Specification language

All software components developed in CoCoViLa are in essence Java classes, but they have extensions. In CoCoViLa, in order to describe the behavior of a component, one has to annotate component's Java class with a *metainterface*. Java class supplied with a metainterface is called a *metaclass*. Metainterface is a textual specification of a model of a component. It contains information about computability of variables occuring in the model. For defining a metainterface of a component, the *specification language* is used.
The metainterface specification is enclosed in Java comment symbols `/*@...@*/`, thus visible to CoCoViLa, but ignored by the Java compiler.

```
class Foo {
    /*@ specification Foo {
        …
    }
    @*/
}
```

Metainterface specification

```
MetaInterface   ::=   'specification'   SpecClassName   [InheritanceDecl]   '{'
Specification '}'
```

A metainterface starts with the `specification` keyword followed by the `SpecClassName` which should conform with the name of a corresponding Java class.

A specification consists of a number of statements, separated by a semicolon.

```
Specification ::= Statement ';' [Specification]
Statement ::=  VariableDecl | ConstantDecl | Binding | Valuation | Axiom |
Alias | Equation
```

Variables

```
VariableDecl ::= ['static'] Type IdentList
IdentList ::= Identifier [',' IdentList]

Type ::= JavaType | SpecClassName | 'void' | 'any'
JavaType ::= JavaReferenceType | PrimitiveType
Identifier ::= Letter(Letter|Number|'_')*

Variable ::= Identifier | Variable'.'Identifier
VariableList ::= Variable [',' VariableList]
```

Variables declared in the metainterface are called *interface variables* and they are not members of the corresponding Java class. The declaration of interface variables follows the Java syntax, i.e. type comes first followed by a single name or a list of distinct variable names (identifiers).
`Type` can be:

- a primitive or a reference type of the Java programming language
- a name of other metaclass declared in the same package
- '`void`' for control variables (see ...)
- '`any`' for polymorphic variables (see ...)

If a variable is declared as `static`, it acts as a global variable for the computational context, and if the value of such variable is changed in a dependent subtask (see ...), the value will remain outside of a subtask as well.
If a variable `x` has been declared in the specification `A` and a specification `B` contains a metainterface variable `a` of the type `A`, then in the specification `B` it is possible to refer to `x` of `a` as `a.x` .

Constants

```
ConstantDecl ::= 'const' JavaType Identifier '=' Value
```

In the declaration of a constant `Value` refers to a constant value in the Java programming language.
Example.

```
const double PI = Math.PI;
```

Bindings

```
Binding ::= Variable '=' Variable
```

Binding is an equality relation between variables i.e. if one variable from this relation has a value, other variable will have the same value. (For bindings of alias variables see ...)

Valuations

```
Valuation ::= Variable '=' Value
```

Valuation declares that a value of a variable exists. The type of a value must be the same as the type of a variable.

Goals

```
Goal ::= [GoalInputs] '->' GoalOutputs
GoalInputs ::= VariableList
GoalOutputs ::= VariableList
```

A goal specifies a computational problem that needs to be solved. Values of inputs are assumed to be given from the outside of a specification. If output variables of a goal are computable from (possibly empty) list of inputs, a goal is solved and the synthesized program is optimized to have only such statements that are required to compute outputs of a goal.

Axioms

```
Axiom ::= (SimpleAxiom | SubtaskAxiom) [',' ExceptionList] '{' Realization '}'
SimpleAxiom ::= VariableList '->' Variable

SubtaskAxiom ::= SubtaskList [',' VariableList] '->' Variable
SubtaskList ::= Subtask [',' SubtaskList]
Subtask ::= IndependentSubtask | DependentSubtask
DependentSubtask ::= '[' VariableList '->' VariableList ']'
IndependentSubtask ::= '[' SubtaskContext'|-'VariableList '->' VariableList ']'
SubtaskContext ::= SpecClassName

ExceptionList ::= '(' ExceptionClass ')' [',' ExceptionList]
ExceptionClass ::= JavaReferenceType
```

Axioms specify functional dependencies between variables. On the left hand of the outermost `'->'` symbol are the inputs of an axiom, while on the right hand side are the outputs. The name of a realization of a functional dependency defined by an axiom (that is, a Java method defined in the same metaclass) is given in the curly brackets. The list of inputs in an axiom is also called a *precondition* and *postcondition* denotes the list of outputs.
The precondition of an axiom may consist of variables and subtasks or it can be empty, meaning that the functional dependency can be applied without any condition. The output of an axiom is a variable corresponding to the output of the realization of an axiom. It is possible to specify possible alternative outcomes of a method (exceptions) in the postcondition by writing classes of exceptions in the parentheses. An axiom states that the realization of a functional dependency can be applied, if its precondition is satisfied, i.e. all values of input variables are computable, and, in the case of subtasks, all subtasks are solvable.
The precondition and postcondition can include one or more control variables. When a functional dependency of an axiom is applied, control variables in the output are considered to be derived.

Example. Here we show the metaclass Example where its metainterface states that the area can be computed using radius as the input parameter of the method `calcArea`, which represents the implementation of the given axiom.

```
class Example {
 /*@ specification Example {
     double radius, area;
     radius -> area {calcArea};
 }@*/

 double calcArea(double r) {
  return Math.PI*Math.pow(r, 2);
 }
}
```

From the example above it is important to notice that the interface variables are not variables of a Java class and how metainterfaces interact with Java methods (via axioms). In addition, there is a restriction that the right-hand side of an axiom cannot be empty. If a method's return type is `void`, a control variable should be used (see ...). This is necessary because each application of an axiom in the synthesis process should add new computed variables into the algorithm, otherwise such axiom will be never applied.

Example. This example shows how to specify exceptions in the metainterface. Consider the metaclass `Matrix`:

```
class Matrix {
 /*@ specification Matrix {
     int row, col, element;
     int[][] m; //matrix as array
     row, col, m -> element, (ArrayIndexOutOfBoundsException) {getElement};
 }@*/
 int getElement( int i, int j, int[][] matrix )
                              throws ArrayIndexOutOfBoundsException {
 return matrix[i][j];
}
```

Here the method `getElement` is used to compute the variable `element`. The provided method may throw `ArrayIndexOutOfBoundsException`, and to handle this situation (e.g. surround the call of a method with a corresponding `try-catch` statement during the code generation), exception class should be specified in the postcondition of an axiom.

Subtasks

```
SubtaskAxiom ::= SubtaskList [',' VariableList] '->' Variable
SubtaskList ::= Subtask [',' SubtaskList]
Subtask ::= IndependentSubtask | DependentSubtask
DependentSubtask ::= '[' VariableList '->' VariableList ']'
IndependentSubtask ::= '[' SubtaskContext '|-' VariableList '->' VariableList
']'
SubtaskContext ::= SpecClassName
```

Subtasks are used to construct loops, branching or recursion, whose control parts are given in the realization of a functional dependency. A subtask is specified in the list of inputs of an axiom and denotes a subgoal which needs to be solved before an axiom can be used.
In the synthesized program, a subtask is represented by a class, an instance of which is passed as a parameter. All subtasks implement a single Java interface

```
public interface Subtask {
   Object[] run(Object[] in);
}
```

where the body of method run is automatically generated (synthesized) by CoCoViLa. Inputs of a subtask are passed to the `run` method as array of objects. Method `run` returns an object array which is mapped to the list of outputs of a subgoal. The advantage of this approach is that it enables to pass any value without any constraint to its type. Object

type casting is done automatically in the synthesized subtask class.
There are two kinds of subtasks – dependent and independent. A subtask is called dependent, if the computations inside a subtask can depend on the context where a subtask is specified.

```
Subtask ::= IndependentSubtask | DependentSubtask
DependentSubtask ::= '[' VariableList '->' VariableList ']'
```

Example. In the specification below, axiom denotes that `b` can be computed by the method `calc` on the condition that `y` can be computed from `x`. The subtask `[ x -> y ]` is a dependent subtask and the computation of `y` from `x` depends on other relations in the given specification.

```
/*@ specification Example1 {
    int a, b, x, y;
    a = 3;
    y = x + a;
    [ x -> y ] -> b { calc };
}@*/
```

```
IndependentSubtask ::= '[' SubtaskContext'|-'VariableList '->' VariableList ']'
```

Independent subtask, on the other hand, does not depend on relations in the specification where it is declared. `SubtaskContext` specifies the context of an independent subtask (i.e. some different metainterface) on which it should be solved. Input and output variables in the declaration of a subtask are variables declared in `SubtaskContext` metainterface.

Example. In this example the usage of independent subtasks is demonstrated.

```
/*@ specification Example2 {
    int z;
    [ Example1  |- x -> y ] -> z { test };
}@*/
```

The axiom contains an independent subtask with the context `Example1` from the previous example. This means that `z` can be computed if given a value of `x`, `y` can be computed using the specification `Example1`.

Example. This example shows how a subtask is used to express a loop. The axiom with a subtask can be read as "synthesize an algorithm for computing the value of `val` from `arg` and use it to compute the value of table from the variables `from`, `step` and `to`". In the method `makeTable`, the variable `from` gives an initial value of the *for*-loop variable, `step` is the increment and `to` is the final value. The subtask instance `st` is iterated inside the loop, in each step calculating a new value of `val` which is added to the table string `result`.

```
public class Table {
    /*@ specification Table {
        double arg, val, from, step, to;
        String table;
        [arg -> val], from, step, to -> table{makeTable};
    } @*/

    public String makeTable( Subtask st, double from, double step, double to ) {
        String result = "\n";
            for ( double i = from; i <= to; i += step ) {
                Object[] out = st.run( new Object[] { i } );
                result += "i=" + i + ", out=" + out[0] + "\n";
            }
        return result;
    }
}
```

Aliases

```
Alias ::= (AliasDeclaration ['=' AliasStructure]) | AliasDefinition
AliasDeclaration ::= 'alias' ['(' Type')'] Identifier
AliasStructure ::= '(' ( VariableList | AliasWildcard )')'
AliasWildcard ::= '*.'Identifier
AliasDefinition ::= Identifier '=' '[' VariableList ']'
```

In some cases it is required to create a structure with several variables. Unlike C/C++, Java syntax does not allow creating structure types, and as it is object-oriented language, classes should be used. In the metainterfaces aliases can be used for this purpose. Alias specifies a structured variable (tuple of variables) called *alias variable*. An alias variable can only be used on the specification level and during code generation aliases are rewritten into corresponding Java syntax as arrays.
If an alias is declared with the concrete type defined in `Type`, only variables of that type can be bound by this alias, this is especially useful in automatic binding with wildcards, see the next section. Aliases can represent lightweight structures containing only variables of the same type, but also may have complex structure containing other aliases. Additionally, if an alias is used in pre- or postcondition of an axiom or a subtask, it is regarded as 1) array of a concrete type if all variables in this alias have the same type, or 2) array of type `Object[]` if variables have different types or `Type` is not defined. In the specification, a constant `<alias_id>.length` can be used to obtain the number of variables bound by an alias.

Example.

```
int x;
int y;
alias coordinates = (x, y);
```

If there are two instances `A` and `B` of the given metaclass, coordinates can be bound together, so that

```
A.coordinates = B.coordinates
```

implies the following equality of variables:

```
A.x = B.x
A.y = B.y .
```

Example. This example presents the usage of an alias as input in an axiom.

```
class TempAvg {
 /*@ specification TempAvg {
     Temp t1, t2, t3;
     double avg;
     alias (double) tempr = ( t1.t, t2.t, t3.t );
     tempr -> avg {calcAvg}; (*)
 }@*/

  double calcAvg( double[] tt ) {
    double result = 0.0;
    for(int i = 0; i < tt.length; i++)
        result += tt[i];
    return result / tt.length;
  }
}
```

Assuming that class `Temp` contains a variable `t`, metaclass `TempAvg` uses the axiom (*) for calculating average temperature, passing alias `tempr` as a parameter to the method `calcAgv` once each variable `t` is computable.
Generated code:

```
        double[] alias_tempr = new double[3];
        alias_tempr[0] = t1.t;
        alias_tempr[1] = t2.t;
        alias_tempr[2] = t3.t;
```

```
        avg = calcAvg(alias_tempr);
```

Example. This example presents the usage of an alias as output in an axiom.

```
 /*@ specification TempDistr {
     Temp t1, t2, t3;
     alias (double) tempr = ( t1.t, t2.t, t3.t );
     tempr.length -> tempr {init};
 }@*/
```

The method `init` is used to distribute initial temperatures to the variables `t` by means of alias `tempr`.
Generated code:

```
    public final int tempr_LENGTH = 3;
    public void compute( Object... cocovilaArgs ) {
        double[] alias_tempr  = init(tempr_LENGTH);
        t1.t = ((java.lang.Double)alias_tempr[0]).doubleValue();
        t2.t = ((java.lang.Double)alias_tempr[1]).doubleValue();
        t3.t = ((java.lang.Double)alias_tempr[2]).doubleValue();
    }
```

The extensive use of aliases can be made when defining visual classes – if more than one variable needs to be bound via a port.

```
Alias ::= (AliasDeclaration ['=' AliasStructure]) | AliasDefinition
AliasDeclaration ::= 'alias' ['(' Type')'] Identifier
AliasDefinition ::= Identifier '=' '[' VariableList ']'
```

Another feature of aliases is the possibility to postpone the definition of the structure of an alias. First, an alias is declared and, second, its structure is defined using another statement.

```
alias (double) x;
...
x = [ a, b, c ];
```

This is particularly useful in visual specifications with *multiports* (see ...). The specification of a component may contain only the declaration of an alias, and its structure will be defined during the composition of a visual scheme.

Wildcards

```
AliasStructure ::= '(' ( VariableList | AliasWildcard )')'
AliasWildcard ::= '*.'Identifier
```

In an alias with a wildcard a list of variables depends on the particular specification where such statement occurs. This list includes all variables of the components defined in the same specification and names of such variables are equal to the identifier specified in `AliasWildcard`. The order of components in the list is not predefined, but it remains fixed during the computations.

Example. Assuming that the specifications of `Resistor`, `Capacitor` and `Par` contain declarations of interface variables `u`, the following specification

```
Resistor r1, r2;
Capacitor c1;
Par p;
alias (double) x = ( *.u );
```

is semantically equivalent to the specification

```
Resistor r1, r2;
Capacitor c1;
Par p;
```

```
alias (double) x = ( r1.u, r2.u, c1.u, p.u );
```

Wildcards are useful when it is needed to define relations before knowing how many and which components will be used in the scheme.

Equations

```
Equation ::== Expression '=' Expression
Expression ::= [ '-' ] Term [ ( '+' | '-' ) Term ]
Term ::= Factor [ ( '*' | '/' ) Factor ]
Factor ::= Primary [ '^' Primary ]
Primary ::= Number | Variable |'(' Expression')' | Function-name '(' Expression
')'
Function-name ::= 'sin' | 'asin' | 'cos' | 'acos' | 'tan' | 'atan' | 'log' |
'exp' | 'abs' | 'toDegrees' | 'toRadians'
```

Equations are used to get rid of excessive axioms in the metainterface. In CoCoViLa, equations are solved analytically and are restricted in such a way that they cannot include occurrences of the same variable simultaneously on both sides of an equation.

Example. Instead of writing the following axioms

```
i, r -> u {calcVoltage};
u, r -> i {calcCurrent};
u, i -> r {calcImpedance};
```

with implementations required for calculating the voltage, current and impedance, a single equation

```
u = i * r;
```

can be used. From this equation, equations for calculating `i` and `r` will be derived by the equation solver. The solver does not solve systems of equations and polynomials of a degree 2 and higher.

Control variables

```
VariableDecl ::= ['static'] Type IdentList
Type ::= JavaType | SpecClassName | 'void' | 'any'
```

Control variables are the interface variables declared with the type `void`. They do not have a computational meaning, however are necessary to specify the order of execution of methods in the generated program. Initially, during the synthesis process, control variables are *not derivable*, and axioms, where control variables appear as inputs cannot be used. Once an axiom containing a control variable as a part of output is used, such control variable becomes *derivable* and axioms with this control variable as an input can be used.

Example. The purpose of the given example is to show how the control variable allows to control the execution order of methods. In the specification below we have to make sure that the method `getStatus` gets executed after the method `init`.

```
void ready;
String path;
int status;
path -> ready { init };
ready -> status { getStatus };
```

Assume the method `init` in the first axiom has a return type `void`. The control variable `ready` is used as an output of the first axiom (outputs of axioms cannot be empty). Once `path` gets a value, the first axiom is applied and `ready` becomes derivable. Only then it is possible to use `ready` as an input of the second axiom to execute the method `getStatus`.
Note: the method `getStatus` does not take any arguments, the control variable `ready` in the second axiom is only used to specify, when this axiom can be used.

Polymorphic types

Variables with type `any` provide an interesting form of polymorphism to the language - they and their components can be used in specifications of equations and axioms before the concrete type is assigned.
Example. Assume the specifications of `Resistor` and `Capacitor`, both contain variables `i,r` and `u`. Third component is `Par` which corresponds to the parallel connection of electrical elements. The metainterface of `Par` can be as follows:

```
/*@ specification Par {
    any x1, x2;
    double i,u,r;
    u = i*r;
    i = x1.i + x2.i;
    1/r = 1/x1.r + 1/x2.r;
    u = x2.u - x1.u;
}@*/
```

It is easy to see that variables `x1` and `x2` can be bound with components `Resistor` or `Capacitor` in a scheme, and the types of `x1` and `x2` can be `Resistor` or `Capacitor` accordingly. Notice that the concrete type is assigned to a variable with the initial type `any` as soon as it is bound with another variable with a concrete type.

Inheritance

```
MetaInterface ::= 'specification' SpecClassName [InheritanceDecl] '{'
Specification '}'
InheritanceDecl ::= 'super' SuperClassList
SuperClassList ::= SpecClassName [',' SuperClassList]
```

The specification language supports the inheritance of metaclasses.
There are two levels of inheritance, Java object-oriented inheritance and inheritance of metainterfaces. Metainterface of class `B` can inherit metainterface of class `A` if and only if `A` is a superclass of `B` in Java, i.e. `B` is declared as `class B extends A`. The inheritance of metainterfaces means that specifications of superclasses are unfolded into the specification of a subclass. The situation when a variable with the same name is declared multiple times in superclasses and/or in the subclass, is regarded as an error.
Example. In this example three metaclasses `Point`, `Shape` and `Rectangle` are used. `Shape` is an abstract class that contains initial coordinates and a variable `S` for square. Metaclass `Rectangle` is a subclass of `Shape`.

```
class Point {
 /*@ specification Point {
     int x,y;
 }@*/
}

class Shape {
 /*@ specification Shape {
     Point p; //initial coordinates
     double S; //square
 }@*/
}

class Rectangle extends Shape {
 /*@ specification Rectangle super Shape {
     double a,b;
     Point x2, y2;
     S = a * b;
     x2 = x + a;
     y2 = y + b;
 }@*/
}
```

Full syntax of the specification language

```
MetaInterface   ::=   'specification'   SpecClassName   [InheritanceDecl]   '{'
Specification '}'

InheritanceDecl ::= 'super' SuperClassList
SuperClassList ::= SpecClassName [',' SuperClassList]

Specification ::= Statement ';' [Specification]
Statement ::=  VariableDecl | ConstantDecl | Binding | Valuation | Axiom | Alias
| Equation

VariableDecl ::= ['static'] Type IdentList
IdentList ::= Identifier [',' IdentList]

ConstantDecl ::= 'const' JavaType Identifier '=' Value

Binding ::= Variable '=' Variable

Variable ::= Identifier | Variable'.'Identifier

Valuation ::= Variable '=' Value

Axiom ::= (SimpleAxiom | SubtaskAxiom) [',' ExceptionList] '{' Realization '}'

SimpleAxiom ::= VariableList '->' Variable
VariableList ::= Variable [',' VariableList]
SubtaskAxiom ::= SubtaskList [',' VariableList] '->' Variable
SubtaskList ::= Subtask [',' SubtaskList]
Subtask ::= IndependentSubtask | DependentSubtask
DependentSubtask ::= '[' VariableList '->' VariableList ']'
IndependentSubtask ::= '[' SubtaskContext '|-' VariableList '->' VariableList
']'
SubtaskContext ::= SpecClassName

ExceptionList ::= '(' ExceptionClass ')' [',' ExceptionList]
ExceptionClass ::= JavaReferenceType

Goal ::= [GoalInputs] '->' GoalOutputs
GoalInputs ::= VariableList
GoalOutputs ::= VariableList

Equation ::== Expression '=' Expression
Expression ::= [ '-' ] Term [ ( '+' | '-' ) Term ]
Term ::= Factor [ ( '*' | '/' ) Factor ]
Factor ::= Primary [ '^' Primary ]
Primary ::= Number | Variable |'(' Expression')' | Function-name '(' Expression
')'
Function-name ::= 'sin' | 'asin' | 'cos' | 'acos' | 'tan' | 'atan' | 'log' |
'exp' | 'abs' | 'toDegrees' | 'toRadians'

Alias ::= (AliasDeclaration ['=' AliasStructure]) | AliasDefinition
AliasDeclaration ::= 'alias' ['(' Type')'] Identifier
AliasStructure ::= '(' ( VariableList | AliasWildcard )')'
AliasWildcard ::= '*.'Identifier
AliasDefinition ::= Identifier '=' '[' VariableList ']'

Type ::= JavaType | SpecClassName | 'void' | 'any'
JavaType ::= JavaReferenceType | PrimitiveType
Identifier ::= Letter(Letter|Number|'_')*
```