CDL and CXF extension for user-extensions (that are not Composite Blocks)

Michael Wetter

Simulation Research Group

October 11, 2022

BERKELEY LAB

Requirements

Functionality

Users/developers must be able to add blocks other than composite blocks

- For CDL, behavior of block may be defined using declarations other than what is allowed for Composite Blocks, such as
  - use a state machine,
  - compile and use a C function,
  - call a function from a compiled library
- For CXF and simulations, computations may be done in “custom” language, such as
  - C,
  - a proprietary language,
  - a function in a compiled library…

Requirements

Interoperability

Users/developers must be able to add blocks other than Composite Blocks.

- For CDL, need mechanism so it works with any Modelica-compliant tool
- For CXF, can either design for
  - "Figure out yourself what the API is, and how to use it":
    But why would we define in an interoperable exchange standard something that is not interoperable?
  - Full interoperability,
    - using solutions developed for the FMI Standard (https://fmi-standard.org/):
      Compliant tools would need to implement an FMI for Model Exchange import (and export) functionality.
    - Any other if they exist?

Some challenges of allowing arbitrary external code.

How do we communicate what are the parameters, inputs and outputs?

What are the units, data types (32 bit, 64 bit), what are the variable names, units, min/max.

Can the function be called at any time stamp or only at fixed time steps?

Is the function self-contained or does it need any other software (library, executable, web services, …) to run?

How does a function signal messages to the user (message, warnings, errors)?

How does a function behave if it is fed invalid input?

What output depends on what input (if known, can close certain loops without requiring a solver).

Suggested approach for CDL

Suggested rules

1. Keep restriction of a Modelica block (same as for composite blocks).

2. Inside the block, allow anything from Modelica (then tools can export a Functional Mockup Unit, generate C code, or simulate it directly.)

This allows:

- Graphical block diagrams
- State machines
- Textual code
- Calling C or Fortran
- Calling a function in a compiled library
- Importing a Functional Mockup Unit that does the calculation
- Reuse existing tools (full compatibility with open source and commercial tools)

Suggested approach for CXF

Approach

1. Expose parameter, inputs and outputs in json format, identical to a composite block.
2. Add fields
    1. Location of external manifest (code, library, I/O and tool information)
    2. Mapping between CXF native values (parameters, inputs and outputs) and names in the external manifest

Suggested implementation:

Use FMI 2.0 for Model Exchange.

Widely supported standard (170+ tools) that has been developed for such a use case.

What is the Functional Mockup Interface Standard?

https://www.fmi-standard.org/

API standard to exchange and interact with simulators or models

ITEA project, 28 partners, 178 person years, 26 Mill. € budget, July 2008 - June 2011.

First version published in 2010.

The intention of the FMI project is to be neutral with respect to tools, technologies (e.g. solvers, OS, files, systems…) and languages (including Modelica).

Supported by 170+ tools.

$$C \frac{dT}{dt} = \dot{Q}$$

$$\dot{Q} = \dot{m}\, c_p \left(T_{in} - T\right)$$

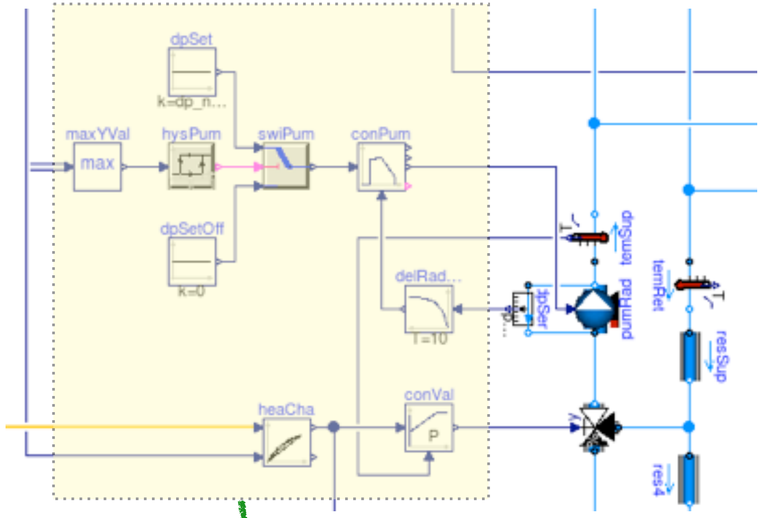
T_{in} C c_p $\left(\frac{dT}{dt}, T(t+\Delta)\right)$

C API
xml declaration

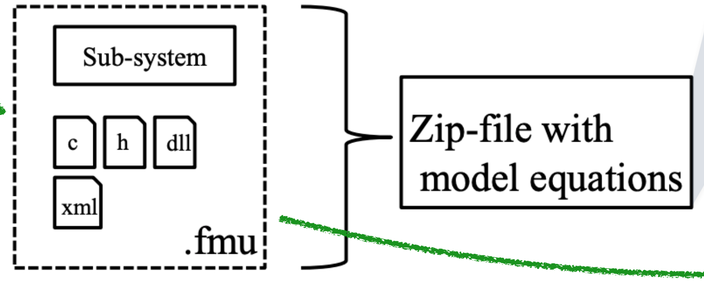
$\dot{m}$ $\dot{Q}$

A Functional Mockup Unit (FMU) is an input/ output block, with an API formally defined through the FMI standard.

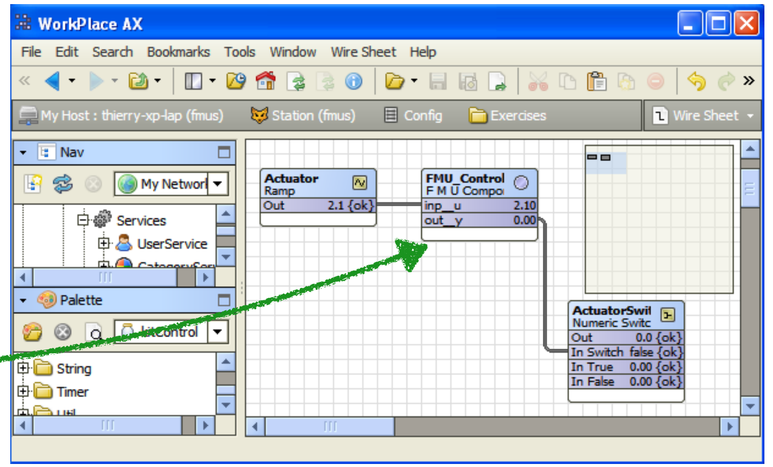
Exporting an FMU and executing it in Tridium Niagara



modelDescription.xml // Description of model (required file)
model.png // Optional image file of model icon
documentation // Optional directory with documentation
sources // Optional directory containing all C-sources
binaries // Optional directory containing the binaries
win32 // Optional binaries for 32-bit Windows
<modelIdentifier>.dll // DLL of the model interface
win64 // Optional binaries for 64-bit Windows
...
linux32 // Optional binaries for 32-bit Linux
<modelIdentifier>.so // DLL of the model interface
...
darwin32 // Optional binaries for 32-bit Darwin
<modelIdentifier>.dylib // DLL of the model interface
...
resources // Optional resources needed by model



Zip-file with model equations



Thierry Stephane Nouidui and Michael Wetter.
Linking Simulation Programs, Advanced Control and FDD Algorithms with a Building Management System Based on the Functional Mock-Up Interface and the Building Automation Java Architecture Standards.
Proc. of ASHRAE/IBPSA-USA Building Simulation Conference, p. 49--55, Atlanta, GA, September 2014.

Discussion and next steps