INFORMATION TECHNOLOGY FOR EUROPEAN ADVANCEMENT

"D.6.6.2 - Prototype for Incremental Compilation of Modelica Models in OpenModelica"

"WP 6.6: Incremental and separate compilation techniques for multi-mode systems"

"WP 6: Modelling and simulation services"

MODRIO (11004)

Version 1.0

Date 2016-04-24

Authors

Martin Sjölund	Linköping University
Peter Fritzson	Linköping University

Executive summary

Separate compilation of packages is very common in development environments. Source code is stored in separate files which are compiled into separate units that are subsequently linked together. Incremental compilation takes this a step further to smaller compilation units, such as functions, and only compiles the changes of a source file, which is faster if the dependency analysis of these differences is faster than separate compilation. Incremental compilation often also implies that part of a model/program can be changed on-line without restarting the program from scratch, which saves time during debugging.

This deliverable, a prototype of incremental/separate compilation in OpenModelica is delivered in three parts:

- Incremental compilation at the function level, supporting on-line changes. The first part of this M45 document.
- Separate compilation of models, using FMI export/import. The second part of this M45 document
- Separate compilation of packages of functions. Reported in the earlier M18 deliverable document.

The first part of this M45 deliverable supports incremental compilation at the function level: a changed function can be recompiled and loaded on-line, while a simulation or function is executing. This is done in a portable way by changing functions into function pointers, which allows a debugger such as GDB to load code and change pointers at run-time. The prototype is working but currently requires manually changing code or giving the names of the changed functions to the debugger.

Summary

1. Introduction

The FPP states for WP 6.6:

> Design and implementation of incremental and separate compilation techniques that allow very fast, also on-line, update of simulation code.

Separate compilation in OpenModelica has been available for some time [1] (since M18), for the compilation of packages consisting Modelica and MetaModelica functions. This makes re-compilation very fast if only a few compilation units have changed, also for large applications of more than 200 000 lines of code, consisting of many packages.

The difference between incremental and separate compilation here is that separate compilation will divide the Modelica code into separate units (according to some restrictions), compile each of these units into object code and then link them together. With incremental compilation, only changed pieces of code at a smaller granularity (e.g. functions) and the parts of the code that depends on these will be changed. The changes can then in most cases (depending on how extensive the changes are) be linked into an already running executable, changing the code on the fly.

2. Design of Incremental Compilation

The design of the incremental compilation is to generate portable code that can be updated on-line. The chosen approach includes changing all compiled functions to be C function pointers instead of regular C functions. The reason for this is that it is possible to use functions for handling shared objects (.so/.dll/.dylib) such as dlopen/LoadLibrary and dlsym/GetProcAddress in order to load compiled code and change the function pointer at run-time. This requires either that the function signature does not change when incrementally compiling code, or to not have any function with a changed function signature on the current call stack when patching the calls. This approach works both for a scripting API to dynamically load functions and for using GDB (Gnu debugger for C, C++) or similar debuggers to patch the call at any given time during run-time (provided the above restrictions hold).

Given the above constraints, the design is how to generate the runnable code and how to change it at run-time. What remains is the design on how to generate the code that will replace the old code. However, the main benefit of the incremental compilation would not only be compilation performance, but even more on-line changes to running code, e.g. during debuging. For this reason, the current design (which has not been completely implemented yet) would be to not only generate the shared object, but also to generate a description (perhaps a SHA-512 hash) of the implementation of the function as well as its function interface, stored in JSON files (one for each compilation unit). Then, when OMEdit or another debugger wants to update the running code, it compares the hashes in the two files and replaces the functions that changed given that all other constraints are fulfilled (checking the call stack of each thread to make sure it is possible to safely change the code).

3. Current Implementation of Incremental Compilation

The following describes the changes performed on the OpenModelica compiler to begin supporting incremental compilation, as described in the design. Note that none of the binary released distributions of OpenModelica have yet been compiled with support to change its functions on-line, since this functionality still is at the prototype stage. The compiler can generate relocatable code (function pointers) as an option (compiler flag: `--debug=relocatableFunctions`). The steps below describe what is implemented in OpenModelica, which includes a scripting function to change its own code at run-time.

1. Change all functions to be function pointers instead of actual functions. This was more work than it sounds, because OM internally has the concept of both function pointers, builtin function pointers, closures, regular function calls, external function calls, and calls to record constructors. This makes the code run slower (1 extra indirection per function call).

2. Export all symbols instead of only symbols for public functions. This prohibits some optimizations that make OM load and compile faster.

3. Write an interface that allows loading a shared object (so/dll/dylib). The OpenModelica implementation uses the following signature:

```
function relocateFunctions
  input String fileName;
  input String names[:,2];
  output Boolean success;
external "builtin";
annotation(
  Documentation(info="<html>
<p><strong>Highly experimental, requires OMC be compiled with special flags to use</strong>.</p>
<p>Update symbols in the running program to ones defined in the given shared object.</p>
<p>This will hot-swap the functions at run-time, enabling a smart build system to do some incremental
compilation
(as long as the function interfaces are the same).</p>
</html>"), preferredView="text");
end relocateFunctions;
```

4. Write the corresponding C-code to the scripting function (GDB/OMEdit will be able to do the same thing in the future):

```
int SystemImpl__relocateFunctions(const char *fileName, void *names)
{
  void *localHandle,*remoteHandle;
  remoteHandle = dlopen(fileName, RTLD_NOW | RTLD_GLOBAL | RTLD_NODELETE);
  if (!remoteHandle) {
    addDLError(gettext("Error opening library %s: %s."), fileName);
    return 0;
  }
  localHandle = dlopen(NULL, RTLD_NOW);
  if (!localHandle) {
    addDLError(gettext("Error opening library %s: %s."), fileName);
    return 0;
  }
  int length = listLength(names);
  void **localSyms[length], *remoteSyms[length];
  for (int i=0; i<length; i++) {
    void *tpl = MMC_CAR(names);
    const char *local = MMC_STRINGDATA(MMC_CAR(tpl));
    const char *remote = MMC_STRINGDATA(MMC_CDR(tpl));

    remoteSyms[i] = dlsym(remoteHandle, remote);
    if (remoteSyms[i]==0) {
      addDLError(gettext("Error opening library %s: %s."), fileName);
    }
    localSyms[i] = (void**) dlsym(localHandle, local);
    if (localSyms[i]==0) {
      addDLError(gettext("Error opening library %s: %s."), fileName);
    }

    names = MMC_CDR(names);
  }
  /* All loaded fine. Now relocate all the symbols. */
  for (int i=0; i<length; i++) {
    *localSyms[i] = remoteSyms[i];
  }
```

```
    return 1;
}
```

5. During run-time of OMC, a function should be modified. Such a tool could be automatic, but the current prototype uses manually changed functions due to time constraints. We modified the generated C-code for omcimpl_Static_elabExp (which is the implementation for the MetaModelica function Static.elabExp). It is also possible to generate a new shared object containing all functions in the compiler and manually loading the ones that the user knows have changed (again, this will be automated in the future):

```
DLLExport
modelica_metatype omcimpl2_Static_elabExp(threadData_t *threadData, modelica_metatype _inCache,
modelica_metatype _inEnv, modelica_metatype _inExp, modelica_boolean _inImplicit, modelica_metatype
_inST, modelica_boolean _inDoVect, modelica_metatype _inPrefix, modelica_metatype _inInfo,
modelica_metatype *out_outExp, modelica_metatype *out_outProperties, modelica_metatype *out_outST)
{
    modelica_metatype _outCache = NULL;
    modelica_metatype _outExp = NULL;
    modelica_metatype _outProperties = NULL;
    modelica_metatype _outST = NULL;
    modelica_metatype _e = NULL;
    modelica_integer _num_errmsgs;
    modelica_metatype _exp = NULL;
    modelica_metatype _exp1 = NULL;
    modelica_metatype _exp2 = NULL;
    modelica_metatype _prop1 = NULL;
    modelica_metatype _prop2 = NULL;
    modelica_metatype _ty = NULL;
    modelica_metatype _c = NULL;
    modelica_fnptr _elabfunc;
    modelica_metatype tmpMeta[2] __attribute__((unused)) = {0};
    MMC_SO();
    _tailrecursive: OMC_LABEL_UNUSED
    _outCache = _inCache;
    _outST = _inST;
    static int expCtr=0;
    fprintf(stderr, "Elaborating expression #%d: %s\n", expCtr++,
MMC_STRINGDATA(omc_Dump_printExpStr(threadData, _inExp)));
#line 313 "/home/marsj/OpenModelica2/OMCompiler/Compiler/FrontEnd/Static.mo"
    _e =
(omc_RewriteRules_noRewriteRulesFrontEnd(threadData)?_inExp:omc_RewriteRules_rewriteFrontEnd(thread
Data, _inExp, NULL));
  ...
```

6. Try out the interface using a mos-script:

```
1+1;
OpenModelica.Scripting.Experimental.relocateFunctions("separate-fn/elabExpNew.so",
{{"omc_Static_elabExp","omcimpl2_Static_elabExp"}});
getErrorString();
1+1;
getErrorString();
```

7. Result:

```
2
true
""
Elaborating expression #0: 1 + 1
Elaborating expression #1: 1
Elaborating expression #2: 1
2
""
```

4. Future Work

The design and implementation are not yet completely aligned. What is missing is that the OpenModelica editor (OMEdit) tool should be able to update code automatically based on the incrementally compiled code, and to automatically run the built-in diff tool to analyze which functions have been changed during package editing. This should be possible according to the design and no obstacles seem to remain except for some lack of time to introduce the features, which are planned to be finalized in the near future. The prototype has currently only been tested on Linux although the design has been made to generate very portable code. Thus, it is expected that the code should work on Windows with only minor changes.

5. References

[1] D6.6.2 – M18 - Prototype for separate compilation of Modelica models consisting of packages of functions using OpenModelica.

[2] D6.6.2 – M45 Prototype for separate compilation of Modelica models to the FMI format using OpenModelica.

"D6.6.2 – Prototype for Incremental Compilation of Modelica Models in OpenModelica"

Separate Compilation using FMI

"Sub WP 6.6: Incremental and separate compilation techniques for multi-mode systems"

"Work Package 6: Modelling and simulation services"

Version	1.0
Date	2016-04-20

Authors

Adeel Asghar	SICS East
Peter Fritzson	LiU

1. Overview

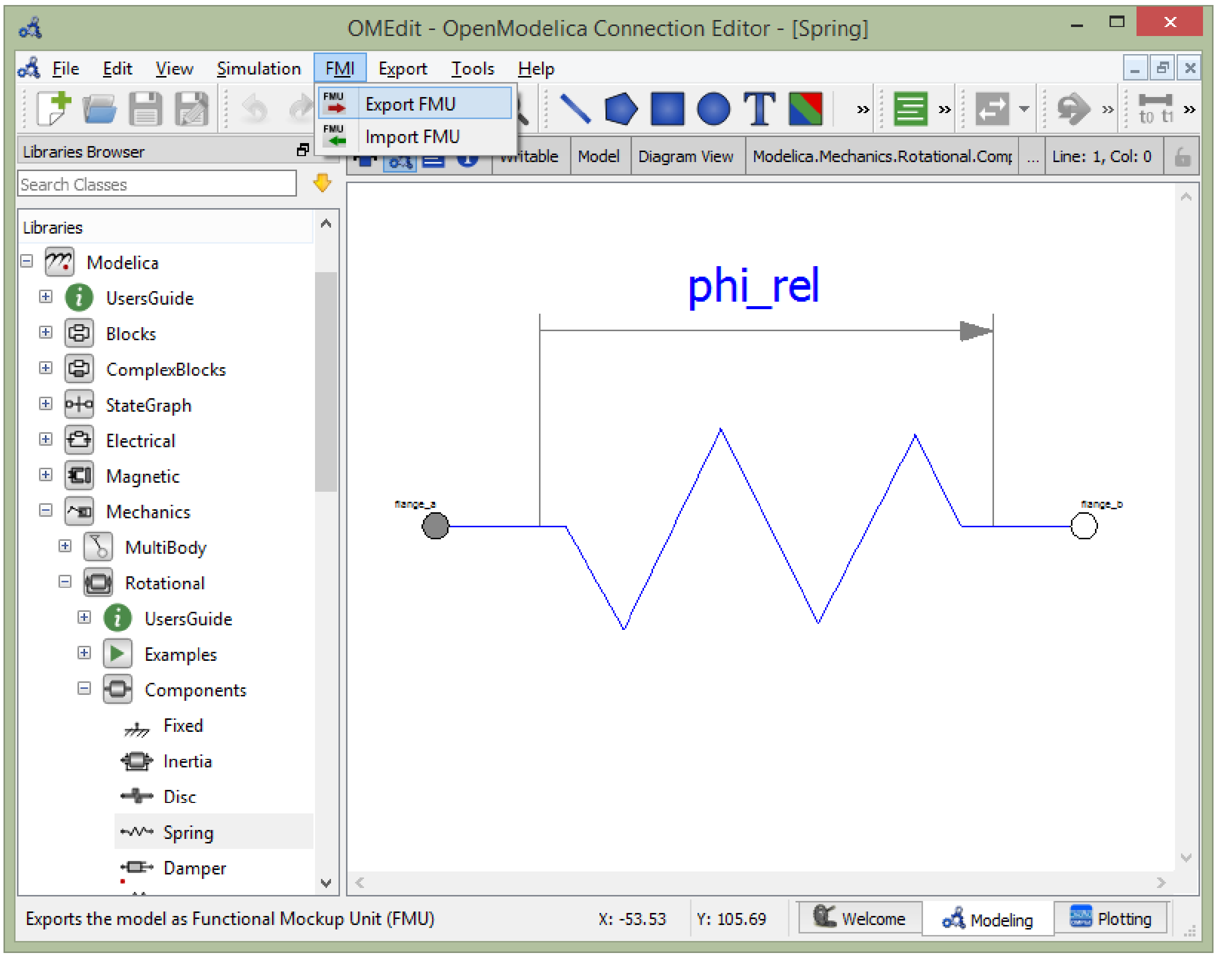
The goal of this task is to develop a separate compilation technique for Modelica models.

Separate compilation improves compilation speed of large and complex models because it allows you to decouple the models and compile the modules in parallel. With separate compilation you only need to recompile the changed modules. In order to produce a final executable the linker uses the object files of different modules created by separate compilation of each module.

Modelica models are acausal but if we make them causal then it's just a block with inputs and outputs. In order to integrate that block into other systems one need to determine the continuous states and discrete events of the block. This is exactly the same what Functional Mockup Interface (FMI) for Model-Exchange [1] does. In order to support separate compilation OpenModelica has implemented FMI export and import for Model-Exchange.

2. Implementation

OpenModelica exports the Modelica models to FMI by generating C code from the model, wrapping the generated C code by following the application programming interface (API) specified by the FMI standard, compiling the code and zipping it as Functional Mockup Unit (FMU). You can export using the OpenModelica Connection Editor (OMEdit) menu `FMI->Export FMU`,



Exports the model as Functional Mockup Unit (FMU)

Figure 1. FMI export from OpenModelica.

Alternatively you can also export using the OpenModelica API,

```
function buildModelFMU
  input TypeName className "the class that should be translated";
  input String version = "2.0" "FMU version, 1.0 or 2.0.";
  input String fmuType = "me" "FMU type, me (model exchange), cs (co-simulation),
me_cs (both model exchange and co-simulation)";
  input String fileNamePrefix = "<default>" "fileNamePrefix. <default> =
\"className\"";
  input String platforms[:] = {"dynamic"} "The list of platforms to generate code
for. \"dynamic\"=current platform, dynamically link the runtime.
\"static\"=current platform, statically link everything. Else, use a host triple,
e.g. \"x86_64-linux-gnu\" or \"x86_64-w64-mingw32\"";
  output String generatedFileName "Returns the full path of the generated FMU.";
end buildModelFMU;
```

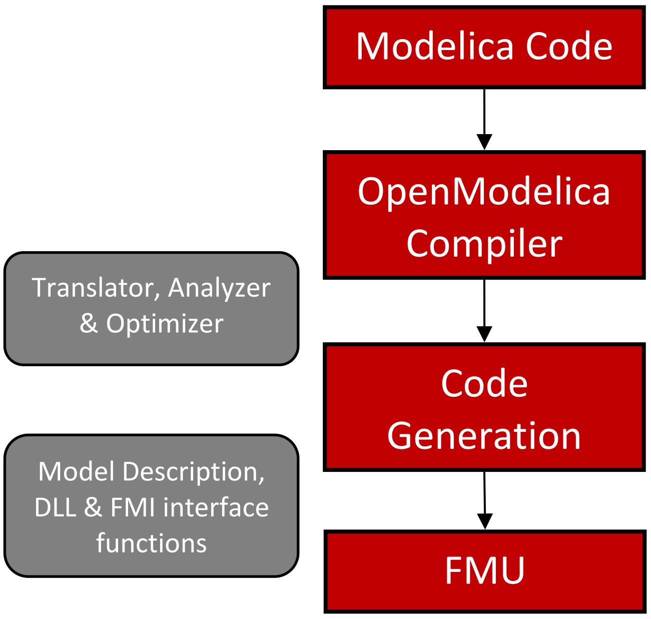
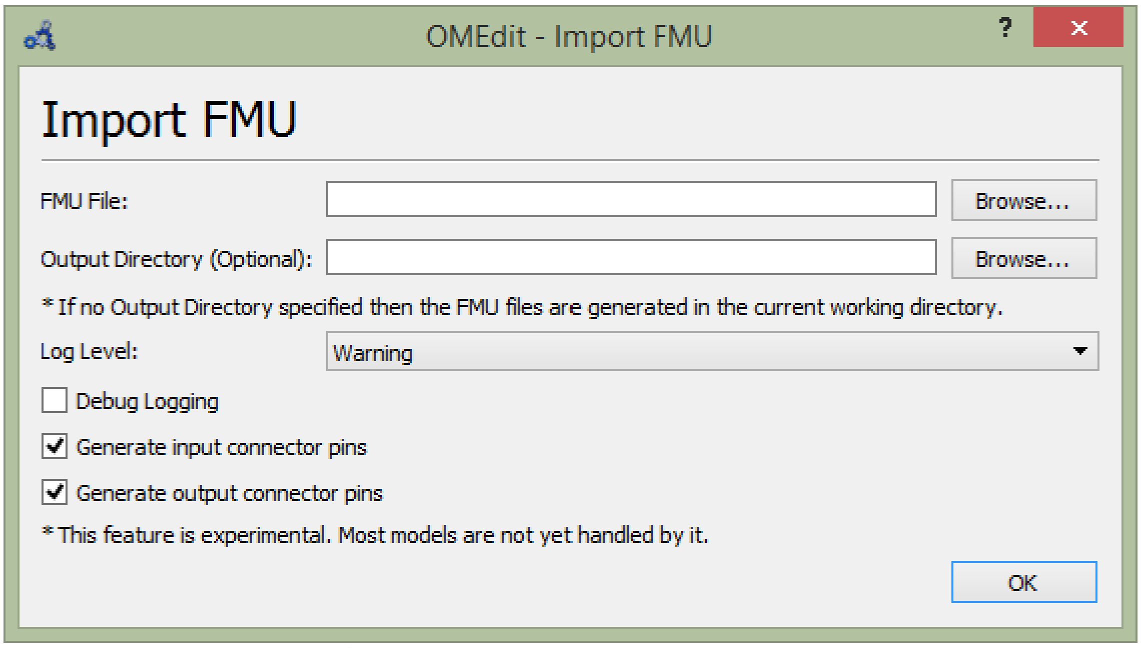

Modelica Code
OpenModelica Compiler
Translator, Analyzer & Optimizer
Code Generation
Model Description, DLL & FMI interface functions
FMU

Figure 2. FMI export process in OpenModelica.

OpenModelica imports the FMU to causal Modelica model with inputs and outputs. The Modelica model consists of external object which represents the FMU and external functions which are mapped to FMI API. Following OpenModelica API can be used to import FMU,

```
function importFMU
  input String filename "the fmu file name";
  input String workdir = "<default>" "The output directory for imported FMU files.
<default> will put the files to current working directory.";
  input Integer loglevel = 3
"loglevel_nothing=0;loglevel_fatal=1;loglevel_error=2;loglevel_warning=3;loglevel_
info=4;loglevel_verbose=5;loglevel_debug=6";
  input Boolean fullPath = false "When true the full output path is returned
otherwise only the file name.";
  input Boolean debugLogging = false "When true the FMU's debug output is
printed.";
  input Boolean generateInputConnectors = true "When true creates the input
connector pins.";
  input Boolean generateOutputConnectors = true "When true creates the output
connector pins.";
  output String generatedFileName "Returns the full path of the generated file.";
end importFMU;
```

You can also import using the OpenModelica Connection Editor (OMEdit) menu `FMI->Import FMU`,


OMEdit - Import FMU

Import FMU

FMU File: [ ] Browse...

Output Directory (Optional): [ ] Browse...

* If no Output Directory specified then the FMU files are generated in the current working directory.

Log Level: Warning

☐ Debug Logging

☑ Generate input connector pins

☑ Generate output connector pins

* This feature is experimental. Most models are not yet handled by it.

OK

Figure 3. FMI import in OpenModelica.

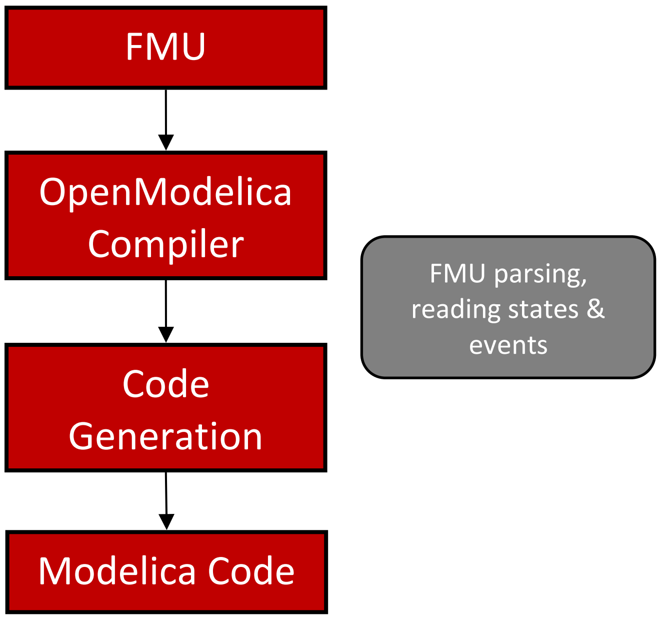


Figure 4. FMI import process in OpenModelica.

3. References

[1] The Functional Mockup Interface (FMI). [Online]. https://www.fmi-standard.org/.