Mo|E – A Communication Service Between Modelica Compilers and Text Editors

Nicola Justus[1] Christopher Schölzel[1] Andreas Dominik[1] Thomas Letschert[1]

[1]KITE, Technische Hochschule Mittelhessen, Giessen, Germany, {nicola.justus, christopher.schoelzel, andreas.dominik, thomas.letschert}@mni.thm.de

Abstract

The Modelica language is becoming increasingly popular among scientists and engineers as platform for modelling physical or biological systems. Although Modelica is maintained as non-proprietary language by the Modelica Association, a considerable number of commercial implementations and development environments is complemented by a surprisingly small number of open source tools.

In this paper, we present the communication service Mo|E that connects any text editor as front-end with a Modelica compiler as back-end. Based on the simple HTTP communication protocol, editor plugins for a software developer's favourite text editor can be developed easily, hence turning any editor into a lightweight Modelica development tool.

We also present a first implementation of a plugin for the text editor Atom that exhibits features necessary for efficient software development, such as display of compile errors, code completion, go to declaration or view of context-sensitive documentation. In addition, Modelica-specific checking of the number of equations in a model is supported.

Keywords: Modelica, open source, integrated development environment, distributed systems, structured editor, ENSIME, OpenModelica, JModelica, MoTE

1 Introduction

Modelica is a powerful object-oriented programming language that facilitates acausal description of physical systems. Although many commercial and open source tools for developing or working with Modelica are available, the OpenModelica suite (Fritzson et al., 2005) is the only comprehensive set of tools for Modelica. OpenModelica provides a standalone Modelica compiler, an Eclipse plugin for developing Modelica inside of Eclipse (MDT), a graphical model editor for connecting components (OMEdit), and a Modelica debugger. The primary tools for developing Modelica are MDT and OMEdit. Both are full-fledged integrated development environments (IDEs).

IDEs are well suited for working with big projects but may have some disadvantages. They often are slow, difficult to use and and may be even scary for novice users. For Modelica additional challenges arises from the differences between Modelica compilers, such as JModelica or OpenModelica which silghtly differ in their understanding of Modelica. In order to develop code compatible with different compilers, the IDE should be able to compile models using different compilers.

Today, when writing source code or any other type of structured text, it is common to use a structured editor which is aware of the document's structure. Structured editors are an essential part of most IDEs. Experienced developers usually prefer them to other – graphical – means of input. A structure aware editor must be able to analyze the text given to it. Thus structure awareness means awareness of the syntax and to some extend also of the semantics of the texts it deals with. The structured editor is deeply integrated with the IDE, rather than being just a mere component.

In this paper we present Modelica | Editor (Mo|E), a development environment for Modelica, centered on editing and checking complex models, refraining form all issues of model execution. A structured editor is its main component and user interface.

A key concept of Mo|E is that the user may use a text editor of her own choice, attach it to a service process that provides syntactic and semantic analysis and transforms the plain text editor to a structured editor.

Thus users may edit texts using the editor they are used to and still benefit from automatic recompilation, code completion, semantic highlighting, go to declaration, refactoring, and so on.

A central part of our solution is a server process that mediates between the text editor and Modelica aware analytical services. These services are provided by existing Modelica compilers, and/or further existing or future tools that may be plugged into this infrastructure (Figure 1). We have enhanced one text editor to a Modelica editor, but other text editors may be integrated with little effort. These editors only have to provide a plugin that implements the service API. This API provides a unique interface to different Modelica compilers and eases the communication with compilers and related tools, protecting users from complex and differing command-line interfaces.

The design of Mo|E was inspired by the ENSIME project (ENSIME Contrib., 2016) with its server process that mediates between text editors and Scala compilers.

Mo|E is an environment for developing Modelica models using editors that are enhanced to be Modelica aware. It was realized as part of the first author's bachelor's thesis (Justus, 2016).

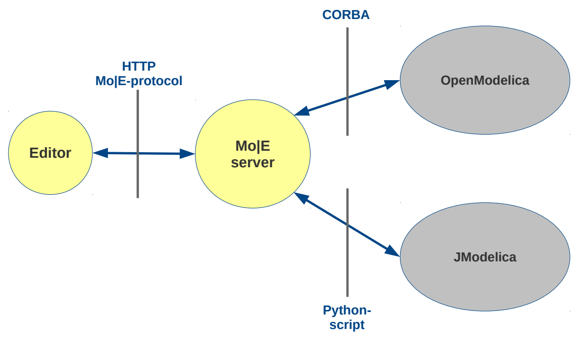


Figure 1. Survey of the communication between a text editor with a Mo|E-plugin, the Mo|E server and OpenModelica or JModelica.

1.1 Structure of the Paper

Section 2 describes which technologies and standards where used to implement Mo|E. Section 3 describes the protocol between client and service process, the communication with OpenModelica as well as with JModelica. Section 4 presents the key features of Modelica | Editor (Mo|E) and their use in the text editor Atom. Finally, section 5 gives a summary and a short outlook on future extensions.

1.2 Naming

The name Modelica | Editor (Mo|E) alludes to the use of the pipe character (|) in UNIX-like operating systems, which establishes a pipeline between two programs. Mo|E can be seen as such a pipeline between the Modelica compiler and a structured editor. In contexts where special characters like the pipe may cause problems, we chose the alternative spelling Modelica–Pipe–Editor (MoPE).

1.3 Goals

Our goals for Mo|E are:

- Provide an extendable client server application which makes it possible to develop Modelica inside existing text editors.
- Provide a client implementation for the text editor Atom as reference for other clients.
- Highlight syntax and type errors, perhaps while typing, inside the text editor.
- Provide code completion for models, data types, and variables.
- Provide jump to the source of a model. This is better known as "go to declaration".
- Provide a view of the documentation of a model.

1.4 Background and Related Work

OneModelica (Samlaus, 2015) is a an Eclipse-based IDE for Modelica models tailored to the domain of fluid dynamics. It was realized using tools and techniques of Model Driven Software Development. It may be compared to our approach in that it restricts itself to syntax and static semantics of the language and refrains from simulation issues. It differs considerably in its technological base, which in the years since its development has lost a lot of its attraction and support, not without reason as we think.

Mo|E is the first tool in a more ambitious project called Modelica Tool Ensemble (MoTE). MoTE aims at the provision of a collection of small user-friendly standalone applications for developing and executing Modelica models, i.e. a lightweight development environment for Modelica.

Modelica does not differ in principle from other languages when it comes to development environments. However, due to its complex static and dynamic semantics, it poses special challenges, mainly for the support of incremental development (see e.g. (Höger, Lorenzen, and Pepper, 2010) or (Broman, Fritzson, and Furic, 2006)).

We are well aware of these problems. Thus, at least for the time being, MoTE and Mo|E do not include a Modelica compiler or tools incorporating compiler features much beyond parsing. Instead we rely on mature compilers like OpenModelica and JModelica.

2 Technologies

2.1 Scala and Akka

Scala (EPFL, 2016) is a hybrid programming language that combines object orientation with functional programming. Because the Scala compiler generates bytecode for the Java Virtual Machine (JVM), it integrates with many available Java libraries. In addition, resulting compiled programs are platform independent. The service process of Mo|E is implemented in Scala.

Akka is a library for concurrent and distributed systems, based on the actor model that facilitates concurrency by providing a high level of abstraction (Allen, 2013). We use Akka as a provider of communication services, such as an implementation of the HTTP-protocol and for structuring the system according the actor model.

2.2 OMC and CORBA

OpenModelica provides the *Advanced Interactive OpenModelica Compiler* (OMC), a server that provides an API to query loaded Modelica code (Asghar et al., 2011).

The Common Object Request Broker Architecture (CORBA) is used by the OpenModelica compiler server OMC as interface to other applications and other programming languages.

CORBA developed by the Object Management Group (OMG) defines a standard for inter-process communication modeled as interaction of distributed objects. Because the public API of remote objects is defined in an Interface Definition Language (IDL), processes may be implemented in different programming languages (OMG, 2012).

2.3 Atom and the Electron Engine

Atom (GitHub, 2016a) is a text editor created by GitHub in the style of Sublime Text (Sublime HQ Pty Ltd, 2016). Basic design concepts of Atom include customization and extensibility through plugins (called packages in the context of Atom). Extending Atom is possible with JavaScript, HTML and CSS by using the Electron Engine (GitHub, 2016c). This allows to *rapidly develop* extensions and to implement communication protocols using *AJAX requests*. Furthermore, Atom already includes a package for syntax highlighting for Modelica (Chenouard et al., 2016), a simple API for completion suggestions (GitHub, 2016b) and a plugin for clicking on text (Facebook, 2016), which is used to implement *go to declaration* functionality.

We have created an Atom plugin as first reference implementation of a Mo|E client.

3 Design

3.1 Mo|E – Editor Protocol

Clients are connected to the service process, by means of Hypertext Transfer Protocol (HTTP)-based communication and JavaScript Object Notation (JSON) data representation. HTTP provides status codes, Uniform Resource Identifiers (URIs) and content negotiation (Fielding and Reschke, 2014). JSON is a compact text format, based on the JavaScript Object Notation (Bray, 2014).

The communication flow follows several steps: Firstly, the client connects to the service process using a *connect request* that communicates the current project. In this context a project is a directory containing Modelica source files.

Secondly, after initialization the service process answers with the respective *project id*. The unique project id identifies the project in the client server communication.

Henceforth, the client uses this project id to *request further IDE functionality* for this project from the service process.

To finally *finish a session,* the client sends a disconnect request that triggers the service process to delete all project-related information and cached data.

The following sections describe each supported IDE functionality in more detail.

3.1.1 Connecting to the server

As introduced in the preceding section, each client needs to connect initially with the server. A `connect` request is initiated through a POST request containing the respective JSON object with the project description. The JSON object contains the full path into the project directory and the relative path to a directory that is used to store compiled files:

```
POST /mope/connect

{
  "path": <String>,
  "outputDirectory": <String>
}
```

This project information is stored in the `mope-project.json` file that is placed in the project directory.

If the request was successful, the server answers with a project id. If not, the server answers with `400 BadRequest` and a detailed error message.

3.1.2 Compiling Modelica source files & Modelica script files

Compiling a Modelica source file is initiated through a `compile` request. The request body contains the path to the currently opened file. As a result of the request a model is instantiated and type errors are retrieved:

```
POST /mope/project/0/compile

{ "path": <String> }
```

If the request was successful, the server answers with a JSON array containing compiler errors:

```
{
  "type": "Error" | "Warning", //type of
     message
  "file": <String>, //path to the file
     which contains the error
  "start": { //start of error
    "line": <Number>,
    "column": <Number>
  },
  "end": { //end of error
    "line": <Number>,
    "column": <Number>
  },
  "message": <String> //compiler error
}
```

Compiling a Modelica script file is initiated by sending an analogous `compileScript` request:

```
POST /mope/project/0/compileScript

{ "path": <String> }
```

Although the request is called “compiling a Script file”, the service process actually executes the script. This action is intended for debugging purposes of smaller scripts and not for scripts that simulate a model, since simulating a model is time-consuming and may freeze or possibly even kill the service process.

3.1.3 Checking a model

To check a model for its number of equations the client sends a `checkModel` request with the model path. The server calls the OpenModelica compiler to run `checkModel` and answers with a string containing the results:

```
POST /mope/project/0/checkModel

{ "path": <String> }
```

This functionality is only available, if the OpenModelica compiler is used.

3.1.4 Go to declaration

To retrieve the declaration of a model, the client sends a `declaration` request. This request contains the model/-class name as query string[1]:

```
GET /mope/project/0/declaration?class=[
    Modelname]
```

The server answers with a JSON object containing the file path and line number of the declaration:

```
{
  "path": <String>, //absolute path to the
      file
  "line": <Number> //line number
}
```

If the project id is unknown or the query string is missing, the server will answer with a `404 NotFound` error.

3.1.5 Go to documentation

A model documentation can be retrieved using a `doc` request with the model name encoded as query string:

```
GET /mope/project/0/doc?class=[Modelname]
```

The server embeds the documentation in a template and returns a HTML document that can be viewed in a web browser.

If the project id is unknown or the query string is missing, the server answers with a `404 NotFound` error.

3.1.6 Code completion

For code completion the client sends a `completion` request with a JSON object that describes the position of the cursor as *file* (name of current file), *line* and *column number* (position of the cursor) and *word* (part of the expression to be completed):

```
POST /mope/project/0/completion

{
  "file": <String>, //absolute path to the
      file
  "position": { //position inside the file
    "line": <Number>,
    "column": <Number>,
  },
  "word": <String>
}
```

The server responds by sending an JSON array of possible completions for the expression:

```
{
  //type of completion; 1 of the listed
      strings
  "kind": "Type" | "Variable" | "Function"
      | "Keyword" | "Package" | "Model" | "
      Class" | "Property",
  "name": <String>, //the completion
  //OPTIONAL: list containing names of
      parameters if kind=function
  "parameters": [
    <String>,
    <String>,
    ...
  ],
  //OPTIONAL: the class comment describing
      the name attribute
  "classComment": <String>,
  //OPTIONAL: the type of name
  "type": <String>
}
```

[1]A query string is a component of a URI, that starts with a **?** (Berners-Lee, Fielding, and Masinter, 2005).

kind defines the type of the completion (such as package, class, function, variable, etc.). *name* is the suggestion for the subexpression.

The optional return values for *parameters*, *classComment* and *type* report the list of argument names if the suggestion is a function, the documentation string if the the suggestion is a class and the data type of the expression (usually the data type of a variable), respectively.

If the given project id is unknown, the server answers with `404 NotFound`.

3.1.7 Display data type of a variable

To retrieve data type and documentation string of a variable, the client sends a `typeOf` request with a body identical to the body of the `completion` request. If the request was successful, the server answers with a JSON object containing the name, type, and documentation string of the variable. Otherwise the server answers with `404 NotFound`:

```
POST /mope/project/0/typeOf

{
  "name": <String>, //name of property
  "type": <String>, //type of property
  //OPTIONAL: property comment
  "comment": <String>
}
```

3.1.8 Disconnecting from the server

A session is terminated by a disconnect request, which initiates the shutdown sequence for this project on the server:

```
POST /mope/project/0/disconnect
```

The server returns `204 NoContent` if the project id is known or `404 NotFound` elsewise.

3.1.9 Stopping the server

The client can stop the whole service process by sending a `stopServer` request. The server answer is `202 Accepted`.

```
POST /mope/stop-server
```

3.2 Communication with OpenModelica

The modeling and development environment OpenModelica (OSMC, 2016) consists of a Modelica compiler (omc), a graphical connection editor (OMEdit), an Eclipse plugin (MDT) and a Modelica debugger (Fritzson et al., 2005). As described in Chapter 2.2 the compiler enables querying for model information via its CORBA interface that provides several types of information:

- list of all models/classes by sending `getClassNames`,
- source file of a model by sending `getSourceFile`,
- documentation annotation of a model by sending `getDocumentationAnnotation`,
- result of a model check for equations by sending `checkModel`,
- documentation string of a model by sending `getClassComment`,
- arguments of a function by sending `getParameterNames`,
- specialization of a class by sending `getClassRestriction`.

An additional difficulty arises from the fact that OpenModelica uses Modelica expressions as arguments for its CORBA interface. As a result, the functions listed above are not implemented explicitely in the CORBA interface. Instead, OpenModelica only provides a single method in its CORBA interface, namely `sendExpression` and sends Modelica source code strings and API function calls as arguments. Therefore, we create the function calls as strings and interpolate them into the function argument, as shown in Listing 1.

Listing 1. API function call through OpenModelica's CORBA interface.

```
val omc:OmcCommunication = ...
val fileName = "/tmp/model.mo"
val errors:String =omc.sendExpression(s"""
  parseFile("$fileName")""")
```

3.3 Communication with JModelica

JModelica (Modelon AB, 2016) is a Modelica compiler developed by Modelon AB (Åkesson et al., 2010). To allow dynamic adjustments during execution, JModelica offers a Python interface which enables code modification at run time. In addition it enables compilation of Modelica code. We are using this Python interface for compilation of the models by delivering the Modelica source files to a custom Python script, which calls the JModelica compiler, parses JModelica's output and encodes the output into JSON. The resulting JSON is printed to `stdout` which is afterwards parsed by the service process and finally decoded as Scala objects. The communication scheme is depicted in Figure 2.

Command	Description
Mope: Disconnect	Disconnect Atom from the service process
Mope: Compile Project	Compile the project
Mope: Run Script	Execute the Modelica script
Mope: Check Model	Check the model for its number of equations
Mope: Show Type	Display the data type of the variable below the cursor
Mope: Open Documentation	Open the documentation of the type below the cursor
Mope: Open Server Log	Open the log file of the service process
Mope: Open Server Config	Open the configuration file of the service process
Mope: Stop Server	Stop the server

Table 1. List of commands implemented in the Atom plugin.

Unfortunately JModelica does not offer access to the parsed model or its abstract syntax tree. That is the reason why code completion is restricted to local variables and go to documentation is not yet supported in the presented Mo|E Atom plugin.

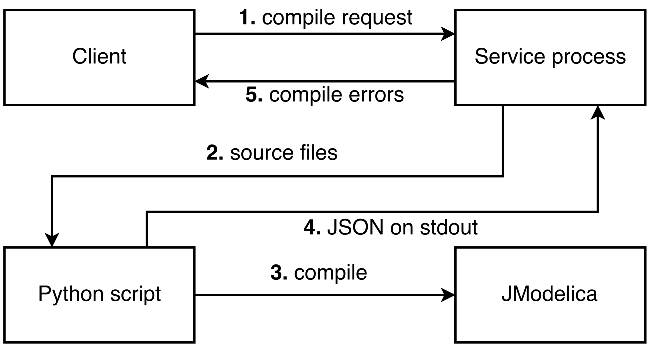


Figure 2. Diagram of the communication between a text editor (client) and JModelica.

4 Features

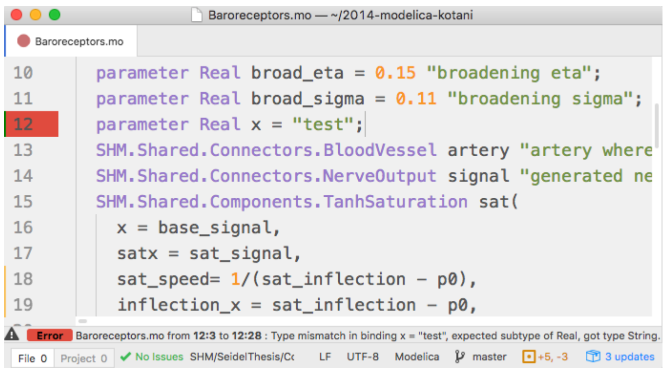
4.1 Client commands

Table 1 gives a full list of the commands available in the Atom plugin.

4.2 Compiler Feedback

Modelica | Editor (Mo|E) provides *instant compiler feedback* for syntax errors and *type errors*. Background compilation is automatically triggered when a file is saved and the errors are highlighted in the editor with a red indicator at the left side of the editor tab. Error messages are displayed at the bottom of the tab (Figure 3). Alternatively automatic compilation can be disabled and triggered manually.

As MoIE supports JModelica and OpenModelica it is possible to use either JModelica or OpenModelica or both compilers for one project.



Baroreceptors.mo — ~/2014-modelica-kotani

Baroreceptors.mo

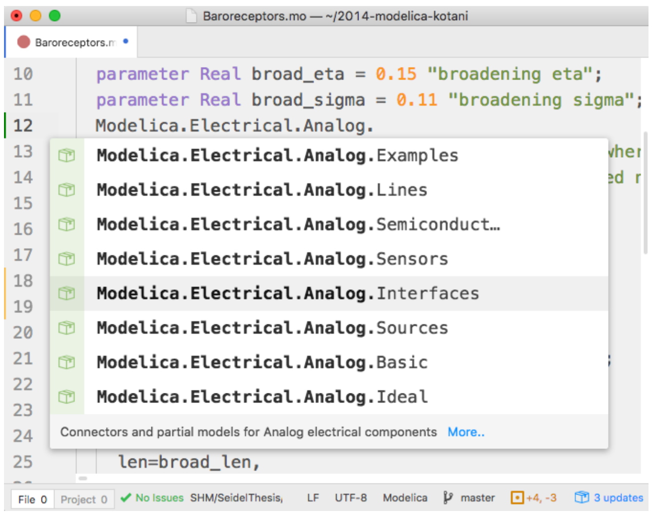
```
parameter Real broad_eta = 0.15 "broadening eta";
parameter Real broad_sigma = 0.11 "broadening sigma";
parameter Real x = "test";
SHM.Shared.Connectors.BloodVessel artery "artery where
SHM.Shared.Connectors.NerveOutput signal "generated ne
SHM.Shared.Components.TanhSaturation sat(
  x = base_signal,
  satx = sat_signal,
  sat_speed= 1/(sat_inflection - p0),
  inflection_x = sat_inflection - p0,
```

Error Baroreceptors.mo from **12:3** to **12:28** : Type mismatch in binding x = "test", expected subtype of Real, got type String.

Figure 3. Compile errors are retrieved form the back-end (OpenModelica or JModelica) by the MoIE server and highlighted in the source code by the editor plugin.

4.3 Code Completion

Modelica | Editor (MoIE) features *enhanced code completion* on keystrokes or by pressing `Ctrl + Space`. Suggestions include classes, models, functions, model parameters and variables, keywords, built-in types as well as local variables. The suggestions contain a type indicator, documentation string and a link to the model's documentation (Figure 4). The type indicator displays the type of the suggestion (package, model, function or variable).



Baroreceptors.mo — ~/2014-modelica-kotani

Baroreceptors.mo

```
parameter Real broad_eta = 0.15 "broadening eta";
parameter Real broad_sigma = 0.11 "broadening sigma";
Modelica.Electrical.Analog.
```

```
Modelica.Electrical.Analog.Examples
Modelica.Electrical.Analog.Lines
Modelica.Electrical.Analog.Semiconduct…
Modelica.Electrical.Analog.Sensors
Modelica.Electrical.Analog.Interfaces
Modelica.Electrical.Analog.Sources
Modelica.Electrical.Analog.Basic
Modelica.Electrical.Analog.Ideal
```

Connectors and partial models for Analog electrical components More..

```
  len=broad_len,
```

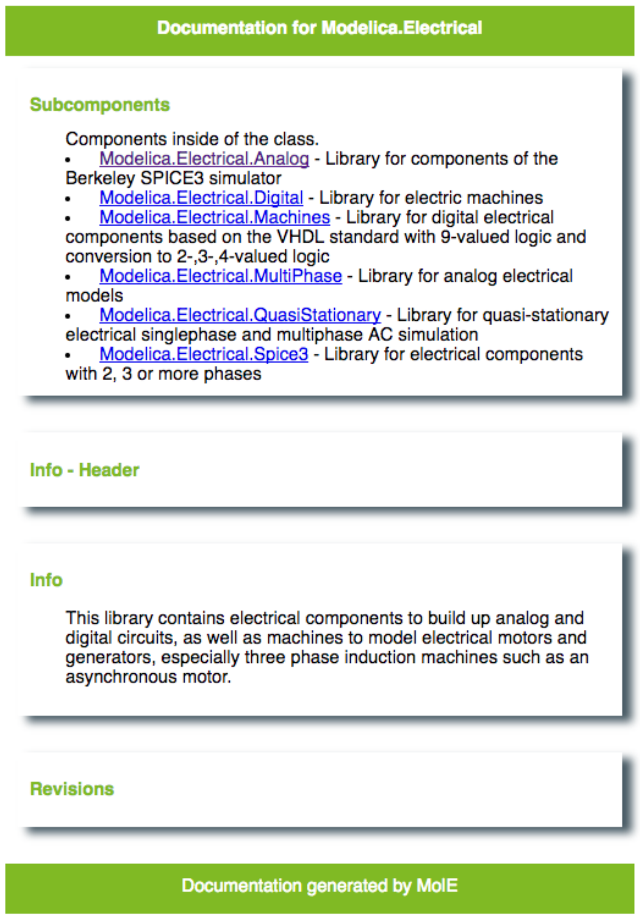
Figure 4. Code completion allows for selecting classes, models, functions, model parameters, variables, keywords or built-in types from a list of suggestions retrieved by the MoIE server.

4.4 Go to Declaration

MoIE provides *go to declaration* by clicking on the model/class name while holding down `Ctrl`. The source file of the model/class is opened in a separate tab. Go to declaration is mostly used for discovering source code or when editing multiple models that are linked to each other.

4.5 Documentation View

MoIE embeds the queried documentation of a model in a predefined template and provides the documentation as HTML document. The implementation in the Atom plugin opens the requested documentation in the default browser. Furthermore it is possible to browse the model's child components using the links in the subcomponents section of the documentation (Figure 5).



Documentation for Modelica.Electrical

Subcomponents

Components inside of the class.
- Modelica.Electrical.Analog - Library for components of the Berkeley SPICE3 simulator
- Modelica.Electrical.Digital - Library for electric machines
- Modelica.Electrical.Machines - Library for digital electrical components based on the VHDL standard with 9-valued logic and conversion to 2-,3-,4-valued logic
- Modelica.Electrical.MultiPhase - Library for analog electrical models
- Modelica.Electrical.QuasiStationary - Library for quasi-stationary electrical singlephase and multiphase AC simulation
- Modelica.Electrical.Spice3 - Library for electrical components with 2, 3 or more phases

Info - Header

Info

This library contains electrical components to build up analog and digital circuits, as well as machines to model electrical motors and generators, especially three phase induction machines such as an asynchronous motor.

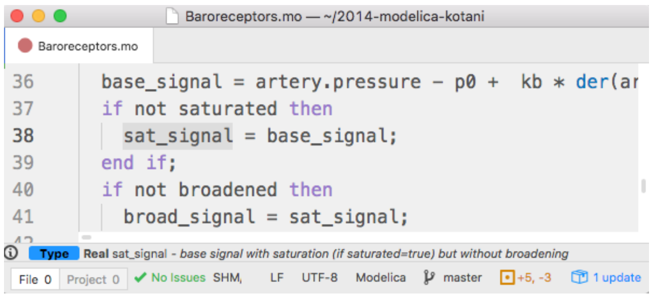
Revisions

Documentation generated by MoIE

Figure 5. Example of a documentation display generated as HTML page by MoIE by embedding the retrieved documentation string with a template page.

4.6 Type & Documentation String Display

MoIE provides a command for displaying the type and documentation string of the variable at the cursor position. Type and documentation are displayed at the bottom of the editor tab (Figure 6).



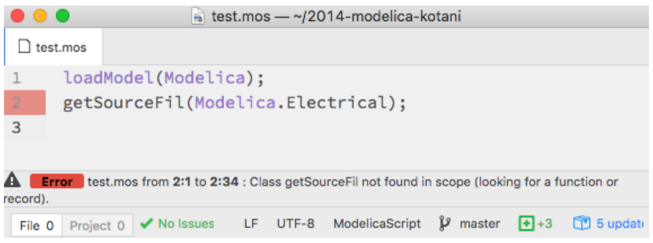
```
base_signal = artery.pressure - p0 +  kb * der(ar
if not saturated then
  sat_signal = base_signal;
end if;
if not broadened then
  broad_signal = sat_signal;
```

Type Real sat_signal - *base signal with saturation (if saturated=true) but without broadening*

Figure 6. Data type and documentation string are displayed in the editor window by the Atom plugin.

4.7 Execution of Modelica Scripts

If the OpenModelica compiler is used, MoIE allows manually triggered execution of Modelica scripts and displays error messages in the editor.



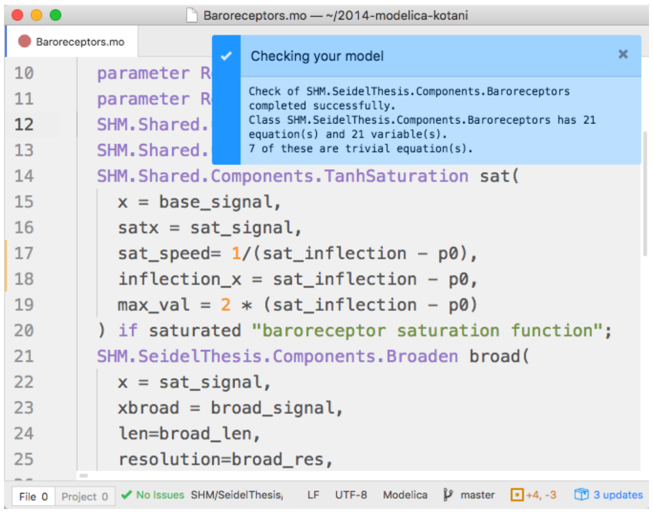
```
loadModel(Modelica);
getSourceFil(Modelica.Electrical);
```

Error test.mos from **2:1** to **2:34** : Class getSourceFil not found in scope (looking for a function or record).

Figure 7. Compile errors of Modelica scripts are displayed in the editor window by the Atom plugin.

4.8 Model Check

MoIE supports checking of a model for the number of equations (Figure 8), if the OpenModelica compiler is used.



Checking your model

```
Check of SHM.SeidelThesis.Components.Baroreceptors
completed successfully.
Class SHM.SeidelThesis.Components.Baroreceptors has 21
equation(s) and 21 variable(s).
7 of these are trivial equation(s).
```

```
parameter R
parameter R
SHM.Shared.
SHM.Shared.
SHM.Shared.Components.TanhSaturation sat(
  x = base_signal,
  satx = sat_signal,
  sat_speed= 1/(sat_inflection - p0),
  inflection_x = sat_inflection - p0,
  max_val = 2 * (sat_inflection - p0)
) if saturated "baroreceptor saturation function";
SHM.SeidelThesis.Components.Broaden broad(
  x = sat_signal,
  xbroad = broad_signal,
  len=broad_len,
  resolution=broad_res,
```

Figure 8. Result of a model check, performed by the OpenModelica back-end, is displayed as pop-up in the editor window by the Atom plugin.

5 Conclusions

This paper presented a extendable client/server application for developing Modelica in enhanced text editors like Atom. It shows how a service process is used to simplify communication with multiple Modelica Compilers and provide IDE features to various text editors through a simple interface. Text editors have to implement a small number of basic HTTP calls, which should be a minimal effort. A minimal setup with compilation and code completion would only require four HTTP calls. Installation instructions for MoIE can be found at `https://github.com/THM-MoTE/mope-server`.

MoIE is a base for further extensions. E.g. we intend to implement plugins for different editors, such as Sublime Text (Sublime HQ Pty Ltd, 2016), Visual Studio Code (Microsoft Corporation, 2016) or vim (Moolenaar, 2016). Including Visual Studio Code should not be a problem because it uses TypeScript for its plugins, which is a superset of Atom's JavaScript.

MoIE is part of a larger ensemble of tools called MoTE (Schölzel et al., 2016). MoTE will also include a vector graphic editor called Modelica Vector Graphics Editor (MoVE) (Justus et al., 2017) and a diagram editor called Modelica Diagram Editor (MoDE) (Hoppe et al., n.d.). Together with MoIE these tools provide alternative user interfaces for the interaction with existing Modelica compilers, which allow a simpler interaction than full-fledged IDEs like OpenModelica.

The projects are open source and hosted on GitHub:
`https://github.com/thm-mote/`

References

Åkesson, J. et al. (2010). "Modeling and Optimization with Optimica and JModelica.org — Languages and Tools for Solving Large-Scale Dynamic Optimization Problems". In: *Computers & Chemical Engineering* 34 (11), pp. 1737–1749.

Allen, Jamie (2013). *Effective Akka*. Sebastopol, USA: O'Reilly Media.

Asghar, Syed Adeel et al. (2011). "An Open Source Modelica Graphic Editor Integrated with Electronic Notebooks and Interactive Simulation". In: *Proceedings of the 8th International Modelica Conference*. Dresden, Germany, pp. 739–747.

Berners-Lee, T., R. Fielding, and L. Masinter (2005). *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. IETF.

Bray, T. (2014). *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159. IETF.

Broman, D., Peter Fritzson, and S. Furic (2006). "Types in the Modelica Language". In: *In Proceedings of the 5th International Modelica Conference*. Ed. by Ch.and Haumer A. Kral. Vienna, Austria: The Modelica Association, pp. 303–317.

Chenouard, Raphael et al. (2016). *Modelica language support in Atom*. GitHub Repository. URL: `https://github.com/modelica-tools/atom-language-modelica` (visited on 11/03/2016).

École Polytechnique Fédérale de Lausanne (2016). *The Scala Programming Language*. URL: `http://www.scala-lang.org/` (visited on 11/01/2016).

ENSIME Contributors (2016). *ENSIME*. URL: `http://ensime.github.io/` (visited on 11/01/2016).

Facebook (2016). *Hyperclick*. GitHub Repository. URL: `https://github.com/facebooknuclide/hyperclick` (visited on 11/03/2016).

Fielding, R. and J. Reschke (2014). *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230. IETF.

Fritzson, Peter et al. (2005). "The OpenModelica Modeling, Simulation, and Development Environment". In: *Proceedings of the 46th Scandinavian Conference on Simulation and Modeling (SIMS)*. Trondheim, Norway.

GitHub (2016a). *Atom*. URL: `https://atom.io` (visited on 11/01/2016).

– (2016b). *Autocomplete+ Package*. GitHub Repository. URL: `https://github.com/atom/autocomplete-plus` (visited on 11/03/2016).

– (2016c). *Electron — Build cross platform desktop apps with JavaScript, HTML, and CSS*. URL: `http://electron.atom.io/` (visited on 09/14/2016).

Höger, Christoph, Florian Lorenzen, and Peter Pepper (2010). "Notes on the Separate Compilation of Modelica". In: *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. Ed. by P. Fritzon et al. Oslo, Norway: Linköping Electronic Conference Proceedings, pp. 43–53.

Hoppe, Marcel, Christopher Schölzel, and Andreas Dominik. *MoDE – A Standalone Modelica Diagram Editor*. unpublished.

Justus, Nicola (2016). "Design and Implementation of a Client/Server Application for Editing Modelica Inside Various Text Editors". BA thesis. Giessen, Germany: Technische Hochschule Mittelhessen.

Justus, Nicola, Christopher Schölzel, and Andreas Dominik (2017). "MoVE – A Standalone Modelica Vector Graphics Editor". In: *12th International Modelica Conference*. Prague, Czech Republic. to be published.

Microsoft Corporation (2016). *Visual Studio Code - Code Editing. Redefined*. URL: `https://code.visualstudio.com/` (visited on 22/12/2016).

Modelon AB (2016). *JModelica.org*. URL: `www.jmodelica.org` (visited on 11/01/2016).

Moolenaar, Bram (2016). *welcome home : vim online*. URL: `http://www.vim.org/` (visited on 12/21/2016).

Object Management Group (2012). *Common Object Request Broker Architecture (CORBA). Part 1: CORBA Interfaces*. OMG document formal/2012-11-12. Version 3.3.

Open Source Modelica Consortium (2016). *OpenModelica*. URL: `https://openmodelica.org` (visited on 11/01/2016).

Samlaus, Roland (2015). "An Integrated Development Environment with Enhanced Domain-Specific Interactive Model Validation". PhD thesis. Linköping University, The Institute of Technology.

Schölzel, Christopher et al. (2016). *Modelica Tool Ensemble*. GitHub Repository. URL: `https://github.com/orgs/THM-MoTE/` (visited on 12/22/2016).

Sublime HQ Pty Ltd (2016). *Sublime Text: The text editor you'll fall in love with*. URL: `https://www.sublimetext.com/` (visited on 11/01/2016).