An OpenModelica Java External Function Interface Supporting MetaProgramming

Martin Sjölund, Peter Fritzson
PELAB Programming Environment Lab, Dept. Computer Science
Linköping University, SE-581 83 Linköping, Sweden
{marsj, petfr}@ida.liu.se

Abstract

A complete Java interface to OpenModelica has been created, supporting both standard Modelica and the metamodeling extensions in MetaModelica. It is bidirectional, and capable of passing both standard Modelica data types, as well as abstract syntax trees and list structures to and from Java and process them in either Java or the OpenModelica Compiler. It currently uses the existing CORBA interface as well as JNI for standard Modelica. It is also capable of automatically generating the Java classes corresponding to MetaModelica code. This interface opens up increased possibilities for tool integration between OpenModelica and Java-based tools, since for example models or model fragments can be extracted from OpenModelica, processed in a Java tool, and put back into the main model representation in OpenModelica.

Keywords: Java, OpenModelica, MetaModelica, external function, abstract syntax

1 Introduction

The main goal of this work is to create a Java interface that can be used to call Modelica functions and evaluate Modelica expressions as described in [4] and [2]. More importantly, it should be possible to use the interface to analyze the abstract syntax tree of OpenModelica from a Java application and create a Java mapping of the code loaded in OpenModelica. To make this possible, the OpenModelica compiler is currently being extended to support uniontypes needed for abstract syntax tree representation. At the time of this writing, most aspects of the OpenModelica abstract syntax tree support are operational. An additional goal is calling Java methods as external Modelica functions, analogous to external C Modelica functions. An external Java interface has not previously been available for OpenModelica. Compared to previous work in Modelica-Java interfaces [6] [5], based on Dymola, this interface also supports the MetaModelica [3] extensions, giving increased possibilities for model manipulation and tool integration.

2 Motivation

External Java functions can be used either as regular Modelica functions, calculating values for models. They could also be used for displaying graphs when simulating models. External Java functions could also be used for MetaProgramming (using MetaModelica types). The external Java interface could be used internally in the OpenModelica Compiler to write functions that can walk any given AST (e.g. to create a `String` representation of any uniontype tree). It is also possible to use the AST together with tools like StringTemplate[9] to transform the AST to e.g. C code or XML. External functions could also perform operations on the AST that are slow in MetaModelica (such as appending to a list or modifying elements without copying parts of the list). However, do note that converting an AST to Java and back is a costly operation and that using external C functions or using a better algorithm might be preferred in this case. An advantage compared to external C MetaModelica functions is that the Java functions are less prone to changes due to changes in Modelica code. If the order of records in a uniontype changes, the constants used to create the uniontype record also changes. OMC does not produce header files containing these constants. External C functions need these constants, while Java functions do not. It is also easier to access the data that MetaModelica structures contain using external Java functions than external C functions because standard Java classes are used as opposed to data pointers.

For communication from Java to OpenModelica, the scenarios are different. Either you have some Meta-

Modelica code that does a transformation, but you have the data accessible in Java and you want to manipulate the result in Java. The interface allows you to either construct the corresponding Modelica data structure using Java classes, or to simply send a String. In both cases you receive a Java class corresponding to the Modelica data. The Java interface is also an Interactive Modelica session. This means you can do more manipulations and even call the OpenModelica API. As such you can access the Absyn AST for the currently loaded Modelica files, and manipulate them in Java. The existing API sends strings back and forth. With new API calls, it would be possible to send the actual AST and access it as an AST in the Java-based code.

By analyzing the code loaded in OpenModelica and creating a Java mapping of the loaded datatypes, you bring the programmer of a Java-based Modelica tool the ability to do some more extensive type checking in Java code. Instead of accessing fields by string and explicit type casting, `(ModelicaInteger)record.get("fieldA"))`, it is possible to use `record.get_fieldA()` instead.

3 Mapping of Datatypes

In order to introduce some compatibility between the Dymola and OpenModelica implementations of external Java functions, it makes sense to declare them in the same way (`'Package.Class.StaticMethod'`). The mappings between datatypes will not be the same because we'll use the same mapping when Java is the calling language as opposed to the Dymola version. By doing it this way you get a consistent interface that can also be naturally extended for MetaModelica types (`ModelicaTuple`, `ModelicaOption`). Because the full MetaModelica mapping (Table 1) uses Modelica-specific classes for all datatypes, it can't be used to call e.g. the Java method `Integer.parseInt` since it uses Java `String` and `int`. By annotating your external Java function declaration using `annotation( JavaMapping = "simple" )`, an alternative mapping (Table 2) will be used. This mapping only supports the most basic Modelica types and only one output value, but it can be used to call standard Java functions. This is a subset of the functionality that Dymola has, which also supports arrays, records and output variables that are not the return value of an external function call.

Table 1: OMC Mapping of Java Datatypes

Modelica	External Java
`Real`	`ModelicaReal`
`Integer`	`ModelicaInteger`
`Boolean`	`ModelicaBoolean`
`String`	`ModelicaString`
`Record`	`ModelicaRecord`
`Uniontype`	`IModelicaRecord`
`List<T>`	`ModelicaArray<T>`
`Tuple<T1,T2>`	`ModelicaTuple`
`Option<T>`	`ModelicaOption<T>`
`T[:]`	`ModelicaArray<T>`

Table 2: OMC Simple Mapping of Java Datatypes

Modelica	External Java
`Real`	`double`
`Integer`	`int`
`Boolean`	`bool`
`String`	`String`

If the `ModelicaRecord` datatype is represented by a `java.util.Map` from `String` to `ModelicaObject`, it follows that it can contain any datatype we use in OMC[1]. By using a `LinkedHashMap` the field keys are in the same order as they are in Modelica[2]. One advantage of this solution is that the Java mapping of a record does not depend on creating a Java class before the program is executed. The disadvantage is that you need to check that you received the correct record type, and then get the fields using the method `ModelicaObject get(String key)`. This is equivalent to performing type checking during runtime. For those who want functions to perform said typecasting, see Section 5.3 for a method that creates Java class definitions from Modelica code.

4 Calling Java External Functions from Modelica

When using external C functions, OMC translates a Modelica file (Listing 1) to a C file (Listing 2). First of all, OpenModelica copies all input variables before the external call is made since arrays (as well as variables

[1]The interface `ModelicaObject` includes MetaModelica constructs. Naming it (Meta)ModelicaObject would be more appropriate, but it isn't a valid identifier in Java.

[2]The Modelica standard enforces a strict field ordering because it is relevant for example in external C functions.

for Fortran functions) are passed by reference. Then the external call is performed and the output is copied into the return `struct` (since Modelica supports multiple output values).

Listing 1: exampleC.mo

```
function logC
  input Real x;
  output Real y;
external "C" y = log(x);
end logC;
```

Listing 2: logC.c

```
logC_rettype _logC(modelica_real x)
{
  logC_rettype out;
  double x_ext;
  double y_ext;
  x_ext = (double)x;
  y_ext = log(x_ext);
  out.targ1 = (modelica_real)y_ext;
  return out;
}
```

When using external Java functions, OMC should generate a C file that is similar to the ones generated by external C functions. External Java calls translated the variables to Java objects, and fetch the correct method from the JVM through the Java Native Interface (JNI). The flow of data in Figure 2 is explained in detail below. Before the call, each argument is translated to a JNI `jobject` (i.e. a C pointer to a Java class) and then after copying the result back to the respective C variable. This ensures that the code works in the same way as external C (and thus the “correct” Modelica behaviour). Compare the C file for external C (Listing 2) to the one for external Java (Listing 4, generated by the Modelica code in Listing 3). The Java code is essentially the same with the difference being that instead of one line of code for an external call, it is 17 lines of code to set up the Java call properly.

Listing 3: exampleJava.mo

```
function logJava
  input Real x;
  output Real y;
external "Java"
  y = 'java.lang.Math.log'(x)
  annotation(
    JavaMapping="simple"
  );
end logJava;
```

Listing 4: logJava.c

```
logJava_rettype _logJava(modelica_real
    x)
{
  logJava_rettype out;
  double x_ext;
  double y_ext;
  JNIEnv* __env = NULL;
  jclass __cls = NULL;
  jmethodID __mid = NULL;
  jdouble x_ext_java;
  jdouble y_ext_java;
  x_ext = (double)x;
  __env = getJavaEnv();
  x_ext_java = x_ext;
  __cls = (*__env)->FindClass(__env, "
      java/lang/Math");
  CHECK_FOR_JAVA_EXCEPTION(__env);
  __mid = (*__env)->GetStaticMethodID(
      __env, __cls, "log","(D)D");
  CHECK_FOR_JAVA_EXCEPTION(__env);
  y_ext_java = (*__env)->
      CallStaticDoubleMethod(__env,
      __cls, __mid, x_ext_java);
  CHECK_FOR_JAVA_EXCEPTION(__env);
  y_ext = y_ext_java;
  (*__env)->DeleteLocalRef(__env, __cls
      );
  out.targ1 = (modelica_real)y_ext;
  return out;
}
```

5 Calling Modelica Functions from Java

OpenModelica communicates with other tools through sockets or CORBA using its Interactive module. The Java interface can do the same, just as the Eclipse plugin (MDT) does. Figure 1 shows the existing Java-OpenModelica communication using CORBA. The OMCProxy class does not only communicate with OMC using CORBA. It also starts OMC in server mode if it can’t find a server to communicate with.

Figure 1: CORBA Communication

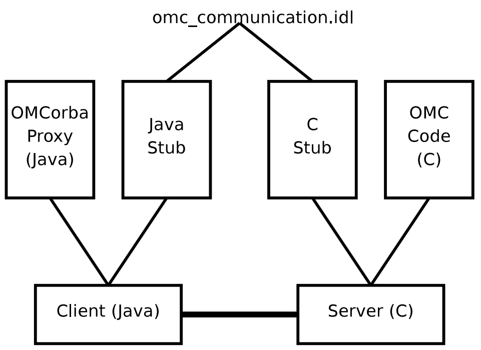


Figure 2: External Java Call (Data Flow)

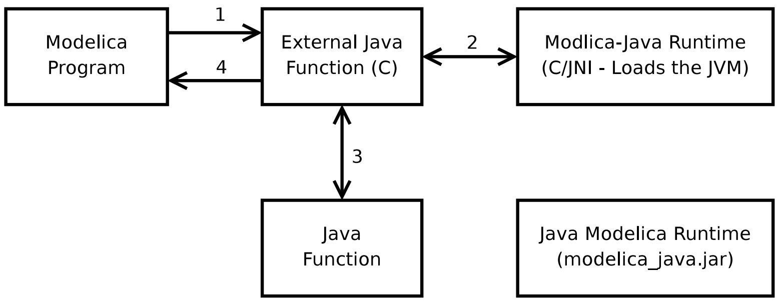
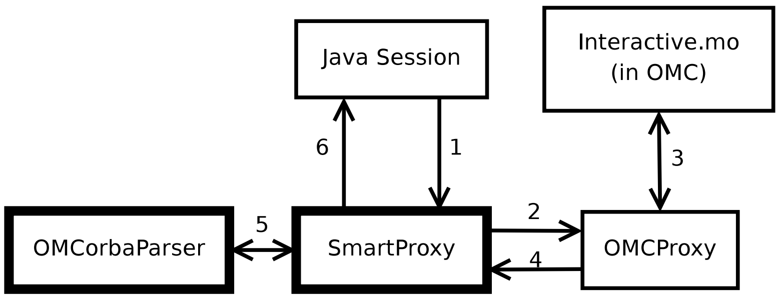

Modelica Program
1
4
External Java Function (C)
2
Modlica-Java Runtime (C/JNI - Loads the JVM)
3
Java Function
Java Modelica Runtime (modelica_java.jar)

Figure 3: Interactive Java Session (data flow)



The marked nodes in Figure 3 are what have been added on top of OMCProxy. SmartProxy only glues OMCProxy and OMCorbaParser together, so the user doesn't need to be aware that those classes exist. The CORBA interface is an untyped string-to-string function which means you can send `{1,2.0,3}` even though the Modelica standard disallows mixed types in arrays [4] [7].

Listing 5 contains an example of an interactive OpenModelica session. The user tells OMC to add a record definition to the AST, and then calls the record constructor. The result is a record.

Listing 5: Interactive OMC Session

```
>> record ABC Integer a;Integer b;
    Integer c; end ABC;
{ABC}
>> ABC(1,2,3)
record ABC
    a = 1,
    b = 2,
    c = 3
end ABC;
```

5.1 Mapping Textual Representations of MetaModelica Constructs to Java

All Modelica objects implement the dummy Java interface `ModelicaObject`, which helps tagging any Modelica data. Table 1 contained the mappings from Modelica types to Java types. The problem with the CORBA interface is that the textual representations are ambiguous. `{1,2,3}` can represent either a MetaModelica list or a Modelica array. `(1,2,3)` can represent either a MetaModelica tuple or multiple function output values. This implementation will treat both cases in the same way. `{1,2,3}` is represented by `ModelicaArray` while `(1,2,3)` is represented by `ModelicaTuple`. Both of these classes extend `java.util.Vector` (which supports both random access and implements the `List` interface).

5.2 Parsing CORBA Output

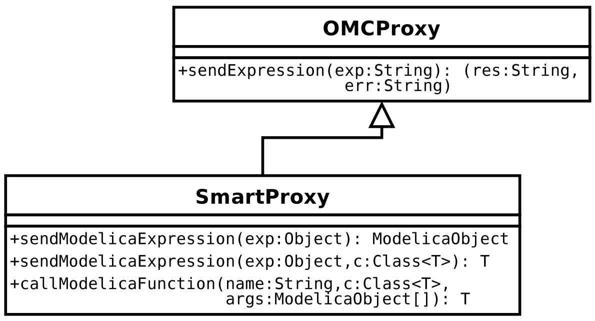
In order to create a reasonably efficient and maintainable parser ANTLRv3 [8] is used to parse the results from the Interactive interface. ANTLRv2 has been used in other parts of OpenModelica with good results, so the choice of parser was quite easy.

What you end up with at this point is an interface that can call Modelica functions, pass Modelica structures and cast the results to the expected type.

This parser translates strings parsed over the OpenModelica interactive interface to the basic Java classes. For example, records are translated to a "generic" record that uses the map interface instead of accessing fields more or less directly). This means you have to write wrapper classes if you want to access these fields without typing lots of code.

When sending an expression from Java to OpenModelica you get back a Java `ModelicaObject`. But if you already know the return type, you don't want to create a lot of code just to cast that object to the expected class. For this reason the Java call `sendModelicaExpression` (and related functions, see Figure 4) can take a Java `Class<ModelicaObject>` and after it has parsed the returned data, the function will attempt to cast the object to the expected class. Should

Figure 4: CORBA Communication Proxies


OMCProxy
+sendExpression(exp:String): (res:String, err:String)
SmartProxy
+sendModelicaExpression(exp:Object): ModelicaObject
+sendModelicaExpression(exp:Object,c:Class<T>): T
+callModelicaFunction(name:String,c:Class<T>, args:ModelicaObject[]): T

the cast fail, it will also try to construct a new object of the return type using the object as the argument. Thus, all classes implementing `ModelicaObject` have a constructor taking a single `ModelicaObject` (where it will determine if the object is indeed a supertype of the expected type). This is because any record is parsed as a generic `ModelicaRecord` rather than e.g. `ExpressionRecord`. The `ExpressionRecord` constructor should analyze the `ModelicaObject` and determine if it is indeed a `ModelicaRecord` with the correct record name, field names and data types in the fields. The process of creating this class can be done automatically, see Section 5.3 for details on the implementation.

5.3 Translating MetaModelica Definitions to Java Classes

Since it would be nice to translate MetaModelica AST definitions to Java AST definitions, in the form of a Java JAR file, a second parser was created. This parser is to be used prior to the application development since it tells OMC to load a number of Modelica files and return an AST containing type definitions, functions, uniontypes and records of the files. Extracting the AST is done by a new API call, `getDefinitions`, in the OpenModelica compiler Interactive module, `Interactive.mo`. The output of the call is a tree in textual prefix notation, similar to LISP syntax. It contains a partial extraction of the syntax tree from the `Absyn` module. Note that the OpenModelica Interactive module uses the `Absyn.Program` AST and not the lowered intermediate tree `SCode.Program` or `DAE` ASTs. Because the AST may contain errors (type checking, syntax, etc), you may get some cryptic error messages in programs containing errors in for example unused functions since RML only compiles referenced functions. The textual extraction format is as follows (Modelica code to textual format to Java code):

5.3.1 Packages

Modelica packages are used to place its parts in its corresponding Java packages.

Modelica: `package myPackage; ... end myPackage;`

Intermediate: `(package myPackage ...)`

5.3.2 Type aliasing

In the example below, all occurrences of `myInt` will eventually be replaced by `ModelicaInteger`. The reason is that Java does not support type aliasing.

Modelica: `type myInt = Integer`

Intermediate: `(type myInt Integer)`

Java: `ModelicaInteger`

5.3.3 Records

Records are transformed into Java classes extending ModelicaRecord. The class has set and get functions for each field in the record. Fields of any extended records are looked up. The Java class will not inherit from a base record class because multiple inheritance is disallowed.

Listing 6: Modelica Record

```
record abc
```

```
  extends ab;
  Integer c;
end abc;
```

Intermediate: `(record abc (extends ab) (Integer c))`

Java: `class abc extends ModelicaRecord ...`

5.3.4 Replaceable Types

Replaceable types are handled using Java generics.

Modelica: `replaceable type T subtypeof Any`

Intermediate: `(replaceable type T)`

Java: `<T extends ModelicaObject>`

5.3.5 Uniontypes

Uniontypes are tagged using interfaces.

Listing 7: MetaModelica Uniontype

```
uniontype ut
  record ab
    Integer a; Integer b;
  end ab;
  record bc
    Integer b; Integer c;
  end bc;
end ut;
```

Intermediate: `(uniontype ut) (metarecord ab 0 ut (Integer a) (Integer b)) (metarecord bc 1 ut (Integer b) (Integer c))`

Listing 8: MetaModelica Uniontype (Java)

```
interface ut extends IModelicaRecord {
}
class ab extends ModelicaRecord
    implements ut {
    ...
}
class bc extends ModelicaRecord
    implements ut {
    ...
}
```

5.3.6 Functions

Functions are translated to classes extending `ModelicaFunction`. The method `call` performs the actual function call over the CORBA interface. Functions with multiple return values have two call methods, one that returns a `ModelicaTuple` and one that performs a call-by-reference.

Listing 9: Modelica Function

```
function add
  input Integer lhs;
  input Integer rhs;
  output Integer out;
algorithm
  out := lhs+rhs;
end add;
```

Intermediate: `(function abc (input Integer lhs) (input Integer rhs) (output Integer out))`

Listing 10: Modelica Function (Java)

```
class  add extends ModelicaFunction {
   ...
   ModelicaInteger call(ModelicaInteger
       lhs, ModelicaInteger rhs) {
     ...
   }
}
```

5.3.7 Partial Functions

Partial functions are undefined function pointers (can also be seen as as types). The Java implementation is essentially an identifier (it discards the in/output).

Listing 11: MetaModelica Partial Function

```
partial function addFn
  input Integer lhs;
  input Integer rhs;
  output Integer out;
end addFn;
```

Intermediate: `(partial function addFn)`

Java: `new ModelicaFunctionReference("addFn")`

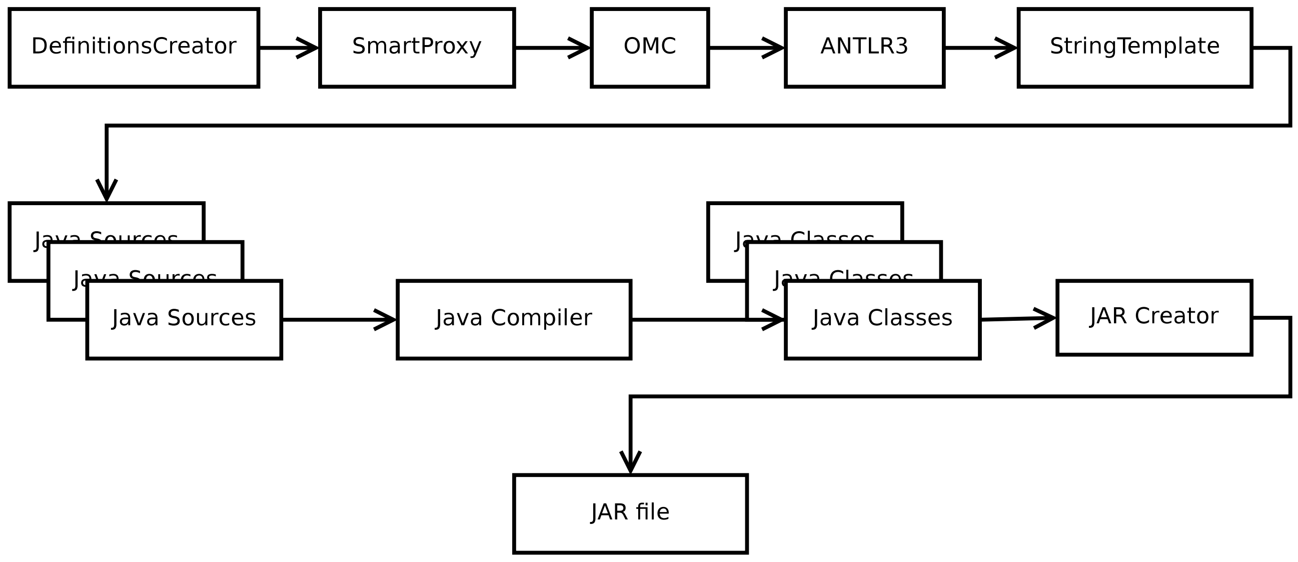
5.4 Translating Two Modelica Functions to Java Classes

The number of steps required to translate a Modelica file into a JAR-file containing all of the definitions is quite large. Figure 5 shows the flow of data and the steps are explained through a simple example. The Modelica code in Listing 12 will be used as the example for the translation from Modelica code to Java classes.

Listing 12: Modelica source to be translated to Java

```
package Simple
```

Figure 5: DefinitionsCreator data flow



```
function AddOne
  input Integer i;
  output Real out;
  Integer one = 1;
algorithm
  out := i+one;
end AddOne;

function AddTwo
  input Integer i;
  output Integer out1;
  output Integer out2;
algorithm
  out1 := i+1;
  out2 := i+2;
end AddTwo;
end Simple;
```

The process starts when you invoke `DefinitionsCreator`. Listing 13 shows how to create `~/examples/simple.jar` (with package prefix `org.openmodelica.example`) from `~/examples/Simple.mo`. The inner workings of the class are described below.

Listing 13: Invoking DefinitionsCreator

```
$ java -classpath $OPENMODELICAHOME/
   share/java/antlr-3.1.3:
   $OPENMODELICAHOME/share/java/
   modelica_java.jar org.openmodelica.
   corba.parser.DefinitionsCreator ~/
   examples/simple.jar org.openmodelica
   .example ~/examples Simple.mo
```

The string representation of the definitions in the AST returned by OMC is:

Listing 14: getDefinitions String corresponding to the Modelica functions

```
(package Simple
(function AddOne
  (input Integer i)
  (output Real out))
(function AddTwo
  (input Integer i)
  (output Integer out1)
  (output Integer out2))
)
```

By using the OMCorbaDefinitions ANTLRv3 grammar [8] and StringTemplate templates [9], Java source files (Listings 15 and 16) corresponding to the definitions are created.

Listing 15: Corresponding Java source for AddOne

```
public class AddOne extends
   ModelicaFunction {
  public AddOne (SmartProxy proxy) {
    super("AddOne", proxy);
  }
  public  ModelicaReal call (
      ModelicaInteger i) throws
      ParseException,ConnectException
  {
    return proxy.callModelicaFunction("
        Simple.AddOne", ModelicaReal.
        class, i);
  }
}
```

Listing 16: Corresponding Java source for AddTwo

```
public class AddTwo extends
    ModelicaFunction {
  public AddTwo (SmartProxy proxy) {
    super("AddTwo", proxy);
  }
  public ModelicaTuple call (
      ModelicaInteger i) throws
      ParseException,ConnectException
  {
    return proxy.callModelicaFunction("
        Simple.AddTwo", ModelicaTuple.
        class, i);
  }
  public void call (ModelicaInteger i,
       ModelicaInteger out1,
      ModelicaInteger out2) throws
      ParseException,ConnectException
  {
    ModelicaTuple __tuple = proxy.
        callModelicaFunction("Simple.
        AddTwo", ModelicaTuple.class, i)
        ;
    java.util.Iterator<ModelicaObject>
        __i = __tuple.iterator();
    if (out1 != null) out1.setObject(
        __i.next()); else __i.next();
    if (out2 != null) out2.setObject(
        __i.next()); else __i.next();
  }
}
```

The Java files are compiled using `javac`, the Java Compiler. They are then archived using the `java.util.jar` class. Because StringTemplate is used, the code could potentially be re-targeted in order to create for example C# definitions, but the Java compilation and JAR steps would need to be replaced with functions that could handle C#.

6 Limitations

The implementation requires access to the OpenModelica CORBA interface or external functions generated by OpenModelica. As such, OpenModelica needs to be fully bootstrapped before the interface can be used internally in OpenModelica. At the moment, it can be used for simple Modelica/MetaModelica programs.

Java is quite limited when it comes to generics. Generics in Java is just something that helps the programmer do static type checking. In running code, Java has no concept of generic types and is totally unchecked. This is one of the reasons why `ModelicaTuple` is untyped in Java.

7 Related Work

Dymola has the capability to call Modelica functions and the Dymola API from external Java functions [5]. Their approach was to use a single entry-point (com.dynasim.dymola.interpretMainStatic). This is probably a bit faster than passing and parsing strings, and would have been possible to accomplish in OpenModelica as well. The OpenModelica CORBA interface is more akin to the Dymola external interface described in [6]. It also uses strings to communicate with applications, but can construct some native types for example when communicating from Modelica to Matlab.

8 Future

The OpenModelica compiler is currently being extended to support the datatypes introduced in MetaModelica needed to represent and communicate abstract syntax trees. Another planned extension is to replace the current text-based CORBA interface with a directly linked version, giving higher performance.

As work progresses, support for new datatypes needs to be added in the Interactive module since the CORBA interface depends on this module being updated. Most of the work so far has been limited to compiling code using these datatypes (e.g. the union-type implementation [1]).

9 Conclusions

A complete bidirectional Java interface to OpenModelica including support of the MetaModelica language extensions has been created. It is capable of passing basic and structured data types including syntax trees to and from Java and process them in either Java or the OpenModelica Compiler. It uses the existing CORBA interface and is capable of automatically generating the Java classes corresponding to MetaModelica code. This new interface opens up new possibilities for tool integration and model manipulation.

References

[1] Björklén S. Extending Modelica with High-Level Data Structures: Design and Implementation in OpenModelica. Linköping, Sweden: Master's thesis, Department of Computer and Information Science, Linköping University, 2008.

[2] Fritzson P. Principles of Object-Oriented Modeling and Simulation with Modelica 2.1, 940 pages. Wiley-IEEE Press, 2004.

[3] Fritzson P. MetaModelica Programming Guide, June 2007 draft. http://openmodelica.org/.

[4] Modelica Association. The Modelica Language Specification Version 3.0, September 2007. http://www.modelica.org/.

[5] López J.D., Olsson H. Dymola interface to Java - A Case Study: Distributed Simulations. In: Proceedings of the 5th International Modelica Conference, Vienna, Austria, 4-5 September 2006.

[6] Olsson H. External Interface to Modelica in Dymola. Proceedings of the 4th International Modelica Conference, Hamburg, Germany, 7-8 March 2005.

[7] OpenModelica. OpenModelica System Documentation, January 2009. http://openmodelica.org/.

[8] Parr T. ANTLR Parser Generator. http://antlr.org/.

[9] Parr T. StringTemplate Template Engine. http://stringtemplate.org/.