Proceedings
of the 4th International Modelica Conference,
Hamburg, March 7-8, 2005,
Gerhard Schmitz (editor)

H. Olsson
Dynasim AB, Sweden
External Interface to Modelica in Dymola
pp. 603-611

Paper presented at the 4th International Modelica Conference, March 7-8, 2005,
Hamburg University of Technology, Hamburg-Harburg, Germany,
organized by The Modelica Association and the Department of Thermodynamics, Hamburg University of Technology

All papers of this conference can be downloaded from
http://www.Modelica.org/events/Conference2005/

External Interface to Modelica in Dymola

Hans Olsson
Dynasim AB
Research Park Ideon
S-223 70 Lund, Sweden

Introduction

Dymola [1] provides an integrated environment for modeling, simulation and scripting based on the Modelica language. However, Modelica is not always the best choice for GUI-design, database access, or canned presentations of libraries (in the last case the usual choices are PowerPoint® presentations, pdf-documents, or animation files).

The solution in these cases is to leverage other tools, such that they can interface *to* Dymola's Modelica functionality and vice-versa for call-backs/links *from* Dymola to external tools. These solutions are already implemented for the forthcoming version Dymola 6, except for a few specific items described as future plans (these might be implemented in time for the release of Dymola 6). Some solutions are also available in previous versions of Dymola as described in the manual [1].

1 Model callbacks in Dymola

One goal of the external interface is to allow a model developer to customize commands for the model by calling external tools. This functionality is provided in two separate parts:

- External tools can be interfaced to any Modelica function.
- Models can be customized to have commands calling Modelica scripts or functions in Dymola.

As a concrete example a model developer can add model-specific commands to Dymola's Commands-menu. Users of the model can then call a command from the Commands-menu, which for example executes a Modelica script calling external Modelica functions implemented in C.

1.1 Variants of commands

The commands can be generic (independent of the selected model, e.g. to select a specific demo model or check that the model fulfils some guidelines), or specific to the currently selected class (e.g. post-process the simulation result).

The command can be called explicitly by the user (from the Commands-menu in Dymola), or implicitly to extend existing functionality (the enable-field in parameter-dialogs is one example where a user can enable the input field based on a predicate call-back). In the future, callbacks will be generated at specific stages of translation, e.g. for users to gather additional statistics of the use of specific models.

Obviously we could for a specific example provide the functionality inside Dymola, but by providing an API and callbacks we allow the customer to extend Dymola. Thus the API to Modelica structure which is presented later in the paper is intended to also be useful for e.g. gathering statistics about the components used in the translated model. Some of these functions need access to browser information (such as the current class) as will be discussed below.

1.2 Calling functions directly

A specific case of running a command is to call a function related to the model e.g. to run an optimization of the model.

The advantage with directly using function calls is that there is no need for any model-specific script files (making it easier to e.g. copy the Modelica model) and that a function call is part of the Modelica language and thus syntactically correct.

Furthermore it is possible to optionally prompt the user to modify the arguments of the function call before the function is called, e.g. to specify the operating point to optimize for. This uses the normal (and extensible) function call dialog.

2 Communication protocols

The interprocess communication is between two running programs one of which is Dymola. The

transport mechanism can be seen as separate from the structure of the messages. Currently Dymola can send and receive DDE-messages. For the future extension of exchanging XML-data in SOAP-encoding SOAP-HTTP is a suitable alternative [2], and is portable to non-Windows platforms (for which the demand is increasing).

2.1 DDE-interface

For DDE-execute, the return value does not allow meaningful result values and thus specially formatted DDE-Request(s) is used for returning data to other programs with a special case for Matlab (below).

Thus Windows programs can call DDE-routines to e.g. perform a parameter sweep from Excel (by programming in Visual Basic for Application). In this case the update of the excel spread-sheet is done by running a macro and there are no links in the Excel-document, which only contains the start-values (J1.w), parameter-values (J2.J), and final values for two variables (J1.w and J2.w):

TestCommunicateWithExcel.xls

	A	B	C	D	E	F
1	CoupledClutches	J1.w	J2.J	=>	J1.w	J2.w
2		1	1		0.134935	0.288287
3		1	2		0.040615	0.238399

Figure 1 Parameter-sweep from Excel

The macro opens a DDE-channel to Dymola, sends the command simulateExtendedModel as a DDE-request to simulate and get the final values of variables. To access the spread-sheet the Excel-routine `Sheets( "Sheet1").Cells(r, c).Value` is used to get and set the values.

It is also possible to use DDE to communicate directly with a running simulation, Dymosim, (provided the compiler option 'Visual C++ with DDE' is selected). This is described in [1] and also allows automatic update of variables after changes.

These protocols are extensible which allows calls between two programs running on different computers, i.e. remote procedure calls. Although remote procedure calls are beneficial, for security reasons remote procedure calls must be explicitly enabled (as is necessary for remote DDE).

Limitations of DDE

Unfortunately DDE has some restrictions (in addition to being platform-specific), in particular on the maximum length of the messages, and no general high-level API for communicating structured data.

For communication with a running simulation (DDE-communication between Dymola and Dymosim) we have found it necessary to use special formats to achieve the high bandwidth needed for e.g. online animation of Modelica models, while respecting the limitations of the protocol.

We do not anticipate similar bandwidth needs for the communication with Dymola, since the natural way of communicating a vector of values is to send it as *one* DDE-message (which automatically solves most of the performance problems).

The DDE-interface in the caller is preferably written as one generic routine (as we have done in Matlab) to make it easy to later extend it e.g. with handling of messages exceeding the maximum length, and optimized alternatives to the CF_TEXT-format.

2.2 Direct interfacing

The above handles the complex case of interprocess communication between two running programs, but sometimes a simpler mechanism suffices.

2.2.1 Call of external functions

The Modelica language offers the possibility to directly interface to C and FORTRAN-functions such that calls of Modelica functions declared as external C/Fortran are mapped into calls of the corresponding C or Fortran functions.

There is no restriction on the use of external functions in Modelica and to allow easy use of them in the interactive environment Dymola performs demand-compilation of external functions. Thus a user can call external functions in the same way as non-external functions.

This C interface provides an extensible mechanism that also handles other languages that can give routines C linkage, such as C++ and languages that provide an interface for calls from C, such as Java. Since the JNI interface to Java allows dynamic loading of Java-libraries this could be done internally in Dymola making it possible to directly call a Java function from Dymola to e.g. show a modal dialog and get the user response back without using any external programs.

Below we demonstrate running a Java function showing a modal dialog box, where the call of the Java-functions has been included as an external function call in Modelica (with suitable arguments), and then compiled by Dymola (the JNI-implementation require that calls in translated C-code use the Visual C++ compiler).

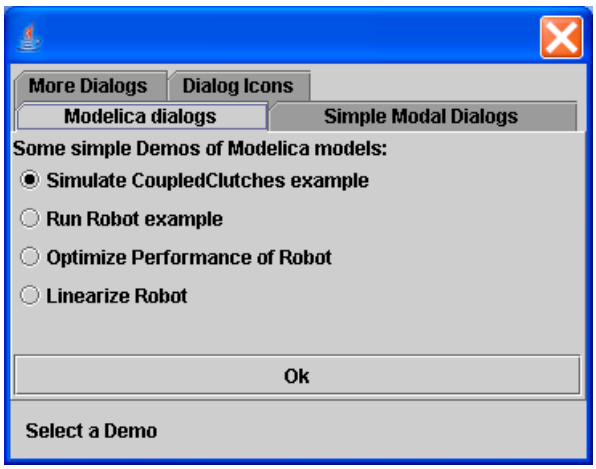

Figure 2 Calling a modal dialog in Java from a Modelica function

Extending Modelica's external interface to directly include Java in addition to C and Fortran 77 is straightforward and the specification was deliberately written to allow such extensions. By using single-quoted names it is possible to directly support hierarchical external function name (i.e. containing a dot) as in Java.

2.2.2 Linking to libraries

It is also possible to link with specific C-libraries (including Dynamic Link Libraries, DLLs). Due to the limitations of the C-compilers used, any libraries must be provided in a binary format compatible with the C-compiler used to compile the Modelica code in Dymola (Visual C++ 6/7, egcs, or Watcom). Provided the external code is portable and available in source-form that is in general possible. Additionally Modelica models are often downloaded and run on realtime platforms, which require different libraries (or that the C-code is provided in source-form and linked together with the model).

Another potential problem on Windows is that some API calls require that the caller is a Windows-program.

This is e.g. necessary to use the DirectX interface from Microsoft. An application of DirectX is to allow users to control a car-model from VehicleDynamics [6] by a steering wheel. In those cases a Windows program must be generated (in Dymola this currently requires that you select the compiler Visual C++ with DDE) and special routines obtain the window handles.

2.2.3 Calling programs

Modelica.Utilities.System [3] enables functions written in Modelica to call external programs. Command line arguments to Dymola enable external tools to for instance run simulations in Dymola.

The Commands-menu, Dymola's Execute-function, and links in the documentation layer also allows opening other files than Modelica scripts using the file associations in Microsoft Windows. This is useful for canned presentations, and selecting a menu entry will automatically open the file in the corresponding tool (e.g. pdf-documents in Acrobat Reader®, animation files in the media player, html-files in the browser).

3 Data-structure encoding

To communicate Modelica data-structures in Dymola to other tools the data-structures must be mapped into other data-structures. Following the C and FORTRAN-interface this is defined in a generic encoding for each interface, i.e. there is no need to specify a mapping for each data-structure. If a special mapping is desired for a specific case that can then be done either in a Modelica function or in the other tool.

The basic idea of the interface is to return a string that when evaluated returns the value, e.g. a numeric value is returned as itself, i.e. 3/2 is returned as the string '1.5' (without the quotes).

For more advanced data-structures, arrays and records, it is necessary to define how the resulting string is evaluated. The two variants that are implemented are Modelica data-types constructors and Matlab.

3.1 Mapping to Modelica

The Modelica-mapping is identical to the output format used in Dymola's command-window and makes use of record and array constructors for complex data-structures. Consider the examples (input in bold and the response-string is given after the '='):

```
Matrices.inv([1,2;3,4])
=
[(-2.0), 1.0;
1.5, (-0.5)]

GetClassAttributes("Modelica")
 = Dymola.AST.ClassAttributes(
        fullName = "Modelica",
       isPartial = false,
     isProtected = false,
      restricted = "package",
```

```
          isInner = false,
          isOuter = false,
  isEncapsulated = false,
    isShortClass = false,
   isReplaceable = false,
    isRedeclared = false
)
```

The mapping is sufficiently straightforward that we will not go into details of it, and if the result is pasted into Dymola's command input and evaluated it returns the same result once more.

To use this functionality the application programmer has to set up a DDE-channel to Dymola, and send a Modelica-function call as string in a DDE-request. Dymola's DDE-server will respond with a string containing the result as a Modelica data-structure.

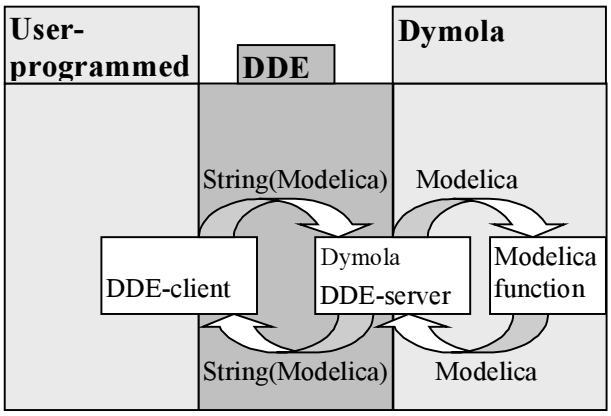

Figure 3 Using Dymola's interface to Modelica functions from other programs.

For this to be possible, all data-structures in Modelica must have an output format that when evaluated gives the data-structure back. This seems straightforward, but the problem is empty arrays, since Modelica as a strongly typed language does not allow {} for empty arrays.

For primitive types a work-around is to use the fill-operator. For an empty array of records this requires using the record-constructor, but in many cases the record constructor does not have defaults for all elements and thus cannot be called without specifying arguments. A future extension of Modelica would be to allow calls of the record constructor with no arguments in this specific case.

3.1.1 Grammar for Modelica subset

We have defined a subset of Modelica for representing any structured data values, that is primitive types, arrays, and records.

The advantage of this format is that the mapping is self-explanatory, complete for Modelica data-types, and to be able to parse the Modelica-format it is only necessary to implement a parser for a subset of expressions from the Modelica grammar:

```
expression:
  primary
| "-" primary

primary :
  UNSIGNED_NUMBER
| STRING
| false
| true
| component_reference  function_call
| "[" expr_list { ";" expr_list } "]"
| "{" expr_list "}"

component_reference :
  IDENT [ "." component_reference ]

expr_list :
  expression { "," expression }

function_call :
  "(" [ function_args ] ")"

function_args :
  expression [ "," function_args  ]
| named_args

named_args:
  named_argument [ "," named_args ]

named_argument: IDENT "=" expression
```

Some of the names in the grammar have been shortened to keep the grammar elements on one line.

The reason to keep component_reference and function_call is to use record constructors to build record data-structures (using named arguments). Function call without named arguments (the line in italics) is only needed for the above-mentioned use of fill to construct empty arrays.

3.2 Mapping for Matlab

Dymola can automatically map data-structures *to* Matlab data-structures. They are first encoded in a string, that is then automatically evaluated by the Matlab-interface to the corresponding data-structure.

Dymola/Modelica	Matlab
Real	double
Integer	double
String	string
Enumeration(planned)	string
Boolean	double
Array	matrix or cell array
Record member	struct member

Figure 4 Mapping to Matlab

Array results are returned as matrices, except array of records and array of strings that are returned as cell arrays.

This provides a complete interface from any data-structure (i.e. return-value) defined in Modelica to a corresponding data-structure in Matlab. This includes the Modelica class and component structure as will be defined later.

The interface for sending requests from Matlab to Dymola cannot provide a similar feature based on the data structures in *Matlab*. The reasons are that Matlab does not distinguish between scalars, vectors and matrices (i.e. ndims in Matlab is always >=2), and that Modelica lacks a counterpart to Matlab's struct, i.e. an untyped record constructor.

However, as will be discussed in a following section an API to the class structure is available and the caller routine in Matlab (dymolaCall) has been extended with code to perform this mapping based on the declaration of the *called* Modelica function. Thus the Matlab-programmer only has to call dymolaCall with the name of the Modelica function and arguments as Matlab data-structures (arrays and structs). Dymola and dymolaCall and internally handle this and the result is a Matlab data-structure.

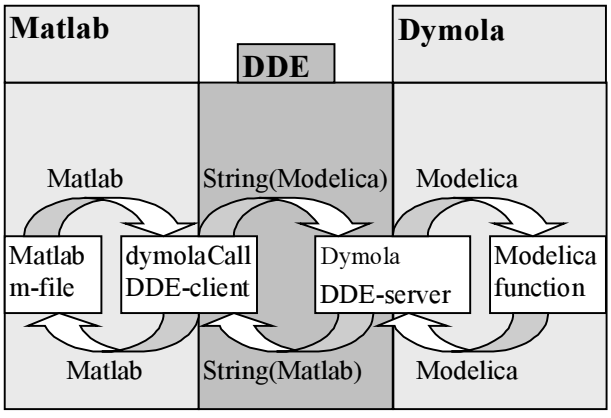

Figure 5 Interface between Matlab and Dymola

This does not include some of the advanced features of Modelica, e.g. the mapping does not automatically handle vectorized arguments to functions or allow you to use named arguments from Matlab.

We have not yet found any performance issues with this interface, but the m-file could be improved to locally cache the Modelica class structure in order to avoid sending the same query several times to Dymola (as will happen with e.g. arrays).

3.2.1 Examples

The following examples are only intended to demonstrate the possibilities and that strings, arrays of doubles, and records (containing strings and booleans) are returned (running from Matlab). The first examples demonstrate sending the entire command as one string:

```
>> dymolaCall('"Hello"+" world"')
ans =
Hello world

>> dymolaCall(…
'Modelica.Math.Matrices.inv([1,2;3,4])')
ans =
  -2.0000    1.0000
   1.5000   -0.5000
```

As indicated above the interface also allows call with function arguments as Matlab data-types (the second argument can optionally be used to specify an existing DDE-channel):

```
>> dymolaCall('Modelica.Math.Matrices.inv',[],..
[1,2;3,4])
ans =
  -2.0000    1.0000
   1.5000   -0.5000

dymolacall('GetClassAttributes',[], 'Modelica')
ans =
        fullName: 'Modelica'
       isPartial: 0
    isProtected: 0
      restricted: 'package'
         …
```

```
>>dymolaCall(…
'Modelica.Math.Matrices.leastSquares',[],…
[1;0;1],[1;2;3])
ans =
    2
```

We will later return to how this uses the API to the Modelica class structure to construct the call.

3.3 Mapping for XML

For the abstract syntax tree one mapping to XML [4] is defined in [5]. A future mapping of data-structures to XML could use a subset of this by viewing them as a function call/expression in this structure (i.e. similar to the subset for the mapping to Modelica).

This can be viewed as too generic and another possibility is to automatically construct a specific document type declaration defining the grammar for the specific Modelica data-structure(s), i.e. one for each record class used, and placing this first in the XML-file [4]. This might still be combined with an external pre-defined data-type declaration for the built-in Modelica types, i.e. Boolean, Integer, Real, and String.

4 API to Modelica class structure

The first problem with defining an API to the Modelica class and component structure is that it is not possible to define a data-structure for the entire class structure in Modelica (at least not as implemented in Dymola), because the class structure is inherently recursive. However, even if it *were* possible to replicate the entire class structure as a set of nested records it would not provide an efficient interface to the class structure for simple queries or modifications.

Therefore we have instead defined access routines that allow tree walking to be built in Modelica (in the future there will also be corresponding routines for modifying and adding classes and components).

4.1 Basic design of API

In order to provide a useful interface to the classes and components three sets of routines were provided as follows in package Dymola.AST. Originally each set was only comprised of two functions and one record, and the intention is to further extend this (e.g. with routines for modifying the elements).

The three sets of routines are for classes, extends-clauses and components. In each set there is a routine for obtaining the elements (as an array), a record defining the "attributes" (protected, inner, full class name, …) and a routine for getting the attributes for a specific element.

These interfaces assume that one can use the name of elements in the queries, which is possible in the cases above (technically excluding the obscure case of repeated identical extends-statements which is legal Modelica, but without any reasonable use). Note that Dymola enforces this semantic restriction in Modelica already during parsing of classes, and thus it is safe to base the API-routines on this assumption.

The requirements also include access to the import-statements in the class. For import-clauses it is hard to define which name to use as a key (when considering both the qualified and the unqualified import-statements, thus a combined routine has been added that returns an array of records defining the import-statements.

This was found to provide such an increase in ease of use that similar routines were added for the other cases. These were trivial to implement based on the existing routines, and we give a full example below (excluding its documentation):

```
function ComponentsInClassAttributes
   "Get components of a class"
   input String className;
   output ComponentAttributes res[:]=
    GetComponentAttributes(className,
       ComponentsInClass(className));
algorithm
end ComponentsInClassAttributes;
```

Here the names of the components is constructed by ComponentsInClass and this is then used in a vectorized call (as defined in Modelica [3]) of GetComponentAttributes to get the attributes of all components.

Thus functions exists for all elements of table given on the next page (where "elements in class" has a class/ package as input and get attributes also exist in a form that returns an array containing the attributes of all elements).

	Record of attributes	Elements in class	Get attributes
Classes	ClassAttributes	ClassesInPackage	GetClassAttributes
Extends	ExtendsAttributes	ExtendsInClass	GetExtendsAttributes
Components	ComponentAttributes	ComponentsInClass	GetComponentAttributes
Import	ImportAttributes	ImportsInClassAttributes	

Figure 6 Overview of API to class structure. The row headings are the element types and the column headings the different functions (and records).

To make it possible to traverse all classes it is also possible to list all top-level classes (optionally limited to the ones defined in a specific Modelica file).

4.1.1 Example

These functions can be used in Modelica to find all restricted classes and provide e.g. the following list of accessible classes (excluding protected and partial ones):

	Modelica 1.6	Modelica 2.1
Model	222	429
Block	71	147
Function	41	199
Type	485	513
Package	50	130

Figure 7 Statistics for Modelica Standard Library

The growth of the standard library is in part due to the fact that ModelicaAdditions libraries were completed and after (in some cases major) revisions included in the Modelica Standard Library.

An alternative to returning all elements as one array of records would be to provide an iterator, or a callback-routine for enumerating the elements (and access routines instead of record elements). This is a traditional style in several environments (iterators in C++, enumeration callbacks in Windows API) since it avoids allocating large arrays. However, it requires additional state (in the iterator or enumerator call), which is contrary to the limitations on functions in Modelica, and therefore also increases the risk of errors in application code.

4.2 API to semantics not only to syntax

The API above defines basic routines that can be used directly. They also provide the basis for writing functions intended to answer higher-level questions, e.g. to search in a hiearchy for all components declared of a certain class.

Programming such queries require that the API answers questions related to the semantics of the declarations instead of questions based on their syntax (i.e. Dymola must not only parse the Modelica classes to answer the question, but also implement e.g. the semantics of look-up in Modelica).

To clarify this consider the declaration of T2 in the coupled clutches demo:

```
parameter SI.Time T2;
```

To obtain information about this declaration we can use the following:

```
Dymola.AST.GetComponentAttributes(
"Modelica.Mechanics.Rotational"+
".Examples.CoupledClutches","T2")
```

which gives the result:

```
Dymola.AST.ComponentAttributes(
  name = "T2",
  fullTypeName="Modelica.SIunits.Time",
  isProtected = false,
  sizes = {},
  variability = "parameter",
  isInput = false,
  isOutput = false,
  isInner = false,
  isOuter = false,
  isReplaceable = false,
  isRedeclared = false,
  isGraphical = false
)
```

By returning the full name of the type ("Modelica.SIunits.Time") and not the type-name part of the declaration ("SI.Time") it is straightforward to program this kind of queries and this also made it easier to program the calling interface from Matlab.

Obviously advanced users would like to also have access to the exact declaration (including modifiers and annotations) and that is planned for the future.

Basing the API on the semantics is also important for future improvement of providing routines to modify the classes through the API, where declaring a new

component might require the addition of import-statements. When a user drags and drops a class to the diagram layer to declare a new component Dymola automatically adds import-statements if necessary. The API can internally re-use this functionality.

Similarly copying (or moving) a class from one Modelica package to another might require changes to its declarations which is done automatically by Dymola's GUI and hidden from the user.

4.3 Simplification in Modelica 2.2

One previous problem with using these functions was that the sizes of non-inputs to functions had to be known from the call according to the Modelica 2.1 standard [3]. That requires complex work-arounds and/or additional functions.

The restriction has now been lifted in Dymola (and accepted for Modelica 2.2) allowing a variable declared with size : (and without any binding assignment) to be re-sized (if necessary) when assigned in the function (note that this includes not only variables declared directly in the function but also their record elements). The change is backward compatible since such variables previously were semantically incorrect.

The change is not limited to working with the Modelica-structure, but is also useful in Modelica for unrelated uses, e.g. a routine that returns the positive eigenvalues. It was also needed to implement the API functions themselves, in particular the size-field for array of a component.

Those wanting an additional rationale can examine the case below where the same function as given in two versions, one written before the feature was implemented and another version rewritten to use it:

4.3.1 Example after simplification

As an example consider a function for finding the attribute of all classes defined in package (including the contents of packages – after the package):

```
function attributeModelsInPackage
  import Dymola.AST.*;
  input String s;
  output ClassAttributes attr[:];
protected
  String localClasses[:]=
    ClassesInPackage(s);
 ClassAttributes attributes;
algorithm
  for i in 1:size(localClasses,1) loop
    attributes:= GetClassAttributes(
       s + "." + localClasses[i]);
    attr:=cat(1,attr,{attributes});
    if attributes.restricted
        == "package" then
      attr := cat(1, attr,
        attributeModelsInPackage(
           attributes.fullName));
    end if;
  end for;
end attributeModelsInPackage;
```

4.3.2 Example before simplification

Before this feature of automatic resizing of arrays was available it was necessary to write two routines, one to determine the length of the array and one to actually return the array.

We consider this for the simpler case of only returning the full names of the classes, first we have to count the size of the output:

```
function countModelsInPackage
  import Dymola.AST.*;
  input String s;
  output Integer count= 0;
protected
 ClassAttributes attributes;
algorithm
  for i in ClassesInPackage(s) loop
    attributes:=GetClassAttributes(
        s + "." + i);
    count:=count+1;
    if attributes.restricted
         =="package" then
      count := count +
        countModelsInPackage(
           attributes.fullName);
    end if;
  end for;
end countModelsInPackage;
```

Note that there is no declared array for the result of ClassesInPackage – instead it is directly iterated over removing the need for any local variable (and the problem of its size) .

```
function attributesModelsInPackage
  import Dymola.AST.*;
  input String s;
  output String
    attr[countModelsInPackage(s)];
protected
 ClassAttributes attributes;
 Integer index=0;
 Integer len;
algorithm
  for i in ClassesInPackage(s) loop
    attributes:=
      GetClassAttributes(s + "." + i);
```

```
    index:=index+1;
    attr[index]:=attributes.fullName;
    if attributes.restricted
         =="package" then
      len :=countModelsInPackage(
        attributes.fullName);
      attr[index+1:index+len] :=
       attributesModelsInPackage(
        attributes.fullName);
      index:=index+len;
    end if;
  end for;
end attributesModelsInPackage;
```

Apart from practical problem of writing such complex functions an additional problem is that there is a need to maintain multiple functions. If requirements change (e.g. only return public classes) it is necessary to update two functions.

4.4 Revisited example from Matlab

When we previously considered the following call from Matlab

```
>>dymolaCall(…
'Modelica.Math.Matrices.leastSquares',[],…
[1;0;1],[1;2;3])
```

we indicated that Dymola's API was used to construct this argument. The calls of Dymola API functions are:

```
Dymola.AST.GetComponentAttributes(
"Modelica.Math.Matrices.leastSquares",
"A")
Dymola.AST.GetClassAttributes("Real")
Dymola.AST.GetClassAttributes("Real")
Dymola.AST.GetClassAttributes("Real")
Dymola.AST.GetComponentAttributes(
"Modelica.Math.Matrices.leastSquares",
"b")
Dymola.AST.GetClassAttributes("Real")
Dymola.AST.GetClassAttributes("Real")
Dymola.AST.GetClassAttributes("Real")
```

Finally the result is the following function call:

```
Modelica.Math.Matrices.leastSquares(
[1;0;1],{1,2,3})
```

The first call, GetClassAttributes, determines that this is a function call and not the call of a record constructor. The next call, ComponentsInClass, is used to determine the components of the function. For each argument the next input component is found by looking at the component attributes (this check is not performed for record constructors). The type of input component is then accessed, GetClassAttributes("Real"), to find that it is a primitive numeric type (since booleans must be treated specially).

The significant part is that in Matlab there are two matrices/columns vectors and based on the Modelica function the first one is sent as matrix to Dymola ([1;0;1]) and the second one as a vector ({1,2,3}). Without the API-calls it would not have been possible to determine that these should be treated differently.

5 Conclusions

This paper shows that Dymola 6's Modelica implementation provides an extendable external interface to use other tools and also be useful from other tools. In addition it shows that an interface to the Modelica class structure is useful in itself and also can be used when implementing the external interface.

References

[1] Dynasim (2005): **Dymola User's Manual**, Dynasim, www.dynasim.com,

[2] XML Protocol Working Group (2000-): **SOAP** http://www.w3.org/2000/xp/Group

[3] Modelica Association (2004): **Modelica Language Specification Version 2.1**, www.modelica.org

[4] World Wide Web Consortium (1996-): **Extensible Markup Language (XML)**, http://www.w3.org/XML/

[5] Pop A., P. Fritzson P. (2003**): ModelicaXML: A Modelica XML Representation with Applications**, Proceedings of the 3rd International Modelica Conference, Linköping Sweden.

[6] Andreasson J. (2003): **VehicleDynamics library**, Proceedings of the 3rd International Modelica Conference, Linköping Sweden.