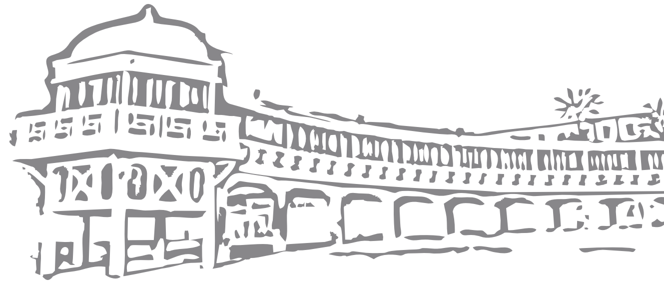

Jornadas Sistedes 2014

Cádiz, del 16 al 19 de septiembre

JISBD PROLE JCIS DC

# XIV Jornadas sobre Programación y Lenguajes

# ACTAS

XIV Jornadas sobre Programación y Lenguajes

Editor:

Santiago Escobar

# Preface

This report contains the informal proceedings of the *XIV Jornadas sobre Programación y Lenguajes (PROLE 2014)*, held at Cádiz, Spain, during September 17th-19th, 2014. Previous editions of the workshop were held in Madrid (2013), Almería (2012), A Coruña (2011), València (2010), San Sebastián (2009), Gijón (2008), Zaragoza (2007), Sitges (2006), Granada (2005), Málaga (2004), Alicante (2003), El Escorial (2002), and Almagro (2001).

Programming languages provide a conceptual framework which is necessary for the development, analysis, optimization and understanding of programs and programming tasks. The aim of the PROLE series of conferences (PROLE stems from the spanish PROgramación y LEnguajes) is to serve as a meeting point for spanish research groups which develop their work in the area of programming and programming languages. The organization of this series of events aims at fostering the exchange of ideas, experiences and results among these groups. Promoting further collaboration is also one of the main goals of PROLE.

PROLE welcomes both theoretical and practical works concerning the specification, design, implementation, analysis, and verification of programs and programming languages. More precisely, the topics of interest include, but are not restricted to:

- Programming paradigms (concurrent, functional, imperative, logic, agent-, aspect-, object oriented, visual, ...) and their integration,
- Specification and specification languages,
- Type systems,
- Languages and techniques for new computation and programming models (DNA and quantum computing, genetic programming, ...),
- Compilation; programming language implementation (tools and techniques),
- Semantics and their application to the design, analysis, verification, and transformation of programs,
- Program analysis techniques,
- Program transformation and optimization, and
- Tools and techniques for supporting the development and connectivity of programs (modularity, generic programming, markup languages, WWW, ...).

The Program Committee of PROLE 2014 collected four reviews for each paper and held an electronic discussion during July 2014. The contributions included in this informal proceedings belong to one of the following categories:

1. Original works (3 contributions)
2. Tutorials (1 contribution)
3. Tool systems (2 contributions)
4. High-level already published papers (10 contributions).
5. Work in progress (7 contributions)

In addition to the selected contributions, the scientific program includes an invited lecture by Michael Ernst from the University of Washington, USA. We would like to thank him for having accepted our invitation.

We would also like to thank all the members of the Program Committee and all the referees for their careful work in the review and selection process. Many thanks to all authors who submitted papers and to all conference participants. Finally, we express our gratitude to all the members of the local organization of SISTEDES 2014 in Cádiz.

Valencia, Spain, Santiago Escobar
July 2014 PROLE 2014 Editor

# Organization

## Program Committee

| | |
|---|---|
| Jesús Almendros | Universidad de Almería, Spain |
| María Alpuente | Universitat Politècnica de València, Spain |
| Puri Arenas | Universidad Complutense de Madrid, Spain |
| Manuel Carro | Universidad Politécnica de Madrid, Spain |
| Laura Castro | Universidade da Coruña, Spain |
| Francisco Durán | Universidad de Málaga, Spain |
| Santiago Escobar | Universitat Politècnica de València, Spain |
| María del Mar Gallardo | Universidad de Málaga, Spain |
| Raúl Gutiérrez | Universitat Politècnica de València, Spain |
| Lars-Ake Fredlund | Universidad Politécnica de Madrid, Spain |
| Salvador Lucas | Universitat Politècnica de València, Spain |
| Paqui Lucio | Euskal Herriko Unibertsitatea, Spain |
| Enrique Martín | Universidad Complutense de Madrid, Spain |
| Ginés Moreno | Universidad de Castilla la Mancha, Spain |
| Marisa Navarro | Euskal Herriko Unibertsitatea, Spain |
| Albert Oliveras | Universitat Politècnica de Catalunya, Spain |
| Fernando Orejas | Universitat Politècnica de Catalunya, Spain |
| Yolanda Ortega | Universidad Complutense de Madrid, Spain |
| Ricardo Peña | Universidad Complutense de Madrid, Spain |
| Adrián Riesco | Universidad Complutense de Madrid, Spain |
| Enric Rodríguez | Universitat Politècnica de Catalunya, Spain |
| Josep Silva | Universitat Politècnica de València, Spain |
| Alberto Verdejo | Universidad Complutense de Madrid, Spain |
| Alicia Villanueva | Universitat Politècnica de València, Spain |

# Contents

Verification games: Making software verification fun (Invite talk). . . . . . . . . . . . . . . . . . . . . . . . 1
*Michael Ernst*

EDD: A Declarative Debugger for Sequential Erlang Programs (High-level Work). . . . . . . . . . 3
*Rafael Caballero, Enrique Martín-Martín, Adrián Riesco and Salvador Tamarit*

Using Big-step and Small-step Semantics in Maude to Perform Declarative Debugging (High-level Work). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
*Adrián Riesco*

Correctness of Incremental Model Synchronization with Triple Graph Grammars (High-level Work). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
*Fernando Orejas and Elvira Pino*

Modular DSLs for flexible analysis: An e- Motions reimplementation of Palladio (High-level Work). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
*Antonio Moreno- Delgado, Francisco Durán, Steffen Zschaler and Javier Troya*

A Fuzzy Approach to Cloud Admission Control for Safe Overbooking (High-level Work). . . . 15
*Carlos Vázquez, Luis Tomás, Ginés Moreno and Johan Tordsson*

An operational framework to reason about policy behavior in trust management systems (High-level Work). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
*Edelmira Pasarella and Jorge Lobo*

Site-Level Template Extraction Based on Hyperlink Analysis (Original Work). . . . . . . . . . . . . 23
*Julián Alarte, David Insa, Salvador Tamarit and Josep Silva*

Space consumption analysis by abstract interpretation: Reductivity properties (High-level Work). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
*Manuel Montenegro, Ricardo Peña and Clara Segura*

Improving the Deductive System DES with Persistence by Using SQL DBMS's (Original Work). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
*Fernando Sáenz-Pérez*

XQOWL: An Extension of XQuery for OWL Querying and Reasoning (Work in Progress). . 55
*Jesús M. Almendros-Jiménez*

Logical Foundations for More Expressive Declarative Temporal Logic Programming Languages (High-level Work). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
*José Gaintzarain and Paqui Lucio*

Towards an XQuery-based implementation of Fuzzy XPath (Work in Progress). . . . . . . . . . . 73
*Jesús M. Almendros-Jiménez, Alejandro Luna Tedesqui and Ginés Moreno*

Modelling Hybrid Systems on a Concurrent Constraint Paradigm (Work in Progress). . . . . . 89
*Damián Adalid and María Del Mar Gallardo*

Enhancing Control over Jason Agents (Work in Progress). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
*Álvaro Fernández Díaz, Clara Benac Earle and Lars-Ake Fredlund*

Validación de tiempos de respuesta usando pruebas basadas en propiedades (Work in Progress). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 117
*Macías López and Laura M. Castro*

SACO: Static Analyzer for Concurrent Objects (High-level Work). . . . . . . . . . . . . . . . . . . . . . 133
*Elvira Albert, Puri Arenas, Antonio E. Flores Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martín-Martín, Germán Puebla and Guillermo Román-Díez*

SpecSatisfiabilityTool: A tool for testing the satisfiability of specifications on XML documents (Tool System). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 135
*Javier Albors and Marisa Navarro*

A Fuzzy Logic Programming Environment for Managing Similarity and Truth Degrees (Tool System). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 145
*Pascual Julián-Iranzo, Ginés Moreno, Jaime Penabad and Carlos Vázquez*

The ModelCC Model-Driven Parser Generator (Tutorial). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 155
*Fernando Berzal, Juan Carlos Cubero and Luis Quesada*

A certified reduction strategy for homological image processing (High-level Work). . . . . . . . . 173
*María Poza, César Domínguez, Jónathan Heras and Julio Rubio*

Lifting Term Rewriting Derivations in Constructor Systems by Using Generators (Original Work). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 175
*Adrián Riesco and Juan Rodríguez-Hortala*

Towards an Incremental and Modular Termination of Context Sensitive Rewriting Systems (Work in Progress). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 189
*Raul Gutierrez and Salvador Lucas*

Launchbury's semantics revisited: On the equivalence of context-heap semantics (Work in Progress). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 203
*Lidia Sánchez Gil, Mercedes Hidalgo-Herrero and Yolanda Ortega-Mallén*

# The ModelCC Model-Driven Parser Generator: A Tutorial

Fernando Berzal Francisco J. Cortijo Juan-Carlos Cubero Luis Quesada
CITIC & Department of Computer Science and Artificial Intelligence
University of Granada, Spain
{berzal|cb|jc.cubero|lquesada}@modelcc.org

Syntax-directed translation tools require the specification of a language by means of a formal grammar. This grammar must also conform to the specific requirements of the parser generator to be used. Software engineers then annotate the resulting grammar with semantic actions for the resulting system to perform its desired functionality. In this paper, we introduce ModelCC, a model-based parser generator that decouples language specification from language processing, avoiding some of the problems caused by grammar-driven parser generators. ModelCC receives a conceptual model as input, along with constraints that annotate it. It is then able to create a parser for the desired textual syntax and the generated parser fully automates the instantiation of the language conceptual model. ModelCC includes a reference resolution mechanism so that, rather than mere abstract syntax trees, it is able to instantiate abstract syntax graphs.

## 1 Introduction

The most widely-used language processing tools typically require language designers to provide a textual description of the language syntax as a BNF-like grammar. The proper specification of such a grammar is a nontrivial process that depends on the lexical and syntactic analysis techniques to be used, since each kind of technique requires the grammar to comply with different constraints. Each analysis technique is characterized by its expression power and this expression power determines whether a given analysis technique is suitable for a particular language. The most significant constraints on formal language specification originate from the need to consider context-sensitivity, the need of performing an efficient analysis, and some techniques' inability to consider grammar ambiguities or resolve conflicts caused by them.

Whenever the language syntax has to be modified, the language designer has to manually propagate changes throughout the entire language processor tool chain. These updates are time-consuming, tedious, and error-prone. By making such changes labor-intensive, the traditional approach hampers the maintainability and evolution of the language [16].

Moreover, it is not uncommon that different tools use the same language, including compilers, code generators, and debuggers. Multiple copies of the same language specification have to be maintained in sync, since language specification (i.e. its grammar) is tightly coupled to language processing (i.e. the semantic actions that annotate that grammar).

A grammar is a model of the language it defines, but a language can also be defined by a conceptual data model that represents the abstract syntax of the desired language, focusing on the elements the language will represent and their relationships. In conjunction with the declarative specification of some constraints, such model can be automatically converted into a grammar-based language specification [21]. The model representing the language can be modified as needed, without having to worry about the language processor and the peculiarities of the chosen parsing technique, since the corresponding language processor will be automatically updated.

Submitted to:
PROLE 2014

© F. Berzal, F.J. Cortijo, J.C. Cubero & L. Quesada
This work is licensed under the
Creative Commons Attribution License.

Furthermore, the conceptual model can be naturally implemented as a set of collaborating classes in object-oriented programming languages. Following proper software design principles, that implementation avoids the embedding of semantic actions within the language specification, as it is typically done with grammar-driven language processors.

Finally, as the language model is not bound to a particular parsing technique, evaluating alternative and/or complementary parsing techniques is possible without having to propagate their constraints into the language model. By using an annotated data model, model-based language specification completely decouples language specification from language processing, which can be performed using whichever parsing techniques that might be suitable for the formal language implicitly defined by the model.

It should be noted that, while, in general, the result of the parsing process is an abstract syntax tree that corresponds to a valid parsing of the input text according to the language concrete syntax, nothing prevents the model-based language designer from modeling non-tree structures. Indeed, a model-driven parser generator can automate the implementation of reference resolution mechanisms, among other syntactic and semantic checks that are typically deferred to later stages in the language processing pipeline [24]. ModelCC is able to resolve references and obtain abstract syntax graphs as the result of the parsing process, rather than the traditional abstract syntax trees obtained from conventional parser generators.

# 2 Model-Based Language Specification

In this Section, we analyze the concepts of abstract and concrete syntax (2.1), discuss the potential advantages of model-based language specification (2.2), and compare our proposed approach with the traditional grammar-driven language design process (2.3).

## 2.1 Abstract Syntax and Concrete Syntaxes

The abstract syntax of a language is just a representation of the structure of the different elements of a language without the superfluous details related to its particular textual representation [17]. On the other hand, a concrete syntax is a particularization of the abstract syntax that defines, with precision, a specific textual or graphical representation of the language. It should also be noted that a single abstract syntax can be shared by several concrete syntaxes [17].

For example, the abstract syntax of the typical *<if>-<then>-<optional else>* statement in imperative programming languages could be described as the concatenation of a conditional expression and one or two statements. Different concrete syntaxes could be defined for such an abstract syntax, which would correspond to different textual representations of a conditional statement, e.g. {"`if`", "`(`", expression, "`)`", statement, optional "`else`" followed by another statement} and {"`if`", expression, "`then`", statement, optional "`else`" followed by another statement, "`endif`"}.

The idea behind model-based language specification is that, starting from a single abstract syntax model (ASM) representing the core concepts in a language, language designers would later develop one or several concrete syntax models (CSMs). These concrete syntax models would suit the specific needs of the desired textual or graphical representation for the language sentences. The ASM-CSM mapping could be performed, for instance, by annotating the abstract syntax model with the constraints needed to transform the elements in the abstract syntax into their concrete representation.

## 2.2 Advantages of Model-Based Language Specification

Focusing on the abstract syntax of a language offers some benefits [17] and provides some potential advantages to model-based language specification over the traditional grammar-based language specification approach:

- When reasoning about the features a language should include, specifying its abstract syntax seems to be a better starting point than working on its concrete syntax details. In fact, we control complexity by building abstractions that hide details when appropriate [1].
- Sometimes, different incarnations of the same abstract syntax might be better suited for different purposes (e.g. an human-friendly syntax for manual coding, a machine-oriented format for automatic code generation, a Fit-like [18] syntax for testing, different architectural views for discussions with project stakeholders...). Therefore, it might be useful for a given language to support multiple syntaxes.
- Since model-based language specification is independent from specific lexical and syntactic analysis techniques, the constraints imposed by specific parsing algorithms do not affect the language design process. In principle, however, it might not be even necessary for the language designer to have advanced knowledge on parser generators when following a model-driven language specification approach.
- A full-blown model-driven language workbench [11, 25, 6, 14, 5, 13] would allow the modification of a language abstract syntax model and the automatic generation of a working IDE on the run. The specification of domain-specific languages would become easier, as the language designer could play with the language specification and obtain a fully-functioning language processor on the fly, without having to worry about the propagation of changes throughout the complete language processor tool chain.

In short, the model-driven language specification approach brings domain-driven design [9] to the domain of language design. It provides the necessary infrastructure for what Evans would call the 'supple design' of language processing tools: the intention-revealing specification of languages by means of abstract syntax models, the separation of concerns in the design of language processing tools by means of declarative ASM-CSM mappings, and the automation of a significant part of the language processor implementation.

## 2.3 Comparison with the Traditional Approach

A diagram summarizing the traditional language design process is shown in Figure 1, whereas the corresponding diagram for the model-based approach proposed in this paper is shown in Figure 2.

When following the traditional grammar-driven approach, the language designer starts by designing the grammar corresponding to the concrete syntax of the desired language, typically in BNF or a similar format. Then, the designer annotates the grammar with attributes and, probably, semantic actions, so that the resulting attribute grammar can be fed into lexer and parser generator tools that produce the corresponding lexer and parser, respectively. The resulting syntax-directed translation process generates abstract syntax trees from the textual representation in the concrete syntax of the language.

When following the model-driven approach, the language designer starts by designing the conceptual model that represents the abstract syntax of the desired language, focusing on the elements the language will represent and their relationships. Instead of dealing with the syntactic details of the language from

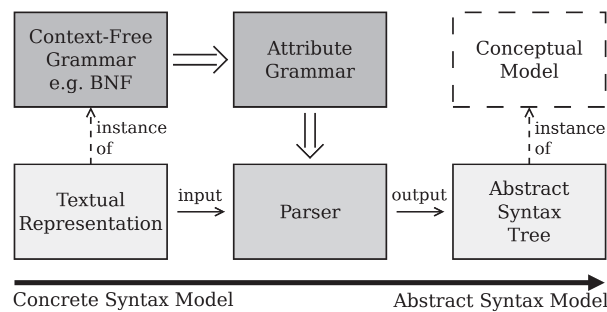


Figure 1: Traditional language processing approach.

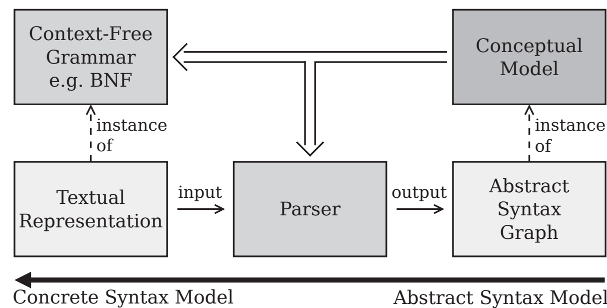


Figure 2: Model-based language processing approach.

the start, the designer devises a conceptual model for it (i.e. the abstract syntax model, or ASM), the same way a database designer starts with an implementation-independent conceptual database schema before he converts that schema into a logical schema that can be implemented in the particular kind of DBMS that will host the final database. In the model-driven language design process, the ASM would play the role of entity-relationship diagrams in database design and each particular CSM would correspond to the final table layout of the physical database schema in a relational DBMS.

Even though the abstract syntax model of the language could be converted into a suitable concrete syntax model automatically, the language designer will often be interested in specifying the details of the ASM-CSM mapping. With the help of constraints imposed over the abstract model, the designer will be able to guide the conversion from the ASM to its concrete representation using a particular CSM. This concrete model, when it corresponds to a textual representation of the abstract model, will be described by a formal grammar. It should be noted, however, that the specification of the ASM is independent from the peculiarities of the desired CSM, as a database designer does not consider foreign keys when designing the conceptual schema of a database. Therefore, the grammar specification constraints enforced by particular parsing tools will not impose limits on the design of the ASM. The model-driven language processing tool will take charge of that and, ideally, it will employ the most efficient parsing technique that works for the language resulting from the ASM-CSM mapping.

While the traditional language designer specifies the grammar for the concrete syntax of the language, annotates it for syntax-directed processing, and obtains an abstract syntax tree that is an instance of the implicit conceptual model defined by the grammar, the model-based language designer starts with an explicit full-fledged conceptual model and specifies the necessary constraints for the ASM-CSM mapping. In both cases, parser generators create the tools that parse the input text in its concrete syntax. The difference lies in the specification of the grammar that drives the parsing process, which is hand-crafted

in the traditional approach and automatically-generated as a result of the ASM-CSM mapping in the model-driven process.

Another difference stems from the fact that the result of the parsing process is an instance of an implicit model in the grammar-driven approach while that model is explicit in the model-driven approach. An explicit conceptual model is absent in the traditional language design process albeit that does not mean that it does not exist. On the other hand, the model-driven approach enforces the existence of an explicit conceptual model, which lets the proposed approach reap the benefits of domain-driven design.

There is a third difference between the grammar-driven and the model-driven approaches to language specification. While, in general, the result of the parsing process is an abstract syntax tree that corresponds to a valid parsing of the input text according to the language concrete syntax, nothing prevents the conceptual model designer from modeling non-tree structures, which describe grammars with a power of expression similar to reference attribute grammars [7]. Hence the use of the 'abstract syntax graph' term in Figure 2. This might be useful, for instance, for modeling graphical languages, which are not constrained by the linear nature of the traditional syntax-driven specification of text-based languages.

Instead of going from a concrete syntax model to an implicit abstract syntax model, as it is typically done, the model-based language specification process goes from the abstract to the concrete. This alternative approach facilitates the proper design and implementation of language processing systems by decoupling language processing from language specification, which is now performed by imposing declarative constraints on the ASM-CSM mapping.

## 3 ModelCC Model Specification

Once we have described model-driven language specification in general terms, we now proceed to introduce ModelCC [23], a tool that supports our proposed approach to the design of language processing systems. ModelCC, at its core, acts as a parser generator. The starting abstract syntax model is created by defining classes that represent language elements and establishing relationships among those elements (associations in UML terms). Once the abstract syntax model is established, its incarnation as a concrete syntax is guided by the constraints imposed over language elements and their relationships as annotations on the abstract syntax model. In other words, the declarative specification of constraints over the ASM establishes the desired ASM-CSM mapping.

In this section, we introduce the basic constructs that allow the specification of abstract syntax models, while we will discuss how model constraints help us establish a particular ASM-CSM mapping in the following section of this paper. Basically, the ASM is built on top of basic language elements, which might be viewed as the tokens in the model-driven specification of a language. Model-driven language processing tools such as ModelCC provide the necessary mechanisms to combine those basic elements into more complex language constructs, which correspond to the use of concatenation, selection, and repetition in the syntax-driven specification of languages.

Our final goal is to allow the specification of languages in the form of abstract syntax models such as the one shown in Figure 6, which will be used as an example in Section 5. This model, in UML format, specifies the abstract syntax model of the language supported by a simple arithmetic expression language. The annotations that accompany the model provide the necessary information for establishing the complete ASM-CSM mapping that corresponds to the traditional infix notation for arithmetic expressions. Moreover, the model also incorporates the method that lets us evaluate such arithmetic expressions. Therefore, Figure 6 represents a complete interpreter for arithmetic expressions in infix notation using ModelCC.

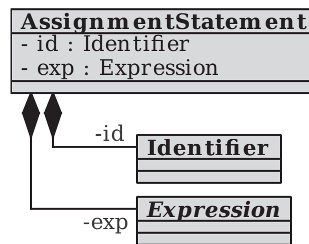


Figure 3: An assignment statement as an example of element composition (concatenation in textual CSM terms).

As mentioned above, the specification of the ASM in ModelCC starts with the definition of basic language elements, which can be modeled as simple classes in an object-oriented programming language. The ASM-CSM mapping of those basic elements will establish their correspondence to the tokens that appear in the concrete syntax of the language whose ASM we design in ModelCC.

In the following subsections, we describe the mechanisms provided by ModelCC to implement the three main constructs that let us specify complete abstract syntax models on top of basic language elements.

## 3.1 Concatenation

Concatenation is the most basic construct we can use to combine sets of language elements into more complex language elements. In textual languages, this is achieved just by joining the strings representing its constituent language elements into a longer string that represents the composite language element.

In ModelCC, concatenation is achieved by object composition. The resulting language element is the composite element and its members are the language elements the composite element collates.

When translating the ASM into a textual CSM, each composite element in a ModelCC model generates a production rule in the grammar representing the CSM. This production, with the nonterminal symbol of the composite element in its left-hand side, concatenates the nonterminal symbols corresponding to the constituent elements of the composite element in its right-hand side. By default, the order of the constituent elements in the production rule is given by the order in which they are specified in the object composition, but such an order is not strictly necessary (e.g. many ambiguous languages might require differently ordered sequences of constituent elements and even some unambiguous languages allow for unordered sequences of constituent elements).

The model in Figure 3 shows an example of object composition in ASM terms that corresponds to string concatenation in CSM terms. In this example, an assignment statement is composed of an identifier, i.e. a reference to its l-value, and an expression, which provides its r-value. In a textual CSM, the composite *AssignmentStatement* element would be translated into the following production rule: *<AssignmentStatement> ::= <Identifier> <Expression>*. Obviously, such production would probably include some syntactic sugar in an actual programming language, either for avoiding potential ambiguities or just for improving its readability and writability, but that is the responsibility of ASM-CSM mappings, which will be analyzed in Section 4.

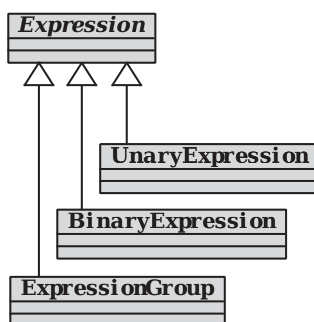


Figure 4: Subtyping for representing choices in ModelCC.

### 3.2 Selection

Selection is necessary as a language modeling primitive operation to represent choices, so that we can specify alternative elements in language constructs.

In ModelCC, selection is achieved by subtyping. Specifying inheritance relationships among language elements in an object-oriented context corresponds to defining 'is-a' relationships in a more traditional database design setting. The language element we wish to establish alternatives for is the superelement (i.e. the superclass in OO or the supertype in DB modeling), whereas the different alternatives are represented as subelements (i.e. subclasses in OO, subtypes in DB modeling). Alternative elements are always kept separate to enhance the modularity of ModelCC abstract syntax models and their integration in language processing systems.

In the current version of ModelCC, multiple inheritance is not supported, albeit the same results can be easily simulated by combining inheritance and composition. We can define subelements for the different inheritance hierarchies representing choices so that those subelements are composed by the single element that appears as a common choice in the different scenarios. This solution fits well with most existing programming languages, which do not always support multiple inheritance, and avoids the pollution of the shared element interface in the ASM, which would appear as a side effect of allowing multiple inheritance in abstract syntax models.

Each inheritance relationship in ModelCC, when converting the ASM into a textual CSM, generates a production rule in the CSM grammar. In those productions, the nonterminal symbol corresponding to the superelement appears in its left-hand side, while the nonterminal symbol of the subelement appears as the only symbol in the production right-hand side. Obviously, if a given superelement has $k$ different subelements, $k$ different productions will be generated representing the $k$ alternatives defined by the abstract syntax model.

The model shown in Figure 4 illustrates how an arithmetic *Expression* can be either an *UnaryExpression*, a *BinaryExpression*, or an *ExpressionGroup* in the language defined for a simple arithmetic calculator, as defined in Section 5. The context-free grammar resulting from the conversion of this ASM into a textual CSM would be: *<Expression> ::= <UnaryExpression>* | *<BinaryExpression>* | *<ExpressionGroup>*.

### 3.3 Repetition

Representing repetition is also necessary in abstract syntax models, since a language element might appear several times in a given language construct. When a variable number of repetitions is allowed,

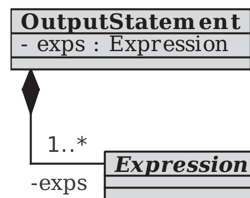


Figure 5: Multiple composition for representing repetition in ModelCC.

mere concatenation does not suffice.

Repetition is also achieved though object composition in ModelCC, just by allowing different multiplicities in the associations that connect composite elements to their constituent elements. The cardinality constraints described in Section 4 can be used to annotate ModelCC models in order to establish specific multiplicities for repeatable language elements.

Each composition relationship representing a repetitive structure in the ASM will lead to two additional production rules in the grammar defining a textual CSM: a recursive production of the form *<ElementList> ::= <Element> <ElementList>* and a complementary production *<ElementList> ::= <Element>*, where *<Element>* is the nonterminal symbol associated to the repeating element.

It should also be noted that *<ElementList>* will take the place of the nonterminal *<Element>* in the production derived from the composition relationship that connects the repeating element with its composite element (see the above section on how composition is employed to represent concatenation in ModelCC).

In practice, repeating elements will often appear separated in the concrete syntax of a textual language, hence repeatable elements can be annotated with separators, as we will see in Section 4. In case separators are employed, the recursive production derived from repeatable elements will be of the form *<ElementList> ::= <Element> <Separator> <ElementList>*.

When a repeatable language element is optional, i.e. its multiplicity can be 0, an additional epsilon production is appended to the grammar defining the textual CSM derived from the ASM: *<ElementList> ::=* $\varepsilon$.

For example, the model in Figure 5 shows that an *OutputStatement* can include several *Expression*s, which will be evaluated for their results in order for them to be sent to the corresponding output stream. This ASM would result in the following textual CSM grammar:

*<OutputStatement> ::= <ExpressionList>*

*<ExpressionList> ::= <Expression> <ExpressionList>* | *<Expression>*

## 4 ModelCC Model Constraints

Once we have examined the mechanisms that let us create abstract syntax models in ModelCC, we now proceed to describe how constraints can be imposed on such models in order to establish the desired ASM-CSM mapping:

- A first set of constraints is used for pattern specification, a necessary feature for defining the lexical elements of the concrete syntax model, i.e. its tokens.
- A second set of constraints is employed for defining delimiters in the concrete syntax model, whose use is common for eliminating language ambiguities or just as syntactic sugar in many languages.

| Constraints on | Annotation | Function |
|---|---|---|
| ... patterns | @Pattern | Pattern matching specification of basic language elements. |
| | @Value | Field where the recognized input token will be stored. |
| ... delimiters | @Prefix | Element prefix(es). |
| | @Suffix | Element suffix(es). |
| | @Separator | Element separator(s) in lists of elements. |
| ... cardinality | @Optional | Optional elements. |
| | @Multiplicity | Minimum and maximum element multiplicity. |
| ... evaluation order | @Associativity | Element associativity (e.g. left-to-right). |
| | @Composition | Eager or lazy composition for nested composites. |
| | @Priority | Element precedence level/relationships. |
| ... composition order | @Position | Element member relative position. |
| | @FreeOrder | When there is no predefined order among element members. |
| ... references | @ID | Identifier of a language element. |
| | @Reference | Reference to a language element. |
| Custom constraints | @Constraint | Custom user-defined constraint. |

Table 1: The constraints supported by the ModelCC model-based parser generator.

- A third set of ModelCC constraints lets us impose cardinalities on language elements, which control element repeatability and optionality.
- A fourth set of constraints lets us impose evaluation order on language elements, which are employed to declaratively resolve further ambiguities in the concrete syntax of a textual language by establishing associativity, precedence, and composition policies, the latter employed, for example, for resolving the ambiguities that cause the typical shift-reduce conflicts in LR parsers.
- A fifth set of constraints lets us specify the element constituent order in composite elements.
- A sixth set of constraints lets us specify referenceable language elements and references to them.
- Finally, custom constraints let us provide specific lexical, syntactic, and semantic constraints that take into consideration context information.

Table 1 summarizes the set of constraints supported by ModelCC for establishing ASM-CSM mappings between abstract syntax models and their concrete representation in textual CSMs.

As soon as that ASM-CSM mapping is established, ModelCC is able to generate the suitable parser for the concrete syntax defined by the CSM. ModelCC allows the definition of ASM-CSM constraints using metadata annotations or a domain-specific language.

Now supported by all the major programming platforms, metadata annotations have been used in reflective programming and code generation [10]. Among many other things, they can be employed for dynamically extending the features of your software development runtime [4] or even for building complete model-driven software development tools that benefit from the infrastructure provided by your compiler and its associated tools [12].

The ModelCC domain-specific language for ASM-CSM mappings [22] supports the separation of concerns in the design of language processing tools by allowing the definition of different CSMs for a common ASM.

# 5 A Simple Example

An interpreter for arithmetic expressions in infix notation can be used to illustrate the differences between ModelCC and more conventional tools. Its full implementation using two well-known parser generators (lex & yacc, and ANTLR) is available at `http://www.modelcc.org/examples`. Albeit the arithmetic expression example is necessarily simplistic, it already provides some hints on the potential benefits model-driven language specification can bring to more challenging endeavors. This simple language is also used in the next section as the basis for a more complex language, which illustrates how ModelCC supports language composition.

Using conventional tools, the language designer would start by specifying the grammar defining the arithmetic expression language in a BNF-like notation.

When using lex & yacc, the language designer converts the BNF grammar into a grammar suitable for LR parsing. Since lex does not support lexical ambiguities, the unary and binary operator nonterminals from the BNF grammar have to be refactored in order to avoid the ambiguities introduced by the use of + and - both as unary and binary operators. A similar solution is required for distinguishing operator priorities. Unfortunately, the resolution of ambiguities involves the introduction of a certain degree of duplication in the language specification: separate token types in the lexer and multiple parallel production rules in the parser. Once the ambiguities have been resolved, the language designer completes the lex & yacc introducing semantic actions to perform the necessary operations: albeit somewhat verbose using the C programming language syntax, the implementation of an arithmetic expression interpreter is relatively straightforward.

When using ANTLR, the language designer converts the BNF grammar into a grammar suitable for LL parsing. LL(*) parsers do not support left-recursion, so left-recursive grammar productions must be refactored. Since ANTLR provides no mechanism for the declarative specification of token precedences, such precedences have to be incorporated into the grammar. The usual solution involves the creation of different nonterminal symbols in the grammar, so that productions corresponding to the same precedence levels are grouped together. Once the grammar is adjusted to satisfy the constraints imposed by the ANTLR parser generator, the language designer can define the semantic actions needed to implement our arithmetic expression interpreter. The streamlined syntax of the scannerless ANTLR parser generator makes this implementation significantly more concise than the equivalent lex & yacc implementation.

When following a model-based language specification approach, the language designer starts by elaborating an abstract syntax model, which will later be mapped to a concrete syntax model by imposing constraints on the abstract syntax model. These constraints can also be specified as metadata annotations on the abstract syntax model and the resulting annotated model can be processed by automated tools, such as ModelCC, to generate the corresponding lexers and parsers. Annotated models can be represented graphically, as the UML diagram in Figure 6, or implemented using conventional programming languages, as the Java implementations available at `http://www.modelcc.org/examples`.

Using modern programming languages, metadata annotations can be used for the ASM-CSM mapping corresponding to the desired concrete syntax model. In case several CSMs were needed, the ModelCC domain-specific language for ASM-CSM mappings could be used to specify alternative CSMs for the language ASM [22].

The parser that ModelCC generates from the arithmetic expression model parses input strings such as "10/(2+3)*0.5+1" and instantiates *Expression* objects from them. The `eval()` method yields the final result for any expression (2, in the previous example). Figure 9 shows the actual code needed to generate and invoke the parser in ModelCC.

In its current version, ModelCC generates Lamb lexers [19] and Fence parsers [20], albeit traditional

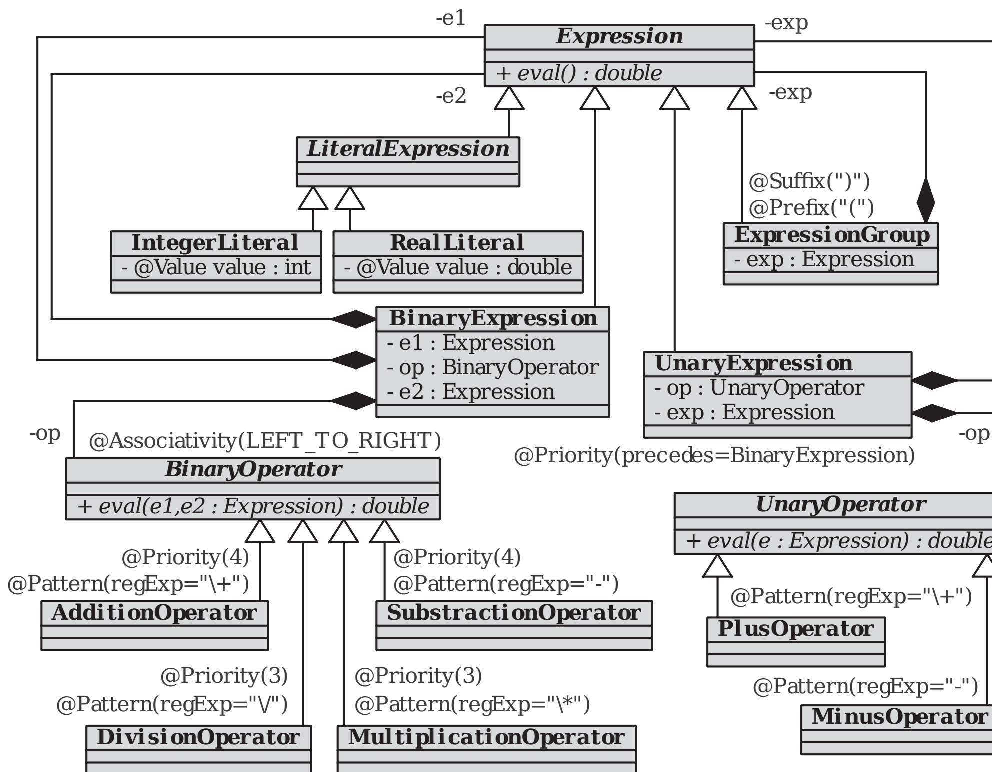


Figure 6: ModelCC specification of the arithmetic expression language.

LL and LR parsers might also be generated whenever the ASM-CSM mapping constraints make LL and LR parsing feasible. Whitespace and comments in the language can be defined by specifying what to ignore in the input text when the parser is created.

It should be noted that parse error handling is also completely dependant on the parser being used. Indeed, most parsers are able to provide comprehensive parsing error tracebacks.

However, ModelCC provides a testing framework that integrates well with existing IDEs and JUnit. Since separate language elements are models themselves, it is possible to implement both unitary and integration tests that focus on specific language elements. For example, assertions can check whether a model matches a certain string, whether a model does not match a certain string, or whether a model matches a string in a specific number of ways (e.g. matching without ambiguities). Assertions can, of course, take into consideration the contents or members of language elements for in-depth testing.

Since the abstract syntax model in ModelCC is not constrained by the vagaries of particular parsing algorithms, the language design process can be focused on its conceptual design, without having to introduce artifacts in the design just to satisfy the demands of particular tools:

- As we saw in the lex & yacc example, conventional tools, unless they are scannerless, force the creation of artificial token types in order to avoid lexical ambiguities, which leads to duplicate grammar production rules and semantic actions in the language specification. As in any other software development project, duplication hinders the evolution of languages and affects the maintainability of language processors. ModelCC, even though it is not scannerless, supports lexical ambiguities and each basic language element is defined as a separate and independent entity, even when their pattern specification are in conflict. Therefore, duplication in the language model does not have

```
public abstract class Expression implements IModel {
  public abstract double eval();
}

@Prefix("\\(") @Suffix("\\)")
public class ExpressionGroup extends Expression implements IModel {
  Expression e;
  @Override public double eval() { return e.eval(); }
}

public abstract class LiteralExpression extends Expression implements IModel {
}

public class UnaryExpression extends Expression implements IModel {
  UnaryOperator op;
  Expression e;
  @Override public double eval() { return op.eval(e); }
}

public class BinaryExpression extends Expression implements IModel {
  Expression e1;
  BinaryOperator op;
  Expression e2;
  @Override public double eval() { return op.eval(e1,e2); }
}

public class IntegerLiteral extends LiteralExpression implements IModel {
  @Value int value;
  @Override public double eval() { return (double)value; }
}

public class RealLiteral extends LiteralExpression implements IModel {
  @Value double value;
  @Override public double eval() { return value; }
}

public abstract class UnaryOperator implements IModel {
  public abstract double eval(Expression e);
}

@Associativity(AssociativityType.LEFT_TO_RIGHT)
public abstract class BinaryOperator implements IModel {
  public abstract double eval(Expression e1,Expression e2);
}
```

Figure 7: Complete Java implementation of the arithmetic expression interpreter using ModelCC (1/2): Java classes define the language ASM, metadata annotations specify the desired ASM-CSM mapping, and the `eval()` method implements arithmetic expression evaluation.

to be included to deal with lexical ambiguities: token type definitions do not have to be adjusted, duplicate syntactic constructs rules will not appear in the language model, and, as a consequence, semantic predicates do not have to be duplicated either.

- As we also saw both in the lex & yacc interpreter and in the ANTLR solution to the same problem, established parser generators require modifications to the language grammar specification in order to comply with parsing constraints, let it be the elimination of left-recursion for LL parsers or

```
@Priority(value=2) @Pattern(regExp="\\+")
public class AdditionOperator extends BinaryOperator {
  @Override public double eval(Expression e1,Expression e2) { return e1.eval()+e2.eval(); }
}

@Priority(value=2) @Pattern(regExp="-")
public class SubtractionOperator extends BinaryOperator {
  @Override public double eval(Expression e1,Expression e2) { return e1.eval()-e2.eval(); }
}

@Priority(value=1) @Pattern(regExp="\\*")
public class MultiplicationOperator extends BinaryOperator {
  @Override public double eval(Expression e1,Expression e2) { return e1.eval()*e2.eval(); }
}

@Priority(value=1) @Pattern(regExp="\\/")
public class DivisionOperator extends BinaryOperator {
  @Override public double eval(Expression e1,Expression e2) { return e1.eval()/e2.eval(); }
}

@Pattern(regExp="\\+")
public class PlusOperator extends UnaryOperator {
  @Override public double eval(Expression e) { return e.eval(); }
}

@Pattern(regExp="-")
public class MinusOperator extends UnaryOperator {
  @Override public double eval(Expression e) { return -e.eval(); }
}
```

Figure 8: Complete Java implementation of the arithmetic expression interpreter using ModelCC (2/2): Arithmetic operators.

```
// Read the model.
Model model = JavaModelReader.read(Expression.class);

// Generate the parser.
Parser<Expression> parser = ParserFactory.create(model);

// Parse the input string and instantiate the corresponding expression.
Expression expr = parser.parse("10/(2+3)*0.5+1");

// Evaluate the expression.
double value = expr.eval();
```

Figure 9: Code snippet showing how the arithmetic expression parser is generated and the resulting interpreter is invoked.

the introduction of new nonterminals to restructure the language specification so that the desired precedence relationships are fulfilled. In the model-driven language specification approach, the left-recursion problem disappears since it is something the underlying tool can easily deal with in a fully automated way when an abstract syntax model is converted into a concrete syntax model. Moreover, the declarative specification of constraints, such as the evaluation order constraints in Section 4, is orthogonal to the abstract syntax model that defines the language. Those constraints

determine the ASM-CSM mapping and, since ModelCC takes charge of everything in that conversion process, the language designer does not have to modify the abstract syntax model just because a given parser generator might prefer its input in a particular format. This is the main benefit that results from raising your abstraction level in model-based language specification.

- When changes in the language specification are necessary, as it is often the case when a software system is successful, the traditional language designer will have to propagate changes throughout the entire language processing tool chain, often introducing significant changes and making profound restructurings in the working code base. The changes can be time-consuming, quite tedious, and extremely error-prone. In contrast, modifications are more easily done when a model-driven language specification approach is followed. Any modifications in a language will affect either to the abstract syntax model, when new capabilities are incorporated into a language, or to the constraints that define the ASM-CSM mapping, whenever syntactic details change or new CSMs are devised for the same ASM. In either case, the more time-consuming, tedious, and error-prone modifications are automated by ModelCC, whereas the language designer can focus his efforts on the essential part of the required changes.
- Traditional parser generators typically mix semantic actions with the syntactic details of the language specification. This approach, which is justified when performance is the top concern, might lead to poorly-designed hard-to-test systems when not done with extreme care. Moreover, when different applications or tools employ the same language, any changes to the syntax of that language have to be replicated in all the applications and tools that use the language. The maintenance of several versions of the same language specification in parallel might also lead to severe maintenance problems. In contrast, the separation of concerns provided by ModelCC, as separate ASM and ASM-CSM mappings, promotes a more elegant design for language processing systems. By decoupling language specification from language processing and providing a conceptual model for the language, different applications and tools can now use the same language without having duplicate language specifications. A similar result could be hand-crafted using traditional parser generators (i.e. making their implicit conceptual model explicit and working on that explicit model), but ModelCC automates this part of the process.

In summary, while traditional language processing tools provide different mechanisms for resolving ambiguities and implementing language constraints, the solutions they provide typically interfere with the conceptual modeling of languages: relatively minor syntactic details might significantly affect the structure of the whole language specification. Model-driven language specification, as exemplified by ModelCC, provides a cleaner separation of concerns: the abstract syntax model is kept separate from its incarnation in concrete syntax models, thereby separating the specification of abstractions in the ASM from the particularities of their textual representation in CSMs.

# 6 More Complex Examples

In this section, we include the model of a full-fledged imperative programming language that illustrates language composition and reference resolution in ModelCC. The UML class diagrams in Figure 11 presents our annotated imperative programming language, which is complemented by the arithmetic expression language in Figure 6 and extended with new binary operators defined as *BinaryOperator* subclasses. This example illustrates ModelCC capabilities for language composition: the simple arithmetic expression language described in Section 5 is not only used within the imperative programming language, but it is also extended with new expression types and binary operators.

```
// Read the model.
Model model = JavaModelReader.read(Expression.class);

// Create the parser.
Parser<Expression> parser = ParserFactory.create(model);

// Define a constant
parser.add(new Constant("pi", 3.1415927));

// Use the predefined constant in JUnit tests for arithmetic expressions
assertEquals(3.1415927,   parser.parse("pi").eval(),   EPSILON);
assertEquals(2*3.1415927, parser.parse("2*pi").eval(), EPSILON);
```

Figure 10: Code snippet showing ModelCC support for separate compilation using predefined model elements.

ModelCC is able to automatically generate a grammar from the ASM defined by the class model and the ASM-CSM mapping, which is specified as a set of metadata annotations on the class model. These annotations also provide a mechanism for reference resolution that allows the automatic instantiation of complete object graphs. References are automatically resolved by ModelCC, resulting in abstract syntax graphs rather than mere abstract syntax trees. In our imperative language example, variables are automatically connected to the expressions and assignment statements where they appear. Likewise, function calls are automatically linked to the corresponding function definitions, without further intervention by the programmer.

Another interesting application of the reference resolution mechanism in ModelCC is illustrated by the code snippet in Figure 10. In this exam, a constant is predefined before the parser is invoked to parse an expression that includes a reference to the predefined constant, whose definition does not have to be included in the textual input of the parser, thus providing a crude but elegant form of separate compilation.

A fully-functional version of ModelCC for Java, additional examples of its use, and a detailed user manual describing all the annotations that can be used to annotate class models in ModelCC can be found at the ModelCC web site: `http://www.modelcc.org`.

# 7 Conclusions and Future Work

In this paper, we have introduced ModelCC, a model-based tool for language specification. ModelCC lets language designers create explicit models of the concepts a language represents, i.e. the abstract syntax model (ASM) of the language. Then, that abstract syntax can be represented in textual or graphical form, using the concrete syntax defined by a concrete syntax model (CSM). ModelCC automates the ASM-CSM mapping by means of metadata annotations on the ASM, which let ModelCC act as a model-based parser generator.

ModelCC is not bound to particular scanning and parsing techniques, so that language designers do not have to tweak their models to comply with the constraints imposed by particular parsing algorithms. ModelCC abstracts away many details traditional language processing tools have to deal with. It cleanly separates language specification from language processing. Given the proper ASM-CSM mapping definition, ModelCC-generated parsers are able to automatically instantiate the ASM given an input string representing the ASM in a concrete syntax.

Apart from being able to deal with ambiguous languages, ModelCC also allows the declarative resolution of any language ambiguities by means of constraints defined over the ASM. The current version of ModelCC also supports lexical ambiguities and custom pattern matching classes.

ModelCC also incorporates reference resolution within the parsing process. Instead of returning abstract syntax trees, ModelCC is able to obtain abstract syntax graphs from its input string. Such abstract syntax graphs are not restricted to directed acyclic graphs, since ModelCC supports the resolution of anaphoric, cataphoric, and recursive references.

The proposed model-driven language specification approach promotes the domain-driven design of language processors. Its model-driven philosophy supports language evolution by improving the maintainability of languages processing system. It also facilitates the reuse of language specifications across product lines and different applications, eliminating the duplication required by conventional tools and improving the modularity of the resulting systems.

In the future, we plan to further study the possibilities tools such as ModelCC open up in different application domains, including traditional language processing systems (compilers and interpreters) [3], domain-specific languages and language workbenches [11], model-driven software development tools [27, 12], natural language processing [15], text mining applications [2], data integration [8], and information extraction [26].

## Acknowledgements

Work partially supported by research project TIN2012-36951, "NOESIS: Network-Oriented Exploration, Simulation, and Induction System", funded by the Spanish Ministry of Economy and the European Regional Development Fund (FEDER).

## References

[1] Harold Abelson & Gerald J. Sussman (1996): *Structure and Interpretation of Computer Programs*, 2nd edition. MIT Press.

[2] Charu C. Aggarwal & ChengXiang Zhai, editors (2012): *Mining Text Data*. Springer.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi & Jeffrey D. Ullman (2006): *Compilers: Principles, Techniques, and Tools*, 2nd edition. Addison Wesley.

[4] Fernando Berzal, Juan-Carlos Cubero, Nicolás Marín & María-Amparo Vila (2005): *Lazy Types: Automating Dynamic Strategy Selection*. *IEEE Software* 22(5), pp. 98–106.

[5] Elizabeth Bjarnason (1996): *APPLAB – A Laboratory for Application Languages*. In: *Proceedings of the 7th Nordic Workshop on Programming Environment Research*, pp. 99–104.

[6] Patrick Borras, Dominique Clement, Th. Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang & V. Pascual (1988): *CENTAUR: the system*. In: *Proceedings of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 14–24.

[7] Christoff Bürger, Sven Karol, Christian Wende & Uwe Aßman (2010): *Reference Attributed Grammars for metamodel semantics*. In: *Proceedings of the 3rd International Conference on Software Language Engineering*, pp. 22–41.

[8] AnHai Doan, Alon Halevy & Zachary Ives (2012): *Principles of Data Integration*. Elsevier Science.

[9] Eric Evans (2003): *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.

[10] Martin Fowler (2002): *Using Metadata*. *IEEE Software* 19(6), pp. 13–17.

[11] Martin Fowler (2005): *Language Workbenches: The Killer-App for Domain Specific Languages?* Http://martinfowler.com/articles/languageWorkbench.html.

[12] Julián Garrido, M. Ángeles Martos & Fernando Berzal (2007): *Model-driven development using standard tools*. In: Proceedings of the 9th International Conference on Enterprise Information Systems, IDEAL 2007 DISI, pp. 433–436.

[13] John Grundy, John Hosking, Karen Na Li, Norhayati Mohd Ali, Jun Huh & Richard Lei Lu (2013): *Generating Domain-Specific Visual Language Tools from Abstract Visual Specifications*. IEEE Transactions on Software Engineering 39, pp. 487–515.

[14] Görel Hedin & Boris Magnusson (1988): *The Mjølner Environment: Direct Interaction with Abstractions*. In: Proceedings of the 2nd European Conference on Object-Oriented Programming, Lecture Notes in Computer Science 322, pp. 41–54.

[15] Daniel Jurafsky & James H. Martin (2009): *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*, 2nd edition. Prentice Hall.

[16] Lennart C. L. Kats, Eelco Visser & Guido Wachsmuth (2010): *Pure and declarative syntax definition: Paradise lost and regained*. In: Proceedings of the ACM International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'10), pp. 918–932.

[17] Anneke Kleppe (2007): *Towards the Generation of a Text-Based IDE from a Language Metamodel*. In: Proceedings of the 4th European Conference on Model-Driven Architecture - Foundations and Applications, Lecture Notes in Computer Science 4530, pp. 114–129.

[18] Rick Mugridge & Ward Cunningham (2005): *Fit for Developing Software: Framework for Integrated Tests (Robert C. Martin)*. Prentice Hall PTR.

[19] Luis Quesada, Fernando Berzal & Francisco J. Cortijo (2011): *Lamb — A Lexical Analyzer with Ambiguity Support*. In: Proceedings of the 6th International Conference on Software and Data Technologies, 1, pp. 297–300, doi:10.5220/0003476802970300.

[20] Luis Quesada, Fernando Berzal & Francisco J. Cortijo (2012): *Fence — A Context-Free Grammar Parser with Constraints for Model-Driven Language Specification*. In: Proceedings of the 7th International Conference on Software Paradigm Trends, pp. 5–13, doi:10.5220/0003949800050013.

[21] Luis Quesada, Fernando Berzal & Juan-Carlos Cubero (2011): *A Language Specification Tool for Model-Based Parsing*. In: Proceedings of the 12th International Conference on Intelligent Data Engineering and Automated Learning. Lecture Notes in Computer Science, 6936, pp. 50–57, doi:10.1007/978-3-642-23878-9_7.

[22] Luis Quesada, Fernando Berzal & Juan-Carlos Cubero (2014): *A Domain-Specific Language for Abstract Syntax Model to Concrete Syntax Model Mappings*. In: Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development, pp. 158–165.

[23] Luis Quesada, Fernando Berzal & Juan-Carlos Cubero (2014): *ModelCC — A Pragmatic Parser Generator*. International Journal of Software Engineering and Knowledge. (accepted for publication).

[24] Luis Quesada, Fernando Berzal & Juan-Carlos Cubero (2014): *Parsing Abstract Syntax Graphs with ModelCC*. In: Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development, pp. 151–157.

[25] Steven P. Reiss (1985): *PECAN: Program Development Systems that Support Multiple Views*. IEEE Transactions on Software Engineering SE-11(3), pp. 276–285.

[26] Sunita Sarawagi (2008): *Information Extraction*. Foundations and Trends in Databases 1(3), pp. 261–377. Available at `http://dx.doi.org/10.1561/1900000003`.

[27] Douglas C. Schmidt (2006): *Model-Driven Engineering*. IEEE Computer 39(2), pp. 25–31.

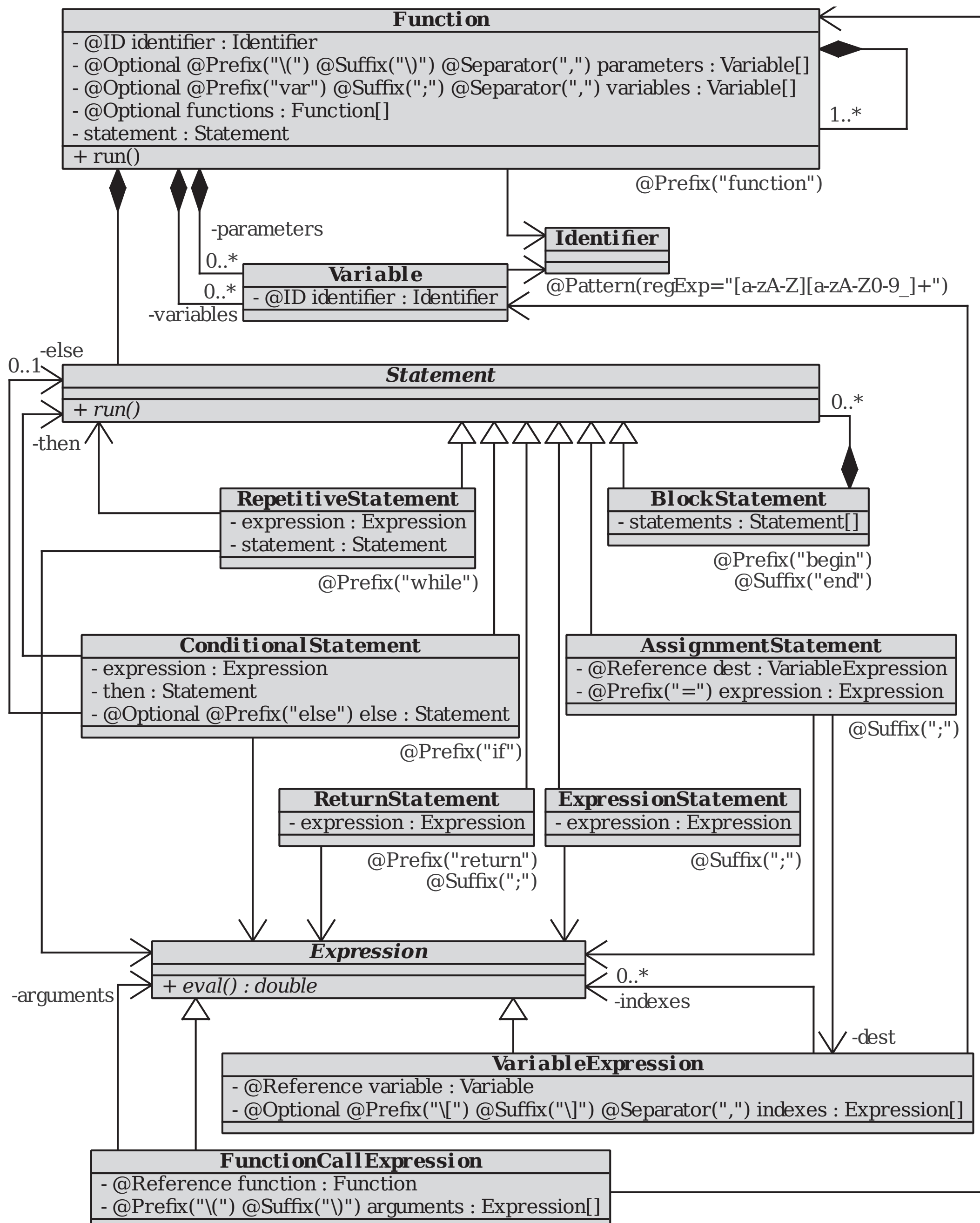


Figure 11: ModelCC specification of an imperative programming language. ModelCC reference resolution support is used to allow the declaration of variables and functions. ModelCC language composition support is used to include *Expressions*, which were defined as a separate language in Figure 6.