A Simulation Environment for Efficiently Mixing Signal Blocks and Modelica Components

Ramine Nikoukhah Masoud Najafi Fady Nassif

ALTAIR ENGINEERING, FRANCE, ramin@altair.com

Abstract

There exist several specialized tools that provide environments for the development and simulation of either pure Modelica models or pure signal based models. These environments have each their own advantages and flaws. solidThinking Activate™ has been developed to mix these domains and take advantage of both of these approaches to system modeling. This paper presents this mixed Signal-Modelica environment, and in particular the efforts and challenges faced in its development.

Keywords: Modelica tool, Signal based tool, FMI

1 Introduction

The Modelica® language[1] and tools are successfully used for modeling physical systems in industrial applications. This success is primarily due to the ability of Modelica to express mathematical equations corresponding to physical phenomena in a natural way (Modelica Association; Peter Fritzson).

For modeling complete systems, for example systems including controllers, Modelica provides other features that makes it go beyond a declarative language for expressing equations. Data types other than reals, algorithm sections, Matlab-like matrix operations are introduced to dispense of the use of other tools, in particular Matlab® and Simulink® for handling models with control components. Yet, still in many applications, the design process requires using Modelica to model the physical plant and exporting the model in the Matlab/Simulink environment for controller design. The reason for this is in part the limitations of the Modelica language, which is not well suited for creating block diagrams, such as the ones used in control applications, for which specialized tools such as Simulink, Scicos (Campbell et al., 2010), and solidThinking Activate™ have been developed.

In an attempt to provide an environment for modeling efficiently both blocks and physical components, in 2002 Modelica was introduced in the Scicos environment in the framework of the publicly funded project RNTL (Réseau National des Technologies Logicielles) **Simpa** (Simulation pour le Process et l'Automatique). This Scicos extension (Najafi et al., 2004, 2005a,b; Nikoukhah, 2006; Nikoukhah and Furic, 2009) allowed Scicos users to mix both standard Scicos blocks and Modelica components in the same environment. A similar extension was later introduced in Simulink with the introduction of the Simscape™ language (Simscape).

Scicos/Modelica environment based on the Modelicac compiler (Furic, 2007) provides a versatile modeling environment, especially thanks to the Coselica library[2]. Even though this extension allows Scicos users to use some Modelica components in the construction of their models, it has many limitations. For example Modelica libraries cannot be automatically imported and used in Scicos.

Activate is a professional simulation tool developed by Altair Engineering based on the open source academic simulation software Scicos. As such, it inherits many of Scicos features including the close integration with a matrix-based scripting and programming language. In Activate, the HyperMath Language (HML) has replaced Scilab[3] and NSP[4]. And for the Modelica extension, Scicos Modelicac has been replaced with the MapleSim™ compiler developed by Maplesoft[5] in Activate.

Activate and Scicos both use the same mechanism to integrate Modelica: at compile time, they aggregate Modelica components and create a Modelica program which is then processed by the Modelica compiler providing the C code corresponding to the simulating function of a block replacing these Modelica components in the original model[6]. The Activate environment however provides specific features that has allowed taking the Modelica integration beyond what is available today in Scicos. This paper presents this new modeling environment.

2 Motivations

It is widely agreed upon that for many applications Modelica today does not provide a viable alternative to block-based modeling tools such as Simulink, Scicos and Activate. The limitations imposed by the language make it difficult to provide the types of blocks that are needed

[1] http://www.modelica.org.

[2] http://www.kybdr.de/software.

[3] http://www.scilab.org

[4] https://cermics.enpc.fr/~jpc/nsp-tiddly

[5] http://www.maplesoft.com

[6] A noteworthy difference is that in Scicos this simulation function represents a DAE (Differential Algebraic Equations) forcing Scicos to use a DAE solver, whereas in Activate the simulation function is provided as a model-exchange FMU representing ODEs. This difference however is not relevant to the presentation here.

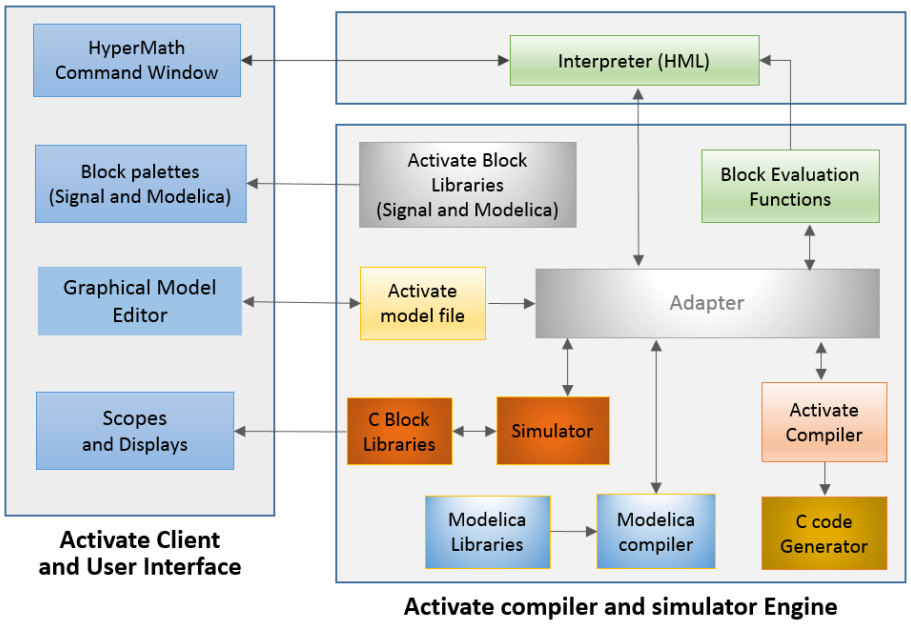

Figure 1. Different modules in Activate and their interactions.

to model control systems. For example creating a simple multiplexer block capable of concatenating a variable number of vectors and scalars of different data types is complicated in Modelica. Same is true for the summation block and many other basic mathematical operations (Elmqvist et al., 2016).

Other limitations come from the lack of a powerful supporting math environment. The computation of model parameters, post processing of the simulation results, etc., require access to math and engineering libraries, which could in theory be developed or interfaced in Modelica, but would require an enormous and lasting effort. In short it would amount to developing alternatives to Matlab, Scilab, HML, or Nsp, including their specialized toolboxes in control, signal processing, communication, optimization, etc. Some Modelica tools already use other languages, for example Maple and Python, for such support.

A reasonable solution to this problem is to base the simulation environment on a "User Language", preferably a matrix-based mathematical language such as Scilab, Matlab, Nsp, Octave, HML, or even on non-matrix based languages such as Python and LUA. The key point is to give users the ability to interact with the simulation model through this language for anything from block/component creation, model construction, parameterization, compilation, code generation and simulation to data collection, post processing, optimization, and more. The Scicos environment was developed in this spirit with Scilab as User Language. Matlab is the User Language for Simulink and Simscape.

A very interesting effort in this direction is undertaken in (Elmqvist et al., 2016), where a complete re-implementation of Modelica is considered with Julia[7] as User Language. This undertaking is very ambitious in that the Underlying Language is also used for defining the dynamics of blocks and components. The Activate/Modelica environment presented here is developed with this consideration in mind and follows the spirit of Scicos but uses HML as the Underlying Language. It does not go as far as defining dynamics of blocks in HML (except for embedded code generation purposes (Chancelier and Nikoukhah, 2015)); but rather it makes a clear distinction between the block/model creation and compilation, and runtime simulation. Model creation, evaluation and compilation, and in general anything that can be done before the start and after the end of runtime simulation are based strongly on the User Language. On the other hand the block dynamics need not be based on the User Language. The "standard" (Signal) Activate blocks have in general their runtime simulation functions expressed in C, and the equations of Activate physical components are expressed in Modelica.

This approach allows the Activate/Modelica environment to take advantage of existing technologies: Activate (synchronous semantics, block libraries, compiler, Simulink import (Weis, 2015) facility) and Modelica (existing Modelica compilers, in particular the MapleSim compiler, and existing Modelica libraries such as MSL).

3 Activate/Modelica environment features

Activate is not a Modelica tool per se; it cannot be used conveniently to build Modelica libraries. Its objective is to propose a unique harmonious environment to allow mixing regular Activate blocks and Modelica components in a

[7] http://julialang.org.

DOI 10.3384/ecp17132831

same model. The user interface and behavior of Modelica blocks and regular Activate blocks are designed to be as similar as possible without being too different from user interface of other Modelica tools. The Modelica components are seen as regular Activate block in this environment.

3.1 Modularity

A key architectural element in Activate is modularity. The diagram in Figure 1 shows different modules that constitute Activate. There are three main modules, the graphical user interface, the interpreter language, and the Activate engine. The graphical user interface and the interpreter can be replaced with similar modules fairly easily. For example the HML interpreter can be replaced with another interpreter, and alternative user graphical interfaces (for example javascript based tools) can be considered. The other module that can easily be replaced is the Modelica compiler. Currently the MapleSim compiler is used. In Scicos, Modelicac was used. Other compilers may be considered in the future.

The modularity between the engine and the graphical user interface is enforced by the usage of file based exchanges. The model, once edited is saved in an XML format and the engine uses this file to proceed with the compilation and simulation. The modularity of the interpreter is guaranteed through the specification of a set of APIs for the exchange with the graphical user interface and the engine.

3.2 Double layer implementation

In the Activate environment, a model is constructed using blocks. The compiler however does not operate on these blocks; it interacts with Atomic Units (AU). In many cases a block is associated with a single AU, but not always: a block may produce a network of AUs. The AU or AUs produced by a block may depend on the values of the block parameters. Specifically, the choice of the AU(s), their parameters, and the topology of the network is specified by an HML function associated with the block based on the values of the block parameters.

The ability to programmatically instantiate an AU or a network of AU(s) is an elementary feature in Activate but provides a particularly useful functionality in the context of Modelica components, as it will be described later.

Atomic unit (AU)

An AU may be presented as a "basic" block, but this would be misleading. An AU has ports that are connected to links, just like a block. It has parameters, like a block, but these parameters are not in general the block parameters. Consider for example the Activate block that implements a transfer function. The block parameters are the numerator and the denominator coefficients of the transfer function. The AU associated with this block operates in time domain and implements the dynamics based on the state-space realization of the transfer function. The parameters of the AU in this case are the A, B, C, D matrices, which are computed by the HML function associated with the block.

In general an AU is a computational unit providing APIs to be used by the simulator. The APIs are C functions that are called by the simulator at different stages of the simulation: computation of the output, of the state derivative, of the next discrete state, etc. But the AUs can also be Modelica components. An AU may also be virtual.

The creation of AUs from Activates blocks based on a User Language script is a process that does not have an equivalent in standard Modelica or in Simulink (S-Functions). This process, which provides a clear separation between the model at the graphical layer and at the compiler layer, has been first implemented in Scicos.

3.3 Modelica components

In Activate, Modelica components are Activate blocks and treated as such in the graphical editor. They are also treated similarly at the evaluation phase, prior to compilation. This means that certain properties of Modelica components that are coded as annotations are handled by the corresponding Activate XML file and HML evaluation script. These properties include in particular the graphical properties and the parameter descriptions. When a Modelica library is imported into Activate, these component annotations are used to create the Activate blocks. These annotations are never directly used in Activate.

So, having the Modelica component as an Activate block means that all graphical features, parameter definitions, code instantiations, ..., are done in the usual Activate way. The use of Activate block to instantiate the Modelica components provides facilities that allows for example the creation of components with variable number of ports or different data types based on block parameters. The Activate block is thus a lot more versatile than a standard Modelica component; even the internal Modelica code of the block/component can be customized. At the extreme case, the Modelica code itself could become a block parameter.

On the graphical editor, the visible difference between a regular Activate block and a Modelica Activate block is that the latter has special (implicit) ports. No connections can be made between these ports and other Activate port types. Two special interface blocks are used to interface the Modelica world with the regular Activate world. One has an implicit input port and a regular output port and the other, the opposite (see Figure 2). Such connections are meaningful only if the the connection on the Modelica side is of type Modelica Signal.

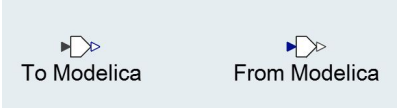

Figure 2. Special blocks for Modelica-Activate world interface

Importing Modelica libraries

The import of a Modelica library is done by the MapleSim compiler, which creates HML scripts, the execution of which create the corresponding Activate library. An Activate library is a collection of XML files, HML functions, image icons and palettes. The MapleSim compiler also uses the definition of component icons described in Modelica language to generate image files (`svg` format) to be used by Activate as block icons. Certain features such as dynamical icons (icons changing during simulation) are not supported.

Currently, most but not all MSL (Modelica Standard Library) blocks are imported and integrated in Activate palettes.

3.4 Model compilation

Compiling a model consists of producing a structure to be used by the simulator. This structure contains all the information needed by the simulator that can be computed before the start of the simulation. It contains in particular type and size information, and scheduling tables specifying the condition and the order in which AU computational functions are to be called during simulation. The same structure is used for code generation.

Model evaluation

The evaluation is the first phase of model compilation. In this phase, the model parameters are evaluated and the HML function associated with the blocks are executed producing the network of AUs associated with the model. Note that this network of AUs, which retains a hierarchical structure, does not in general present a one to one correspondence with the original block diagram model.

At the end of model evaluation phase, all model and block scripts and parameters are removed. They are used in this phase to construct the AUs and evaluate the numerical values of their parameters. They are not available or needed for the rest of the compilation process, which deals exclusively with the network of AUs.

Model flattening

Model flattening is the second phase of the compilation. The hierarchical network of AUs produced by the model evaluation phase is converted into a flat network of computational units. All virtual AUs are removed and all Modelica AUs have been replaced with computational AUs (in particular derived from an FMU produced by the Modelica compiler).

A simple example is provided in Figure 3. This model contains an electrical circuit, modeled for the most part using Modelica components. The regular Activate blocks are the sine wave generator and the Scope. There are three interfacing blocks connecting the Activate environment to the Modelica environment.

The Modelica part is aggregated into a single block as shown in Figure 4. This step is of course fully transparent to the user and is presented here as an illustration of the way the mechanism operates. The newly created block has one input and two outputs, as expected.

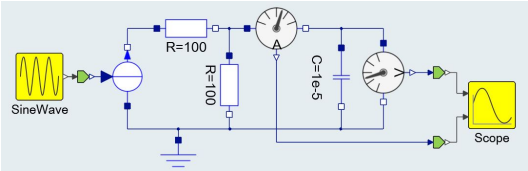

Figure 3. Simple Activate diagram containing Modelica components.

Figure 4. Equivalent Activate model after aggregation of Modelica components.

The Modelica code corresponding to the Modelica part is generated automatically by Activate and sent to the Modelica compiler for compilation. The Modelica compiler then generates a corresponding FMU, which replaces the Modelica part as shown in Figure 5. This step is of course again transparent to the user and is presented here as an illustration

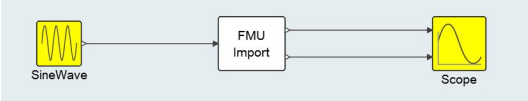

Figure 5. Resulting regular Activate model with no Modelica components.

Back-end compiler

In this phase, which consists of computing the scheduling tables for the simulator, the structure contains no trace of the Modelica components; they have been replaced with computational AUs in the previous phase. So the introduction of the Modelica extension does not affect this phase.

4 Modelica integration through FMI

The way Activate handles the Modelica components is by grouping them into a single Modelica model with inputs and outputs that are clearly specified by special interfacing blocks, as presented in the previous section. In the Modelica code generated by the Activate compiler, the interfacing blocks (shown in Figure 2) are instantiated as

```
Modelica.Blocks.Interfaces.RealInput
Modelica.Blocks.Interfaces.RealOutput.
```

The Modelica model is then compiled by the Modelica compiler, which in turn generates a code executable in Activate. This code is then imported in the Activate model as an FMU to replace the Modelica part. The FMI has been

DOI 10.3384/ecp17132831

chosen as the exchange format because it is a standard already supported both by Activate and MapleSim.

The FMI format is a rich interface format and quite compatible with Activate and Modelica. There are however a few shortcomings that need to be considered. Some challenges encountered in the usage of FMI standard in this context is discussed in this section.

4.1 Choice of the FMI type: Model-Exchange or Co-Simulation

The Modelica part of the Activate model is converted into an FMU and imported as a regular Activate block. In the exported FMU, both Model-Exchange and CoSimulation implementations are available. Using the model-exchange implementation allows taking advantage of different numerical solvers of Activate. The co-simulation implementation is useful for complex models where different parts of the model are needed to be simulated separately or even in parallel. Currently only the model-exchange implementation is used in Activate.

4.2 FMI import preserving full output/input dependency property

A challenge in importing the FMI generated from the Modelica code (or more generally any FMI) in Activate is the treatment of output/input dependencies. In the Activate block (or more specifically its AU) output/input dependencies are expressed as a vector of dependencies specifying which inputs affect any of the outputs. So the dependency is solely a property of an input port. The reason is that an AU computes all of its outputs in the same call, so all its dependent inputs must be up to date when the call is made. An FMU on the other hand specifies output/input dependencies as a matrix specifying which output depends on which known variables including individual inputs. The FMU provides routines that allow the computation of output ports separately and take advantage of variable caching.

A way to deal with this situation, which is the way the Modelica extension is implemented in Scicos, is to simply project the matrix of dependencies into a vector. This conservative approach properly assigns dependencies in Activate but "loses" information along the way. This may lead in particular to detection of algebraic loops by the Activate compiler that are not true algebraic loops (artificial algebraic loops). Even though there are ways to break algebraic loops in an Activate model, it is not the best way to deal with this situation. A very simple example that illustrates this problem is shown in Figure 6.

After compiling the Modelica part, a model similar to what is shown in Figure 7 is obtained in Activate. In the generated FMU, there is a direct dependency between the `SignalCurrent` input port (**in**) and the `CurrentSensor` output signal (**A**). The dependency is depicted by a red dashed line in Figure 7. If the dependency matrix is projected into a vector, both the output ports **A** and **V** are considered depend on the input port **in**, which results in an artificial algebraic loop.

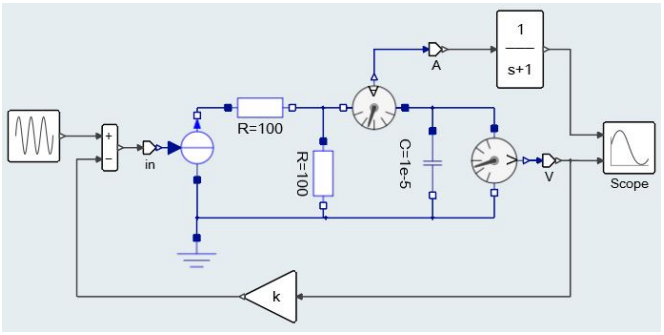

Figure 6. A simple model mixing Modelica and Activate blocks

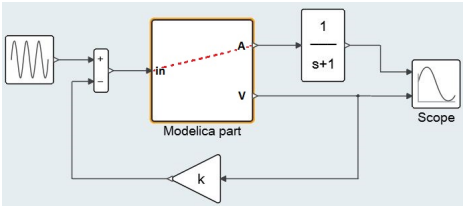

Figure 7. The model in Figure 6, after converting the Modelica part into an FMU block.

There is no solution to this problem as long as the FMU block implements a single AU. But as it was stated previously, Activate blocks can implement a network of AUs, the topology of which can depend on block parameters. It turns out that the matrix output/input dependency can be properly implemented by a properly constructed network of AUs to implement the FMU.

In this case the block parameters are provided by the FMU XML file. By reading and parsing the XML inside the FMU, the block generates a network of AUs, as shown for example in Figure 8 in the case of a 2 input 4 output FMU block. The network contains a central AU, always present, and an AU associated with each output port. The input dependency associated with an output is specified in the AU associated with that output. In this particular example it can be seen that the first output depends on both inputs whereas the second output has no input dependency, the third output depends only on the first input and the last output depends on the second input.

The central AU includes the simulation APIs for state derivative computation and discrete state updates and does not have any input dependency. All the AUs in the network use the same internal structure, which is instantiated by the central AU. The central AU provides a pointer to this structure to the other AUs through its output port.

In the case of the model in Figure 6, the network of AUs is generated as in Figure 9. By using this network to replace the FMU block, the resulting Activate model contains no algebraic loop.

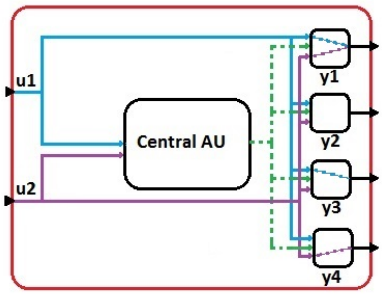

Figure 8. Automatically generated network of AUs from FMU import for an FMU with two input and four output ports.

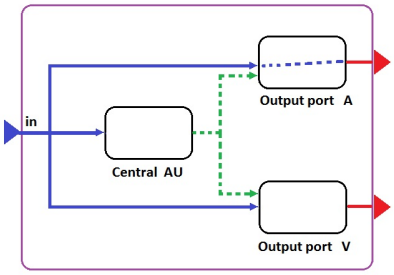

Figure 9. The network of AUs corresponding to the example in Figure 6.

4.3 DAE support and constraints on states

The current FMI standard is powerful enough to be used for implementing the Modelica extension in Activate for many situations but some extensions would be particularly useful.

Compiling complex Modelica models, in particular mechanical models, very often results in high index DAEs or sometime ODEs and DAEs with constraints. Keeping the constraints valid is important to avoid drift in the solution. In the current FMI specification, only ODEs are supported. Activate currently supports both DAEs, and ODEs with constraints. But these solvers cannot be used for the Modelica extension since the FMI does not support DAEs and ODEs with constraints.

The DAE support is currently being considered for FMI. ODEs with constraints, should also be considered. If it is known that an ODE $\dot{x} = f(x)$ satisfies a constraint $C(x) = 0$, information that could be available in various scenarios, then the solver should take advantage of this information to reduce drift in the solution. The constraint information may be provided as a residual function returning the constraint value, i.e., $C(x)$, or as a projection function such as $J^T(JJ^T)^{-1}$ where $J = \frac{\partial C}{\partial x}$.

This FMI extension can be done in several ways. One way would be to add one of these APIs to FMI interface:

```
fmi2Projection(fmiComponent c, double *J)
fmi2Constraint(fmiComponent c, double *C)
```

If the second API is used, then the number of constraints should also be declared as an attribute in the XML file inside the FMU.

Another way is to add a new function to the set of FMI APIs in order to bring back the solution on the constraint after each completed integration step.

```
fmi2ApplyProjection(fmiComponent c)
```

This function would apply a near-minimal projection to the continuous states in the model. This is often done via a Newton-based method, and terminates when it achieves the desired precision. This method can be applied on single-step solvers where memory of the past solution is not used. It will be necessary to call `fmi2GetContinuousStates` after the projection to obtain the continuous states satisfying the solution. Having this as a separate function allows the simulator to choose when it is applied (e.g. at the end of an integration step, internal to the step, after events, etc.).

A third way, which does not require adding a new API, a projection is implicitly applied when `fmi2CompletedIntegratorStep` is called by the simulator. This solution would work only with single-step solvers. No error tolerance control can be used on the constraints in that case.

4.4 Handling input derivatives

Consider the simple example shown in Figure 10. In this model the derivative of the input is required.

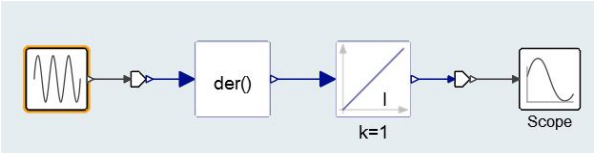

Figure 10. model requiring the derivative of inputs

When the time derivative of an input is required, the derivative can be computed numerically inside the FMU, but this does not always work for variable-step size solvers since the derivative value is not necessarily stable as the integrator step-size changes. Furthermore, at initial step or just after an event that changes the internal model configuration, no derivative can be computed. If there are constraints that depend on these derivatives, the integration step rapidly reduces to zero, stalling the simulation. In FMI for CoSimulation, the derivative of inputs can be provided via the API `fmi2SetRealInputDerivatives`, but nothing is available for ModelExchange FMI. The only robust alternative currently is to add an extra input port to provide the derivative of input from the environment, if available.

4.5 Using the Jacobian of the FMU

The numerical solvers often need the Jacobian of the model for numerical integration. The Jacobian can either be provided analytically or computed numerically. In

DOI 10.3384/ecp17132831

complex models providing the analytical Jacobian is crucial for obtaining reliable results. The FMU block[8] may provide directional derivatives of its state derivatives and outputs with respect to its states and inputs. These directional derivatives can be used to compute the equivalent linear model of the block. If an FMU block with nonlinear dynamics defined as (1) provides its directional derivatives

$$\begin{cases} \dot{x} &= f(x,u) \\ y &= g(x,u), \end{cases} \tag{1}$$

The matrices (A,B,C,D) as defined in (2) can be obtained by repeated calls to `fmi2GetDirectionalDerivative` function in FMI.

$$\begin{array}{ll} A = \frac{\partial f}{\partial x} & B = \frac{\partial f}{\partial u} \\ C = \frac{\partial g}{\partial x} & D = \frac{\partial g}{\partial u} \end{array} \tag{2}$$

The (A,B,C,D) matrices are equivalent linear system of the FMU block. The numerical solver, on the other hand, needs the complete Jacobian of the entire model which may be composed of other FMU blocks and other regular Activate blocks. In order to obtain the complete Jacobian of the model, Activate offers the following solutions.

- Computing a pure numerical Jacobian, i.e., ignoring the local analytical linear system of blocks and compute the complete Jacobian of the model using the numerical differentiation method. This method usually works fine and it is fairly fast, but may fail for complex stiff models.
- Mixing numerical and analytical Jacobian. In many cases, the highly nonlinear part of the Jacobian of the model is present in matrix A of the block. The analytically obtained matrix A of blocks may be used to populate the Jacobian matrix of the model, then the rest of the Jacobian matrix can be filled numerically. This methods works fine, and is the default method in Activate.
- Fully analytical method. This method which is more complex than other two methods is useful if all blocks provide their analytical equivalent linear system matrices (A,B,C,D). Since this method does not require calling the $f(x,u)$ and $g(x,u)$ function in (1), it is useful when calling these functions is expensive.

5 Challenges

Activate is not a Modelica tool and cannot provide the same Modelica functionalities as do pure Modelica tools such as Dymola or OpenModelica. Modelica is an extension for the modeling and simulation environment Activate. Efforts have been made to provide a user-friendly interface both for native Activate users as well as Modelica component users in this environment. There are currently a number of limitations in this extension.

[8] Only FMI-2.0 blocks provide directional derivative.

Modelica expressions, records and functions

The parameters of Modelica components present in an Activate models follow the scoping rules of Activate. So the records and functions used in the definition of parameters in Modelica are not always consistent with the way Activate handles parameters. This creates a complex problem for importing Modelica components. A translator of expressions is being developed to deal with this issue. For importing models, the records should be converted into HML scripts to be placed in Activate diagram contexts. This is a complex task, in general, but solutions have been found in special cases.

Initial equations

Initial equations in Modelica are global information that are not related to a specific component. Adding such information, even in specialized Modelica tools, cannot be easily done in the user interface and must be added textually. Since Activate does not provide a textual interface, the addition of initial equations currently is not possible. Various solutions are being considered but for the moment Activate does not allow the definition of initial equations in models. Initial equations in library components are of course handled by the compiler as usual.

6 Conclusion

Activate provides a complete environment for modeling systems with both physical components and signal based control parts where the physical components are modeled in Modelica. The integration of Activate and Modelica is done by respecting the semantics of the two languages. But there remain issues for going towards full Modelica support. This paper has presented the Modelica extension in Activate and the issues that remain open.

References

Stephen L. Campbell, Jean-Philippe Chancelier, and Ramine Nikoukhah. *Modeling and Simulation in Scilab/Scicos with ScicosLab 4.4*. Springer-Verlag New York, 2010. ISBN 978-1-4419-5526-5.

Jean-Philippe Chancelier and Ramine Nikoukhah. A novel code generation methodology for block diagram modeler and simulators scicos and VSS. *CoRR*, abs/1510.02789, 2015. URL `http://arxiv.org/abs/1510.02789`.

Hilding Elmqvist, Toivo Henningsson, and Martin Otter. Systems modeling and programming in a unified environment based on julia. In *Proceedings of the ISoLA 2016 - 7TH International Symposium On Leveraging Applications of formal methods, verification and validation; 2016*, pages 198–217, 2016.

Sébastien Furic. Using modelica under scilab/scicos, 2007. URL `http://www.scicos.org/ScicosModelica/Formation/Documentation/IntroductiontoModelica.pdf`.

Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. Wiley, 2014. ISBN 9781-118-859124.

Modelica Association. The Modelica Language Specification, Version 3.3 Revision 1, 2014. URL `https://www.modelica.org/documents/ModelicaSpec33Revision1.pdf`.

Masoud Najafi, Azzedine Azil, and Ramine Nikoukhah. Extending scicos from system to component level simulation. In *Proceedings of the ESMc2004 international Conference;, Paris; France; October, 2004*, 2004.

Masoud Najafi, Sébastien Furic, and Ramine Nikoukhah. Scicos: a general purpose modeling and simulation environment. In *Proceedings of the 4th International Modelica Conference; Hamburg; 2005*, 2005a.

Masoud Najafi, Ramine Nikoukhah, Serge Steer, and Sébastien Furic. New features and new challenges in modeling and simulation in scicos. In *Proceedings of the IEEE conference on control application; Toronto; Canada; August, 2005*, 2005b.

Ramine Nikoukhah. Challenges in integrating modelica in the hybrid system formalism scicos. In Claude Gomez Shi Li, Long-Hua Ma, editor, *The Oxford Handbook of Innovation*. Tsinghua University Press, Beijing, 2006.

Ramine Nikoukhah and Sébastien Furic. Towards a full integration of modelica models in the scicos environment. In *Proceedings of the 7th International Modelica Conference; Como; Italy; 20-22 September 2009*, pages 641–645, 2009.

Simscape. Physical systems simulation. URL `https://www.mathworks.com/products/simscape.html`.

Pierre Weis. Simport: A simulink model importer for scicos. In *Proceedings of The 3rd International Workshop on Simulation at the System Level for Industrial Applications; Ecole Normale Supérieure de Cachan, France, October*, 2015.

 DOI 10.3384/ecp17132831