadrado and Jesús G. Molina. Building domain-specific languages for model-driven development. *IEEE Software*, 24(5):48–55, 2007.
- [45] J. V. Cugini. *General purpose programming languages*. Petrocelli Books, Inc., Princeton, NJ, USA, 1986.
- [46] John V. Cugini. *Selection and Use of General-Purpose Programming Languages – Overview*, volume 1. Superintendent of Documents, U.S. Government Printing Office, Washington, DC 20402., October 1984.

- [47] Conrad H. Cunningham. A little language for surveys: Constructing an internal DSL in Ruby. In *Proceedings of the ACM SouthEast Conference*, Auburn, Alabama, March 2008.
- [48] Daniela da Cruz, Pedro Rangel Henriques, and Maria João Varanda Pereira. 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, 2007.
- [49] Daniela da Cruz, Pedro Rangel Henriques, and Maria João Varanda Pereira. Alma vs DDD. *ComSIS – Computer Science an Information Systems Journal, Special Issue on Advances in Programming Languages*, 2, December 2008.
- [50] Daniela da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Code analysis: Past and present. In *Proceedings of the Third International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2009)*, March 2009.
- [51] Daniela da Cruz, Maria João Vrandá Pereira, Mario Beron, Rúben Fonseca, and Pedro Rangel Henriques. Comparing generators for language-based tools. In *Proceedings of the 1.st Conference on Compiler Related Technologies and Applications, CoRTA'07 – Universidade da Beira Interior, Portugal*, July 2007.
- [52] Brian de Alwis, Gail C. Murphy, and Martin P. Robillard. A comparative study of three program exploration tools. In *ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 103–112, Washington, DC, USA, 2007. IEEE Computer Society.
- [53] Yunbo Deng and Suraj Kothari. Recovering conceptual roles of data in a program. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, pages 342–351, Washington, DC, USA, 2002. IEEE Computer Society.
- [54] Michael Denny. Ontology building: A survey of editing tools, November 2002. <http://www.xml.com/pub/a/2002/11/06/ontologies.html>.
- [55] Massimiliano Di Penta, R. E. K. Stirewalt, and Eileen Kraemer. Designing your next empirical study on program comprehension. In *ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 281–285, Washington, DC, USA, 2007. IEEE Computer Society.
- [56] Stephan Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.
- [57] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, June 2000.

- [58] Marc Eaddy, Alfred V. Aho, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. CERBERUS: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pages 53–62, Washington, DC, USA, 2008. IEEE Computer Society.
- [59] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, March 2006.
- [60] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Aiding program comprehension by static and dynamic feature analysis. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, pages 602–611, Washington, DC, USA, 2001. IEEE Computer Society.
- [61] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. *SIGPLAN Not.*, 42(10):1–18, 2007.
- [62] Christopher Exton. Constructivism and program comprehension strategies. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, Washington, DC, USA, 2002. IEEE Computer Society.
- [63] Rickard E. Faith, Lars S. Nyland, and Jan F. Prins. Khepera: a system for rapid implementation of domain specific languages. In *DSL'97: Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, pages 19–31, Berkeley, CA, USA, 1997. USENIX Association.
- [64] Ricardo Falbo, Giancarlo Guizzardi, and Katia Duarte. An ontological approach to domain engineering. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 351–358, New York, NY, USA, 2002. ACM.
- [65] R. K. Fjeldstad and W. T. Hamlen. Application program maintenance study: Report to our respondents. In *Proceedings GUIDE 48*, April 1983.
- [66] Martin Fowler and Eric Evans. Fluent interfaces, December 2005. Available at: <http://www.martinfowler.com/bliki/FluentInterface.html>; Visited on May 2009.
- [67] William Frakes, Ruben P. Diaz, and Christopher Fox. DARE: Domain analysis and reuse environment. *Ann. Softw. Eng.*, 5:125–141, 1998.
- [68] Steve Freeman and Nat Pryce. Evolving an embedded domain-specific language in java. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 855–865, New York, NY, USA, 2006. ACM.
- [69] E. M. Gagnon and L. J. Hendren. SableCC, an object-oriented compiler framework. In *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*, pages 140–154, 1998.

-
- [70] Jeff Gray, Kathleen Fisher, Charles Consel, Gabor Karsai, Marjan Mernik, and Juha P. Tolvanen. Dsls: the good, the bad, and the ugly. In *OOP-SLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 791–794, New York, NY, USA, 2008. ACM.
- [71] T. R. G. Green. Cognitive dimensions of notations. In A. Sutcliffe and L. Macaulay, editors, *Proceedings of the fifth conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and computers V*, pages 443–460, Cambridge, UK, 1989. Cambridge University Press.
- [72] T. R. G. Green. The cognitive dimension of viscosity: A sticky problem for hci. In *INTERACT '90: Proceedings of the IFIP TC13 Third Interantional Conference on Human-Computer Interaction*, pages 79–86. North-Holland, 1990.
- [73] T. R. G. Green and A. Blackwell. Cognitive dimensions of information artefacts: A tutorial, October 1998.
- [74] T. R. G. Green, A. E. Blandford, L. Church, C. R. Roast, and S. Clarke. Cognitive dimensions: achievements, new directions, and open questions. *Journal of Visual Languages & Computing*, 17(4):328–365, August 2006.
- [75] Yann-Gaël Guéhéneuc. A theory of program comprehension: Joining vision science and program comprehension. *International Journal of Software Science and Computational Intelligence*, 1(2):54–72, 2009.
- [76] J. G. Hall, M. Jackson, R. C. Laney, B. Nuseibeh, and L. Rapanotti. Relating software requirements and architectures using problem frames. In *RE '02: Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, pages 137–144, Washington, DC, USA, 2002. IEEE Computer Society.
- [77] Donald E. Harter, Mayuram S. Krishnan, and Sandra A. Slaughter. The life cycle effects of software process improvement: a longitudinal analysis. In *ICIS '98: Proceedings of the international conference on Information systems*, pages 346–351, Atlanta, GA, USA, 1998. Association for Information Systems.
- [78] Jan Heering. Developments in domain-specific languages, November 2007. Colloquium Talk. Available at: http://homepages.cwi.nl/~jan/Developments_in_DSLs.pdf.
- [79] P. R. Henriques, M. J. V. Pereira, M. Mernik, M. Lenic, J. Gray, and H. Wu. Automatic generation of language-based tools using the lisa system. *Software, IEE Proceedings -*, 152(2):54–69, 2005.
- [80] Luis M. Gómez Henríquez. Software visualization: An overview. *UPGRADE*, 2(2):4–7, April 2001.

- [81] R. M. Herndon and V. A. Berzins. The realizable benefits of a language prototyping language. *IEEE Transactions on Software Engineering*, 14(6):803–809, 1988.
- [82] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4):196–202, June 1996.
- [83] Paul Hudak. Modular domain specific languages and tools. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, Washington, DC, USA, 1998. IEEE Computer Society.
- [84] Brad L. Hutchings and Brent E. Nelson. Using general-purpose programming languages for FPGA design. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 561–566, New York, NY, USA, 2000. ACM.
- [85] Darrel C. Ince and Derek Andrews, editors. *The Software Life Cycle*. Butterworth-Heinemann, Newton, MA, USA, 1990.
- [86] Michael Jackson. *Software Requirements And Specifications: A Lexicon of Practice, Principles and Prejudices*. Addison-Wesley Professional, August 1995.
- [87] Michael Jackson. Problem frames and software engineering. *Information and Software Technology*, 47(14):903–912, November 2005.
- [88] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.
- [89] Jevgeni Kabanov and Rein Raudjärv. Embedded typesafe domain specific languages for java. In *PPPJ '08: Proceedings of the 6th international symposium on Principles and practice of programming in Java*, pages 189–197, New York, NY, USA, 2008. ACM.
- [90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.
- [91] Amela Karahasanovic, Annette K. Levine, and Richard Thomas. Comprehension strategies and difficulties in maintaining object-oriented systems: An explorative study. *J. Syst. Softw.*, 80(9):1541–1559, 2007.
- [92] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Pr, March 2008.
- [93] Babak Khazaei and Emma Triffitt. Applying cognitive dimensions to evaluate and improve the usability of z formalism. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 571–577, New York, NY, USA, 2002. ACM.

-
- [94] Richard B. Kieburtz, Laura Mckinney, Jeffrey M. Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino P. Oliva, Tim Sheard, Ira Smith, and Lisa Walton. A software engineering experiment in software component generation. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 542–552, Washington, DC, USA, 1996. IEEE Computer Society.
- [95] Holger M. Kienle and Hausi A. Müller. Requirements of software visualization tools: A literature survey. *Visualizing Software for Understanding and Analysis, International Workshop on*, 0:2–9, 2007.
- [96] Bill Kinnersley. The language list. Available at: <http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm>; Visited on May 2009.
- [97] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 2008.
- [98] Claire Knight. Visualisation for program comprehension: Information and issues. Technical report, Visualisation Research Group, Centre for Software Maintenance. Department of Computer Science, University of Durham, December 1998.
- [99] Claire Knight and Malcolm Munro. Comprehension with[in] virtual environment visualisations. In *Proceedings of the IEEE 7th International Workshop on Program Comprehension*, pages 4–11, 1999.
- [100] Claire Knight and Malcolm Munro. Virtual but visible software. In *Information Visualization, 2000. Proceedings. IEEE International Conference on*, pages 198–205, 2000.
- [101] Donald E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, June 1968.
- [102] Jürgen Koenemann and Scott P. Robertson. Expert problem solving strategies for program comprehension. In *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 125–130, New York, NY, USA, 1991. ACM.
- [103] Dimitrios S. Kolovos, Richard F. Paige, Tim Kelly, and Fiona A. C. Polack. Requirements for domain-specific languages. In *Proc. 1st ECOOP Workshop on Domain-Specific Program Development (DSPD 2006)*, Nantes, France, July 2006.
- [104] Fedor A. Kolpakov. BioUML - framework for visual modelling and simulation biological systems. In *Proceedings of the International Conference on Bioinformatics of Genome Regulation and Structure*, 2002.
- [105] Tomaž Kosar, Pablo E. Martínez López, Pablo A. Barrientos, and Marjan Mernik. A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology.*, 50(5):390–405, April 2008.

- [106] Tomaž Kosar, Marjan Mernik, Matej Črepinšek, Pedro Rangel Henriques, Daniela da Cruz, Maria João Varanda Pereira, and Nuno Oliveira. Influence of domain-specific notation to program understanding. In *Proceedings of the International Multiconference on Computer Science and Information Technology - 2nd Workshop on Advances in Programming Languages (WAPL'2009)*, pages 673 — 680, Mragowo, Poland, October 2009. IEEE Computer Society Press.
- [107] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, June 1992.
- [108] David A. Ladd and Christopher J. Ramming. Two application languages in software production. In *VHLLS'94: USENIX 1994 Very High Level Languages Symposium Proceedings*, page 10, Berkeley, CA, USA, 1994. USENIX Association.
- [109] Michele Lanza. CodeCrawler - lessons learned in building a software visualization tool. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, pages 409–418, Washington, DC, USA, 2003. IEEE Computer Society.
- [110] Michele Lanza and Stéphane Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of OOPSLA 2001*, pages 300–311, 2001.
- [111] Seok W. Lee, Robin Gandhi, Divya Muthurajan, Deepak Yavagal, and Gail J. Ahn. Building problem domain ontology from security requirements in regulatory documents. In *SESS '06: Proceedings of the 2006 international workshop on Software engineering for secure systems*, pages 43–50, New York, NY, USA, 2006. ACM.
- [112] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. *UNIX: research system*, 2:375–387, 1990.
- [113] Stanley Letovsky and Elliot Soloway. Delocalized plans and program comprehension. *Software, IEEE*, 3(3):41–49, 1986.
- [114] Yang Li, Hongji Yang, and William Chu. A concept-oriented belief revision approach to domain knowledge recovery from source code. *Journal of Software Maintenance*, 13(1):31–52, 2001.
- [115] Christian Lindig and Gregor Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 349–359, New York, NY, USA, 1997. ACM.
- [116] Rob Lintern, Jeff Michaud, Margaret-Anne Storey, and Xiaomin Wu. Plugging-in visualization: experiences integrating a visualization tool with eclipse. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 47–56, New York, NY, USA, 2003. ACM.

- [117] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. *J. Syst. Softw.*, 7(4):341–355, 1987.
- [118] Na Liu, John Hosking, and John Grundy. MaramaTatau: Extending a domain specific visual language meta tool with a declarative constraint mechanism. In *VLHCC '07: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 95–103, Washington, DC, USA, 2007. IEEE Computer Society.
- [119] Linda Lohr and Shujen Chang. Psychology of learning for instruction. *Educational Technology Research and Development*, 53(1):108–110, March 2005.
- [120] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3D representations for software visualization. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 27–36, New York, NY, USA, 2003. ACM Press.
- [121] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan I. Maletic. An information retrieval approach to concept location in source code. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 214–223, Washington, DC, USA, 2004. IEEE Computer Society.
- [122] Vijay Menon and Keshav Pingali. A case for source-level transformations in matlab. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*, pages 53–65, New York, NY, USA, 1999. ACM.
- [123] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys.*, 37(4):316–344, December 2005.
- [124] Marjan Mernik, Tomaž Kosar, Matej Črepinšek, Pedro Rangel Henriques, Daniela da Cruz, Maria João Varanda Pereira, and Nuno Oliveira. Comparison of XAML and C# forms using cognitive dimensions framework. In *INForum'09 — Simpósio de Informática: 3rd Compilers, Programming Languages, Related Technologies and Applications (CoRTA 2009)*, pages 180 — 191, Lisbon, Portugal, September 2009. Faculdade de Ciências da Universidade de Lisboa.
- [125] Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer. Lisa: An interactive environment for programming language development. In *Compiler Construction*, pages 1–4. Springer Berlin / Heidelberg, 2002.
- [126] Marjan Mernik and Viljem Žumer. Domain-specific languages for software engineering. *Hawaii International Conference on System Sciences*, 9:9071+, 2001.
- [127] Tamás Mészáros and Tihamér 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.

- [128] Thomas P. Moran and Stuart K. Card. Applying cognitive psychology to computer systems: A graduate seminar in psychology. In *Proceedings of the 1982 conference on Human factors in computing systems*, pages 295–298, New York, NY, USA, 1982. ACM.
- [129] Jan-Erik Moström and David A. Carr. Programming paradigms and program comprehension by novices. In *PPIG'10 - Workshop 10 of the Psychology of Programmers Interest Group, Milton Keynes, Great Britain*, 1998.
- [130] H. A. Müller and K. Klashinsky. Rigi-a system for programming-in-the-large. In *ICSE '88: Proceedings of the 10th international conference on Software engineering*, pages 80–86, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.
- [131] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 18–28, New York, NY, USA, 1995. ACM Press.
- [132] Brad A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, March 1990.
- [133] Danish Nadeen and Leo Sauermann. From philosophy and mental-models to semantic desktop research: Theoretical overview. In *Proceedings of I-Semantics' 07*, 2007.
- [134] Lloyd H. Nakatani and Mark A. Jones. Jargons and infocentrism. In *In First ACM SIGPLAN Workshop on Domain-Specific Languages*, pages 59–74, 1997.
- [135] Ulric Neisser. *Cognitive Psychology*. Appleton-Century-Crofts, New York, 1967.
- [136] Michael L. Nelson. A survey of reverse engineering and program comprehension. In *In ODU CS 551 - Software Engineering Survey*, page 2, 1996.
- [137] Hanne R. Nielson and Flemming Nielson. *Semantics with applications: a formal introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1992.
- [138] Michael P. O'Brien. Software comprehension – a review & research direction. Technical report, University of Limerick, November 2003.
- [139] Nuno Oliveira, Pedro Rangel Henriques, Daniela da Cruz, and Maria João Varanda Pereira. Visuallisa: Visual programming environment for attribute grammars specification. In *Proceedings of the International Multi-conference on Computer Science and Information Technology - 2nd Workshop on Advances in Programming Languages (WAPL'2009)*, pages 689 – 696, Mragowo, Poland, October 2009. IEEE Computer Society Press.

-
- [140] Nuno Oliveira, Pedro Rangel Henriques, Daniela da Cruz, Maria João Varanda Pereira, Marjan Mernik, Tomaž Kosar, and Matej Črepinšek. Applying program comprehension techniques to karel robot programs. In *Proceedings of the International Multiconference on Computer Science and Information Technology - 2nd Workshop on Advances in Programming Languages (WAPL'2009)*, pages 697–704, Mragowo, Poland, October 2009. IEEE Computer Society Press.
- [141] Nuno Oliveira, Maria João Varanda Pereira, Daniela da Cruz, and Pedro Rangel Henriques. VisualLISA. Technical report, Universidade do Minho, February 2009. www.di.uminho.pt/~gepl/VisualLISA/documentation.php.
- [142] Nuno Oliveira, Maria João Varanda Pereira, Pedro Rangel Henriques, and Daniela da Cruz. Domain specific languages: A theoretical survey. In *INForum'09 — Simpósio de Informática: 3rd Compilers, Programming Languages, Related Technologies and Applications (CoRTA'2009)*, pages 35 — 46, Lisbon, Portugal, September 2009. Faculdade de Ciências da Universidade de Lisboa.
- [143] Nuno Oliveira, Maria João Varanda Pereira, Pedro Rangel Henriques, and Daniela da Cruz. Visualization of domain-specific program's behavior. In *Proceedings of VISSOFT 2009, 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 37–40, Edmonton, Alberta, Canada, September 2009. IEEE Computer Society.
- [144] Nuno Oliveira, Maria João Varanda Pereira, Pedro Rangel Henriques, Daniela da Cruz, and Bastian Cramer. Visuallisa: A domain specific visual language for attribute grammars. In *INForum'09 — Simpósio de Informática: 3rd Compilers, Programming Languages, Related Technologies and Applications (CoRTA'2009)*, pages 155—167, Lisbon, Portugal, September 2009. Faculdade de Ciências da Universidade de Lisboa.
- [145] Cristóbal Pareja-Flores and J. Ángel Velázquez-Iturbide. Representing abstractions. *UPGRADE*, 2(2):2–3, April 2001.
- [146] Terence Parr and Russell W. Quong. AntLR: A predicated-LL(K) parser generator. *Software Practice and Experience*, 25(7):789–810, July 1995.
- [147] Richard Pattis. *Karel, The Robot: A Gentle Introduction to the Art of Programming*. John Wiley and Sons, Inc., 1st edition, 1981.
- [148] Ryan Paul. Designing and implementing a domain-specific language. *Linux J.*, 2005(135):7+, 2005.
- [149] Luís Pedro, Vasco Amaral, and Didier Buchs. Foundations for a domain specific modeling language prototyping environment: A compositional approach. In *Proc. 8th OOPSLA ACM-SIGPLAN Workshop on Domain-Specific Modeling (DSM)*. University of Jyväskylä, October 2008.
- [150] Nancy Pennington. Comprehension strategies in programming. *Empirical studies of programmers: second workshop*, pages 100–113, 1987.

- [151] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.
- [152] Maria João Varanda Pereira and Pedro Rangel Henriques. Visualization/animation of programs in Alma: obtaining different results. In *HCC '03: Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments*, pages 260–262, Washington, DC, USA, 2003. IEEE Computer Society.
- [153] Maria João Varanda Pereira, Marjan Mernik, Daniela da Cruz, and Pedro Rangel Henriques. Program comprehension for domain-specific languages. *ComSIS – Computer Science and Information Systems Journal, Special Issue on Compilers, Related Technologies and Applications*, 5(2):1–17, Dec 2008.
- [154] John Peterson. A language for mathematical visualization. In *Proceedings of FPDE'02: Functional and Declarative Languages in Education*, 2002.
- [155] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with haskell. In *PADL '99: Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, pages 91–105, London, UK, 1998. Springer-Verlag.
- [156] Marian Petre and Ed de Quincey. A gentle overview of software visualisation. *Psychology of Programming Interest Group (PPIG)*, September 2006.
- [157] 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.
- [158] Denys Poshyvanyk, Andrian Marcus, and Yubo Dong. Jiriss - an eclipse plugin for source code exploration. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 252–255, Washington, DC, USA, 2006. IEEE Computer Society.
- [159] James F. Power and Brian A. Malloy. A metrics suite for grammar-based software: Research articles. *J. Softw. Maint. Evol.*, 16(6):405–426, 2004.
- [160] Blaine A. Price, Ronald M. Baecker, and Ian S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, September 1993.
- [161] Blaine A. Price, Ronald M. Baecker, and Ian S. Small. An introduction to software visualization. In J. Stasko, J. Dominique, M. Brown, and B. Price, editors, *Software Visualization*, pages 4–26, London, England, 1998. MIT Press.
- [162] Blaine A. Price, Ian S. Small, and Ronald M. Baecker. A taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4:211–221, 1992.
- [163] Christophe Prud'homme. A domain specific embedded language in c++ for automatic differentiation, projection, integration and variational formulations. *Sci. Program.*, 14(2):81–110, 2006.

- [164] Ayende Rahien. *Building Domain-Specific Languages in Boo*. Manning, February 2008. (Note: Unedited Draft).
- [165] Václav Rajlich and Norman Wilde. The role of concepts in program comprehension. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, pages 271–278, Washington, DC, USA, 2002. IEEE Computer Society.
- [166] Vennila Ramalingam and Susan Wiedenbeck. An empirical study of novice program comprehension in the imperative and object-oriented styles. In *ESP '97: Papers presented at the seventh workshop on Empirical studies of programmers*, pages 124–139, New York, NY, USA, 1997. ACM Press.
- [167] Aoun Raza, Gunther Vogel, and Erhard Plödereder. Bauhaus - a tool suite for program analysis and reverse engineering. In *Reliable Software Technologies - Ada-Europe 2006*, pages 71–82. LNCS (4006), June 2006.
- [168] Steven P. Reiss. An engine for the 3D visualization of program information. Technical report, Brown University, Providence, RI, USA, 1995.
- [169] Steven P. Reiss. Visual representations of executing programs. *J. Vis. Lang. Comput.*, 18(2):126–148, April 2007.
- [170] G. C. Roman and K. C. Cox. A taxonomy of program visualization systems. *Computer*, 26(12):11–24, 1993.
- [171] S. Rugaber, S. B. Ornburn, and R. J. Leblanc. Recognizing design decisions in programs. *Software, IEEE*, 7(1):46–54, 1990.
- [172] Spencer Rugaber. Program comprehension. Technical report, Georgia Institute of Technology, May 1995.
- [173] Spencer Rugaber. The use of domain knowledge in program understanding. *Annals of Software Engineering*, 9(1-4):143–192, 2000.
- [174] Kamran Sartipi, Lingdong Ye, and Hossein Safyallah. Alborz: An interactive toolkit to extract static and dynamic views of a software system. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 256–259, Washington, DC, USA, 2006. IEEE Computer Society.
- [175] Ben Shneiderman and Richard Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Parallel Programming*, 8(3):219–238, June 1979.
- [176] João C. Silva, João Saraiva, and José C. Campos. A generic library for GUI reasoning and testing. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 121–128, New York, NY, USA, 2009. ACM.

- [177] Susan E. Sim, Margaret-Anne Storey, and Andreas Winter. A structured demonstration of five program comprehension tools: Lessons learnt. In *WCRE '00: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, page 210, Washington, DC, USA, 2000. IEEE Computer Society.
- [178] Mark A. Simos. Organization domain modeling (ODM): formalizing the core domain modeling life cycle. *SIGSOFT Softw. Eng. Notes*, 20(SI):196–205, 1995.
- [179] Emin G. Sirer and Brian N. Bershad. Using production grammars in software testing. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*, pages 1–13, New York, NY, USA, 1999. ACM.
- [180] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *Software Reusability*, 2:235–267, 1989.
- [181] Diomidis Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99, February 2001.
- [182] Richard M. Stallman, Roland Pesch, and Stan Shebs. *Debugging with GDB - The GNU Source-Level Debugger*. GNU Press, March 2002.
- [183] M. A. Storey, F. D. Fracchia, and H. A. Muller. Cognitive design elements to support the construction of a mental model during software visualization. *Program Comprehension, 1997. IWPC'97. Proceedings of Fifth International Workshop on PC*, pages 17–28, 1997.
- [184] Margaret-Anne Storey. Designing a software exploration tool using a cognitive framework of design elements. In Kang Zhang, editor, *Software Visualization: From Theory to Practice*, pages 113–148. Springer, 2003.
- [185] Margaret-Anne Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, 14(3):187–208, 2006.
- [186] Margaret-Anne Storey, Casey Best, Jeff Michaud, Derek Rayside, Marin Litoiu, and Mark Musen. SHriMP views: an interactive environment for information visualization and navigation. In *CHI '02: CHI '02 extended abstracts on Human factors in computing systems*, pages 520–521, New York, NY, USA, 2002. ACM.
- [187] Yu Sun, Zekai Demirezen, Marjan Mernik, Jeff Gray, and Barrett Bryant. Is my dsl a modeling or programming language? In *Proceedings of 2nd International Workshop on Domain-Specific Program Development (DSPD)*, Nashville, Tennessee, 2008.
- [188] Richard N. Taylor, Will Tracz, and Lou Coglianese. Software development using domain-specific software architectures. *SIGSOFT Softw. Eng. Notes*, 20(5):27–38, 1995.

- [189] Tim Tiemens. Cognitive models of program comprehension. Technical report, Software Engineering Research Center, December 1989.
- [190] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Efficient instrumentation for code coverage testing. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 86–96, New York, NY, USA, 2002. ACM.
- [191] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.
- [192] Paolo Tonella, Marco Torchiano, Bart Du Bois, and Tarja Systä. Empirical studies in reverse engineering: state of the art and future trends. *Empirical Softw. Engg.*, 12(5):551–571, 2007.
- [193] M. G. J. Van den Brand, B. Cornelissen, P. A. Oliver, and J. J. Vinju. TIDE: A generic debugging framework - tool demonstration. *Electronic Notes in Theoretical Computer Science*, 141(4):161–165, December 2005.
- [194] Arie van Deursen, Jan Heering, and Paul Klint. *Language Prototyping: An Algebraic Specification Approach: Vol. V*. World Scientific Publishing Co., Inc., 1996.
- [195] Arie van Deursen and Paul Klint. Little languages: little maintenance? Technical report, University of Amsterdam, Amsterdam, The Netherlands, 1997.
- [196] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35:26–36, 2000.
- [197] Eelco Visser. WebDSL: A case study in domain-specific language engineering. In R. Lammel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, Lecture Notes in Computer Science. Springer, 2008.
- [198] A. von Mayrhauser and A. M. Vans. From program comprehension to tool requirements for an industrial environment. In *In Proceedings of IEEE Workshop on Program Comprehension*, pages 78–86, 1993.
- [199] A. von Mayrhauser and A. M. Vans. Program understanding - a survey. Technical report, Colorado State University, August 1994.
- [200] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.
- [201] David A. Watt. *Programming language concepts and paradigms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.
- [202] David M. Weiss and Chi T. R. Lay. *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

- [203] Richard Wettel and Michele Lanza. Visualizing software systems as cities. In *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*, pages 92–99, 2007.
- [204] David S. Wile. Supporting the dsl spectrum. *Journal of Computing and Information Technology*, 9(4):263–287, 2001.
- [205] Hui Wu. *Grammar-Driven Generation of Domain-Specific Language Testing Tools Using Aspects*. PhD thesis, University of Alabama, Birmingham, Alabama, 2007.
- [206] Hui Wu, Jeff Gray, and Marjan Mernik. Grammar-driven generation of domain-specific language debuggers. *Softw. Pract. Exper.*, 38(10):1073–1103, 2008.
- [207] Tao Xie, Jian Pei, and Ahmed E. Hassan. Mining software engineering data. In *Software Engineering - Companion, 2007. ICSE 2007 Companion. 29th International Conference on*, pages 172–173, 2007.
- [208] Shaochun Xu. A cognitive model for program comprehension. In *SERA '05: Proceedings of the Third ACIS Int'l Conference on Software Engineering Research, Management and Applications*, pages 392–398, Washington, DC, USA, 2005. IEEE Computer Society.
- [209] Peter Young. Program comprehension, May 1996. Available at: <http://vrg.dur.ac.uk/misc/PeterYoung/pages/work/documents/lit-survey/prog-comp/index.html>; Visited on June 2009.
- [210] Andy Zaidman, Bram Adams, and Kris De Schutter. Applying dynamic analysis in a legacy context: An industrial experience report. In *Proceedings of the 1st International Workshop on Program Comprehension Through Dynamic Analysis (PCODA)*, Pittsburgh, PE, USA, 2005.
- [211] Carmen Zannier, Grigori Melnik, and Frank Maurer. On the success of empirical studies in the international conference on software engineering. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 341–350, New York, NY, USA, 2006. ACM Press.
- [212] Andreas Zeller and Dorothea Lütkehaus. DDD - a free graphical front-end for UNIX debuggers. *SIGPLAN Not.*, 31(1):22–27, January 1996.
- [213] Qunxiong Zhu and Xiaoyong Lin. Top-down and bottom-up strategies for incremental maintenance of frequent patterns. In *Emerging Technologies in Knowledge Discovery and Data Mining*, pages 445–456. Springer Berlin / Heidelberg, 2009.

Appendix A

AnimXXXNode Family of Classes

The *AnimXXXNode* family of classes joins all the necessary constructs to build a D_{ap} AST. See Chapter 5 for more information on this topic. In this appendix are listed all the classes pertaining to that family. For each class is presented the name and the most important attributes to be filled with information.

- *AnimAssignNode* — a Variable, a Statement , list of Animation Patterns
- *AnimCallNode* — an Identifier, Arguments , list of Animation Patterns
- *AnimConstNode* — a Constant, list of Animation Patterns
- *AnimDeclNode* — a Variable, a Type, a Structure, an AssignNode, a list of Animation Patterns
- *AnimFuncNode* — a Variable, a Type , Arguments, a Body, list of Animation Patterns
- *AnimIfElseNode* — a BoolOperation, a Body , a Body, list of Animation Patterns
- *AnimIfNode* — a BoolOperation, aBody , list of Animation Patterns
- *AnimOperNode* — a Statement, a Statement , an Operation, list of Animation Patterns
- *AnimOperRelNode* — a Statement, a Statement , an Operation, list of Animation Patterns
- *AnimReadNode* — a Variable, list of Animation Patterns
- *AnimReturnNode* — a Statement , list of Animation Patterns
- *AnimVarNode* — a Variable, a Type , list of Animation Patterns
- *AnimWhileNode* — a Test, a Body , list of Animation Patterns
- *AnimWriteNode* — a Statement , list of Animation Patterns

Some Considerations. The common factor of these constructs is the list of animation patterns. Some of them can be declared with less attributes than the presented here. The `AnimOperRelNode` is an example. Since it describes a logical operation, the user may be interested in using an unary operator rather than a binary one.

Appendix B

Experiment Questionnaires

In this appendix are presented all the pages composing the structured questionnaires used in the experiment reported in Chapter 7.

Figure B.1 presents the form used to identify the participants, and retrieve important characteristics to the analysis of the results obtained from the experiment.

Figures B.2 to B.6 show the eight tasks proposed to the participants, spread by the five pages of the questionnaire.

Experimental Proof:
Alma², a DSL Program Comprehension Tool

Participant Identification

This part of the document is used to identify the characteristics of the participants in the experiment. Please answer the questions with sincerity, for a correct evaluation of the experiment's results. Thank you for your attention!

1. **Gender:** ___ M ___ F

2. **Academic Degree:**

___ B.Sc. Student (Bachelor of Science – equivalent to the first cycle of Bologna's)

___ M.Sc. Student (Master of Science – equivalent to the second cycle of Bologna's)

___ Ph.D. Student (Philosophy Doctor)

___ Other. Name it: _____

3. **Informatics Area:** _____

4. The following table presents a set of questions to evaluate the experience of the participants on performing some tasks related with the experiment being realized. Use the numbers from 1 to 5 to score the times (☺) you performed these tasks, when developing software or related jobs. Use again the numbers from 1 to 5 to score your expertise (★) on these tasks.

TASK	☺	★
Reading programs to understand their functionalities (it involves analysis of the source-code, read documentation, etc.)		
Using Program Comprehension tools (e.g. CodeCrawler, SHriMP, Bauhaus, Rigi, etc.)		
Using IDEs with debugging features to develop software (e.g. Eclipse, Netbeans, Visual Studio, etc.)		
Using Domain Specific Languages (DSLs) to develop software or to other purposes (e.g. LaTeX, HTML, VHDL, DOT, etc.)		
Comprehending Programs written in DSLs		

Assume scores as:

1 ☺ – Never

2 ☺ – Rarely

3 ☺ – Sometimes

4 ☺ – Several times

5 ☺ – Always

1 ★ – Null

2 ★ – Low

3 ★ – Medium

4 ★ – High

5 ★ – Expert

Master in Informatics – Nuno Oliveira

Figure B.1: Participant Identification Form

Experimental Proof:
Alma², a DSL Program Comprehension Tool

Questionnaire

Before starting:

The following questionnaire is composed of 8 (eight) tasks. Each task has four steps:

1. The participant is invited to stamp the time at the beginning of the task execution.
2. A question (Q) is asked, concerning a program written either in Karel or Lavanda Languages. Besides the question, it is provided a set of tips (T) to execute Alma or Alma², and when needed, some tips to help solving the question. In the end, a space for answer (A) is given when it is required an open answer, or four possible answers are provided when it is required a multiple-choice answer.
3. The participant is invited to score the utility of the problem domain visualizations. For this the participant must use the numbers from 1 to 5, according to the following:

- 1 ☆ – None
- 2 ☆ – Negligible
- 3 ☆ – Medium
- 4 ☆ – High
- 5 ☆ – Excellent

4. The participant is invited to stamp the time at the end of the task execution.

Thank you very much!

Task 1 – Karel Language with Alma² (PQ)

⌚ _____ (Please, stamp the start time)

Q: Read carefully:

Identify the function, which always provokes the following output:

- 3 beepers picked up and
- The robot is turned 270°.

T: - Invoke **run.sh 1 (run.bat 1** on windows) on command line inside the folder 'Experiment' to access the program in Alma².

A: Choose the correct answer:

- | | |
|-------------|-------------|
| 1. __ func1 | 3. __ func3 |
| 2. __ func2 | 4. __ func4 |

Please, classify the help that problem domain animation synchronized with the program domain visualization gave you on the resolution of this task:

☆ _____ (Regard classification table above)

⌚ _____ (Please, stamp the finish time)

Master in Informatics – Nuno Oliveira

Experimental Proof:
Alma², a DSL Program Comprehension Tool

Task 2 – Karel Language with Alma (PQ)

⌚ _____ (Please, stamp the start time)

Q: Read carefully:

Identify the functions, which may provoke, exactly, the following output:

- 1 beeper picked up;
- 1 beeper dropped and
- The robot steps twice in a direction and steps once in a different direction.

T: - Invoke **run.sh 2 (run.bat 2** on windows) on command line inside the folder 'Experiment' to access the program in Alma.

A: Choose the correct answer(s):

- | | |
|-------------|-------------|
| 1. __ func1 | 3. __ func3 |
| 2. __ func2 | 4. __ func4 |

Please, classify the help that problem domain animation synchronized with the program domain visualization, seen before in Alma², would have given you on the resolution of this task:

★ _____ (Regard classification table above)

⌚ _____ (Please, stamp the finish time)

Task 3 – Karel Language with Alma² (EQ)

⌚ _____ (Please, stamp the start time)

Q: Read carefully:

Modify the program (or explain how you'd do it) in order to make the robot catch two beepers in three distinct rows.

T: - Invoke **run.sh 3 (run.bat 3** on windows) on command line inside the folder 'Experiment' to access the program in Alma².

- You can change the source program in the following path: /Source/q3.txt, and invoke run.sh 3 again to verify your answer.

A: Write your answer (use the back of this sheet if you don't have space):

Please, classify the help that problem domain animation synchronized with the program domain visualization gave you on the resolution of this task:

★ _____ (Regard classification table above)

⌚ _____ (Please, stamp the finish time)

Experimental Proof:
Alma², a DSL Program Comprehension Tool

Task 4 – Karel Language with Alma (EQ)

⌚ _____ (Please, stamp the start time)

Q: Read carefully:

Improve the robot's efficiency, by removing all the pairs of instructions¹ that cancel² each other, in the main program.

T: - Invoke **run.sh 4 (run.bat 4)** on windows) on command line inside the folder 'Experiment' to access the program in Alma.

-¹ By pair of instructions is intended 2 sequent instructions in the program: e.g. if i1 and i2 cancel each other, they form a pair in this sequence: i1 i2 i3, but not in this sequence i1, i3, i2.

-² Two instructions cancel each other if the robot's state before executing the first instruction is the same after executing the second instruction.

A: Name all the pairs (use the back of this sheet if you don't have space):

Note: write your answer following this format: 1 – {inst1, inst2}; 2 – {inst3, inst4}; ...

Please, classify the help that problem domain animation synchronized with the program domain visualization, seen before in Alma², would have given you on the resolution of this task:

★ _____ (Regard classification table above)

⌚ _____ (Please, stamp the finish time)

Task 5 – Lavanda Language with Alma² (PQ)

⌚ _____ (Please, stamp the start time)

Q: Read carefully:

*How many pieces (in the overall) exist with the following configuration:
- body-color-wool*

T: - Invoke **run.sh 5 (run.bat 5)** on windows) on command line inside the folder 'Experiment' to access the program in Alma².

A: Choose the correct answer:

- | | |
|-----------------|-----------------|
| 1. __ 5 (five) | 3. __ 6 (six) |
| 2. __ 7 (seven) | 4. __ 8 (eight) |
| 5. __ 9 (nine) | 6. __ 10 (ten) |

Please, classify the help that problem domain animation synchronized with the program domain visualization gave you on the resolution of this task:

★ _____ (Regard classification table above)

⌚ _____ (Please, stamp the finish time)

Experimental Proof:
Alma², a DSL Program Comprehension Tool

Task 6 – Lavanda Language with Alma (PQ)

⌚ _____ (Please, stamp the start time)

Q: Read carefully:

*In how many bags exist 5 (five) or more pieces with the following configuration:
- house-white-fiber*

T: - Invoke **run.sh 6 (run.bat 6)** on windows) on command line inside the folder 'Experiment' to access the program in Alma.

A: Choose the correct answer:

- | | |
|-----------------|----------------|
| 1. __ 1 (one) | 3. __ 2 (two) |
| 2. __ 3 (three) | 4. __ 4 (four) |

Please, classify the help that problem domain animation synchronized with the program domain visualization, seen before in Alma², would have given you on the resolution of this task:

★ _____ (Regard classification table above)

⌚ _____ (Please, stamp the finish time)

Task 7 – Lavanda Language with Alma² (EQ)

⌚ _____ (Please, stamp the start time)

Q: Read carefully:

Remove one bag, in order to have 24 pieces of body cloth, in the overall.

T: - Invoke **run.sh 7 (run.bat 7)** on windows) on command line inside the folder 'Experiment' to access the program in Alma².

A: Write the name of the removed bag's owner:

1. _____

Please, classify the help that problem domain animation synchronized with the program domain visualization gave you on the resolution of this task:

★ _____ (Regard classification table above)

⌚ _____ (Please, stamp the finish time)

Experimental Proof:
Alma², a DSL Program Comprehension Tool

Task 8 – Lavanda Language with Alma (EQ)

🕒 _____ (Please, stamp the start time)

Q: Read carefully:

What bag should be added to have, in the overall, the indicated number of pieces with the following configuration (notice that the other possible configurations may exist in the bags):

- *body-color-cotton = 10*
- *body-white-wool = 5*
- *house-color-fiber = 8*

T: - Invoke **run.sh 8** (**run.bat 8** on windows) on command line inside the folder 'Experiment' to access the program in Alma.

A: Choose the correct answer:

- | | |
|--|--|
| <p>1. ____</p> <p><i>5 Ulysses</i>
(<i>body-color-cotton 1,</i>
<i>house-white-fiber 4,</i>
<i>house-color-cotton 2)</i></p> | <p>3. ____</p> <p><i>5 Ulysses</i>
(<i>house-color-fiber 2,</i>
<i>body-white-fiber 4,</i>
<i>body-color-cotton 10,</i>
<i>body-white-wool 1)</i></p> |
| <p>2. ____</p> <p><i>5 Ulysses</i>
(<i>body-color-cotton 8,</i>
<i>body-white-wool 4,</i>
<i>house-color-cotton 2)</i></p> | <p>4. ____</p> <p><i>5 Ulysses</i>
(<i>house-color-fiber 1,</i>
<i>body-color-cotton 10,</i>
<i>body-white-cotton 2,</i>
<i>body-white-wool 3)</i></p> |

Please, classify the help that problem domain animation synchronized with the program domain visualization, seen before in Alma², would have given you on the resolution of this task:

★ _____ (Regard classification table above)
🕒 _____ (Please, stamp the finish time)
