Parsing Abstract Syntax Graphs with ModelCC

Luis Quesada, Fernando Berzal and Juan-Carlos Cubero
Department of Computer Science and Artificial Intelligence
University of Granada, CITIC, 18071, Granada, Spain

Keywords: Model-driven Software Development, Parser Generators, Abstract Syntax Graphs.

Abstract: The tight coupling between language design and language processing in traditional language processing tools is avoided by model-based parser generators such as ModelCC. By decoupling language specification from language processing, ModelCC avoids the limitations imposed by traditional parser generators, which constrain language designers to specific kinds of grammars. Apart from providing an alternative approach to language specification, ModelCC incorporates reference resolution within the parsing process. Instead of returning mere abstract syntax trees, ModelCC is able to obtain abstract syntax graphs from its input string. Moreover, such abstract syntax graphs are not restricted to directed acyclic graphs, since ModelCC supports anaphoric, cataphoric, and recursive references.

1 INTRODUCTION

A formal language represents a set of strings (Jurafsky and Martin, 2009). Formal languages consist of an alphabet, which describes the basic symbol or character set of the language, and a grammar, which describes how to write valid sentences of the language (Ginsburg, 1975; Harrison, 1978). In Computer Science, formal languages are used, among other things, for the precise definition of data formats and the syntax of programming languages.

Most existing language specification techniques (Aho et al., 2006) require the language designer to provide a textual specification of the language grammar. The proper specification of such a grammar is a nontrivial process that depends on the lexical and syntax analysis techniques to be used, since each kind of technique requires the grammar to comply with a specific set of 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 to perform an efficient analysis, and some techniques' inability to resolve conflicts caused by grammar ambiguities.

As an alternative approach, model-based language specification techniques (Kleppe, 2007) decouple language design from language processing and automatically generate the corresponding language grammar, thus making the language design process less arduous.

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.

Typically, syntax analysis defers some analysis tasks to later stages in the language processing pipeline, such as reference resolution and other semantic checks. However, a model-driven parser generator can be employed to automate some parts of this process.

ModelCC (Quesada et al., 2011) is a model-based parser generator that includes support for dealing with references between language elements, thus incorporating the reference resolution that is traditionally hand-crafted with the help of a symbol table into the parsing process.

In this paper, we explain how ModelCC (Quesada et al., 2011) 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.

Section 2 introduces model-based language specification. Section 3 explains the reference resolution support in the ModelCC model-based parser generator. Section 4 introduces a working example that illustrates abstract syntax graph parsing. Finally, Section

Quesada L., Berzal F. and Cubero J..
Parsing Abstract Syntax Graphs with ModelCC.
DOI: 10.5220/0004671601510157
In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2014), pages 151-157
ISBN: 978-989-758-007-9

5 presents our conclusions and future work.

2 BACKGROUND

In its most general sense, a model is anything used in any way to represent something else. In such sense, a grammar is a model of the language it defines. In Software Engineering, data models are also common. Data models explicitly determine the structure of data. Roughly speaking, they describe the elements they represent and the relationships existing among them. From a formal point of view, it should be noted that data models and grammar-based language specifications are not equivalent, even though both of them can be used to represent data structures. A data model can express relationships a grammar-based language specification cannot, and does not need to comply with the constraints a grammar-based language specification has to comply with. Typically, describing a data model is generally easier than describing the corresponding grammar-based language specification.

In practice, when we want to build a complex data structure from the contents of a file, the implementation of the mandatory language processor needed to parse the file requires the software engineer to build a grammar-based language specification for the data as represented in the file and also to implement the conversion from the parse tree returned by the parser to the desired data structure, which is an instance of the data model that describes the data in the file.

Whenever the language specification has to be modified, the language designer has to manually propagate changes throughout the entire language processor tool chain, from the specification of the grammar defining the formal language (and its adaptation to specific parsing tools) to the corresponding data model. These updates are time-consuming, tedious, and error-prone. By making such changes labor-intensive, the traditional language processing approach hampers the maintainability and evolution of the language used to represent the data (Kats et al., 2010).

Moreover, it is not uncommon for different applications to use the same language. For example, the compiler, different code generators, and other tools within an IDE, such as the editor or the debugger, typically need to grapple with the full syntax of a programming language. Unfortunately, their maintenance typically requires keeping several copies of the same language specification synchronized.

The idea behind model-based language specification is that, starting from a single abstract syntax model (ASM) that represents the core concepts in a language, language designers can develop one or several concrete syntax models (CSMs). These CSMs can suit the specific needs of the desired textual or graphical representation. The ASM-CSM mapping can 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.

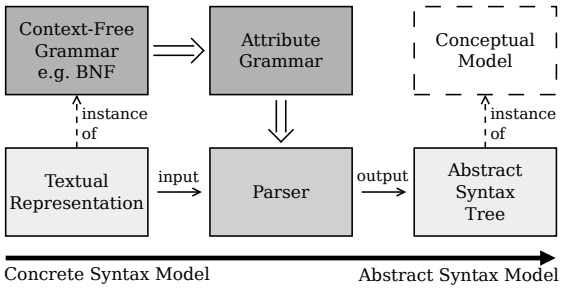

Figure 1: Traditional language processing.

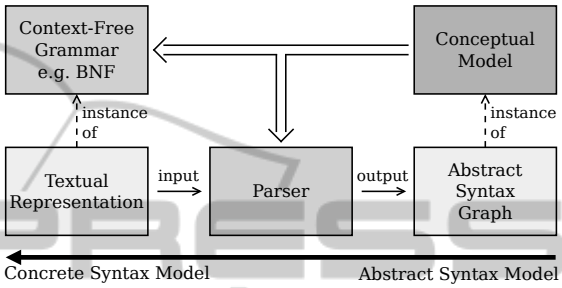

Figure 2: Model-based language processing.

This way, the ASM 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.

Finally, as the ASM 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. Therefore, by using an annotated ASM, model-based language specification completely decouples language specification from language processing, which can be performed using whichever parsing techniques are suitable for the formal language implicitly defined by the abstract model and its concrete mapping.

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

It should be noted that ASMs may represent non-tree structures. Hence the use of the 'abstract syntax graph' term in Figure 2.

ModelCC (Quesada et al., 2011) is a parser gen-

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

Constraints on...	Annotation	Function
...patterns	@Pattern	Pattern matching definition of basic language elements.
	@Value	Field where the recognized input element will be stored.
...delimiters	@Prefix	Element prefix(es).
	@Suffix	Element suffix(es).
	@Separator	Element separator(s) in lists of elements.
...cardinality	@Optional	Optional elements.
	@Minimum	Minimum element multiplicity.
	@Maximum	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	Define an element member position relative to other.
	@FreeOrder	All the element members positions may vary.
...references	@ID	Identifier of a language element.
	@Reference	Reference to a language element.
Custom constraints	@Constraint	Custom user-defined constraint.

erator that supports a model-based approach to the design of language processing systems. Its starting ASM is created by defining classes that represent language elements and establishing relationships among those elements. Once the ASM is established, constraints can be imposed over language elements and their relationships as annotations in order to produce the desired ASM-CSM mapping.

The ASM is built on top of basic language elements, which can be viewed as the tokens in the model-driven specification of a language. ModelCC provides 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.

In ModelCC, the constraints imposed over ASMs to define a particular ASM-CSM mapping are declared as metadata annotations on the model itself. Now supported by all the major programming platforms, metadata annotations are often used in reflective programming and code generation (Fowler, 2002). Table 1 summarizes the set of constraints supported by ModelCC for establishing ASM-CSM mappings.

When the ASM represents a tree-like structure, a model-based parser generator is equivalent to a traditional grammar-based parser generator in terms of expression power. When the ASM represents non-tree structures, reference resolution techniques can be employed to make model-based parser generators more powerful than grammar-based ones, as we will see in the next Section.

3 REFERENCE RESOLUTION SUPPORT IN MODELCC

Reference resolution consists of finding the object a reference refers to and, in the case of ModelCC, automatically linking the reference to the corresponding object instantiation. Reference resolution leads to abstract syntax graphs instead of trees in model-driven language processing.

In ModelCC, an object reference is embodied by a subset of the elements in its full object definition. This subset of elements acts as an identifier (or key in database terms) that, when found in the input text, can be recognized as a reference to the corresponding object in the model and linked to its instantiation in the ASM.

References in ModelCC can be anaphoric, when they are preceded by the corresponding object definition; cataphoric, when the references precede the definition; and recursive, when they appear within the definition they refer to.

Subsection 3.1 introduces the *@ID* metadata annotation, which allows the specification of identifiers for language elements. Subsection 3.2 presents the *@Reference* metadata annotation, which allows the specification of references to other language elements.

3.1 The @ID Annotation

ModelCC uses an *@ID* metadata annotation to support reference specification. This annotation is ap-

plied to a subset of the members of a language element model. This subset determines the syntax of references to particular instances of such elements in the concrete syntax of the corresponding language. That is, any appearance of the same set of values will be interpreted as a reference to the same instance of the referred language element.

The use of references is resolved in our implementation of ModelCC by the introduction of grammar productions that characterize such references and semantic actions that map them to the corresponding language elements.

In Figure 3, the *@ID* annotation is employed to identify users by a single number.

However, the *@ID* annotation can be used together with other ModelCC annotations, such as *@FreeOrder*, which allows the members of a language element to be shuffled in their textual representation, and *@Prefix* and *@Suffix*, which add syntactic sugar to the incarnation of the abstract syntax model as a concrete textual language.

The inadvertent definition of two entities of the same class with the same identifier results in a runtime warning produced by ModelCC when parsing its input.

3.2 The @Reference Annotation

ModelCC resorts to the *@Reference* metadata annotation to complete its support for reference resolution. The *@Reference* annotation applies to individual members of any language element, provided that the referenced types contain at least one *@ID*-annotated member in their model.

Whenever a language element member is annotated with *@Reference*, the corresponding grammar productions are modified so that they refer to the symbol corresponding to the element reference specification rather than the symbol that corresponds to its full specification. These productions are then associated to a semantic action that resolves the references at the end of the parsing process, in order to support cataphoric and recursive references, apart from the anaphoric references that could be resolved on the fly during the parsing process.

In Figure 3, the textual syntax of messages includes numbers that, as identifiers, refer to particular users. ModelCC will parse such identifiers, recognize the references, resolve them, and return the correct object graph.

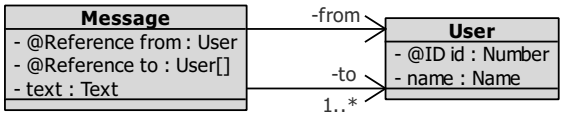

Figure 3: ModelCC specification of *Messages*, their senders, and their receivers.

4 A WORKING EXAMPLE

In this section, we present an example language that allows the specification and rendering of complex 3D objects using the reference resolution capabilities of ModelCC.

First, we will outline the features we wish to include in our 3D object specification language. Then, we will provide the full language specification for ModelCC by defining an abstract syntax model, which will be annotated to specify the desired ASM-CSM mapping. Lastly, we will see an example input and output pair for our 3D object specification language.

4.1 Language Description

Our 3D object specification language is designed to support the following features:

- A special section, denoted by the “scene” keyword, delimits the statements that will be used for rendering the scene.
- The definition of custom objects, which are identified by an object name. As references can be lazily resolved, recursion is allowed.
- Scoped statements, delimited by “{” and “}”, that allow the specification of lists of statements that will run in a new scope.
- Composite statements, delimited by “[” and “]”, that allow the specification of lists of statements that will run in the current scope.
- Repeated statements that allow the repetition of a statement, a group of statements, or a block of statements, a number of times.
- Draw statements, which draw either basic objects (e.g. a cube) or user-defined objects. Draw statements allow the specification of a numeric parameter. The “next” keyword, when used as this numeric parameter, is replaced in runtime by the current parameter decreased by one, and draw statements will not run when the parameter is 0.
- Scale transformation statements, which support the specification of a combination of x, y, and z

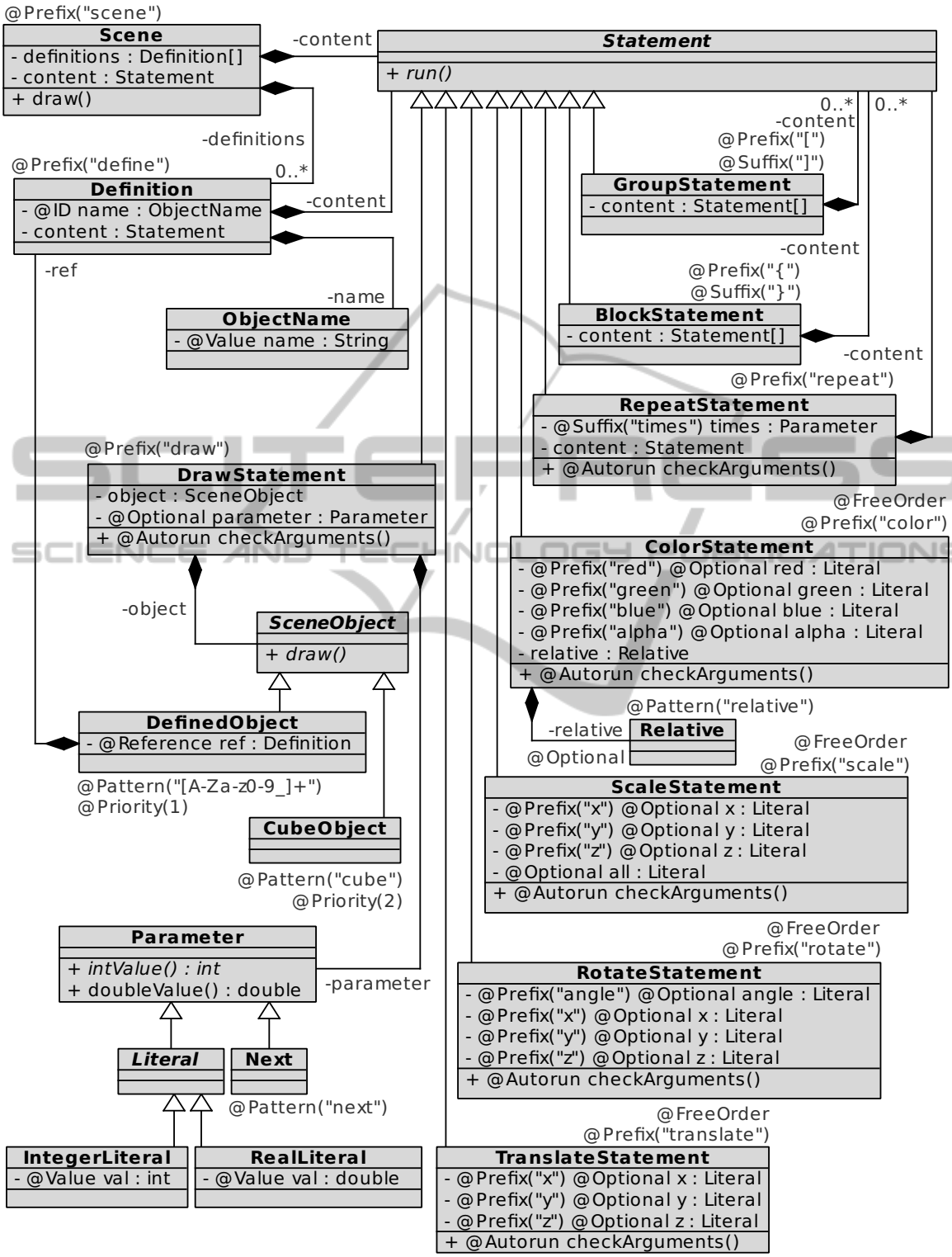

Figure 4: ModelCC definition of a 3D object specification language. ModelCC reference resolution support is used to allow the specification of complex 3D objects in the *Definition* class.

values in any order, or a single scaling factor that will be applied to the three axes.

- Rotate transformation statements, which support the specification of the angle and a combination of x, y, and z axis values in any order.
- Translate transformation statements, that support the specification of a combination of x, y, and z values in any order.
- Color setting statements, which support the specification of a combination of red, green, blue, and alpha values in any order, and allow either absolute (by default) or relative color adjustments.

4.2 ModelCC Implementation

In ModelCC, the abstract syntax model is designed, then mapped to a concrete syntax model by imposing constraints by means of metadata annotations on the abstract syntax model.

The resulting model can be processed by ModelCC to automatically generate the corresponding parser. The UML class diagram in Figure 4 presents our annotated 3D object specification language model.

The reference support extension we propose in this paper can be observed in the *Definition*, *ObjectName*, and *DefinedObject* classes. The *name* member of the *Definition* class is annotated with *@ID*, which means that a *Definition* instance can be identified by an *ObjectName*. Then, the *ref* member of a *DefinedObject* is annotated with *@Reference*, which means that, in textual form, a *DefinedObject* can refer to a *Definition* by its *ObjectName*. ModelCC reference resolution allows references to be resolved during the parsing process and makes the implementation of a traditional symbol table unnecessary.

It should be noted that certain constraints cannot be expressed in the abstract syntax model. However, these constraints can be expressed as custom constraints using the *@Constraint* annotation. In our example, some statements corresponding to elements in our model, such as draw statements and repeat statements, will not accept real values as parameters. These custom semantic constraints are implemented in the *checkArguments()* methods of the language elements classes corresponding to those statements.

ModelCC is able to automatically generate a grammar from the ASM defined by a class model and the ASM-CSM mapping defined as a set of metadata annotations on the class model. References in that grammar are automatically resolved by ModelCC so that further work is not needed.

```
define trunk {
  color red 0.87 green 0.50 blue 0.10
        alpha 1
  draw cube
  repeat 10 times [
    scale x 1.02 z 1.02 y 0.98
    color relative red -0.03 green -0.02
                    blue -0.01
    draw cube
  ]
}

define leaves {
  color red 0.2 green 0.9 blue 0.3
        alpha 0.9
  translate x -1
  {
    scale z 0.6 y 0.05
    repeat 100 times [
      color relative red +0.005
                     alpha -0.005
      translate x -0.04 y -0.3
      draw cube
    ]
  }
}

define palmtree {
  repeat 8 times [
    draw trunk
    translate y 1
  ]
  repeat 3 times [
    translate y -0.5
    scale 0.7
    rotate angle 8 y 1
    repeat 15 times [
      rotate angle 24 y 1
      draw leaves
    ]
  ]
}

scene {
  draw palmtree
}
```

Figure 5: Specification of a palmtree in our 3D object specification language.

4.3 Example of 3D Object Specification

Figures 5 and 6 illustrate the specification and rendering of a 3D palmtree in our 3D object specification language. The *palmtree* object is defined as eight

Figure 6: Rendering of the palm tree from Figure 5.

trunk sections with leaves on the top.

5 CONCLUSIONS AND FUTURE WORK

ModelCC is a model-based parser generator that employs metadata annotations to implement ASM-CSM mappings.

We have described how ModelCC supports reference resolution and allows parsing abstract syntax graphs rather than conventional abstract syntax trees, as obtained by traditional grammar-driven parser generators.

We have demonstrated the use of ModelCC reference resolution support with a fully-functional abstract syntax graph parser for a 3D object specification language.

In the future, we plan to apply model-based language specification techniques to problems such as data integration. We also plan to implement metadata annotations that support more complex scoping rules for reference resolution.

ACKNOWLEDGEMENTS

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

REFERENCES

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition.

Fowler, M. (2002). Using metadata. *IEEE Software*, 19(6):13–17.

Ginsburg, S. (1975). *Algebraic and automata theoretic properties of formal languages*. North-Holland.

Harrison, M. A. (1978). *Introduction to Formal Language Theory*. Reading, Mass: Addison-Wesley Publishing Company.

Jurafsky, D. and Martin, J. H. (2009). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*. Prentice Hall, 2nd edition.

Kats, L. C. L., Visser, E., and Wachsmuth, G. (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)*, pages 918–932.

Kleppe, A. (2007). Towards the generation of a text-based IDE from a language metamodel. volume 4530 of *Lecture Notes in Computer Science*, pages 114–129.

Quesada, L., Berzal, F., and Cubero, J.-C. (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*, volume 6936, pages 50–57.