INFORMATION TECHNOLOGY FOR EUROPEAN ADVANCEMENT

"D.6.6.1 - Report on Technology for Incremental Compilation of Modelica Models"

"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

Peter Fritzson (Section 1-4, 6)	Linköping University
Martin Sjölund (Section 1-4)	Linköping University
Hilding Elmqvist (Section 5)	DS AB
Karl Wernersson (Section 5)	DS AB

Executive Summary

Incremental compilation is a technique to avoid long turnaround time during software and model development. This is achieved by only compiling new or changed parts. Previously compiled parts can be re-used instead of re-compiling everything from scratch. This can also enable update, even on-line update, for example of compiled embedded controller code.

Incremental compilation can be done at different levels of granularity, e.g. single statement or equation, single function or basic model, or whole packages or modules. Advanced versions of incremental compilation even allow stopping execution, making some changes, and incrementally re-compiling while preserving most (or all) of the currently executing state. In that way, and incremental compiling system can give the same user experience as a fast interactive interpreted environment, while still keeping the high performance of compiled code.

Separate compilation of general models is quite different from separate compilation of functions. Different interfaces are needed to communicate states, derivatives, inputs, outputs, parameters, etc. Furthermore, special care is needed for handling of events in models. Provisions are needed to handle mutual dependencies between models, i.e. algebraic loops. All these aspects are the same as for coupling models from different modeling tools, i.e. the needs for FMI.

This report gives a short overview of several approaches to incremental compilation of Modelica models:

- Incremental compilation at the function level, supporting on-line changes.
- Separate compilation of packages of Modelica functions, i.e., algorithmic code
- Separate compilation of models to a fully acausal pre-compiled format.
- Separate compilation of models, using FMI export/import and the FMU pre-compiled format.

Contents

1 Introduction

The MODRIO 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.

What is incremental compilation? It is a technique to avoid long turnaround time during software and model development. This is achieved by only compiling new or changed parts. Previously compiled parts can be re-used instead of re-compiling everything from scratch. This can also enable update, even on-line update, for example of compiled embedded controller code.

Research on incremental compilation techniques was especially active in the 1980s, with the advent of interactive programming environments supporting compilation. Incremental compilation can be done at different levels of granularity. Fine-grained incremental compilation at the statement level, e.g. for online update of embedded code over slow communication lines, is described in [4] and [5], which also shows how a high-level interactive debugger can be integrated with an incremental compiler.

Incremental compilation at the code object level, for an object-oriented language, is described in [6]. The Cornell program synthesizer [7] presents an even more fine-grained incremental compilation/interpretation approach, integrated with a structure editor. Incremental compilation at the function level, also mixed with interpretation, is available in the rather early InterLisp [9]environment, as well as in the more recent Mathematica [10] environment.

The main 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 files 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 Incremental Compilation at the Function Level

Incremental compilation aims at providing even shorter re-compilation than separate compilation and often aiming at preserving the currently executing state. Such a mechanism is natural to be integrated with an interactive debugger.

For the algorithmic subset of Modelica, functions can be an appropriate granularity. An updated function can be compiled and linked into the running process. This is useful with interactive sessions where the execution state is kept between commands. Another important usage is interactive on-line update of compiled embedded controller code.

When a function has been incrementally compiled or re-compiled, it needs to be loaded into the executing, temporarily stopped, process. After re-compilation the existing function need to be replaced with the newly compiled version.

One way is to generate code that calls all functions indirectly through function pointers. There will be pointers to all functions in the application. After a function update, the relevant pointer(s) are simply updated to point to the newly compiled functions loaded into the executing process.

Thus design of the fine-grained incremental compilation at the function level has been done to generate portable code that can be updated on-line. More details on the OpenModelica implementation are available in [3]. As mentioned, 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 debugging. For this reason, the current design 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 the model editor 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).

2.1 An Implementation of Function-Level Incremental Compilation

The following (see also [3]) describes the changes performed on the OpenModelica compiler to begin supporting incremental compilation at the function level, 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 OpenModelica internally has the concept of both function pointers, built-in function pointers, closures, regular function calls, external function calls, and calls to record constructors. This makes the code run slightly 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 the OpenModelica compiler, 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
""
```

3 Separate Compilation of Packages of Functions

As a step towards incremental and separate compilation of full Modelica, we have designed and implemented separate compilation of a package structure containing Modelica functions, roughly the algorithmic subset of Modelica. More details and performance measurements are available in [2]. The separate compilation supports both Modelica and its MetaModelica extension, is very efficient, is implemented in OpenModelica, and is routinely used on large applications with 250 packages, with a total of more than 200 000 lines of code.

Compilation is performed in a few distinct steps, most of which can be run in parallel on a multi-core computer in order to speed-up compilation. The key to separate compilation is using interface files, which are regular Modelica files that have been stripped of information that is not needed in order to determine the types of public functions in the package.

3.1 Implementation Approach

In OpenModelica, implemented in MetaModelica, the full program is stored as a set of files (each top-level Modelica package is stored in a separate file), for example Absyn.mo and Main.mo. In the following we use the OpenModelica compiler as a large example use case.

We will introduce separate compilation of packages for both Modelica and MetaModelica, the same restrictions apply in both cases.

Note that only generation of functions (and constants) in packages is considered. There is no separate compilation of models.

Parsing the entire source code of the OpenModelica compiler in a single process takes 5.633 seconds (on the laptop where this document is being compiled). 5.633 seconds is too long to wait for simply parsing code that is going to be re-compiled. Parsing text should be a small part of the total compilation time.

By making the restriction that all top-level packages are encapsulated and the initial dot before a name is not allowed, all dependencies of a module must be marked by an import statement. This improves performance in subsequent steps.

An additional useful restriction is that any public function, constant, or type may only refer to other public elements. By introducing this restriction, it is possible to create an interface file for each package that strips out protected elements and algorithm sections. Everything that remains in the resulting file is part of the interface, and loading each and every file in the interface takes 0.660 seconds.

For performance, a distinction between protected and public import elements in Modelica has been introduced. When calculating the list of interface files that a package depends on, we must start with all the import elements in the package that is going to be compiled. For each of these packages, add the public imports to the list of packages that are going to be loaded (and perform this recursively if any of those packages contain public imports). This is the list of all interfaces needed to calculate all types used in functions of the package that is going to be compiled. This set is substantially smaller than loading every single interface file.

Thus, to compile a package, the interface dependencies and the main file is loaded. For each function in the package, perform instantiation of that function and send it to the code generator. Compile each of those files with a C compiler and perform linking together with some C code to start the garbage collector and call the function Main.main with a list of strings (the argument vector).

The steps in detail:

- Create Makefile targets needed to compile files (at first run only, and when adding new files to the compiler)
- Whenever a file is updated, also update its interface file (run in parallel, parses only the file that was updated)
- If the interface of a file changed, load all interface files and write out the dependencies of the program to a Makefile (if only bodies of functions are updated, or only protected elements change, this is run only once)
- Translate files from MetaModelica to C-code. Only changed files or files that depend on an updated interface file are translated. This step can be run in parallel and parses a minimal set of small interface files.
- Compile C-files to object code (in parallel). Furthermore, if the C code did not change after translation, the old object code is used instead of running the C compiler again.
- Run the linker

4 Separate Compilation of Modelica Models to an Acausal Pre-Compiled Format

Separate compilation of subset Modelica is already in industrial use. The IDA Simulation tool [8] from Equa Simulation AB is using a separately compiled model object code format since more than 10 years. Models (including acausal equation-based models) in model libraries are distributed in pre-compiled form. Therefore the user does not need any Modelica compiler and C-compiler in order to run simulations, only a standard linker. The separately compiled format has roughly the following properties:

- The supported language corresponds to a rather large subset of Modelica; but not all Modelica features are supported.
- During compilation, casualization is not performed. Instead sparse data structures and sparse system solvers are used. The generated code is on equation residue form. The casualization is essentially done at run-time by re-arranging coefficients in a sparse array. The pre-compiled models can thus be fully general acausal models. This approach also gives efficient simulations, but perhaps slightly slower compared to casualization performed at compile-time.
- The separately compiled format uses non-expanded arrays, i.e., array equations are not expanded to element-wise equations as done by almost all other Modelica tools. This has the advantage of easy and efficient scalability to models with very large arrays, and the additional advantage that the pre-compiled models can adapt to arrays of different sizes without recompilation. It also has a small disadvantage: in some cases, (primarily for small arrays), symbolic optimization is prevented which can give slightly lower simulation performance.
- Index reduction is not performed. The equation systems are of index 1. This works well for Equa's building applications, but would work less well for many mechanical or mechatronic applications.

5 Separate Compilation of Modelica Models using FMI

The FMI (Functional Mockup Interface) standard allows pre-compiling general Modelica models to causal C-code in FMUs (Functional Mockup Units), which in turn typically is compiled into object code, i.e., separately compiled code. Thus, if an application has the same causality constraints on a model component as the pre-compiled FMU representation of that model, this approach to separate compilation will work. A true acausal FMU representation is not yet available but is planned for a future upgrade of the FMI standard.

5.1 Goal

The goal is to select one or more sub-models in a large model, export them as an FMU, and import them back as FMU. It is expected that this approach can considerably reduce the compilation time of large Modelica models. For example when changes are made only to a control system, the complete plant (a detailed vehicle model, a detailed power plant model, or a detailed robot model) can be just compiled once and then reused. This approach requires that a user does not see a difference between the original Modelica sub-model and the imported FMU model, both in the user interface and the numerical properties (at least as long as the causality of the variables in the interface do not change).

5.2 FMI extensions to import FMUs without loss of information

In FMI 2.0 considerable improvements have been made for this to work (initialization, algebraic loops and event iteration over connected FMUs supported, all not possible in FMI 1.0). Still further extensions are needed:

(1) **Connectors**
The information about connectors/ports, especially physical/acausal connectors such as electrical pins, and bus connectors, must be stored in the FMU, in order that a re-import with this infor-

mation becomes possible. Furthermore, the information about the computational causality in the connector variables must be stored as well.

(2) **Local index reduction**
Index reduction might have been applied to the sub-model to be exported. This might add derivatives to connector variables, These variables must be stored in the FMU as well and it must be marked that these variables are differentiated variables of the connector variables, to be used for index reduction when re-importing the FMU.

(3) **Modelica annotations**
Modelica models and connectors have associated graphical information. When re-importing an FMU, this information must be used to get the same “look-and-feel”. Therefore, this information must be stored in the FMU as well. FMI 2.0 has prepared this by supporting tool specific XML definitions for every signal definition (element “annotation” in the XML schema definition). In order that all Modelica tools store and retrieve this information in the same way, the additional information about Modelica graphical annotations needs be standardized. There is an XML schema proposal by DS. This needs to be discussed in the Modelica and FMI standardization committees.

(4) **Hierarchical data structures**
FMI 2.0 supports only scalar variables, for simplicity. As a result hierarchical data structures and arrays have to be mapped to scalars and when importing an FMU in a modeling environment, the information about hierarchical data structures and/or arrays is basically lost. There are two different draft proposals to support hierarchical data structures and arrays in FMI. When re-importing an FMU, it is essential that the data structure is the same as in the exported FMU, and therefore a hierarchical data structure is essential. For multi-mode models (WP4), a sub-tree of variables associated with a modeling part may be activated or deactivated. It is essential to be able to identify such sub-trees uniquely during simulation. (1) and (2) are related to (4) and should be solved together.

(5) **Clocks and discrete states**
When exporting a Modelica model in the FMI 2.0 format, information about clocks and clocked variables is not supported and therefore lost. When re-importing such an FMI, this information is lost as well. As a result, clocks and clocked variables need to be supported in a future FMI standard, in order that any Modelica model can be exported as FMI and re-imported.
FMI 2.0 is currently only of limited use for embedded systems, one central application area of MODRIO. The reason is that FMI 2.0 basically describes physical systems with events. However, sampled data systems (as needed by embedded systems) cannot be directly defined. The goal is to introduce new types of variables: clocks and clocked variables (so variables associated uniquely with a clock). As a special case, discrete states will be introduced, which are clocked variables that need an initial value. Supporting clocks and clocked variables in FMI will allow describing sampled-data systems and especially, precise synchronization of clocks. Several different draft proposals from ITI, DLR and DS are available.

(6) **Differential-algebraic equations**
FMI 2.0 describes systems as ordinary differential equations in state space form. An FMU modeling a physical device usually contains algebraic equation systems. In FMI 2.0, these systems must be hidden in the FMU and the FMU is responsible to solve them. This approach has a severe drawback, because connected FMUs may result in algebraic loops over FMUs, and then algebraic loops have to be solved in a nested form. This can result in singular points and therefore the connected FMUs may have no unique solution due to the nested formulation, but would have a unique mathematical solution if the residue equations would be handled as one (non-nested) algebraic loop. Therefore, when re-importing an FMU in an model, it is essential to handle algebraic

loops in a cleaner way, in order to not destroy the numerical properties. Additionally, it is usually much better to formulate Nonlinear Model Predictive Control (NMPC) problems (see WP5) with Differential-Algebraic Equations (DAE), instead of Ordinary Differential Equations (ODE), because the sparseness structure is preserved for DAEs, and is usually destroyed for ODEs.
The goal is to introduce additional types of variables, **algebraic** and **residue** variables. The environment has to propose values for the algebraic variables, such that the residue variables become zero. This feature will allow describing DAEs. A sketch of this proposal is available. An open question is whether consistently overdetermined DAEs shall be handled as well that are the result of a DAE index reduction and introduce, for example, DAE invariants [5].

(7) **Hybrid Co-Simulation – Co-Simulation with Events**
FMI 2.0 for Co-Simulation is defined for a restricted class of systems: co-simulation of continuous-time systems and of sampled data systems with fixed sample period. The goal is to generalize co-simulation by combining it with FMI for Model Exchange: Time, state, and clock events can occur. At an event instant, the event handling for Model Exchange takes place. Between events, systems are simulated as co-simulation slaves with communication only at communication points. A sketch of this proposal is available. Once available, separate compilation gets the new option that a subsystem might be co-simulated when re-imported. As a result, the complete system can then run with different integration methods in different parts, which might give considerable speedup, if the original system has parts with largely different time constants.

6 Conclusion

This report presents an overview of different approaches to incremental and separate compilation of Modelica models. Future work includes generalization and integration of these approaches. Some of the mechanisms for function-based incremental compilation might also be useful for FMUs compiled into function form.

References

[1] Deliverable D4.1.3: FMI extensions for multi-mode DAE systems, ITEA2 project MODRIO (11004), M12 (1.0), Sept. 5, 2013.

[2] Deliverable D6.6.2 – M25 Prototype for separate compilation of Modelica models consisting of packages of functions using OpenModelica. ITEA2 project MODRIO (11004), M25, Sept, 2014.

[3] Deliverable D6.6.2 – M45 Prototype for Incremental Compilation of Modelica Functions and Separate Compilation of Modelica models to the FMI format using OpenModelica. ITEA2 project MODRIO (11004), M45, April, 2016.

[4] Peter Fritzson: Symbolic Debugging through Incremental Compilation in an Integrated Environment. The *Journal of Systems and Software* 3, 285-294, 1983.

[5] Peter Fritzson, PhD Thesis: *Towards a Distributed Programming Environment based on Incremental Compilation*. 161 pages. Dissertation no 109, Linköping University, April 13 1984.

[6] Boris Magnusson and Sten Minör. III—an integrated interactive incremental programming environment based on compilation. In *SigSmall Proceedings of the 1985 ACM SIGSMALL symposium on Small systems*, pp 235-244, ISBN:0-89791-154-7, 1985.

[7] Tim Teitelbaum and Tom Reps. The Cornell Program Synthesizer: A Syntax-Direct Programming Environment. *Communications of the ACM* 24, 9, Sept, 1981.

[8] Equa Simulation AB. The IDA Simulation Environment. Accessed 2016-04-25 from www.equa.se

[9] Warren Teitelman. *InterLisp Reference Manual*. Xerox Palo Alto Research Center, 1976.

[10] Stephen Wolfram. *The Mathematica Book*. Wolfram Media Inc. 1997.