Client-side Modelica powered by Python or JavaScript

Rüdiger Franke, ABB, Germany – Ruediger.Franke@de.abb.com

Abstract

Modelica is primarily supported by simulation environments for the treatment of equation based models and model libraries. As of today Modelica is rarely used for the exchange of engineering data, visualization or interactive computing, even though the Modelica language offers a lot of interesting features for such applications.

This paper investigates the potential of lightweight Modelica tools that run directly in scripting or web clients. Two Modelica parsers have been implemented in the popular client-side languages Python and JavaScript.

The Modelica parser in Python is extended with a backend translating algorithmic Modelica definitions to Python. This gives access to existing Python packages from scripted Modelica. It also enables the interactive debugging of algorithmic Modelica code.

The Modelica parser in JavaScript offers a generic backend interface. The paper demonstrates two applications. First a simple analysis tool for Modelica packages running from the command line is demonstrated. The true potential of JavaScript is the embedding of engineering data as Modelica code with HTML5 documents and their processing on the client side, e.g. in Web browsers. The paper shows a Modelica text editor and parameter GUI generator running in a web browser.

Keywords: Modelica, scripting, interactive computing, data exchange, Lex, Yacc, Python, JavaScript, jQuery, HTML5.

1 Introduction

Today's Modelica simulation environments act as servers that offer proprietary client interfaces, basing on Modelica Script, COM, or CORBA, for instance. There has been no success in standardizing Modelica client interfaces so far. XML has been selected for tool coupling in the FMI standard.

Major advantages of XML are that it is both: human and machine readable. XML parsers are readily available. The major drawbacks of XML are its very basic syntax, making it bulky and hard to read for humans. The semantics still needs to be defined.

This paper investigates the use of Modelica itself as interface language. Modelica is human and machine readable as well. The major advantages of Modelica are its more compact, richer syntax. The semantics is already standardized, tailored for modeling and simulation, including:

- rich syntax for high-level definitions, like packages, classes, records, enumerations, doc strings, and physical units;
- modification syntax for predefined classes that is comparable to XML documents for XML schemata;
- convenient formulation of matrix expressions, statements and functions;
- embedded graphical representation;
- embedded HTML documentation;
- embedded version management;
- enable automatic generation of graphical user interfaces out of Modelica definitions.

The features are unique in their combination and could directly be exploited without the need to define some new XML schema first.

The drawback is that parsing Modelica by machines is not as simple as parsing XML. But a Modelica parser neither is a miracle since the concrete syntax is specified and tools like Lex and Yacc exist.

2 MoiPy – Modelica in Python

MoiPy is intended to bridge the gap between the powerful Modelica language and convenient scripting. This is done by adding a thin syntactic layer on top of the existing scripting language Python, translating between Modelica and Python, and by using the NumPy package for scientific computing.

MoiPy is not an alternative to other Modelica tools; it is an optional addition. MoiPy allows staying in the Modelica world even if the simulation environment at hand has only limited support for model-based applications or scripting.

2.1 Implementation overview

MoiPy implements the Modelica syntax (Modelica 3.3 specification, Appendix B; see [1]) using Python Lex-Yacc. PLY provides Lex and Yacc entirely in Python, including extensive error checking and logging. The syntax specification can directly be executed. A parser table is generated on demand and cached in the background.

Of course the execution speed is slower, compared to a parser explicitly generated and readily compiled. This is why MoiPy only reads Modelica files as needed.

The parser produces an abstract syntax tree (AST). Each node of the tree is a Python object. The AST reduces the concrete syntax for simpler further processing. For instance, a class_definition is represented as object of ClassDefinition. The attributes of class_prefixes, class_specifier and composition appear directly in the ClassDefinition. The elements and sections of the composition are further reduced into one elementList, one initialEquationList / initialStatementList and one equationList / statementList.

An exemplary rule reads:

```
def p_annotation(p):
  ''' annotation : ANNOTATION \
        class_modification '''
    p[0] = Annotation(
      classModification = p[2],
      track = Track(p, 1))
```

The syntax rule is specified in the documentation string of a Python function that implements the respective production. A new object of the class Annotation is created in the example. The second argument tracks the location in the Modelica code.

The class definition objects have the common method toPython that generates Python code from the Modelica definitions. The toPython method of PrimaryUnsignedNumber, for instance, constructs a predefined Real or Integer object that adds attributes like min, max and unit to the value itself. Note that the Python code generation only covers the algorithmic part of the Modelica language, i.e. a subset of all possible class definitions:

- Modelica packages are treated as Python modules
- Modelica functions are translated to Python functions
- Modelica records are translated to Python classes
- Modelica enumerations are translated to Python objects

Moreover expressions and statements are covered:

- Modelica expressions are evaluated as Python expressions
- Modelica arrays are treated as NumPy arrays
- Modelica builtin functions are forwarded to Python functions
- Modelica statements are executed as Python statements

2.2 Extended Modelica syntax for scripting

MoiPy attempts to stay as close as possible to Modelica, using the same rules as specified in the Modelica concrete syntax and without introducing any new keywords. Two extensions are needed for scripting though: support for commands and use of variables without declaration on the top-level scope.

The Modelica concrete syntax covers stored definitions in the form of class definitions. MoiPy additionally accepts commands on the top-level scope that are Modelica expressions, statements or import clauses.

In Modelica each variable must be declared before use. MoiPy follows this rule inside class definitions. Also on the top-level scope assignments like:

```
v := {1,2,3}
```

are declaration errors if `v` has not been declared before. On the top-level scope, outside class definitions, it is allowed though to implicitly declare a variable by defining it equal to an existing object. For instance:

```
v = {1,2,3}
```

defines `v` to be the vector `{1,2,3}`. Afterwards

```
v := {4,5,6}
```

assigns a new value to the existing vector `v`,

```
v := "a string"
```

is an error, whereas

```
v = "a string"
```

(re)defines `v` to be a string.

MoiPy uses the file extension `.moi` to distinguish interpreted script files from regular Modelica definitions.

2.3 Basic uses

Start MoiPy with

```
python moi.py
```

A parser table for Modelica is generated when called the first time. Afterwards the `moi>>` command prompt appears.

Do some matrix calculations, e.g.:

```
moi>> A = [1,2;3,4]
moi>> b = {1,1}
moi>> A*b
```

or matrix concatenations:

```
moi>> [A,b]
moi>> [A; transpose([b])]
```

Call a Modelica function from Python:

```
moi>> t = 0:0.01:1
moi>> y = Modelica.Math.sin(
...   2*Modelica.Constants.pi*t)
```

This will look for the Modelica definitions under `MOIPYPATH`, load the files `Modelica/Math/package.mo` and `Modelica/Constants.mo`, translate the function `sin` and the constant `pi` to Python, call the function, and assign the result to the vector y. Note that the function call is vectorized automatically.

2.4 Enhance Modelica with Python

Python offers many interesting modules, such as NymPy, SciPy and matplotlib. MoiPy enables to access them from Modelica. Type:

```
moi>> import Plt = matplotlib.pyplot
```

MoiPy attempts to find a Modelica definition first. As `matplotlib` is not found in Modelica, the import gets forwarded to Python and found there, provided you have `matplotlib` installed. Type:

```
moi>> Plt.plot(t, y, "ro");
moi>> Plt.title(
...   "pyplot for Modelica");
moi>> Plt.show();
```

A plot window pops up; see Figure 1.

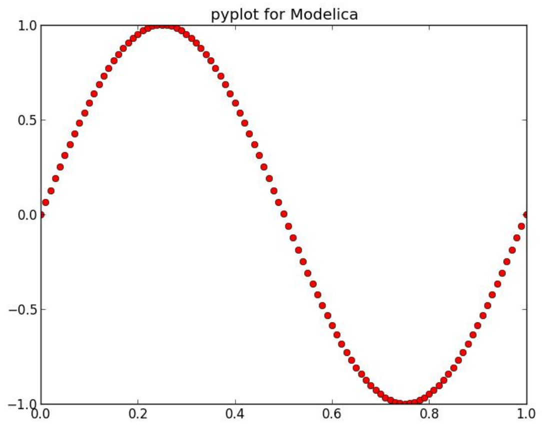


Figure 1: pyplot generated from Modelica

2.5 Access Modelica from Python

MoiPy translates Modelica definitions to Python. This means that the translated definitions may also be called from Python directly. When started in interactive mode:

```
python –i moi.py
```

and done some Modelica scripting, e.g.:

```
moi>> import
...   Modelica.SIunits.Conversions.*
moi>> from_degF(70)
```

one may leave MoiPy with Ctrl-D (Ctrl-C under Windows). A Python command prompt appears. Type:

```
>>> from_degF(70)
```

to directly call the translated Python function. One may switch back to Modelica with:

```
>>> moipy()
```

2.6 Advanced Modelica functions

Modelica.Media defines more advanced functions. They serve as example for 3D plotting and for debugging in the subsequent section. Take the example:

```
moi>> Modelica.Media.Water.
...   IF97_Utilities.h_pT(1e5, 300);
```

In order to evaluate the function for the specific enthalpy, MoiPy needs to load 7 Modelica files and translate 19 functions as well as 7 data records out of 13 packages in MSL 3.2.1. The parsing of the files takes a few seconds. Once loaded and translated, subsequent calls go fluently.

```
moi>>  p = linspace(1, 300, 30)
moi>>  T = linspace(0, 600, 30);
moi>>  (pp, TT) =
...    numpy.meshgrid(p, T);
moi>>  hh = Modelica.Media.Water.
...    IF97_Utilities.h_pT(
...      pp * 1e5,
...      TT .+ 273.15);
```

This evaluates the function `h_pT` at 30x30=900 grid points. See also `examples/mplot3dDemo.moi` resp. call it:

```
moi>>  import examples.mplot3dDemo
```

A wireframe plot should pop up; see Figure 2.

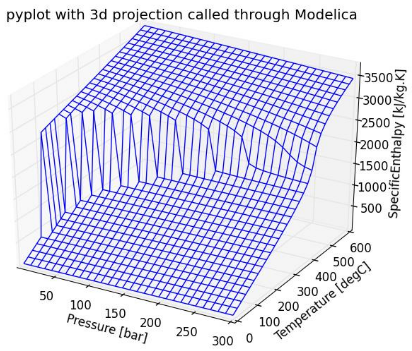


Figure 2: Wireframe plot for IF97_Utilities.h_pT

2.7 Debugging of Modelica functions

Python offers the extensible debugger pdb. This is exploited by MoiPy to transform the debugger outputs to the originating Modelica code. It uses the prompt (modb). This gives features like entering the debugger in case of errors, treating break points, stepping through Modelica code, walking up and down on the call stack and inspecting variables.

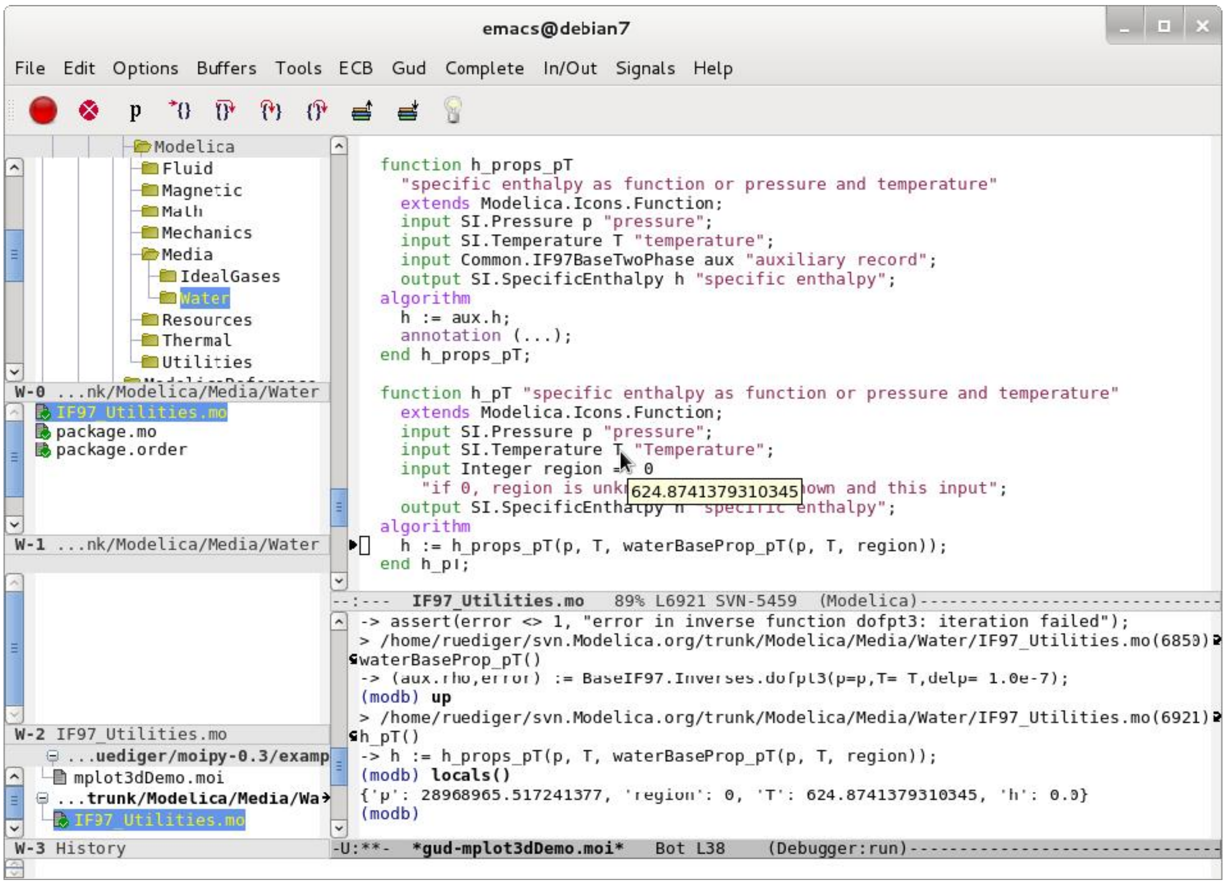
See also below for an example – the debugger shows an error in the IF97_Utilities.h_pT function that was present prior to r6066 around 290 bar and 350 °C – it has been fixed in MSL 3.2.1.

2.8 Integration with an IDE

An Integrated Development Environment (IDE) typically integrates several command line tools and can be extended to support new ones. shows Emacs running modb as example.

Several Emacs extensions have been loaded, such as Emacs Code Browser for the directory tree, SVN status and revision, besides Modelica mode for syntax highlighting and annotation folding. The Emacs debugger framework parses the output of modb and add a graphical user interface. Moreover it opens and shows the respective source file at the right position.

Figure 3: MoiPy in the GNU Emacs IDE



```
function h_props_pT
  "specific enthalpy as function or pressure and temperature"
  extends Modelica.Icons.Function;
  input SI.Pressure p "pressure";
  input SI.Temperature T "temperature";
  input Common.IF97BaseTwoPhase aux "auxiliary record";
  output SI.SpecificEnthalpy h "specific enthalpy";
algorithm
  h := aux.h;
  annotation (...);
end h_props_pT;

function h_pT "specific enthalpy as function or pressure and temperature"
  extends Modelica.Icons.Function;
  input SI.Pressure p "pressure";
  input SI.Temperature T "Temperature";
  input Integer region = 0
    "if 0, region is unk                   own and this input";
  output SI.SpecificEnthalpy h "specific enthalpy";
algorithm
  h := h_props_pT(p, T, waterBaseProp_pT(p, T, region));
end h_pT;
--:---  IF97_Utilities.mo   89% L6921 SVN-5459  (Modelica)-----------------------------
 -> assert(error <> 1, "error in inverse function dofpt3: iteration failed");
 > /home/ruediger/svn.Modelica.org/trunk/Modelica/Media/Water/IF97_Utilities.mo(6850)
waterBaseProp_pT()
 -> (aux.rho,error) := BaseIF97.Inverses.dofpt3(p=p,T= T,delp= 1.0e-7);
 (modb) up
 > /home/ruediger/svn.Modelica.org/trunk/Modelica/Media/Water/IF97_Utilities.mo(6921)
h_pT()
 -> h := h_props_pT(p, T, waterBaseProp_pT(p, T, region));
 (modb) locals()
 {'p': 28968965.517241377, 'region': 0, 'T': 624.8741379310345, 'h': 0.0}
 (modb)
-U:**-  *gud-mplot3dDemo.moi*   Bot L38    (Debugger:run)------------------------------
```

3 MoiJS – Modelica in JavaScript

Besides its convenient syntax for scientific computing, Modelica is strong in supporting graphical user interfaces. A Modelica model may contain annotations with flowsheet graphics and parameter dialogs, besides documentation – this is important for the specification and exchange of engineering data.

Graphical user interfaces are currently undergoing fundamental changes. Powerful, standardized GUI clients are running on virtually any device, exploiting HTML5 (HTML, CSS and JavaScript). What does this mean for a Modelica client?

Instead of dealing with proprietary server interfaces, events and callbacks, the client could receive a Modelica definition, parse it, build the user dialog, manage user interactions autonomously, and post back user inputs as Modelica definition or modification.

A Modelica parser in JavaScript is needed. Initially developed at Netscape almost 20 years ago, JavaScript grew to a multi-paradigm language covering functional, imperative and object-oriented programming. It gained a lot of momentum since the standardization of HTML5; see [3], the appearance of fast just-in-time compilers, development tools, powerful libraries, such as jQuery [4], and the adoption for server side programming as well, e.g. by Node.js [5].

3.1 Implementation overview

MoiJS implements the Modelica syntax (Modelica 3.3 specification, Appendix B; see [1]) using Jison. Jison provides Flex and Bison (Lex and Yacc) in JavaScript; see [6]. The exemplary rule given in section 2.1 reads:

```
annotation:
  ANNOTATION class_modification
  {
    $$ = new Annotation(track(@$));
    $$.classModification = $2;
  }
  ;
```

MoiJS generates a reduced AST as well. A significant difference to MoiPy is that JavaScript objects forming the nodes of the AST are based on prototypes. The prototypes can be extended later on. This means that arbitrary backends can be added without having to touch the original parser code (or requiring the parser to offer a specific plug-in architecture).

3.2 Adding a backend

Assume all executable models of a Modelica library shall be identified, in order to automate testing. Figure 4 shows a MoiJS backend for this.

Figure 4: Exemplary MoiJS backend

```
// load Modelica parser as CommonJS module
var moparser = require("./moparser").parser;

// add method forModifier to each modification and annotation
moparser.Modification.prototype.forModifier =
moparser.Annotation.prototype.forModifier = function (name, callback) {
    (this.classModification || []).forEach(function(argument) {
        if (argument.name == name)
            callback(argument.modification);
    });
}

// add method logStopTime to each class_definition, calling forModifier
moparser.ClassDefinition.prototype.logStopTime = function(within) {
  var definition = this;
  // check for experiment StopTime annotation
  if (definition.annotation) {
    definition.annotation.forModifier("experiment", function(experiment) {
      experiment.forModifier("StopTime", function(modification) {
        console.log(within + ".\n   " + definition.ident
                    + "(StopTime=" + modification.expression.value + ");");
      });
    });
  }
  // treat subclasses recursively
  (definition.classDefinitionList || []).forEach(function(classDefinition) {
      classDefinition.logStopTime(within + "." + definition.ident);
  });
}
```

It may appear confusing how to dive into the syntax tree. MoiJS closely follows the rules and names of the Modelica concrete syntax specification, in order to simplify the understanding. The capitalization is changed to camel style. If for instance the concrete syntax defines a class_modification, then a classModification instance of ClassModification appears in the AST. The suffix List is used if curly braces find in the Modelica concrete syntax.

The syntax tree may also be explored in the JavaScript Object Notation (JSON) – see below.

The backend (and the parser) can be executed in Node.js that provides, besides a JavaScript runtime, a portable operating system interface through POSIX wrappers. An exemplary Modelica file is processed with:

```
var fs = require("fs");

fs.readFile("Modelica/StateGraph.mo",
            function(err, data) {
  if (err) throw err;
  // parse the file
  var ast =
    moparser.parse(data.toString());
  // call the new backend
  for (i in ast.classDefinitionList)
    ast.classDefinitionList[i]
      .logStopTime(ast.name || "");
});
```

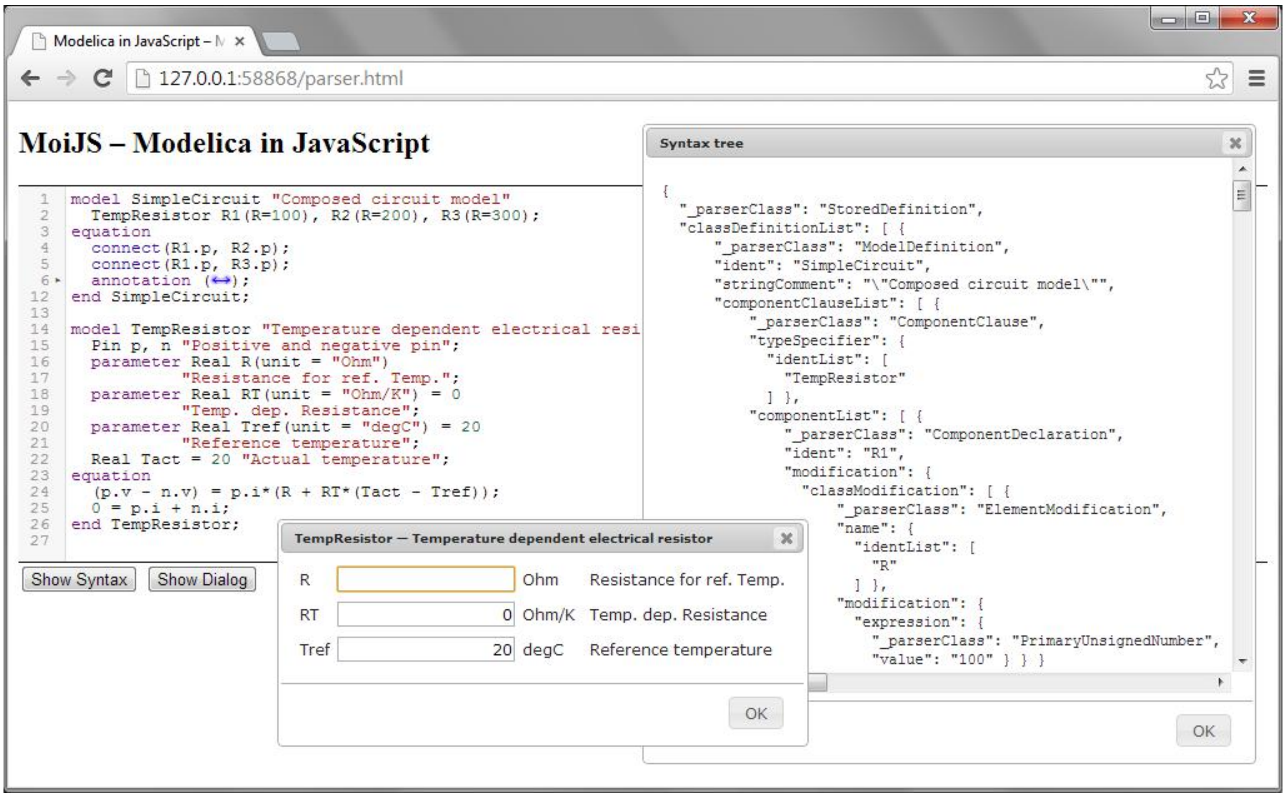
This produces the output:

```
Modelica.StateGraph.Examples.
   FirstExample(StopTime=5);
Modelica.StateGraph.Examples.
   FirstExample_Variant2(StopTime=5);
Modelica.StateGraph.Examples.
   FirstExample_Variant3(StopTime=5);
Modelica.StateGraph.Examples.
   ExecutionPaths(StopTime=15);
Modelica.StateGraph.Examples.
   ShowCompositeStep(StopTime=15);
Modelica.StateGraph.Examples.
   ShowExceptions(StopTime=20);
Modelica.StateGraph.Examples.
   ControlledTanks(StopTime=100);
```

3.3 Modelica in a Web browser

Having a Modelica parser in JavaScript, it is a small step to run it in a Web browser, using solely HTML5 standards. Figure 5 shows first results. The MoiJS lexer provides information for code coloring and annotation folding in an HTML textarea. The button "Show Syntax" invokes the MoiJS parser and shows the JSON representation of the resulting syntax tree in a popup window. The button "Show Dialog" invokes a backend to generate a parameter dialog out of the syntax tree. The GUI has been implemented using the jQuery UI library [4].

Figure 5: Modelica in the Google Chrome browser


Modelica in JavaScript – M
127.0.0.1:58868/parser.html
MoiJS – Modelica in JavaScript
Show Syntax
Show Dialog
Syntax tree
OK

```
model SimpleCircuit "Composed circuit model"
  TempResistor R1(R=100), R2(R=200), R3(R=300);
equation
  connect(R1.p, R2.p);
  connect(R1.p, R3.p);
  annotation (↔);
end SimpleCircuit;

model TempResistor "Temperature dependent electrical resi
  Pin p, n "Positive and negative pin";
  parameter Real R(unit = "Ohm")
            "Resistance for ref. Temp.";
  parameter Real RT(unit = "Ohm/K") = 0
            "Temp. dep. Resistance";
  parameter Real Tref(unit = "degC") = 20
            "Reference temperature";
  Real Tact = 20 "Actual temperature";
equation
  (p.v - n.v) = p.i*(R + RT*(Tact - Tref));
  0 = p.i + n.i;
end TempResistor;
```

```
{
  "_parserClass": "StoredDefinition",
  "classDefinitionList": [ {
      "_parserClass": "ModelDefinition",
      "ident": "SimpleCircuit",
      "stringComment": "\"Composed circuit model\"",
      "componentClauseList": [ {
          "_parserClass": "ComponentClause",
          "typeSpecifier": {
            "identList": [
              "TempResistor"
            ] },
          "componentList": [ {
              "_parserClass": "ComponentDeclaration",
              "ident": "R1",
              "modification": {
                "classModification": [ {
                    "_parserClass": "ElementModification",
                    "name": {
                      "identList": [
                        "R"
                      ] },
                    "modification": {
                      "expression": {
                        "_parserClass": "PrimaryUnsignedNumber",
                        "value": "100" } } }
```

TempResistor – Temperature dependent electrical resistor

R		Ohm	Resistance for ref. Temp.
RT	0	Ohm/K	Temp. dep. Resistance
Tref	20	degC	Reference temperature

OK

4 Caveats on the Modelica syntax

It is a pleasure to experience that the Modelica concrete syntax can be passed to a general-purpose parser generator, such as PLY or Jison, and something useful comes out. Nevertheless there find some things that might be improved.

4.1 Syntax of primary numbers

The Modelica syntax does generally not rely on the use of whitespaces. With one exception: a primary unsigned number may end with a dot and an arithmetic operator might begin with a dot; see also [1], section 10.6.6. The expression

```
2.+[1,2;3,4]
```

is wrong, because the dimensions of the scalar “2.” and the array [1,2;3,4] do not match. The expression

```
2 .+[1,2;3,4]
```

is fine. The additional space makes clear that the dot shall belong to the element-wise addition operator.

The Modelica syntax should be changed to not allow primary numbers ending with a dot. Generally a “2” without dot is fine, in particular because the Modelica division operator “/” is non-truncating, e.g. “1/2” gives 0.5 and not 0. If nevertheless a primary number shall be forced to be a Real, then one can add a 0 behind the dot, e.g. write “2.0”.

4.2 Expression syntax

The Modelica expression syntax defines operator precedence and associativity with grammar rules. This leads to very long productions. When a primary number is passed as function argument, for instance, then this primary goes through factor, term, arithmetic_expression, relation, logical_factor, logical_term, logical_expression, and simple_expression, to finally become an expression.

This is not only hard to read, but also slows down parsers. The expression syntax can alternatively be specified with all operators in one rule, i.e.

```
expr :   primary
     |   expr or expr
     |   expr and expr
     |   not expr
     |   expr rel_op expr
     |   expr add_op expr
     |   add_op expr
     |   expr mul_op expr
     |   expr ("^" | ".^") expr
```

The operator precedence and associativity can be defined in a separate table.

Prec	Operators	Associativity
7	or	left
6	and	left
5	not	right
4	< <= == <> >= >	left
3	+ - .+ .-	left
2	* / .* ./	left
1	^ .^	right

MoiJS, for instance, parses the Modelica Standard Library, version 3.2.1, more than 20% faster with this handy expression syntax.

4.3 Syntax of embedded HTML documentation

Working with HTML5 one gets used to documents that contain multiple special-purpose syntaxes, like HTML for content, CSS for styling and JavaScript for behavior – and maybe Modelica for engineering physics. Looking ugly initially, the richer syntax finally helps to faster grasp the different facets of the document. Modern text editors support such documents with mixed modes.

Modelica models may contain embedded HTML documentation. Unfortunately the HTML code needs to be encoded into Modelica strings, meaning that all double quotes used inside the HTML documentation need to be escaped. A general purpose text editor cannot detect and highlight the HTML code.

It should be considered to switch the syntax of embedded HTML documentation from Modelica strings to regular HTML, i.e. allow double quotes up to the ending </html> tag. In the simplest case a new Modelica string delimiter could be introduced, like """ for multi-line strings in Python. This way the readability improves and the mixed mode support of modern text editors could be exploited.

5 Conclusions and Outlook

Today’s Modelica simulation environments offer proprietary client interfaces, basing on Modelica Script, COM or CORBA, besides more. Only XML has been considered as standardized interface format for tool coupling so far. The major drawbacks of XML are its clumsy syntax and that the semantics still needs to be defined.

This paper investigates the use of the Modelica language itself as interface format for client/server architectures and for model exchange, instead of XML. This offers the advantage of having the semantics already standardized. The price to pay is a Modelica parser on the client side. This price turns out to be affordable in modern scripting or Web environments.

Two Modelica parsers have been implemented: MoiPy in Python and MoiJS in JavaScript. Both use the same grammar rules, exploiting parser generators with Lex and Yacc functionality.

It is not the aim of client-side Modelica to compete with simulation environments. The focus is on additional tasks in model-based applications, like scripting, testing, documentation, visualization and graphical user interfaces.

Python is strong in scientific computing. Due to its rich syntax, including e.g. operator overloading, and available packages, such as NumPy, it was straightforward to add a Python backend to the MoiPy parser, resulting in a Modelica interpreter and debugger for algorithmic models.

JavaScript is strong in dynamic user interfaces and in connecting them to servers. This is the main motivation for MoiJS. This paper shows how Modelica code is processed either in a console application or in an HTML5 user interface running the same parser.

The availability of compatible JavaScript implementations by multiple vendors and fast just-in-time compilers being preinstalled on virtually any device make JavaScript attractive for more applications. Examples are HTML5 pages containing verbatim Modelica code that is evaluated on the client side, e.g. in a Web browser or in a modern mobile device.

MoiJS is extensible with new backends, exploiting the prototype based inheritance of JavaScript. MoiJS is available under the MIT license at http://omuses.github.io/moijs.

References

[1] Modelica – A Unified Language for Systems Modeling, Language Specification, version 3.3, May 9, 2012. https://www.modelica.org/documents/ModelicaSpec33.pdf

[2] David Beazley: PLY (Python Lex Yacc), version 3.4, 2011. http://www.dabeaz.com/ply/

[3] Steve Jobs: Thoughts on Flash, April 2010. http://www.apple.com/hotnews/thoughts-on-flash/

[4] jQuery – write less, do more; and jQuery UI. http://jquery.com, http://jqueryui.com

[5] Node.js – a platform for easily building fast, scalable network applications. http://nodejs.org

[6] Zachary Carter: Jison – your friendly JavaScript parser generator, version 0.4.13, 2013. http://zaach.github.io/jison/

Acknowledgements

This work was supported in parts by the Federal Ministry of Education and Research (BMBF) within the ITEA2 project MODRIO (Model Driven Physical Systems Operation) – BMBF funding code: 01IS12022A.