Com
SIS

Computer Science and Information Systems

Published by ComSIS Consortium

Special Issue on Advances in Languages, Related Technologies and Applications

Volume 7, Number 2
April 2010

ComSIS Editorial office:
University of Novi Sad, Faculty of Sciences,
Department of Mathematics and Informatics
Trg Dositeja Obradovica 4, 21000 Novi Sad, Serbia
Phone: +381 21 458 888; **Fax:** +381 21 6350 458
www.comsis.org; Email: comsis@uns.ac.rs

Volume 7, Number 2, Special Issue, 2010
Novi Sad

Computer Science and Information Systems

ISSN: 1820-0214

ComSIS Journal is sponsored by:

Ministry of Science and Technological Development of Republic of Serbia - http://www.nauka.gov.rs/lat

Polskie Towarzystwo Informatyczne - PTI (Polish Information Processing Society) - http://www.pti.org.pl/

Computer Science and Information Systems

AIMS AND SCOPE

Computer Science and Information Systems (ComSIS) is an international refereed journal, published in Serbia. The objective of ComSIS is to communicate important research and development results in the areas of computer science, software engineering, and information systems.

We publish original papers of lasting value covering both theoretical foundations of computer science and commercial, industrial, or educational aspects that provide new insights into design and implementation of software and information systems. ComSIS also welcomes surveys papers that contribute to the understanding of emerging and important fields of computer science. Regular columns of the journal cover reviews of newly published books, presentations of selected PhD and master theses, as well as information on forthcoming professional meetings. In addition to wide-scope regular issues, ComSIS also includes special issues covering specific topics in all areas of computer science and information systems.

ComSIS publishes invited and regular papers in English. Papers that pass a strict reviewing procedure are accepted for publishing. The acceptance rate so far was approximately 40%. ComSIS is published semiannually.

Indexing Information

ComSIS is indexed and abstracted in the following:

- Science Citation Index Expanded (also known as SciSearch) and Journal Citation Reports / Science Edition by Thomson Reuters,
- Computer Science Bibliography, University of Trier (DBLP),
- EMBASE (Elsevier),
- Scopus (Elsevier),
- Summon (Serials Solutions),
- Selected for coverage in EBSCO bibliographic databases,
- Selected for coverage in IET bibliographic database Inspec, starting with 2009 published material,
- Selected for coverage in FIZ Karlsruhe bibliographic database io-port,
- Google Scholar,
- Center for Evaluation in Education and Science and Ministry of Science of Republic of Serbia (CEON) in cooperation with the National Library of Serbia,
- Serbian Citation Index (SCIndeks),
- doiSerbia.

Information for Contributors

The Editors will be pleased to receive contributions from all parts of the world. An electronic version (MS Word), or three hard-copies of the manuscript written in English, intended for publication and prepared as described in "Manuscript Requirements" (which may be downloaded from http://www.comsis.org), along with a cover letter containing the corresponding author's details should be sent to one of the Editors.

Criteria for Acceptance

Criteria for acceptance will be appropriateness to the field of Journal, as described in the Aims and Scope, taking into account the merit of the content and presentation. There is no page limit on manuscripts submitted.
Manuscripts will be refereed in the manner customary with scientific journals before being accepted for publication.

Copyright and Use Agreement

All authors are requested to sign the "Transfer of Copyright" agreement before the paper may be published. The copyright transfer covers the exclusive rights to reproduce and distribute the paper, including reprints, photographic reproductions, microform, electronic form, or any other reproductions of similar nature and translations. Authors are responsible for obtaining from the copyright holder permission to reproduce the paper or any part of it, for which copyright exists.

Computer Science and Information Systems

Volume 7, Number 2, Special Issue, April 2010

CONTENTS

Editorial

Guest Editorial

Papers

247 **Comparing General-Purpose and Domain-Specific Languages: An Empirical Study**
Tomaž Kosar, Nuno Oliveira, Marjan Mernik, Maria João Varanda Pereira, Matej Črepinšek, Daniela da Cruz, Pedro Rangel Henriques

265 **VisualLISA: A Visual Environment to Develop Attribute Grammars**
Nuno Oliveira, Maria João Varanda Pereira, Pedro Rangel Henriques, Daniela da Cruz, Bastian Cramer

291 **Annotation Based Parser Generator**
Jaroslav Porubän, Michal Forgáč, Miroslav Sabo, Marek Běhálek

309 **On Automata and Language Based Grammar Metrics**
Matej Črepinšek, Tomaž Kosar, Marjan Mernik, Julien Cervelle, Rémi Forax, Gilles Roussel

331 **Subtree Matching by Pushdown Automata**
Tomáš Flouri, Jan Janoušek, Bořivoj Melichar

359 **A Tool for Modeling Form Type Check Constraints and Complex Functionalities of Business Applications**
Ivan Luković, Aleksandar Popović, Jovo Mostić, and Sonja Ristić

EDITORIAL

Due to a considerably increasing interest of the authors for ComSIS in recent two years, we may say that this seventh year of publishing is a challenging one for a whole Editorial Board in many aspects. This is the first year in which we decided to publish two extra special issues, apart from two regular ones. The one of them, titled *Advances in Languages, Related Technologies and Applications*, is in front of you. We created it with a hope to provide exchanging useful research ideas and experiences in the area of programming and domain specific languages, and compilers.

On behalf of the ComSIS Consortium, let us use this opportunity to give our thanks to the reviewers and all of the authors for their high-quality work and remarkable enthusiasm. Above all, let us give great thanks to the guest editors, António Menezes Leitao and Boštjan Slivnik, who invested a lot of their efforts in creation of this special issue, as well as to Maria Ganzha, Marcin Paprzycki (WAPL'2009), and Pedro Rangel Henriques (CoRTA'2009), for their fruitful organizational support.

Mirjana Ivanović,
Editor-in-Chief

M. Ivanović

Ivan Luković,
Vice-Editor-in-Chief

[illegible]

GUEST EDITOR’S MESSAGE

Programming languages are the most fundamental form of expression for programmers. Compilers are, therefore, the most important tools to turn into action the ideas expressed in a programming language. At the same time, the fields of Computer Science and Software Engineering are becoming broader every day. New programming languages must support, in the most simple and economic way possible, the expression of new ideas in Computer Science and of ever more complex designs in Software Engineering. Likewise, compilers should be able to effectively implement new programming language constructs, both from the programmer’s as well as from the final user’s point of view.

This special issue contains revised and expanded versions of the best papers presented either at the Conference on Compilers, Programming Languages, Related Technologies and Applications (CoRTA’2009) or at the Workshop on Advances in Programming Languages (WAPL’09). The former, organized by the Faculty of Sciences of the University of Lisbon, took place on September 10-11, 2009 in Lisbon, Portugal. The latter was a workshop of an International Multiconference on Computer and Information Technology (IMCSIT’2009) which was organized by Polish Information Processing Society and took place in Mragowo, Poland on October 12–14, 2009.

But just as one of the events took place near the geographical center of Europe and the other took place at the continent’s extreme point where centuries ago European expansion around the globe started, so the papers of this special issue address, on one hand, the most central questions in contemporary research and, on the other hand, extend them to the edge of research.

As the names of both events suggest, they shared a similar scope. The list of suggested topics included, among others
- compiling techniques,
- domain-specific languages,
- programming language concept design,
- formal techniques and tools, and
- automata theory and applications.

The program committees of both events were pleased to observe that theoretical and practical papers were submitted, providing them the opportunity to produce a balanced selection of both kinds of papers.

At the end of both events, the best papers submitted were selected for possible publication in a special issue of ComSIS and their authors were invited to prepare extended versions of their papers. These extended versions were then reviewed by experts in the field. In order to ensure the best possible quality, the improved papers were then submitted to a second round of reviewing.

The paper Comparing General-Purpose and Domain-Specific Languages: An Empirical Study by Tomaž Kosar, Nuno Oliveira, Marjan Mernik, Maria João Varanda Pereira, Matej Črepinšek, Daniela da Cruz, and Pedro Rangel Henriques, describes an experiment that was carried out to clarify whether domain-specific languages have significant advantages over general-purpose languages for the construction of graphical user interfaces. The comparison between languages was done using the Cognitive Dimension Framework and the experiment shows that domain-specific languages are indeed advantageous, thus settling an important question in the programming language research area.

The paper VisualLISA: A Visual Environment to Develop Attribute Grammars by Nuno Oliveira, Maria João Varanda Pereira, Pedro Henriques, Daniela da Cruz, and Bastian Cramer, discusses the design of a new visual language for attribute grammars and the development of the associated programming environment. The author’s solution is based on the use of DEVil, a system that generates a visual programming environment from high-level specifications. This solution is a relevant contribution that combines concepts from the areas of Program Comprehension, Attribute Grammars and Visual languages.

Jaroslav Porubän, Michal Forgáč and Miroslav Sabo contributed a paper entitled Annotation Based Parser Generator. It describes a parser generator which focuses on the abstract grammar: the parser is generated from the annotations associated with Java classes specifying the abstract syntax. Hence, by simply changing annotations one can produce parsers for different concrete grammars that all correspond to the same abstract syntax.

The paper On Automata and Language Based Grammar Metrics by Matej Črepinšek, Tomaž Kosar, Marjan Mernik, Julien Carvelle, Rémi Forax and Gilles Roussel concentrates on grammar metrics. In other words, how programming languages can be compared based on their syntactic structure and what a programming language designer can learn from it.

Another paper, Subtree Matching by Pushdown Automata by Tomas Flouri, Jan Janousek, and Borivoj Melichar, is about a subtree matching, a problem which often appears in compiler technology as trees are widely used for internal representations of all kinds. In this particular case, the problem has been solved using a pushdown automaton, another formalism known well to the compiler writers.

Finally, the paper A Tool for Modeling Form Type Check Constraints and Complex Functionalities of Business Applications, by Ivan Luković, Aleksandar Popović, Jovo Mostić, and Sonja Ristić, is about a domain specific language for specifying check constraints and a tool that enables visually oriented design and parsing check constraints.

As guest editors, we would like to thank the authors for their valuable contribution to this special issue, and the referees for their dedicated work and high-quality remarks that helped improve the papers. Furthermore, we would like to thank the members of both program committees as they have carried out the first and thus the most painful selection of submitted papers, and to the organizers of both events, namely CoRTA'2009 and WAPL'09 and, by extension, to the organizers of the main conferences, namely, INForum'2009 and IMCSIT'2009.

Finally, we are grateful to Prof. Ivan Luković, Vice Editor-in-Chief of ComSIS, who provided unlimited support and assistance during the selection and reviewing process that culminated in this special issue.

Boštjan Slivnik
António Menezes Leitao
Guest Editors

UDC 004.438, DOI: 10.2298/CSIS1002247K

Comparing General-Purpose and Domain-Specific Languages: An Empirical Study

Tomaž Kosar[1], Nuno Oliveira[2], Marjan Mernik[1], Maria João Varanda Pereira[3], Matej Črepinšek[1], Daniela da Cruz[2], and Pedro Rangel Henriques[2]

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

Abstract. Many domain-specific languages, that try to bring feasible alternatives for existing solutions while simplifying programming work, have come up in recent years. Although, these little languages seem to be easy to use, there is an open issue whether they bring advantages in comparison to the application libraries, which are the most commonly used implementation approach. In this work, we present an experiment, which was carried out to compare such a domain-specific language with a comparable application library. The experiment was conducted with 36 programmers, who have answered a questionnaire on both implementation approaches. The questionnaire is more than 100 pages long. For a domain-specific language and the application library, the same problem domain has been used – construction of graphical user interfaces. In terms of a domain-specific language, XAML has been used and C# Forms for the application library. A cognitive dimension framework has been used for a comparison between XAML and C# Forms.

Keywords: domain-specific languages; general-purpose languages; program comprehension; empirical software engineering.

1. Introduction

The primary goal in developing a new programming language is to make programming more efficient. The perfect programming language should provide the right level of abstraction, meaning that it describes solutions naturally and hides unnecessary details. Also, it should be expressive enough in the problem domain and should provide guarantees on properties that are critical for the problem domain. It should also have precise semantics to enable formal reasoning about a program. With general-purpose languages (GPLs), this is difficult to achieve, since GPLs tend to be general, resulting in poor support for domain-specific notation. On the other hand, domain-specific languages (DSLs) can be designed in many problem domains to exactly have properties mentioned above. A DSL is a language that is tailored to a specific application domain that offers appropriate notations and abstractions [1]. For a domain in question, DSLs are more expressive and are easier to use than GPLs, with gains in productivity and maintenance costs [2 - 4].

GPLs are perfectly established in the life-cycle of software development. Their characteristics are widely spread amongst software engineers. On the other hand, the integration of DSLs into the software development life-cycle is not so smooth [5]. However, many DSL studies during the last ten years [1, 3, 6 – 12], reveal the importance of these languages in software engineering. The concentration on a definition of notation that would only express concepts of a single application domain, brings the possibility to sharpen the edges of a language, which makes it more and more *efficient* in various directions, which are briefly elaborated below. One of these directions is the efficiency of being read and learned by the domain experts [13]. To use DSLs that allow focusing on the problem and not on the solution, can be profitable at earlier stages of the software life-cycle as well [14], such as requirements analysis and management [15]. Moreover, there is the possibility of integrating domain experts in the later stages of the software development life-cycle [2, 16]. Since the usage of GPLs requires good programming skills, the domain-experts, who are not proficient in that area, can do very little on this matter. However, with the use of DSLs, they can concentrate on the programming tasks and they can even do programming. Another benefit of DSLs is that software maintenance is simplified [2], since DSLs provide self-documentation that avoids the search for documentation resources, which may be unavailable in the first place. DSLs are also claimed to be a good approach for software reuse [17]. In this context, not only the pieces of software are reused, but also the knowledge embodied in the language. Another facet of efficiency can be observed in the tools that give support to a language. Their processors, for instance, can be improved to offer better results, as the domain is restricted and the knowledge is centralized [18, 19]. All together, these aspects diminish the costs of engineering and reengineering, and increase reliability and maintainability of the software constructed with DSLs [20].

Although, DSLs have proven their usefulness, GPLs together with application libraries (APIs) are still the most commonly used programmer's choice when preparing new solutions for their problems. One of the reasons that DSLs are not accepted among the practitioners is the lack of DSLs' promotion. Further, studies that would point out the benefits of DSL over GPL solution are rare. In this paper, we will use the cognitive dimension framework (CDF) [21, 22, 23] to compare DSL and GPL programs and to expose properties that are enhanced in the context of DSLs. The goal of the project[1] is to measure how easy it is to understand programs written in DSLs compared to GPLs. In this manner, the experiment is conducted with the use of questionnaires to measure programmers' understanding of DSL and GPL programs on the same problem domain, a construction of graphical user interfaces (GUIs). More precisely, with these questionnaires, we attempt to confirm that DSL programs are easier to understand than GPL programs. This hypothesis is defended with an experiment in a controlled environment, using direct observations of the experiment evaluation model involving CDF.

The organization of this paper is as follows. Related work on the preparation of an experiment and CDF is discussed in Section 2. The experiment skeleton, the identification of its main goals, and experiment details are introduced in Section 3. The experiment results, with the cognitive dimension framework, are given in Section 4. Concluding remarks are summarized in Section 5.

2. Related work

This work can be classified within the category of empirical software engineering. Empirical research in software engineering is an important discipline that shows practical results on how practitioners (developers, end-users) come to accept and use technologies, techniques, etc. In order to avoid questionable results and to have an option to repeat the research, giving the same results, experiments must be prepared with caution. One of the most well known frameworks for software experiments is described in [24]. This framework concentrates on building the knowledge concerning the context of an experiment and is based on organizing sets of related studies (family of studies). Such studies contribute to common hypotheses, which do not vary for individual experiments. In order to prepare this experiment we have followed guidelines from a framework [24]. We have also defined: context of the study, experiment hypothesis, comparison validity, and measurement framework.

Teaching environments give us an opportunity to conduct experiments in computer science programs as well. However, a lot of concerns are connected with the accuracy of results in such environments and several threats to the validity of experiments have to be identified as well as to interpret the results correctly. For those who are interested to read more about this topic, a checklist for integrating empirical studies in teaching activities can be found in the work [25].

As stated above, an important step in the experiment preparation is to set down the measurement framework – how the results of an experiment are evaluated and interpreted. In cognitive theory, guidelines on how to measure

[1]This work is sponsored by bilateral project "Program Comprehension for Domain-Specific Languages" (code BI-PT/08-09-008) between Slovenia and Portugal.

a human's ability to program are defined. CDF [21] provides cognitively-relevant aspects which can be used to determine how easy it is to understand a program. In our study, CDF is used to compare user understanding of DSL and GPL programs. In the past the CDF has been used to assess the usability of visual programming languages [26, 27] and spreadsheets [28].

Recently, another application for cognitive dimensions can be found in [29], where a method for designing Framework-Specific Modeling Languages (FSMLs) is presented. From FSML specifications, a user can build applications based on object-oriented frameworks. In FSML software, artifacts (models, languages, etc.) are evaluated according to their goals with different quality methods. Particularly, the quality of notation is measured with cognitive dimensions – a heuristic measure that evaluates the notation and its environment.

Before this experiment, the authors of the paper were involved in another similar experiment [30]. That work is important for an interested reader, since the information on experiment skeletons is described in great detail. Difference between both experiments is in the hypothesis and exclusion/inclusion of CDF. Also, the problem domain in experiment [30] is different (graph description with DOT language [31]) than in this paper (construction of GUI with XAML).

This paper is also closely related to the field of Program Comprehension, which is a hard cognitive task, done by a software analyst. In the process of program comprehension, the use of tools to interconnect different views (operational, behavioral, etc.) to understand the results of applications, are indispensable. Traditional techniques on program comprehension from GPLs (visualizers, animators, etc.) have been studied and applied to DSLs in our previous work [32], where CDF was also briefly described and applied to DSLs.

3. Presentation of experiment

In this section the preparation, execution, and experiment evaluation model is given.

3.1. Objective of the experiment

In [3] the empirical results that compare ten diverse implementation approaches for DSLs, conducted on the same representative language, are provided. Among the implementation approaches, the comparison also included the XML-based approach. From this study, it can be concluded that XML-based approach has some disadvantages [3]. Although, XML usage and its tool support are spreading, this is one of the reasons that XAML [33], as a representative DSL, has been chosen for this study. XAML, the Extensible Application Markup Language, is a language for construction of graphical user interfaces in Windows Presentation Foundation and Silverlight applications of .NET Framework 3.5. C# Forms [34] has been used for the comparison since it covers the same domain of graphical user interfaces.

3.2. Hypothesis of the experiment

In order to perform the comparison on XAML and C# Forms, two separate questionnaires have been prepared. The study that was carried out had a task to observe programmers' efficiency on understanding programs with both approaches, compare the results obtained through questionnaires and use them to investigate the following hypothesis:

$H1_{null}$

There is no significant difference in program understanding between domain-specific or general-purpose languages, when using XAML or C# Forms for comparison.

$H1_{alt}$

There is a significant difference in program understanding between domain-specific or general-purpose languages when using XAML instead of C# Forms.

This hypothesis is the object of investigation in the conducted experiment and is further examined in the Section 4, where the questionnaires results are presented.

3.3. Preparation of experiment

The results from an experiment are reliable if the repetition of the experiment can be proven [35]. Repetition is strongly connected to agreements set down before the start of the experiment [24]. Therefore, some rules and constraints were defined for the questionnaire implementers:

– the same group of questions for both experiments on a GPL and a DSL must be used,

– the questions for two applications on the same question were prepared (easier and harder application domain),

– the equal questions in DSL and GPL questionnaires must be defined by the same number of components in order to obtain the same level of question complexity, and

– the questions and the given choices (programs) must be reviewed by other domain experts, to obtain a code as optimal as possible.

As stated above, two questionnaires have been prepared for program understanding of DSL and GPL programs. Then, the structure of questionnaires has been defined to cover the following three topics of program understanding: learn, perceive, and evolve. In the first group, questions on learning notation and meaning of programs have been given to the programmers. In the second group, questions on program perceiving have been defined, such as identification of correct meaning from a given program, language constructs, new construct meaning, and meaning of a program with given comments. In the third group, programmers had been challenged to expand/remove/replace program functionality.

For these three groups, 11 questions have been defined:

– Learn
• Q1 Select syntactically correct statements.
• Q2 Select program statements with no sense (unreasonable).
• Q3 Select a valid program with the given result.
– Perceive
• Q4 Select a correct result for the given program.
• Q5 Identify language constructs.
• Q6 Select a program with the same result.
• Q7 Select a correct meaning for the new language construct.
• Q8 Identify language constructs in the program with comments.
– Evolve
• Q9 Expand the program with new functionality.
• Q10 Remove functionality from the program.
• Q11 Change functionality in the program.

Learning and perceiving questions have been defined as a multiple-choice question, and questions under evolve have been defined as an essay question (programmers are challenged to modify existing code). Both, XAML and C# Forms questionnaires have been constructed with the use of the above questions.

To illustrate the style of the questions, used in the questionnaires, an example is presented in Figure 1. Because of the question size only the correct choice is given. Complete questionnaires can be found on a project group webpage[2]. The above questions (Q1-Q11) have been used as templates to define DSL and GPL questionnaires. The first version of DSL and GPL questionnaires has been given to a small group as a training set, in order to receive feedback. Further, results from a training set have been studied and used to refine the questions before applying them to the target groups of students. Most correctness issues were related to the program failures, question understandability and question complexity comparability between DSL and GPL questions.

3.4. Execution of the experiment

Besides well-structured questionnaires, other factors have also been controlled in the experiment. The list of experiment execution actions in the classroom (just before starting, as soon as it begins, and during the experiment) are provided below:

– a short tutorial, for end-users, has been given on the problem domain (graphical user interfaces),

[2] http://epl.di.uminho.pt/~ gepl/DSL/

– a tutorial on domain specific notation (XAML), together with an example of a program, has been given to end-users,

– a tutorial on application library (C# Forms) together with an example of the program has been given to end-users,

– a tutorial has been given to end-users in their native language, however the slides, programs and experiment questionnaires were in English, and

– the slides and the examples have been given to end-users and could be used during the experiment.

If necessary, individual help to better understand the questions was provided to the programmers.

3.5. Processing the results

There were also issues that needed to be controlled, after the programmers completed the questionnaire. One of those is submission completeness. When students submitted their questionnaires, it was checked if the questionnaire contained answers to all questions. Most of the programmers answered the questions, however if some answers were missing, the programmers were advised to complete the questionnaire. Still, if some answers were found missing during the processing of the results, the complete programmer questionnaire was eliminated from the further experiment analysis. For previously mentioned reasons, one submission has been eliminated from the results.

3.6. Threats to validity

In each experiment, there are several threats to the validity of results. Those threats need to be identified and handled before the start of the experiment. To restrict the impact of the experiment environment on the results, the following issues have been identified for our study.

Chosen domain Results of the experiment are strongly connected to programmers' experiences and knowledge of the chosen problem domain. In Table 1, programmers' familiarity with the construction of the GUI is presented, together with the experience on XAML and C# Forms library application. From Table 1, we can conclude that programmers are experienced in the construction GUIs domain. However, their experience in implementation technique differs – programmers were unfamiliar with XAML on one hand (median value 1), and had good knowledge in constructing GUIs with C# Forms (median value 4) on the other. Uneven knowledge on both notations could have made an influence on comparison results.

Programmers' experience In Table 2, results from the self evaluation test are presented, where students (second year of undergraduate computer science) grade their general knowledge on programming, programming in C# language and prior experience with DSLs. Comparing knowledge on C# (median value 4) and prior experience with DSLs (median value 2) could have also made an influence on experiment results.

Table 1. Programmers' knowledge in construction of graphical user interfaces (N = 36)

	Average[3]	Median	St. dev.
Familiarity GUI domain	3.39	4	1.18
Knowledge of XAML	1.36	1	0.68
Knowledge of C# Forms application library	3.5	4	1.11

Table 2. Programmers experiences in programming (N = 36)

	Average	Median	St. dev.
Skills in programming	3.41	3.5	0.65
Skills of programming in C#	3.53	4	0.74
Prior experience with DSLs	2.28	2	0.70

Comparability of questionnaires The same type of questions in DSL and GPL questionnaires contain a similar number of graphical components (labels, text fields, buttons, etc), to obtain the same level of complexity.

Order of questionnaires An experiment on program understanding was carried out twice, at different times and on different students. The first group started the experiment with the questionnaire on DSL and proceeded with a GPL questionnaire. The second group started the questionnaire on GPL and finished with the DSL questionnaire. In such a way, the influence of starting the experiment on the same questionnaire with all subjects was avoided and with such, the order of questionnaires is not relevant for the outcome of the study.

4. Results

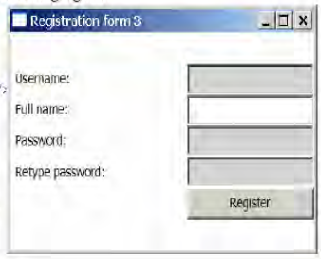
All together, programmers answered 22 questions on both questionnaires. Success rate for questions varied from 27.14% for Q6 to 79.73% for Q9 (Table 3). Differences in success rate in the same language (DSL/GPL) can be explained with different difficulty level (some questions were harder than

[3] A five-grade scale, starting from very bad (1) to very good (5) was used for self-evaluation questionnaires (in Tables 1 and 2). Note, that column "Average" shows the average value given by 36 programmers, "Median" stands for middle value in set of programmers grades and "St. dev." represents standard deviation on given grades.

Question 5

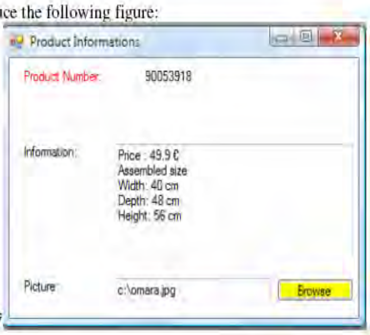
QL031 XAML-DSPL-RegistrationForm: Select program for the following figure:

```
<Window x:Class="WpfRegistration.Registration3"
  Title="Registration form 3" Height="200" Width="300">
  <Grid ShowGridLines="False">
    <Grid.ColumnDefinitions>
     <ColumnDefinition Width="12*"/> <ColumnDefinition Width="2*"/>
     <ColumnDefinition Width="10*"/> </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
   <RowDefinition Height="10*"/> <RowDefinition Height="10*"/>
   <RowDefinition Height="10*"/> <RowDefinition Height="10*"/>
   <RowDefinition Height="10*"/> <RowDefinition Height="10*"/>
   <RowDefinition Height="10*"/> </Grid.RowDefinitions>
   <Label   Grid.Column="0" Grid.Row="1"> Username:</Label>
   <TextBox Grid.Column="2" Grid.Row="1" Background="LightGray"/>
   <Label   Grid.Column="0" Grid.Row="2"> Full name:</Label>
   <TextBox Grid.Column="2" Grid.Row="2"/>
   <Label   Grid.Column="0" Grid.Row="3"> Password:</Label>
   <TextBox Grid.Column="2" Grid.Row="3" Background="LightGray"/>
   <Label   Grid.Column="0" Grid.Row="4"> Retype password:</Label>
   <TextBox Grid.Column="2" Grid.Row="4" Background="LightGray"/>
   <Button  Grid.Column="2" Grid.Row="5"> Register</Button> </Grid> </Window>
```



Question 5

QL031 WFORM-GPL Select the correct program to produce the following figure:


Product Number: 90053918

Information: Price : 49,9 €
Assembled size
Width: 40 cm
Depth: 48 cm
Height: 56 cm

Picture: c:\omara.jpg

```
private Label labelNumber, labelFilledNumber;
private Label labelInf, labelPic;
private TextBox textBoxInfo, textBox1;
private Button btnBrowse;

private void InitializeComponent() {
  labelNumber = new Label();
  labelFilledNumber = new Label();
  textBoxInfo = new TextBox();
  labelInf = new Label(); labelPic = new Label();
  btnBrowse = new Button(); textBox1 = new TextBox();
  labelNumber.ForeColor = System.Drawing.Color.Red;
  labelNumber.Location = new System.Drawing.Point(12, 9);
  labelNumber.Name = "labelNumber";
  labelNumber.Size = new System.Drawing.Size(119, 17);
  labelNumber.TabIndex = 5;
  labelNumber.Text = "Product Number: ";
  labelFilledNumber.Name = "labelFilledNumber";
  labelFilledNumber.Size = new System.Drawing.Size(72, 17);
  labelFilledNumber.Text = "90053918";
  labelFilledNumber.Location = new System.Drawing.Point(137, 9);
  textBoxInfo.Location = new System.Drawing.Point(109, 74);
  textBoxInfo.Multiline = true; textBoxInfo.Name = "textBoxInfo";
  textBoxInfo.Size = new System.Drawing.Size(246, 97);
  textBoxInfo.Text = "Price : 49,9 €\r\nAssembled size\r\nWidth: 40 cm\r\nDepth: 48 cm\r\nHeight: 56 cm";
  labelInf.Location = new System.Drawing.Point(12, 74);
  labelInf.Name = "labelInf";
  labelInf.Size = new System.Drawing.Size(82, 17);  labelInf.Text = "Information:";
  labelPic.Location = new System.Drawing.Point(12, 190);
  labelPic.Name = "labelPic";
  labelPic.Size = new System.Drawing.Size(56, 17);  labelPic.Text = "Picture:";
  btnBrowse.BackColor = System.Drawing.Color.Yellow;
  btnBrowse.Location = new System.Drawing.Point(276, 190);
  btnBrowse.Name = "btnBrowse";
  btnBrowse.Size = new System.Drawing.Size(79, 20);  btnBrowse.Text = "Browse";
  btnBrowse.UseVisualStyleBackColor = false;
  textBox1.Location = new System.Drawing.Point(109, 190);
  textBox1.Name = "textBox1";
  textBox1.Size = new System.Drawing.Size(161, 20);
  Controls.Add(textBox1); Controls.Add(btnBrowse); Controls.Add(labelPic); Controls.Add(labelInf);
  Controls.Add(textBoxInfo); Controls.Add(labelFilledNumber); Controls.Add(labelNumber);
}
```

Fig. 1. Question 5 in DSL and GPL questionnaires with the correct choice others). On the other hand the biggest difference between GPL and DSL is 51.16% in the case of Q9. The smallest difference is found in Q2, where the difference is just 3.44%. In this case, the success rate was even slightly better for GPL than DSL. In our opinion, this is due to the difficultness of Q2 (success rate was less than 39%), where syntactically correct programs with no sense have to be identified. Since programmers have more experience in C# Forms than XAML (Table 1), they were more successful with GPL than DSL, in finding programs with no sense.

Table 3. Average programmer success rate (N = 35)

Question	DSL	GPL	Difference
	XAML	C# Forms	
Q1	72.97%	48.57%	24.4%
Q2	35.14%	38.57%	-3.44%
Q3	64.86%	35.71%	29.15%
Q4	77.03%	70.00%	7.03%
Q5	64.86%	48.57%	16.29%
Q6	39.19%	27.14%	12.05%
Q7	75.68%	62.86%	12.82%
Q8	62.16%	45.71%	16.45%
Q9	79.73%	28.57%	51.16%
Q10	68.92%	41.43%	27.49%
Q11	66.22%	30.00%	36.22%

Table 4. Average programmer success rate on learn, perceive and evolve (N=35)

	Question	DSL XAML			GPL C# Forms		
		Mean	Std. dev.	Std. err. mean	Mean	Std. dev.	Std. err. mean
Learn	Q1, Q2, and Q3	57.62%	21.90%	3.70%	40.95%	22.99%	3.89%
Perceive	Q4, Q5, Q6, Q7, and Q8	64.57%	19.45%	3.29%	50.86%	18.69%	3.16%
Evolve	Q9, Q10, and Q11	70.95%	20.35%	3.44%	33.33%	26.20%	4.43%
Total	All questions	64.34%	14.81%	2.50%	43.37%	15.99%	2.70%

However, drawing conclusions based on an average value of a single question can be extremely risky. Therefore, by grouping questions into learn, perceive and evolve categories, we can obtain more reliable results. In Table 4, the success rate on questions by the individual group is presented with a mean value, standard deviation, and standard error mean. Table 4 confirms our presumption that program understanding, in terms of learn, perceive and evolve, is much better for DSL programs than for GPL programs. Later observation is especially obvious from the results on evolve questions – the mean value of the success rate was 37.62% better for DSL than on GPL questions (see mean values in Table 4). Similar results were also obtained on the other problem domain described in [30]. To support the results in Table 3 and Table 4, statistical tests have been performed to evaluate whether the comparison shows the statistical significant difference. Efficiency on both questionnaires (see last row from Table 4) has been compared for all programmers. The results from questionnaires were statistically tested with t-test, since the means of two independent groups were compared. The threshold for the independent t-test was set to $\alpha = 0.05$ and the test results are shown in Table 5. The most important column in the table is the significance

Table 5. Independent t-test for program understanding (N = 35)

	t	Sig. (two-tailed)	Mean	Std. dev.	Std. err. mean	95% Confidence Interval of the Difference	
						Lower	Upper
XAML vs. C# Forms	5.474	0.000	20.971	22.664	3.831	13.186	28.757

column. Observing this data confirms that the difference in mean value between XAML and C# Forms program understanding was significant, since a significant level was not reached. With observations from Table 5, we could reject null hypothesis $H1_{null}$ since subjects did better on XAML program understanding and accept the alternative hypothesis: ($H1_{alt}$): there is a significant difference in program understanding between domain-specific or general-purpose languages when using XAML instead of C# Forms.

While this experiment indeed shows superiority of DSLs on an end-user ability to learn, perceive and evolve programs in this particular domain, it does not provide possible explanations why DSLs programs are easier to understand. The “psychology of programming” [36, 37] is a research field which tries to identify, understand and explain those cognitive processes which take place during reasoning (e.g., programming, program understanding). In this context, CDF [21] provides useful dimensions, which help us to better explain why DSL programs are easier to understand than GPL programs. The CDF has been used before to assess the usability of visual programming languages [26], while no such study exists for DSLs. These cognitive dimensions are:

– Closeness of mapping – languages should be task-specific;
– Viscosity – revisions should be painless;
– Hidden dependencies – the consequences of changes should be clear;
– Hard mental operations – no enigmatic is allowed;
– Imposed guess-ahead – no premature commitment;
– Secondary notation – allow to encompass additional information;
– Visibility – search trails should be short;
– Consistency – user expectations should not be broken;

– Diffuseness – language should not be too verbose;
– Error-proneness – notation should catch mistakes avoiding errors;
– Progressive evaluation – get immediate feedback;
– Role expressiveness – see the relations among components clearly;
– Abstraction gradient – languages should allow different abstraction levels.

The next step was to connect cognitive dimensions with our questions. We identified which dimensions are relevant for a particular question (Table 6). As it can be seen D_i (dimension i of CDF) can be related to several questions used in our questionnaires.

Table 6. Questions connection to cognitive dimensions

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
Closeness of mapping	1	1	1	1	1	1	1	1	1	1	1
Viscosity	0	0	0	0	0	0	0	0	1	1	1
Hidden dependencies	0	0	1	1	1	1	0	1	0	1	0
Hard mental operations	0	1	1	1	1	1	1	1	0	0	0
Imposed guess-ahead	0	0	0	0	0	0	0	0	1	0	1
Secondary notation	0	0	0	0	0	0	0	1	0	0	0
Visibility	0	0	1	1	1	1	1	1	0	0	0
Consistency	0	0	0	0	0	1	1	0	0	0	0
Diffuseness	1	1	1	1	1	1	1	1	1	1	1
Error-proneness	1	1	1	1	1	1	1	1	1	1	1
Progressive evaluation	0	0	0	0	0	0	0	0	0	0	0
Role expressiveness	0	1	1	1	1	1	1	1	1	1	1
Abstraction gradient	0	0	1	1	1	1	1	1	0	0	0

Questions have been designed in such a way that they directly reflect cognitive dimensions as much as possible. However, not all cognitive dimensions play an important role in all questions. Hence, to evaluate a single cognitive dimension (D_i) we proposed the following formula:

$$D_i = \sum_{j=1}^{11} Q_{ij} * \frac{S_j}{C_j}$$

where Q_{ij} stands for the value from Table 6, which means whether dimension D_i is connected to the question Q_j. Variable S_j represents an average programmer's success rate on question Q_j (Table 3). For example, if 4 programmers out of 5 answered question Q_1 correctly, the value of S_1 would be 0.8. Finally, C_j represents the number of cognitive dimensions relevant for Q_j (for example, C_1 = 3). This formula is used for XAML as well as for C# Forms. Intuitively, it means that cognitive dimensions contribute to the success of a particular question. Here, we assume that contribution of involved cognitive dimensions was equally distributed (one cognitive dimension is not more important than the other, if it is involved). Moreover, we assume that the higher values always mean a positive influence of particular cognitive dimension. For example, higher values for 'closeness of mapping' mean that the semantic gap between the problem and the solution space is small, or higher values for 'hidden dependencies' mean that short and long-range interactions among program components are immediately visible.

Table 7 roughly shows how a particular cognitive dimension contributes to the questionnaires' success for XAML, as well as for C# Forms. From Table 7 it can be seen that in our experiment, the most influential for DSL/GPL program understanding were: closeness of mappings, diffuseness, error-proneness, role expressiveness, and hard mental operations. More than particular values, the difference among cognitive dimensions for XAML and C# Forms is far more important. The biggest difference among cognitive dimensions was in closeness of mappings, diffuseness, error-proneness, role expressiveness, and viscosity.

Table 7. Influence of cognitive dimension to XAML and C# Forms

	DSL	GPL	Difference
	XAML	C# Forms	
Closeness of mapping	1.127	0.749	0.377
Viscosity	0.442	0.237	0.206
Hidden dependencies	0.486	0.343	0.143
Hard mental operations	0.525	0.421	0.105
Imposed guess-ahead	0.243	0.098	0.146
Secondary notation	0.069	0.051	0.018
Visibility	0.455	0.344	0.111
Consistency	0.128	0.100	0.028
Diffuseness	1.127	0.749	0.377
Error-proneness	1.127	0.749	0.377
Progressive evaluation	N/A	N/A	N/A
Role expressiveness	0.884	0.587	0.296
Abstraction gradient	0.455	0.344	0.111

Closeness of mapping refers to the width of the semantic gap between the problem and the solution spaces. Diffuseness refers to the number of symbols needed to express the meaning. By definition, DSLs use existing domain notation, which should be at an appropriate level of verbosity, so it is expected that they exhibit low diffuseness. On the other hand, it was shown in [38] that plenty of low-level primitives, which are often purely syntactical, are one of the biggest cognitive barriers for end-user programmers. Error proneness refers to the capability of a language to induce careless mistakes. GPLs, due to their extension and intrinsic complexity, are usually error-prone, while DSLs, due to the narrow domain they are designed for, are usually less error prone. Role expressiveness refers to the ability to see how each component of a program relates to the whole. The high role expressiveness can be more easily achieved in DSLs due to domain specifics and shorter programs. It is shown in our experiment that differences in closeness of mapping, diffuseness, error proneness, and role expressiveness among

XAML and C# Forms are the biggest and the source of main contribution for easier understanding of XAML programs than programs written in C# Forms.

Viscosity refers to the amount of effort that is needed to perform small changes. Since DSLs are usually at a high abstraction level and have natural notation, small changes should be easier to perform. It is shown in our experiment that the difference in viscosity between XAML and C# Forms was among the largest. Viscosity was involved only in questions Q9-Q11, which were much better solved with the use of XAML than using C# Forms. We can conclude that viscosity had an important influence on this success.

5. Conclusion and future work

The purpose of this paper is to promote formal studies on the advantages of DSLs over GPLs. In this paper we have tried to explain the difference between DSL/GPL program understanding, using the cognitive dimension framework. Questionnaires on understanding programs have been prepared and given to the programmers. Each programmer answered a 100 page long questionnaires and on an average spent more than 3 hours solving 44 questions.

Results show that programmers' success rate was around 15% better for DSL in all three groups of questions: learn, perceive and evolve, despite the fact that programmers were significantly less experienced in XAML than C# Forms. Further, the experiment measurement framework included cognitive dimensions to identify the aspects among these dimensions that are enhanced in the context of DSL. It can be learned from the study that DSLs are superior to GPLs in all cognitive dimensions. The cognitive dimensions, with the biggest influence in the experiment, are closeness of mappings, diffuseness, error-proneness, role expressiveness, and viscosity.

We consider that the results of this experiment are reliable despite the fact that the experiment has been done only on a single domain. One of the future tasks of this project is to conduct similar experiments in different domains.

References

1. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys. 37(4) (December 2005) 316–344
2. Deursen, A.v., Klint, P.: Little languages: Little maintenance? Journal of Software Maintenance 10 (1998) 75–92
3. Kosar, T., Martínez López, P.E., Barrientos, P.A., Mernik, M.: A preliminary study on various implementation approaches of domain-specific language. Information and Software Technology 50(5) (2008) 390–405
4. Živanov, v., Rakić, P., Hajduković, M.: Using code generation approach in developing kiosk applications. Journal on Computer Science and Information Systems 5(1) (2008) 41–59

5. Sprinkle, J., Mernik, M., Tolvanen, J-P., Spinellis, D.: What Kinds of Nails Need a Domain-Specific Hammer? IEEE Software, 26(4) (2009) 15–18
6. Wile, D.S.: Supporting the DSL spectrum. Journal of Computing and Information Technology 9(4) (2001) 263–287
7. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. ACM SIGPLAN Notices 35 (2000) 26–36
8. Hudak, P.: Building domain-specific embedded languages. ACM Computing Surveys 28(4) (June 1996) 196–202
9. Hudak, P.: Modular domain specific languages and tools. In: ICSR ’98: Proceedings of the 5th International Conference on Software Reuse, Washington, DC, USA, IEEE Computer Society (1998)
10. Kieburtz, R.B., Mckinney, L., Bell, J.M., Hook, J., Kotov, A., Lewis, J., Oliva, D.P., Sheard, T., Smith, I., Walton, L.: A software engineering experiment in software component generation. In: ICSE ’96: Proceedings of the 18th international conference on Software engineering, Washington, DC, USA, IEEE Computer Society (1996) 542–552
11. Sirer, E.G., Bershad, B.N.: Using production grammars in software testing. In: Proceedings of the 2nd conference on Domain-specific languages, New York, NY, USA, ACM (1999) 1–13
12. Kolovos, D.S., Paige, R.F., Kelly, T., Polack, F.A.C.: Requirements for domain-specific languages. In: Proc. 1st ECOOP Workshop on Domain-Specific Program Development (DSPD 2006), Nantes, France (July 2006)
13. Consel, C., Latry, F., Réveillère, L., Cointe, P.: A generative programming approach to developing DSL compilers. In Gluck, R., Lowry, M., eds.: Fourth International Conference on Generative Programming and Component Engineering (GPCE). Volume 3676 of Lecture Notes in Computer Science, Tallinn, Estonia, Springer-Verlag (September 2005) 29–46
14. Jackson, M.: Problem frames and software engineering. Information and Software Technology 47(14) (November 2005) 903–912
15. Aurum, A., Wohlin, C.: Engineering and Managing Software Requirements. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2005)
16. Gray, J., Fisher, K., Consel, C., Karsai, G., Mernik, M., Tolvanen, J.P.: DSLs: the good, the bad, and the ugly. In: OOPSLA Companion ’08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, New York, NY, USA, ACM (2008) 791–794
17. Krueger, C.W.: Software reuse. ACM Computing Surveys 24(2) (June 1992) 131–183
18. Javed, F., Mernik, M., Bryant, B., Sprague, A.: An unsupervised incremental learning algorithm for domain-specific language development. Applied Artificial Intelligence 22(7) (2008) 707-729
19. Wu, H., Gray, J., Mernik, M.: Grammar-driven generation of domain-specific language debuggers. Software Practice and Experience 38(10) (2008) 1073-1103
20. Herndon, R.M., Berzins, V.A.: The realizable benefits of a language prototyping language. IEEE Transactions on Software Engineering 14(6) (1988) 803–809
21. Green, T., Petre, M.: Usability analysis of visual programming environments: a ”cognitive dimensions” framework. Journal of Visual Languages and Computing 7(2) (1996) 131–174
22. Blackwell, A., Britton, C., Cox, A., Green, T.R.G., Gurr, C., Kadoda, G., Kutar, M., Loomes, M., Nehaniv, C., Petre, M., Roast, C., Roe, C., Wong, A., Young, R.: Cognitive dimensions of notations: Design tools for cognitive technology. In: Cognitive Technology: Instruments of Mind. Springer-Verlag (2001) 325–341

23. Green, T.R.G., Blandford, A.E., Church, L., Roast, C.R., Clarke, S.: Cognitive dimensions: achievements, new directions, and open questions. Journal of Visual Languages & Computing 17(4) (August 2006) 328–365
24. Basili, V., Shull, F., Lanubile, F.: Building knowledge through families of experiments. IEEE Transactions on Software Engineering 25(4) (1999) 456–473
25. Carver, J., Jaccheri, L., Morasca. S., Shull. F.: A checklist for integrating student empirical studies with research and teaching goals. Empirical Software Engineering 15(1) 2010 35-59
26. Blackwell, A.: Ten years of cognitive dimensions in visual languages and computing: Guest editor's introduction to special issue. Journal of Visual Languages and Computing 17(4) (2006) 285–287
27. Yang, S., Burnett, M., DeKoven, E., Zloof, M.: Representation design benchmarks: a design-time aid for VPL navigable static representations. Journal of Visual Languages and Computing 8(5/6) (1997) 563–599
28. Peyton Jones, S., Blackwell, A., Burnett, M.: A user-centred approach to functions in excel. In: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming. (2003) 165–176
29. Antkiewicz, M., Czarnecki, K., Stephan, M.: Engineering of framework-specific modeling languages. IEEE Transactions on Software Engineering 35 (6) (2009) 795–824
30. Kosar, T., Mernik, M., Črepinšek, M., Henriques, P.R., Cruz, D.d., Varanda Pereira, M.J., Oliveira, N.: Influence of domain-specific notation to program understanding. In: Proceedings of 2nd IMCSIT Workshop on Advances in Programming Languages (WAPL'09), Mrągowo, Poland (October 2009) 675 – 682
31. Dot: Graph description language, available at: http://en.wikipedia.org/wiki/DOT language
32. Varanda Pereira, M.J., Mernik, M., Cruz, D.d., Henriques, P.R.: Program comprehension for domain-specific languages. Journal on Computer Science and Information Systems 5(2) (2008) 1–17
33. XAML: Extensible application markup language, available at: http://en.wikipedia.org/wiki/Extensible Application Markup Language
34. C# Forms, available at: http://en.wikipedia.org/wiki/Windows Forms
35. Shull, F., Carver, J., Vegas, S., Juristo, N.: The role of replications in empirical software engineering. Empirical Software Engineering 13(2) (2008) 211–218
36. Weinberg, G.M.: The Psychology of Computer Programming. Van Nostrand Reinhold (1971)
37. A. Blackwell. Psychological issues in end-user programming. In H. Lieberman, F. Paterno, and V. Wulf, editors, End User Development Springer (2006) 9–30
38. Lewis, C., Olson, G.: Can principles of cognition lower the barriers to programming? In: 2nd workshop on Empirical Studies of Programmers. (1987) 248 – 263

Tomaž Kosar received the Ph.D. degree in computer science at the University of Maribor, Slovenia in 2007. His research is mainly concerned with design and implementation of domain-specific languages. Other research interest in computer science include also domain-specific visual languages, empirical software engineering, software security, generative programming, compiler construction, object oriented programming, object-oriented design, refactoring, and unit testing. He is currently a teaching assistant at the

University of Maribor, Faculty of Electrical Engineering and Computer Science.

Nuno Oliveira received, from University of Minho, a B.Sc. in Computer Science (2007) and a M.Sc. in Informatics (2009). He is a member of the Language Processing group at CCTC (Computer Science and Technology Center) , University of Minho. He participated in several projects with focus on Visual Languages and Program Comprehension; VisualLISA and Alma2 the main outcome of his master thesis entitled "Program Comprehension Tools for Domain-Specific Languages", are the most relevant works. The latter came under "Program Comprehension for Domain-Specific Languages", a bilateral project between Portugal and Slovenia, funded by FCT. Currently, he is starting his PhD studies on Patterns for Architectures Coordination Analysis and Self-Adaptive Architectures, under MathIS, a research project also funded by FCT. Meanwhile he is an assistant-lecturer (practical classes) in a course on Imperative Programming, at University of Minho.

Marjan Mernik received the M.Sc. and Ph.D. degrees in computer science from the University of Maribor in 1994 and 1998 respectively. He is currently a professor at the University of Maribor, Faculty of Electrical Engineering and Computer Science. He is also an adjunct professor at the University of Alabama at Birmingham, Department of Computer and Information Sciences. His research interests include programming languages, compilers, grammar-based systems, grammatical inference, and evolutionary computations. He is a member of the IEEE, ACM and EAPLS.

Maria João Varanda Pereira received the M.Sc. and Ph.D. degrees in computer science from the University of Minho in 1996 and 2003 respectively. She is a member of the Language Processing group in the Computer Science and Technology Center, at the University of Minho. She is currently an adjunct professor at the Technology and Management School of the Polytechnic Institute of Bragança, on the Informatics and Communications Department and vice-president of the same school. She usually teaches courses under the broader area of programming: programming languages, algorithms and language processing. But also some courses about project management. As a researcher of gEPL, she is working with the development of compilers based on attribute grammars, automatic generation tools, visual languages and program understanding. She was also responsible for PCVIA project (Program Comprehension by Visual Inspection and Animation), a FCT funded national research project; She was involved in several bilateral cooperation projects with University of Maribor (Slovenia) since 2000. The last one was about the subject "Program Comprehension for Domain Specific Languages".

Matej Črepinšek received the Ph.D. degree in computer science at the University of Maribor, Slovenia in 2007. His research interests include grammatical inference, evolutionary computations, object-oriented programming, compilers and grammar-based systems. He is currently a teaching assistant at the University of Maribor, Faculty of Electrical Engineering and Computer Science.

Daniela da Cruz received a degree in “Mathematics and Computer Science”, at University of Minho (UM), and now she is a Ph.D. student of ”Computer Science” also at University of Minho, under the MAPi doctoral program. She joined the research and teaching team of “gEPL, the Language Processing group” in 2005. She is teaching assistant in different courses in the area of Compilers and Formal Development of Language Processors; and Programming Languages and Paradigms (Procedural, Logic, and OO). As a researcher of gEPL, Daniela is working with the development of compilers based on attribute grammars and automatic generation tools. She developed a completed compiler and a virtual machine for the LISS language (Language of Integers, Sequences and Sets - an imperative and powerful programming language conceived at UM). She was also involved in the PCVIA (Program Comprehension by Visual Inspection and Animation), a FCT funded national research project; in that context, Daniela worked in the implementation of “Alma”, a program visualizer and animator tool for program understanding. Now she is working in the intersection of formal verification (design by contract) and code analysis techniques, mainly slicing.

Pedro Rangel Henriques got a degree in "Electrotechnical/Electronics Engineering", at FEUP (Porto University), and finished a Ph.D. thesis in "Formal Languages and Attribute Grammars" at University of Minho. In 1981 he joined the Computer Science Department of University of Minho, where he is a teacher/researcher. Since 1995 he is the coordinator of the "Language Processing group" at CCTC (Computer Science and Technologies Center).
He teaches many different courses under the broader area of programming: Programming Languages and Paradigms; Compilers, Grammar Engineering and Software Analysis and Transformation; etc. Pedro Rangel Henriques has supervised Ph.D. (11), and M.Sc. (13) thesis, and more than 50 graduating trainingships/projects, in the areas of: language processing (textual and visual), and structured document processing; code analysis, program visualization/animation and program comprehension; knowledge discovery from databases, data-mining, and data-cleaning. He is co-author of the "XML & XSL: da teoria a prática" book, publish by FCA in 2002; and has published 3 chapters in books, and 20 journal papers.

Received: November 15, 2009; Accepted: February 24, 2010.

UDC 004.439, DOI: 10.2298/CSIS1002265O

VisualLISA: A Visual Environment to Develop Attribute Grammars

Nuno Oliveira[1], Maria João Varanda Pereira[2], Pedro Rangel Henriques[1], Daniela da Cruz[1], and Bastian Cramer[3]

[1] University of Minho - Department of Computer Science, Campus de Gualtar, 4715-057, Braga, Portugal
{nunooliveira,prh,danieladacruz}@di.uminho.pt
[2] Polytechnic Institute of Bragança
Campus de Sta. Apolónia, Apartado 134 - 5301-857, Bragança, Portugal
mjoao@ipb.pt
[3] University of Paderborn - Department of Informatics
Fürstenallee 11, 33102, Paderborn, Germany
bcramer@upb.de

Abstract. The focus of this paper is on crafting a new visual language for attribute grammars (AGs), and on the development of the associated programming environment. We present a solution for rapid development of VisualLISA editor using DEViL. DEViL uses traditional attribute grammars, to specify the language's syntax and semantics, extended by visual representations to be associated with grammar symbols. From these specifications a visual programming environment is automatically generated. In our case, the environment allows us to edit a visual description of an AG that is automatically translated into textual notations, including an XML-based representation for attribute grammars ($\mathcal{X}$AGra), and is intended to be helpful for beginners and rapid development of small AGs. $\mathcal{X}$AGra allows us to use VisualLISA with other compiler-compiler tools.

Keywords: Attribute Grammar, Visual Languages, XML Dialect, DEViL, VisualLISA, XAGra.

1. Introduction

An AG can be formally defined as the following tuple: $AG = (G, A, R, C)$, where G is a context-free grammar, A is the set of attributes, R is the set of evaluation rules, and C is the set of contextual conditions. Each attribute has a type, and represents a specific property of a symbol X; we write $X.a$ to indicate that attribute a is an element of the set of attributes of X, denoted by $A(X)$. For each X (terminal or non-terminal), $A(X)$ is divided into two disjoint sets: the *inherited* and the *synthesized* attributes. Each R is a set of formulas, like $X.a = func(..., Y.b, ...)$, that define how to compute, in the precise context of a production, the value of each attribute. Each C is a set of predicates, $pred(..., X.a, ...)$, describing the requirements that must be satisfied in the precise context of a production.

As can be deduced from this complex definition of AGs they are not as easy to specify as people would desire because there is a gap between the problem solution (the desired output) and the source language that must be interpreted. The user must take care on choosing the appropriate attributes and their evaluation rules. Since the beginning, the literature related with compilers presents AGs using syntax trees decorated with attributes. So it is usual to sketch up on paper trees with attributes representing an AG. This strategy allows the developers to imagine a global solution of the problem (in a higher abstraction level) and to detect complex dependencies between attributes, symbols and functions, avoiding spending time with syntax details. However, such informal drawings require the designer to translate them manually into the input notation of a compiler generator. The person who drew it must go through the translation of the pencil strokes into the concrete syntax of the compiler generator. These inconveniences make the developers avoid the usage of AGs and go through non systematic ways to implement the languages and supporting tools. So, in this paper, we develop a *Visual Language* (VL), as a meta-language to write AGs, based on a previous conceptualization that we have proposed in [1]. The idea of this VL is not only about having a nice visual depiction and then to translate it into a target notation, but also about syntactic and semantic consistency checks.

VLs and consequently the *Visual Programming Languages* (VPLs) aim at offering the possibility to solve complex problems by describing their properties or their behavior through graphical/iconic definitions [2]. Icons are used to be composed in a space with two or more dimensions, defining sentences that are formally accepted by parsers, where shape, color and relative position of the icons are relevant issues. A visual programming language implies the existence of a *Visual Programming Environment* (VPE) [3, 4], because its absence makes the language useless. Commonly, a visual programming environment consists of an editor, enriched by several tools to analyze, to process and to transform the drawings.

The main idea of this work is the development of a VPE, named VisualLISA, that assures the possibility of specifying AGs visually, and to translate them into plain text specifications or, alternatively, into a universal XML representation designed to support generic AG specifications. The original objective of this environment is to be used as front-end for LISA [5] system, diminishing the difficulties regarding the specification of AGs in LISA. However, the generality of the environment enables its use with systems other than LISA.

The visual programming environment is automatically generated by DEViL, our choice among many other tools studied; so, in this paper, the system is introduced and its use explained. However, our objective in this paper is not concerned with the discussion of compiler development tools, but show the befits of using an effective one.

In section 2, related work is presented. In Section 3 and 4, VisualLISA language and editor are informally described. In Section 5 the language is formally specified, defining syntactic rules, semantic constraints and a valid translation scheme for both LISA (the first target), and $\mathcal{X}$AGra notations. In Section 6, the DEViL generator framework, used for the automatic generation of the visual editor, will be presented. In Section 7, following the informal conception and its formalization, using DEViL, the visual language and the editor implementation is shown. An overview on how to use the editor to describe an AG, is given in Section 8.

In Section 9, $\mathcal{X}$AGra dialect is formally presented. Its main idea is to generalize the output of AG editing tools; instead of generating a description for a specific compiler generator, the editor under development can produce this general purpose dialect. Then to use this editor as a *Front End* (FE) for a specific generator, it is only necessary to resort to a simple translator to convert the XML description into the specific notation of that CG. This approach raises the usefulness of the editor, as it can be used as a FE for a larger range of grammar-based generators. However, as its applicability does not end here, we introduce, in Appendix A, $\mathcal{X}$AGraAl, a tool that, based on $\mathcal{X}$AGra specifications, performs grammar analysis and transformations.

The paper is concluded in Section 10.

2. Related Work

Despite of existing many other applications for AGs, they are commonly associated with the development of computer languages and related tools like parsers, translators, compilers, and others. In this context, the language engineers, started to develop tools to systematize and automatize the process of defining AGs. So, several works on this area may be cited.

LISA [5, 6] is a compiler generator based on attribute grammars, developed at University of Maribor at Slovenia. Its main objective is to generate a compiler for a language. The compiler is created by the specification of a textual attribute grammar. It automatically generates graphical and visualization tools [7] to inspect the written grammar, but it always need a textual specification of the AG.

In the same way, AnTLR [8], a powerful compiler generator, requires textual specifications for the language grammar. This system provides *online* visualization of the grammar productions but it does not provide any visualization about the attributes neither the semantic rules of each production.

Other similar compiler generators like UltraGram [9] or ProGrammar [10] also produce graphical tools to ease the understanding of the grammar. But still, the input for these compiler generators is always a text-based specification.

The same happens in the visual languages generation area. DEViL [11], a generator of visual programming languages and editors, takes advantage of AGs to define these visual outcomes. But, despite of providing excellent and usable results, the engineer needs to grasp a whole new syntax to define the AG used to produce the visual language.

In [12], *Ikezoe et al.* present a systematic debugger for attribute grammars integrated in a visual tool, and it provides visualizations of the grammar showing the dependency between the symbols and the attributes. Although this is a useful tool, it is only used after constructing the AG, not being a good help to map the mental construction of an AG into its specifications.

There are, indeed, several tools to support the specification and development of AGs and their associated tools, however, and according to our knowledge, acquired through years of research work on the area, there are no tools that allow the specification of AGs using a visual notation.

3. VisualLISA - A Domain Specific Visual Language

For many years we have been thinking about and working with AGs. Inevitably we created an abstract mental representation of how it can be regarded and then sketched, for an easier comprehension and use. So we decided to implement a framework that follows that representation. The conception of that framework is described in this section.

3.1. The Language Conception

VisualLISA, as a new *Domain Specific Visual Language* (DSVL) for attribute grammar specification, shall have an attractive and comprehensible layout, besides the easiness of specifying the grammar model.

We think that a desirable way to draw an AG is to make it production oriented, and from there design each production as a tree. The *Right-Hand Side* (RHS) symbols should be connected, by means of a visible line, to the *Left-Hand Side* (LHS) symbol. The attributes should be connected to the respective symbols, using a connection line different from the one referred before, as both have different purposes (see Figure 6). The rules to compute the values of each attribute should exhibit the shape of a function with its arguments (input attributes) and its results (the output attributes). Two kinds of functions should be represented: the identity function (when we just want to copy values) or a generic function (for other kind of computations). Often a production has a considerable number of attributes and nontrivial computations. Therefore we think that for visualization purposes, the layout of each production should work as a reusable template to draw several computation rules. Hence, the rules are drawn separated from each other, but associated to a production.

All these features can be seen in the following example, which gives a big picture of how things get easier when dealing with the visual notation on a real language. For this illustration we resort to LISS [13], which is a programming language allowing the operation with atomic or structured integers values. Moreover, it is fully specified using an AG, from where two semantic productions are shown in Listing 1. For illustration purposes, some semantic rules from the actual productions were dismissed.

Figure 3.1 shows the visual specification of production P1, taking advantage of the *production layout reuse* feature, for rapid development and clarity. This means that the user doesn't have to draw the production each time he needs to specify the computation of another attribute.

Listing 1. One Production form LISS

```
P1: Expr → Expr RelOp SingExp {
  (...)
  Expr[1].inRow   = Expr[0].inRow;
  Expr[1].inCol   = Expr[0].inCol;
  SingExp.inRow   = Expr[0].inRow;
  (..)
  Expr[0].out = Expr[1].out + RelOp.out + SingExp.out;
}

P2: SingExp → Term {
    SingExp.out = Term.out;
    (...)
}
```

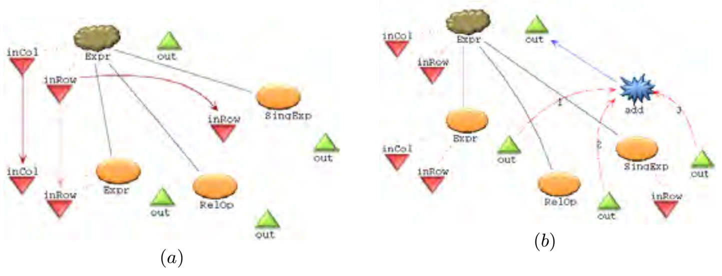


Fig. 1. LISS production on `VisualLISA`, with associated semantic rules

This way, Figure 3.1 (a) we copy the values from the inherited attributes of the `LHS` symbol to the symbols at `RHS`; and on another computation associated with the same production (c.f. Figure 3.1 (b)), we assign a value to the `LHS`'s *out* attribute using a function and the values of other attributes implied in the production.

4. VisualLISA: The Environment

`VisualLISA` editor should be compliant with the idea of offering a nice and non error-prone way of sketching the `AG`, as a first step; and an easy translation of the model into a target language, as a second step. So, three main features are highlighted: (i) syntax validation, (ii) semantics verification and (iii) code generation. The syntax validation restricts some spatial combinations among the icons of the language. In order to avoid syntactic mistakes, the edition should be syntax-directed. The semantics verification copes with the static and dynamic semantics of the language. Finally, the code generation feature generates code from the drawings sketched up. The target code would be LISAsl or $\mathcal{X}$AGra. LISAsl specification generated is intended to be passed to LISA system in a straightforward step. $\mathcal{X}$AGra specification generated is intended to give the system more versatility and further usage perspectives.

5. Specification of VisualLISA

The specification of VisualLISA bases on three main issues: $i)$ the definition of the underlying language's syntax; $ii)$ the language semantics and $iii)$ the description of the textual specifications into which the iconic compositions will be translated.

5.1. Syntax

The *Picture Layout Grammar* (PLG) formalism [14], is an attribute grammar to formally specify visual languages. It assumes the existence of pre-defined terminal symbols and a set of spatial relation operators. Our acquaintance with PLG formalism, from previous works, led us to use it to specify the syntax of VisualLISA. Listings 2 present some rules of the language specification. For the sake of space we only present the key rules of the specification; the missing productions are comparable to those shown.

Figure 2 shows the concrete and connector icons used for VisualLISA specifications. *LeftSymbol* is the LHS of a production, while *NonTerminal* and *Terminal* are used to compose the RHS. The second line of icons in Figure 2 presents the several classes of attributes. *Function* and *Identity*, both representing operations, are used to compute the attribute values. The other icons connect the concrete symbols with each other, to rig up the AG.

Listing 2. VisualLISA Partial Syntax Definition.

```
AG → contains(VIEW, ROOT)

VIEW → labels(text, rectangle)

ROOT → left_to(PRODS, SPECS)

SPECS → contains(VIEW,
          over(LEXEMES, USER_FUNCS))

PRODS → group_of(SEMPROD)

SEMPROD → contains(VIEW, left_to(
   group_of(group_of(RULE_ELEM)),
   group_of(AG_ELEM)))

AG_ELEM → LEFT_SYMBOL
          | NON_TERMINAL
          | TERMINAL
          | SYNT_ATTRIBUTE
          | INH_ATTRIBUTE
          | TREE_BRANCH
          | INT_ATTRIBUTE
          | SYNT_CONNECTION
          | INH_CONNECTION
          | INT_CONNECTION
```

```
RULE_ELEM → FUNCTION
           | IDENTITY
           | FUNCTION_ARG
           | FUNCTION_OUT

TERMINAL → labels(text, rectangle)

INT_ATTRIBUTE → labels(text, triangle)

INT_CONNECTION → points_from(
                   points_to(
                    dash_line,
                    ~INT_ATTRIBUTE),
                    ~TERMINAL)

FUNCTION → over(rectangle, text)

FUNCTION_OUT → points_from(
            points_to(arrow,
            ~INH_ATTRIBUTE),
            ~FUNCTION)
              | points_from(
            points_to(arrow,
            ~SYNT_ATTRIBUTE),
            ~FUNCTION)
```

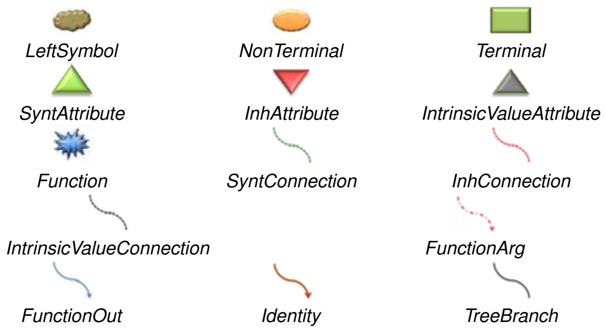


Fig. 2. The Icons of `VisualLISA`

5.2. Semantics

In order to correctly specify an AG, many semantic constraints must hold. These constraints are related with the attribute values that depend on the context in which the associated symbols occur in a sentence. We separated these constraints into two major groups. One concerning the syntactic rules, *Production Constraints* (PC), and another the respective computation rules, *Computation Rules Constraints* (CRC).

The following statements are representative constraints of `VisualLISA`'s semantic correctness, concerning the two groups identified before:

PC: *The data type of an attribute X.a in a production, must be the same in any production where X.a occurs.*

CRC: *The type of the target attribute and the return type of a function, when they are connected by a FunctionOut symbol, must match.*

The complete set of constraints can be seen in [9].

5.3. Translation

The translation ($\mathcal{L}_s \rightarrow \tau \rightarrow \mathcal{L}_t$) is the transformation of a source language into a target language. τ is a mapping between the productions of the $\mathcal{L}_s$ (`VisualLISA`) and the fragments of $\mathcal{L}_t$ (`LISAsl` $\cup$ $\mathcal{X}$`AGra`). These fragments will be specified in this sub-section.

A *Context Free Grammar* (CFG) is a formal and robust way of representing LISA specifications' structure. Listing 3 presents that high-level CFG.

Listing 3. LISA structure in a CFG.

```
p1 : LisaML          → language id { Body }
p2 : Body            → Lexicon Attributes Productions Methods
p3 : Lexicon         → lexicon { LexBody }
p4 : LexBody         → (regName regExp)⋆
p5 : Attributes      → attributes (type symbol . attName ;)⋆
p6 : Productions     → rule id { Derivation } ;
p7 : Derivation      → symbol ::= Symbs compute { SemOperations }
p8 : Symbs           → symbol+
p9 :                 | epsilon
p10 : SemOperations  → symbol . attName = Operation ;
p11 : Operation      → ...
p12 : Methods        → method id { javaDeclarations }
```

Reserved words, written in bold, enhance the main fragments in a LISA sentence, making it more readable. The definition of smaller chunks, introduced by each keyword, enables a more modular processing (code generation. . .)

Regarding the literature, there is not an XML standard notation for AGs. So that, $\mathcal{X}$AGra was defined using a schema. The whole structure of this schema can be seen in detail in Section 9.

6. DEViL - A Tool for Automatic Generation of Visual Programming Environments

We searched for VPE generators like MetaEdit+ [15], but their commercial nature was not viable for an academic research. Also, we experimented VLDesk [16], Tiger [17], Atom^3 [18] and other similar tools, however none of them gave us the flexibility that DEViL offered, as described below.

The DEViL system generates editors for visual languages from high-level specifications. DEViL (*Development Environment for Visual Languages*) has been developed at the University of Paderborn in Germany and is used in many nameable industrial and educational projects.

The editors generated by DEViL offer syntax-directed editing and all features of commonly used editors like multi-document environment, copy-and-paste, printing, save and load of examples. Usability of the generated editors and DEViL itself can be found in [11]. DEViL is based on the compiler generator framework Eli [19], hence all of Eli's features can be used as well. Specially the semantic analysis module can be used to verify a visual language instance and to produce a source-to-source translation.

To specify an editor in DEViL we have to define the semantic model of the visual language at first. It is defined by the domain specific language *DEViL Structure Specification Language* (DSSL) which is inspired by object-oriented languages and offers classes, inheritance, aggregation and the definition of attributes. The next specification step is to define a concrete graphical representation for the visual language. It is done by attaching so called visual patterns to the semantic model of the VL specified in DSSL. Classes and attributes of DSSL inherit from these visual patterns. Visual patterns [20] describe in what way parts of the syntax tree of the VL are represented graphically, e.g. we can model that some part should be represented as a “set” or as a “matrix”. DEViL offers a huge library of precoined patterns like formulae, lists, tables or image primitives. All visual patterns can be adapted through control attributes. E.g. we can define paddings or colors of all graphical primitives. Technically visual patterns are decorated to the syntax tree by specifying some easy inheritance rules in a DSL called LIDO.

To analyse the visual language, DEViL offers several ways. The first one results from the fact that editors generated by DEViL are syntax directed. Hence, the user cannot construct *wrong* instances of the VL It is limited by its syntax and cardinalities expressed in DSSL. Another way is to define check rules e.g. to check the range of an integer attribute or to do a simple name analysis on a name attribute. To navigate through the structure tree of the VL, DEViL offers so called path expressions which are inspired by XPath. They can be used in a small simple DSL to reach every node in the tree. After analysis, DEViL can generate code from the VL instance. This is done with the help of Eli which offers unparsers, template mechanism (*Pattern-based Text Generator* — PTG) and the well-known attribute evaluators from compiler construction.

7. Implementation of VisualLISA

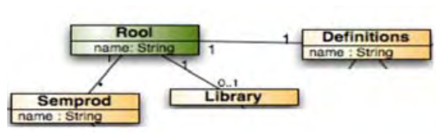
To implement VisualLISA, we could have followed a non systematic way, resorting to usual software development methods. But our know-how on compiler construction led us to reuse the systematic generative approach followed in that area. In this case, the implementation process will be supported by the formal specification made in Section 5, and automated by the VPE chosen, DEViL. Adopting the standard compiler construction process to the DEViL usage peculiarities, we will follow a four-step process: i) Abstract Syntax Specification; ii) Interaction and Layout Definition; iii) Semantics Implementation; and iv) Code Generation.

7.1. Abstract Syntax

The specification of the abstract syntax of VisualLISA, in DEViL, follows an object-oriented notation, as referred previously. This means that the nonterminal symbols of the grammar are defined modularly: the symbols can be seen as classes and the attributes of the symbols as class attributes.

The syntax of the visual language is determined by the relations among their symbols. Therefore, for an high level representation of the language's syntax, a class diagram can be used. This diagram should meet the structure of the PLG model in Figure 2. The final specification for the language is then an easy manual process of converting the diagram into DSSL. Figure 3 shows a small example of the diagram and the resultant specification.

There are two types of classes in this notation: concrete and abstract. The concrete classes are used to produce a syntax tree, which is manipulated in the other steps of the environment implementation. The abstract classes, besides the normal inheritance properties can be used to define syntactic constraints. These classes generate the syntax-directed editor.



```
CLASS Root {
  name: VAL VLString;
  semprods: SUB Semprod*;
  defs: SUB Definitions!;
  library: SUB Library?;
}
```

Fig. 3. Class Diagram and Respective `DEViL` Notation

In order to make possible the specification of separated computation rules reusing the same layout of a production, we used `DEViL`'s concept of coupled structures [21]. It couples the syntactic structure of two structure tree — for `VisualLISA` we used the structure of symbol *Semprod*, which is used to model a production. In practice, it means that the layout defined for a production is replicated whenever a computation rule is defined, maintaining both models synchronized all the time.

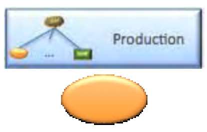
7.2. Interaction and Layout

The implementation of this part, in `DEViL`, consists of the definition of views. A view can be seen as a window with a dock and an editing area where the language icons are used to specify the drawing.

`VisualLISA` Editor is based on four views: *rootView*, to create a list of productions; *prodsView*, to model the production layout; *rulesView*, to specify the semantic rules reusing the production layout and *defsView*, to declare global definitions of the grammar.

At first the buttons of the dock, used to drag structure-objects into the edition area, are defined. Then the visual shape of the symbols of the grammar for the respective view are defined. Figure 4 shows parts of view definitions and the respective results in the editor. The code on the left side of Figure 4 defines the view, the buttons and the behavior of the buttons. The default action is the insertion of a symbol in the editing area. The bluish rectangular image represents the button resultant from that code.

```
VIEW rootView ROOT Root{
 BUTTON IMAGE "img::btnProd"
   INSERTS Semprod
     INFO "Production";
}
```



```
SYMBOL pview_NonTerminal
INHERITS VPForm
COMPUTE
  SYNT.drawing =
       ADDROF(ntDrawing);
END;
```

Fig. 4. Parts of View Definitions and Respective Visual Outcomes

Symbol *NonTerminal* is represented by the orange oval in Figure 4. The code on the right reveals the semantic computation to define the shape of that symbol. Shape and other visual aspects of the tree-grammar symbols are automatically defined associating, by inheritance, visual patterns.

7.3. Semantics

As long as VisualLISA is defined by an AG, the contextual conditions could be checked using the traditional approach. DEViL is very flexible and offers some other ways to implement this verification module. The approach used to develop VisualLISA, is completely focused on the contexts of the generated syntax tree. DEViL offers a tree-walker, that traverses the tree and for a given context — a symbol of that tree — executes a verification code (callback-functions), returning an error whenever it occurs. With this approach it is easy to define data-structures helping the verification process. This approach is very similar to the generic AG approach, but instead of attributes and semantic rules, it uses variables which are assigned by the result of queries on the tree of the model.

Listing 4 shows the code for the implementation of a constraint defined in [22].

Listing 4. Implementation of Constraint: “Every *NonTerminal* specified in the grammar must be root of one production”

```
checkutil::addCheck Semprod {
 set n [llength [c::getList {$obj.grammarElements.CHILDREN[LeftSymbol]}]]
 set symbName [c::get {$obj.name.VALUE}]
 if { $n == 0 } {
  return "Production '$symbName' must have one Root symbol!"
 } elseif {$n > 1} {
  return "Production '$symbName' must have only one Root symbol!"
 }
 return ""
}
```

A considerable amount of the constraints defined in Section 5.2 were verified resorting to the Identifier Table, which is a well known strategy in language processing for that purpose.

7.4. Code Generation

The last step of the implementation, concerning the translation of the visual AG into LISA or $\mathcal{X}$AGra, can be done using the AG underlying the visual language (as usual in language processing). For this task, DEViL supports $i)$ powerful mechanisms to ease the semantic rules definition; $ii)$ facilities to extend the semantic rules by using functions and $iii)$ a template language (PTG of Eli system) incorporation to structure out the output code.

The use of patterns (templates) is not mandatory. But, as seen in the formal definition of LISA and $\mathcal{X}$AGra notation (Section 5.3), both of them have static parts which do not vary from specification to specification. Hence templates are very handy here. Even with templates, the translation of the visual AG into text is not an easy task. Some problems arise from the fact that there is not a notion of order in a visual specification. We used auxiliary functions to sort the RHS symbols by regarding their disposition over an imaginary X-axe. Based on this approach we also solved issues like the numbering of repeated symbols in the production definition.

The templates (invoked like functions) and the auxiliary functions, together with other specific entities, were assembled into semantic rules in order to define the translation module. One module was defined for each target notation. New translation modules can be added, to support new target notations.

8. AG Specification in `VisualLISA`

Figure 5 shows the editor look and feel, presenting the four views of our editor.

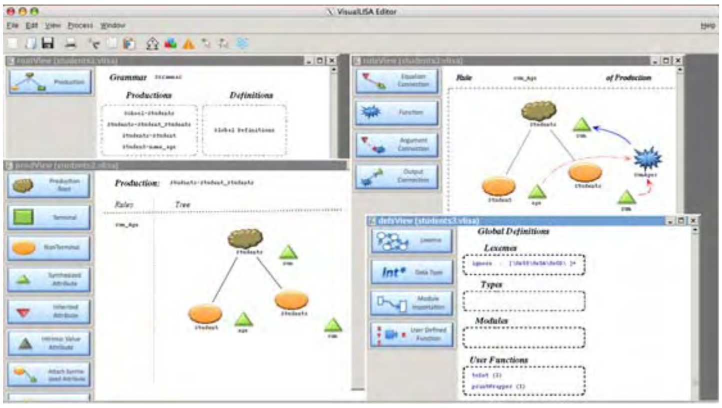


Fig. 5. `VisualLISA` Editor Environment

To specify an attribute grammar the user starts by declaring the productions (in *rootView*) and rigging them up by dragging the symbols from the dock to the editing area (in *prodsView*), as commonly done in VPEs. The combination of the symbols is almost automatic, since the editing is syntax-directed. When the production is specified, and the attributes are already attached to the symbols, the next step is to define the computation rules. Once again, the user drags the symbols from the dock, in *rulesView*, to the editing area, and compounds the computations by linking attributes to each other using functions. Sometimes it is necessary to resort to user-defined functions that should be described in *defsView*. In addition, he can import packages, define new data-types or define global lexemes.

As example we present a simple AG , called Students Grammar, used to process a list of students, described by their names and ages. The objective of this AG is to sum the ages of all the students. This grammar can be textually defined as shown in Listing 5.

Listing 5. Students Grammar

```
P1: Students → Student Students {Students0.sum = Student.age + Students1.sum}
P2: Students → Student          {Students.sum = Student.age}
P3: Student → name age          {Student.age = age.value}
```

Figures 6 and 7 show the three productions that constitute the grammar.

In Figure 6, the attributes are associated with the symbols of the production. Moreover, the production has a semantic rule that computes the value of the LHS's attribute, *sum*, by adding the value of the attributes in the RHS symbols, *sum* and *age*, using an *inline* function named *SumAges*

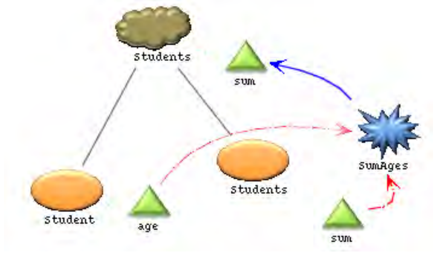


Fig. 6. Specification of production P1 with associated semantics

In Figure 7 (a), the identity function is used to copy the value of the attribute *age* to the attribute *sum*. In Figure 7 (b), the third production, makes use of terminal symbols and associated intrinsic values. The computation rule, in this production, is based on the conversion of the textual value of the age into an integer.

When the grammar is completely specified and semantically correct, code can be generated. Figure 8 shows, in LISA and $\mathcal{X}$AGra notations, the code generated for production P1 in Figure 6.

9. $\mathcal{X}$AGra- An XML dialect for Attribute Grammars

In this section is defined an XML dialect to cope with attribute grammars. We called it $\mathcal{X}$AGra, which stands for ***X**ML dialect for **A**ttribute **Gra**mmars*.

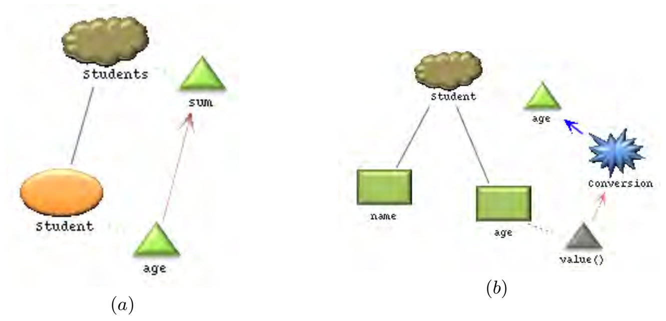
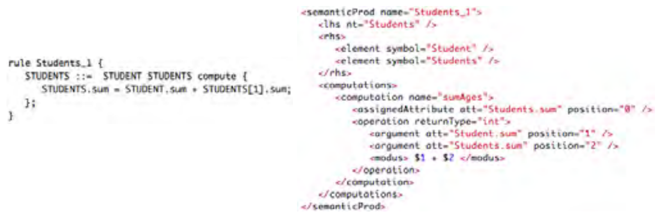


Fig. 7. Specification of Productions P2 and P3, (a) and (b) respectively, with associated semantic rules.



```
rule Students_1 {
   STUDENTS ::=  STUDENT STUDENTS compute {
      STUDENTS.sum = STUDENT.sum + STUDENTS[1].sum;
   };
}
```

```
<semanticProd name="Students_1">
   <lhs nt="Students" />
   <rhs>
      <element symbol="Student" />
      <element symbol="Students" />
   </rhs>
   <computations>
      <computation name="sumAges">
         <assignedAttribute att="Students.sum" position="0" />
         <operation returnType="int">
            <argument att="Student.sum" position="1" />
            <argument att="Students.sum" position="2" />
            <modus> $1 + $2 </modus>
         </operation>
      </computation>
   </computations>
</semanticProd>
```

(a) (b)

Fig. 8. Code Generated for LISA (a) and XAGra (b) specifications.

$\mathcal{X}$AGra denotes the abstract representation of an AG. The notation defined here, is mainly based on the definition of AG presented in the Introduction, but it also borrows parts from the notations inherent to various AG-based compiler generator tools.

One of the standardized ways to define a new XML dialect is the creation of a schema, using the standard XML Schema Definition (XSD) language. For the sake of space, the integral textual definition of $\mathcal{X}$AGra's schema is not presented, and for reasons of visibility and readability, the complete drawing of the schema is broken into several important sub-parts. Figures 9 to 13 are used to support the explanation of the dialect.

$\mathcal{X}$AGra's root element was defined as `attributeGrammar`. This element has a single attribute, `name`, whose objective is to store the name of the grammar, or the language that the grammar defines; and is a sequence of several elements. These elements represent components of the formal definition of an AG, incremented with extra parts related to the usage of AG-based compiler generators.

Table 1 defines a relation of inclusion between the $\mathcal{X}$AGra notation elements and the components that constitute the formal definition of an AG, which is recovered next:

$$AG = (T, N, S, P, A, R, C, \mathcal{T})$$

Table 1. Derivation of $\mathcal{X}$AGra Notation From the Formal Definition of AG

$\mathcal{X}$AGra Element	$\supseteq$ AG Components
symbols	T, N, S
attributesDecl	A
semanticProds	$P, R, C, \mathcal{T}$
importations	$\emptyset$
functions	$\emptyset$

The relations depicted in Table 1 give an overview about the information that each element of $\mathcal{X}$AGra notation will store. The following sections will describe with more detail such elements and the information they store.

Listing 6 presents a fragment of a grammar that computes the age of a set of students. This example is used to compare the concrete notation of a compiler generator to the XML fragments that are shown in the sequent figures.

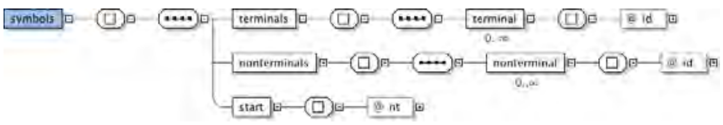
Next sections present a complete description of the elements of $\mathcal{X}$AGra scheme. However, the *importations* and *functions* elements are skipped, because their structure is simple and similar to the other parts shown.

9.1. Element *symbols*

Figure 9 presents the schema for the element `symbols`. As the name suggests, this element contains the declaration of the grammar's vocabulary.

Listing 6. Example of Students Grammar

```
language StudentsGra {
   lexicon{
      Name     [A–Z][a–z]+
   ...
   }
   attributes
      int   STUDENTS.sum;
   ...
   rule Students_1 {
      STUDENTS ::=  STUDENT STUDENTS compute {
         STUDENTS.sum = STUDENTS[1].sum + STUDENT.age;
      };
   }
   ...
   method user_Definitions {
      import java.util.ArrayList
      public int sum(int x, int y){
         return x+y;
      }
   }
}
```



```
<symbols>
    <terminals>
        <terminal id="name">[A–Z][a–z]+</terminal>
    </terminals>
    <nonterminals>
        <nonterminal id="students" />
    </nonterminals>
    <start nt="students" />
</symbols>
```

Fig. 9. $\mathcal{X}$AGra Schema – Element **Symbols**: definition and example

It is composed of a sequence of three elements: `terminals`, `nonterminals` and `start`.

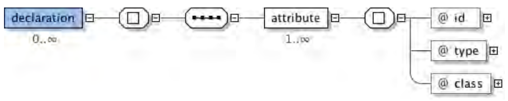
The element `terminals` is a sequence of zero or more elements named `terminal`, which, in its turn, has one attribute, `id`, used to store the name of a terminal symbol. This attribute is an identifier, hence any instance of it, must be different from the others, and must be always instantiated. Besides the information kept on the attribute, this element has a textual content where the respective *Regular Expression* (RE) can be declared.

The element `nonterminals` has similar structure. The difference lays on the fact that it represents a sequence of zero or more elements `nonterminal` which have no textual content. The attribute `id` has the same purpose as the attribute with the same name in the element `terminal`.

Finally, the element `start` has a single attribute named `nt`. This attribute is used to refer the nonterminal (already defined in the $\mathcal{X}$AGra specification), correspondent to the start symbol (or Axiom) of the AG.

9.2. Element *attributesDecl*

This element is composed of a sequence of zero or more elements `declaration`. For the sake of readability, Figure 10 only depicts the structure of the element `declaration`, which is a sequence of one or more elements `attribute`. This one has three mandatory attributes: $i)$ `id` – stores the name of the attribute being declared. Any kind of text can be used to define it, but it is always better to use the following notation: $X.a$, where X is the name of a symbol in $T \cup N$ and a is the name of an attribute in $A(X)$. As it is an identifier, it must be different from all other identifiers on the specification; $ii)$ `type` – stores the data type of the current attribute value and $iii)$ `class` – defines the class of the attribute. It must be one of: InhAttribute, SyntAttribute and IntrinsicValueAttribute.



```
<attributesDecl>
    <declaration>
        <attribute id="students.sum" type="int" class="SyntAttribute" />
    </declaration>
</attributesDecl>
```

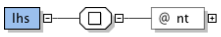
Fig. 10. $\mathcal{X}$AGra Schema – Element **Attribute Declarations**: definition and example

9.3. Element *semanticProds*

The element `semanticProds` represents the structure to define productions and associated semantic rules in $\mathcal{X}$`AGra` specifications. This structure is composed of a sequence of zero or more elements `semanticProd`. Each `semanticProd` has one single attribute, `name`, used to store the mandatory name of the production, as an identifier.

Element `semanticProd` has three direct descendants: `lhs`, `rhs`, `computation`, whose structure is explained in the next paragraphs and that are depicted in Figures 11, 12 and 13.

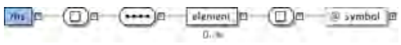
Element `lhs` (Figure 11) is used to refer to the nonterminal symbol on the `LHS` of the production. This element has a single attribute, `nt`, to refer to an existent `nonterminal`.



```
<lhs nt="students" />
```

Fig. 11. $\mathcal{X}$`AGra` Schema – Element **Semantic Productions**: `LHS` definition and example

Element `rhs` (Figure 12), stores the nonterminals on the `RHS` of a production. It is composed of a sequence of zero or more elements `element`. For this purpose, each `element`, has a single attribute, `symbol`, which is mandatory and represents a reference to a terminal or nonterminal symbol, already instantiated in the initial `symbols` structure.



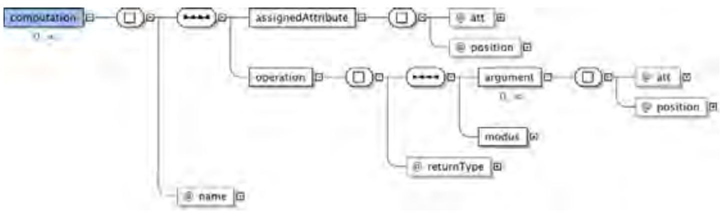
```
<rhs>
    <element symbol="student" />
    <element symbol="students" />
</rhs>
```

Fig. 12. $\mathcal{X}$`AGra` Schema – Element **Semantic Productions**: `RHS` definition and example

Element `computation` (Figure 13) is the last child of the element `semanticProds`. It represents an hard concept of `AG`s: the semantic rules.

This element has one attribute, `name`, used to give a name to the computation being declared. This attribute, despite being mandatory, is not a unique identifier: different computations can have equal names.

The structure of `computation` represents a pure abstraction of what is a semantic rule in an `AG` definition: the attribute to which a value is assigned, and



```
<computation name="getTheSum">
    <assignedAttribute att="students.sum" position="0" />
    <operation returnType="int">
        <argument att="student.age" position="1" />
        <argument att="students.sum" position="2" />
        <modus> $1 + $2 </modus>
    </operation>
</computation>
```

Fig. 13. $\mathcal{X}$AGra Schema – Element **Semantic Productions**: `Computation` definition and example

the operation that computes this value. Thus, the element `computation` has two children: the elements `assignedAttribute` and `operation`.

Element `assignedAttribute` is composed of two mandatory attributes: `att`, which is used to refer to an attribute; and `position`, which is a number that identifies the position of the symbol associated to the attribute in the list of elements of the production. That is, if the attribute is connected to the `LHS`, then the value for `position` must be 0. If the associated symbol belongs to the `RHS`, then its value should correspond to the position that the symbol occupies in the `RHS` sequence of symbols, starting with 1.

The element `operation` aggregates a sequence of zero or more elements `argument` and a single element `modus`. In addition to the elements, it has an attribute, `returnType`, used to store the data type of the value returned by the operation.

Elements `argument` are, in all aspects, equal to the `assignedAttribute` element. Each one has two attributes with the same name and the same semantic value underlaying, therefore they are used to refer to previous declared attributes. The difference is on the fact that this time, the attributes referenced are those used to compute the value in the operation.

The last element, `modus`[4], which is a simple text field to write the expression used to compute the value. Somehow, in this element's text, a reference to the argument attributes should be made. An example (and the convention established) is using $\$x$, where $x > 0$ is the position of the attribute in the sequence of arguments.

[4] *modus* is a latin expression for *way* (of computing something, in our case)

Aside the *importations* and the *functions* parts, the $\mathcal{X}$AGra's schema is now completely defined and explained, revealing the universality needed to store any AG for any AG-based compiler generator.

Next section briefly presents $\mathcal{X}$AGraAl, demonstrating one interesting applicability of the $\mathcal{X}$AGra dialect.

10. Conclusion

After many years working in specification and implementation of compilers supported by *Attribute Grammars*, it became clear that a modular and reusable approach to AG development is highly recommendable and necessary. On the other hand, the work on program comprehension tools emphasized the importance of software/data visualization. The combination of those two areas of R&D with a third one, the development of *Visual Languages*, gave rise to the proposal of creating a VL for AGs, since there are no other tools allowing it, according to our knowledge. The obligation to write text-based AG specifications imposed by several compiler generator tools and the habitual way of sketching AGs on paper in the form of a decorated tree, shortening the gap to the mental representation of an AG, reinforced the appropriateness of that proposal.

In this paper we introduced VisualLISA, a new *Domain Specific Visual Language*, which enables the specification of AGs in a visual manner and the translation of that visual AG into LISA or $\mathcal{X}$AGra (an XML notation to support generic AG specifications). $\mathcal{X}$AGra allows us to use this visual editor with other compiler-compiler tools.

We were mainly concerned with the design of the language, its formal and automatic implementation. In this phase of our project we neither focused on the usability of the language nor on its scalability. We focused on the specification, aiming at showing the formal work behind the visual outcome, and on the implementation of the underlying environment to specify AGs. At this point we highlighted the use of DEViL in order to create the desired environment, through a systematic approach of development. Also, an example was presented to show the steps to build an AG with VisualLISA.

In the future, it is our objective to perform at least, two experimental studies involving VisualLISA: one to assess the usability of the language regarding the visual vs textual approaches for developing AGs; and another one to test the scalability of the language and environment, regarding the hypothesis that it was created to cope with small AGs. We are also interested in assessing the comprehension of AGs; maybe VisualLISA would be very handy on this matter, working as AGs visualizer.

Concerning the applicability of $\mathcal{X}$AGra, we can translate it into the specific notation of any compiler generator tool. We call $\mathcal{X}$AGra loader to the program that performs this translation. As future work, the following translators are planed: $\mathcal{X}$AGra into LISA (a traditional LR parser generator); $\mathcal{X}$AGra into AntLR (an LL parser generator, based on an extended BNF grammar); $\mathcal{X}$AGra into `Eli` (an LR parser generator with special constructors). Finally, a translator from $\mathcal{X}$`AGra` to `Yacc` could be a challenging project.

Also, we developed a *Grammar Analyzer and Transformation* tool, $\mathcal{X}$`AGraAl`. that takes as input an `AG` written in $\mathcal{X}$`AGra`. Thus, the implementation of this tool shows the applicability of $\mathcal{X}$`AGra` as a universal and multi-purpose `AG` specification language.

References

1. Pereira, M.J.V., Mernik, M., da Cruz, D., Henriques, P.R.: VisualLISA: a visual interface for an attribute grammar based compiler-compiler (short paper). In: CoRTA08 — Compilers, Related Technologies and Applications, Bragança, Portugal. (July 2008)
2. Boshernitsan, M., Downes, M.: Visual programming languages: A survey. Technical report, University of California, Berkeley, California 94720 (December 2004)
3. Kastens, U., Schmidt, C.: VL-Eli: A generator for visual languages - system demonstration. Electr. Notes Theor. Comput. Sci. **65**(3) (2002)
4. Costagliola, G., Tortora, G., Orefice, S., De Lucia, A.: Automatic generation of visual programming environments. Computer **28**(3) (1995) 56–66
5. Mernik, M., Lenič, M., Avdičaušević, E., Žumer, V.: LISA: An interactive environment for programming language development. Compiler Construction (2002) 1–4
6. Mernik, M., Korbar, N., Žumer, V.: LISA: a tool for automatic language implementation. SIGPLAN Not. **30**(4) (1995) 71–79
7. Henriques, P.R., Pereira, M.J.V., Mernik, M., Lenič, M., Gray, J., Wu, H.: Automatic generation of language-based tools using the lisa system. Software, IEE Proceedings - **152**(2) (2005) 54–69
8. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. First edn. Pragmatic Programmers. Pragmatic Bookshelf (May 2007)
9. Solutions, U.: Ultragram - parser generator. http://www.ultragram.com/ (January 2010)
10. NorKen Technologies, I.: Programmar. http://www.programmar.com (January 2010)
11. Schmidt, C., Cramer, B., Kastens, U.: Usability evaluation of a system for implementation of visual languages. In: Symposium on Visual Languages and Human-Centric Computing, Coeur d'Alène, Idaho, USA, IEEE Computer Society Press (September 2007) 231–238
12. Ikezoe, Y., Sasaki, A., Ohshima, Y., Wakita, K., Sassa, M.: Systematic debugging of attribute grammars. In: AADEBUG. (2000)
13. da Cruz, D., Henriques, P.R.: Liss — the language and the compiler. In: Proceedings of the 1.st Conference on Compiler Related Technologies and Applications, CoRTA'07 — Universidade da Beira Interior, Portugal. (Jul 2007)
14. Golin, E.J.: A Method for the Specification and Parsing of Visual Languages. PhD thesis, Brown University, Department of Computer Science, Providence, RI, USA (May 1991)
15. Tolvanen, J.P., Rossi, M.: Metaedit+: defining and using domain-specific modeling languages and code generators. In: OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM (2003) 92–93
16. Costagliola, G., Deufemia, V., Polese, G.: A framework for modeling and implementing visual notations with applications to software engineering. ACM Trans. Softw. Eng. Methodol. **13**(4) (2004) 431–487

17. Ehrig, K., Ermel, C., Hänsgen, S., Taentzer, G.: Generation of visual editors as eclipse plug-ins. In: ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, New York, NY, USA, ACM Press (2005) 134–143
18. de Lara, J., Vangheluwe, H.: Atom3: A tool for multi-formalism and meta-modelling. In: FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering, London, UK, Springer-Verlag (2002) 174–188
19. Gray, R.W., Heuring, V.P., Levi, S.P., Sloane, A.M., Waite, W.M.: Eli: A complete, flexible compiler construction system. Communications of the ACM **35**(2) (February 1992) 121–131
20. Schmidt, C., Kastens, U., Cramer, B.: Using DEViL for implementation of domain-specific visual languages. In: Proceedings of the 1st Workshop on Domain-Specific Program Development, Nantes, France (July 2006)
21. Schmidt, C.: Generierung von Struktureditoren für anspruchsvolle visuelle Sprachen. Dissertation, Universität Paderborn (January 2006)
22. Oliveira, N., Pereira, M.J.V., da Cruz, D., Henriques, P.R.: VisualLISA. Technical report, Universidade do Minho (February 2009) `www.di.uminho.pt/~gepl/VisualLISA/documentation.php`.
23. GlassFish: Java architecture for XML binding. `https://jaxb.dev.java.net/` (June 2009)
24. GlassFish: Java API for XML processing. `https://jaxp.dev.java.net/` (June 2009)
25. da Cruz, D., Oliveira, N., Henriques, P.R.: Graal - a grammar analyzer (September 2009) Available at: `http://inforum.org.pt/INForum2009/programa`.

A. $\mathcal{X}$**`AGraAl`: A grammar analyzer based on** $\mathcal{X}$**`AGra`**

In this section we give a brief introduction to $\mathcal{X}$`AGraAl`, a *Grammar Analyzer and Transformation tool* that computes dependencies among symbols, grammar metrics, and grammar slices for a given criterion; moreover, $\mathcal{X}$`AGraAl` can also derive, from the original, shorter grammars combining slices or removing unitary productions (similar to re-factoring a program). $\mathcal{X}$`AGraAl` takes as input an `AG` written in $\mathcal{X}$`AGra`.

$\mathcal{X}$`AGraAl` is a platform independent tool, developed using Java. Java Architecture for XML Binding (`JAXB`) [23] and Java API for XML Processing (`JAXP`) [24] were used to process the input.

While parsing a $\mathcal{X}$`AGra` grammar using JAXB, $\mathcal{X}$`AGraAl` builds the identifiers table (`IdTab`) where it collects all grammar symbols and attributes; each identifier is associated with all its characteristics extracted or inferred from the source document. The identifiers table — that can be pretty-printed in `HTML` — complemented by the dependence graph (`DG`) — also printable using `Dot` and `GraphViz` — constitute the core of the tool. Traversing those internal representation structures, it is possible to implement the other $\mathcal{X}$`AGraAl` functionalities:

– Metrics, to assess grammar quality;
– Slicing, to ease the analysis producing sub-grammars focussed in a specific symbol or attribute;

– Re-factoring, to optimize grammars generating smaller and more efficient versions.

Metrics are organized in three groups of assessment parameters:

– Size metrics, that measure the number of symbols, productions, and so on (grammar and parser sizes);
– Form metrics, that describe the recursion pattern and measure the dependencies between symbols (the grammar complexity);
– Lexicographic metrics, that qualify the clearness/readablity of grammar identifiers, based on a domain ontology.

Slicing operation builds partial grammar with the elements that derive in zero or more steps on the criterion (backward slicing), or that are reachable from the criterion (forward slicing). The criterion can be either a symbol or an attribute. Slices are usually presented as paths over the dependence graphs. Figures 14 (a) and (b) illustrate a forward and a backward slice w.r.t the symbol *age*.

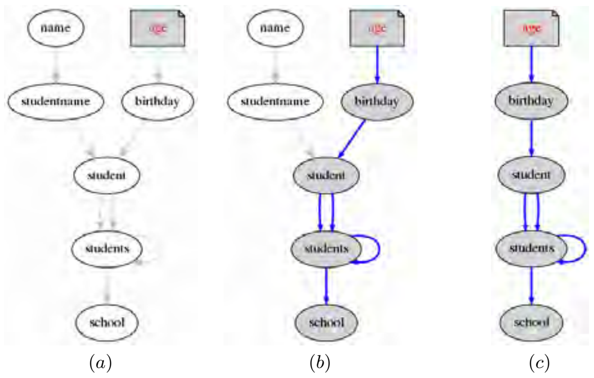


Fig. 14. Slices with respect to symbol *age*: (a) Forward slice; (b) Backward slice and (c) Combination of Forward and Backward slices

Re-factoring is a not so usual functionality that transforms the original grammar into a minimal one, removing all the *useless productions*. Another transformation also provided is the generation of a new grammar combining forward and backward slices with respect to the same symbol (see Figure 14 (c)).

Built in a similar way, `GraAL` [25] accepts as input a grammar written in `AntLR 3` and produces the same outputs. However, $\mathcal{X}$`AGraAl` beats `GraAL` in terms of generality as it consumes a grammar written in `XML`.

Nuno Oliveira, received, from University of Minho, a B.Sc. in Computer Science (2007) and a M.Sc. in Informatics (2009). He is a member of the Language Processing group at CCTC (Computer Science and Technology Center), University of Minho. He participated in several projects with focus on Visual Languages and Program Comprehension; VisualLISA and Alma2 the main outcome of his master thesis entitled "Program Comprehension Tools for Domain-Specific Languages", are the most relevant works. The latter came under "Program Comprehension for Domain-Specific Languages", a bilateral project between Portugal and Slovenia, funded by FCT. Currently, he is starting his PhD studies on Patterns for Architectures Coordination Analysis and Self-Adaptive Architectures, under MathIS, a research project also funded by FCT. Meanwhile he is an assistant-lecturer (practical classes) in a course on Imperative Programming, at University of Minho.

Maria João Varanda Pereira, received the M.Sc. and Ph.D. degrees in computer science from the University of Minho in 1996 and 2003 respectively. She is a member of the Language Processing group in the Computer Science and Technology Center , at the University of Minho. She is currently an adjunct professor at the Technology and Management School of the Polytechnic Institute of Bragança, on the Informatics and Communications Department and vice-president of the same school. She usually teaches courses under the broader area of programming: programming languages, algorithms and language processing. But also some courses about project management. As a researcher of gEPL, she is working with the development of compilers based on attribute grammars, automatic generation tools, visual languages and program understanding. She was also responsible for PCVIA project (Program Comprehension by Visual Inspection and Animation), a FCT funded national research project; She was involved in several bilateral cooperation projects with University of Maribor (Slovenia) since 2000. The last one was about the subject "Program Comprehension for Domain Specific Languages".

Pedro Rangel Henriques, got a degree in "Electrotechnical/Electronics Engineering", at FEUP (Porto University), and finished a Ph.D. thesis in "Formal Languages and Attribute Grammars" at University of Minho. In 1981 he joined the Computer Science Department of University of Minho, where he is a teacher/researcher. Since 1995 he is the coordinator of the "Language Processing group" at CCTC (Computer Science and Technologies Center). He teaches many different courses under the broader area of programming: Programming Languages and Paradigms; Compilers, Grammar Engineering and Software Analysis and Transformation; etc. Pedro Rangel Henriques has supervised Ph.D. (11), and M.Sc. (13) thesis, and more than 50 graduating trainingships/projects, in the areas of: language processing (textual and visual), and structured document processing; code analysis, program visulaization/animation and program comprehension; knowledge discovery from databases, data-mining, and data-cleaning. He is co-author of the “XML & XSL: da teoria a prática” book, published by FCA in 2002; and has published 3 chapters in books, and 20 journal papers.

Daniela da Cruz*,* received a degree in “Mathematics and Computer Science”, at University of Minho, and now she is a Ph.D. student of “Computer Science” also at University of Minho, under the MAPi doctoral program. She joined the research and teaching team of “gEPL, the Language Processing group” in 2005. She is teaching assistant in different courses in the area of Compilers and Formal Development of Language Processors; and Programming Languages and Paradigms (Procedural, Logic, and OO). As a researcher of gEPL, Daniela is working with the development of compilers based on attribute grammars and automatic generation tools. She developed a completed compiler and a virtual machine for the LISS language (an imperative and powerful programming language conceived at UM). She was also involved in the PCVIA (Program Comprehension by Visual Inspection and Animation), a FCT funded national research project; in that context, Daniela worked in the implementation of “Alm”, a program visualizer and animator tool for program understanding. Now she is working in the intersection of formal verification (design by contract) and code analysis techniques, mainly slicing.

Bastian Cramer*,* received his degree in Computer Science from the University of Paderborn, Germany in 2005. Then he joined the research group ’Programming Languages and Compilers’ of Prof. Kastens at the same university. His research focus is the generation of software from specifications and especially the generation of environments for visual domain specific languages. He has several years of experience in language design in corporation with the automotive industry. Currently he is working on his PhD concerning simulation and animation of visual languages.

Received: November 15, 2009; Accepted: February 24, 2010.

UDC 004.4, DOI: 10.2298/CSIS1002291P

Annotation Based Parser Generator*

Jaroslav Porubän[1], Michal Forgáč[1], Miroslav Sabo[1], and Marek Běhálek[2]

[1] Department of Computers and Informatics, Technical University of Košice,
Letná 9, 042 00 Košice, Slovak Republic
{Jaroslav.Poruban, Michal.Forgac, Miroslav.Sabo}@tuke.sk
[2] Department of Computer Science, FEI VŠB Technical University of Ostrava,
17. listopadu 15, 708 33 Ostrava-Poruba, Czech Republic
marek.behalek@vsb.cz

Abstract. The paper presents innovative parser construction method and parser generator prototype which generates a computer language parser directly from a set of annotated classes in contrast to standard parser generators which specify concrete syntax of a computer language using BNF notation. A language with textual concrete syntax is defined upon the abstract syntax definition extended with annotations in the presented approach. Annotations define instances of concrete syntax patterns in a language. Abstract syntax of a language is inevitable input of the parser generator as well as language's concrete syntax pattern definitions. The process of parser implementation is presented on the concrete computer language – the Simple Arithmetic Language. The paper summarizes results of the studies of implemented parser generator and describes its role in the university courses.

Keywords: parser generator; annotated model; abstract syntax; model to grammar transformation.

1. Introduction

Computer languages are crucial tools in the development of software systems. By using computer languages we define the structure of a system and its behavior. Today's common industry practice is to create a software system as a composition of software artifacts written in more than one computer language. Developers use different languages and paradigms throughout the development of a software system according to a nature of concrete subproblem and their preferences. Besides the general-purpose programming languages (e. g. Java, C#) the domain-specific languages (DSL) [1][2] have become popular in the last decade. Nowadays, DSLs have their stable position in the development of software systems in many different

* This work is supported by APVV Grant No. SK-CZ-0095-07 – Cooperation in Design and Implementation of Language Systems and by VEGA Grant No. 1/4073/07 – Aspect-oriented Evolution of Complex Software System.

forms. Concerning abstraction level, it is possible to program closer to a domain. Furthermore DSLs enables explicit separation of knowledge in the system in natural structured form of domain. The growth of their popularity is probably connected with the growth of XML technology and using of standardized industry XML document parsers as a preferable option to the construction of language specific processors. A developer with minimal knowledge about language parsing is able to create a DSL with XML compliant concrete syntax using tools like JAXB [3].

Computer languages come in many flavors – as well known GPLs, DSLs, but also as APIs, ontologies [4], and even others. The one of the today's hottest research topics in the field of computer language development is the tooling support. In the paper we concentrate on the parser generators for DSLs. Even though the research in the field of computer languages has the long history and parser generators for a textual language processing like YACC [5], Bison [6], JavaCC [7] and ANTLR [8] have their stable position in the computer language development the task of developing a computer language is still an expert task. Cook et al. [9] conclude that implementing a textual DSL by implementing its grammar can be a difficult and error-prone task, requiring significant expertise in language design and the use of a parser generator. Similarly, Mernik et al. [1] argue that DSL development is hard, requiring both domain knowledge and language development expertise. We present the novel method of a computer language design and implementation in the paper – *abstract syntax driven parser generation*.

The rest of the paper has the following structure: In the section 2 we present main ideas behind our approach to a computer language development. The section 3 explains the method on example of simple but extensible arithmetic language. Section 4 describes the parser generator prototype –*YAJCo*. Section 5 summarizes the results of our experiments with *YAJCo* parser generator. Section 6 compares our work with the state of the art in the field of parser generators. The last section 7 concludes the paper and outlines the possibilities for further research in the field of parser generators and computer language development in general.

2. Abstract Syntax Directed Language Definition

This section sketches the main ideas behind the innovative approach to the definition of a concrete syntax for a computer language with textual notation. Contrary to traditional methods of parser generation (e. g. YACC, JavaCC), we focus on the definition of abstract syntax rather than giving an excessive concentration on concrete syntax (see Fig. 1). In our approach the abstract syntax of a language is formally defined using standard classes well known from object-oriented programming and metamodels. Kleppe argues for concentrating on abstract syntax and metamodels when we define a computer language in [10].

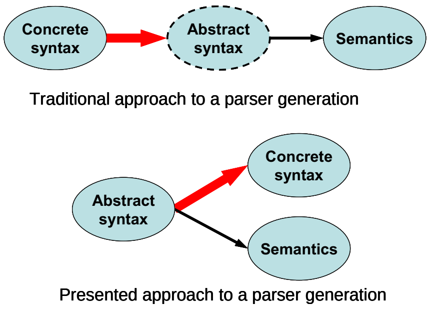

Concrete syntax
Abstract syntax
Semantics
Traditional approach to a parser generation
Concrete syntax
Abstract syntax
Semantics
Presented approach to a parser generation

Fig. 1. Comparing traditional and presented approaches to a computer language parser generation

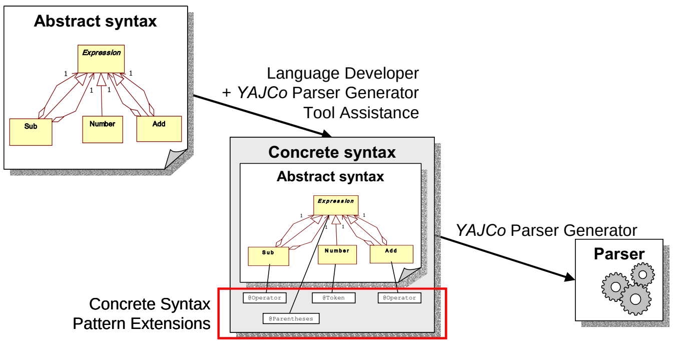


Fig. 2. Generating Language Parser using YAJCo's parser generation approach

In our approach the language implementation begins with the concept formalization in the form of abstract syntax. Language concepts are defined as classes and relationships between them. Upon such defined abstract syntax a developer defines both the concrete syntax through a set of source code annotations and the language semantics through the object methods. Annotations (called also attributes [11]) are structured way of additional knowledge incorporated directly into the source code. During the phase of concrete syntax definition the parser generator assists a developer with suggestions and hints for making the concrete syntax unambiguous. Fig. 2 shows the whole process of parser implementation using the described approach. If the concrete syntax is unambiguously defined then parser generator automatically generates the parser from annotated classes.

It is quite common to have multiple notations for one language. RELAX NG [12] is an example of such a language with two different notations – XML concrete syntax and compact concrete syntax. By using our approach different notations of the same language can share both abstract syntax and semantics. In some cases the evolution of concrete syntax does not require the modification of abstract syntax and semantics at all. This means that other notations of the same language are not affected by this type of language evolution. For instance, Fig. 3 presents the language with four different notations sharing the same abstract syntax and semantics. These notations (concrete syntaxes) are textual notation, XML notation, in-memory object notation and graphical notation. Concrete notations are interchangeable and developer selects among them according to his preferences.

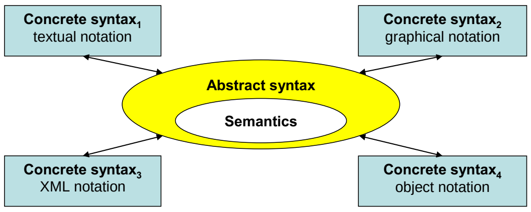


Fig. 3. Computer language with multiple notations and shared abstract syntax and semantics

3. SAL Example

This section presents our approach to a computer language definition using annotated classes on the example of Simple Arithmetic Language (SAL). This language expresses the arithmetic expressions with basic arithmetic binary operations of addition and multiplication, and unary operation of arithmetic negation. The expressions also contain integer numbers. The abstract syntax of SAL can be formally defined using BNF as follows.

$e, e_1, e_2 \in \text{Expression}, n \in \text{Number}$
$e ::= \text{Number } n \mid \text{UnaryMinus } e \mid \text{Add } e_1\ e_2 \mid \text{Mul } e_1\ e_2$

Variables e, e_1, e_2 are metavariables from the **Expression** syntactical domain and n is the metavariable from the **Number** syntactical domain. The infix form of arithmetic operation is intentionally omitted to avoid the confusion with concrete syntax. Prefix names in productions (e. g. UnaryMinus, Mul) are used just to uniquely name the productions for semantic equations.

The semantics of SAL is formally defined using *Eval* function which maps a value from syntactic domain **Expression** to a value from semantic domain **Z** (integers) and *Value* function which maps a value from syntactic domain **Number** to a value from semantic domain **Z**.

$Eval$: Expression $\rightarrow$ Z

$Value$: **Number** $\rightarrow$ **Z**

The semantic function *Eval* is defined by the following equations.

$Eval \llbracket \text{Number } n \rrbracket = Value \llbracket n \rrbracket$

$Eval \llbracket \text{UnaryMinus } e \rrbracket = - Eval \llbracket e \rrbracket$

$Eval \llbracket \text{Add } e_1\ e_2 \rrbracket = Eval \llbracket e_1 \rrbracket + Eval \llbracket e_2 \rrbracket$

$Eval \llbracket \text{Mul } e_1\ e_2 \rrbracket = Eval \llbracket e_1 \rrbracket * Eval \llbracket e_2 \rrbracket$

Certainly we can find many different notations for SAL. For example, we can write down a sentence from SAL in the following notation using standard symbols and the operator infix form.

```
1 + 2 * 7
```

In the Fig. 4, abstract syntax tree of the sentence above is depicted.

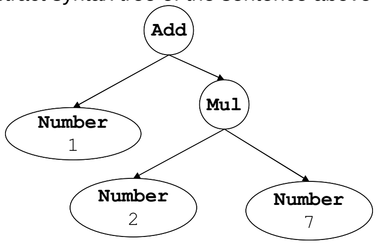


Fig. 4. The abstract syntax tree of the expression `1 + 2 * 7`

From the Fig. 4 it is apparent that depicted abstract syntax tree contains typed nodes corresponding to language concepts. The node types are Add, Mul and Number. Add node represents the binary operation of addition. It always has two child nodes respecting the nature of the binary operation of addition. The Mul node represents multiplicative operation and the leaf node Number represents an integer number. The Number node is attributed with the notation of a number.

Unlike traditional approach, language definition will not start with the definition of SAL's concrete syntax written in BNF. According to our approach, the object classes representing syntactic domains (language concept) are created at first. These classes define the abstract syntax of the language and also the semantics of the language as stated in the previous formal definition of the SAL. The concrete syntax will be specified later using source code annotations, expressing the concrete syntax patterns and their correspondence to the abstract syntax concepts.

The main concept of the SAL is the `Expression`. It is pretty straightforward because SAL is the language of expressions. On the other side it is an abstract concept and it does not have concrete representation. From the semantic point of view every expression can be evaluated to a single integer value. This fact is denoted by semantic function `eval` of `Expression` class.

```
abstract class Expression {
  //Semantic function – OOP method
  abstract int eval();
}
```

The `Expression` class is declared to be abstract because it only defines the abstract concept of an expression from SAL and does not represent any abstract syntax graph node. Next, the different types of expressions can be incorporated into the SAL. The simplest form of an expression is a number expression. Number has its notation and the value. Firstly we will focus is on its value. The notation will be defined later during the definition of the concrete syntax. It needs to be expressed that number is a simple expression as well. This is done using “is-a” relationship, denoted with **extends** keyword in Java. Corresponding semantic equations are denoted in the comments above the methods. The code snippet below shows the class `Number` for integer numbers.

```
class Number extends Expression {
  int value;

  //Eval [| Number n |] = Value [| n |]
  int eval() {
    return value;
  }
}
```

The unary operation of negation is defined in the following snippet of the `UnaryMinus` class.

```
class UnaryMinus extends Expression {
  Expression expression;

  //Eval [| UnaryMinus e |] = – Eval [| e |]
  int eval() {
    return -expression.eval();
  }
}
```

Since the addition is a kind of arithmetic expression in SAL, the binary operation of addition is defined in the class `Add`. Relationship “is-a” is therefore used again.

```
class Add extends Expression {
  Expression expression1;
  Expression expression2;

  //Eval [| Add e1 e2 |] = Eval [| e1 |] + Eval [| e2 |]
  int eval() {
    return expression1.eval() + expression2.eval();
  }
}
```

Operation of multiplication is defined in the same style as binary operation `Add`.

The class diagram in the Fig. 5 shows the hierarchy of SAL classes. The abstract syntax of arithmetic expression language has already been defined as well as the semantic function *Eval* using the classic OOP notation. The next step in the development of SAL is to define the concrete syntax for the language. Concrete syntax will be used when expression (sentence) will be stored in the textual form.

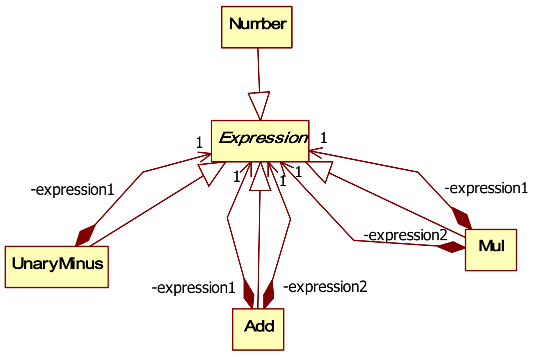


Fig. 5. Classes and their hierarchy in the simple arithmetic language (SAL)

The specification of concrete syntax requires some additional information about textual representation of the language concepts. In SAL it is:

- a number representation (notation),
- notation for operations,
  - symbols for the operations of addition, multiplication and negation,
  - the form of the notation, the priority and associativity of all operations.

The operations will be expressed in infix form using standard symbols `+` and `*`. Unary operation of negation will be in the prefix form denoted with the symbol `-`. The priority, associativity and symbols for the operations are listed in Table 1. The integer numbers are written using standard decimal notation with digits `0`, `1`, …, `9`.

Table 1. Priority and associativity of SAL operators

Operator	Priority	Associativity
+	1 (lowest)	left
*	2	left
-	3 (highest)	right

The class for integer numbers is augmented with concrete syntax source annotations in the following code snippet.

```
class Number extends Expression {
  int value;

  Number(@Token("VALUE") long value) {
    this.value = value;
  }

  int eval() {
    return value;
  }
}
```

The `@Token` annotation with `VALUE` attribute defines the name of a regular expression for the number notation. As seen on the snippet the class constructor is augmented with the concrete syntax pattern. The regular expression can be defined as follows.

```
@TokenDef(name = "VALUE", regexp = "[0-9]+")
```

The format of a regular expression depends on the syntax for definition of regular expressions. The annotation `@Token("VALUE")` can even be omitted because the name of token can be derived directly from the name of the parameter (`value` in this case). The domain class for binary operation of addition augmented with concrete syntax annotations is shown below.

```
class Add extends Expression {
  Expression expression1;
  Expression expression2;

  @Operator(
    associativity = Associativity.LEFT,
    priority = 1
  )
  Add(Expression expression1,
      @Before("+")
      Expression expression2) {
    this.expression1 = expression1;
    this.expression2 = expression2;
  }

  int eval() {
    return expression1.eval() + expression2.eval();
  }
}
```

Concrete syntax for the operation of addition is defined in the class constructor. Parameters of constructor define the rule of composition of the operation. In the constructor body it can be observed that addition is composed of two expressions in textual form. It is important to notice that after the first expression (and before the second expression at the same time) token + will follow.

Binary operation of multiplication is defined accordingly to the definition of addition. The domain class for unary operation of arithmetic negation is augmented with concrete syntax annotations as shown in the code snippet below.

```
class UnaryMinus extends Expression {
  Expression expression;

  @Operator(priority = 3)
  UnaryMinus(
      @Before("-")
      Expression expression) {
    this.expression = expression;
  }

  int eval() {
    return -expression.eval();
  }
}
```

As seen in the constructor the operation is defined as unary prefix operation.

The last step in definition of the SAL's concrete syntax is the definition for parentheses. This can be achieved simply by using the annotation on abstract class for expressions as shown below.

```
@Parentheses(left = "(", right = ")")
  abstract class Expression {
    //...
}
```

Finally the concrete syntax for the language has been defined. The implemented *YAJCo* parser generator generates the language parser from annotated classes. The concrete syntax of SAL is automatically derived from these classes, their relationships and concrete syntax annotations. In the current implementation of the *YAJCo* it is the following LL(1) context-free grammar.

```
Expr1 ::= Expr2 {"+" Expr2}
Expr2 ::= Expr3 {"*" Expr3}
Expr3 ::= "-" Expr3 | Expr
Expr ::= Number | "(" Expr1 ")"
Number ::= [0-9]+
```

4. *YAJCo* Parser Generator

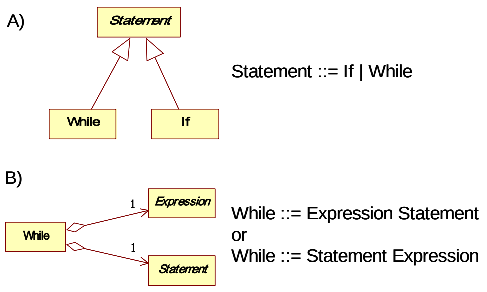
The main goal of the approach is not to create a new parsing technology based on context-free grammars theory. The main idea is to integrate existing technologies into the higher level abstraction in which the language developer does not have to concentrate on concrete parsing technology but on the language itself describing the concepts and relationships between them with abstract syntax in mind. The main characteristics of the approach are:

- Orientation on abstract syntax and semantics of the language.
- Definition of the concrete syntax independent from a parsing technology.
- Automatic construction of abstract syntax tree from an input sentence.
- Automatic construction of references between concept instances.
- Error reporting in terms of language domain concepts.
- Separation of language concepts on implementation level (concept types).
- Tool support for language evolution (concept refactoring).

As a proof of concept the parser generator *YAJCo* (Yet Another Java Compiler cOmpiler) has been implemented. *YAJCo* generates language parser from annotated classes. It is implemented as a standard Java annotation processor which traverses through the source code of classes looking for cocnrete syntax pattern annotations. *YAJCo* discovers relations between classes. Two main relationships between classes used in the definition of an abstract syntax are:

- “is-a” relationship,
- “has-a” relationship.

Together with corresponding BNF productions they are depicted in the Fig. 6.



Statement ::= If | While

While ::= Expression Statement
or
While ::= Statement Expression

Fig. 6. Abstract syntax relationships: A) “is-a” relationship, B) “has-a” relationship

The “is-a” relationship is used also in the definition of concrete syntax, but the “has-a” relationship has following drawbacks when defining the concrete syntax:

- Multiple notations for a single concept.
- Lack of natural ordering for member variables defined in a class (except the order in a source code).
- Data type conversion between concrete and abstract syntax (e. g. dropping the quotes from the string literal).

All these drawbacks can be eliminated by using class constructor notation (or factory methods notation) for the definition of concrete syntax. This is the main reason why we annotate constructors and their parameters instead of object fields as shown in the following example.

```
While(
  @Before({"while", "("})
  @After(")")
  Expression expr,
  Statement stmt) {…}
```

The previous example corresponds to the following BNF production of a concrete syntax.

```
While ::= 'while' '('Expression ')' Statement
```

To define a transformation from abstract to concrete syntax a set of concrete syntax annotation types has been created:

- Structural annotations – mark the concept as optional or set the minimum and maximum number of occurrences - `@Optional, @Range`
- Token annotations – specify binding of lexical units to abstract syntax concepts - `@Before, @After, @Token, @Separator`
- Language pattern annotations – identify common computer language patterns
  - Operators: `@Operator, @Parentheses`
  - Identifiers and references: `@Identifier, @References`
- Parser configuration annotations - `@Parser, @TokenDef, @Skip`

Following print statement example presents the usage of some of the annotations mentioned above.

```
class Print extends Statement {
  @Before("print")
  @After(";")
  Print(
    @Separator(",")
    @Range(minOccurs = 1)
    Expression[] expressions) { ... }
...
}
```

The corresponding `print` language concept has the following notation.

print $expr_1$, ... , $expr_n$;

The next example presents annotated C language `if` statement.

```
class If extends Statement {
  If(
    @Before({"if", "("})
    @After(")")
    Expression expression,
    Statement trueStatement,
    @Optional
    @Before("else")
    Statement falseStatement) {...}
  ...
  }
```

Currently the JavaCC parser generator is used as the underlying parsing technology. As an output *YAJCo* generates JavaCC grammar file augmented with actions for constructing abstract syntax tree. Since the annotations are independent of concrete parsing technology the output can also be generated for other top-down or bottom-up parser generators (e. g. ANTLR [8]).

Finally the parser for SAL with tokens and blank characters is defined using `@Parser` annotation as shown in the code snippet.

```
@Parser(
  className = "parser.expr.ExpressionParser",
  rootNode = "Expression",
  tokens = {
    @TokenDef(name = "VALUE", regexp = "[0-9]+")
  },
  skips = {
    @Skip(" "),
    @Skip("\t"),
    @Skip("\n")
  }
)
```

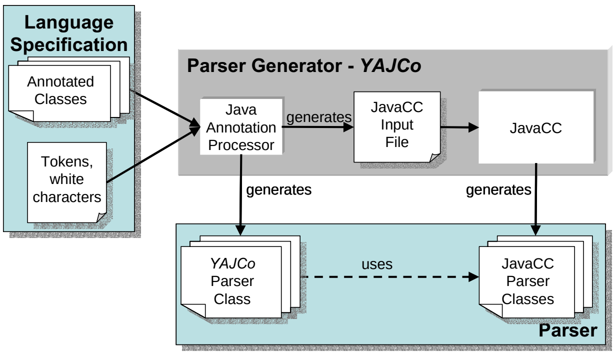


Fig. 7. Generating parser using YAJCo parser generator – YAJCo architecture overview

Processing of annotated classes with developed parser generator *YAJCo* is depicted in the Fig 7. After the generation of parser is complete it can be simply embedded in any existing Java application. The following code snippet is an example of embedding generated source code parser for SAL.

```
String expr = "1 + 2 * 7";
Expression expression = new
         ExpressionParser().parse(expr);
long result = expression.eval();
```

5. Experiments

To explore the full potential of the implemented approach and *YAJCo* parser generator we have implemented seven computer languages, each of them having a different character:

- SAL – Simple Arithmetic Language,
- AL – Arithmetic Language,
- SIL – Structured Imperative Language,
- PIL – Procedural Imperative Language,
- GUIIL – Graphical User Interface Interaction Language,
- SML – State Machine Language,
- LAD – Language of Annotation Designator.

SAL [17] and AL languages are simple computer languages for expressing the arithmetic expressions. AL has been created incrementally from the SAL in a few evolutionary steps. In every step some new constructs have been incorporated into the language. SIL and PIL languages are the representatives of general-purpose programming languages. PIL is procedural Pascal-like language. These languages are greatly inspired by traditional university compiler course languages. On the other hand the last three languages GUIIL, SML and LAD are DSL languages oriented to concrete domains. GUIIL is the language which describes the recipes for graphical user interface task automation. SML is classic DSL for state machines description [9]. LAD is DSL language for expressing the annotation constraints. All mentioned languages were successfully implemented using *YAJCo* parser generator. During the implementation of these computer languages we have also defined some metrics to measure the following implementation characteristics:

- number of language concepts (defined by types - classes, interfaces, enumerations),
- number of annotations in the implementation of language concepts categorized by annotation types,
- comparison of annotated and unannotated language concepts,
- characteristics of generated source code (number of source lines of code, number of characters).

The results of experiments are summarized in Table 2 According to our measurements the most complex language is PIL. This language contains the largest number of language concepts. From the point of view of the number of language concepts the simplest languages are SML and GUIIL. According to the results the most common language concept representation is a concrete class. Interfaces and abstract classes are interchangeable by the choice of language developer. The most common concrete syntax annotation used in experimental languages is `@Before`. This is a reasonable outcome since the annotation specifies the lexical symbol preceding a concept. It is natural to specify the concept with leading keyword (e.g. if, while, procedure). The interesting fact is that approximately 25% of language concept types contain no annotation. It is the fulfillment of the one of our aims **-** to minimize the number of used annotations. The results also show that the SML language is considerably verbose. The average number of concrete syntax annotations per concept type in SML is 2.5. The following part from the SML sentence presents the level of verbosity of the SML language.

Table 2. Results from the implementation of experimental languages using YAJCo parser generator

Types	**SAL**	**AL**	**SIL**	**PIL**	**GUIIL**	**SML**	**LAD**
Concrete class	6	11	30	42	4	6	26
Abstract class	1	3	2	3	1		7
Interface				2			
Enumeration			1				1
Total	7	14	33	47	5	6	34
Annotation	**SAL**	**AL**	**SIL**	**PIL**	**GUIIL**	**SML**	**LAD**
After		1	9	13	1	1	13
Before	5	10	28	37	2	8	23
Operator	5	10	17	27			8
Optional			1	1	1	1	3
Parentheses	1	1	1	1			2
Range			3			2	2
Separator			3	2	1		5
Token			2			3	2
Total	11	22	64	81	5	15	58
Category	**SAL**	**AL**	**SIL**	**PIL**	**GUIIL**	**SML**	**LAD**
Number of annotated types	6	11	25	39	2	4	23
Number of types without annotation	1	3	8	8	3	2	10
Average number of annotations per type	1.57	1.57	1.94	1.72	1.00	2.50	1.71
Ratio of unannotated types to all types	0.14	0.21	0.24	0.17	0.60	0.33	0.29
Characteristics	**SAL**	**AL**	**SIL**	**PIL**	**GUIIL**	**SML**	**LAD**
Number of lexical units	8	16	37	40	7	11	37
Number of BNF rules	5	7	24	27	5	6	29
Number of source lines of code generated by *YAJCo*	128	187	570	655	124	168	693
Number of characters generated by *YAJCo*	2458	3775	15912	17554	2854	4245	19559
Number of source lines of code generated by	1487	1603	2567	2580	1516	1843	3849

JavaCC							
Number of characters generated by JavaCC	42103	45370	78450	78687	43869	52950	114002

```
transition from Ready to Running when water_high
```

The average usage of concrete syntax annotation per one concept type in all languages is less than 2. The main goal of the metrics definition was the measurement of a language complexity based on abstract syntax since abstract syntax directly defines concepts from a domain.

The successful implementation of experimental languages proves the viability of *YAJCo* parser generator. That was the main reason why we decided to incorporate the tool in the university master course concerning DSL implementation and model driven software development. More than 30 students have successfully used the *YAJCo* parser generator as a part of their projects.

6. Related Works

Currently there are a lot of parser generators for various programming languages [5][6]. Classic parser generators like JavaCC [7] generate the parser as a single huge class ignoring the concept of composition of language concepts and concentrating on the concrete syntax of a language. These tools are still greatly inspired by procedural nature of YACC-like tools. The concrete syntax is specified in DSL of parser generator. It is usually a language for writing the context free grammar enriched with constructs for language semantics definition. During language development the developer is often dealing with the type of parsing algorithm which is supported by selected parser generator (e. g. LL, LR, LALR) and his decisions are forced by the type of grammar supported by the tool. Even JJTree, a tool provided by JavaCC for generating the abstract syntax tree from the textual representation, is still driven by the point of view of concrete syntax grammar rather than abstract syntax language concept. Consequently, changes made to grammar must be also reflected in the representation of abstract syntax nodes in programming languages. The semi-automatic refactoring of generator’s DSL is still missing.

On the other side, there is a notable growth in the field of language workbenches [13] on the market. MDSD [14] tools like Microsoft Visual Studio DSL Tools (software factories representative [15]) are being incorporated into the programming IDEs. The primary orientation of these tools is graphical notation of computer languages. However, the special support for textual language notation is not provided.

Authors in [16] propose another approach to mapping from abstract syntax to concrete and back. Their solution is based on complex language rather than concrete syntax patterns.

7. Conclusion

In the paper we have presented solution for generating parsers for textual languages. The language itself is specified by a set of annotated classes. Annotations extend the classes with additional information required for specification of concrete syntax, for example keywords and operator notations. The developer can start with the definition of abstract syntax and continue with creation of language in incremental way using the standard refactoring tools. In proposed solution there is only one form of definition of abstract syntax graph nodes – by the classes. The grammar is derived directly from the source code of annotated domain classes. Even the examples are written in object-oriented programming language Java our solution is not strictly connected to Java language and can be easily ported to any other object-oriented language supporting the attribute-oriented programming. We believe that our solution can simplify the development of textual software languages.

References

1. Mernik, M., Heering, J., Sloane, A. M.: When and How to Develop Domain-Specific Languages. ACM Computing Surveys, Vol. 37, No. 4, 316–344. (2005)
2. Pereira, M. J. V., Mernik, M., da Cruz, D., Henriques, P. R.: Program Comprehension for Domain-Specific Languages. ComSIS, Vol. 5, No. 2. (2008)
3. Ort, E., Mehta, B.: Java Architecture for XML Binding. Sun Microsystems, [Online]. Available: http://java.sun.com/developer/technicalArticles/WebServices/jaxb (current November 2009)
4. Návrat, P., Bieliková, M., Chudá, D., Rozinajová, V.: Intelligent Information Processing in Semantically Enriched Web. Lecture Notes in Computer Science, Vol. 5722/2009, 331-340. (2009)
5. Johnson, S. C.: YACC: Yet Another Compiler-Compiler. Unix Programmer's Manual Volume 2b. (1979)
6. Donnelly, C., Stallman, R.: Bison: The Yacc-compatible Parser Generator. (2006).
7. Java Compiler Compiler – The Java Parser Generator, (2009). [Online]. Available: https://javacc.dev.java.net (current November 2009)
8. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages, Pragmatic Bookshelf, 376 pp. (2007)
9. Cook, S., Jones, G., Kent, S., Wills, A. C.: Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley Professional, 576 pp. (2007)
10. Kleppe, A. G.: A Language Description is More than a Metamodel. In: Fourth International Workshop on Software Language Engineering, 1 Oct 2007, Nashville, USA.
11. Cepa, V.: Attribute Enabled Software Development, VDM Verlag, 216 p. (2007)
12. van der Vlist, E.: Relax NG. O'Reilly Media, 304 pp. (2003)
13. Fowler, M.: Language Workbenches: The Killer-App for Domain Specific Languages? (2005) [Online]. (current November 2009) Available: http://www.martinfowler.com/articles/languageWorkbench.html
14. Stahl, T., Voelter, M.:Model-Driven Software Development: Technology, Engineering, Management. Wiley, 444 p. (2006)

15. Greenfield, J., Short, K., Cook, S., Kent, S., Crupi, J.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, 500 p. (2004)
16. Muller, P. A., Fondement, F., Fleurey, F., Hassenforder, M., Schneckenburger, R., Gérard, S., Jézéquel, J. M.: Model-Driven Analysis and Synthesis of Textual Concrete Syntax. Journal on Software and Systems Modeling (SoSyM), Volume 7 (4), Springer, 423-441. (2008)
17. Kollár, J., Václavík, P., Wassermann, Ľ.: Data driven Executable Language Model. In Proceedings of the International Multiconference on Computer Science and Information Technology, Mragowo, Poland, IEEE Computer Society Press, 667-675. (2009), ISSN 1896-7094.

Jaroslav Porubän is Associate professor at Department of Computers and Informatics, Technical university of Košice, Slovakia. He received his MSc. in Computer Science in 2000 and his PhD. in Computer Science in 2004. Since 2003 he is the member of the Department of Computers and Informatics at Technical University of Košice. He was involved in the research of profiling tools for process functional programming language. Currently the main subject of his research is the computer language engineering concentrating on design and implementation of domain-specific languages and computer language composition and evolution.

Michal Forgáč is Assistant professor at Department of Computers and Informatics, Technical university of Košice, Slovakia. He received his MSc. in 2006 and his PhD. in Computer Science in 2009. Since 2009 he is the member of the Department of Computers and Informatics at Technical University of Košice. His scientific research is focused on the software evolution, software language engineering and adaptation of complex software systems.

Miroslav Sabo is doctoral student at Department of Computers and Informatics, Technical University of Košice, Slovakia. He received his MSc. in Computer Science in 2008. The subject of his research is the utilization of generative methods in development and evolution of software systems in permanently changing environment.

Marek Běhálek is Assistant professor at Department of Computer Science, FEI VŠB Technical University of Ostrava, Czech Republic. He received his MSc. in 2002. Since 2004 he is the member of the Department of Computer Science at Technical University of Ostrava. His scientific research is focused on programming languages, their evolution and application. Currently he is developing a tool for modeling of embedded systems based on functional programming paradigm.

Received: November 16, 2009; Accepted: December 25, 2009.

UDC 004.4, DOI: 10.2298/CSIS1002309C

On Automata and Language Based Grammar Metrics

Matej Črepinšek[1], Tomaž Kosar[1], Marjan Mernik[1],
Julien Cervelle[2], Rémi Forax[2], Gilles Roussel[2]

[1] University of Maribor, Faculty of Electrical Engineering and Computer Science,
Smetanova 17, 2000 Maribor, Slovenia
Email: {matej.crepinsek, tomaz.kosar, marjan.mernik}@uni-mb.si
[2] Université Paris-Est, Laboratoire d'Informatique Gaspard-Monge,
77454 Marne-la-Vallée, France
Email: {julien.cervelle, remi.forax, gilles.roussel}@univ-mlv.fr

Abstract. Grammar metrics have been introduced to measure the quality and the complexity of the formal grammars. The aim of this paper is to explore the meaning of these notions and to experiment, on several grammars of domain specific languages and of general-purpose languages, existing grammar metrics together with the new metrics that are based on grammar LR automaton and on the language recognized. We discuss the results of this experiment and focus on the comparison between grammars of domain specific languages as well as of general-purpose languages and on the evolution of the metrics between several versions of the same language.

Keywords: grammar metrics, software language engineering, grammar engineering, grammarware.

1. Introduction

Grammar metrics were introduced to measure the quality and complexity of a given grammar in order to orient grammar engineering (grammarware [17]). We consider that existing metrics [23], more or less deduced from classical program metrics or from the structure of the specification, could be upgraded with new metrics that are specific to grammar behavior and could provide additional insights about the complexity of the grammar and the language generated by this grammar. Of course, a single metrics alone cannot capture the quality of the grammar, however a set of well chosen metrics could give interesting hints to grammar developers.

In order to complete the existing set of metrics, we propose two different kinds of metrics[3]. A first set of metrics is computed from the LR automaton,

[3] This work is sponsored by bilateral project "Advanced Topics in Grammar Engineering" (code BI-FR/08-09-PROTEUS-008) between Slovenia and France.

generated from the grammar. A second set is more related to the language recognized, than to the grammar itself. These different kinds of metrics produce results that are complementary for the grammar developers.

In order to compute these metrics, we have developed a tool. As an input, it takes ANTLR [22] or Tatoo [4, 5] grammars and computes classical metrics [23] together with our new metrics. It uses Tatoo engine to construct the LR automaton for these grammars.

Using this tool we have computed the values of these metrics on several grammars that form a good benchmark of grammars. These grammars cover domain specific languages (DSL [20]) and general-purpose languages (GPL [28]). They also cover the evolution of a grammar between different versions of the language. From these experimentations, we discuss the different values of the metrics.

The structure of the paper is as follows. Presented in the Section 2 is related work and existing metrics. In Section 3, the new metrics are defined. Section 4 describes the tool and how it is linked to Tatoo. In Section 5, experimental results on the grammars are detailed and discussed. In Section 6, some usage guidelines are presented. Section 7 carries conclusions and remarks. In the appendix, the computation of the closure application of rules is detailed.

2. Overview of Related Work

In the field of grammar metrics, only a few tools and papers exist. The most pertinent of these tools is *SynC tool* by Power and Malloy [23]. In *SynC tool*, grammar metrics are divided into size and structural metrics. In the first metrics group, an adaptation of standard metrics for programs [13], the following grammar size metrics are defined [23]:

- `term` – number of terminals,
- `var` – number of non-terminals,
- `mcc` – McCabe cyclomatic complexity,
- `avs` – average size of right hand side, and
- `hal` – Halstead effort.

Size metrics feature useful information about the grammars. More maintenance is expected for grammars with large numbers of non-terminals (`var`). The `mcc` provides the number of alternatives for non-terminals. The `mcc` value indicates the effort required for grammar testing and a greater potential for parsing conflicts. A big `avs` value points to less readable grammar as well as it impacts on the parsers' performance, because symbols have to be placed on the parser stack. The `hal` value evaluates grammar designers' efforts to understand the grammar.

Structural metrics for grammars are derived from grammatical levels [8], where a grammar is represented as a graph. In the graph, the nodes are non-terminals and the edges represent a successor relationship between a left hand side non-terminal and a non-terminal on the right hand side. In order to compute structural metrics, we compute the strongly connected components of the graph, which leads to a partition of the set of non-terminals into *grammatical levels*. We use the following structural metrics, as defined in [23]:

– `timp` – tree impurity,
– `clev` – normalized counts of levels,
– `nslev` – number of non-singleton levels,
– `dep` – size of largest level, and
– `vhei` – Varju height metrics.

Tree impurity (`timp`) measures how much the graph resembles a tree (0% – graph is a tree, 100% – graph is fully connected). A high `timp` value for a grammar means that refactoring, the grammar will be complicated, since a change in one rule may impact many other rules. A normalized count of levels (`clev`) is the normalization of the number of grammatical levels by the total number of non-terminals expressed in percentage. A high `clev` indicates more opportunities for grammar modularization. Many of the equivalence classes are of size 1, while language concepts such as declarations, expressions, and commands tend to be represented by larger classes. The (`nslev`) metrics identifies the number of such classes. The size of the largest level (`dep`) metrics measures the number of non-terminals in the largest grammatical level. A high `dep` indicates an uneven distribution of the non-terminals among grammatical levels. The Varju height metrics (`vhei`) is the maximum distance of any non-terminal from the start symbol, and is expressed as a percentage of the number of equivalence classes.

Paper [3] presents a methodology for iterative grammar development. Well-known techniques from software engineering are applied to the development

– version control,
– grammar metrics,
– unit testing, and
– test coverage analysis.

It demonstrates how these techniques can make grammar development a controlled process. As mentioned above, one of the techniques used involves grammar metrics. Authors use size and structural metrics defined in [23] and extend them with disambiguation metrics, which are SDF [15] specific:

– `frst` – number of follow restrictions,
– `rejp` – number of reject productions,
– `assoc` – number of associativity attributes, and
– `upp` – number of unique productions in priorities.

These metrics merely count the various types of disambiguation in the SDF notation. Aside from Halstead's effort metrics, some of the ingredient metrics and related metrics are presented and used for grammar engineering. In a similar manner, as done in this work, we propose new metrics, which brings additional insights into grammar development.

One of the applications for grammar metrics is also in the field of grammar testing. The concept of grammar testing is explained in [18]. This paper presents context-dependent branch coverage on parser testing and grammar recovery; it proposes new tests for checking the accuracy and the completeness of grammars. We believe that our grammars' metrics could be a valuable contribution to the field of grammar testing, used as effort estimation in grammar engineering (i.e., software engineering applied to grammars).

3. Proposed New Metrics

In this section, we describe in detail two new kinds of metrics, LR table-based metrics and generated-language based metrics.

3.1. LR Table Metrics

The first set of metrics is based on the LR automaton that is used to produce an efficient bottom-up parsers for the grammar, but could also simulate top-down parsing [25], comparable to LL parsers. It is surprising that information given by this automaton has never been used before to qualify grammars.

The LR states are built using the following algorithm. A more detailed description of this algorithm can be found in [2].

First, the grammar is increased by adding a new production

$$X \to S\,{}^{\mathsf{E}}\mathsf{O}_{\mathsf{T}}$$

where S is the start symbol of the grammar, X a fresh non-terminal which becomes the new axiom and ${}^{\mathsf{E}}\mathsf{O}_{\mathsf{T}}$, a fresh terminal which symbolizes the end of input.

$$E \to (E)|E+E|E-E|-E|\mathrm{id}$$

Fig. 1. Grammar G_1

States of the LR automaton are defined by a set of items. An item is a production where an inter-letter space is marked (usually with a dot) on the right-hand side. The set of items defines all of the productions that can be found at this stage of the parsing. For instance, for the grammar G_1 described in Fig. 1, after reading «$(E+E$» (the E means that a word derived from E is recognized) the state contains:

$$
\begin{array}{l}
E \rightarrow E \cdot +E \\
E \rightarrow E \cdot -E \\
E \rightarrow E + E \cdot
\end{array}
$$

The item $E \rightarrow E + E\cdot$ indicates that «$E + E$» has been read and will be considered as a single E, while the item $E \rightarrow E \cdot -E$ indicates that the second E is part of an $E - E$ expression that will be considered as a single E.

Note that the information that a «(» has been read, is kept in the state stack of the parser, not in the LR state.

Only the initial state contains the item:

$$X \rightarrow \cdot S\,^{\mathsf{E}}\mathsf{O}_{\mathsf{T}}$$

States are built by applying a creation rule to existing states, until a new state cannot be built.

To explain this rule, we first define the closure $\mathcal{C}(I)$ of an item I as the smallest set to verify the following set of equations, where $\mathcal{P}$ is the set of productions of the grammar:

– $I \in \mathcal{C}(I)$;
– $\forall E \rightarrow \alpha \cdot X\beta \in \mathcal{C}(I)$ and $\forall X \rightarrow \gamma \in \mathcal{P}$,
  then $X \rightarrow \cdot\gamma \in \mathcal{C}(I)$.

The closure of a state is defined as the union of the closure of its items.

Then, new states are built from a state St by applying the following rule: for each terminal or non-terminal $\mathfrak{v}$ such that an item $X \rightarrow \alpha \cdot \mathfrak{v}\beta$ is in $\mathcal{C}(\mathrm{St})$, the following state is created

$$\{Y \rightarrow \delta\mathfrak{v} \cdot \zeta | \, Y \rightarrow \delta \cdot \mathfrak{v}\zeta \in \mathcal{C}(\mathrm{St})\}$$

If $\mathfrak{v}$ is a terminal, we say that the state St can shift the terminal $\mathfrak{v}$.

The set of LR states for the grammar G_1 computed using previous algorithm is the following:

$$
\begin{array}{l}
\{X \rightarrow \cdot E\,^{\mathsf{E}}\mathsf{O}_{\mathsf{T}}\} \\
\{X \rightarrow E\,^{\mathsf{E}}\mathsf{O}_{\mathsf{T}}\cdot\} \\
\{E \rightarrow (\cdot E)\} \\
\{E \rightarrow - \cdot E\} \\
\{E \rightarrow \mathrm{id}\cdot\} \\
\{E \rightarrow (E)\cdot\} \\
\{X \rightarrow E \cdot {}^{\mathsf{E}}\mathsf{O}_{\mathsf{T}},\, E \rightarrow E \cdot +E,\, E \rightarrow E \cdot -E\} \\
\{E \rightarrow E + \cdot E\} \\
\{E \rightarrow E - \cdot E\} \\
\{E \rightarrow (E\cdot),\, E \rightarrow E \cdot +E,\, E \rightarrow E \cdot -E\} \\
\{E \rightarrow -E\cdot,\, E \rightarrow E \cdot +E,\, E \rightarrow E \cdot -E\} \\
\{E \rightarrow E + E\cdot,\, E \rightarrow E \cdot +E,\, E \rightarrow E \cdot -E\} \\
\{E \rightarrow E - E\cdot,\, E \rightarrow E \cdot +E,\, E \rightarrow E \cdot -E\}
\end{array}
$$

In the last state, for instance, the terminals $+$ and $-$ can be shifted.

From the possible metrics that can be extracted from the LR automaton we have chosen the following:

– The metrics `lrs` represents the number of states in the LR automaton. This number is 13 for the grammar G_1. This metrics captures the complexity of the grammar.
– The `lat` metrics sums, for each terminal in the grammar, the number of states in the LR automaton that does not lead to an error when this terminal is in the lookahead, and it is normalized by the number of terminals. This metrics gives an idea for each terminal of the probability to accept it during the parsing.
– Metrics `lat` can be further normalized by the number of states `lrs`. Metrics `lat/lrs` computes complexity of relations between the terminals and the states in LR tables.
– The metrics `lrtla` sums, for each state in the automaton, the number of terminals that, when they are in the lookahead, do not lead to an error in this state, and it is normalized by the number of states. It indicates the complexity of each state in the automaton. If we normalize metrics `lrtla` by the number of terminals we get metrics `lat/lrs`.
– The metrics `lcc` counts the number of LR conflicts. These conflicts are shift-reduce or reduce-reduce conflicts found in some of the states. The `lcc` metrics gives an insight into the complexity of the grammar. Indeed, the conflicts solved by priority/associativity rules could also be solved by rewriting the grammar instead of introducing unnecessary productions that degrade the readability of the grammar [2]. Note that conflicts may be implicitly resolved by many compiler generators. A shift-reduce conflict is resolved implicitly by choosing to shift over reduce. On the other hand a reduce-reduce conflict is resolved implicitly by choosing to reduce the rule that first appears in the grammar. However, every such conflict should be carefully studied and checked if default behavior is indeed appropriate. Conflicts certainly raise the complexity of the grammar and `lcc` metric captures it.

3.2. Generated Language Metrics

The second set of proposed metrics is based on some of the characteristics of the language recognized. The following metrics, discussed in more detail below, are proposed: `ss`, `ssm`, `ltps`, `ltpsm`, `ltpsa`, and `ltpsn`.

The metrics `ss` builds, for each production, the shortest sample that uses it and stores the average size of these samples. This metrics provides hints of the verbosity of the language produced by the grammar. The metrics `ssm` is the maximum size of the samples.

The shortest sample for a production is produced using a recursive algorithm. More precisely, the algorithm for computing the shortest sample using a production in a grammar consists in three steps. The first is the computation, from each non-terminal, of the shortest word made of terminals derived from this non-terminal. The second step is the computation, for any non-terminal N, of the shortest word generated by an axiom, that is only made of terminals and one occurrence of N, which we call the sequel of the *shortest word leading to* X. Details of these first two steps can be found in appendix. Once these first two steps are accomplished, to get the shortest sample using production $X \rightarrow \alpha$, one starts with the word w obtained in step two for non-terminal X, replace X by α in w and finally replace all remaining non-terminals with the shortest words computed in step one.

The word produced is still the shortest since, if a shorter one exists, either its derivation tree would lead to a shortest way to produce a word that contains X or the shortest words for non-terminals of α. Since the first two steps are done using a closure operation on rules, if only the shortest sample for one single production needs to be computed, one can save computation time using a lazy and dynamic programming style, as it is done in Tatoo.

For instance for grammar G_1, the set of the shortest samples produced by this algorithm is:

$$\{(\mathrm{id}), \mathrm{id} + \mathrm{id}, \mathrm{id} - \mathrm{id}, -\mathrm{id}, \mathrm{id}\}$$

The average size of these samples is $\texttt{ss} = 2.4$, while the maximum size of the shortest sample is $\texttt{ssm} = 3$.

The other metrics are only concerned with the sequences of two terminals (terminal pairs) that may be found in the language recognized by the grammar.

$$\begin{aligned} S &\rightarrow L | L.L \\ L &\rightarrow B | L\, B \\ B &\rightarrow 0 | 1 \end{aligned}$$

Fig. 2. Grammar G_2

For instance, the grammar for Knuth's binary numbers, described in Fig. 2, allows 8 different terminal pairs. Combinations are presented in Table 1, where the first column and the first row represent all grammar terminals (ter) and `true` or `false` on position (ter_i, ter_j) indicate that, there exists a sentence recognized by this grammar which contains the pair of terminals (ter_i, ter_j).

Table 1. Allowed terminal pairs

i/j	0	1	.
0	true	true	true
1	true	true	true
.	true	true	false

From the table of allowed terminal pairs, we have defined four different metrics:

- The metrics `ltps` computes the number of different terminal pairs acceptable in the language. In case of G_2 value of `ltps` metrics is 8.
- The metrics `ltpsm` computes the maximum number of different pairs for one terminal. In case of G_2 value of `ltpsm` metrics is 3, because after terminal 1 one can find three different terminals (the same as in the case of terminal 0).
- The metrics `ltpsa` computes, given a terminal, the average number of terminals that can directly follow this terminal. In case of G_2 the value of `ltpsa` metrics is $(3+3+2)/3 \approx 2.666$.
- The metrics `ltpsn` normalizes the metrics `ltps`, by the number of possible combinations of terminals and is presented as a percentage. In the case of G_2 the value of `ltpsn`, the metrics is $(3+3+2)/9 \approx 88.888\%$.

Table 1 is calculated directly from the grammar by computing, for all non-terminal X, the sets of first $F(X) = \{a|X \Rightarrow^* a\beta\}$ and last $L(X) = \{a|X \Rightarrow^* \beta a\}$ possibly derived terminals, where a is a terminal. From these sets, it is easy to calculate pairs from the right hand sides of productions. For all occurrences of two consecutive terminals or non-terminals $\mathfrak{v}_1$ and $\mathfrak{v}_2$, one adds all of the pairs of $L(\mathfrak{v}_1)F(\mathfrak{v}_2)$ where $L(a) = F(a) = a$ in case a is a terminal.

4. Tool Description

In this section, we present the *gMetrics* tool. This tool extracts information from grammars and calculates the metrics proposed by Power and Malloy [23] as well as the new ones, LR-based metrics and generated language-based metrics, proposed in the previous section. The global activity diagram of gMetrics is presented in Fig. 3.

The main objective of this tool is to extract from input grammars as much information as possible and perform as few modifications as possible in relation to the original grammar. Indeed, we would like to avoid potential metric disturbance and to process all metrics from the same specification.

We currently support the formats of compiler construction tools ANTLR version 3 [22] (an LL parser generator) and Tatoo [5] (a LR parser generator). The metrics are divided, as explained before, into four categories: size metrics, structural metrics, LR automaton-based metrics and generated language-based metrics. In practice, to reuse an existing grammar specification, one must take into account the grammar form (BNF, EBNF, CNF, etc.), the grammar type (LL, LR, LALR, IELR [9], etc.), the file format (a tool mainly dependent and potentially customized with semantics and other annotations) and the version of the tool. In this work, we limited ourselves to grammar specifications used by ANTLR and Tatoo. Moreover, automata-based metrics are calculated from LR automaton, despite that a particular grammar can be of different type (e.g.,

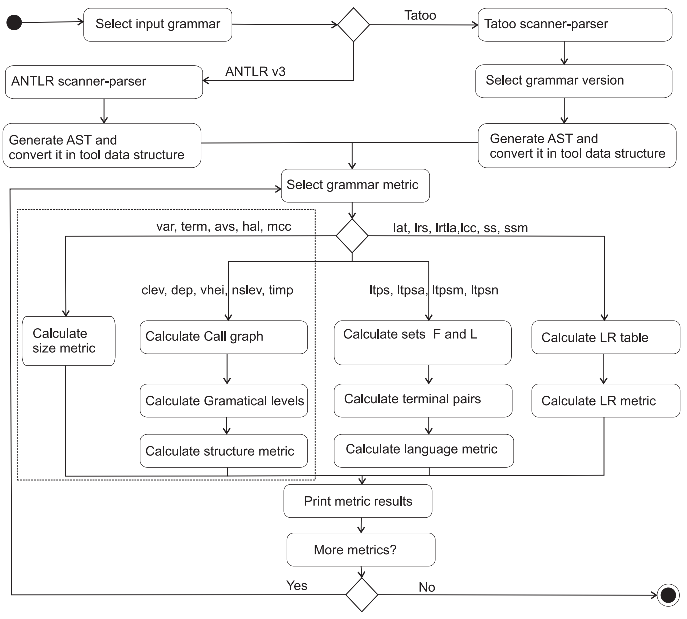


Fig. 3. Activity diagram

LL, LALR, IELR). Although, we report the number of shift-reduce and reduce-reduce conflicts that indicate that a grammar might not be the LR grammar (some conflicts are resolved by additional rules in parser generators). Our future work lies in identifying the correct type of grammar while calculating the automata-based metrics, as well to analyze the conflicts, as presented in [9].

Nevertheless, some transformations are unavoidable because the original input format of the grammars is different and some metrics make use of specific algorithms that require constrained input. However, `gMetrics` minimizes such transformations.

To solve the problem of a unique grammar representation, we have chosen to use an in-memory intermediate form close to an EBNF notation.

Indeed, even if neither ANTLR nor Tatoo uses complete EBNF form as input, the input format of each parser generator is close to this notation. Moreover, this format has the advantage to avoid to choose between left of right recursion in the specification, since lists may be specified using star ('*') or plus ('+') constructions.

The ANTLR format has been selected because it provides a great deal of interesting existing grammars and Tatoo was chosen because of its open architecture that facilitates the computation of some of the metrics. Both tools use priorities or/and associativity rules for solving automata conflicts.

In the future, we plan to broaden our tool to support other input formats (e.g., LISA [21]). Meanwhile, users may use this tool as a Java application library. In this case, users need to implement the `IGrammar` interface, which describes grammar in our EBNF internal form.

Because ANTLR and Tatoo are both implemented in Java, the simplest choice was to implement *gMetrics* in Java. The first challenge was to create and to fill internal data structure from ANTLR and Tatoo grammar specification. More precisely, in Tatoo, we directly used the memory representation exported by Tatoo. Moreover, since Tatoo supports grammar versioning, one input grammar may include several versions (usually specified in different grammars).

The metrics implementations are divided into four groups: size metrics, structural metrics, LR-based metrics and language-based metrics.

- To calculate size metrics, we implement a visitor pattern that counts different grammar properties.
- To compute structural metrics, the call graph is derived from the productions. It is then used to calculate grammatical levels. The structure metrics [8] are deduced from this information.
- To construct the LR automaton, information about the grammar associativeness (left or right) is first established. Next, the LR table is computed by the Tatoo, engine together with the associated LR actions.
- For the language-based metrics, terminal pairs are computed, as explained in the previous section.

When dealing with metrics implementation, we try to provide as much information for interpretation as possible. For most of the metrics we provide histograms, which are used to calculate concrete metrics values. These histograms can also be used for the computation and the analysis of different statistical values.

For example, in Fig. 4 we present the metric `ltps` histogram for the ANSI C grammar. The metric `ltps` describes, for a given terminal, the number of different terminals that can follow it. In this histogram one can notice that many terminals (34 in y axle) have between 11 and 20 (x axle) terminals that can follow them and only one terminal may be followed by almost all terminals. With this histogram one can monitor the introduction of a new terminal in the grammar.

The *gMetrics* tool is an open source project and can be found on the following web page http://code.google.com/p/cfgmetrics.

5. Empirical Study on Metrics for GPL and DSL Grammars

The grammar samples used for this experiment come from examples drawn from the ANTLR [22] samples and from several versions of Java grammars for

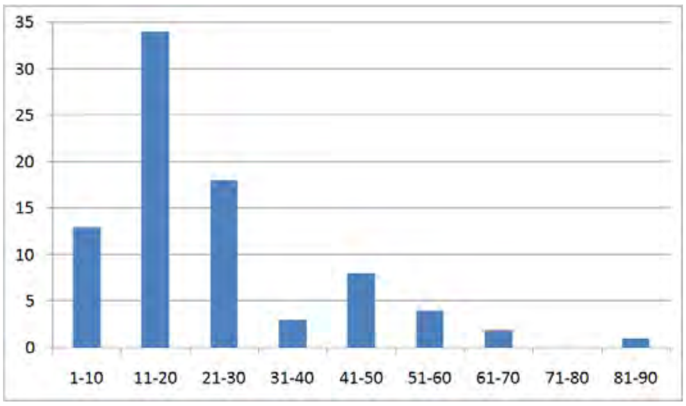


Fig. 4. Metrics `ltps` histogram for the ANSI C grammar

Tatoo [4, 5]. They cover domain-specific languages and general-purpose languages. These grammar examples are representative of current practice in grammar-ware engineering and can be considered as good benchmarks to evaluate the pertinence of metrics. They result from a collaborative work that usually involves several developers thereby ensuring their global quality.

More precisely, the DSL grammars studied are:

– EXPR, a grammar for arithmetic expressions [2].
– FDL, a grammar that enables the specification of sets of features [10].
– EBNF, a grammar for grammar specifications in Extended Backus Normal Form [1].
– CFDG, a grammar of a simple programming language for generating pictures [11].
– GAL, a grammar to describe video devices [26].
– ANTLR V3, a grammar for grammar definitions in ANTLR version 3 format [22].

General-purpose language grammars studied are those from

– Ruby 1.8.5 [27];
– ANSI C [14];
– Python 2.5 [19], and
– versions of Java [12] from 1.0 to 1.6.

The version 1.6 of the Java grammar comes from ANTLR samples, whereas the other versions come from Tatoo samples.

The results shown in Table 2 show the outcome of the size metrics for the different kinds of grammars. This table indicates that some of these metrics (e.g.,

Table 2. Results for classical metrics

Lang	term	var	mcc	avs	hal
Expr	9	5	1.6	4	1
FDL	14	6	2.17	6.5	2.63
EBNF	12	7	1.71	3.29	1.17
CFG Design	24	13	2.39	6	6.57
GAL	71	74	1.2	3.88	33.36
ANTLR V3	49	45	2.42	4.98	29.55
Ruby 1.8.5	88	83	2.61	4.74	54.44
Java 1.6	98	110	2.46	5.96	122.66
ANSI C	83	66	2.21	5.09	42.34
Python 2.5	85	86	2.22	4.93	63.41
Java 1.5	102	129	1.75	5.85	140.38
Java 1.4	100	116	1.75	5.8	118.21
Java 1.3	99	114	1.76	5.84	116.85
Java 1.2	99	114	1.76	5.84	116.85
Java 1.1	98	114	1.75	5.83	116.98
Java 1.0	98	112	1.63	5.38	98.54

mcc, avg) can not be used to differentiate DSL from GPL grammars. Some DSLs (e.g., GAL) are comparable to GPLs in relation to the number of terminals, nonterminals, and hal indicates that the size of such DSLs grammars can be comparable to GPLs.

It is noteworthy that all of the metrics for Java versions are extremely stable, with the exception of the mcc metrics, which is much larger for Java 1.6. This is without a doubt because it is an LL grammar, designed for ANTLR, whereas other versions of Java are LR, designed for Tatoo.

Table 3 provides the results of the structural metrics for the grammars. Among these results, it is interesting to note that the clev metrics indicates that the first versions of Java support have better modularization than the newer versions, probably due to new constructions such as internal classes. However, it is surprising that DSL, such as Expr, also has large values.

Most of the structural metrics are not very relevant where they concern the difference between DSL and GPL, with the exception of the dep metrics. However, there is marked variation between the values of this metrics between GPL, in particular in Java. This provides a good means for the interweaving of the grammar. From our point of view, these structural metrics are difficult to understand for grammar developers.

The results of Table 4 show that the values of the metrics lrs mainly depend on the type of the language. The DSL grammars have smaller values than GPL grammars, below 1000 states. Second, this metrics is not directly connected to the size of the grammar since grammars with similar numbers of terminals or non-terminals produce completely different values (e.g. Ruby vs. Python from

Table 3. Results for structural metrics

Lang	timp	clev	nslev	vhei	dep
Expr	56.25	60	1	3	3
FDL	32	83.33	1	1	2
EBNF	69.44	42.86	1	3	5
CFG Design	31.25	100	0	1	1
GAL	14.6	95.95	1	1	4
ANTLR V3	21.69	75.56	2	1	8
Ruby 1.8.5	62.37	37.35	1	1	53
Java 1.6	72.35	25.46	2	1	80
ANSI C	67.93	30.3	3	1	41
Python 2.5	44.39	46.51	3	1	35
Java 1.5	76.53	22.48	2	1	100
Java 1.4	59.59	40.52	2	1	69
Java 1.3	58.98	41.23	2	1	67
Java 1.2	58.98	41.23	2	1	67
Java 1.1	58.98	41.23	2	1	67
Java 1.0	33.29	65.18	3	1	24

Table 4. Results for LR based metrics

Lang	lrs	lat	lrtla	lat/lrs	lcc
Expr	59	18	3.05	0.31	0
FDL	115	20.13	2.63	0.18	0
EBNF	129	63.08	6.36	0.49	2
CFG Design	151	45.72	7.57	0.32	0
GAL	873	40.31	3.14	0.05	1
ANTLR V3	958	165.92	8.66	0.17	60
Ruby 1.8.5	13474	3509.2	23.18	0.26	7446
Java 1.6	6244	1107.34	17.56	0.18	658
ANSI C	2512	448.04	14.98	0.18	165
Python 2.5	3909	646.98	14.23	0.17	57
Java 1.5	7741	1342.88	17.87	0.17	5715
Java 1.4	7183	1279.29	17.99	0.18	5715
Java 1.3	6698	1189.24	17.76	0.18	5144
Java 1.2	6698	1189.24	17.76	0.18	5144
Java 1.1	6693	1193.28	17.65	0.18	5144
Java 1.0	5611	901.02	15.9	0.16	5144

Table 2). However, the evolution of this metrics is also directly connected to the complexity of the different versions of Java, but varies smoothly. Finally, the metrics value for the Java 1.6 grammar is not comparable to Java 1.5 grammar value, even if the language is the same. This is due to the fact that Java 1.6 grammar is designed for LL parsing and LL grammars are known to be more complex than LR ones. It is also probably due to a important use of conflict resolution mechanism in the Java 1.5 version (measured by the `lcc` metrics) that simplifies the grammar. From this result, it would appear that the metrics `lrs` is a good measure of the complexity of the grammar.

For each terminal in the grammar the `lat` metrics sums, the number of states in the LR automaton that does not lead to an error when this terminal is in the lookahead, and it is normalized by the number of terminals. Because `lat` metrics depends on the number of states, it is also interesting to normalize it by the number of states (`lrs`).

Indeed, the value for different versions of Java is very stable. It is also comparable to C, Java and Python grammar values. On the contrary, the value for the language GAL is very low. A low value for this metrics indicates that each terminal only appears in few of the grammar's states (5% of the states for GAL) and thus that the language is very controlled and probably easy to learn. On the other hand, a high value for this metrics, such as 49% for EBNF, indicates that the language may accept any terminal in approximately every state.

The results show that the metrics `lrtla` is closely related to the type of language. DSL grammars have smaller values than GPL grammars. Normalizing this metrics by the number of terminals in the grammar, it produces the same results as the normalized value of `lat`. This is to be expected since those two normalized metrics compute respectively the probability of being able to shift a given terminal in a given state and the probability of a given state to be able to shift a given terminal.

Results for the metrics `lcc` indicate that for most of the tested DSLs we do not have a LR conflict. Priorities/associativities are expressed in a recursive manner. If we compare ANTLR grammar for Java 1.6 and Tatoo grammar for Java 1.5, we can notice big difference in metrics `lcc` value. Value of Tatoo Java grammars for metrics `lcc` is high, because priorities/associativities are massively used in expressions.

In Table 5, the metrics `ss` provides results that are not related to the size of the grammar nor to the expressive power of the language. GPL and DSL grammars have similar values as well. This metrics evolves moderately with the different versions of Java. It seems to measure the verbosity of the grammar. Indeed, C or Python are known to be less verbose than Java. One surprising result is again, the smaller value of this metrics for the 1.6 version of Java. This is probably due to the larger number of productions for the LL version. Although these complementary productions have small sample sizes, the average value is smaller.

The `ltps`, `lptsm` and `ltpsa` metrics are directly related to the type of language. DSL have values below 1000, whereas GPL have values above 1000.

Table 5. Results for language based metrics

Lang	ss	ssm	ltps	ltpsm	ltpsa	ltpsn
Expr	1.56	4	35	6	4.29	0.43
FDL	2.53	6	47	8	4.34	0.23
EBNF	1.59	3	102	11	5.56	0.70
CFG Design	2.33	7	82	17	12.72	0.14
GAL	2.73	13	349	43	35.34	0.06
ANTLR V3	1.73	8	435	29	24.97	0.18
Ruby 1.8.5	1.47	7	3200	88	44.87	0.41
Java 1.6	2.04	10	2691	92	48.61	0.28
ANSI C	1.73	7	1777	81	41.92	0.25
Python 2.5	1.73	8	1576	61	48.33	0.21
Java 1.5	2.98	14	2370	83	50.06	0.22
Java 1.4	2.94	14	2272	81	49.78	0.22
Java 1.3	2.95	14	2239	80	49.25	0.22
Java 1.2	2.95	14	2239	80	49.25	0.22
Java 1.1	2.96	14	2200	79	48.81	0.22
Java 1.0	2.89	14	1734	78	48.77	0.18

They increase smoothly with the version of Java. The `ltpsn` value is very comparable to `lat/lrs`, since it measures the constraints on the language. The only language that gives different results is CDFG: the `ltpsn` stresses that the language is moderately constrained (14%) whereas the other metrics indicates 30%, which is quite high. Since, these two metrics are not exactly related, it is normal, that they produce different results, although large differences probably indicate an interesting property of the grammar. At this time we are not able to explain this behavior.

6. Usage Guidelines

There is currently no empirical study for grammar based metrics, which would ascertain or suggest when and how to use them. But from an in-depth understanding of metric design, we can give some useful usage guidelines. To do this we describe the advantages/disadvantages for each proposed metrics.

Metrics can be used for different purposes and in different stages of the grammar/language development life-cycle. The importance of metrics and their resulting interpretation is also dependent on their purpose. This is why it is important to identify the objective of the use. Three usages of proposed metrics are discussed in detail below.

Grammar-based Language Comparison

This is most common scenario in which we have different grammars and would like to compare them. First, we need to be aware that the same language can be described with different grammars, and a grammar can be described with different forms, or it can be specialized for different parsing techniques. The best way to compare grammars is to compare grammars that are in the same form and that they use the same parsing technique. In this case, the simplest metrics to use are `lrs`, `lat` and `ltpsm`. These metrics indicate the size and complexity of grammars and because of that, they can be used to rank the grammars. For a precise comparison with common languages we suggest using the results from Table 4 and Table 5. For quick reference of grammar size and complexity we propose four different groups.

– Tiny, mainly toy grammars,
– Small, mainly DSL grammars,
– Intermediate, 3rd generation GPL languages grammar,
– Big, modern object-oriented language grammars.

The groups are defined using an analysis of tested grammars. The intent of these groups is not to classify grammars by size or complexity, but merely for quick reference in order to have a first impression about the grammar size.

Table 6. Metrics orientation values

Grammar size	lrs		lat		ltpsm	
	from	to	from	to	from	to
Tiny	0	100	0	20	0	6
Small	101	1000	21	200	7	50
Intermediate	1001	3000	201	500	51	70
Big	3001	$\geq$ 3001	501	$\geq$ 501	71	$\geq$ 71

Orientation values for each group and metrics are stated in Table 6.

Metrics that are less size-dependent are `lrtla` and `lrpsa`. `lrtla` indicates average complexity of each state and `lrpsa` indicates complexity of terminal pairs. In the third group we find metrics and normalizations that are not size-dependent `lat/lrs`, `lcc`, `ss` and `ltpsn`. In most cases these values are averaged by size. Because of this, they are less adapted to comparing larger grammars.

Developing a New Language

In this scenario, we develop a new language from scratch. In practice, this means that in most cases we develop new DSL (because they are most common [16, 24]). In order to compare the new language with others, we can use metrics as suggested in the previous section. In addition, the developer can monitor metrics after every incremental developing stage to evaluate its influence on the complexity and the verbosity of grammars. For this purpose, all of the suggested metrics are appropriate, but if the developer wants to measure the verbosity of the language, he/she should look at the `lat/lrs`, `ss` or `lptsn` metrics, whereas, if the developer is interested in the complexity of the grammar he/she should look at other metrics. For developers it is also interesting to monitor number of conflicts `lcc`.

Sample Based Language Comparison

In this case, languages are compared based on samples/sentences. Two scenarios are possible: either formal grammar does not exist yet or a sample-based comparison is carried out in addition to a grammar-based comparison. To be able to compare or evaluate the language, we calculate the metrics `ltpsm`, `ltpsa` and `ltpsn` (`ltpX`). These metrics can be calculated directly from samples, with the condition that samples involved all allow combinations of terminals. In this scenario, there are two main problems. First, to identify all terminals and second to get some degree of confidence that our set of samples (S) is diverse enough. The first problem is lexically-related and solvable. The second problem can neither be tested nor computed. To overcome this obstacle, we use the relation between metrics with the unknown grammar G and metrics calculated from samples S. Value of `ltpX(G)` is always greater or equal to `ltpX(S)`. In practice this means that we can compare the language to a language that has smaller metrics values. If our goal is to infer grammar from samples, metrics `ltpsn` can help to evaluate a number of different non-terminals. Higher numbers mean fewer constraints in terms of language; this usually means less differing non-terminals. With this information we have more direct grammar inference search [6, 7].

7. Conclusion and Future Work

This paper explores the usefulness of several new metrics for grammar engineering. It presents experimental results for traditional metrics and for these new metrics on several grammars. These grammars cover domain-specific languages and general-purpose languages. Existing metrics are directly computed from the grammar itself. A first set of new metrics uses the LR automaton produced from the grammar. A second is related to the language recognized. We believe that these metrics provide interesting results that are not all covered by existing metrics. From this point of view, LR-based metrics are probably more suitable for grammar experts familiar with LR parsing, whereas other metrics could also be applicable for non-specialists in grammar development.

From experimental results, we see that some metrics are directly linked to the size or the complexity of the grammar whereas others remain stable even if the size or complexity of the grammar varies. Our findings show that the metrics in both cases qualify for evaluating the quality of the grammar. However, we consider that the quality of the grammar cannot be captured by a single metrics but by a range of the metrics explored in this paper. Moreover, this quality is not an absolute value but is relative to other grammars.

In this paper we only explore the metrics of the grammar portion of the analyzer, without looking at the lexing portion of the analyzer. However, the complexity of these two parts is closely related. For instance, one could specify in the lexer a different token for `true` and `false` or establish a generic token for the booleans. In this case, the grammar would necessarily be different and may produce different metrics. Therefore, we believe that metrics on token definitions could also be useful to capture the entire complexity of a language analyzer. An analyzer with complex lexer and simpler parser may be less maintainable than a complex parser with a simple lexer.

References

1. International standard EBNF syntax notation. ISO/IEC standard n° 14977 (1996)
2. Aho, A., Lam, M., Sethi, R., Ullman, J.: Compiler: Principles, Techniques, and Tools. Addison Wesley, 2nd edn. (2007)
3. Alves, T.L., Visser, J.: A case study in grammar engineering. In: Proceedings of the 1st International Conference on Software Language Engineering (SLE 2008). pp. 285–304. Lecture Notes in Computer Science Series, Springer Verlag (2008)
4. Cervelle, J., Forax, R., Roussel, G.: Tatoo: An innovative parser generator. In: 4th International Conference on Programming Principal and Practice in Java (PPPJ'06). pp. 13–20. ACM International Conference Proceedings, Mannheim, Germany (Aug 2006)
5. Cervelle, J., Forax, R., Roussel, G.: A simple implementation of grammar libraries. Computer Science and Information Systems 4(2), 65–77 (2007)
6. Črepinšek, M., Mernik, M., Javed, F., Bryant, B.R., Sprague, A.P.: Extracting grammar from programs: evolutionary approach. SIGPLAN Notices 40(4), 39–46 (2005)
7. Črepinšek, M., Mernik, M., Žumer, V.: Extracting grammar from programs: brute force approach. SIGPLAN Notices 40(4), 29–38 (2005)
8. Csuhaj-Varjú, E., Kelemenová, A.: Descriptional complexity of context-free grammar forms. Theoretical Computer Science 112(2), 277–289 (May 1993)
9. Denny, J.E., Malloy, B.A.: The ielr(1) algorithm for generating minimal lr(1) parser tables for non-lr(1) grammars with conflict resolution. Science of Computer Programming (September 2009), http://dx.doi.org/10.1016/j.scico.2009.08.001
10. Deursen, A. van., Klint, P.: Domain-specific language design requires feature descriptions. Journal of Computing and Information Technology 10(1), 1–17 (2002)
11. Elliott, C., Finne, S., Moor, O.D.: Compiling embedded languages. In: Proceedings of the Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG'00). pp. 9–27. Springer-Verlag (Sep 2000)
12. Gosling, J., Joy, B., Steele, G., Bracha, G.: Java Language Specification, Second Edition: The Java Series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2000)
13. Halstead, M.H.: Elements of Software Science. Elsevier, New York (1977)
14. Harbison, S.P., Steele Jr, G.L.: C A Reference Manual, Fourth Edition. Prentice-Hall, Upper Saddle River, NJ 07458, USA (1995)
15. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism sdf—reference manual—. SIGPLAN Not. 24(11), 43–75 (1989)

16. Javed, F., Mernik, M., Bryant, B., Sprague, A.: An unsupervised incremental learning algorithm for domain-specific language development. Applied Artificial Intelligence 22(7), 707–729 (2008)
17. Klint, P., Lämmel, R., Verhoef, C.: Toward an engineering discipline for grammarware. ACM Transactions on Software Engineering and Methodology 14(3), 331–380 (2005)
18. Lämmel, R.: Grammar Testing. In: Proc. of Fundamental Approaches to Software Engineering (FASE) 2001. LNCS, vol. 2029, pp. 201–216. Springer-Verlag (2001)
19. Lutz, M., Ascher, D.: Learning Python, Second Edition. O'Reilly Media, Inc., Sebastopol, CA (2003)
20. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys 37(4), 316–344 (2005)
21. Mernik, M., Korbar, N., Žumer, V.: LISA: A tool for automatic language implementation. ACM SIGPLAN Notices 30(4), 71–79 (Apr 1995)
22. Parr, T.J., Quong, R.W.: ANTLR: A Predicated-LL(k) Parser Generator. Software Practice and Experience 25(7), 789 – 810 (1995)
23. Power, J.F., Malloy, J.F.: A metrics suite for grammar-based software. Journal of Software Maintenance and Evolution: Research and Practice 16(6), 405–426 (2004)
24. Rebernak, D., Mernik, M., Wu, H., Gray, J.G.: Domain-specific aspect languages for modularising crosscutting concerns in grammars. IET software 3(3), 184–200 (2009)
25. Slivnik, B., Vilfan, B.: Producing the left parse during bottom-up parsing. Information Processing Letters (96), 220–224 (Dec 2005)
26. Thibault, S., Marlet, R., Consel, C.: Domain-specific languages: from design to implementation – application to video device drivers generation. IEEE Transactions on Software Engineering 25(3), 363–377 (May 1999)
27. Thomas, D., Fowler, C., Hunt, A.: Programming Ruby. The Pragmatic Programmer's Guide. Pragmatic Programmers (2004)
28. Watt, D.A.: Programming Language Concepts and Paradigms. Prentice-Hall (1990)

Appendix

7.1. Computation of closure application of rules

The first two steps of the computation of the shortest sample for a production require a same closure mechanism which is already implemented in the parser generator Tatoo to compute the first and the follow set [2].

The problem this mechanism solves can be formalized in the following way: we associate to each non-terminal X a mathematical object (a set of terminals for first and follow set, but a word in the process describe below) which we note $\mathcal{M}[X]$. The process fills the map $\mathcal{M}$ starting from some non-terminals and updates it using rules until the whole map is filled. In particular, an object associated to a non-terminal can change during the process. For instance, in the sequel, we try to compute the shortest words; in these cases, a word can be replaced by a shorter one.

The problem is expressed giving two rules. The first one, the *initiation rule*, tells how to initiate the process by giving an answer for some non-terminals and putting it in the map $\mathcal{M}$. The second one, the *iteration rule*, gives how to construct new objects from others, leading to the construction of dependency maps which store, for a non-terminal X:

i. the non-terminals Y such that $\mathcal{M}[Y]$ changes when $\mathcal{M}[X]$ is updated
ii. the non-terminals Y such that $\mathcal{M}[Y]$ has to be computed in order to get $\mathcal{M}[X]$.

Note that map *i.* is easily computed from iteration rules and map *ii.* is the reverse of map *i.* In order to compute the object associated to X, the solver first recursively uses map *ii.* to get all the words that have to be computed and then uses the iteration rule in a loop until no more changes are made into the map $\mathcal{M}$.

7.2. Computation of a shortest word generated by X

We note this map $\mathcal{M}_1$.

The rule for the computation are the following:

- **initiation rule** : if $X \to \alpha$ is a production such that α is the shortest only made of terminals, X generates α, $\mathcal{M}_1[X] = \alpha$.
- **iteration rule** : if $X \to \alpha$ is a production such that α does not contains X, then, if smaller or not yet defined, $\mathcal{M}_1[X]$ is replaced by the word obtained replacing each non-terminals Y of α by $\mathcal{M}_1[Y]$.

Note that cycles in dependency map are not a problem since they always lead to longer words.

7.3. Computation of a shortest word leading to X

We note this map $\mathcal{M}_2$.

The rule for the computation are the following:

- if S is an axiom, S is a shortest word leading to S, that is $\mathcal{M}_2[S] = S$.
- if $X \to \alpha Y \beta$ is a production, then, if smaller or not yet defined, $\mathcal{M}_2[Y]$ is replaced by the word $\mathcal{M}_2[X]$ where X is replaced by $\alpha' Y \beta'$, α' [resp. β'] being the word α [resp. β] where all non-terminals Z in this word are replaced by $\mathcal{M}_1[Z]$.

Here again, cycles are not a problem for the same reason.

Matej Črepinšek received the Ph.D. degree in computer science at the University of Maribor, Slovenia in 2007. His research interests include grammatical inference, evolutionary computations, object-oriented programming, compilers and grammar-based systems. He is currently a teaching assistant at the University of Maribor, Faculty of Electrical Engineering and Computer Science.

Tomaž Kosar received the Ph.D. degree in computer science at the University of Maribor, Slovenia in 2007. His research is mainly concerned with design and implementation of domain-specific languages. Other research interest in computer science include also domain-specific visual languages, empirical software engineering, software security, generative programming, compiler construction, object oriented programming, object-oriented design, refactoring, and unit testing. He is currently a teaching assistant at the University of Maribor, Faculty of Electrical Engineering and Computer Science.

Marjan Mernik received the M.Sc. and Ph.D. degrees in computer science from the University of Maribor in 1994 and 1998 respectively. He is currently a professor at the University of Maribor, Faculty of Electrical Engineering and Computer Science. He is also an adjunct professor at the University of Alabama at Birmingham, Department of Computer and Information Sciences. His research interests include programming languages, compilers, grammar-based systems, grammatical inference, and evolutionary computations. He is a member of the IEEE, ACM and EAPLS.

Julien Cervelle received the Ph. D. degree in computer science at Provence University, Marseille, France in 2002 and its habilitation thesis at Paris-Est University, France in 2007. His research interest are grammar based systems, parser generators, dynamical systems and cellular automata. He is currently Professor at Paris-Est University, France and adjunct professor at Ecole Polytechnique, Palaiseau, France.

Rémi Forax received the Ph. D. degree in Computer Science at Paris-Est University, France in 2001. He is currently a Teaching Assistant at Paris-Est University and a Java Community Process Expert for JSR 292. His main research areas concern design and implementation of programming languages, compiler construction, parser generators, virtual machines and executing environment.

Gilles Roussel received in computer science at UPMC, Paris, France in 1994 and its habilitation thesis at University of Marne-la-Vallée, France in 2003. He is currently professor at Paris-Est University and is deputy director of LIGM computer laboratory at Paris-Est University. His research interests are programming languages parsing, object-oriented design, program plagiarism detection, network programming and network routing algorithms.

Received: November 15, 2009; Accepted: March 30, 2010.

UDC 004.635.3, DOI: 10.2298/CSIS1002331F

Subtree Matching by Pushdown Automata[*]

Tomáš Flouri[1], Jan Janoušek[2], and Bořivoj Melichar[2]

[1] Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague
Karlovo nám. 13, 121 35 Prague 2, Czech Republic
flourtom@fel.cvut.cz,

[2] Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague
Kolejní 550/2, 160 00 Prague 6, Czech Republic
{Jan.Janousek,Borivoj.Melichar}@fit.cvut.cz

Abstract. Subtree matching is an important problem in Computer Science on which a number of tasks, such as mechanical theorem proving, term-rewriting, symbolic computation and nonprocedural programming languages are based on. A systematic approach to the construction of subtree pattern matchers by deterministic pushdown automata, which read subject trees in prefix and postfix notation, is presented. The method is analogous to the construction of string pattern matchers: for a given pattern, a nondeterministic pushdown automaton is created and is then determinised. In addition, it is shown that the size of the resulting deterministic pushdown automata directly corresponds to the size of the existing string pattern matchers based on finite automata.

Keywords: subtree, subtree matching, pushdown automata.

1. Introduction

The theory of formal string (or word) languages [2, 16, 24] and the theory of formal tree languages [6, 8, 14] are important parts of the theory of formal languages [23]. While the models of computation of the theory of string languages are finite automata, pushdown automata, linear bounded automata and Turing machines, the most famous models of computation of the theory of tree languages are various kinds of tree automata [6, 8, 14]. Trees, however, can also be seen as strings, for example in their prefix (also called preorder) or postfix (also called postorder) notation. Recently it has been shown that the deterministic pushdown automaton (PDA) is an appropriate model of computation for labelled, ordered, ranked trees in postfix notation and that the trees in postfix notation, acceptable by deterministic PDA, form a proper superclass of the class of regular tree languages, which are accepted by finite tree automata [18].

[*] This research has been partially supported by the Ministry of Education, Youth and Sports under research program MSM 6840770014, and by the Czech Science Foundation as project No. 201/09/0807.

Trees represent one of the fundamental data structures used in Computer Science and thus tree pattern matching, the process of finding occurrences of subtrees in trees, is an important problem with many applications, such as compiler code selection, interpretation of non-procedural languages or various tree finding and tree replacement systems.

Tree pattern matching is often declared to be analogous to the problem of string pattern matching [6]. One of the basic approaches used for string pattern matching can be represented by finite automata constructed for the pattern, which means that the pattern is preprocessed. Examples of these automata are the string matching automata [9, 10, 22, 26]. Given a pattern P of size m, the string matching automaton can be constructed for the pattern P in time linear to m. The constructed string matching automaton accepts the set of words containing pattern P as a suffix, and thus it can find all occurrences of string P in a given text T. The main advantage of this kind of finite automata is that the deterministic string matching automaton can be constructed in time linear to the size of the given pattern P, and the search phase is in time linear to the input text. A generalization of the mentioned string matching problem can be the string matching problem with multiple patterns [1, 22, 26]. Given a set of patterns $P = \{p_1, p_2, \dots, p_m\}$, the string matching automaton can be constructed in time linear to the number of symbols of patterns in set P. The constructed string matching automaton accepts the set of words having any of the patterns in P as a suffix, and thus it can find all occurrences of strings $p_1, \dots, p_m$ in a given text T.

Although there are many tree pattern matching methods (see [5–7, 11, 15, 25] for these methods), they fail to present a simple and systematic approach with a linear time searching phase which would also be directly analogous to the basic string pattern matching method.

This paper, being an extended version of [12], presents a new kind of PDAs for trees in prefix and postfix notations called subtree matching PDAs, which are directly analogous to string matching automata and their properties. A subtree matching PDA, constructed from a given tree s, can find all occurrences of subtree s within a given tree t in time $\mathcal{O}(n)$, where n is the number of nodes of t. Subtree matching, as with string matching, can also be generalized to subtree matching with multiple patterns. Subtree matching PDAs can be constructed from a set of trees $P = \{t_1, t_2, \dots, t_m\}$ in the same manner as string matching automata, retaining their property of linear searching phase $\mathcal{O}(n)$, where n is the number of nodes of the subject tree t.

Moreover, the presented subtree matching PDAs have the following two other properties. First, they are input–driven PDAs [28], which means that each pushdown operation is determined only by the input symbol. The input–driven PDAs can be always determinised [28]. Second, their pushdown symbol alphabets contain just one pushdown symbol and therefore their pushdown store can be implemented by a single integer counter. This means that the presented PDAs can be transformed to counter automata [4, 27], which is a weaker and simpler model of computation than the PDA.

The rest of the paper is organised as follows. Basic definitions are given in section 2. Some properties of subtrees in prefix notation are discussed in the third section. Sections 4 and 5 deal with the subtree matching PDA constructed over a single and multiple patterns, respectively. Section 6 shows the dual principle for the postfix notation and the last section is the conclusion.

2. Basic Notions

2.1. Ranked alphabet, tree, prefix notation, postfix notation, subtree matching

We define notions on trees similarly as they are defined in [2, 6, 8, 14].

We denote the set of natural numbers by $\mathbb{N}$. A *ranked alphabet* is a finite, nonempty set of symbols, each of which has a unique nonnegative *arity* (or *rank*). Given a ranked alphabet $\mathcal{A}$, the arity of a symbol $a \in \mathcal{A}$ is denoted by $Arity(a)$. The set of symbols of arity p is denoted by $\mathcal{A}_p$. Elements of arity $0, 1, 2, \dots, p$ are respectively called nullary (constants), unary, binary, ... , p-ary symbols. We assume that $\mathcal{A}$ contains at least one constant. In the examples we use numbers at the end of identifiers for a short declaration of symbols with arity. For instance, $a2$ is a short declaration of a binary symbol a.

Based on concepts from graph theory (see [2]), a labelled, ordered, ranked tree over a ranked alphabet $\mathcal{A}$ can be defined as follows:

An *ordered directed graph* G is a pair (N, R), where N is a set of nodes and R is a set of linearly ordered lists of edges such that each element of R is of the form $((f, g_1), (f, g_2), \dots, (f, g_n))$, where $f, g_1, g_2, \dots, g_n \in N$, $n \geq 0$. This element would indicate that, for node f, there are n edges leaving f, the first entering node g_1, the second entering node g_2, and so forth.

A sequence of nodes $(f_0, f_1, \dots, f_n)$, $n \geq 1$, is a *path* of length n from node f_0 to node f_n if there is an edge which leaves node f_{i-1} and enters node f_i for $1 \leq i \leq n$. A *cycle* is a path $(f_0, f_1, \dots, f_n)$, where $f_0 = f_n$. An ordered *dag* (dag stands for Directed Acyclic Graph) is an ordered directed graph that has no cycle. *Labelling* of an ordered graph $G = (A, R)$ is a mapping of A into a set of labels. In the examples we use a_f for a short declaration of node f, labelled by symbol a.

Given a node f, its *out-degree* is the number of distinct pairs $(f, g) \in R$, where $g \in A$. By analogy, *in-degree* of node f is the number of distinct pairs $(g, f) \in R$, where $g \in A$.

A *labelled, ordered, ranked and rooted tree* t over a ranked alphabet $\mathcal{A}$ is an ordered dag $t = (N, R)$ with a special node $r \in A$ called the *root* such that
(1) r has in-degree 0,
(2) all other nodes of t have in-degree 1,
(3) there is just one path from the root r to every $f \in N$, where $f \neq r$,
(4) every node $f \in N$ is labelled by a symbol $a \in \mathcal{A}$ and out-degree of a_f is *Arity*(a).

Nodes labelled by nullary symbols (constants) are called *leaves*.

Prefix notation pref(t) of a labelled, ordered, ranked and rooted tree t is obtained by applying the following *Step* recursively, beginning at the root of t:
Step: Let this application of *Step* be node a_f. If a_f is a leaf, list a and halt. If a_f is not a leaf, having direct descendants $a_{f_1}, a_{f_2}, \ldots, a_{f_n}$, then list a and subsequently apply *Step* to $a_{f_1}, a_{f_2}, \ldots, a_{f_n}$ in that order.
Postfix notation post(t) of t is formed by changing the last sentence of *Step* to read “Apply *Step* to $a_{f_1}, a_{f_2}, \ldots, a_{f_n}$ in that order and then list a.”

Example 1. Consider a tree $t_1 = (\ \{a2_1, a2_2, a0_3, a1_4, a0_5, a1_6, a0_7\}, R\ )$ over $\mathcal{A} = \{a2, a1, a0\}$, where R is a set of the following ordered sequences of pairs:

$$((a2_1, a2_2), (a2_1, a1_6)),$$
$$((a2_2, a0_3), (a2_2, a1_4)),$$
$$((a1_4, a0_5)),$$
$$((a1_6, a0_7))$$

The prefix and postfix notations of tree t_1 are strings *pref*$(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ and $post(t_1) = a0\ a0\ a1\ a2\ a0\ a1\ a2$, respectively. Trees can be represented graphically, and tree t_1 is illustrated in Fig. 1. □

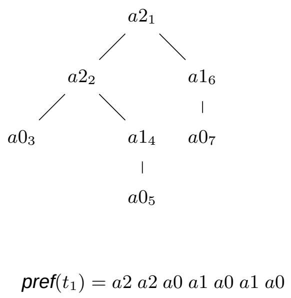


Fig. 1. Tree t_1 from Example 1 and its prefix notation

The number of nodes of a tree t is denoted by $|t|$.

The height of a tree t, denoted by *Height(t)*, is defined as the maximal length of a path from the root of t to a leaf of t.

A subtree p *matches* an object tree t at node n if p is equal to the subtree of t rooted at n.

2.2. Alphabet, language, pushdown automaton

We define notions from the theory of string languages similarly as they are defined in [2, 16].

Let an *alphabet* be a finite nonempty set of symbols. A *string* x over a given alphabet is a finite, possibly empty sequence of symbols. A *language* over an alphabet $\mathcal{A}$ is a set of strings over $\mathcal{A}$. Set $\mathcal{A}^*$ denotes the set of all strings over $\mathcal{A}$ including the empty string, denoted by ε. Set $\mathcal{A}^+$ is defined as $\mathcal{A}^+ = \mathcal{A}^* \setminus \{\varepsilon\}$. Similarly for string $x \in \mathcal{A}^*$, x^m, $m \geq 0$, denotes the m-fold concatenation of x with $x^0 = \varepsilon$. Set x^* is defined as $x^* = \{x^m : m \geq 0\}$ and $x^+ = x^* \setminus \{\varepsilon\} = \{x^m : m \geq 1\}$.

An (extended) *nondeterministic pushdown automaton* (nondeterministic PDA) is a seven-tuple $M = (Q, \mathcal{A}, G, \delta, q_0, Z_0, F)$, where Q is a finite set of *states*, $\mathcal{A}$ is the *input alphabet*, G is the *pushdown store alphabet*, δ is a mapping from $Q \times (\mathcal{A} \cup \{\varepsilon\}) \times G^*$ into a set of finite subsets of $Q \times G^*$, $q_0 \in Q$ is the initial state, $Z_0 \in G$ is the initial content of the pushdown store, and $F \subseteq Q$ is the set of final (accepting) states. The triplet $(q, w, x) \in Q \times \mathcal{A}^* \times G^*$ denotes the configuration of a pushdown automaton. In this paper we will write the top of the pushdown store x on its left hand side. The initial configuration of a pushdown automaton is a triplet (q_0, w, Z_0) for the input string $w \in \mathcal{A}^*$.

The relation $\vdash_M \subset (Q \times \mathcal{A}^* \times G^*) \times (Q \times \mathcal{A}^* \times G^*)$ is a *transition* of a pushdown automaton M. It holds that $(q, aw, \alpha\beta) \vdash_M (p, w, \gamma\beta)$ if $(p, \gamma) \in \delta(q, a, \alpha)$. The k-th power, transitive closure, and transitive and reflexive closure of the relation $\vdash_M$ is denoted $\vdash_M^k$, $\vdash_M^+$, $\vdash_M^*$, respectively. A pushdown automaton M is a *deterministic* pushdown automaton (deterministic PDA), if it holds:

1. $|\delta(q, a, \gamma)| \leq 1$ for all $q \in Q$, $a \in \mathcal{A} \cup \{\varepsilon\}$, $\gamma \in G^*$.
2. If $\delta(q, a, \alpha) \neq \emptyset$, $\delta(q, a, \beta) \neq \emptyset$ and $\alpha \neq \beta$ then α is not a suffix of β and β is not a suffix of α.
3. If $\delta(q, a, \alpha) \neq \emptyset$, $\delta(q, \varepsilon, \beta) \neq \emptyset$, then α is not a suffix of β and β is not a suffix of α.

A pushdown automaton is *input–driven* if its each pushdown operation is determined only by the input symbol.
A language L accepted by a pushdown automaton M is defined in two distinct ways:

1. *Accepting by final state:*

$$L(M) = \{x : \delta(q_0, x, Z_0) \vdash_M^* (q, \varepsilon, \gamma) \wedge x \in \mathcal{A}^* \wedge \gamma \in G^* \wedge q \in F\}.$$

2. *Accepting by empty pushdown store:*

$$L_\varepsilon(M) = \{x : (q_0, x, Z_0) \vdash_M^* (q, \varepsilon, \varepsilon) \wedge x \in \mathcal{A}^* \wedge q \in Q\}.$$

If a PDA accepts the language by empty pushdown store then the set F of final states may be the empty set. The subtree PDAs accept the languages by empty pushdown store.

In the rest of the text, we use the following notation for labelling edges when illustrating transition diagrams of various PDAs: For each transition rule $\delta_1(p, a, \alpha) = (q, \beta)$ from the transition mapping δ of a PDA, we label its edge leading from state p to state q by the triplet of the form $a|\alpha \mapsto \beta$.

For more details on pushdown automata see [2, 16].

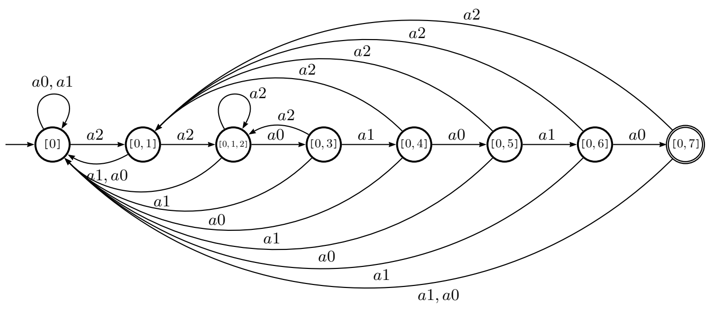


Fig. 2. Transition diagram of deterministic string matching automaton for pattern $x = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 2

2.3. Examples of string matching automaton

Example 2. The transition diagram of the deterministic string matching automaton constructed for string $a2\ a2\ a0\ a1\ a0\ a1\ a0$ is illustrated in Fig. 2. □

Example 3. The transition diagram of the deterministic string matching automaton constructed for a set of strings $P = \{a2\ a2\ a0\ a0\ b0, a2\ b1\ a0\ a0, a2\ a0\ a0\}$ is illustrated in Fig. 3. □

See [2, 9, 22] for definitions of finite automata and construction of the deterministic string matching automaton.

3. Properties of subtrees in prefix notation

In this section we describe some general properties of the prefix notation of a tree and of its subtrees. These properties are important for the construction of the subtree matching PDA, which is described in the next two sections.

Example 4. Consider tree t_1 in prefix notation *pref*$(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 1, which is illustrated in Fig. 1. Tree t_1 contains only subtrees shown in Fig. 4.

Generally, for any tree, the following theorem holds.

Theorem 1. *Given a tree t and its prefix notation pref(t), all subtrees of t in prefix notation are substrings of pref(t).*

Proof. By induction on the height of the subtree.

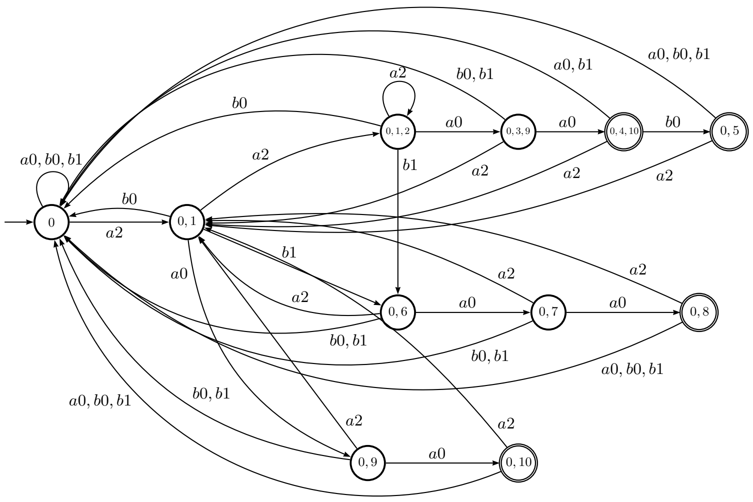
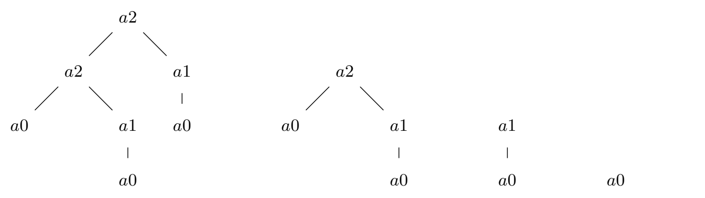


Fig. 3. Transition diagram of deterministic string matching automaton (Aho-Corasick) for patterns $\{a2\ a2\ a0\ a0\ b0,\ a2\ b1\ a0\ a0,\ a2\ a0\ a0\}$

1. If a subtree t' has just one node a, where *Arity*$(a) = 0$, then *Height*$(t') = 0$, *pref*$(t') = a$ and the claim holds for that subtree.
2. Assume that the claim holds for subtrees $t_1, t_2, \ldots, t_p$, where $p \geq 1$ and *Height*$(t_1) \leq m$, *Height*$(t_2) \leq m$, ..., *Height*$(t_p) \leq m$, $m \geq 0$. We have to prove that the claim holds also for each subtree $t' = at_1t_2 \ldots t_p$, where *Arity*$(a) = p$ and *Height*$(t') = m + 1$:
   As *pref*$(t') = a$ *pref*(t_1) *pref*$(t_2) \ldots$*pref*(t_p), the claim holds for the subtree t'.

Thus, the theorem holds. $\square$

However, not every substring of a tree in prefix notation is its subtree in prefix notation. This can be easily seen on the fact that for a given tree with n nodes in prefix notation, there can be $\mathcal{O}(n^2)$ distinct substrings but there is just n subtrees – each node of the tree is the root of just one subtree. Just those substrings which themselves are trees in prefix notation are those which are the subtrees in prefix notation. This property is formalised by the following definition and theorem.



pref$(t) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ *pref*$(t) = a2\ a0\ a1\ a0$ *pref*$(t) = a1\ a0$ *pref*$(t) = a0$
post$(t) = a0\ a0\ a1\ a2\ a0\ a1\ a2$ *post*$(t) = a0\ a0\ a1\ a2$ *post*$(t) = a0\ a1$ *post*$(t) = a0$

Fig. 4. All subtrees of tree t_1 from Example 1, and their prefix and postfix notations

Definition 1. *Let $w = a_1a_2 \dots a_m$, $m \geq 1$, be a string over a ranked alphabet $\mathcal{A}$. Then, the* arity checksum **ac**$(w) =$ *Arity*$(a_1)+$*Arity*$(a_2)+\ldots+$*Arity*$(a_m)-m+1$= $\sum_{i=1}^{m}$ *Arity*$(a_i) - m + 1$.

Theorem 2. *Let pref(t) and w be a tree t in prefix notation and a substring of pref(t), respectively. Then, w is the prefix notation of a subtree of t, if and only if $ac(w) = 0$, and $ac(w_1) \geq 1$ for each w_1, where $w = w_1x$, $x \neq \varepsilon$.*

Proof. It is easy to see that for any two subtrees st_1 and st_2 it holds that *pref*(st_1) and *pref*(st_2) are either two different strings or one is a substring of the other. The former case occurs if the subtrees st_1 and st_2 are two different trees with no shared part and the latter case occurs if one tree is a subtree of the other tree. No partial overlapping of subtrees is possible in ranked ordered trees. Moreover, for any two neighbouring subtrees it holds that their prefix notations are two adjacent substrings.

– *If:* By induction on the height of a subtree st, where $w =$ *pref*(st):
  1. We assume that *Heigth*$(st) = 1$, which means we consider the case $w = a$, where *Arity*$(a) = 0$. Then, **ac**$(w) = 0$. Thus, the claim holds for the case *Height*$(st) = 1$.
  2. Assume that the claim holds for the subtrees $st_1, st_2, \dots, st_p$ where $p \geq 1$, *Height*$(st_1) \leq m$, *Height*$(st_2) \leq m$, …, *Height*$(st_p) \leq m$ and **ac**(*pref*$(st_1)) = 0$, **ac**(*pref*$(st_2)) = 0$, …, **ac**(*pref*$(st_p)) = 0$.
     We are to prove that it holds also for a subtree of height $m + 1$. Assume $w = a$ *pref*(st_1) *pref*(st_2) …*pref*(st_p), where *Arity*$(a) = p$. Then
     ac$(w) = p+$**ac**(*pref*(st_1))+**ac**(*pref*(st_2))+…+**ac**(*pref*(st_p))$-(p+1)+1 = 0$ and **ac**$(w_1) \geq 1$ for each w_1 , where $w = w_1x$, $x \neq \varepsilon$.
     Thus, the claim holds for the case *Height*$(st) = m + 1$.

– *Only if*: Assume **ac**$(w) = 0$, and $w = a_1a_2 \dots a_k$, where $k \geq 1$, *Arity*$(a_1) = p$. Since **ac**$(w_1) \geq 1$ for each w_1, where $w = w_1x$, $x \neq \varepsilon$, none of the substrings w_1 can be a subtree in prefix notation. This means that the only possibility

for $ac(w) = 0$ is that w is of the form $w = a$ *pref*(t_1) *pref*$(t_2) \dots$ *pref*(t_p), where $p \geq 0$, and $t_1, t_2 \dots t_p$ are neighbouring subtrees. In such case, *ac*$(w) = p + 0 - (p + 1) + 1 = 0$.
No other possibility of the form of w for *ac*$(w) = 0$ is possible. Thus, the claim holds.

Thus, the theorem holds. □

We note that in subtree matching PDAs, the arity checksum is computed by pushdown operations, where the contents of the pushdown store represents the corresponding arity checksum. For example, the empty pushdown store means that the corresponding arity checksum is equal to 0.

4. Subtree Matching pushdown automaton

This section deals with the subtree matching PDA for trees in prefix notation: algorithms and theorems are given and the subtree matching PDA and its construction are demonstrated with an example.

Problem 1 (Subtree Matching). Given two trees s and t, find all occurrences of tree s in tree t.

Definition 2. *Let s and pref(s) be a tree and its prefix notation, respectively. Given an input tree t, a subtree pushdown automaton constructed over pref(s) accepts all matches of tree s in the input tree t by final state.*

First, we start with a PDA which accepts the whole subject tree in prefix notation. The construction of the PDA accepting a tree in prefix notation is described by Alg. 1. The constructed PDA is deterministic.

Algorithm 1. Construction of a PDA accepting a tree t in prefix notation *pref*(t).
Input: A tree t over a ranked alphabet $\mathcal{A}$; prefix notation *pref*$(t) = a_1 a_2 \dots a_n$, $n \geq 1$.
Output: PDA $M_p(t) = (\{0, 1, 2, \dots, n\}, \mathcal{A}, \{S\}, \delta, 0, S, \{n\})$.
Method:

1. For each state i, where $1 \leq i \leq n$, create a new transition $\delta(i - 1, a_i, S) = (i, S^{\mathit{Arity}(a_i)})$, where $S^0 = \varepsilon$. □

Example 5. The PDA constructed by Alg. 1, accepting the prefix notation *pref*$(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ of tree t_1 from Example 1, is the deterministic PDA $M_p(t_1) = (\{0, 1, 2, 3, 4, 5, 6, 7\}, \mathcal{A}, \{S\}, \delta_1, 0, S, \{n\}))$, where the mapping δ_1 is a set of the following transitions:

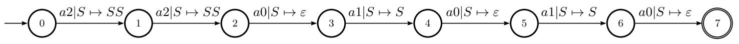

0
$a2|S \mapsto SS$
1
$a2|S \mapsto SS$
2
$a0|S \mapsto \varepsilon$
3
$a1|S \mapsto S$
4
$a0|S \mapsto \varepsilon$
5
$a1|S \mapsto S$
6
$a0|S \mapsto \varepsilon$
7

Fig. 5. Transition diagram of deterministic PDA $M_p(t_1)$ accepting tree t_1 in prefix notation *pref*$(t_1) = a2\ a0\ a2\ a0\ a0\ a0$ from Example 5

$$\begin{aligned}
\delta_1(0, a2, S) &= (1, SS)\\
\delta_1(1, a2, S) &= (2, SS)\\
\delta_1(2, a0, S) &= (3, \varepsilon)\\
\delta_1(3, a1, S) &= (4, S)\\
\delta_1(4, a0, S) &= (5, \varepsilon)\\
\delta_1(5, a1, S) &= (6, S)\\
\delta_1(6, a0, S) &= (7, \varepsilon)
\end{aligned}$$

The transition diagram of deterministic PDA $M_p(t_1)$ is illustrated in Fig. 5. Fig. 6 shows the sequence of transitions (trace) performed by deterministic PDA $M_p(t_1)$ for tree t_1 in prefix notation. □

State	Input	Pushdown Store
0	$a2\ a2\ a0\ a1\ a0\ a1\ a0$	S
1	$a2\ a0\ a1\ a0\ a1\ a0$	$S\ S$
2	$a0\ a1\ a0\ a1\ a0$	$S\ S\ S$
3	$a1\ a0\ a1\ a0$	$S\ S$
4	$a0\ a1\ a0$	$S\ S$
5	$a1\ a0$	S
6	$a0$	S
7	ε	ε
accept		

Fig. 6. Trace of deterministic PDA $M_p(t_1)$ from Example 5 for tree t_1 in prefix notation *pref*$(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$

Theorem 3. *Let $M = (\{Q, \mathcal{A}, \{S\}, \delta, 0, S, F)$ be an input–driven PDA whose each transition from δ is of the form $\delta(q_1, a, S) = (q_2, S^i)$, where $i =$ Arity(a). Then, if $(q_3, w, S) \vdash_M^+ (q_4, \varepsilon, S^j)$, then $j =$ ac(w).*

Proof. By induction on the length of w:

1. Assume $w = a$. Then, $(q_3, a, S) \vdash_M (q_4, \varepsilon, S^j)$, where $j =$ *Arity*$(a) = ac(a)$. Thus, the claim holds for the case $w = a$.
2. Assume that the claim holds for a string $w = a_1a_2 \dots a_k$, where $k \geq 1$. This means that $(q_3, a_1a_2 \dots a_k, S) \vdash_M^k (q_4, \varepsilon, S^j)$, where $j =$ ac$(a_1a_2 \dots a_k)$. We have to prove that the claim holds also for $w = a_1a_2 \dots a_k\ a$.

It holds that $(q_3, a_1a_2 \dots a_ka, S) \vdash_M^k (q_4, a, S^j) \vdash_M (q_5, \varepsilon, S^l)$, where $l = j + \textit{Arity}(a) - 1 = ac(w) + \textit{Arity}(a) - 1 = \textit{Arity}(a_1) + \textit{Arity}(a_2) + \dots + \textit{Arity}(a_k) - k + 1 + \textit{Arity}(a) - 1 = \textit{ac}(a_1a_2 \dots a_ka)$.
Thus, the claim holds for the case $w = a_1a_2 \dots a_k\ a$.

Thus, the theorem holds. □

The correctness of the deterministic PDA constructed by Alg. 1, which accepts trees in prefix notation, is described by the following lemma.

Lemma 1. *Given a tree t and its prefix notation pref(t), the PDA $M_p(t) = (\{0, 1, 2, \dots, n\}, \mathcal{A}, \{S\}, \delta, 0, S, F)$, where $n = |t|$, constructed by Alg. 1, accepts pref(t).*

Proof. By induction on the height of the tree t:

1. If tree t has just one node a, where *Arity*$(a) = 0$, then *Height*$(t) = 0$, *pref*$(t) = a$, $\delta(0, a, S) = (1, \varepsilon) \in \delta$, $(0, a, S) \vdash_{M_p(t)} (1, \varepsilon, \varepsilon)$ and the claim holds for that tree.
2. Assume that claim holds for trees $t_1, t_2, \dots, t_p$, where $p \geq 1$, *Height*$(t_1) \leq m$, *Height*$(t_2) \leq m$, ..., *Height*$(t_p) \leq m$, $m \geq 0$.
   We have to prove that the claim holds also for each tree t such that
   pref$(t) = a$ *pref*(t_1)*pref*$(t_2) \dots$ *pref*(t_p), *Arity*$(a) = p$, and *Height*$(t) \geq m + 1$:
   Since $\delta(0, a, S) = (1, S^p) \in \delta$, and $(0, a\ \textit{pref}(t_1)\textit{pref}(t_2) \dots \textit{pref}(t_p), S)$
   $\vdash_{M_p(t)} (1, \textit{pref}(t_1)\textit{pref}(t_2) \dots \textit{pref}(t_p), S^p)$
   $\vdash_{M_p(t)}^* (i, \textit{pref}(t_2) \dots \textit{pref}(t_p), S^{p-1})$
   $\vdash_{M_p(t)}^* \cdots$
   $\vdash_{M_p(t)}^* (j, \textit{pref}(t_p), S)$
   $\vdash_{M_p(t)}^* (k, \varepsilon, \varepsilon)$,
   the claim holds for that tree.

Thus, the lemma holds. □

We present the construction of the deterministic subtree matching PDA for trees in prefix notation. The construction consists of two steps. First, a nondeterministic subtree matching PDA is constructed by Alg. 2. This nondeterministic subtree matching PDA is an extension of the PDA accepting trees in prefix notation, which is constructed by Alg. 1. Second, the constructed nondeterministic subtree matching PDA is transformed to the equivalent deterministic subtree matching PDA. In spite of the fact that the determinisation of a nondeterministic PDA is not possible generally, the constructed nondeterministic subtree matching PDA is an input–driven PDA and therefore can be determinised [28].

Algorithm 2. Construction of a nondeterministic subtree matching PDA for a tree t in prefix notation *pref*(t).
Input: A tree t over a ranked alphabet $\mathcal{A}$; prefix notation *pref*$(t) = a_1a_2 \dots a_n$, $n \geq 1$.

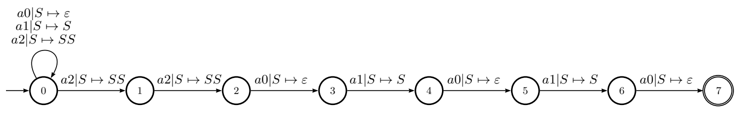


Fig. 7. Transition diagram of nondeterministic subtree matching PDA $M_p(t_1)$ for tree t_1 in prefix notation *pref*$(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 6

Output: Nondeterministic subtree matching PDA $M_{nps}(t) = (\{0, 1, 2, \ldots, n\}, \mathcal{A}, \{S\}, \delta, 0, S, \{n\})$.
Method:

1. Create PDA $M_{nps}(t)$ as PDA $M_p(t)$ by Alg. 1.
2. For each symbol $a \in \mathcal{A}$ create a new transition $\delta(0, a, S) = (0, S^{\textit{Arity}(a)})$, where $S^0 = \varepsilon$.

Example 6. The subtree matching PDA, constructed by Alg. 2 from tree t_1 having prefix notation *pref*$(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$, is the nondeterministic PDA $M_{nps}(t_1) = (\{0, 1, 2, 3, 4, 5, 6, 7\}, \mathcal{A}, \{S\}, \delta_2, 0, S, \{7\}))$, where mapping δ_2 is a set of the following transitions:

$$\begin{array}{ll}
\delta_2(0, a2, S) = (1, SS) & \\
\delta_2(1, a2, S) = (2, SS) & \delta_2(0, a2, S) = (0, SS) \\
\delta_2(2, a0, S) = (3, \varepsilon) & \delta_2(0, a1, S) = (0, S) \\
\delta_2(3, a1, S) = (4, S) & \delta_2(0, a0, S) = (0, \varepsilon) \\
\delta_2(4, a0, S) = (5, \varepsilon) & \\
\delta_2(5, a1, S) = (6, S) & \\
\delta_2(6, a0, S) = (7, \varepsilon) &
\end{array}$$

The transition diagram of the nondeterministic PDA $M_{nps}(t_1)$ is illustrated in Fig. 7. □

Theorem 4. *Given a tree t and its prefix notation pref(t), the PDA $M_{nps}(t)$ constructed by Alg. 2 is a subtree matching PDA for pref(t).*

Proof. According to Theorem 2, given an input tree t, each subtree in prefix notation is a substring of *pref*(t). Since the PDA $M_{nps}(s)$ has just states and transitions equivalent to the states and transitions, respectively, of the string matching automaton, the PDA $M_{nps}(t)$ accepts all matches of subtree s in tree t by final state. □

For the construction of deterministic subtree PDA, we use the transformation described by Alg. 3. This transformation is based on the well known transformation of nondeterministic finite automaton to an equivalent deterministic one, which constructs the states of the deterministic automaton as subsets of states of the nondeterministic automaton and selects only a set of accessible states (i.e. subsets) [16]. Again, states of the resulting deterministic PDA correspond to subsets of states of the original nondeterministic PDA.

Algorithm 3. Transformation of an input–driven nondeterministic PDA to an equivalent deterministic PDA.
Input: Input–driven nondeterministic PDA $M_{nx}(t) = (\{0, 1, 2, \ldots, n\}, \mathcal{A}, \{S\}, \delta, 0, S, F)$
Output: Equivalent deterministic PDA $M_{dx}(t) = (Q', \mathcal{A}, \{S\}, \delta', q_I, S, F')$.
Method:

```
1. Initially, $Q' = \{\{0\}\}$, $q_I = \{0\}$ and $\{0\}$ is an unmarked state.
2. (a) Select an unmarked state $q'$ from $Q'$.
   (b) For each input symbol $a \in \mathcal{A}$:
       i. $q'' = \{q : \delta(p, a, \alpha) = (q, \beta)$ for all $p \in q'\}$.
       ii. Add transition $\delta'(q', a, S) = (q'', S^{\text{Arity}}(a))$.
       iii. If $q'' \notin Q$ then add $q''$ to $Q$ and set it as unmarked state.
   (c) Set state $q'$ as marked.
3. Repeat step 2 until all states in $Q'$ are marked.
4. $F' = \{ q' \mid q' \in Q' \wedge q' \cap F \neq \emptyset \}$.   □
```

The deterministic subtree matching automaton $M_{dps}(t)$ for a tree t with prefix notation *pref*(t) is demonstrated by the following example.

Example 7. The deterministic subtree matching PDA for tree t_1 in prefix notation *pref*$(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 1 , which has been constructed by Alg. 3 from nondeterministic subtree matching PDA $M_{nps}(t_1)$ from Example 6, is the deterministic PDA $M_{dps}(t_1) = (\{[0], [0, 1], [0, 1, 2], [0, 3], [0, 4], [0, 5], [0, 6], [0, 7]\}, \mathcal{A}, \{S\}, \delta_3, [0], S, \{[0, 7]\})$, where its transition diagram is illustrated in Fig. 9.

We note that the deterministic subtree matching PDA $M_{dps}(t_1)$ has a very similar transition diagram to the deterministic string matching automaton constructed for *pref*(t_1) [9, 22], as can be seen by comparing Figs. 2 and 9. The only difference between the two types of automata are the pushdown operations appearing in the subtree matching PDA, which ensure the validity of the input tree. The input tree is valid only if the pushdown store of the subtree PDA is emptied after the last symbol from the prefix notation of the input tree is read.

Fig. 8 shows the sequence of transitions (trace) performed by the deterministic subtree PDA $M_{dps}(t_1)$ for an input tree t_2 in prefix notation *pref*$(t_2) = a2\ a2\ a2\ a0\ a1\ a0\ a1\ a0\ a1\ a1\ a2\ a0\ a0$. The accepting state is $\{0, 7\}$. Fig. 10 depicts the pattern subtree t_1 and input tree t_2. □

Theorem 5. *Given a nondeterministic input–driven PDA* $M_{nx}(t) = (Q, \mathcal{A}, \{S\}, \delta, q_0, S, F)$*, the deterministic PDA* $M_{dx}(t) = (Q', \mathcal{A}, \{S\}, \delta', \{q_0\}, S, F')$ *which is constructed by Alg. 3 is equivalent to PDA* $M_{nx}(t)$.

State	Input	PDS
$\{0\}$	$a2\ a2\ a2\ a0\ a1\ a0\ a1\ a0\ a1\ a1\ a2\ a0\ a0$	S
$\{0,1\}$	$a2\ a2\ a0\ a1\ a0\ a1\ a0\ a1\ a1\ a2\ a0\ a0$	SS
$\{0,1,2\}$	$a2\ a0\ a1\ a0\ a1\ a0\ a1\ a1\ a2\ a0\ a0$	SSS
$\{0,1,2\}$	$a0\ a1\ a0\ a1\ a0\ a1\ a1\ a2\ a0\ a0$	$SSSS$
$\{0,3\}$	$a1\ a0\ a1\ a0\ a1\ a1\ a2\ a0\ a0$	SSS
$\{0,4\}$	$a0\ a1\ a0\ a1\ a1\ a2\ a0\ a0$	SSS
$\{0,5\}$	$a1\ a0\ a1\ a1\ a2\ a0\ a0$	SS
$\{0,6\}$	$a0\ a1\ a1\ a2\ a0\ a0$	SS
$\{0,7\}$	$a1\ a1\ a2\ a0\ a0$ match	S
$\{0\}$	$a1\ a2\ a0\ a0$	S
$\{0\}$	$a2\ a0\ a0$	S
$\{0,1\}$	$a0\ a0$	SS
$\{0\}$	$a0$	S
$\{0\}$	ε	ε

Fig. 8. Trace of deterministic subtree PDA $M_{dps}(t_1)$ from Example 7 for an input subtree t_2 in prefix notation *pref*$(t_2) = a2\ a2\ a2\ a0\ a1\ a0\ a1\ a0\ a1\ a1\ a2\ a0\ a0$

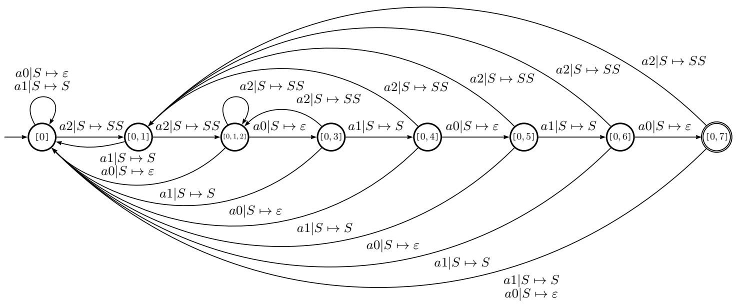
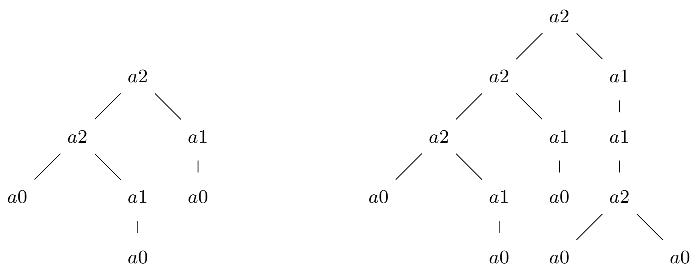


Fig. 9. Transition diagram of deterministic PDA $M_{dps}(t_1)$ for tree t_1 in prefix notation *pref*$(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 7

Proof. First, we prove the following claim by induction on i:
(*): $(q_1', w, S) \vdash^i_{M_{dx}(t)} (q_2', \varepsilon, S^j)$ if and only if
$q_2' = \{p : (q, w, S) \vdash^i_{M_{nx}(t)} (p, \varepsilon, S^j) \text{ for some } q \in q_1'\}$.

1. Assume i=1.
   – *if*: if $(q_1', a, S) \vdash_{M_{dx}(t)} (q_2', \varepsilon, S^j)$, then there exists a state $q \in q_1'$, where $(q, a, S) \vdash_{M_{nx}(t)} (p, \varepsilon, S^j)$, $p \in q_2'$.
   – *only if*: if $(q, a, S) \vdash_{M_{nx}(t)} (p, \varepsilon, \beta)$, then for each $q_1' \in Q'$, where $q \in q_1'$, it holds that $(q_1', a, S) \vdash_{M_{dx}(t)} (q_2', \varepsilon, S^j)$, where $p \in q_2'$.
2. Assume that claim (*) holds for $i = 1, 2, \ldots, k$, $k \geq 1$.
   This means that $(q_1', w, S) \vdash^k_{M_{dx}(t)} (q_2', \varepsilon, S^j)$ if and only if
   $q_2' = \{p : (q, S, w) \vdash^k_{M_{nx}(t)} (p, \varepsilon, S^j) \text{ for some } q \in q_1'\}$. We have to prove that claim (*) holds also for $i = k + 1$.
   – *if*: if $(q_1', w, S) \vdash^k_{M_{dx}(t)} (q_2', a, S^l) \vdash_{M_{dx}(t)} (q_3', \varepsilon, S^j)$, then there exists a state $q \in q_2'$, where $(q, a, S^l) \vdash_{M_{nx}(t)} (p, \varepsilon, S^j)$, $p \in q_3'$.
   – *only if*: if $(q_0, pref(t), S) \vdash^k_{M_{nx}(t)} (q, a, S^l) \vdash_{M_{nx}(t)} (p, \varepsilon, S^j)$, then for each $q_1' \in Q'$, where $q \in q_1'$, it holds that $(q_1', a, S^l) \vdash_{M_{dx}(t)} (q_2', \varepsilon, S^j)$, where $p \in q_2'$.

As a special case of claim (*), $(\{q_0\}, pref(t), S) \vdash^i_{M_{dx}(t)} (q', \varepsilon, \varepsilon)$ if and only if $(q_0, S, pref(t)) \vdash^i_{M_{nx}(t)} (q_1, \varepsilon, \varepsilon)$. Thus, the theorem holds.



pref$(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ *pref*$(t_2) = a2\ a2\ a2\ a0\ a1\ a0\ a1\ a0\ a1\ a1\ a2\ a0\ a0$
post$(t_1) = a0\ a0\ a1\ a2\ a0\ a1\ a2$ *post*$(t_2) = a0\ a0\ a1\ a2\ a0\ a1\ a2\ a0\ a0\ a2\ a1\ a1\ a2$

Fig. 10. Trees t_1 and t_2 from Example 7 along with their prefix and postfix notations

Theorem 6. *Given a tree t with n nodes in its prefix or postfix notation, the deterministic subtree matching PDA $M_{pds}(t)$ constructed by Alg. 2 and 3 is made of exactly $n + 1$ states, one pushdown symbol and $|\mathcal{A}|(n + 1)$ transitions.*

Proof. Let $M_{nps}(t) = (\{0, 1, 2, \dots, n\}, \mathcal{A}, \{S\}, \delta, 0, S, \{n\}$ be an automaton constructed from tree t with a prefix notation *pref*$(t) = a_1\ a_2\ \dots\ a_n$ over ranked alphabet $\mathcal{A}$ by Alg. 2. We will prove that this automaton is directly analogous to the string matching automaton and accepts the same language if we ignore the pushdown operations, which actually do not affect the process of determinisation as M_{pds} is an input–driven automaton. From Alg 2 and 3, $M_{nps}(t)$ has transitions $\delta(0, a, S) = (0, S^{\mathit{Arity}(a)})$ for all $a \in \mathcal{A}$ and $\delta(i-1, a_i, S) = (i, \varepsilon, S^{\mathit{Arity}(a_i)})$. The proof is a mutual induction of the following $n+1$ statements:
(1) $\delta^*(0, w, S) = (0, \varepsilon, S^{\mathit{ac}(w)})$, $w \in \mathcal{A}^*$.
(2) $\delta^*(0, w, S) = (1, \varepsilon, S^{\mathit{ac}(w)})$ if and only if $w = w_1 a_1$, $w_1 \in \mathcal{A}^*$
(i) $\delta^*(0, w, S) = (i-1, \varepsilon, S^{\mathit{ac}(w)})$ if and only if $w = w_1 a_1 a_2 \dots a_{i-1}$, $w_1 \in \mathcal{A}^*$

1. Assume that $|w| = 0$, which means $w = \varepsilon$. Statement (1) holds, since $\delta^*(0, \varepsilon, S) = (0, \varepsilon, S)$. Statements ($i$), $1 < i \leq n+1$, do not hold as $\delta^*(0, \varepsilon, S)$ contains, from its basic definition, only $(0, \varepsilon, S)$.
2. Assume $w = w_1 a$, where $w_1 \in \mathcal{A}^k$, that is $|w_1| = k$ and $a \in \mathcal{A}$. We may assume that statements (i) $1 < i \leq n+1$ hold for w_1, and we need to prove them for w. We assume the inductive hypothesis for k and prove it for $k+1$.
   (a) There exists a series of transitions $(0, w_1, S) \vdash^* (0, \varepsilon, S^{\mathit{ac}(w_1)})$, since $\delta(0, a, S) = (0, \varepsilon, S^{\mathit{Arity}(a)})$ are transitions of automaton M_{nps}.Thus statement (1) is proved for w.
   (b) We now prove statements i, where $1 < i \leq n+1$:
      – *If:* Assume that $w_1 = w_2 a_1 a_2 \dots a_{i-2}$, where $w_2 \in \mathcal{A}^*$ and $a = a_{i-1}$. By statement ($i-1$) applied to w_1, we know from our induction hypothesis that there exists a series of transitions $(0, w_1, S) \vdash^* (i-2, \varepsilon, S^{\mathit{ac}(w_1)})$. Since for all $1 \leq j \leq n$ there exists a transition $\delta(j-1, a_j, S) = (j, S^{\mathit{Arity}(a_j)})$, we conclude that $\delta^*(0, w, S) = (i-1, \varepsilon, S^{\mathit{ac}(w)})$.
      – *Only if:* Suppose there exists a series of transitions $(0, w, S) \vdash^* (i-1, \varepsilon, S^{\mathit{ac}(w)})$. From the inductive assumption we know that there exists a series of transitions $(0, w_1, S) \vdash^* (i-2, \varepsilon, S^{\mathit{ac}(w_1)})$. By statement $(i-1)$ applied to w_1, we know that $w_1 = w_2 a_1 a_2 \dots a_{i-2}$. Thus $w = w_2 a_1 a_2 \dots a_{i-1}$, and we have proved statement (i).

Thus, from statements $1, \dots, n+1$, if we ignore the pushdown operations, M_{pds} accepts the language $L = \{w.\mathit{pref}(t)\}$, where $w \in \mathcal{A}^*$. Since the subtree matching PDA is directly analogous to the string matching automaton, we can use the proof from [10, 22] for space and time complexities. □

Theorem 7. *Given an input tree t with n nodes, the searching phase of the deterministic subtree matching automaton constructed by Algs. 2 and 3 is $\mathcal{O}(n)$.*

Proof. The searching phase consists of reading tree t once, symbol by symbol from left to right. The appropriate transition is taken each time a symbol is read, resulting in exactly n transitions. Each transition consumes a constant time because the time of each pushdown operation is limited by the maximal arity of nodes. Occurrences of the subtree to find are matched by transitions leading to the final states. □

Finally, we note that trees having structure *pref*$(t) = (a1)^{n-1}a0$ represent strings. The deterministic subtree matching PDA for such trees has the same number of states and transitions as the deterministic string matching automaton constructed for *pref*(t) and accepts the same language.

5. Multiple subtree matching

In this section we present a generalization of Problem 1. We deal with the construction of subtree matching PDA over a finite set of trees. The whole concept is demonstrated with an example.

Problem 2 (Multiple Subtree Matching). Given a tree t and a set of m trees $P = \{t_1, t_2, \dots, t_m\}$, find all occurrences of trees $t_1, t_2, \dots, t_m$ in tree t.

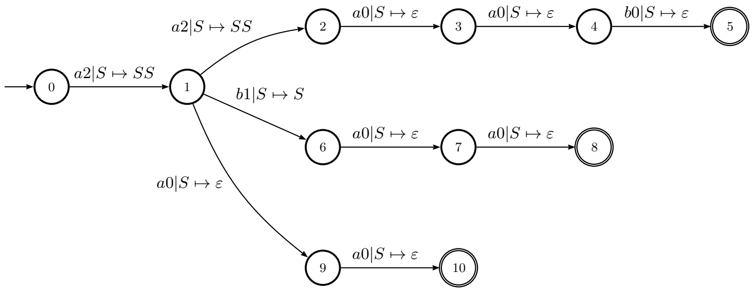
Definition 3. *Let $P = \{t_1, t_2, \dots, t_m\}$ be a set of m trees and pref$(t_i), 1 \le i \le m$ be the prefix notation of the i-th tree in P. Given an input tree t, a subtree pushdown automaton constructed over set P accepts all matches of subtrees $t_1, t_2, \dots, t_m$ in the input tree t by final state.*

Similarly as in Section 4, our method begins with a PDA which accepts trees $t_1, t_2, \dots, t_m$ in their prefix notation. The construction of this PDA is described by Alg. 4

Algorithm 4. Construction of a PDA accepting a set of trees $P = \{t_1, t_2, \dots, t_m\}$ in their prefix notation.
Input: A set of trees $P = \{t_1, t_2, \dots, t_m\}$ over a ranked alphabet $\mathcal{A}$; prefix notation *pref*$(t_i) = a_1 a_2 \dots a_{n_i}$, $1 \le i \le m$, $n_i \ge 1$.
Output: PDA $M_p(P) = (\{0, 1, 2, \dots, q\}, \mathcal{A}, \{S\}, \delta, 0, S, F)$.
Method:

```
1. Let $q \leftarrow 0$ and $F \leftarrow \emptyset$
2. For each tree $t_i = a^i_1\ a^i_2\ \dots\ a^i_{|t_i|}$, $1 \le i \le m$, do
   (a) Let $l \leftarrow 0$
   (b) For $j = 1$ to $|t_i|$ do
       i. If the transition $\delta(l, a^i_j, S)$ is not defined then
          A. Let $q \leftarrow q + 1$
          B. Create a transition $\delta(l, a^i_j, S) \leftarrow (q, S^{Arity(a^i_j)})$
          C. Let $l \leftarrow q$
       ii. Else if transition $\delta(l, a^i_j, S)$ is defined
          A. $l \leftarrow p$ where $(p, \gamma) \leftarrow \delta(l, a_j, S)$
   (c) $F \leftarrow F \cup \{l\}$
```

Example 8. Consider a set of trees $P = \{t_1, t_2, t_3\}$, with their prefix notations being *pref*$(t_1) = a2\ a2\ a0\ a0\ b0$, *pref*$(t_2) = a2\ b1\ a0\ a0$ and *pref*$(t_3) = a2\ a0\ a0$. The deterministic PDA constructed by Alg. 4 accepting the prefix notation of trees in P is $M_p(P) = (\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}, \mathcal{A}, \{S\}, \delta_1, 0, S, \{5, 8, 10\}))$, where mapping δ_1 is a set of the following transitions:



$a2|S \mapsto SS$
$a2|S \mapsto SS$
$a0|S \mapsto \varepsilon$
$a0|S \mapsto \varepsilon$
$b0|S \mapsto \varepsilon$
$b1|S \mapsto S$
$a0|S \mapsto \varepsilon$
$a0|S \mapsto \varepsilon$
$a0|S \mapsto \varepsilon$
$a0|S \mapsto \varepsilon$
0
1
2
3
4
5
6
7
8
9
10

Fig. 11. Transition diagram of deterministic PDA $M_p(P)$ accepting the trees with prefix notation $\{a2\ a2\ a0\ a0\ b0, a2\ b1\ a0\ a0, a2\ a0\ a0\}$ from Example 8

$$\begin{aligned}
&\delta_1(0, a2, S) = (1, SS)\\
&\delta_1(1, a2, S) = (2, SS)\\
&\delta_1(2, a0, S) = (3, \varepsilon)\\
&\delta_1(3, a0, S) = (4, \varepsilon)\\
&\delta_1(4, b0, S) = (5, \varepsilon)\\
&\delta_1(1, b1, S) = (6, S)\\
&\delta_1(6, a0, S) = (7, \varepsilon)\\
&\delta_1(7, a0, S) = (8, \varepsilon)\\
&\delta_1(1, a0, S) = (9, \varepsilon)\\
&\delta_1(9, a0, S) = (10, \varepsilon)
\end{aligned}$$

The transition diagram of deterministic PDA $M_p(P)$ is illustrated in Fig. 11.

Fig. 12 shows the sequence of transitions (trace) performed by deterministic PDA $M_p(P)$ for trees $t_1, t_2, t_3 \in P$ in prefix notation. $\square$

The correctness of the deterministic PDA constructed by Alg. 4, which accepts trees in prefix notation, is described by the following lemma.

Lemma 2. *Given a set of k trees $P = \{t_1, t_2, \ldots, t_m\}$ and their prefix notation pref(t_i), $1 \le i \le m$, the PDA $M_p(P) = (\{0, 1, 2, \ldots, n\}, \mathcal{A}, \{S\}, \delta, 0, S, F)$, where $1 + min(|t_1|, |t_2|, \ldots, |t_m|) \le n \le 1 + \sum_{j=1}^{k} |t_j|$, constructed by Alg. 4 accepts pref(t_i), where $1 \le t_i \le m$.*

Proof. By induction on the height of trees $t_1, t_2, \ldots, t_m$:

1. If trees $t_1, t_2, \ldots, t_m$ have just one node, $a_1, a_2, \ldots, a_k$ respectively, where *Arity*$(a_i) = 0$, for all $1 \le i \le k$, then *Height*$(t_i) = 0$, *pref*$(t_i) = a_i$, $\delta(0, a_i, S) = (i, \varepsilon) \in \delta$, $(0, a_i, S) \vdash_{M_p(P)} (i, \varepsilon, \varepsilon)$ for all $1 \le i \le k$ and the claim holds.

State	Input	Pushdown Store
0	$a2\ a2\ a0\ a0\ b0$	S
1	$a2\ a0\ a0\ b0$	$S\ S$
2	$a0\ a0\ b0$	$S\ S\ S$
3	$a0\ b0$	$S\ S$
4	$b0$	S
5	ε	ε
accept		
0	$a2\ b1\ a0\ a0$	S
1	$b1\ a0\ a0$	$S\ S$
6	$a0\ a0$	$S\ S$
7	$a0$	S
8	ε	ε
accept		
0	$a2\ a0\ a0$	S
1	$a0\ a0$	$S\ S$
9	$a0$	S
10	ε	ε
accept		

Fig. 12. Trace of deterministic PDA $M_p(P)$ from Example 8 for trees in prefix notation $\{a2\ a2\ a0\ a0\ b0,\ a2\ b1\ a0\ a0,\ a2\ a0\ a0\}$

2. Assume that the claim holds for trees $t^1_1, t^1_2, \dots, t^1_{p_1}, t^2_1, t^2_2, \dots, t^2_{p_2}, \dots, t^k_1, t^k_2, \dots, t^k_{p_k}$ where $p_i \geq 1$ for all $1 \leq i \leq k$, *Height*$(t^i_1) \leq m$, *Height*$(t^i_2) \leq m$, …, *Height*$(t^i_p) \leq m$, $m \geq 0$, for all $1 \leq i \leq k$.
   We have to prove that the claim holds also for each tree t_i, $1 \leq i \leq k$, such that
   pref$(t_i) = a_i$ *pref*(t^i_1)*pref*$(t^i_2) \dots$*pref*$(t^i_{p_i})$, *Arity*$(a_i) = p_i$, and *Height*$(t_i) \geq m+1$:
   Since $\delta(0, a_i, S) = (i, S^p) \in \delta$, and $(0, a$ *pref*(t^i_1)*pref*$(t^i_2) \dots$*pref*$(t^i_{p_i}), S)$
   $\vdash_{M_p(t_i)} (i,$*pref*(t^i_1)*pref*$(t^i_2) \dots$*pref*$(t^i_{p_i}), S^p)$
   $\vdash^*_{M_p(t_i)} (j^i,$*pref*$(t^i_2) \dots$*pref*$(t^i_{p_i}), S^{p_i-1})$
   $\vdash^*_{M_p(t_i)} \cdots$
   $\vdash^*_{M_p(t_i)} (\ell^i,$*pref*$(t^i_{p_i}), S)$
   $\vdash^*_{M_p(t_i)} (f^i, \varepsilon, \varepsilon)$
   the claim holds for that tree.

Thus, the lemma holds. □

The deterministic subtree matching PDA for multiple tree patterns in prefix notation can be constructed in a similar fashion to the subtree matching PDA for a single pattern. First, the PDA accepting a set of trees in their prefix notations, constructed by Alg. 4, is used to construct a nondeterministic subtree matching PDA by Alg. 5. The constructed nondeterministic subtree matching PDA is then transformed to the equivalent deterministic subtree matching PDA.

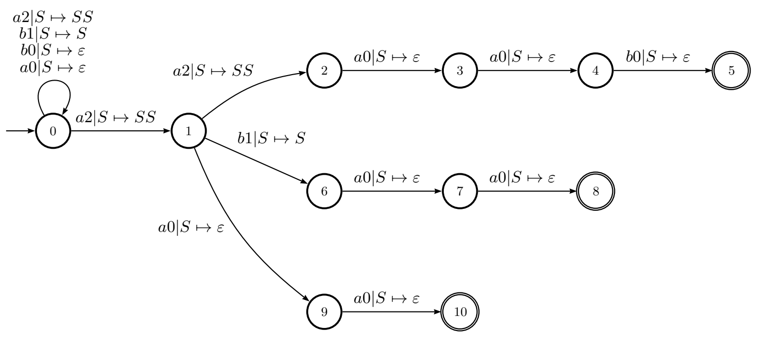

$a2|S \mapsto SS$
$b1|S \mapsto S$
$b0|S \mapsto \varepsilon$
$a0|S \mapsto \varepsilon$
$a2|S \mapsto SS$
$a2|S \mapsto SS$
$a0|S \mapsto \varepsilon$
$a0|S \mapsto \varepsilon$
$b0|S \mapsto \varepsilon$
$b1|S \mapsto S$
$a0|S \mapsto \varepsilon$
$a0|S \mapsto \varepsilon$
$a0|S \mapsto \varepsilon$
$a0|S \mapsto \varepsilon$
0
1
2
3
4
5
6
7
8
9
10

Fig. 13. Transition diagram of nondeterministic subtree matching PDA $M_p(P)$ constructed over trees in set P from Example 9

Algorithm 5. Construction of a nondeterministic subtree matching PDA for a set of trees $P = \{t_1, t_2, \ldots, t_m\}$ in their prefix notation.
Input: A tree t over a ranked alphabet $\mathcal{A}$; prefix notation *pref*$(t) = a_1 a_2 \ldots a_n$, $n \geq 1$.
Output: Nondeterministic subtree matching PDA $M_{nps}(t) = (Q, \mathcal{A}, \{S\}, \delta, 0, S, F)$.
Method:

1. Create PDA $M_{nps}(t)$ as PDA $M_p(t) = (Q, \mathcal{A}, \{S\}, \delta, 0, S, F)$ by Alg. 4.
2. For each symbol $a \in \mathcal{A}$ create a new transition $\delta(0, a, S) = (0, S^{\mathit{Arity}(a)})$, where $S^0 = \varepsilon$.

□

Example 9. The subtree matching PDA constructed by Alg. 2 over the set of trees P from Example 8 is the nondeterministic PDA $M_{nps}(P) = (\{0,1,2,3,4,5,6,7,8,9,10\}, \mathcal{A}, \{S\}, \delta_2, 0, S, \{5, 8, 10\}))$, where mapping δ_2 is a set of the following transitions:

$$\begin{array}{ll}
\delta_2(0, a2, S) = (1, SS) & \\
\delta_2(1, a2, S) = (2, SS) & \delta_2(0, a2, S) = (0, SS) \\
\delta_2(2, a0, S) = (3, \varepsilon) & \delta_2(0, b1, S) = (0, S) \\
\delta_2(3, a0, S) = (4, \varepsilon) & \delta_2(0, b0, S) = (0, \varepsilon) \\
\delta_2(4, b0, S) = (5, \varepsilon) & \delta_2(0, a0, S) = (0, \varepsilon) \\
\delta_2(1, b1, S) = (6, S) & \\
\delta_2(6, a0, S) = (7, \varepsilon) & \\
\delta_2(7, a0, S) = (8, \varepsilon) & \\
\delta_2(1, a0, S) = (9, \varepsilon) & \\
\delta_2(9, a0, S) = (10, \varepsilon) &
\end{array}$$

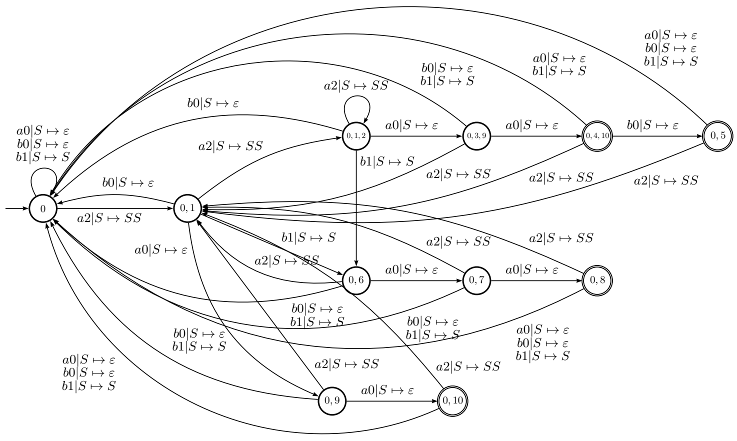


Fig. 14. Transition diagram of deterministic PDA $M_{dps}(P)$ constructed over trees in set P from Example 10

The transition diagram of nondeterministic PDA $M_{nps}(P)$ is illustrated in Fig. 13. $\square$

Theorem 8. *Given a set of m trees $P = \{t_1, t_2, \ldots, t_m\}$ and their prefix notation pref(t_i), $1 \leq i \leq m$, the PDA $M_{nps}(P)$ constructed by Alg. 5 is a subtree matching PDA for tree patterns $t_1, t_2, \ldots, t_m$.*

Proof. According to Theorem 2, given an input tree t, each subtree in prefix notation is a substring of *pref*(t). Since the PDA $M_{nps}(P)$ has just states and transitions equivalent to the states and transitions, respectively, of the Aho-Corasick string matching automaton , the PDA $M_{nps}(P)$ accepts all matches of subtrees $t_1, t_2, \ldots, t_m$ in tree t by final state. $\square$

For the construction of deterministic subtree PDA, we use the transformation described by Alg. 3 from Section 4.

The deterministic subtree matching automaton $M_{dps}(P)$ for a set of trees $P = \{t_1, t_2, \ldots, t_m\}$ with prefix notations *pref*$(t_i), 1 \leq i \leq k$ is demonstrated by the following example.

Example 10. The deterministic subtree matching PDA for the set of trees P from Example 8, constructed by Alg. 3 from the nondeterministic subtree matching PDA $M_{nps}(P)$ from Example 9, is $M_{dps}(P) = (\{[0], [0,1], [0,1,2], [0,3,9], [0,4,10], [0,5], [0,6], [0,7], [0,8], [0,9], [0,10]\}, \mathcal{A}, \{S\}, \delta_3, [0], S, \{[0,4,10], [0,5], [0,8], [0,10]\})$, with its transition diagram illustrated in Fig. 14.

We note that the deterministic subtree matching PDA $M_{dps}(P)$ has a very similar transition diagram to the Aho-Corasick string matching automaton constructed for the strings representing the prefix notations of trees in set P from Example 8 (see also [1, 9, 22]), as can be seen by comparing Figs. 4 and 14.

Fig. 15 shows the sequence of transitions (trace) performed by the deterministic subtree PDA $M_{dps}(P)$ for the input tree t having prefix notation *pref*(t) $= a2\ a2\ a2\ a0\ a0\ a2\ a2\ a0\ a0\ b0\ \ a2\ b1\ a0\ a0$. The final states are $\{[0,4,10],[0,5],[0,8],\ [0,10]\}$. Fig. 16 depicts the pattern subtrees from set P and the input tree t. □

State	Input		PDS
$\{0\}$	$a2\ a2\ a2\ a0\ a0\ a2\ a2\ a0\ a0\ b0\ a2\ b1\ a0\ a0$		S
$\{0,1\}$	$a2\ a2\ a0\ a0\ a2\ a2\ a0\ a0\ b0\ a2\ b1\ a0\ a0$		SS
$\{0,1,2\}$	$a2\ a0\ a0\ a2\ a2\ a0\ a0\ b0\ a2\ b1\ a0\ a0$		SSS
$\{0,1,2\}$	$a0\ a0\ a2\ a2\ a0\ a0\ b0\ a2\ b1\ a0\ a0$		$SSSS$
$\{0,3,9\}$	$a0\ a2\ a2\ a0\ a0\ b0\ a2\ b1\ a0\ a0$		SSS
$\{0,4,10\}$	$a2\ a2\ a0\ a0\ b0\ a2\ b1\ a0\ a0$	match	SS
$\{0,1\}$	$a2\ a0\ a0\ b0\ a2\ b1\ a0\ a0$		SSS
$\{0,1,2\}$	$a0\ a0\ b0\ a2\ b1\ a0\ a0$		$SSSS$
$\{0,3,9\}$	$a0\ b0\ a2\ b1\ a0\ a0$		SSS
$\{0,4,10\}$	$b0\ a2\ b1\ a0\ a0$	match	SS
$\{0,5\}$	$a2\ b1\ a0\ a0$	match	S
$\{0,1\}$	$b1\ a0\ a0$		SS
$\{0,6\}$	$a0\ a0$		SS
$\{0,7\}$	$a0$		S
$\{0,8\}$	ε	match	ε

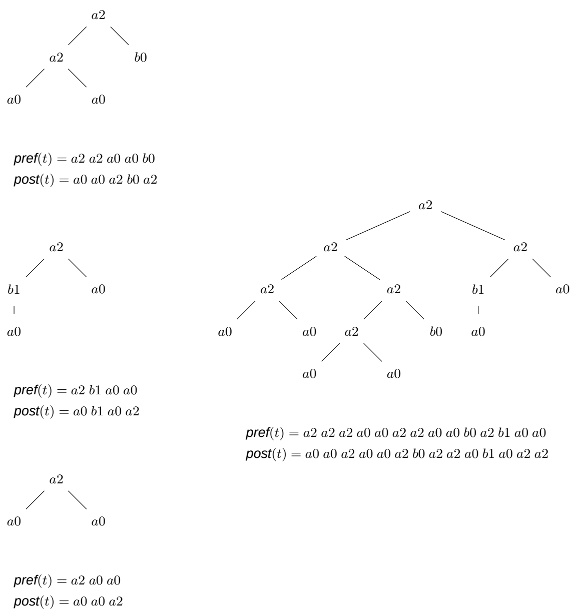
Fig. 15. Trace of deterministic subtree PDA $M_{dps}(P)$ from Example 10 for tree t_2 in prefix notation *pref*$(t) = a2\ a2\ a2\ a0\ a0\ a2\ a2\ a0\ a0\ b0\ a2\ b1\ a0\ a0$.

Theorem 9. *Given a set of m trees $P = \{t_1, t_2, \ldots, t_m\}$ over a ranked alphabet $\mathcal{A}$, the deterministic subtree matching PDA $M_{pds}(P)$ is constructed by Alg. 5 and 3 in time $\Theta(|\mathcal{A}|s)$, requires $\Theta(|\mathcal{A}|s)$ storage, where $s = \sum_{i=1}^{m} |t_i|$, and its pushdown store alphabet consists of one symbol.*

Proof. Since the subtree matching PDA for multiple patterns is directly analogous to the Aho-Corasick string matching automaton (this can be proved from proof of Theorem 6), we can use the proof from [1] and [26]. □

Theorem 10. *Given an input tree t with n nodes, the searching phase of the deterministic subtree matching automaton constructed by Algs. 2 and 3 over a set of m trees P is $\mathcal{O}(n)$.*

Proof. The searching phase consists of reading tree t once, symbol by symbol from left to right. The appropriate transition is taken each time a symbol is read,



pref$(t) = a2\ a2\ a0\ a0\ b0$
post$(t) = a0\ a0\ a2\ b0\ a2$

pref$(t) = a2\ b1\ a0\ a0$
post$(t) = a0\ b1\ a0\ a2$

pref$(t) = a2\ a0\ a0$
post$(t) = a0\ a0\ a2$

pref$(t) = a2\ a2\ a2\ a0\ a0\ a2\ a2\ a0\ a0\ b0\ a2\ b1\ a0\ a0$
post$(t) = a0\ a0\ a2\ a0\ a0\ a2\ b0\ a2\ a2\ a0\ b1\ a0\ a2\ a2$

Fig. 16. Pattern subtrees from set P and the input tree from Example 10 along with their prefix and postfix notations

resulting in exactly n transitions. Each transition consumes a constant time because the time of each pushdown operation is limited by the maximal arity of nodes. Occurrences of the subtree to find are matched by transitions leading to the final states. $\square$

6. Subtree matching in postfix notation

In this section we show the dual principle for the postfix notation. Theorems 11 and 12 present the direct analogy of properties of the prefix and postfix notations. Theorem 13 is analogous to Theorem 3.

Theorem 11. *Given a tree t and its postfix notation post(t), all subtrees of t in postfix notation are substrings of post(t).*

Theorem 12. *Let post(t) and w be a tree t in postfix notation and a substring of post(t), respectively. Then, w is the postfix notation of a subtree of t, if and only if $ac(w) = 0$, and $ac(w_1) \leq -1$ for each w_1, where $w = xw_1$, $x \neq \varepsilon$.*

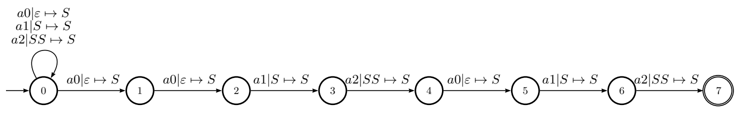


Fig. 17. Transition diagram of nondeterministic subtree matching PDA $M_p(t_1)$ for tree t_1 in postfix notation *post*$(t_1) = a0\ a0\ a1\ a2\ a0\ a1\ a2$ from Example 6

Theorem 13. *Let* $M = (\{Q, \mathcal{A}, \{S\}, \delta, 0, S, F)$ *be an input–driven PDA whose each transition from* δ *is of the form* $\delta(q_1, a, S^i) = (q_2, S)$*, where* $i =$ *Arity*(a)*. Then, if* $(q_3, w, \varepsilon) \vdash_M^+ (q_4, \varepsilon, S^j)$*, then* $j = -ac(w) + 1$*.*

From the above Theorems, we can easily transform Algorithms 1-5 to work with the postfix notation of trees. The only change required is in the pushdown operations. All transitions of the form $\delta(q, a, S) = (p, S^{\mathit{Arity}(a_i)})$ must be changed to the form $\delta(q, a, S^{\mathit{Arity}(a_i)}) = (p, S)$. The subtree matching PDA also requires no initial pushdown store symbol, while after processing a valid tree in postfix notation, the pushdown store contains a single symbol 'S'.

Fig. 17 illustrates the nondeterministic subtree matching PDA $M_p(t_1)$ constructed from the postfix notation of the tree from Example 6.

Fig. 18 illustrates the deterministic subtree matching PDA $M_dps(t_1)$ constructed from the postfix notation of the tree from Example 6.

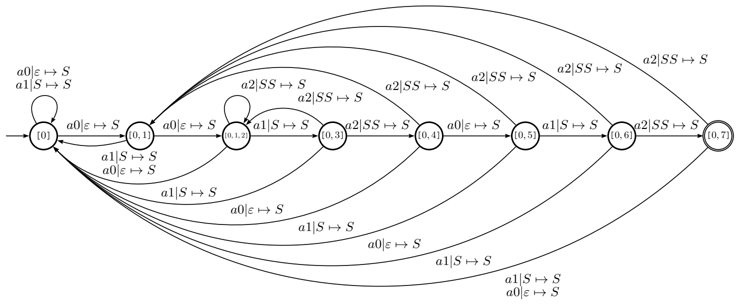


Fig. 18. Transition diagram of deterministic PDA $M_{dps}(t_1)$ for tree t_1 in postfix notation *post*$(t_1) = a0\ a0\ a1\ a2\ a0\ a1\ a2$ from Example 7

7. Conclusion

We have introduced a new kind of pushdown automata: subtree matching PDAs for trees in prefix and postfix notations. These pushdown automata are in their properties analogous to string matching automata, which are widely used in stringology [9, 10, 22, 26].

Regarding specific tree algorithms whose model of computation is the standard deterministic pushdown automaton, we have recently introduced principles of other three new algorithms. First, the tree pattern matching PDA [13, 21] which is an extension of the subtree matching PDA presented in this paper. Second, the subtree and tree pattern PDAs, which represent a complete index of a given tree by preprocessing it. Searching for all occurrences of a subtree or a tree pattern of size m is then performed in time linear to m and not depending on the size of the preprocessed tree [17, 19, 21]. These automata representing indexes of trees are analogous in their properties to the string suffix and factor automata [9, 10, 22, 26]. Third, a method on how to find all repeats of connected subgraphs in trees with the use of subtree or tree pattern PDAs [21, 20]. More details on these results and related information can also be found on [3].

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun. ACM 18(6), 333–340 (1975)
2. Aho, A.V., Ullman, J.D.: The theory of parsing, translation, and compiling. Prentice-Hall Englewood Cliffs, N.J. (1972)
3. Arbology www pages. Available on: http://www.arbology.org/ (2009), december 2009
4. Berstel, J.: Transductions and Context-Free Languages. Teubner Studienbucher, Stuttgart (1979)
5. Chase, D.R.: An improvement to bottom-up tree pattern matching. In: POPL. pp. 168–177 (1987)
6. Cleophas, L.: Tree Algorithms. Two Taxonomies and a Toolkit. Ph.D. thesis, Technische Universiteit Eindhoven, Eindhoven (2008)
7. Cole, R., Hariharan, R., Indyk, P.: Tree pattern matching and subset matching in deterministic (log^3)-time. In: SODA. pp. 245–254 (1999)
8. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata (2007), release October, 12th 2007
9. Crochemore, M., Hancart, C.: Automata for matching patterns. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, vol. 2 Linear Modeling: Background and Application, chap. 9, pp. 399–462. Springer–Verlag, Berlin (1997)
10. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific, New Jersey (1994)
11. Dubiner, M., Galil, Z., Magen, E.: Faster tree pattern matching. J. ACM 41(2), 205–213 (1994)
12. Flouri, T., Janoušek, J., Melichar, B.: Subtree matching by deterministic pushdown automata. In: Ganzha, M., Paprzycki, M. (eds.) Proceedings of the IMCSIT, Vol. 4. pp. 659–666. IEEE Computer Society Press (2009)

13. Flouri, T., Janoušek, J., Melichar, B.: Tree pattern matching by deterministic pushdown automata (2009), draft
14. Gecseg, F., Steinby, M.: Tree languages. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, vol. 3 Beyond Words. Handbook of Formal Languages, pp. 1–68. Springer–Verlag, Berlin (1997)
15. Hoffmann, C.M., O'Donnell, M.J.: Pattern matching in trees. J. ACM 29(1), 68–95 (1982)
16. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation. Addison-Wesley, Boston, 2nd edn. (2001)
17. Janoušek, J.: String suffix automata and subtree pushdown automata. In: Holub, J., Žďárek, J. (eds.) Proceedings of the Prague Stringology Conference 2009. pp. 160–172. Czech Technical University in Prague, Czech Republic (2009), available on: http://www.stringology.org/event/2009
18. Janoušek, J., Melichar, B.: On regular tree languages and deterministic pushdown automata. Acta Inf. 46(7), 533–547 (2009)
19. Janoušek, J., Melichar, B.: Subtree and tree pattern pushdown automata for trees in prefix notation (2009), submitted for publication
20. Janoušek, J., Melichar, B.: Finding repeats of subtrees in a tree using pushdown automata (2010), submitted for publication
21. London stringology days 2009 conference presentations. Available on: http://www.dcs.kcl.ac.uk/events/LSD&LAW09/, King's College London, London (2009)
22. Melichar, B., Holub, J., Polcar, J.: Text searching algorithms. Available on: http://stringology.org/athens/ (2005), release November 2005
23. Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages. Springer–Verlag, Berlin (1997)
24. Rozenberg, G., Salomaa, A. (eds.): Vol. 1: Word, Language, Grammar, Handbook of Formal Languages. Springer–Verlag, Berlin (1997)
25. Shankar, P., Gantait, A., Yuvaraj, A.R., Madhavan, M.: A new algorithm for linear regular tree pattern matching. Theor. Comput. Sci. 242(1-2), 125–142 (2000)
26. Smyth, B.: Computing Patterns in Strings. Addison-Wesley-Pearson Education Limited, Essex, England (2003)
27. Valiant, L.G., Paterson, M.: Deterministic one-counter automata. In: Automaten theorie und Formale Sprachen. pp. 104–115 (1973)
28. Wagner, K., Wechsung, G.: Computational Complexity. Springer–Verlag, Berlin (2001)

Tomáš Flouri graduated at the Department of Computer Science and Engineering, Faculty of Electrical Engineering of the Czech Technical University in Prague in 2008. Since 2008 he has been a Ph.D. student at the same department. His scientific research focuses on tree algorithms using pushdown automata.

Jan Janoušek graduated the Department of Computer Science and Engineering, Faculty of Electrical Engineering of the Czech Technical University in Prague in 1994. He received his Ph.D. in the field of parsing and translation in 2001. His research interests include tree algorithms, parsing and translation algorithms and attribute grammars. Since July 2009 he has been working as an assistant professor at the Department of Theoretical Computer Science, Faculty of Information Technology of the Czech Technical University in Prague.

Bořivoj Melichar graduated the Faculty of Electrical Engineering of the Czech Technical University in Prague in 1964. During 1964 - 2009 he worked at the Department of Computer Science and Engineering, Faculty of Electrical Engineering of the Czech Technical University in Prague. His interests include parsing and translation algorithms, attribute grammars, and text and tree algorithms. Since the beginning of 2010 he has been working as a full professor at the Department of Theoretical Computer Science, Faculty of Information Technology of the Czech Technical University in Prague.

Received: November 16, 2009; Accepted: December 25, 2009.

UDC 007:681.32, DOI: 10.2298/CSIS1002359L

A Tool for Modeling Form Type Check Constraints and Complex Functionalities of Business Applications

Ivan Luković[1], Aleksandar Popović[2], Jovo Mostić[1], and Sonja Ristić[1]

[1] University of Novi Sad, Faculty of Technical Sciences,
Trg D. Obradovića 6, 21000 Novi Sad, Serbia
{ivan, sdristic}@uns.ac.rs, jovom@t-com.me
[2] University of Montenegro, Faculty of Science,
Džordža Vašingtona bb, 81000 Podgorica, Montenegro
aleksandarp@rc.pmf.ac.me

Abstract. IIS*Case is a software tool that provides information system modeling and prototypes generation. At the level of platform independent model specifications, IIS*Case provides conceptual modeling of database schemas that include specifications of various database constraints, such as domain, not null, key and unique constraints, as well as various kinds of inclusion dependencies. It also provides conceptual modeling of business applications. In the paper, we present new concepts and a tool embedded into IIS*Case, that is aimed at supporting specification of check constraints. We present a domain specific language for specifying check constraints and a tool that enables visually oriented design and parsing check constraints. Also, we present concepts and a tool that is aimed at supporting specification of complex (i.e. "nonstandard") functionalities of business applications. It is provided visually oriented and platform independent specification of business application functions.

Keywords: Information system design; Platform Independent Models and Model Driven Software Development; Check constraint specification; Function specification.

1. Introduction

Integrated Information Systems CASE Tool (IIS*Case) is a software tool aimed at assisting the information system (IS) design and at generating executable application prototypes. Currently, IIS*Case provides:

- Conceptual modeling of database schemas, transaction programs, and business applications of an IS;
- Automated design of relational database subschemas in the 3rd normal form (3NF);

- Automated integration of subschemas into a unified database schema in the 3NF;
- Automated generation of SQL/DDL code for various database management systems (DBMSs);
- Conceptual design of common user-interface (UI) models; and
- Automated generation of executable prototypes of business applications.

Apart from the tool, we also define a methodological approach to the application of IIS*Case in the software development process. By this approach, the software development process provided by IIS*Case is, in general, evolutive and incremental. We believe that it enables an efficient and continuous development of a software system, as well as an early delivery of software prototypes that can be easily upgraded or amended according to the new or changed users' requirements.

In the paper [11] we considered the application of the model-driven software engineering (MDSE) principles in IIS*Case. In our approach we strictly differentiate between the specification of a system and its implementation on a particular platform. Therefore, modeling is performed at the high abstraction level, because a designer creates an IS model without specifying any implementation details. Such a model may be classified as a Platform-Independent Model (PIM) of the MDA pattern ([9], [16], [17], [21], [22], [23]). Besides, IIS*Case provides some model-to-model transformations from PIM to Platform-Specific Models (PSM) and model-to-code transformations from PSMs to the executable program code.

In the paper [1] we argued that IIS*Case and our approach are suitable for end-user development (EUD), as it was considered in [3], [4], [20], and [25]. Besides, there are many EUD approaches and tools that provide the assistance to designers and end-users in creating IS specifications. One of them is presented in [24]. We also considered IIS*Case as a tool from the class of domain oriented design environments (DODE), as it is defined in [20]. In [1] we also present basic features of SQL Generator that are already implemented into IIS*Case, and aspects of its application. We also present methods for implementation of a selected database constraint, using mechanisms provided by a relational DBMS.

A case study illustrating main features of IIS*Case and the methodological aspects of its usage is given in [10], and accordingly we do not repeat the same explanations here. Apart from [1], [10] and [11], detailed information about IIS*Case may be found in several authors' references, as well as in [15] and [19]. The methodological approach to the application of IIS*Case is presented in more details in [13], while an approach to the formal specification of database constraints provided by IIS*Case is presented in [12].

At the abstraction level of PIMs, IIS*Case provides conceptual modeling of database schemas that include specifications of various database constraints, such as domain, not null, key and unique constraints, as well as various kinds of inclusion dependencies. Such a model is automatically transformed into a model of relational database schema, which is still technology independent specification. An SQL generator is embedded into IIS*Case. It provides further transformation of database schema into the platform specific SQL/DDL code, for various target DBMS platforms [1]. It is an example of model-to-code transformations provided by IIS*Case. Apart from the generation of key, unique, not null, and native referential integrity constraints, SQL Generator also provides the implementation of the default, partial and full referential integrity constraints, and the selection of an appropriate action from the set {No Action, Cascade, Set Default, Set Null}. It also provides the implementation of the inverse referential integrity constraints [1].

Previous versions of IIS*Case did not provide formal specification of check constraints, at all. Research efforts presented in this paper were directed toward introducing new concepts and a tool that enable a designer to formally specify and validate such constraints. An important expectation was to introduce new concepts that are platform independent, so as to provide formal specification of check constraints at the abstraction level of PIMs.

In the paper we present a domain specific language (DSL) aimed at defining check constraints at the level of PIMs. By means of this language, a designer may specify logical expressions of an arbitrary complexity for validating attribute values. The language provides a recognition and usage of other necessary PIM concepts embedded into IIS*Case, and therefore helps a designer in specifying expressions using problem domain concepts, as it is considered in [6], [8] and [14]. Besides, the language does not comprise any platform specific concepts, so check expressions are created at high abstraction level. In the paper we also present a tool aimed at specifying and parsing check constraints in a visually oriented way.

By this, in the process of database constraint design, we provide designers a possibility to concentrate mainly on the constraint semantics in a problem domain, instead of wasting time on their formal specification and validation. To achieve this goal, we need the appropriate DSLs and PIM concepts embedded into IIS*Case that are mostly problem oriented, instead of using relational data model concepts that are more technology specific, or even SQL DDL syntax, which is fully technology oriented programming language. Therefore, SQL DDL normally may be used to implement database schema specifications under a DBMS, but should not be directly used in the design of IS specifications, particularly at the conceptual level, i.e. at the abstraction level of PIMs.

At the abstraction level of PIMs, IIS*Case also provides conceptual modeling of business applications that include specifications of: (i) UI, (ii) structures of transaction programs aimed to execute over a database, and (iii) basic application functionality that includes the following "standard" operations: data retrieval, inserts, updates, and deletes. Also, a PIM model of business applications is automatically transformed into a program code of business applications. In this way, fully executable application prototypes are generated. For these purposes, User Interface Markup Language (UIML) and Java Render by Harmonia Incorporation® are chosen programming and run-time environment [19]. Such a generator is also an example of model-to-code transformations provided by IIS*Case and its development is almost finished.

Transaction programs of business applications may often comprise not only basic operations, but also more complex functionalities that cannot be expressed by a sole retrieve, insert, update, or delete operation. Such functionality may comprise complex calculations, as well as series of database operations. Therefore, such functionality we call specific, complex, or "nonstandard" application functionality. Besides, specifications of check constraints may reference various complex functions that should be specified also formally, i.e. in the same way as complex application functionality.

Basic data operations such as retrieve, insert, update and delete are common for various problem domains and can be easily specified by means of IIS*Case concepts. However, business applications from various problem domains usually comprise complex functionalities. If such functionalities would not be embedded into the PIM of a software system being designed, a programmer has to create latter a program code of such functionalities, or at least has to amend a generated program code, "by hand". In this way, complex functionalities are modeled at the lowest level of abstraction, by means of a target programming language which is always platform specific. As a rule, such created program code becomes unsynchronized with the initial PIM models of the system during the time. As a consequence, the operational maintenance of such systems becomes more difficult, with a lot of problems arising during the software exploitation.

Previous versions of IIS*Case did not provide formal specification of complex application functionalities or functions referenced in check constraints, at the level of PIMs. Research efforts presented in this paper were directed toward introducing new concepts and a tool that enable a designer to formally specify complex functionalities. An important expectation was to introduce new concepts that are platform independent, so as to provide formal specification of complex functionalities at the abstraction level of PIMs.

In the paper we also present concepts and a repository oriented tool aimed at the specification of functions at the level of PIMs. The name of the tool is *Function Specification Editor* or *Function Editor* for short. By means of *Function Editor* a designer may specify functions of an arbitrary complexity. It provides usage of necessary PIM concepts embedded into IIS*Case, and helps a designer in specifying functions using not only programming language concepts, but also problem domain concepts in a certain extent. Besides, *Function Editor* does not comprise any platform specific concepts, so functions are specified at high abstraction level. Also, it provides specifying functions completely in a visually oriented way. On the basis of *Function Editor* and the appropriate repository definitions used by *Function Editor* as a part of IIS*Case, it is possible to create a Domain Specific Language (DSL) for specifying business functions at the level of PIMs, as it is considered in [6], [8] and [14].

Apart from Introduction and Conclusion, the paper consists of six sections. In Section 2 we briefly describe main concepts of the IIS*Case tool that are important for specification of check constraints and function specifications. Check constraint expressions are introduced in Section 3, where grammar rules are presented. The main features and functionalities of the *Expression Editor* tool are presented in Section 4, while the implementation details concerning parsing of check expressions are presented in Section 5. Function specifications and related concepts are introduced in Section 6, while the main features and functionalities of the *Function Editor* tool are presented in Section 7.

2. Preliminaries

At the abstraction level of PIMs, IIS*Case currently provides conceptual modeling of database schemas and software applications of an IS. Starting from such PIM models as a source, a chain of transformations is performed so as to obtain executable program code of software applications and database SQL/DDL scripts for a selected target platform. The similar idea may be found also in [2]. For the purpose of readability, in this section we briefly describe main modeling concepts of IIS*Case that are used at the abstraction level of PIMs and have an influence on the specification of check constraints, as well as on the specification and referencing of functions defined in IIS*Case repository.

A *form type* is the main modeling concept in IIS*Case ([10], [12], [15]). It generalizes document types, i.e. screen forms that users utilize to communicate with an information system. The similar concept of the form type may be found in [5] and [7], as well as in many other references. Using the form type concept in IIS*Case, a designer specifies screen or report forms of transaction programs and, indirectly, specifies (i) an initial set of attributes and constraints, (ii) basic functionalities of future transaction programs and (iii) components of their UI. Each particular business document is observed as an instance of a form type. A form type concept, as well as related concepts of a domain and attribute, is platform independent. Here, we use a notion of the form type instead of a document type, because it is always a structure defined at the abstraction level of schema. It represents not only a layout structure (i.e. screen or a report form) of a document, but also a set of database schema attributes and constraints embedded into a future screen or a report form of an IS transaction program.

A form type is a named tree structure, whose nodes are called component types. Each *component type* is identified by its name in the scope of the form type, and has nonempty sets of attributes and keys, and a set of unique constraints that may be empty. Besides, to each component type must be associated a set of allowed database operations. It must be a nonempty subset of the set of "standard" operations {retrieve, insert, update, delete}. Each attribute of a component type is chosen from the set of all information system attributes.

Attributes are globally identified only by their names. IIS*Case imposes strict rules for specifying attributes and their domains. Attributes in IIS*Case are classified as elementary or derived. An attribute is elementary if it represents values given by end-users directly. Otherwise, it is derived. Values of a derived attribute are generated (i.e. calculated) from the values of the other attributes, by applying some algorithm. Such algorithms in IIS*Case are expressed by a concept of *function*. Therefore, a specification of a derived attribute must reference at least one previously defined (elementary or derived) attribute, and at least one function that is used for calculating its values.

Domains in IIS*Case are also globally identified only by their names. They are classified as primitive and user-defined. Primitive domains are defined "per se" as primitive data types. They are predefined into the repository of IIS*Case. An initial collection of primitive domains stored in the repository may be customized by adding, changing, or even removing specifications of primitive domains. Each user-defined domain in IIS*Case is created by referencing a primitive domain, or an already existing user-defined domain. In this way, user-defined domains are derived from primitive or previously created user-defined domains. There are four derivation rules that may be applied to create a user-defined domain from the existing domains: a) inheritance rule, b) tuple rule, c) set rule, and d) choice rule. A domain obtained by one of the aforementioned rules is called inherited, tuple, set, or choice domain, respectively. Tuple, set, or choice domains are also called complex domains. Recursive multiple application of the aforementioned rules is allowed.

Inherited domain inherits all the properties from its source (parent) domain. If a domain D is defined by the inheritance rule from the parent domain D_s, we denote it by $D = Inherits(D_s)$. Besides, a separate check expression is to be assigned to an inherited domain. Therefore, it is more or at least equally restrictive as its parent domain. If check expressions are defined for both inherited and its parent domain, in evaluation they are connected by the logical AND operator. Consequently, in a recursive application of the inheritance rule, all the domain check expressions in a hierarchy are connected by the logical AND operators.

Tuple domain represents tuples (records) of values over source domains. Therefore, it is defined as a structure $D = Tuple(A_1 : D_1,..., A_n : D_n)$, where D is a tuple domain, and for each $i \in \{1,...,n\}$, $(A_i : D_i)$ is a tuple item, i.e. a member, where A_i is an attribute with an associated source domain D_i.

Set domain represents values that are sets, each over the same source domain. Therefore, it is defined as a structure $D = Set\{D_s\}$, where D is a set domain, and D_s is a source domain.

Choice domain represents values over exactly one of the source domains. Therefore, it is defined as a structure $D = Choice(A_1 : D_1,..., A_n : D_n)$, where D is a choice domain, and for each $i \in \{1,...,n\}$, $(A_i : D_i)$ is a choice item, i.e. a member, where A_i is an attribute with an associated source domain D_i.

Check constraints in IIS*Case may be specified at the level of a domain, attribute or a component type of a form type. A check constraint associated to a domain or attribute is used to specify a logical condition constraining allowable values of a sole attribute. A check constraint associated to a component type is used to specify a logical condition constraining some values of each component type instance. Logical conditions of the check constraints may also reference functions, defined in IIS*Case repository.

3. Check Expressions

The quality of a whole database schema is substantially influenced by the quality of constraint specifications. It is very important to define these specifications at early stages of database schema design process, at abstraction level of PIMs, if possible. IIS*Case provides specification of various types of constraints, such as domain, not null, key and unique constraints, as well as various kinds of inclusion dependencies, at the abstraction level of PIMs.

Commercial CASE tools that provide modeling conceptual database schema specifications by means of Entity-Relationship (ER) data model and their transforming into the relational data model either provide only partial specifications of check constraints at the conceptual level, and/or provide a usage of standard SQL syntax for that purposes. Accordingly, check constraints may be fully defined only at the level of an implementation database schema specification, expressed commonly by relational data model and SQL syntax. For example, Oracle Designer does not allow all kind of check constraints to be formally defined at the level of an ER database schema. Sybase Power Designer provides a usage of SQL syntax for that purposes. On the contrary, check constraints in the IIS*Case tool are defined at the level of a conceptual database schema as a PIM model, which is expressed by a set of created form types. For these purposes, we developed a DSL to create check expressions of various complexity, in a platform independent way. Such a DSL and a tool embedded into IIS*Case enable a designer to specify check constraints using problem domain concepts, in a visually oriented way.

A check expression is a logical expression. In general, it may include attribute references, arithmetic, comparison and logical operators, as well as function calls. As implemented at the level of a target DBMS, it is usually evaluated in a ternary logic as a value from the set {*true*, *false*, *unknown*}, where *true* means that an expression is valid, *false* that it is violated, and *unknown* that it is neither valid nor violated. The value *unknown* is possible to obtain whenever there are null (missing) values of attributes in the evaluation of an expression.

By means of the DSL embedded into IIS*Case, check expressions may be specified at the level of a (i) domain, (ii) attribute or (iii) component type of a form type, in a similar way.

3.1. Domain Check Expressions

IIS*Case provides a "universal" set of all domains of a project as a whole. Domains in IIS*Case are used to express domain constraints, as it is proposed in [12]. Each specification of a user-defined domain allows defining a check expression, as a property of the domain specification. Such check expressions are named domain check expressions.

A formal specification of the grammar for domain check expressions is shown in Table 1, in the Extended Backus-Naur Form (EBNF) notation.

Table 1. Specification of the grammar for domain check expressions

```
Exp = Exp bin_operator Exp | un_operator Exp | Primary_Exp;
Primary_Exp = constant | value ['.' fieldName] |  function_name
'(' [Exp_List] ')' | '(' Exp ')';
Exp_List = Exp { ',' Exp_List};
```

The list of standard operators includes the following ones:

- Additive (+, -),
- Multiplicative (*, /),
- Comparison (<, <=, >, =>),
- Equality (==, !=),
- Concatenation (||),
- Boolean (NOT, AND, OR, XOR, =>),
- Inclusion (IN), and
- Pattern matching (LIKE).

All the operators and parentheses are introduced with the common meaning and priorities when applying the rules for evaluation of expressions.

Apart from introducing standard arithmetic, string, comparison and logical operators existing in all general-purpose languages, we decided also to introduce the operators LIKE and IN, which are common in database languages, like SQL. In this way, the language for check expressions becomes more problem oriented.

The grammar in Table 1 also provides function calls by referencing the appropriate function names. It is allowed to reference only the functions existing in the IIS*Case repository. It is supposed that both built-in and user-defined functions are stored in the repository. IIS*Case also provides a specialized DSL and a visually oriented tool for specifying various functions in a project. By this, it is possible to specify function header, a list of formal parameters, return value, all local declarations, function body and the exception handler in a structural way. Functions are specified by means of the technology independent concepts, at the abstraction level of PIMs, as it is presented in Sections 6 and 7.

The grammar in Table 1 allows the use of constants in check expressions. The common rules for specification and interpretation of constants are applied, and accordingly we do not describe them in more detail.

The only variable symbol allowed in domain check expression is *value* symbol (VALUE). VALUE denotes any value for which a domain check expression is validated.

Only in check expressions associated to a tuple or choice domain it is possible to qualify VALUE by the attribute name of an item. Therefore, VALUE.Ai denotes a value of a tuple or choice member (A_i : D_i), while nonqualified VALUE denotes a complete tuple or a choice value.

Example 1. A domain check expression for a numeric domain *DGRADE* is given:

```
VALUE >= 5 AND VALUE <= 10.
```

It constrains allowable values of *DGRADE* to the interval from 5 to 10. □

Example 2. A domain check expression for a string domain *DPHONE* is given:

```
VALUE LIKE '5%' AND StrLen(VALUE) == 7.
```

It constrains allowable values of *DPHONE* to exactly the 7 character long strings, beginning with '5'. StrLen is a function call that references a function already specified in the IIS*Case repository. □

Example 3. A domain check expression for a string domain *DSEMESTER* is given:

```
VALUE IN {'I', 'II','III','IV','V','VI','VII','VIII','IX', 'X'}.
```

It constrains allowable values of *DSEMESTER* to the list of string values specified after the inclusion operator IN. □

Example 4. A tuple domain *DDATE* is defined as *DATE* = *Tuple*(*DAY* : *INTEGER*, *MONTH* : *INTEGER*, *YEAR* : *INTEGER*), where INTEGER is primitive domain. A domain check expression for a tuple domain *DDATE* is given:

```
VALUE.DAY <= 31 AND VALUE.DAY >= 1.
```

It constrains allowable values of DAY member to the interval from 1 to 31. □

3.2. Attribute Check Expressions

IIS*Case provides a "universal" set of all attributes of a project as a whole. According to the universal relationship existence assumption (URSA) adopted from the relational data model, each attribute in IIS*Case is uniquely identified only by its name. Exactly one domain must be associated to each attribute in a project. In this way, allowable values of an attribute are constrained by the appropriate domain constraint.

IIS*Case allows defining a check expression as a property of the attribute specification. Such check expressions are named attribute check expressions. Our DSL has the appropriate grammar rules for specification of attribute check expressions.

Suppose that we have an attribute *A* to which a domain *D* is associated. We denote it as (*A* : *D*). If a domain check expression is associated to *D*, then each attribute *A* with the associated domain *D* inherits its domain check expression. Besides, if we have an attribute check expression assigned to an attribute *A*, and a domain check constraint assigned to *D*, where (*A* : *D*) holds, in evaluation they are connected by the logical AND operator. Obviously, if we have (possibly a recursive) application of the inheritance rule for the domain *D*, all the domain check expressions in a hierarchy are connected alongside with the attribute check expression by the logical AND operators.

A formal specification of the grammar for attribute check expressions is shown in Table 2, in EBNF notation. It is almost identical to the grammar specification for domain check constraints given in Table 1. The only difference is in the following. If we specify the attribute check expression for an attribute with the name *A*, the only variable symbol allowed in attribute check constraints, which may replace `attName`, is *A*. It is with the same meaning as it is the symbol VALUE in domain check expressions. Analogously to the domain check constraints, we may additionally qualify *A* in the case of a tuple or choice domain associated to *A*. Therefore, A.Ai denotes a value of a tuple or choice member (A_i : D_i), while nonqualified A denotes a complete tuple or a choice value.

Table 2. Specification of the grammar for attribute check expressions

```
Exp = Exp bin_operator Exp | un_operator Exp | Primary_Exp;
Primary_Exp   =   constant   |   attName   ['.'   fieldName]   |
function_name '(' [Exp_List] ')' | '(' Exp ')';
Exp_List = Exp { ',' Exp_List};
```

Example 5. An attribute check expression for a numeric attribute *GRADE* is given:

```
GRADE >= 6.
```

It constrains allowable values of *GRADE* to be greater or equal 6. If (*GRADE* : *DGRADE*) holds, where *DGRADE* is a domain from Example 1, then this check expression is connected to the one from Example 1 by the operator AND. Consequently, allowable values of *GRADE* are constrained to the interval from 6 to 10. □

3.3. Component Type Check Expressions

In IIS*Case, a form type is a hierarchical tree structure of component types, each of them having nonempty sets of attributes and keys, and a possibly empty set of unique constraints. Each attribute of a component type is selected from the set of all attributes of a project, i.e. from the IIS*Case repository. Therefore, it inherits all its constraints defined at the levels of the appropriate attribute and domain specifications.

IIS*Case also allows defining a check expression as a property of the component type specification. Such check expressions are named component type check expressions. Our DSL has the appropriate grammar rules for specification of component type check expressions.

The main purpose of domain and attribute check expressions is to constrain allowable values of a sole attribute. On the contrary, component type check constraints are used to specify logical conditions that constrain a tuple of values representing each component type instance.

A formal specification of the grammar for component type check expressions is shown in Table 3, in EBNF notation. It is almost identical to the grammar specification for attribute check constraints given in Table 2. The only difference is in the following. If we specify the component type check constraint for a component type *N*, we may use as variable symbols that are to replace `cmpattName`, any of attributes from the component type *N*, as well as any of attributes from any superordinated component type in a form type hierarchy.

Table 3. Specification of the grammar for component type check expressions

```
Exp = Exp bin_operator Exp | un_operator Exp | Primary_Exp;
Primary_Exp   =   constant   |   cmpattName   ['.'   fieldName]   |
function_name '(' [Exp_List] ')' | '(' Exp ')';
Exp_List = Exp { ',' Exp_List};
```

Analogously to the attribute check constraints, we may additionally qualify variable *A* in the case of a tuple or choice domain associated to *A*. Therefore, A.Ai denotes a value of a tuple or choice member (A_i : D_i), while nonqualified A denotes a complete tuple or a choice value.

Example 6. In Fig. 1 it is presented a form type *Student Records*. The form type is structured as a tree having two component types, STUDENT and GRADES, which are graphically represented by rectangles. The component type attributes are shown in italic letters. The key attribute of each component type is underlined by a solid line, whereas the attribute of a uniqueness constraint is underlined by a dashed line. Allowed operations for both component types are shown in small rectangles in the upper-right corners.

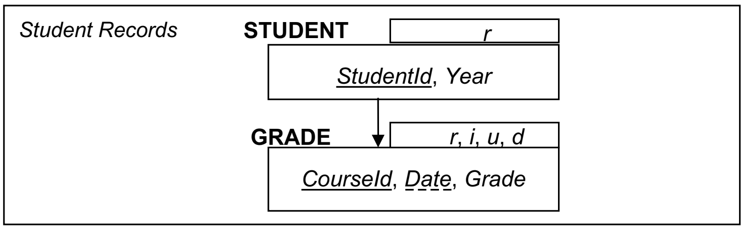


Fig. 1. A representation of the form type *Student Records*.

A check expression for the *GRADES* component type is given:

```
(Year IN {1, 2, 3} => Grade IN {1, 2, 3, 4})
                        AND (Year IN {4, 5} => Grade IN {4, 5}).
```

It constrains the possible combinations of values for Year and Grade. If Year is 1, 2, or 3, Grade must be 1, 2, 3, or 4, and if Year is 4 or 5, Grade must be 4 or 5. □

4. Check Expression Editor

Check Expression Editor, or *Expression Editor* for short, is a tool that we developed and embedded into IIS*Case. It is aimed at specification and validation of check expressions. It may be called from the

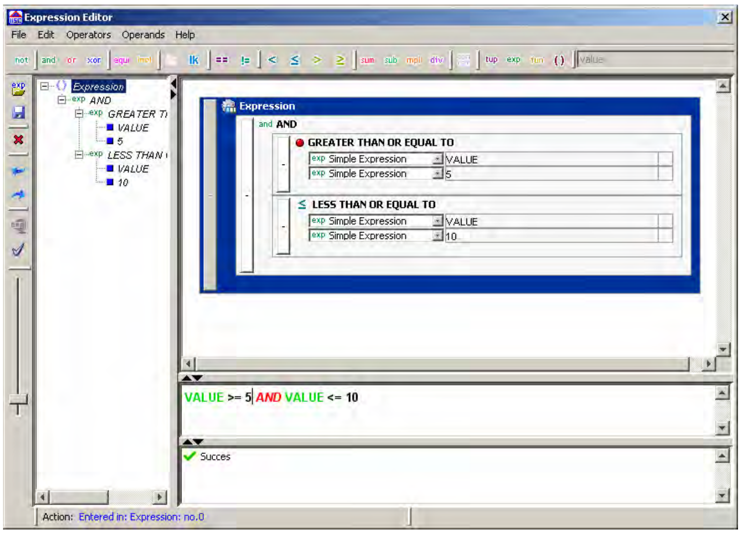
- Domain specification form of IIS*Case, if a domain check constraint need to be defined;
- Attribute specification form of IIS*Case, if an attribute check constraint need to be defined; or
- Component type specification form of IIS*Case, if a component type check constraint need to be defined.

By this, *Expression Editor* will support the appropriate check expression grammar, in a context-sensitive way.

Expression Editor provides two options for specification of check expressions: (i) guided, by means of a *Visual Editor*, and (ii) "free form", by means of a *Text Editor*. The first option is more suitable for less experienced users, not knowing the precise grammar rules and therefore needing a guide in specifying check expressions. The second one is more suitable for more experienced users, well knowing the precise grammar rules, and wishing to be as fast as possible in specifying check expressions. The main screen form of *Check Expression Editor* is presented in Fig. 2. *Visual Editor* is positioned in the center, while *Text Editor* is positioned in the bottom of the main form of *Expression Editor*.

Text Editor provides direct writing check expressions in a free form way. Besides, it supports context-sensitive syntax highlighting, as well as standard text processing commands such as: cut, copy, undo, etc. These commands are included in the *Edit* submenu of the main menu, and also in the toolbar positioned on the left hand side of the main form. Also, the toolbar comprises a command for performing expression validation.

By means of *Visual Editor*, check expressions are modeled by building the expression trees. Expression tree navigator, as a part of *Visual Editor*, is positioned on the left hand side of the main form from Fig. 2. Each node of an expression tree represents a subexpression, while the root node represents the main expression. Non-leaf nodes are named complex nodes, because they represent complex expressions, for example the expressions enclosed by parentheses, or operator inclusions. Leaf nodes are named simple nodes, because they correspond to simple, i.e. primary expressions, like constants, variables (such as VALUE or attribute names), or function calls.



```
VALUE >= 5 AND VALUE <= 10
```

✓ Succes

Action: Entered in: Expression: no.0

Fig. 2. Main screen form of *Expression Editor*

Visual Editor provides all common functions for editing an expression tree. These are: inserting, deleting, moving, and editing a node. The last function is available only for leaf nodes, representing simple expressions.

When a user wants to insert a complex node, he or she has to select a language operator or the parentheses symbol from the main toolbar. Each operator of the language is represented by an appropriate iconic button in the main toolbar.

Inserting a simple node into the expression tree is performed by selecting the *exp* command from the main toolbar. After selecting the *exp* command, a node is inserted and a textbox for specifying the simple expression appears within the node. According to grammar rules, simple expressions may be constants from a domain, variables, or function calls. A combo box positioned on the upper-right corner is aimed to assist a user to select an appropriate attribute, or a function from the IIS*Case repository.

Example 7. Suppose the following domain complex expression has to be specified by means of *Visual Editor*:

```
VALUE >= 5 AND VALUE <= 10.
```

A user needs first to insert a complex node for AND operator, and then two descendant complex nodes, one for ">=" and the other for "<=" operators.

Below the ">=" complex node, he or she needs to insert two simple nodes, one for variable VALUE, and the other for a constant 5. In a similar way, two simple nodes are to be defined below the "<=" complex node, one for VALUE, and the other for 10. □

Expression Editor always keeps *Visual Editor* and *Text Editor* synchronized. When a user creates and validates an expression by means of *Visual Editor*, the expression will be also shown in its full syntax in *Text Editor*. Also, when a user creates and validates an expression by means of *Text Editor*, the corresponding expression tree will be shown in *Visual Editor* automatically.

5. Validation of Check Expressions

Expression Editor provides validation of check expressions. Parser is created by means of the ANTRL 4.0 tool. ANTRL enables a user to formally specify grammar. Furthermore, it supports transformation of grammar specifications into the program code of a parser for target programming environment. As a result, it is obtained a recursive-descent parser expressed in a program code that is human-readable and easily customizable. [18].

According to the specified language definition presented in Section 3, ANTLR is used to generate Java program code of a parser that checks whether sentences created by *Expression Editor* conform to the language specification.

ANTLR generally provides amending grammar rules by adding source program code, i.e. code snippets to the grammar definition. Then, such code snippets are inserted into the program code of a generated parser, "as is". In our case, grammar rules for check expressions are amended by inserting code snippets that translate input sentences into an XML specification, and perform some semantic analysis, at the same time. In this way, apart from syntax validation, *Expression Editor* provides some semantic analysis. For example, check constraints may contain variables that reference members of a tuple or choice domain. The semantic analyzer verifies if reference to a tuple or choice member is valid, by seeking the appropriate domain specifications from the IIS*Case repository. Currently, type checking is not supported, at all. It is because the domain specification in our repository model still does not provide specification of allowed operators over a domain.

Example 8. In Table 4 two grammar rules for domain check expressions are presented. These rules contain code snippets that provide performing semantic analysis and creating a node in the appropriate XML specification. The grammar rules are specified in ANTLR notation.

Table 4. Grammar rules for domain check expressions containing code snippets

```
sentence returns [String val]
@init{ tmp = "";}
:
tmp = expression
{val="<block name=\"Expression\" group=\"1\">"+ tmp+"</block>";
val = val.replaceAll("><",">\n<") + "\n\n";}
;

domain_ref
@init{ tmp = "";}
: value ( '.' tmp = memberName )?
{checkMember(tmp);}
;
```

A code snippet that provides creating a node in the XML specification of a check expression is included in the *sentence* grammar rule in Table 4. It is given as follows:

```
{val = "<block name=\"Expression\" group=\"-1\">" + tmp +
                                                    "</block>";
val = val.replaceAll("><",">\n<") + "\n\n";}
```

A code snippet that provides performing semantic analysis is included in the *domain_ref* grammar rule in Table 4. It is given as follows:

```
{checkMember(tmp);}
```

When member name is identified, the snippet verifies if a reference to a tuple or choice member is valid, by seeking the appropriate tuple or choice domain specifications from the IIS*Case repository. □

Apart from being used for a semantic analysis, XML specifications of check expressions may also be used to provide further necessary transformations of check constraints. Our future research work is oriented towards providing a chain of transformations that result in PSM specifications of check constraints, expressed as the SQL/DDL program code.

The main idea how to design the transformation process from check expressions specified at the level of PIMs to the SQL/DDL program code is as follows. The process should be generally organized in two phases. By our methodology ([10], [13]), in the first phase, a set of form types representing a PIM model of a conceptual database schema is transformed into a relational database schema. Accordingly, all the constraints specified at the conceptual PIM level should be transformed into the equivalent relational database schema constraints. Therefore, each component type check expression specified at the level of a PIM, should be transformed into the one or more appropriate check or extended check expressions ([12]) defined at the level of the corresponding relation schemes. It is an issue how to create and implement an algorithm that will (i) provide inference problem solving for check expressions and (ii) preserve logical equivalency during transformations of component type check constraints. In this phase, domain and attribute check expressions remain unchanged.

A relational database schema generated in the first phase is still technology independent of any particular DBMS. Therefore, in the second phase, it is transformed into the SQL/DDL specification justified to the syntax of a chosen DBMS or ANSI SQL standard ([1]). Accordingly, each check expression defined at the level of a sole domain, attribute or a relation scheme, should be transformed into a corresponding SQL/DDL check constraint. Such a transformation is easily possible because of using a syntax for our check expressions that is very similar to the syntax for expressions in SQL check constraints. It is an issue here how to transform check expressions that contain references to the members of tuple or choice domains if a target DBMS does not support necessary object-relational concepts. On the other hand, with respect to the current level of supporting ANSI SQL standard by commercial DBMSs, extended check constraints in a relational database schema may only be transformed into the SQL code of a target DBMS that includes triggers and stored procedures.

6. Modeling Complex Functionalities in IIS*Case

Software development in IIS*Case is organized through projects. Each project in IIS*Case is further organized trough application systems and represented by a project tree. A set of fundamental specifications, comprising domains, attributes, inclusion dependencies, and program units is associated to each project. Fundamental specifications are independent of any application system given in a project. IIS*Case provides the following program unit concepts from the class of fundamental concepts necessary to express complex application functionalities at the level of PIMs: (i) Function; (ii) Package; and (iii) Event. A part of IIS*Case project tree representing these concepts is presented in Fig. 3.

A concept of a function is used to specify complex functionalities. Functions in IIS*Case are defined at the level of a project, and may be referenced from various IIS*Case specifications. A concept of a function is presented in the following text in more details.

A package is a collection of arbitrary selected functions defined in IIS*Case repository. Usually, packages are organized in a "thematic" way. Depending on a selected layer for the package deployment in multi-tier distributed software architecture, at the level of PIMs, we differentiate between database server, application server and client packages. Database server packages are to be deployed at the database server layer. The analogous is for application server and client packages.

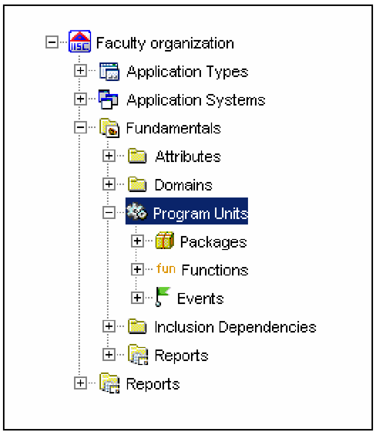


Fig. 3. A part of IIS*Case Project Tree.

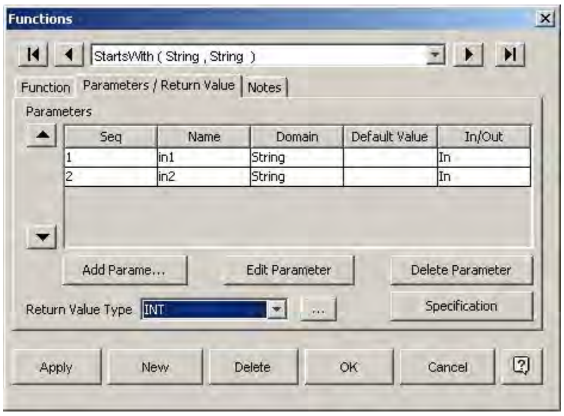
A concept of event is used at the level of PIMs, to represent any software event that may trigger some action under a specified condition. We also differentiate between database server, application server and client events. Database server events may be database triggers or exceptions. Application server and client events may be: keyboard events, mouse events, or exceptions. Each event should be associated to a PIM specification. For example, a database trigger should be associated to a relation scheme. A keyboard event may be associated to a form type, component type, or an attribute of a component type. A concept of event is not fully implemented in IIS*Case yet. Its full implementation is a matter of further research.

A formal specification of a function in IIS*Case includes the following:

- Function name that is unique in the IIS*Case project;
- List of formal parameters (i.e. arguments);
- Return value type; and
- Function body.

In Fig. 4 it is presented the IIS*Case screen form for specifying a function with the list of formal parameters and the return value type. The "Specification" button invokes the *Function Editor* tool aimed at formal specification of the function body. *Function Editor* is presented in the next section.

For each function, an arbitrary number of formal parameters may be defined. Each formal parameter is specified by the following properties: (i) sequence number defining a position of the parameter in the list; (ii) name; (iii) reference to IIS*Case domain defining a data type of a parameter; (iv) default value; and (v) type, where possible parameter types are: input (In), output (Out) and input/output (InOut), with a usual meaning inherited from various programming languages. Return value type is a reference to the domain previously defined in IIS*Case repository.



Seq	Name	Domain	Default Value	In/Out
1	in1	String		In
2	in2	String		In

Return Value Type INT

Fig. 4. A screen form for specification of a function, its formal parameters and a return type.

Function body is specified by means of PIM concepts that are mostly inherited from the third generation languages, particularly database procedural languages, and structural programming paradigm. Function body is a tree structure comprising blocks, declarations, statements, and comments. We differentiate between execution blocks and declaration blocks. Execution blocks may include nested declaration and execution blocks. In this way, multi-level nesting of blocks is provided. The following concepts are provided for specifying a function body:

- Sequential structures defining sequences of statements, declarations or comments;
- Declaration blocks that represent sequences of various declarations and comments;
- Declarations of local types, variables, constants, functions, cursors and exceptions;
- Execution blocks that represent sequences of embedded blocks, various statements and comments;
- Iteration structures with FOR, DO-WHILE, and WHILE-DO statements;
- Selection structures with IF-THEN-ELSE and ELSEIF-THEN-ELSE statements;
- Exception handler structure with TRY, CATCH, and FINALLY statements;
- Simple statements, like various kind of expressions and assignment statements; and
- Single-line comments denoted as /* */.

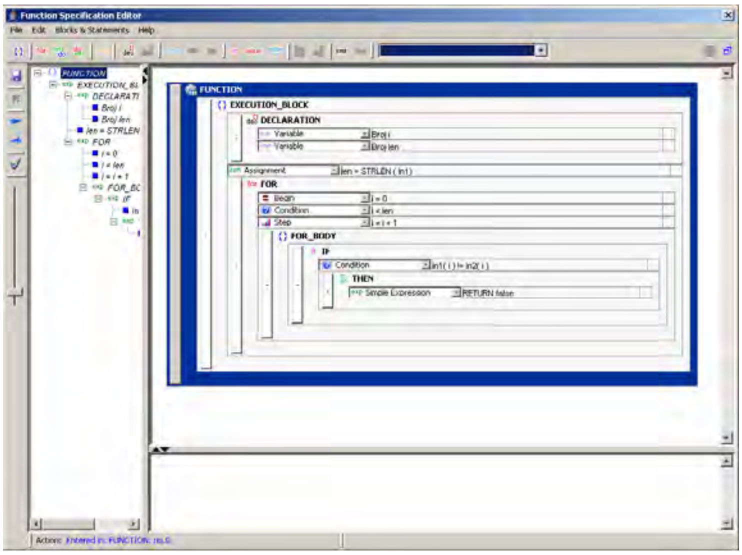
Despite that these concepts are mostly inherited from the third generation languages, they are syntactically independent of any particular programming language. Therefore, function specifications in IIS*Case are platform independent.

A specified function may be referenced many times in the same IIS*Case project. Currently, a function may be referenced in:

- Declarations and expressions of other IIS*Case functions;
- Packages, to express an inclusion of the function into a package;
- Events, to express the activity of an event associated to a PIM specification;
- Logical expressions of domain check constraints, attribute check constraints and component type check constraints; or
- Specifications of derived attributes.

7. The Function Editor Tool

Function Editor is the IIS*Case tool that provides repository based specification of a function body in a visually oriented way. The main screen form of *Function Editor* is presented in Fig. 5.



```
FUNCTION
  EXECUTION_BLOCK
    DECLARATION
      Variable             Broj i
      Variable             Broj len
    Assignment             len = STRLEN ( In1)
    FOR
      Begin                i = 0
      Condition            i < len
      Step                 i = i + 1
      FOR_BODY
        IF
          Condition        In1( i ) != In2( i )
          THEN
            Simple Expression   RETURN false
```

Fig. 5. The main screen form of *Function Specification Editor*.

By means of *Function Editor*, a function body is represented as a tree, whose nodes represent blocks, declarations, statements, or comments. *Tree Structure Navigator* is placed on the left hand side of the *Function Editor* screen form from Fig. 5, while the complete specification of a function body is represented in a panel placed in the central part of the screen form from Fig. 5. At the bottom of the screen form a message panel is placed.

Function Editor provides common tree operations, like creating a new node, removing an existing node, or reconnecting (cut & paste) a node in the tree. A notion of a current node in *Tree Structure Navigator* is recognized and all the tree operations are performed in the context of the current node. The current node is marked by a different color. Tree operations are available from the main menu, horizontal and vertical toolbars, as well as from the right-mouse-click context menu.

Creating a new node is a context sensitive operation. It is performed by selecting an appropriate toolbar option or "Blocks & Statements" menu option. A designer may select only one of the options that are available in the context of the current node. In this way, he or she specifies the type of the node being created. A list of all possible node types with their descriptions is given in Table 5 included in Section 10, Appendix.

By a context sensitive selection of options for node types that are available in the context of current node, *Function Editor* assists a designer in creating valid function specifications. For example, if the current node represents a FOR statement, a creation of ELSE descendant node is unavailable. According to common structural programming rules imposed by general purpose procedural languages, *Function Editor* only allows the combinations of node types that make sense in specifying a function body. In this way, *Function Editor* just allows building valid structures of a function body.

Besides, *Function Editor* also provides a syntax and semantic analysis tool. A designer may use the tool during the whole process of creating function specifications, just by selecting an appropriate toolbar option. The syntax analysis also checks validity of the structure of function body specification. As it concerns semantic analysis, currently *Function Editor* only checks variable and constant declarations, if specified data type is a reference to a domain specification from the IIS*Case repository. Type checking is not supported, at all. It is because the domain specification in our repository model still does not provide specification of allowed operators over a domain.

8. Conclusion

Commercial CASE tools that provide modeling conceptual database schema specifications by means of ER data model and their transforming into a relational data model either provide only partial specifications of check constraints at the conceptual level, and/or provide a usage of standard SQL syntax for that purposes. Therefore, check constraints are usually fully defined at the level of an implementation database schema. On the contrary, in our approach, check constraints in the IIS*Case tool are defined at the level of a conceptual database schema as a PIM model. For these purposes, we developed a DSL and the *Check Expression Editor* tool to create and parse check expressions defined in a platform independent way. In this way, a designer may specify check constraints using problem domain concepts, in a visually oriented way.

Besides, by our approach, function specifications, which may be referenced from check constraint expressions as well as from the other IIS*Case specifications, are defined at the level of a conceptual specification of an IS, as a PIM model. For these purposes, we developed a specialized tool, named *Function Editor*, by means of it is possible to create and analyze function specifications defined in a platform independent way. In this way, a designer may specify functions using not only programming concepts, but also problem domain concepts, in a visually oriented way.

Among all, our current or future research and development efforts are oriented towards the following:

- Development of the algorithms providing transformations of check constraint specifications created at the level of form types as PIMs, to the equivalent specifications at the level of an implementation database schema (usually expressed by the relational data model), and then to the executable PSM specifications expressed as the SQL/DDL program code;
- Development of a DSL for an equivalent representation of the current repository based function specifications at the level of PIMs;
- Extensions of the IIS*Case repository definition and the appropriate specifications (like event specifications) by new concepts, so as to make better foundation for (i) semantic analysis of check constraint expressions; and (ii) using function specifications in specifying business application logic, as well as their syntax and semantic analysis;
- Development of the algorithms providing transformations of function specifications created at the level of PIMs, to the equivalent executable PSM specifications expressed in a target programming environment and in the context of generated business applications; and
- Using the Meta-Object Facility Specification (MOF) in order to raise our repository based DSL specifications at meta-meta abstraction level.

Acknowledgment

A part of the research presented in this paper was supported by Ministry of Science and Technological Development of Republic of Serbia, Grant TR-13029, Title: *A Development of an Intelligent Environment for the Design and Implementation of Information Systems*.

References

1. Aleksić, S., Luković, I., Mogin, P., Govedarica, M.: A Generator of SQL Schema Specifications. Computer Science and Information Systems (ComSIS), Consortium of Faculties of Serbia and Montenegro, Novi Sad, Serbia, ISSN:1820-0214, Vol.4, No. 2, 79-98. (2007)
2. ARTech. *DeKlarit*™ (TheModel-Driven Tool for Microsoft Visual Studio 2005). Chicago, USA (June, 2009). [Online]. Available: http://www.deklarit.com.
3. Berti, S., Paterno, F., Santoro, C.: Natural Development of Ubiquitous Interfaces. Communications of the ACM (CACM), Association for Computing Machinery, USA, Vol. 47, No. 9, 63-64. (2004)
4. Burnett, M., Cook, C., Rothermel, G.: End-User Software Engineering. Communications of the ACM (CACM), Association for Computing Machinery, USA, Vol. 47, No. 9, 53-58. (2004)
5. Choobinch, J., Mannio, V. M., Nunamaker, F. J., Konsynski, R. B.: An Expert Database Design System Based on Analysis of Forms. IEEE Transactions on Software Engineering, Vol.14, No 2, 242-253. (1988)
6. Deursen van, A., Klint, P. Visser, J.: Domain-Specific Languages: An Annotated Bibliography. ACM SIGPLAN Notices, Association for Computing Machinery, USA, Vol. 35, No. 6, 26-36. (2000)
7. Diet, J., Lochovsky, F.: Interactive Specification and Integration of User Views Using Forms. In: Proceedings of the Eight International Conference on Entity-Relationship Approach, Toronto, Canada, 171-185. (1989)
8. João Pereira, M., Mernik, M., Cruz, D., Rangel Henriques, P.: Program Comprehension for Domain-Specific Languages. Computer Science and Information Systems (ComSIS), Vol. 5, No. 2, 1-17. (2008)
9. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise, Addison Wesley. (2003)
10. Luković, I., Mogin, P., Pavićević, J., Ristić, S.: An Approach to Developing Complex Database Schemas Using Form Types. Software: Practice and Experience, John Wiley & Sons Inc, Hoboken, USA, DOI: 10.1002/spe.820, Vol. 37, No. 15, 1621-1656. (2007)
11. Luković, I., Ristić, S., Aleksic, S., Popović, A.: An Application of the MDSE Principles in IIS*Case. In: Proceedings of III Workshop on Model Driven Software Engineering (MDSE 2008), Berlin, Germany, TFH, University of Applied Sciences Berlin, 53-62. (2008)
12. Luković, I., Ristić, S., Mogin, P.: On The Formal Specification of Database Schema Constraints. In: Proceedings of I Serbian – Hungarian Joint Symposium on Intelligent Systems, Subotica, Serbia, 125-136. (2003)
13. Luković, I., Ristić, S., Mogin, P., Pavicević, J.: Database Schema Integration Process – A Methodology and Aspects of Its Applying. Novi Sad Journal of Mathematics (Formerly Review of Research, Faculty of Science, Mathematic Series), Serbia, Vol. 36, No. 1, 115-150. (2006)
14. Mernik, M., Heering, J., Sloane, M. A.: When and How to Develop Domain-Specific Languages. ACM Computing Surveys (CSUR), Association for Computing Machinery, USA, Vol. 37, No. 4, 316-344. (2005)
15. Mogin, P., Luković, I., Karadžić, Z.: Relational Database Schema Design and Application Generating using IIS*CASE Tool. In: Proceedings of International Conference on Technical Informatics, Timisoara, Romania, Vol. 5, 49-58. (1994)
16. Object Management Group: MDA Guide. Version 1.0.1, Volume 1, document omg/03-06-01. (2003)

17. Object Management Group: Unified Modeling Language Specification. Version 1.4.2, document formal/05-05-01. (2005)
18. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. The Pragmatic Programmers, USA. (2007).
19. Pavićević, J., Luković, I., Mogin, P., Govedarica, M.: Information System Design and Prototyping Using Form Types. In: Proceedings of INSTICC I International Conference on Software and Data Technologies (ICSOFT), Setubal, Portugal, Vol. 2, 157-160. (2006)
20. Reppening, A., Ioannidou, A.: Agent Based End-User Development. Communications of the ACM (CACM), Association for Computing Machinery, USA, Vol. 47, No. 9, 43-46. (2004)
21. Rumbaugh, J., Jacobson, I.: The Unified Modeling Language Reference Manual. Addison-Wesley, USA. (1999)
22. Schmidt, D. C.: Model-driven engineering. IEEE Computer, Vol.39, No.2, 25-31. (2006)
23. Seidewitz, E.: What models mean. IEEE Software, Vol. 20, No. 5, 26-32. (2003)
24. Stanley, E., Mogin, P., Andreae, P.: S.E.A.L.-A Query Language for Entity-Association Queries. In Proceedings of the 20th Australasian Database Conference (ADC 2009), Wellington, New Zealand, Vol 92, 67-76. (2009)
25. Sutcliffe, A., Mehandjiev, N.: End-User Development. Communications of the ACM (CACM), Association for Computing Machinery, USA, Vol. 47, No. 9, 31-32. (2004)

Appendix

In Table 5 it is presented a list of all possible *Function Editor* node types with their descriptions.

Table 5. A list of node types available when creating a new node.

Node Type	Description
Execution Block	A new execution block as a sequence of statements, blocks and comments is created. The node is named EXECUTION_BLOCK. In its context, it is possible to create new subordinated nodes, and therefore such a node is called the complex node.
FOR structure	A new node named FOR and representing the counting FOR structure is created. Four new subordinated nodes are automatically created, denoted as: (i) Begin, (ii) Condition, (iii) Step, and (iv) FOR_BODY. The first three are text items that define: start value, end value and the step of a FOR program counter. These are the simple nodes, because they cannot have any subordinated nodes. FOR_BODY is a complex node. It represents a sequence of state-ments and blocks defining the body of a FOR structure.

WHILE-DO structure	A new node named WHILE and representing the WHILE-DO structure is created. Two new subordinated nodes are automatically created, denoted as: (i) Condition and (ii) WHILE_BODY. Condition is a text item that defines "pre-while" test condition. It is a simple node. WHILE_BODY is a complex node. It represents a sequence of statements and blocks defining the body of a WHILE-DO structure.
DO-WHILE structure	A new node named DO_WHILE and representing the DO-WHILE structure is created. Two new subordinated nodes are automatically created, denoted as: (i) DO_WHILE_BODY and (ii) Condition. Condition is a text item that defines "post-while" test condition. It is a simple node. DO_WHILE_BODY is a complex node. It represents a sequence of statements and blocks defining the body of a DO-WHILE structure.
IF-THEN-ELSE structure	A new node named IF and representing the IF selection structure is created. Three new subordinated nodes are automatically created, denoted as: (i) Condition, (ii) THEN, and (iii) ELSE, as an optional node. Condition is a text item that defines IF test condition. It is a simple node. THEN and ELSE are complex nodes. They represent sequences of statements and blocks defining the main body and the alternative body of an IF structure.
ELSE clause	A new node named ELSE in the context of an IF selection structure is created, with the same role as it would be created initially trough an IF-THEN-ELSE structure.
ELSEIF structure	A new node named ELSEIF in the context of an IF selection structure is created with a usual meaning. Three new subordinated nodes are automatically created, denoted as: (i) Condition, (ii) THEN, and (iii) ELSE, as an optional node. Condition is a text item that defines ELSEIF test condition. It is a simple node. THEN and ELSE are complex nodes. They represent sequences of statements and blocks defining the main body and the alternative body of an ELSEIF structure.

TRY-CATCH-FINALLY structure	Three complex nodes named TRY, CATCH and FINAL-LY are automatically created to specify an exception handler structure. CATCH and FINALLY nodes are the optional ones. They represent sequences of statements and blocks defining the exception handler. In the scope of CATCH, two new subordinated nodes are automatically created, denoted as: (i) Exception and (ii) CATCH_BLOCK. Exception is a simple node. It is a text item that references a previously declared exception. CATCH_BLOCK is a complex node. It represents a sequence of statements and blocks aimed to handle a raised exception. Multiple nesting of TRY nodes is allowed. In the scope of a current TRY node it is possible to create many CATCH or FINALLY nodes.
Statement	A new node representing a simple statement is created in the context of a block. It is a simple node structured as a text item. Currently, there are two types of simple statements: assignments and expressions. In the future research, we also plan to embed SQL statements.
Declaration Block	A new declaration block as a sequence of declarations and comments is created. The node is named DECLARATION. It is a complex node. In its context, it is possible to create new declarations of types, variables, constants, cursors, exceptions, and local functions.
Declaration	A new declaration is created in the context of a declaration block. A declaration is a simple node. It represents a text item that defines particular declaration of a type, variable, constant, cursor, exception or function inclusion.
LOCAL_FUNCTION declaration	A new node named LOCAL_FUNCTION is created in the scope of a declaration block. It represents a declaration of a local function. Three new subordinated nodes are automatically created, denoted as: (i) Function Name, (ii) ARGUMENTS, and (iii) LOCAL_FUNCTION_BODY. Function Name is a simple node. It is a text item that defines local function name. ARGUMENTS is a complex node. It comprises declarations of local function arguments only. LOCAL_-FUNCTION_BODY is a complex node. It represents a whole function body of a local function being declared.

Local Function Argument	A new node in the context of an ARGUMENTS node in a LOCAL_FUNCTION declaration is created. It is a simple node structured as a text item. It represents a formal argument of a local function given with the name and an association to a domain from the repository.
Comment	A new node in the context of a block is created. It is a simple node structured as a text item. It represents a single-line comment.

Ivan Luković received his M.Sc. (5 year, former Diploma) degree in Informatics from the Faculty of Military and Technical Sciences in Zagreb in 1990. He completed his Mr (2 year) degree at the University of Belgrade, Faculty of Electrical Engineering in 1993, and his Ph.D. at the University of Novi Sad, Faculty of Technical Sciences in 1996. Currently, he works as a Full Professor at the Faculty of Technical Sciences at the University of Novi Sad, where he lectures in several Computer Science and Informatics courses. His research interests are related to Database Systems and Software Engineering. He is the author or coauthor of over 70 papers, 4 books, and 30 industry projects and software solutions in the area.

Aleksandar Popović graduated from Faculty of Science at the University of Montenegro. He completed his Mr (2 year) degree at the University of Novi Sad, Faculty of Technical Sciences. Currently, he is a Ph.D. student and teaching assistant at the University of Montenegro, Faculty of Science. He assists in teaching several Computer Science and Informatics courses. His research interests include Software Engineering, Database Systems and Domain Specific Languages.

Jovo Mostić received his M.Sc. (4 year, former Diploma) degree from the University of Montenegro, Faculty of Science in Podgorica. He completed his Mr (2 year) degree at the University of Novi Sad, Faculty of Technical Sciences. Currently, he works as an IT project manager in Erste & Steiermärkische Bank in Podgorica. His research interests are related to Information Systems, Database Systems and Software Engineering.

Sonja Ristić is holding a position of an associate professor at the University of Novi Sad, Faculty of Technical Sciences, Serbia. She received two bachelor degrees (4 year, former Diploma) from University of Novi Sad, one in Mathematics, Faculty of Science in 1983, and the other in Economics from Faculty of Economics, in 1989. She also received her Mr (2 year) and Ph.D. degrees in Informatics, both from Faculty of Economics, in 1994 and 2003, respectively. From 1984 till 1990 she worked with the Novi Sad Cable Company "NOVKABEL" – Factory of Electronic Computers. From 1990 till 2006 she was with High School of Professional Business Studies -Novi Sad, and since 2006 she has been with the Faculty of Technical Sciences, University of Novi Sad. Her research interests are related to Database Systems and Software Engineering. She is the author or coauthor of over 40 papers in the area.

Received: November 12, 2009; Accepted: January 27, 2010.

CIP – Каталогизација у публикацији
Народна библиотека Србије, Београд

004

COMPUTER Science and Information Systems : the International journal / Editor-in-Chief Mirjana Ivanović. – Vol. 7, No 2 (2010) - . – Novi Sad (Trg D. Obradovića 3): ComSIS Consortium, 2010 - (Belgrade : Sigra star). –30 cm

Polugodišnje. – Tekst na engleskom jeziku

ISSN 1820-0214 = Computer Science and Information Systems
COBISS.SR-ID 112261644

Cover design: V. Štavljanin
Library Administration: M. Petković
Printed by: Sigra star, Beograd

Computer Science and Information Systems

Volume 7, Number 2, Special Issue, April 2010

Contents

Editorial

Guest Editorial

Papers

247 Comparing General-Purpose and Domain-Specific Languages: An Empirical Study
Tomaž Kosar, Nuno Oliveira, Marjan Mernik, Maria João Varanda Pereira, Matej Črepinšek, Daniela da Cruz, and Pedro Rangel Henriques

265 VisualLISA: A Visual Environment to Develop Attribute Grammars
Nuno Oliveira, Maria João Varanda Pereira, Pedro Rangel Henriques, Daniela da Cruz, and Bastian Cramer

291 Annotation Based Parser Generator
Jaroslav Porubän, Michal Forgáč, Miroslav Sabo, and Marek Běhálek

309 On Automata and Language Based Grammar Metrics
Matej Črepinšek, Tomaž Kosar, Marjan Mernik, Julien Cervelle, Rémi Forax, Gilles Roussel

331 Subtree Matching by Pushdown Automata
Tomáš Flouri, Jan Janoušek, and Bořivoj Melichar

359 A Tool for Modeling Form Type Check Constraints and Complex Functionalities of Business Applications
Ivan Luković, Aleksandar Popović, Jovo Mostić, and Sonja Ristić

ISSN: 1820-0214