ModelCC

v2.02 (2013-05-25)

User Manual

The ModelCC Development Team

www.modelcc.org

- Luis Quesada (lquesada@modelcc.org)
- Fernando Berzal (berzal@modelcc.org)
- Francisco J. Cortijo (cb@modelcc.org)
- Juan-Carlos Cubero (jc.cubero@modelcc.org)

Contents

1 Introduction

It should be noted that this user manual is under heavy construction. A complete version will be published in the website.

ModelCC is a model-based parser generator that decouples language specification from language processing.

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. 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.

Section 2 comments on ModelCC building blocks.

Section 3 describes the model constraints supported by ModelCC, which declaratively define the features of the formal language defined by the model.

Section 4 defines ModelCC predefined types.

Section 5 explains how to use ModelCC.

Section 6 exposes examples of languages specified using ModelCC.

2 Model Specification

Any class implementing the *IModel* interface is considered a language element.

The specification of a language in ModelCC starts with the definition of basic language elements, which might be viewed as the tokens in the model-driven specification of a language. Basic language elements can be modeled as simple classes in an object-oriented programming language annotated with pattern constraints, as described in Section 3.1. 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.

Then, 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 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.

2.1 Concatenation

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 object composition. The resulting language element is the composite element and its members are the language elements the composite element collates.

The following code shows an example of object composition, in which an *AssignmentStatement* is composed of an *Identifier* and an *Expression*:

```
import org.modelcc.*;

public class AssignmentStatement implements IModel {

   Identifier id;

   Expression exp;

}
```

2.2 Selection

Selection allows the specification of 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. The language element we wish to establish alternatives for is the superelement, whereas the different alternatives are represented as subelements.

The following code shows an example of selection, in which an *Expression* can be either an *UnaryExpression*, a *BinaryExpression*, or a *ParenthesizedExpression*:

```
import org.modelcc.*;

public class Expression implements IModel {

}
```

```
import org.modelcc.*;

public class UnaryExpression extends Expression implements IModel {

}
```

```
import org.modelcc.*;

public class BinaryExpression extends Expression implements IModel {

}
```

```
import org.modelcc.*;

public class ParenthesizedExpression extends Expression
                                                        implements IModel {

}
```

2.3 Repetition

Repetition allows the specification of a language element that appears several times in a given language construct.

In ModelCC, repetition is also achieved through object composition, just by allowing different multiplicities in the associations that connect composite elements to their constituent elements, that is, by declaring container members such as arrays, Lists/ArrayLists, or Sets/HashSets. The cardinality constraints described in Section 3.3 can be used to annotate ModelCC models in order to establish specific multiplicities for repeatable language elements.

The following five pieces of code show examples of repetition, in which an *OutputStatement* can contain an array, a List, an ArrayList, a Set, or an HashSet of *Expressions*.

```
import org.modelcc.*;

public class OutputStatement extends Statement implements IModel {

  Expression[] exp;

}
```

```
import org.modelcc.*;

public class OutputStatement extends Statement implements IModel {

  List<Expression> exp;

}
```

```
import org.modelcc.*;

public class OutputStatement extends Statement implements IModel {

  ArrayList<Expression> exp;

}
```

```
import org.modelcc.*;

public class OutputStatement extends Statement implements IModel {

  Set<Expression> exp;

}
```

```
import org.modelcc.*;

public class OutputStatement extends Statement implements IModel {

  HashSet<Expression> exp;

}
```

3 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 ModelCC, the constraints are declared as metadata annotations on the model itself, and all constraints except prefixes and suffixes are inherited from superelements to subelements.

3.1 Pattern Specification Constraints

Pattern specification constraints allow the specification of the basic language elements, i.e. the different token types defined for a textual language.

3.1.1 The @Pattern annotation

The *@Pattern* annotation allows the specification of the pattern that will be used to match a basic language element in the input string. Two mutually exclusive mechanisms are provided for pattern specification in ModelCC: regular expressions and user-defined pattern matching classes. Regular expressions can be specified in ModelCC to build standard lexers, whereas custom pattern matching classes allow the language designer to use any custom-defined matching element to recognize basic language elements in the input string.

When used with regular expressions, the *@Pattern* annotation includes an argument representing the regular expression. This regular expression, specified as the *regExp* field of the annotation, corresponds to the traditional token type definition in lex-like scanners.

When used with custom pattern matching classes, the *@Pattern* annotation is used to specify the name of the class implementing the matching algorithm (this class should extend the abstract class *PatternRecognizer*) in the *matcher* field of the annotation and an argument string in the *args* field of the annotation.

It should be noted that ModelCC supports lexical and syntactic ambiguities.

The following code shows an example of regular expression pattern specification, in which an *Identifier* can be specified by the regular expression *[a-zA-Z][_a-zA-Z0-9]**:

```
import org.modelcc.*;

 @Pattern(regExp=”[a-zA-Z][˙a-zA-Z0-9]*”)
public class Identifier implements IModel {

}
```

The following code shows an example of matcher class pattern specification, in which *JavaDoc* can be specified by a *JavaDocRecognizer* class and the “simple” argument:

```
import org.modelcc.*;

 @Pattern(matcher=JavaDocRecognizer.class,args=”simple”)
public class JavaDoc implements IModel {

}
```

3.1.2 The @Value annotation

The *@Value* annotation can be used in combination with the *@Pattern* annotation in ModelCC to indicate the location where the recognized token value will be stored in the abstract syntax graph, so that value can be used once the input string has been parsed.

Associated to a field of the class defining a basic language element, that field will contain the value obtained from the input string that matches the token type pattern specification.

When a numeric or boolean field is annotated with the *@Value* annotation, it is not necessary to specify the corresponding *@Pattern* annotation for recognizing the numeric or boolean tokens. When the *@Value*-annotated field is not numeric nor boolean (e.g. a string, a single character, or any non-primitive data type), the use of the *@Pattern* annotation is mandatory.

The following code shows an example of an *@Value* annotation defining a primitive pattern, in which *IntegerLiteral* will have a *long*-matching pattern and the matched value will be stored in the *value* member:

```
import org.modelcc.*;

public class IntegerLiteral implements IModel {

  @Value
  long value;

}
```

The following code shows an example of an *@Value* annotation defining where to store the matched value when a non-primitive pattern is used, in which a *StringLiteral* will be recognized whenever a pair of double quotes encloses a string not containing a double quote, a constraint that can be specified by the regular expression *\“[^\“]*\“*:

```
import org.modelcc.*;

@Pattern(regExp="\"[^\"]*\""
public class StringLiteral implements IModel {

  @Value
  String text;

}
```

3.2 Delimiter Constraints

Delimiter constraints allow the specification of language element delimiters in a concrete syntax model. Delimiters include prefixes, suffixes, and separators. Such kinds of delimiters are often used to eliminate language ambiguities and facilitate parsing, but they can appear just as syntactic sugar to make languages more readable.

Usually, reserved words in programming languages act just as delimiters. As such, they will not appear in the language abstract syntax model. They will be specified as metadata annotations in the ASM-CSM mapping corresponding to the concrete syntax of the language.

It should be noted that delimiters should always be specified as constraints on the ASM-CSM mapping rather than language elements in the ASM, since they do not provide any relevant information from the perspective of the abstract syntax model, even though they might be necessary to define unambiguous textual languages.

3.2.1 The @Prefix annotation

The *@Prefix* annotation allows the specification of prefixes for language elements and specific constituents in composite language elements.

The *value* field of the *@Prefix* annotation is used to specify the list of regular expressions that define the prefixes that precede the corresponding language element (or a specific constituent element within a composite element) in the concrete syntax of a textual CSM.

The following pieces of code show an example of prefixes, in which a *Program* is prefixed by the reserved word “main”:

```
import org.modelcc.*;

 @Prefix(”main”)
public class Program implements IModel {

  Statement main;

}
```

```
import org.modelcc.*;

public class Program implements IModel {

  @Prefix(”main”)
  Statement main;

}
```

3.2.2 The @Suffix annotation

The *@Suffix* annotation allows the specification of suffixes for language elements and specific constituents in composite language elements.

The *value* field of the *@Suffix* annotation is used to specify the list of regular expressions that define the suffixes that follow the corresponding language element (or a specific constituent element within a composite element) in the concrete syntax of a textual CSM.

The following code shows an example of suffixes, in which an *InputStatement* is suffixed by ";" and the list of *Identifiers* it contains is suffixed by ")":

```
import org.modelcc.*;

@Prefix("input")
 @Suffix(";")
public class InputStatement implements IModel {

  @Prefix("(")
  @Suffix(")")
  List<Identifier> ids;

}
```

3.2.3 The @Separator annotation

The *@Separator* annotation allows the specification of separators between consecutive instances of elements within a repetition. Separators can be defined in ModelCC by annotating a language element in the ASM or just its appearance within a particular repetition construct. In the first case, a default separator is established for the language element: the specified separator will be used for separating consecutive instances of the annotated language element whenever a sequence of such language elements appears in a textual CSM. In the second case, an *ad hoc* separator is defined: the specified separator will be used only when consecutive instances of the language element appear within the context of the annotated repetition construct.

The *ad hoc* definition of separators with the *@Separator* annotation within repetition constructs can be used to override the default sequence of separators associated to the repeatable element in a repetition construct (or just to disable the use of separators for that specific repetition).

The following code shows an example of default separators, in which an *Identifier* is set the default separator ",", which will be used in the set of identifiers of a *VariableDeclaration*:

```
import org.modelcc.*;

public class VariableDeclaration implements IModel {

  Type type;

  Set<Identifier> ids;

}
```

```
import org.modelcc.*;

 @Separator(",")
public class Identifier implements IModel {

}
```

The following code shows an example of *ad hoc* separators, in which an *Identifier* is set the ad hoc separator "," in a *VariableDeclaration*:

```
import org.modelcc.*;

public class VariableDeclaration implements IModel {

  Type type;

  @Separator(",")
  Set<Identifier> ids;

}
```

3.3 Cardinality Constraints

Cardinality constraints on language elements control element repeatability and optionality.

3.3.1 The @Optional annotation

The *@Optional* annotation allows the specification of optional elements in textual CSMs.

When one of the constituent elements within a composite language element is optional, the textual representation of the composite element might include the optional element, along with its corresponding delimiters, or not. In the latter case, the missing element delimiters are not included in the textual representation the composite element either, even though a prefix and a suffix might have been defined for the missing constituent element.

The following code shows an example of optionality, in which a *ConditionalStatement* contains an *Expression* (the condition), the “if” *Statement*, and an optional “else” *Statement*:

```
import org.modelcc.*;

@Prefix("if")
@Suffix(";")
public class ConditionalStatement implements IModel {

  Expression cond;

  Statement ifst;

  @Prefix("else")
  @Optional
  Statement elsest;

}
```

3.3.2 The @Minimum annotation

The *@Minimum* annotation allows the specification of the lower bound for the multiplicity of repeatable language elements within repetition constructs. This lower bound is 1 by default.

It should be noted that, when the minimum multiplicity is 0, no elements might appear in a particular instance of the repetition. However, delimiters would still be represented in the textual CSM unless the *@Optional* annotation were explicitly employed.

The following code shows an example of minimum cardinality, in which an *ExpressionSet* contains a set of 0 or more *Expressions* delimited by “” and “”:

```
import org.modelcc.*;

@Prefix("")
@Suffix("")
public class ExpressionSet implements IModel {

   @Minimum(0)
  Set<Expression> exps;

}
```

3.3.3 The @Maximum annotation

The *@Maximum* annotation, depicted as a maximum multiplicity constraint in standard UML associations, allows the specification of the upper bound for the multiplicity of repeatable language elements within repetition constructs. This upper bound is undefined by default.

The following code shows an example of maximum cardinality, in which an *Program* contains an array of 0 to 2 *Parameters*:

```
import org.modelcc.*;

public class Program implements IModel {

  @Minimum(0)
   @Maximum(2)
  Parameter[] pars;

}
```

3.4 Evaluation Order Constraints

Evaluation order constraints are employed to declaratively resolve syntactic ambiguities in the concrete syntax of a textual language by establishing associativity, composition, and precedence constraints for CSMs.

3.4.1 The @Associativity annotation

The *@Associativity* annotation allows the specification of the operator-like associativity of language elements. ModelCC supports the following options for specifying associativity constraints:

- *AssociativityType.UNDEFINED*, when no associativity is declared (by default), all possibilities are considered.
- *AssociativityType.LEFT_TO_RIGHT*, for left-associative operations (e.g. substraction, division, or function application).
- *AssociativityType.RIGHT_TO_LEFT*, for right-associative elements (e.g. exponentiation and function definition).
- *AssociativityType.NON_ASSOCIATIVE*, for non-associative elements (e.g. cross of three vectors).

The following pieces of code show an example of left to right associative *BinaryOperator*:

```
import org.modelcc.*;

public class Expression implements IModel {

}
```

```
import org.modelcc.*;

public class BinaryExpression extends Expression implements IModel {

  Expression e1;

  BinaryOperator op;

  Expression e2;

}
```

```
import org.modelcc.*;

@Pattern(regExp="\+|-")
 @Associativity(AssociativityType.LEFT_TO_RIGHT)
public class BinaryOperator implements IModel {

  @Value
  char operator;

}
```

3.4.2 The @Composition annotation

The *@Composition* annotation allows the specification of the suitable order of evaluation of compositions represented in a CSM, a situation that appears whenever the composite design pattern is present in the ASM, no delimiters are employed to eliminate potential ambiguities, and the composite contains several consecutive components of the same type of the composite. When such composites are nested, different interpretations are possible unless we specify composition constraints in the CSM. This is the case of the typical shift-reduce conflicts that appear in LR parsers when parsing nested if-then-else statements.

Hence, a specific constraint on element composition must be used to enforce a particular interpretation of such nested compositions in the ASM-CSM mapping. ModelCC supports the following options for composition constraints:

- *CompositionType.UNDEFINED*, when no composition constraints are defined and potential ambiguities are taken into account.
- *CompositionType.EAGER*, when the matching of constituent elements is performed as soon as possible. This corresponds to the typical interpretation of nested if-then-else statements in programming languages, where the else clause is attached to the innermost if statement.
- *CompositionType.LAZY*, when the matching of constituent elements is deferred as much as possible. Then, a rightmost derivation is obtained; i.e. when an element might accompany any of two nested language constructs, it is associated to the outermost one.
- *CompositionType.EXPLICIT*, when no composition constraints are defined and any ambiguities should be resolved with the help of delimiters.

The following pieces of code show an example of eager *ConditionalSentence*, in which the “else” sentence of “if cond if cond2 else sent” will correspond to the internal “if” sentence:

```
import org.modelcc.*;

public class Statement implements IModel {

}
```

```
import org.modelcc.*;

@Prefix("if")
@Suffix(";")
 @Composition(CompositionType.EAGER)
public class ConditionalStatement extends Statement implements IModel {

  Expression cond;

  Statement ifst;

  @Prefix("else")
  @Optional
  Statement elsest;

}
```

3.5 Composite Element Order Constraints

Element order constraints allow modifying the order of elements in compositions.

3.5.1 The @FreeOrder annotation

The *@FreeOrder* annotation allows the specification of composite elements in which their component elements may appear in any order.

The default behavior is that the elements may appear only in the same order that specified. The annotation accepts an optional boolean argument which enables or disables (default enables) the free order. It should be noted that as this constraint is inherited, if a superelement has free order enabled, subelements will have it enabled too unless specified otherwise.

The following code shows an example of free order, in which a *Program* can contain a set of *FunctionDeclarations* and a main *Sentence* in any order:

```
import org.modelcc.*;

 @FreeOrder
public class Program implements IModel {

  Set<FunctionDeclaration> functions;

  Sentence main;

}
```

3.6 Reference Constraints

Reference constraints allow specifying references to elements.

3.6.1 The @ID annotation [since ModelCC v2.00]

The *@ID* annotation allows the specification of the components of an element that represent its key.

The following code shows an example of key specification, in which an *IDNumber* preceded by the word "ID" identifies an *User*:

```
import org.modelcc.*;

public class User implements IModel {

   @ID
  @Prefix("ID")
  IDNumber id;

  Name userName;

}
```

3.6.2 The @Reference annotation [since ModelCC v2.00]

The *@Reference* annotation allows the specification of references to an element. Whenever a key for that element is found, it is considered a reference to it, and therefore the very same object is linked to the reference.

The following code shows an example of reference, in which an *IDNumber* preceded by the word “ID” is enough to identify the *User* in a *Sale*:

```
import org.modelcc.*;

public class Sale implements IModel {

   @Reference
  User user;

  Product productName;

  Number quantity;

}
```

3.7 User defined constraints

ModelCC provides a mechanism for the user to implement any other constraint and semantic actions he needs.

3.7.1 The @Autorun annotation

the *@Autorun* annotation allows the specification of a method that will be run after an instance of a language element has been generated. That method can perform any action involving the *@Value*-annotated language element member or any other element member. It has to accept no parameters and must return a boolean that specifies if the element is valid and can be generated.

The following code shows an example of user defined constraint, in which a *BoundedInteger* has to be between 4 and 17:

```
import org.modelcc.*;

public class BoundedInteger implements IModel {

  @Value
  int value;

   @Autorun
  boolean check() {
    if (value >= 4 && value <= 17)
      return true;
    else
      return false;
  }

}
```

4 Predefined types [since ModelCC v2.01]

ModelCC provides several predefined models for common data types.

These models are found in the package org.modelcc.types and are:

- ByteModel: A model that recognizes decimal integers and stores them as a Byte object would.
- ShortModel: A model that recognizes decimal integers and stores them as a Short object would.
- IntegerModel: A model that recognizes decimal integers and stores them as a Integer object would.
- LongModel: A model that recognizes decimal integers and stores them as a Long object would.
- FloatModel: A model that recognizes decimal integers and stores them as a Float object would. It supports scientific notation.
- DoubleModel: A model that recognizes decimal integers and stores them as a Double object would. It supports scientific notation.

5 Getting Started on ModelCC

This section explains how to use ModelCC.

Subsection 5.1 shows how to set ModelCC up.

Subsection 5.2 explains how to implement an example language model.

Subsection 5.3 comments on how to generate and use a parser from a language model.

5.1 Setting up ModelCC

You can get started to ModelCC in a few very easy steps.

5.1.1 Downloading ModelCC

Be sure to agree with the ModelCC Shared Software License before downloading ModleCC.

1. Download the ModelCC binary distribution package (modelcc-[version].zip) from the ModelCC site (www.modelcc.org).
2. Extract it.

5.1.2 Setting up ModelCC in a NetBeans project

(If you are not planning on using ModelCC in NetBeans, you may skip this section.)

Whenever you want to use ModelCC in a NetBeans project, you have to add the ModelCC library to it.

1. Right click your project.
2. Go to **Properties**.
3. Go to the **Libraries** tab.
4. In the **Compile** tab (which is open by default), click the **Add JAR/Folder...** button.
5. Go to the directory where you extracted **modelcc-[version].zip** and select **ModelCC.jar**. Do not select ModelCCExamples.jar, which is a stand-alone application for testing example languages.
6. Accept.

5.1.3 Setting up ModelCC in an Eclipse project

(If you are not planning on using ModelCC in Eclipse, you may skip this section.)

Whenever you want to use ModelCC in an Eclipse project, you have to add the ModelCC library to it.

1. Right click your project.
2. Go to **Properties**.
3. Go to the **Java Build Path** section.
4. In the **Libraries** tab, click the **Add External JARs...** button.
5. Go to the directory where you extracted **modelcc-[version].zip** and select **ModelCC.jar**. Do not select ModelCCExamples.jar, which is a stand-alone application for testing example languages.
6. Accept.

5.2 Implementing an example language model

The following class describes a very simple language. The entity **MySimpleLanguage** is a lexical entity that consists of an integer value, which is deducted from the **@Value** annotation that is assigned to an int data type. Any integer number pertains to this language.

```
import org.modelcc.*;

public class MySimpleLanguage implements IModel

    @Value
    private int data;

    public int getData()
        return data;
```

More complex example languages can be found in the Examples section of the ModelCC site (www.modelcc.org) and in the **ModelCCExamples** library, which is included in the source and binary distribution packages of ModelCC.

5.3 Generating and using a parser for the example language model

The parser generation step consists of reading the model of a language and generating a parser from the model.

Subsection 5.3.1 explains how to read the model of a language. Subsection 5.3.2 explains how to generate a parser from the language model. Subsection 5.3.3 explains how to use the parser.

5.3.1 Reading the model of a language

The *Model* class stores language models.

A model can be readed by using a *ModelReader*.

ModelCC currently provides *JavaModelReader*, which is able to produce a model from a set of annotated classes. The following code illustrates the usage of this model reader:

```
import org.modelcc.io.ModelReader;
import org.modelcc.io.java.JavaModelReader;
import org.modelcc.metamodel.Model;



[...]

//being MySimpleLanguage the main model class.

try {
  ModelReader jmr = new JavaModelReader(MySimpleLanguage.class);
  Model m = jmr.read();
} catch (Exception e) {
  System.out.println(e.getMessage());
}
```

When JavaModelReader is instantiated that way, it provides the *getWarnings()* method, which returns a list of Strings corresponding to the warnings generated during the model reading.

The following code illustrates another way to use this model reader:

```
import org.modelcc.io.java.JavaModelReader;
import org.modelcc.metamodel.Model;

[...]

//being MySimpleLanguage the main model class.

try {
  Model m = JavaModelReader.read(MySimpleLanguage.class);
} catch (Exception e) {
  System.out.println(e.getMessage());
}
```

5.3.2 Generating a parser from the language model

A *Parser* can be generated from a model.

ModelCC currently provides *FenceParserGenerator*, which is able to produce a Fence parser with a subyacent Lamb lexer from a model and an optional *skip* model which determines the patterns that should be ignored by the lexer (i.e. comments). It should be noted that, in the current implementation, the Lamb lexer ignores any character that does not match a pattern, and the *skip* model can be used to ignore patterns that would be matched as other tokens. The following code illustrates the usage of this parser generator:

```
import org.modelcc.metamodel.Model;
import org.modelcc.parser.fence.adapter.FenceParserGenerator;
import org.modelcc.parser.Parser;

[...]

//being m the model.

try {
  Parser<MySimpleLanguage> parser = FenceParserGenerator.create(m);
} catch (Exception e) {
  System.out.println(e.getMessage());
}
```

The following code illustrates the usage of this parser generator with a *skip* model:

```
import org.modelcc.metamodel.Model;
import org.modelcc.parser.fence.adapter.FenceParserGenerator;
import org.modelcc.parser.Parser;

[...]

//being m the model.
//being skip the skip model.
//being MySimpleLanguage the main model class.

try {
  Parser<MySimpleLanguage> parser = FenceParserGenerator.create(m,skip);
} catch (Exception e) {
  System.out.println(e.getMessage());
}
```

5.3.3 Using the parser

After the parser has been generated, it accepts strings or readers as input and produce instances of the main class. The parser provides several methods that produce different sets of results, considering one or several interpretation (in the case of ambiguities). The following code illustrates the usage of the parser:

```
import org.modelcc.parser.Parser;

[...]

//being parser a Parser<MySimpleLanguage>.
//being MySimpleLanguage the main model class.
//being input an input String or Reader.

MySimpleLanguage result = parser.parse(input);
```

The parser also provide the *parseAll* method, which return a *Collection* of instances of the main class, one for each possible interpretation; and the *parseIterator* method, which return an *Iterator* to the instances of the main class, which allows considering each possible interpretation.

5.3.4 Full example

The following code reads a model from a set of annotated Java classes, generates a parser, and uses it to generate an object instance from an input string:

```
import org.modelcc.io.java.JavaModelReader;
import org.modelcc.metamodel.Model;
import org.modelcc.parser.fence.adapter.FenceParserGenerator;
import org.modelcc.parser.Parser;

[...]

try {

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

  // Generate a parser from the model.
  Parser<Expression> parser = FenceParserGenerator.create(m);

  // Parse an input string.
  Expression result = parser.parse("3+(2+5)");

  // Produce output.
  if (result != null)
    System.out.println(result.eval());
  else
    System.out.println("Invalid string.");

} catch (Exception e) {
  System.out.println(e.getMessage());
}
```

6 Examples

This section exposes some ModelCC examples.
Subsection 6.1 shows the Simple Arithmetic Expression language.
Subsection 6.2 shows the Canvas Draw language.
Subsection 6.3 shows the Imperative Arithmetic language.

6.1 Simple Arithmetic Expression language

The Simple Arithmetic Expression language processes arithmetic expressions in infix notation.

6.1.1 Language Specification

The Simple Arithmetic Expression language features the following capabilities:

- Unary operators: +, and -.
- Binary operators: +, -, *, and /, being - and / left-associative.
- Operator priorities: * and / precede + and -.
- Parenthesized expressions.
- Integer and floating-point literals.
- Binary operators: +, and -.

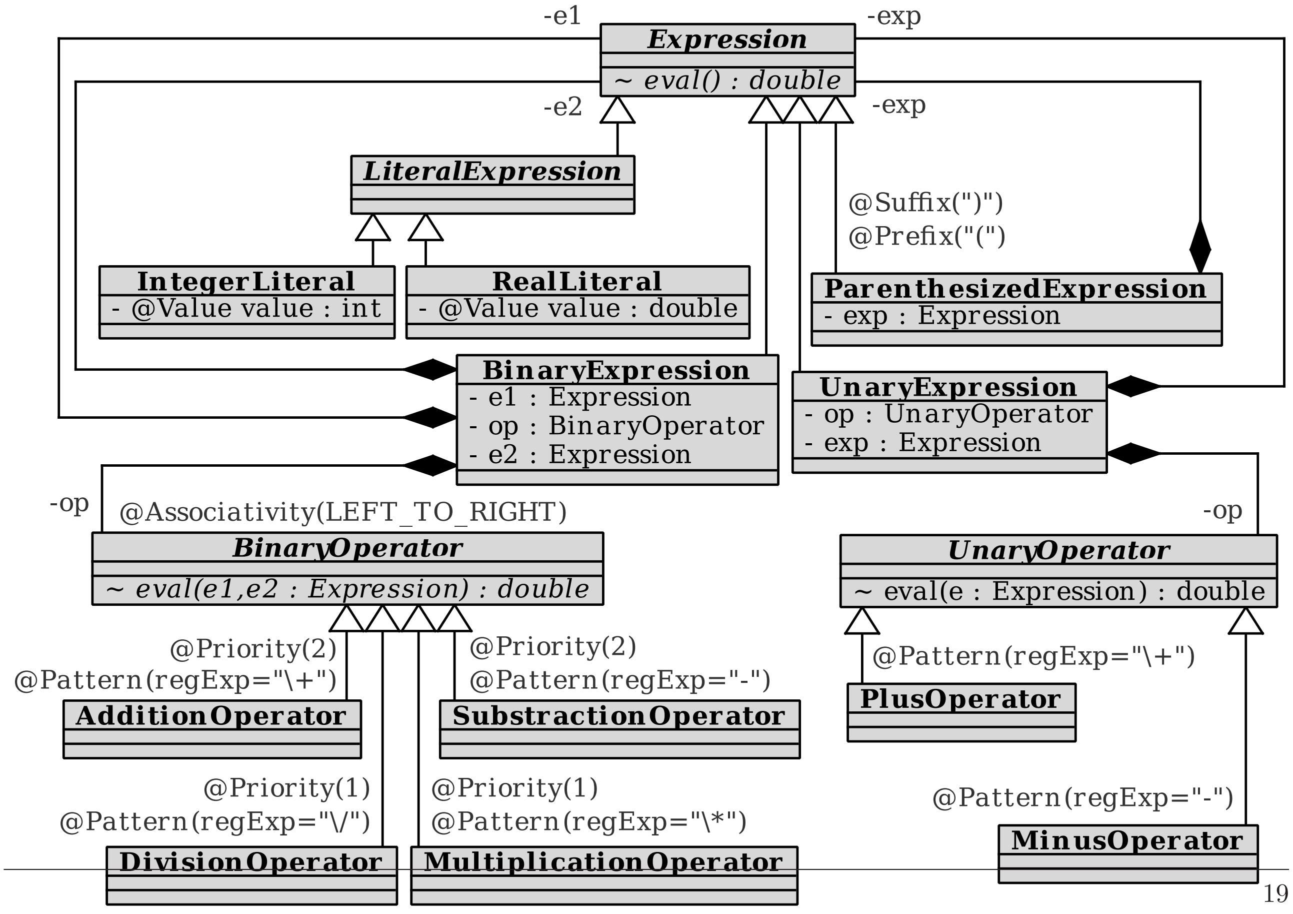
Example input and output:

- Input: (3+2)*2

  - Output: 10.0

- Input: 3+2*2
  - Output: 7.0
- Input: 2*3+2
  - Output: 8.0
- Input: (2*2)+2
  - Output: 6.0
- Input: 10*5/5
  - Output: 10.0
- Input: 6*2-3*3
  - Output: 3.0
- Input: 1+2+3+4+5+6+7+8+9*10
  - Output: 126.0
- Input: 10*4+5-22+(0.6*52-22)
  - Output: 32.2

6.1.2 UML class diagram



6.1.3 Java class implementation

The full functional implementation of this example (including valid import and package statements) can be found in the ModelCCExamples library, which is included in the source and binary distribution packages in the download section of the site (www.modelcc.org).

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

@Prefix("\\(") @Suffix("\\)")
public class ParenthesizedExpression
         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);
}

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

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

@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();
  }
}
```

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

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

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

6.2 Canvas Draw language

The Canvas Draw language allows the imperative generation of drawings.

6.2.1 Language Specification

The Canvas Draw language features the following capabilities:

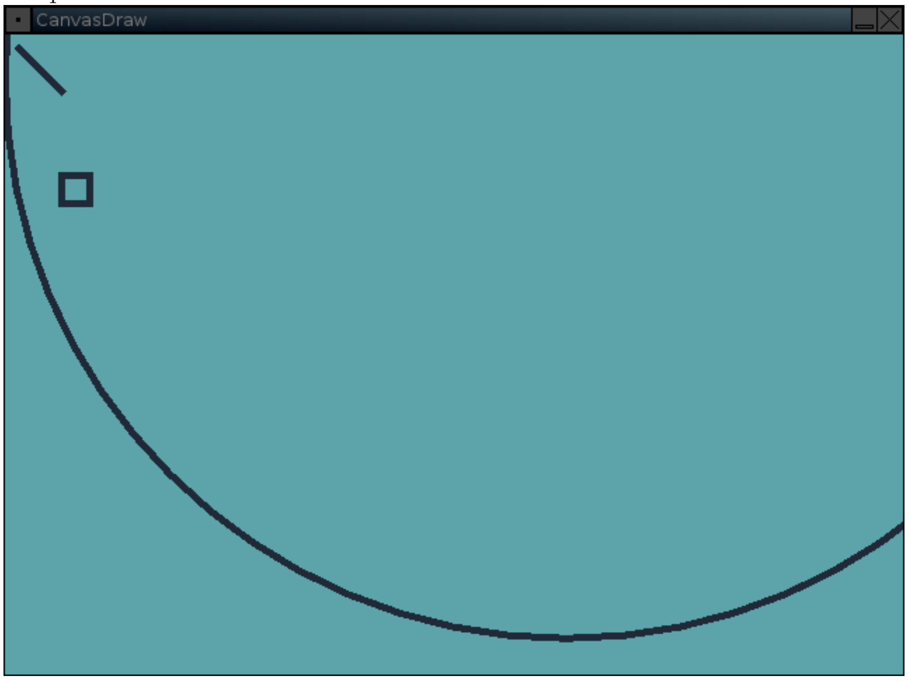
- Setting the canvas width, height, and background color.
- Changing the current color.
- Changing the current stroke width.
- Drawing lines, rectangles, circles, ovals, polygons, filled rectangles, filled circles, filled ovals, and filled polygons.
- Each of the elements is described in a particular way (list of coordinates, coordinates and radius, ...).

Example input and output:

- Input:

```
canvas                                  % Creates a canvas
     width 640                          % with width 640,
     height 480                         % height 480,
     background (92,164,169)            % and a cyan background.
                                        % Then:
   color (31,41,56)                     % Sets the color to dark cyan
   stroke 5                             % Sets the stroke to 5
   line [(10,10),(40,40)]               % Draws a line at (10,10) to (40,40)
   rectangle [(40,100),(60,120)]        % Draws a rectangle at (40,100) to (60,120)
   circle (400,30),400
```

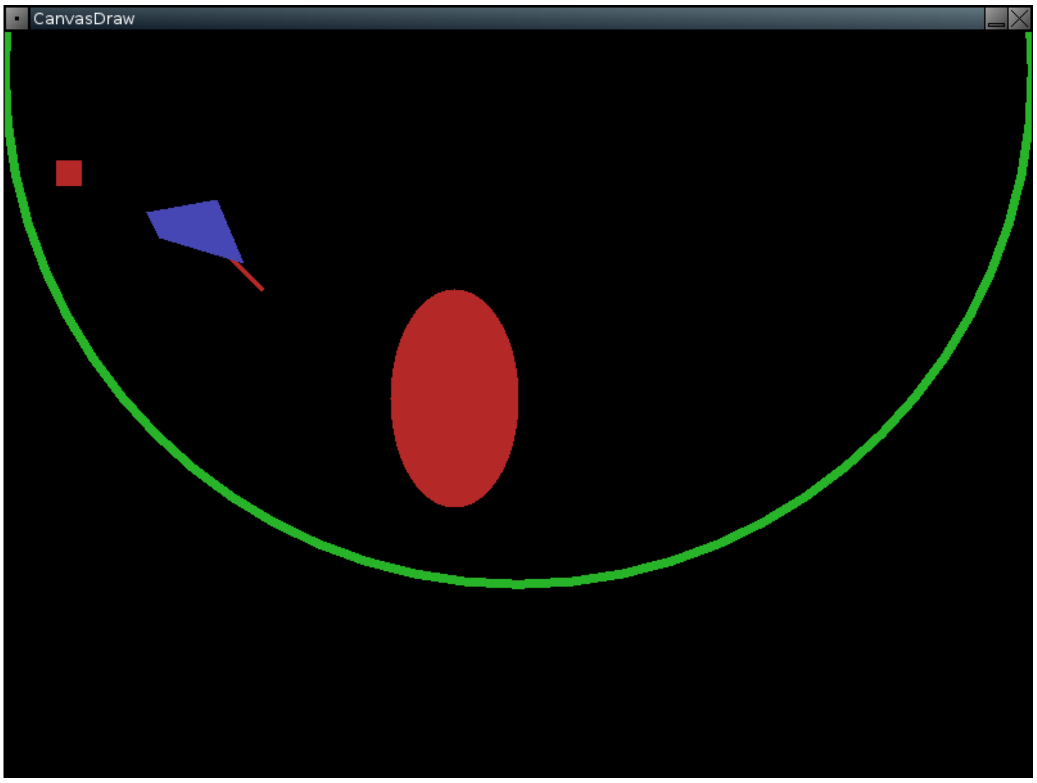
– Output:



- Input:

```
canvas                                   % Creates a canvas
     width 800                           % with width 800,
     height 600                          % height 600,
     background (0,0,0)                  % and a black background.
                                         % Then:
   color (180,40,40)                     % Sets the color to red
   rectangle [(40,100),(60,120)] fill    % Draws a fill rectangle at (40,100) to (60,120)
   oval [(300,200),(400,370)] fill       % Draws an oval at (300,200) to (400,370)
   stroke 3                              % Sets the stroke to 3
   line [(150,150),(200,200)]            % Draws a line at (150,150) to (200,200)
   color (40,180,40)                     % Sets the color to green
   stroke 7                              % Sets the stroke to 7
   circle (400,30),400                   % Draws a circle of radius 400 at (400,30)
   color (70,70,180)                     % Sets the color the purple
   polygon [(120,160),(110,140),         % Draws a fill polygon.
           (165,130),(186,180)] fill
```

– Output:



6.3 Imperative Arithmetic language

The Imperative Arithmetic language processes processes assignment and output sentences containing arithmetic expressions in infix notation.

6.3.1 Language Specification

The Imperative Arithmetic language features the following capabilities:

- Unary operators: +, and -.
- Binary operators: +, -, *, and /, being - and / left-associative.
- Operator priorities: * and / precede + and -.
- Parenthesized expressions.
- Integer and floating-point literals.
- Binary operators: +, and -.
- Variables: different references to the same variable are links to the same object.

Example input and output:

- Input:

```
variables
  var a
  var b
  var c
sentences
  a = 1
```

```
b = a+2
c = (a+b)*2
output a
output b
output c
```

– Output:

```
1.0
3.0
8.0
```