Simulation of high-index DAEs and ODEs with constraints in FMI

Masoud Najafi

Altair Engineering, France, masoud@altair.com

Abstract

In the current FMI standard the dynamical behavior of a model can only be defined as a system of Ordinary Differential Equations (ODE). The dynamics of many physical systems, such as the equations of motion of constrained mechanical multibody systems, are expressed by high-index Differential Algebraic Equations (DAE) so they cannot be simulated directly using standard ODE or DAE solvers. These systems can be converted through index-reduction into ODE or index 1 DAE systems. However FMUs based solely on these latter systems suffer from drift in hidden constraints on the states. As a consequence, the simulation may results in physically meaningless solutions. In this paper, we propose an extension of the FMI standard to handle DAE Systems of index 1 or higher and ODE with constraints. This FMI extension requires only few additions to the FMI specification, all of which can be omitted for FMUs that represent ODE systems or FMUs that do not support DAE handling. The extension has been implemented in solidThinking Activate™ and two examples that illustrate the ease of implementation and the effectiveness of the method will be discussed.

Keywords: Modelica, FMI, High index DAE, ODE with constraints, Coordinate Projection

1 Introduction

Activate is primarily a signal-based modeling and simulation environment, but it supports also the Modelica language. Modelica components can be mixed with standard signal blocks in a same diagram. Activate formalism proposes a unique harmonious environment in which signal-based Activate blocks and Modelica components can co-exist in a same model.

In order to simulate an Activate model, the model should be compiled. 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 all signal types and sizes information, in addition to scheduling tables specifying the condition and the order in which the computational functions of the blocks are to be called during simulation.

The way Activate compiler 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. This Modelica model is then compiled by the Modelica compiler (the MapleSim™ Modelica compiler is used in Activate), which in turn generates an FMU for ModelExchange to replace the Modelica part. The FMI has been chosen as the exchange format because it is a standard already supported both by Activate and MapleSim. The ModelExchange implementation is used because it allows taking advantage of different numerical solvers available in Activate.

A simple example is provided in Figure 1. 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 (green blocks) connecting the Activate environment to the Modelica environment.

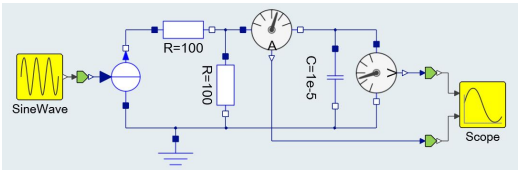

Figure 1. Simple Activate diagram containing Modelica components.

The Modelica part is aggregated into a single block as shown in Figure 2. This step is of course 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.

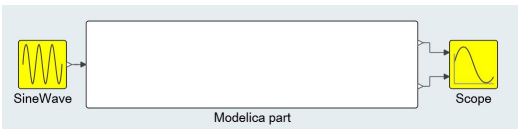

Figure 2. Equivalent Activate model after aggregation of Modelica components.

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 3. This step is of course again transparent to the user and is presented here as an illustration. Interested readers are referred to (Nikoukhah, 2017) for more details.

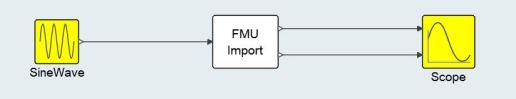

Figure 3. Resulting regular Activate model with Modelica parts replaced by an FMU block.

The compiling process requires that any Modelica model can be converted into FMU. The current FMI standard allows this conversion for many situations but in some cases an extension of this standard would be 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 and making sure they are satisfied is important to avoid drift in the solution. In the current FMI specification, only ODEs are supported. Activate currently supports, ODEs, index 1 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.[1]

Consider the overdetermined system:

$$\dot{x} = f(x) \quad x(t_0) = x_0 \tag{1a}$$
$$0 = \phi(x), \tag{1b}$$

The constraint (1b) is supposed to be consistent with the ODE (1a) in the sense that the solution of this ODE satisfies (1b). So theoretically, Constraint (1b) is redundant. However for numerical simulation, it provides valuable information that can be used by the solver to reduce numerical errors. This can be done by keeping $\phi(x)$ close to zero.

Such constraints may be available in different scenarios. For example in a conservative physical system, where the total energy is conserved, the conservation of energy may be expressed as such a constraint. But the scenario that is of particular interest here is when the ODE (1a) is obtained by differentiating algebraic equations such as (1b). This is done when the original system is a DAE. The algebraic equations are differentiated until an ODE is obtained so that an ODE solver can be used for simulation. In such cases ignoring the original algebraic constraints results often in unacceptable drift in the numerical solution of the system.

[1] The DAE support is currently being considered in FMI design meetings for next releases of FMI.

In the current FMI 2.0 standard, which supports only ODEs, a way to impose the constraint and avoid drift in the solution is to trigger step-events. At solver steps, *i.e.,* at `fmi2CompletedIntegrationStep` calls, the constraint can be checked, and if the error is found to be larger than some user defined tolerance, a coordinate projection method (will be discussed in 2.3) can be performed bringing back the state of the system on the constraint manifold. Then `fmi2CompletedIntegrationStep` should report a step-event, *i.e.*, `enterEventMode=fmi2True`. The simulator treats the step-event and the FMU reports a change in the value of continuous-time states, *i.e.*, `valuesOfContinuousStatesChanged= fmi2True`. A change in the value of continuous-time states usually requires restarting the numerical solver (specially multi-step solvers) with smaller step-sizes and lower order methods which slows down the simulation. This way of treating constraints works but due to large number of step events, the use of multi-step solvers and in some extends single-step solvers becomes uneconomical.

We have extended the current FMI APIs to provide functionalities to take into account the system constraints. This information can be used by Activate or any importing tool to correct the solution so that the constraints are satisfied.

In the following sections, first high index DAE and the coordinate projection method will be discussed. Then our extension of the FMI standard to avoid drift and obtain fast simulation will be discussed. Finally, two test examples to illustrate the advantages of this FMI extension will be presented.

2 DAE Description

DAE systems arise in many applications such as constrained mechanical systems. One attribute of DAE systems is the differentiation index of the system, which can be defined as the number of differentiations of each equation necessary to convert the system into an ODE system. For the sake of simplicity, in this paper, index is used instead of differentiation index. ODE can be considered as an index 0 DAE. Further information on DAEs, and numerical methods for DAEs, can be found in (Ascher and Petzold, 1988) and (Hairer and Wanner, 1996).

2.1 index 1 DAE case:

As an example, consider the following system

$$\dot{x} = f(x,y) \tag{2a}$$
$$0 = g(x,y), \tag{2b}$$

DOI 10.3384/ecp18148213

Differentiation of (2b) once gives:

$$\dot{x}\frac{\partial g}{\partial x}+\dot{y}\frac{\partial g}{\partial y}=0 \tag{3}$$

If $\frac{\partial g}{\partial y}$ is not singular, then (2) with (3) can be used to compute the values of $\dot{x}$ and $\dot{y}$. Hence we now have an ODE system. Equation (2) is an index 1 DAE, because one differentiation yields an ODE. This process (of differentiation to obtain an ODE system from a DAE system) is called index-reduction (Pantelides, 1988).

Index 1 systems can be treated in this way, but this introduces a constraint (*i.e.,* $g(x,y)=0$) that will not be taken into account when using a standard numerical ODE solver.

An alternative approach for index 1 problems is to treat the original system as-is, using the equation (2a) to solve for $\dot{x}$, and treating y as a purely algebraic variable, to be solved using the equation (2b). Solution of this system requires modifications to standard ODE solvers to accommodate the algebraic variables. Ideally these variables should have some error control measures applied that is similar in effect to the error control on $\dot{y}$ of the index reduced system. Advantages of the direct approach are twofold. No unnecessary state y is introduced. Fewer constraints is always better, both making the error through constraint handling smaller, and in this case removing the need for constraint handling altogether. But this method needs the numerical integrator to be modified to accommodate error control on algebraic variables. There are also standard DAE solvers such as DASSL, IDA[2], or RADAU-IIA[3] that can take the equation (2) as input and solve it over time. In these DAE solvers, consistent initial values of x and y are provided by the user. Some solvers can help the user to initialize the DAE by solving the initialization equation. In this case, the user should indicate which variable are differential and which are algebraic.

There is also an alternative approach which is usually used for FMU export. In this approach the y variable is left as an internal variable, and only the x is exposed. This has the advantage that an FMU constructed in this way can be used directly with an ODE solver. In our extension of the FMI specification for handling of algebraic variables, the algebraic section can be safely ignored, and a pure ODE solver can be used for the FMU.

2.2 Higher index DAE case:

For higher index systems, constraints cannot be avoided, even if only performing index-reduction to make the system index 1 DAE or ODE, so a mechanism for handling constraints is required. As an example, consider the unit length planar pendulum in Cartesian coordinates, which can be expressed by the following equations:

$$\begin{cases} \ddot{x} &= Fx \\ \ddot{y} &= Fy-g \\ 0 &= x^2+y^2-1 \end{cases} \tag{4}$$

One differentiation of the constraint in x, y gives:

$$0=2x\dot{x}+2y\dot{y}$$

And a second differentiation gives:

$$0=2x\ddot{x}+2y\ddot{y}+2\dot{x}^2+2\dot{y}^2$$

which after replacing $\ddot{x}$ and $\ddot{y}$ from (4) and simplification we get:

$$0=2Fx^2+2Fy^2-2yg+2\dot{x}^2+2\dot{y}^2$$

which gives

$$F=yg-(\dot{x}^2+\dot{y}^2). \tag{5}$$

In order to fully index reduce this model, one further differentiation of (5) would be needed to obtain an equation that can be used to get $\dot{F}$ as a function of other states, so this problem is an index 3 system.

Leaving F in algebraic form, *i.e.*, keeping (5) in the system, instead of its derivative) gives us the following system of equations

$$\begin{cases} F &= yg-(\dot{x}^2+\dot{y}^2) \\ \ddot{x} &= Fx \\ \ddot{y} &= Fy-g \end{cases} \tag{6}$$

where x, y, $\dot{x}$, $\dot{y}$ are differential states and F is the algebraic variable. The two hidden constraints on the states are:

$$\begin{cases} 0 &= x^2+y^2-1 \\ 0 &= 2x\dot{x}+2y\dot{y} \end{cases} \tag{7}$$

In order to solve such a system with constraints, various approaches are possible:

- Simply treat the ODE and index 1 portion of the system ignoring the hidden constraints. Problem: Over time the solution will drift away from the constraints giving an inaccurate or even non-physical solution for the model.
- Use Baumgarte constraint stabilization (Baumgarte, 1972), by adding correcting terms to the ODEs. Problem: This can only reduce (not eliminate) the drift for the problem. Furthermore, the parameter values for Baumgarte are not known in advance.
- High index DAEs may also be handled with FMI in some special cases. For example, if the FMU is exported from a Modelica model

[2] https://computation.llnl.gov/projects/sundials/ida

[3] https://www.unige.ch/~hairer/software.html

and the modeler has enough knowledge about the states of the model, by using `stateSelect = StateSelect.always` in an appropriate way, it is possible to transform the Modelica model to ODE and export it as FMU. There are several drawbacks. First, the user needs to have a good knowledge about the model to provide appropriate `stateSelection`. Also, nonlinear algebraic equations might need to be solved inside the FMU. Furthermore, the static selection of states might not valid over the whole simulation run and dynamic state selection may be required. With the dynamic dummy derivative method (Mattsson and Soderlind, 1993), it is possible to transform to an ODE and export the Modelica model as FMU. During simulation, step events might be used to hold integration and switch to a new set of states that is numerically more appropriate (S.E. Mattsson and Elmqvist, 2000).

- Pantelides index reduction and dummy derivatives algorithms (Pantelides, 1988) , (Mattsson and Soderlind, 1993), (S.E. Mattsson and Elmqvist, 2000), usually reduce the DAE index to zero or one. Hence, another solution would be enriching the FMI standard to support directly index 1 DAEs. Then, index reduction methods can be used to reduce the DAE index to one and exporting it to FMI (Otter and Elmqvist, 2017). One of drawbacks of this method is the lack of backward compatibility, *i.e.*, the FMUs exported in this way can no longer be simulated with FMI-2.0 compatible simulators.

- Another solution is simulating the ODE part of the system using an ODE integrator, but project back the solution onto the constraint manifold after each time step. After completion of each integrator step, the required projection is computed, and if its norm is large enough, it is applied to the solution so that the constraints are satisfied. In this method, monitoring the magnitude of the projection and integrate it into the error control mechanism is required. We chose this solution to implement our FMI export which requires adding a few new APIs for handling constraints and projection. This method will be explained in the rest of this paper.

2.3 Coordinate projection

The key idea to reduce or avoid drift is to project the solution points found by the numerical solver of the index 1 DAE or ODE system back on the manifold defined by the original system. Consider the ODE with constraints (1). The coordinate projection method essentially consists of two steps for each integration step.

1. Suppose that x_{n-1} is a point consistent with the original system (1). Using x_{n-1} as the initial value, the ODE numerical solver takes a step applying some numerical integration method on the equation (1a), and gets the point $\tilde{x}_n$ at t_n.

2. The solution point $\tilde{x}_n$, computed by the ODE solver, is then projected orthogonally back onto the manifold (1b) given by constraints, *i.e.,* the projected solution is computed as the solution of (8)

$$\begin{cases} \|x_n - \tilde{x}_n\|_2 &= \underset{x_n}{\text{minimize}} \\ \phi(x_n) &= 0 \end{cases} \tag{8}$$

   which is a nonlinear constrained least squares problem. The projection gives the orthogonal projection to the manifold to get the next point x_n. The projected value x_n is then used to advance the solution for the next step (Eich-Soellner and Fuhrer, 1998).

In (Shampine, 1986), (Gear, 1986), (Ascher et al., 1994), (Ascher and Petzold, 1992), and (Hairer and Wanner, 1996) the coordinate projection was discussed for one-step methods such as Runge-Kutta methods. In case of BDF-methods or, more generally, multi-step methods, the projection is more complex, since the correction computed by the projection method should enter into the error equation (Eich, 1993).

3 Implementation of ODE with constraint in FMI

Applying the index-reduction algorithm (Pantelides, 1988) to a high index DAE to convert it to an ODE, introduces hidden constraints. In this paper we assume that the index reduction algorithm reduces the index to one or zero (ODE). In case of index 1 DAE, the algebraic variables are treated as local variable in the FMU which are computed as a function of continuous-time states, so they can be ignored. As a result, we will consider only ODE with constraint case, *i.e.*, the equation set (1).

The projection process simply computes the changes required for each state variable so that the current values of the system lie on the constraint manifold. Ideally this should be computed as the minimum (or near minimum) change to accomplish this, as the constraint problem is typically under-determined, so many solutions are possible.

For error-controlled integrators, the change required to move the solution back to the constraint manifold can be integrated into the error control mechanism, so if too large a change is needed, the step can be rejected, and step with a smaller step size can be attempted.

When a high index DAE or an ODE with constraint is exported as FMU, the importer tool needs to know the number of constraints present in the FMU. We have used the attribute `maxNumberOfConstraints` in fmi://ModelDescription.xml element to indicate the

DOI 10.3384/ecp18148213

maximum number of constraints in a model. The default value is zero to keep the FMU backward compatible.

If `maxNumberOfConstraints` is non zero then the FMU should define one or more of the following API functions (depending on the capability flags defined below). .

- `fmi2Status fmi2Constraint(`
  `fmi2Component c, fmi2Real C[])`

  It computes the residual values for all constraints in the FMU. Argument C is `maxNumberOfConstraints` in length. When the solution is on the manifold of the constraint, the norm of the `C` vector is zero or nearly zero.

- `fmi2Status fmi2ConstraintJacobian(`
  `fmi2Component c, fmi2Real J[])`

  It computes the Jacobian for all residual values for all constraints in the FMU with respect to state variables. The length of the array `J` is `maxNumberOfConstraints x numberOfContinuousStates`, and the Jacobian matrix data is storage in row-major. Note that in many cases the constraint Jacobian can become rank-deficient even at non-event points (e.g. bifurcation points in mechanical systems), so caution must be used in using these functions for projection.

  The use of `fmi2Constraint` and `fmi2ConstraintJacobian` provides the master with complete control over the projection process, so one can implement his own scaling method or apply a different solution technique than least squared. In our implementation in Activate, since only one FMU is being used for the Modelica part, the `fmi2ConstraintJacobian` is sufficient, but in case of multiple FMUs with constraint, directional derivative of the constraint should be used.

  Note that `fmi2ConstraintJacobian` is of no use unless `fmi2Constraint` is also defined. If only `fmi2Constraint` is provided, the Jacobian of the constraints can be computed through numerical differentiation. Please note that `fmi2GetDirectionalDerivative` can also be used for our purpose, but some modifications in the variables that can be used in the argument list of this API, *i.e.*, `vKnown_ref` and `vUnknown_ref` would be required.

- `fmi2Status fmi2ProjectionStep(`
  `fmi2Component c, fmi2Real S[])`

  It provides the current local projection step for the constraint residual minimization problem. Argument `S` is `numberOfContinuousStates` in length. For example, in a very simple case, `S`, can be computed as follows. First the Constraint vector `C` and its Jacobian `D` are updated from the model.

```
fmi2Constraint(c,C)
fmi2ConstraintJacobian(c,D)
```

  Then `S` is computed as the pseudo inverse of the matrix `D`, *i.e.,*

$$S = (D^T D)^{-1} D^T C$$

  Note that this is just for illustration purposes and here we have not considered variable scaling, or cautious handling for rank deficiency. This function returns `fmi2Error` if it is unable to compute the step (for example, the above simplified algorithm is used and $D^T D$ is singular), otherwise it returns `fmi2OK`.

  Interface `fmi2ProjectionStep` is also useful when the master has to iterate on a system composed of multiple FMUs with constraint. In such as system it may not be possible to apply the projection once, because a projection might affect an output of an FMU or when it depends on another FMU's inputs. So it may require some iterations. Note that this is not an issue when only one FMU has constraints, and there is no feedback mechanism present for the FMU with constraints.

- `fmi2Status fmi2Projection(`
  `fmi2Component c, fmi2Real P[],`
  `fmi2Real projectionTolerance,`
  `size_t iterationLimit,`
  `fmi2Boolean apply)`

  It provides a full projection of the current solution back onto the constraint manifold. The projection is applied to the current state until the states satisfy the constraints to within `projectionTolerance` or until we exceed `iterationLimit`. The length of array `P` is `numberOfContinuousStates`. The option `apply` specifies if projection should be applied to the FMU state. If `apply=fmi2False` then this function will return only the difference for the states. If `apply=fmi2True` then the projection will be performed and the internal state of the FMU will be updated. The updated states can be retrieved with a subsequent call to `fmi2GetContinuousStates`. This function returns `fmi2Discard` if it is unable to project onto the manifold, otherwise it returns `fmi2OK`. Note that this function returns `fmi2Discard` for the failure case, so the numerical ODE solver can go back and try to take a smaller step until get a successful step and projection. In the `fmi2Discard` case, the current state of the FMU is not altered,

even if called with `apply=fmi2True`. The following pseudo-code demonstrates a simple way for implementation of `fmi2Projection` using `fmi2ProjectionStep`.

```
c_copy = c
delta = infinity
niter = 0
while delta>Tol and niter<=iterationLimit do
 if fmi2ProjectionStep(c,S) != fmi2OK then
   return fmi2error
 end
 delta = |S|
 Update states 'X' in c: X = X+S
 niter++
end while
P = ('X' in c) - ('X' in c_copy)
if not apply then
  c = c_copy
end
if niter>iterationLimit then
  return fmi2Error
end
```

Interface `fmi2Projection` is provided for ease-of-use, but for a system with multiple DAE FMUs it may be necessary to iterate this function at each time step.

- `fmi2Status fmi2GetNumberOfConstraints( fmi2Component c, size_t *N)`

  The number of constraints in a model may change during the simulation when the model configuration changes due to a discrete event. We call this kind of systems variable constraint systems. Since in a variable constraint system, the number of (active) constraints can change, we have used this API function to query the number of constraints that are currently active. So in case of variable constraint systems, the use of this function together with `fmi2Constraint` and `fmi2ConstraintJacobian` is necessary. If this function returns zero, the model does not require coordinate projection. This may happen during the simulation of a variable constraint system.

  This function needs only be defined when `maxNumberOfConstraints` is non zero. If the current number of constraints is less than `maxNumberOfConstraints`, say `Ncon`, then only the first `Ncon` entries of constraints and rows of the Jacobian matrix will be populated.

The following three capability flags are being used for Model Exchange FMUs to indicate to the importing tool which of the API functions are supported within the FMU:

- `providesProjection` (Boolean): If `true` the FMU computes projection via fmi2Projection interface. The default value is `false`.
- `providesProjectionStep` (Boolean): If `true` the FMU provides projection step vector via `fmi2ProjectionStep` interface. The default value is `false`.
- `providesConstraints` (Enumeration with `true`, `false`, and `withJacobian`): If `true` the FMU can compute the constraint residual via fmi2Constraint interface. If set to `withJacobian` then additionally the FMU can compute the constraint Jacobian via the fmi2ConstraintJacobian interface.

3.1 Multiple FMU model

The extension to FMI should support the case where several FMUs containing constraints are interconnected such as in a *System Structure Parameterization (SSP)*[4] module. In multiple FMU case, the internal constraints of FMU may depend on FMU inputs which are outputs of other FMUs with constraints. For multiple FMU case, `fmi2Constraint` and `fmi2ProjectionStep` are quite useful. In order to integrate such systems with multiple FMUs, after a complete input/output update of FMUs at a given time and given continuous-time state value, `fmi2Constraint` or `fmi2ProjectionStep` functions can be called by the solver to check and compute the required projection.

3.2 Backward compatibility

This extension in the FMI APIs does not introduce any backward compatibility issue in the FMI standard. Any importing tools will simply have to make sure they can ignore the new capability flags. The exporting tools do not have to generate the new features as the capability flags are `false` by default. So if the importing tool cannot take advantage of the new APIs such as constraints and projections, they are ignored and only ODE is integrated with the price of possible drifts in states.

3.3 Required numerical solvers

In order to take advantages of this extension, the importing tool should include solvers that can support overdetermined systems, in particular, ODEs coupled with algebraic constraints. An example of such solver is CPODES[5]. CPODES is a numerical integrator for solving ODE problems using coordinate projection. It is based on the CVODES integrator which is part of the DOE Sundials[6] suite. CPODES is a multi-step integrator providing variable order Adams (up to 12th order)

[4] https://modelica.github.io/ssp-standard.org/

[5] https://simtk.org/projects/cpodes

[6] https://computation.llnl.gov/projects/sundials

DOI 10.3384/ecp18148213

and BDF (up to 5th order) methods for non-stiff problems and BDF (up to 5th order) for stiff problems. It uses CVODES to advance the ODE (2a), and then performs coordinate projection back to the constraint manifold (2b) to exactly solve the DAE (2). The projection is also incorporated back into the error test where it permits larger steps.

Other single-step solvers can also be modified to support the coordinate projection method. In Activate we have modified the RADAU-IIA solver[7] to apply projection computed by `fmi2Projection`.

4 Test cases

4.1 Pendulum model

The pendulum model explained in section 2.2 can be defined with the following Modelica model.

```
model Pendulum_DAE
   constant Real g=9.81;
   constant Real L=1;
   constant Real Ls=L*L;
   Real x(start=L, fixed=true), y(start=0), Lambda ;
   Real vx(start=0, fixed=true), vy(start=0);
   Real drift, totalEnergy;
equation
   der(x)=vx;
   der(y)=vy;
   der(vx)=Lambda*x;
   der(vy)=Lambda*y-g;
   x*x+y*y=Ls;
   drift=x*x+y*y-Ls;
   totalEnergy=(vx*vx+vy*vy)/2+g*y;
end Pendulum_DAE;
```

In this Modelica model, `drift` and `totalEnergy` are output variables defining the drift in the pendulum length and the total energy of the system, respectively. A standard FMU can be generated for this Modelica model. If the two constraints (7) are needed to be satisfied, the FMU can be augmented with the following new APIs.

```
fmi2Status fmi2GetDerivatives(fmi2Component c,
    fmi2Real derivatives[],  size_t nx) {
    double F, g=9.81;
    ModelInstance* comp = (ModelInstance *)c;

    x =comp->state[0];
    y =comp->state[1];
    xd=comp->state[2];
    yd=comp->state[3];

    F = y*g-(xd^2+yd^2);
    derivatives[0]=xd;
    derivatives[1]=yd;
    derivatives[2]=F*x;
    derivatives[3]=F*y-g;
    return fmi2OK;
}

fmi2Status fmi2Constraint( fmi2Component c,
    fmi2Real C[]) {
    ModelInstance* comp = (ModelInstance *)c;
    x =comp->state[0];
    y =comp->state[1];
    xd=comp->state[2];
    yd=comp->state[3];

    C[0]=x*x + y*y- 1;
    C[1]=x*xd+ y*yd;

    return fmi2OK;
}

fmi2Status fmi2Projection(fmi2Component
c, fmi2Real P[], fmi2Real projectionTolerance,
   size_t iterationLimit, fmi2Boolean apply) {
      ModelInstance* comp = (ModelInstance *)c;
      double R;
      x =comp->state[0];
      y =comp->state[1];
      xd=comp->state[2];
      yd=comp->state[3];

      R = sqrt(x*x+y*y);
      P[0] = x/R - x;
      P[1] = y/R - y;
      P[2] = ( xd*y*y- yd*x*y)/R/R - xd;
      P[3] = (-xd*x*y+ yd*x*x)/R/R - yd;

      If (apply){
         comp->state[0] = x/R;
         comp->state[1] = y/R;
         comp->state[2] = ( xd*y*y- yd*x*y)/R/R;
         comp->state[3] = (-xd*x*y+ yd*x*x)/R/R;
      }
      return fmi2OK;
}
```

In the first test, the pendulum model is exported as a standard FMU, i.e., exported as a pure ODE without constraints. The RadauII-A solver with error tolerance=1e-4 is used. In Figure 4, the left plot displays the `x` and `y` variables and the right plot is the `drift` variable. `drift` is growing as time advances.

Then the same model is exported as an FMU with additional constraints and projection APIs. As shown in Figure 5, the drift in the solution is kept below the requested error tolerance. Whenever the drift exceeds the error tolerance, the projection is applied and the drift becomes zero.

In these tests, only constraints (7) have been considered. We need also to consider the energy conservation law, *i.e.,* the amount of total energy of the system should not change. The energy constraint has not been considered in the Modelica model. So a drift in the total energy due to numerical errors may happen. In order to check the total energy the simulation time is increased to T=300 seconds. A drift in the energy with the value of 9e-3 is obtained. If we add the additional energy constraint to the FMU, *i.e.,* adding the following constraint to the FMU, we can keep the energy constraint valid.

```
C[2]= (xd*xd + yd*yd)/2+g*y
```

The result is given in Figure 6 where the left-hand plot is the total energy of the pendulum with there is no energy constraint in the model and and the right side plot is the amount of drift in the energy when the above ad-

[7] https://www.unige.ch/~hairer/software.html

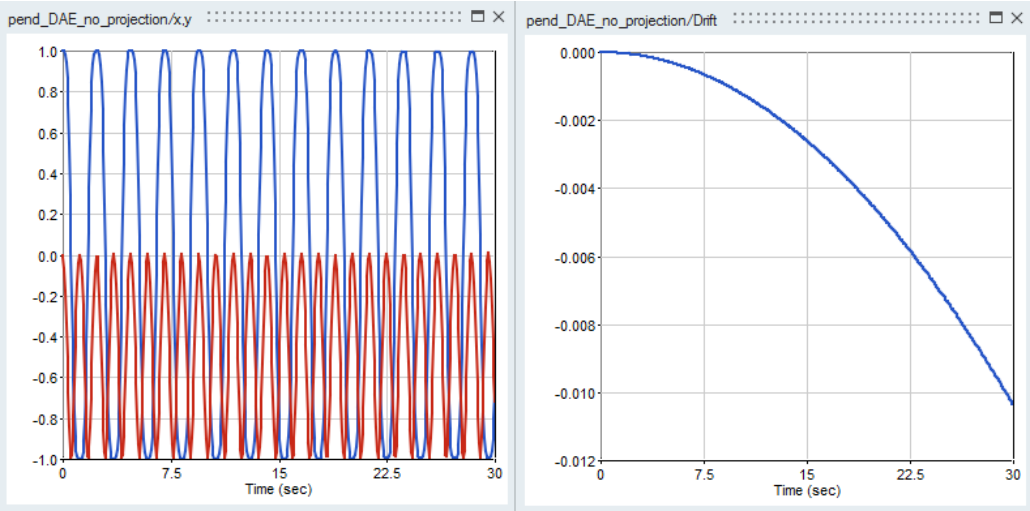

Figure 4. Simulation result of the Pendulum without constraints.

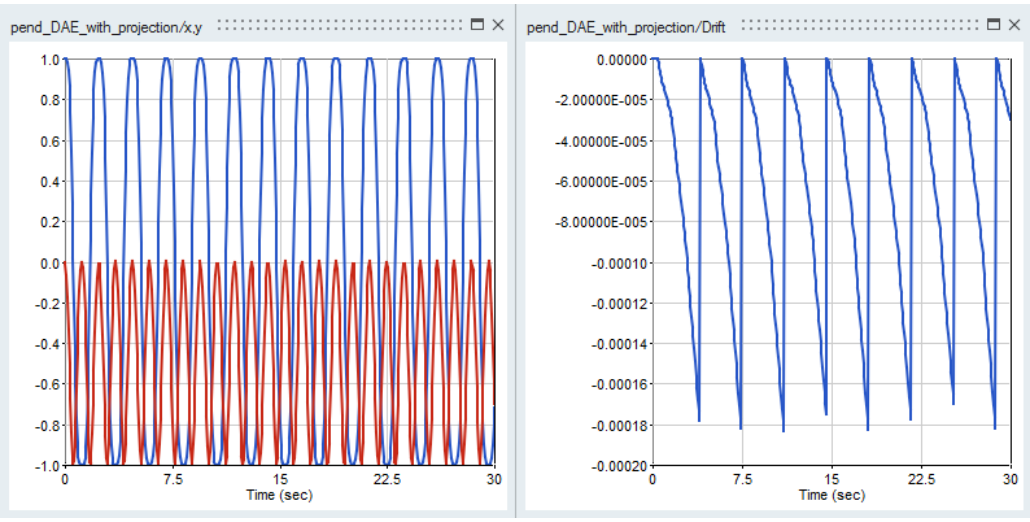

Figure 5. Simulation result of the Pendulum with constraints.

ditional constraint is applied. For this experiment, the modified RADAU-IIA solver with ATOL=RTOL= 1e-5 and maximum step-size=0.1 has been used.

Although the drift in the constraints (middle figure) is kept below the requested tolerance, the amplitude of the `y` variable as well as the total energy of the system is decreased. This is due to lack of the energy constraint in the original model.

4.2 Li-Ion Battery model

The second model (taken from the MapleSoft's Battery Library) represents an electric vehicle, powered by a battery stack consisting of 99 Li-ion cells wired in series. The model features battery temperature changes while the vehicle is controlled to follow an EPA highway drive cycle, defined in a lookup table. *LiFePO4* is used as the cathode material to provide good thermal stability. The model, as shown in Figure 7 is developed in Activate using Modelica components. This example was one of the motivation to handle the constraints efficiently.

This model contains a constraint that should be monitored to keep it near zero during the simulation. In order to ensure that the constraint stays near zero, in the standard FMI, on every `fmi2CompletedIntegrationStep` call, the constraint is checked, if exceeds the error tolerance, the numerical solver is restarted which, as explained in section 1, slows down the simulation. After the development of the new extension of FMI in in the MapleSim Modelica compiler and in Activate, we were able to simulate this model in a few seconds compared to hours. The simulation result is given in Figure 8.

5 Conclusion

In order to simulate Modelica components in Activate, they are regrouped and exported into an FMU. Since, in the current FMI standard only ODE is supported, the

DOI 10.3384/ecp18148213

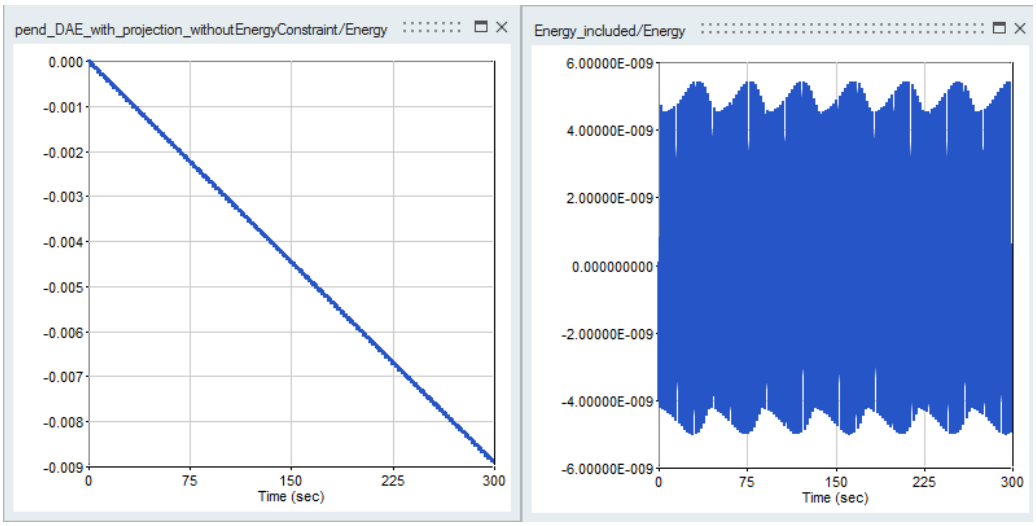

Figure 6. Drift in the total energy of the pendulum: with considering the energy constraint (right) and without considering the energy constraint (left).

Modelica models, even high index ones, are converted into ODE to be exportable into FMU. Due to conversion of high index DAEs to ODE, some constraints in the DAE may be ignored and that may cause the solution drift off the constraints. In this paper we have presented the way the FMI standard can be extended in a backward compatible way to to deal with systems whose states should satisfy hidden constraints on its continuous-time states, such as constraints resulting from DAE index reduction of mass or energy conservation. In this extension, after applying the index reduction method to high index DAE, an ODE with constraints is obtained. The resulting ODE is simulated using standard ODE integrators, but the solution is projected back onto the constraints after each time step. In other words, after completion of each integrator step of the ODE numerical solver, the required projection to bring back the solution on the constraints is computed, and if its norm is large enough, it is applied to the solution so that the constraints are satisfied. The new extension of FMI has been implemented in Activate and two examples are illustrated in this paper.

6 Acknowledgement

This work has been done with close collaboration with MapleSim™ development team who provides the Modelica compiler of Activate.

References

U. Ascher and L. Petzold. *Computer methods for ordinary differential equations and differential-algebraic equations*. SIAM, 1988.

U. M. Ascher and L. R. Petzold. Projected implicit runge-kutta methods for differential-algebraic equations. *SIAM, Numerical Analysis, 28(4), 1097-1120.*, 1992.

U. M. Ascher, H. Chin, and S. Reich. Stabilization of daes and invariant manifolds. *Numer. Math. 67: 131*, 1994.

J. Baumgarte. Stabilization of constraints and integrals of motion in dynamical systems. *Computer Methods in Applied Mechanics and Engineering Volume 1, Issue 1, Pages 1-16*, 1972.

E. Eich. Convergence results for a coordinate projection method applied to mechanical systems with algebraic constraints. *SIAM J. on Numerical Analysis 30(5):1467-1482*, 1993.

E. Eich-Soellner and C. Fuhrer. *Numerical Methods in Multibody Dynamics*. European Consortium for Mathematics in Industry, B.G. Teubner, 1998.

C.W. Gear. Maintaining solution invariants in the numerical solution of odes. *Journal on Scientific and Statistical Computing, Vol. 7, No. 3*, 1986.

E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Springer, 1996.

S.E. Mattsson and G. Soderlind. Index reduction in differential-algebraic equations using dummy derivatives. *SIAM Journal of Scientific Computing. 14(3), pp. 677-692*, 1993.

R. Nikoukhah. A simulation environment for efficiently mixing signal blocks and modelica components. *12'th International Modelica conference*, 2017.

M. Otter and H. Elmqvist. Transformation of differential algebraic array equations to index one form. *Proceedings of the 12th International Modelica Conference, Prag, Czech Republic*, 2017.

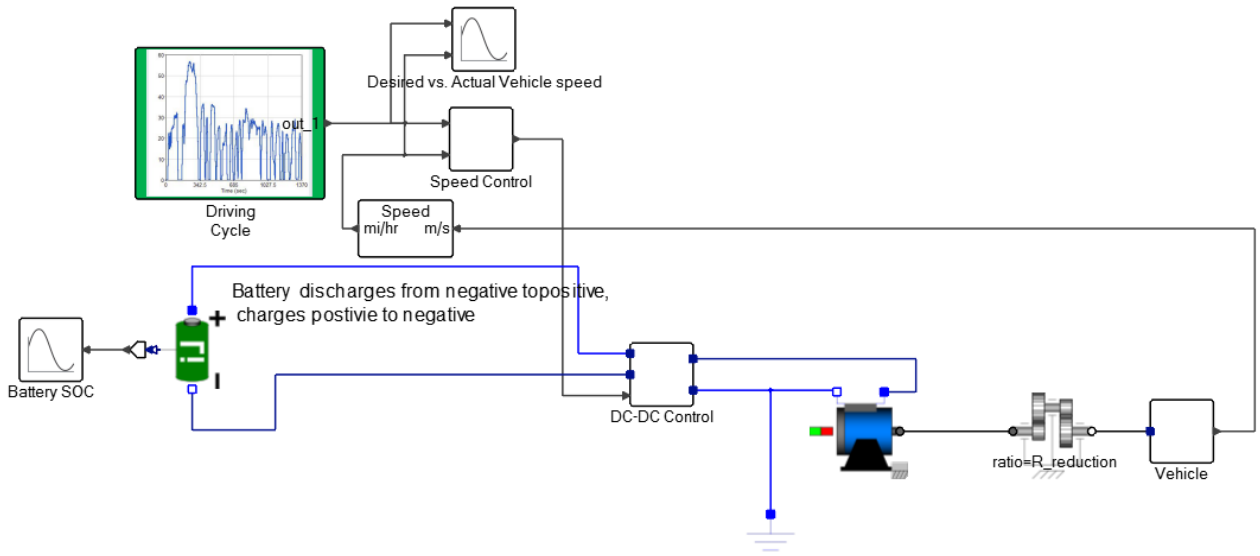

Figure 7. Battery Electric Vehicle model

C. C. Pantelides. The consistent initialization of differential-algebraic systems. *SIAM Journal on Scientific and Statistical Computing*, 9(2):213–231, 1988. doi:10.1137/0909014.

H. Olsson S.E. Mattsson and H. Elmqvist. Dynamic selection of states in dymola. *Modelica Workshop 2000, Lund, Sweden, pp. 61-67*, 2000.

L. Shampine. Conservation laws and the numerical solution of odes,. *Comput. Math. Appls, Part B., 12, pp. 1287-1296*, 1986.

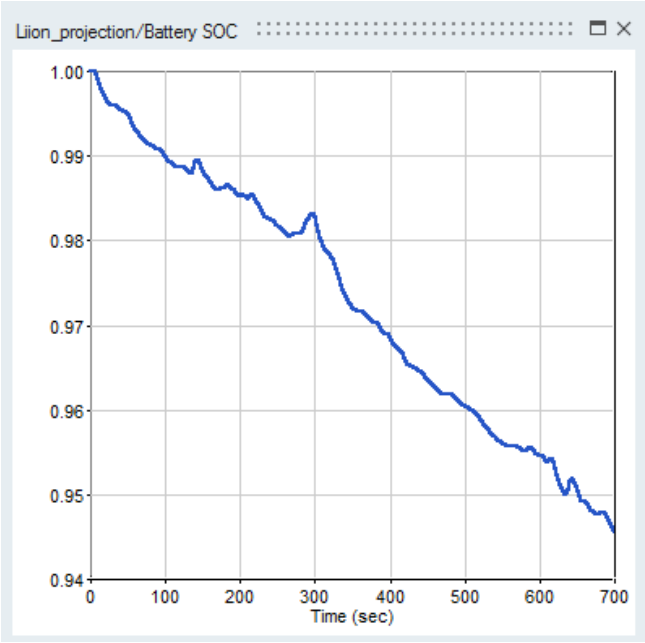

Figure 8. State of charge of the battery

DOI
10.3384/ecp18148213