International Journal of Software Engineering and Knowledge Engineering
© World Scientific Publishing Company

ModelCC — A Pragmatic Parser Generator

Luis Quesada, Fernando Berzal, and Juan-Carlos Cubero
Department of Computer Science and Artificial Intelligence,
CITIC, University of Granada, Granada 18071, Spain
{*lquesada,fberzal,jc.cubero*}*@decsai.ugr.es*

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

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. Whenever the input text format is modified, the grammar has to be updated and the subsequent changes propagate throughout the entire language processing tool chain. Moreover, if several applications use the same language, multiple copies of the same language specification have to be maintained in sync, since language specification (i.e. the grammar) is tightly coupled to language processing (i.e. the semantic actions that annotate that grammar). In this paper, we introduce ModelCC, a model-based parser generator that decouples language specification from language processing, hence avoiding the aforementioned problems that are 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 a given textual representation format, so that the generated parser fully automates the instantiation of the desired conceptual model. ModelCC is also a powerful tool for the design of domain-specific languages.

Keywords: Model-based parser generators, language workbenches, model-driven software development.

1. Introduction

Well-designed software systems are typically built around a conceptual model of the domain they address. However, they also have to get their data from different data sources. When data comes from a relational database, object-relational mapping tools are employed. When data comes in as text, software engineers resort to syntax-driven tools in order to populate the conceptual model with data from the textual data sources.

In general, you can define an algebra of generic operators that could be applied to schemas and mappings across a broad range of applications [4]. These operators are known as model management operators, where model is a generic term that can refer to a database schema, a set of classes in an object-oriented design, or an ontology. Model management operators take schemas and/or mappings as input, and they also return schemas or mappings. Object-relational mapping tools (ORM) are just a particular incarnation of model management tools. Such tools, designed to manage multiple models and the translations among them, can be the foundation for integrated CASE environments [2].

Translations between models are ubiquitous where you must handle data from a variety of sources. However, existing model management tools typically require data to be structured, e.g. for translation between XML and relational models or between the object model of a programming languages (such as Java or C#) and a relational database schema. Dealing with data in unstructured form (i.e. text) poses its own challenges and typically involves the development of parsers, often with the help of a parser generator.

Building a parser requires the specification of an ideally context-free grammar describing the language to be parsed. This grammar, which has to conform to the requirements of the particular parsing algorithm, is annotated with semantic actions [1]. Semantic actions are responsible for the desired functionality of the parser, e.g. instantiating the desired conceptual model.

The proper specification of a formal language description by means of an annotated grammar is a nontrivial process that heavily depends on the peculiarities of the lexical and syntactic analysis techniques to be used, given that every particular technique requires the grammar to comply with particular constraints. Existing parsing algorithms are suitable for specific classes of languages, such as LL, LALR, or LR languages. Once a suitable parsing algorithm is chosen for the language to be parsed, the language designer must adapt its language specification to the corresponding grammar form, i.e. an LL, LALR, or LR grammar.

In practice, the formal description of the language in fed into a parser generator in order to create a parser for the language. However, traditional syntax-driven tools also require a time-consuming, tedious, and error-prone maintenance process. Whenever the language has to be updated, the language designer has to manually propagate changes throughout the entire language processing tool chain [10], from the specification of the grammar defining the formal language (and its adaptation to specific parsing tools) to the semantic actions that annotate such grammar, which might need substantial rework even for small language changes.

Moreover, multiple copies of the same language specification must be maintained in sync when different applications use the same language, such as compilers, editors, debuggers, validators, and style checkers. This is a direct consequence of the fact that the formal grammar used to describe the language is tightly coupled to the functionality of each particular application, which is implemented through the use of semantic actions that annotate the grammar. In other words, language specification is tightly coupled to language processing.

In this paper, we introduce ModelCC, a model-based parser generator that decouples language specification from language processing, hence avoiding the aforementioned problems caused by grammar-driven parser generators. ModelCC follows

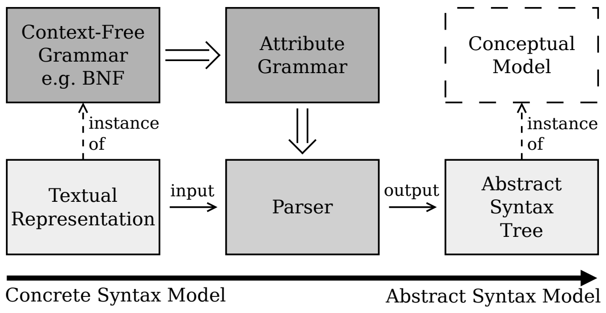

Fig. 1. Traditional language processing.

the model-based language specification approach [17]: it receives a conceptual model as input, along with constraints that annotate it, and creates a parser for the desired language that populates the conceptual model from a textual input. The generated parser fully automates the instantiation of the conceptual model, which can then be shared by different applications. Therefore, ModelCC supports domain-driven design [6], a software development approach that connects software development to an evolving model by placing the primary focus on the core domain logic and using a conceptual model of the domain to support complex designs.

From the ModelCC model-based perspective, we view the conceptual model of the domain as an abstract syntax model (ASM) of the language that represents its structure without the superfluous details related to a particular textual representation of the model instances [11]. We can then describe how ASM instances can be mapped to the concrete syntax model (CSM) of their textual representation by specifying constraints for the elements in the ASM [11].

ModelCC decouples language processing from the language model, the parsing algorithm from the underlying semantic model [7]. The abstract syntax model of the language can be modified as needed, without having to worry about the peculiarities of particular parsing techniques, since the corresponding parser will be automatically updated given the ASM and its mapping to the CSM. A proper use of sound software design principles allows such separation of concerns: formal grammar and semantic actions are no longer entwined.

Figure 1 illustrates the traditional language processing approach, whereas the model-based approach proposed in this paper is shown in Figure 2.

In the traditional language processing approach, the language designer starts by designing the grammar of the desired language, typically in BNF or a similar format. Then, the designer annotates the grammar with attributes and semantic actions. The resulting attribute grammar is then fed into lexer and parser generators that create the corresponding lexers and parsers. Syntax-directed translation tools automate the process of creating abstract syntax trees from a textual representation in the concrete syntax of the language.

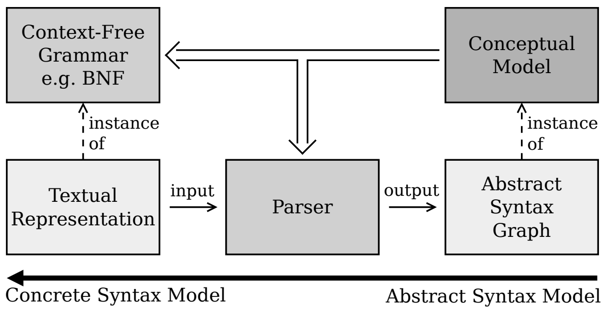

Fig. 2. Model-based language processing.

In the model-driven language processing approach, the language designer starts by designing the conceptual model that represents the abstract syntax of the desired language, focusing on the elements of the domain the language will represent and their relationships. Instead of dealing with the syntactic details of the language from the start, the designer devises its abstract syntax model. Then, the language designer will specify the details of the ASM-CSM mapping by imposing constraints on the abstract syntax model and these constraints will help automate the conversion from the ASM to a particular CSM representation.

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 instantiates 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 automate the creation of parsers for the concrete language 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 approach.

There is another key difference between the grammar-driven and the model-driven approaches to language specification. In practice, while the result of the parsing process is an abstract syntax tree that corresponds to a valid parsing of the input text according to the concrete language syntax, nothing prevents the conceptual model designer from modeling non-tree structures, which can be described, for instance, by reference attribute grammars [3]. Hence the use of the ‘abstract syntax graph’ term in Figure 2. In fact, ModelCC incorporates reference resolution and is able to obtain abstract syntax graphs from its input string [19, 21].

2. A Simple Example

Let us start by considering a classical example: the language of a simple calculator that evaluates arithmetic expressions such as “6*4-5/(2+3)”. Figure 3 shows, as

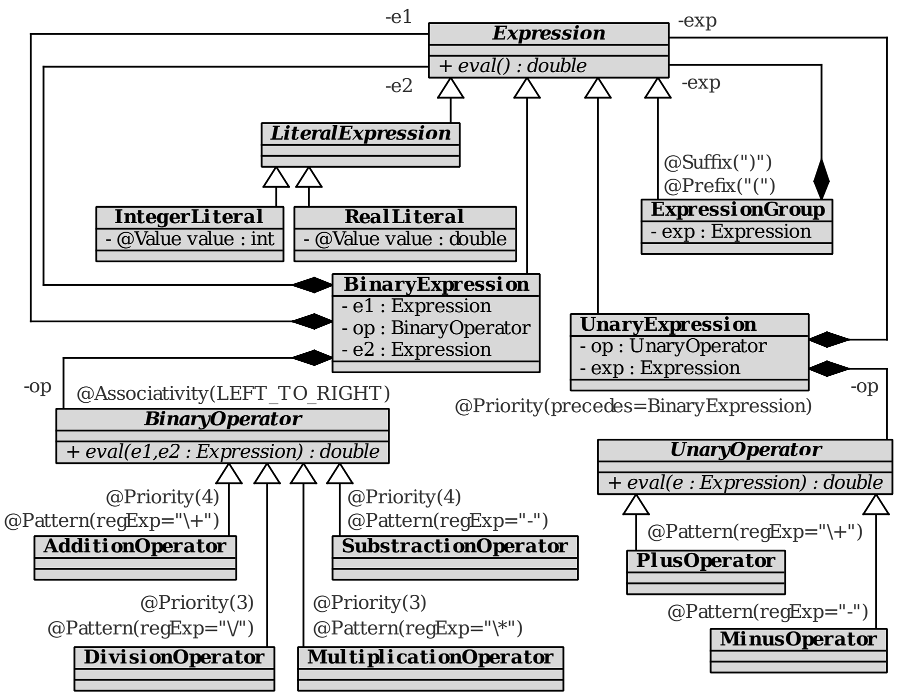

Fig. 3. ModelCC specification of a simple calculator.

an UML class diagram, the conceptual model that represents the abstract syntax model of the desired language. Java-like metadata annotations are also specified to define the desired ASM-CSM mapping.

Our conceptual model includes four kinds of expressions: literal expressions, which comprise both integer and real literals; expression groups, which are parenthesized; binary expressions, which consist of two expressions separated by a binary operator (either +, -, *, or /); and unary expressions, which consist of a single expression preceded by an unary operator (either + or -).

Figure 4 shows the fragment of the traditional lex/yacc [14] implementation for binary expressions. Even this ten-line code snippet illustrates some of the drawbacks of traditional syntax-driven language processing techniques:

- Limitations on the scanning and parsing techniques impose constraints on the language specification. For instance, tokens with different semantics, such as the minus unary operator and the subtraction binary operator, have to be aggregated into a single token in order to avoid lexical ambiguities. This, in turn, forces a verbose implementation of the binary expression interpreter and leads to code duplication.
- Semantic actions are tightly coupled to the language grammar. Changes in the language specification have to be propagated to the semantic actions,

```
Expression : Expression UnaryOrPriority2BinaryOperator Expression
             {
               if ($2.operator == PLUSORADDITION)
                 $$.value = $1.value+$3.value;
               else /*$2.operator == MINUSORSUBTRACTION*/
                 $$.value = $1.value-$3.value;
             }
           | Expression Priority1BinaryOperator Expression
             {
               if ($2.operator == MULTIPLICATION)
                 $$.value = $1.value*$3.value;
               else /*$2.operator == DIVISION*/
                 $$.value = $1.value/$3.value;
             }
```

Fig. 4. Implementation of a binary expression interpreter using lex/yacc.

often requiring a substantial amount of work.

- Semantic actions entwined with the language grammar lead to a poor separation of concerns. System functionality is often located in the wrong place from a conceptual point of view, such as the operator evaluation implemented within the binary expression grammar.

In contrast, Figure 5 shows how the binary expression fragment of the ModelCC specification is implemented in Java. Even this simple example illustrates some of the advantages of ModelCC:

- Constraints are not imposed on the abstract syntax model of the language. Tokens with different semantics, even when they share the same syntax, can be independently implemented, such as the minus unary operator and the subtraction binary operator. This avoids the unnecessary code duplication that appears in the traditional implementation.
- Semantic actions are decoupled from the language syntax. Changes in the language syntax do not have to be propagated to the rest of the system by hand. ModelCC seamlessly supports an evolving abstract syntax model.
- ModelCC also supports a better separation of concerns. Since semantic actions are no longer tied to grammar production rules, a well-designed domain model allows the proper assignment of responsibilities to the different system modules. For instance, operator evaluation exploits polymorphism and the addition of new kinds of expressions and operators to the language is straightforward.

The parser that ModelCC generates from the simple calculator ASM can parse input strings such as "10/(2+3)*0.5+1". It automatically instantiates *Expression* objects given a textual input, which can be evaluated by the *eval* method to yield their associated value (2 in the previous example). Figure 6 shows the actual code

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

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

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

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

Fig. 5. Implementation of a binary expression interpreter using ModelCC in the Java programming language.

```
// Read the model.
Model model = JavaModelReader.read(Expression.class);
// Generate the parser.
Parser<Expression> parser = ParserFactory.create(model);
// Parse a given string and instantiate the corresponding expression.
Expression expr = parser.parse("10/(2+3)*0.5+1");
// Evaluate the expression.
double value = expr.eval();
```

Fig. 6. A code snippet showing how the ModelCC arithmetic expression parser is created and invoked from Java.

snippet needed to create and invoke the parser in ModelCC using the Java programming language.

3. Model-Based Language Specification

In this section, we proceed to briefly describe the model-driven approach that ModelCC employs for the design of abstract syntax models (ASMs) as well as the mechanisms used to define abstract syntax to concrete syntax model mappings

(ASM-CSM mappings).

3.1. *Abstract Syntax Model Specification*

The abstract syntax model of a language corresponds to the domain model in domain-driven design or the conceptual model in database design. The specification of the ASM in ModelCC starts with the definition of basic language elements, which are the simplest elements that have meaning in the domain of the language. Basic language elements can be modeled as simple classes in object-oriented programming languages. They correspond to value objects in domain-driven design and attributes in an entity-relationship model.

Once the basic language elements are defined, the abstract syntax model is completed by establishing relationships among language elements, which help us define entities and aggregates in domain-driven and database design terms:

- **Concatenation** is the most basic construct we can use to combine sets of language elements into more complex language elements. In ModelCC, concatenation is achieved by composition. The resulting language element is the composite element and its members are the language elements the composite element collates.
- **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, mere concatenation does not suffice. In ModelCC, repetition is also achieved through object composition, just by allowing the desired multiplicities in the associations that connect composite elements to their constituent elements.
- **Selection** is the final key language modeling construct in ModelCC. Selection lets us represent choices, i.e. alternative elements in language constructs. In ModelCC, selection is achieved by derivation (i.e. subtyping). The language element we want to establish alternatives for is represented by a superclass in an object-oriented model, whereas the different alternatives are represented as subclasses.

It should be noted that the abstract syntax model is used as a conceptual model of the language domain. It is not bound to a particular scanning or parsing algorithm and, therefore, it is not necessary to consider technique-specific constraints while designing the language ASM. Moreover, ASMs are not constrained by tree-like syntax derivations and ModelCC is able to parse arbitrary abstract syntax graphs rather than plain abstract syntax trees.

In this paper, UML class diagrams are employed to illustrate abstract syntax models. Graphical modeling notations are useful for discussing design alternatives and suitable for understanding the overall system design. However, a textual notation using conventional object-oriented programming languages, such as C++, C#, or Java, is often preferable in practice. Creating models in a declarative, textual notation offers a number of advantages [23]. First of all, the model composition mechanism matches well both a programmer's textual abstract formalization of concrete concepts and the manipulation of text using editors and other text-based tools. The declarative representation is highly malleable, since the existing visual structure does not hinder drastic changes, nor is effort wasted on the tidy arrangement of graph nodes a psychological barrier against design refactoring [23]. Declarative models are also highly automatable: they can be easily generated from even higher-level descriptions (e.g. UML diagrams), they can be put under configuration management using the same tools that are used for managing source code, and they can readily exploit existing text processing tools for tasks that a drawing editor may not provide [23].

3.2. *Abstract Syntax Model to Concrete Syntax Model Mapping*

Once the ASM is complete, constraints can be imposed on the abstract syntax model in order to establish the desired ASM-CSM mapping. ModelCC supports the following kinds of constraints:

- Pattern specification constraints are employed to specify the textual representation of basic language elements in the CSM. They define the token types recognized by the lexer and indicate where the recognized tokens will be stored in the ASM.
- Delimiter constraints determine the prefixes, suffixes, and separators that will be used to mark the boundaries of language elements in the CSM. They can be used for eliminating language ambiguities or just as syntactic sugar in text-based CSMs.
- Cardinality constraints restrict the multiplicity of elements in the ASM and determine the optionality of language elements. The CSM must consider such constraints for defining the grammar of the language recognized by the generated parser.
- Evaluation order constraints allow the explicit resolution of different kinds of lexical and syntactic ambiguities that might appear in the ASM-CSM mapping (e.g. associativity, precedence, and ambiguity resolution in nested composites).
- Composition order constraints allow the specification of the constituent element orderings in composite language elements.
- Reference constraints allow the resolution of references between language elements (for parsing abstract syntax graphs).
- Finally, custom constraints allow the specification of special-purpose lexical, syntactic, and semantic constraints that the language designer can implement himself.

The aforementioned kinds of constraints are defined over ASM elements. When using the UML class diagram notation, these constraints annotate ASM classes and

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 kinds of constraints supported by the ModelCC model-based parser generator and the Java metadata annotations defined in its Java incarnation.

associations. When using a textual notation, such as object-oriented programming language, constraints can be specified with the help of metadata annotations, a feature supported by many modern programming languages. For example, the set of metadata annotations defined by our ModelCC implementation in Java are shown in Table 1. Such metadata annotations are enough for the ModelCC parser generator when applications deal with a single textual representation of their ASM. When applications must deal with multiple textual representations of the same ASM, multiple ASM-CSM mappings must be specified. ModelCC provides a domain-specific language for such ASM-CSM mappings [22].

Constraints on the ASM, therefore, define a particular ASM-CSM mapping. By using an annotated abstract syntax model, model-based tools such as ModelCC completely decouple language specification from language processing, which can then be performed using whichever parsing algorithm might be suitable for the formal language implicitly defined by the abstract syntax model and a particular ASM-CSM mapping.

4. ModelCC Internals

In this section, we describe the mechanisms that ModelCC employs for the translation of abstract syntax models (ASMs) and the constraints defined by the abstract syntax to concrete syntax model mappings (ASM-CSM mappings) into working parsers.

4.1. *From the ASM to the CSM*

Given a particular ASM and the desired ASM-CSM mapping, ModelCC first generates the specification of the language tokens for the lexer and then creates an annotated grammar that consists of production rules and semantic actions:

- **Patterns**: A formal token specification is derived from each basic language element in the ASM and its corresponding pattern definition in the ASM-CSM mapping. Patterns are defined in terms of regular expressions, as usual in lexical analysis tools, but can also be provided by custom pattern matchers, which can be developed to address specific requirements.
- **Delimiters**: Each delimiter in the ASM-CSM mapping (i.e. prefixes, suffixes, and separators) results in a token defined by a regular expression. These tokens are automatically included in the grammar production rules where the delimited language element is involved.
- **Composition**: Each composite element in the ASM yields a production rule in the context-free grammar describing the language. The resulting production rule has the nonterminal symbol of the composite element in its left-hand side and a concatenation of the nonterminal symbols corresponding to the composite element members in its right-hand side. Terminal symbols corresponding to delimiters are automatically inserted where appropriate. The order of the nonterminal symbols in the production rule right-hand side is, by default, given by the order in which they are specified in the object composition, but it can be modified by imposing constraints on the composition order when needed.
- **Repetition**: Each composite element including a repetitive construct in the ASM (i.e. lists of elements) will lead to additional recursive production rules in the grammar to allow for the appearance of lists of elements.
- **Selection**: Inheritance relationships in the abstract syntax model generate additional production rules in the language grammar. In these production rules, the nonterminal symbols corresponding to the superclasses appear in their left-hand side, while the nonterminal symbols of the subclasses appear as the only symbols in the production rule right-hand sides.
- **Multiplicity**: When language elements within a composite are optional, alternative grammar production rules that do not contain the optional elements are generated and appended to the language grammar. When the minimum multiplicity of a repeatable language element is 0, an additional epsilon production rule has to be appended to the grammar defining the textual CSM derived from the repetitive construct in the ASM. Every multiplicity constraint defined over repeating language elements generates a semantic action that checks whether the specific lower and upper bounds for the element multiplicity are satisfied.
- **Associativity**: Associativity constraints generate semantic actions for each production rule in the grammar where the nonterminal of the language el-

ement with associativity constraints is preceded and/or followed by the nonterminal that appears on the left-hand side of the production rule or any of the nonterminals representing the constrained language element superclasses in the ASM. For left-to-right associative language elements, the reduction of the production rule is inhibited whenever the element that follows the left-to-right associative element is generated by a reduction of the same production rule. For right-to-left associative language elements, the reduction of the production rule is inhibited whenever the element that precedes the right-to-left associative element is generated by a reduction of the same production rule. For non-associative language elements, the reduction of a production rule is inhibited whenever the element that precedes or follows the non-associative element is generated by a reduction of the same production rule.

- **Composite disambiguation**: Composition constraints, used to resolve shift/reduce conflicts, translate into the definition of precedences for tokens and grammar production rules. When composition is eager, shift operations will precede reduce operations. In contrast, when composition is lazy, reduce operations will have precedence over shift operations. Finally, when the composition must be explicit in the CSM, ambiguous grammar production rules will be removed and the use of delimiters determines whether shift or reduce operations are performed on a case by case basis.
- **Priority constraints**: Another disambiguation mechanism in ModelCC involves the use of precedence or priority constraints. Each priority constraint directly translates into the definition of precedences over tokens or precedences over grammar production rules, depending on whether the constrained language element is a basic language element or not.
- **Order constraints**: Composition order constraints let us define partial order relationships among the language elements within a composite language element. Such constraints are incorporated into the language grammar by the generation of a set of production rules that represent each and every valid ordering of the composite element members.
- **Reference resolution**: ModelCC includes a reference resolution mechanism that enables the possibility of parsing abstract syntax graphs. The use of references in the ASM-CSM mapping is automatically resolved by our implementation of ModelCC with the introduction of the production rules that characterize such references and the semantic actions that map them to the corresponding language elements. This reference resolution mechanism lets ModelCC build a full-fledged object graph from a linear text input, rather than the abstract syntax tree that would result from the use of traditional parser generators. Such graphs are not restricted to directed acyclic graphs, since ModelCC supports the resolution of anaphoric, cataphoric, and recursive references.

- **Custom constraints**: ModelCC also allows the definition of user-defined lexical, syntactic, and semantic constraints. Such custom constraints are treated as semantic actions by the parser generator and they provide a powerful open-ended mechanism that can support any additional feature we might want to incorporate into the parser generation process.

4.2. *ModelCC Scanning and Parsing Algorithms*

The ModelCC model-based language specification approach allows the specification of ambiguous languages. Hence, a general ModelCC-generated parser must be able to produce a compact parse forest comprising all the valid interpretations of the input data. In our current implementation of ModelCC, this flexibility is achieved by the use of Lamb lexers and Fence parsers.

Lamb [16] is a lexer with lexical ambiguity support that allows the specification of tokens not only by regular expressions, but also by arbitrary pattern matching algorithms. Lamb also supports token type precedences. When used with regular expressions, the efficiency of Lamb lexers is $O(n^2t^2)$, where n is the input string length and t is the number of different token types.

The Fence parser [18] is an optimized Earley-like algorithm that supports lexical and syntactic ambiguities, the specification of constraints, the definition of precedences among production rules, and the inhibition of production rule reductions by means of semantic actions. Earley parsers [5] use dynamic programming for parsing arbitrary context-free grammars in cubic time ($O(n^3)$), yet typically require quadratic time ($O(n^2)$) for unambiguous grammars and linear time ($O(n)$) for almost all LR(k) grammars.

It should be noted that, although the current version of ModelCC generates Fence parsers and Lamb lexers (as well as their probabilistic extensions), ModelCC could, in principle, be used with any known parsing algorithm. For unambiguous languages, LL(k), LALR, and GLR parsers might be used. The final use of a specific parsing algorithm is completely orthogonal to the conceptual model specification at the heart of the language specification approach we advocate for in ModelCC.

5. A More Complex Example

As a more complex example, we present a 3D object specification language in this section. The UML class diagram in Figure 7 represents the fully-annotated 3D object specification language model, enough for ModelCC to generate the language parser.

ModelCC reference resolution mechanism is used in the *Definition* and *Defined-Object* classes. The *name* member of the *Definition* class is annotated by *@ID*, which indicates that a *Definition* instance can be referred to by its *ObjectName*. Likewise, the *ref* member of a *DefinedObject* is annotated by *@Reference*, which means that a *DefinedObject* can refer to a *Definition* by its *ObjectName*. ModelCC automates reference resolution and makes the implementation of a symbol table unnecessary.

Finally, Figures 9 and 10 illustrate the specification and rendering of a particular 3D scene that includes a single object (namely, a stylized snail), which is referenced by its name. Figure 8 shows all the code that is actually needed to generate and invoke the parser for such a scene in our Java implementation of ModelCC.

This domain-specific language lets us compare the features of the model-driven approach provided by ModelCC and those of the traditional syntax-driven approach followed by conventional parser generators [9][12][15]:

- Using conventional language processing tools, the language designer has to choose the scanning and parsing algorithms to be used. Since each algorithm requires the language to conform to specific constraints, and not every parsing algorithm supports certain language features, the designer must have an intimate knowledge of the specific scanning and parsing techniques. In contrast, ModelCC decouples language processing from the language abstract syntax model. ModelCC lets designers focus on the conceptual design of their language using the domain-driven design approach.
- Using the traditional approach, the language designer must derive the complete formal grammar for the language. Moreover, the designer has to adapt its language grammar to comply with the constraints imposed by the particular parser generator to be used (e.g. no left-recursion, no right-recursion, or no ambiguities). However, ModelCC does not require tweaking the language specification to conform to the peculiarities of the chosen scanning and parsing algorithms. The translation process from the ASM to the CSM is completely automated with the help of the constraints that define the ASM-CSM mapping.
- When using the traditional approach, the conceptual model of the language is implicit. Only the language grammar is made explicit. In sharp contrast, when using ModelCC, the language designer develops an explicit conceptual model of the language (i.e. its abstract syntax model). Later, the designer can devise one or more ASM-CSM mappings to define the desired syntax for the particular textual representations of the conceptual model.
- Using the traditional approach to language design, the resulting parser produces parse trees that must be traversed to populate a conceptual model (either explicitly when the parse trees are fully built in memory or implicitly by means of semantic actions that are intertwined with the production rules describing the language grammar that is fed to the parser generator). This process, which can be tedious to implement and is error-prone, is fully automated by ModelCC, thus ensuring a better separation of concerns.
- Finally, ModelCC incorporates reference resolution into the model specification itself. Rather than handcrafting symbol tables for the definition of entities and the resolution of references to such entities, ModelCC manages

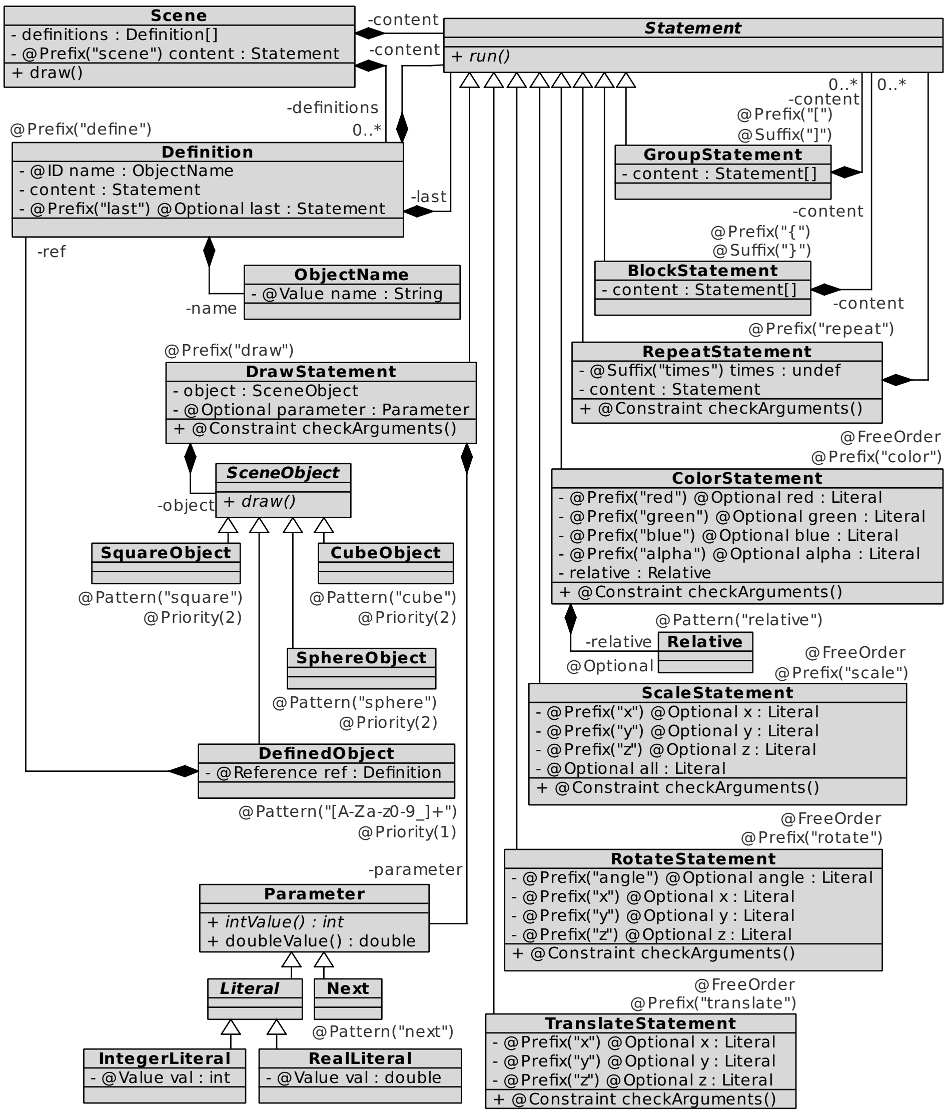

Fig. 7. ModelCC definition of a 3D object specification language. ModelCC reference resolution support is used to allow the specification of complex 3D objects in the *Definition* class.

```
// Read the model.
Model model = JavaModelReader.read(Scene.class);
// Generate the parser.
Parser<Scene> parser = ParserFactory.create(model);
// Parse a text file and instantiate the corresponding scene.
Scene scene = parser.parse(new FileInputStream("snail.txt"));
// Draw the scene.
scene.draw();
```

Fig. 8. A code snippet that shows how the 3D object specification language parser is generated and invoked.

```
define snail [
  color red 1 green 0.6 blue 0.2
  draw tail next
]
define tail [
  {
    scale 0.6
    draw cube
  }
  {
    color blue 0.3
    scale 0.1
    translate y 2.5
    repeat 11 times [
      translate y 1
      color relative blue +0.01 alpha -0.1
      draw cube
    ]
  }
  rotate angle 1 z 1
  translate x 0.3
  rotate z 1 angle 3
  scale 0.995
  color relative red -0.003 green -0.001 blue -0.001
  draw tail next
]
scene {
  draw snail 700
}
```

Fig. 9. The textual representation of a snail in our 3D object specification language.

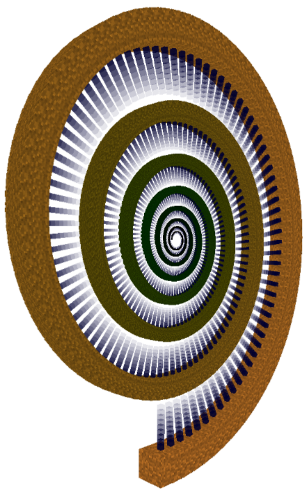

Fig. 10. Rendering of the snail from Figure 9.

such references seamlessly. The whole process is automated by ModelCC and it is completely transparent to the user, who directly obtains a full-fledged object graph as a result of the parsing process.

6. Conclusions

ModelCC is a model-based parser generator. ModelCC lets language designers create an explicit model of the concepts a language represents, i.e. the abstract syntax model of the language. Then, ModelCC allows the specification of abstract syntax model to concrete syntax model mappings by imposing constraints on abstract syntax model. Such constraints can be specified by metadata annotations directly on the source code of the ASM (in Java or .NET), as notes on UML class diagrams (as in OCL), or by using the ModelCC domain-specific language for ASM-CSM mappings (when multiple mappings are required).

ModelCC frees language designers from many of the burdens of traditional syntax-driven language processing tools. ModelCC allows designers to focus on the model domain rather than on the syntax specification, on what they really want to program rather than on how to perform the task [24]. Unlike conventional syntax-driven language workbenches [8], ModelCC does not requires the explicit specification of the the language grammar. ModelCC also avoids the time-consuming, tedious, and error-prone [13] updates associated to traditional parser generators, as the parser is fully derived from the model on the fly. ModelCC model-driven philosophy supports language evolution and improves the maintainability of languages processing systems.

Unlike conventional parser generators, ModelCC is not bound to specific parsing techniques and, therefore, it does not impose constraints on the design of the language abstract syntax model. In other words, language designers do not have to tweak their models to comply with the constraints imposed by particular parsing algorithms, since ModelCC abstracts away many of the details traditional language processing tools impose to their users. ModelCC provides a clean separation between 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 an instance of the ASM in the concrete syntax of the CSM. Since separate language elements are syntax models themselves, ModelCC can generate parsers for their associated grammars. This feature is useful for supporting unit testing (of specific language elements, whose parsers can be automatically generated and tested in isolation), but it is also key for language composition (by combining the models of independently designed languages). ModelCC facilitates the reuse of language models across product lines and applications, eliminating the duplication required by conventional tools and improving the modularity of the resulting systems.

Apart from being able to deal with inherently ambiguous languages, ModelCC allows disambiguation by means of constraints defined over the ASM. ModelCC also incorporates a reference resolution mechanism that yields full-fledged object graphs from a textual input, rather than the mere abstract syntax trees obtained by traditional parser generators. Using probabilistic extensions of the algorithms we have described in this paper, ModelCC is also useful for natural language processing [20]. In the future, ModelCC can serve as the foundation of a full-fledged model management system.

A working implementation of ModelCC in Java is available online at http://www.modelcc.org.

Acknowledgments

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] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*, Addison Wesley, 2nd edition, 2006.

[2] P. Atzeni and R. Torlone, Management of multiple models in an extensible database design tool, in *Proceedings of the 5th International Conference on Extending Database Technology*, 1996, pp. 77–95.

[3] C. Bürger, S. Karol, C. Wende, and Uwe Aßman, Reference attributed grammars for metamodel semantics, in *Proceedings of the 3rd International Conference on Software Language Engineering*, 2010, pp. 22–41.

[4] A. Doan, A. Halevy, and Z. Ives, *Principles of Data Integration*, Morgan Kaufmann, 1st edition, 2012.

[5] J. Earley, An efficient context-free parsing algorithm, in *Communications of the ACM* **13**(2), 1970, pp. 94–102.

[6] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 1st edition, 2004.

[7] M. Fowler, A pedagogical framework for domain-specific languages, in *IEEE Software* **26**(4), 2009, pp. 13–14.

[8] A. Hen-Tov, D. H. Lorenz, A. Pinhasi, and L. Schachter, ModelTalk: when everything is a domain-specific language, in *IEEE Software* **26**(4), 2009, pp. 39–46.

[9] Stephen C. Johnson, YACC: Yet Another Compiler Compiler, as *Computing Science Technical Report* 32, AT&T Bell Laboratories, 1979.

[10] L. C. L. Kats, E. Visser, and G. Wachsmuth, Pure and declarative syntax definition: Paradise lost and regained, in *Proceedings of the 2010 ACM International Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2010, pp. 918–932.

[11] A. Kleppe, 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**, 2007, pp. 114–129.

[12] V. Kodaganallur, Incorporating language processing into Java applications: a JavaCC tutorial, in *IEEE Software* **21**(4), 2004, pp. 70–77.

[13] R. Lämmel and C. Verhoef, Cracking the 500-language problem, in *IEEE Software* **18**(6), 2001, pp. 78–88.

[14] J. R. Levine, T. Mason, and D. Brown, *lex&yacc*, O'Reilly, 2nd edition, 1992.

[15] T.J. Parr and Russell W. Quong, ANTLR: A Predicated-LL(k) Parser Generator, in *Software Practice and Experience*, **25**(7), 1995, pp. 789–810.

[16] L. Quesada, F. Berzal, and F.J. Cortijo, Lamb — a lexical analyzer with ambiguity support, in *Proceedings of the 6th International Conference on Software and Data Technologies* **1**, 2011, pp. 297–300.

[17] L. Quesada, F. Berzal, and J.C. Cubero, 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**, 2011, pp. 50–57.

[18] L. Quesada, F. Berzal, and F.J. Cortijo, Fence — a context-free grammar parser with constraints for model-driven language specification, in *Proceedings of the 7th International Conference on Software Paradigm Trends* **1**, 2012, pp. 5–13.

[19] L. Quesada, F. Berzal, and J.C. Cubero, A model-driven parser generator with reference resolution support, in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 394–397.

[20] L. Quesada, F. Berzal, and J.C. Cubero, A model-based multilingual language parser — Implementing Chomsky's X-bar theory in ModelCC, in *Proceedings of the 10th International Conference on Flexible Query Answering Systems. Lecture Notes in Artificial Intelligence* **8132**, 2013, pp. 293–304.

[21] L. Quesada, F. Berzal, and J.C. Cubero, Parsing abstract syntax graphs with ModelCC, in *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development*, 2014, in press.

[22] L. Quesada, F. Berzal, and J.C. Cubero, 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*, 2014, in press.

[23] D. Spinellis, On the declarative specification of models, in *IEEE Software* **20**(2), 2013, pp. 94–96.

[24] D. Spinellis, The importance of being declarative, in *IEEE Software* **30**(1), 2013, pp. 90–91.