Model Embedded Control: A Method to Rapidly Synthesize Controllers in a Modeling Environment

E. D. Tate Michael Sasena† Jesse Gohl† Michael Tiller†

Hybrid Powertrain Engineering, General Motors Corp.
1870 Troy Tech Park, Troy, Michigan, 48009

†Emmeskay, Inc, 47119 Five Mile Road
Plymouth, Michigan, 48170

ed.d.tate@gm.com msasena@emmeskay.com jbgohl@emmeskay.com mtiller@emmeskay.com

Abstract

One of the challenges in modeling complex systems is the creation of quality controllers. In some projects, the effort to develop even a reasonable prototype controller dwarfs the effort required to develop a physical model. For a limited class of problems, it is possible and tractable to directly synthesize a controller from a mathematical statement of control objectives and a model of the plant. To do this, a system model is decomposed into a controls model and a plant model. The controls model is further decomposed into an optimization problem and a 'zero-time' plant model. The zero-time plant model in the controller is a copy or a reasonable representation of the real plant model. It is used to evaluate the future impact of possible control actions. This type of controller is referred to as a Model Embedded Controller (MEC) and can be used to realize controllers designed using Dynamic Programming (DP).

To illustrate this approach, an approximation to the problem of starting an engine is considered. In this problem, an electric machine with a flywheel is connected to crank and slider with a spring attached to the slider. The machine torque is constrained to a value which is insufficient to statically overcome the force of the spring. This constraint prevents the motor from achieving the desired speed from some initial conditions if it only supplies maximal torque in the desired direction of rotation. By using DP, a control strategy that achieves the desired speed from any initial condition is generated. This controller is realized in the model using MEC.

The controller for this example is created by forming an optimization problem and calling an embedded copy of the plant model. Furthermore, this controller is calibrated by conducting a large scale Design of Experiments (DOE). The experiments are processed to generate the calibrations for the controller such that it achieves its design objectives when used for closed loop control of the plant model.

It is well understood that Modelica includes many language features that allow plant models to be developed quickly. As discussed previously, the development of quality control strategies generally remains a bottleneck. In this paper we show how existing features along with appropriate tool support and potential language changes can make a significant impact on the model development process by supporting an automated control synthesis process.

Keywords: Control, Dynamic Programming, Model Embedded Control, Model Based Control, Optimal Control

1 Introduction

The use of modeling is well established in the development of complex products. Modern tools have significantly reduced the effort required to model and tune physical systems. Acausal or topological modeling reduces the effort required to model a system's physics. The use of optimization allows systematic tuning of parameters to improve a design. The combination of parameter optimization and rapid modeling allows a large set of potential designs to be quickly evaluated. However for systems which include controls, the development is, in general, a man-power intensive process subject to large uncertainty in development time and optimality. The optimization of both controls and design must be solved in many problems [1-3]. One way to address this problem is to use numerical techniques to construct controllers. For certain classes of problems, tractable numerical techniques can be used to develop an approximately minimizing controller [4]. A minimizing controller is a controller which achieves the best possible performance from a system as measured against an objective. There may exist more than one controller able to achieve this minimum, but no controller can perform better than a minimizing controller. For this work, the terms minimizing controller and optimal controller are used interchangeably.

To construct a minimizing controller, an object cost, J, is defined. This is a function which maps the state and input trajectory of the system to a scalar:

$$J = C(x,u). \tag{1}$$

Consider the special case of a plant described by ordinary differential equations with inputs that are piecewise constant. These piecewise constant inputs are updated periodically at the 'decision instances' by a controller at intervals of Δt. The total operating cost is calculated as a sum over an infinite time horizon. Furthermore, the sum of costs is discounted by the term α which is greater than zero and less than or equal to one. The total cost is calculated by an additive function that operates on the instantaneous state and the control inputs. This cost may take a form similar to

$$J\left(x(0)\right) = \sum_{k=0}^{\infty} \alpha^k \cdot \left(\int_{t=t_k}^{t_{k+1}} c_{cont}\left(x(\tau),u_k\right)\cdot d\tau \right). \tag{2}$$

The total cost in (2) is a function of the initial state of the system. To simplify notation, let the state at the decision instances be represented by

$$x_k = x\left(t_k\right). \tag{3}$$

Let the discrete time samples occur at

$$t_k = k \cdot \Delta t. \tag{4}$$

Furthermore, let the continuous-time instantaneous cost, c_{cont}, in (2) be represented in discrete time notation as an additive cost over an interval,

$$c\left(x_k,u_k\right) = \int_{t=t_k}^{t_{k+1}} c_{cont}\left(x(\tau),u_k\right)\cdot d\tau. \tag{5}$$

Using the notation developed in (2) through (5), the continuous-time system's total cost is expressed in discrete time notation as

$$J\left(x_0\right) = \sum_{k=0}^{\infty} \alpha^k \cdot c\left(x_k,u_k\right). \tag{6}$$

To simplify the continuous-time dynamics, let

$$f_d\left(x,u\right) = \left\{ \int_0^{\Delta t} f\left(\zeta(\tau),u\right)\cdot d\tau, \zeta(0) = x \right\}. \tag{7}$$

Hence,

$$x_{k+1} = f_d\left(x_k,u_k\right). \tag{8}$$

An optimal control choice for each time step can be found using the dynamic programming equations,

$$u^*\left(x\right) \in \underset{u\in U(x)}{\arg\min} \left\{ c\left(x,u\right) + \alpha \cdot V\left(f_d\left(x,u\right)\right) \right\}. \tag{9}$$

The function $V\left(x\right)$ is known as the value function. By using the dynamic programming (DP) equations to find the value function, a minimizing controller is obtained. The DP equations are

$$V\left(x\right) = \min_{u\in U(x)} \left\{ c\left(x,u\right) + \alpha \cdot V\left(f_d\left(x,u\right)\right) \right\}, \tag{10}$$

where

$$U\left(x\right) = \left\{ u \middle| g\left(x,u\right) \le 0 \right\} \tag{11}$$

defines the set of feasible actions, $U(x)$. For the case where the total cost is considered over an infinite horizon and $u_k = u^*\left(x_k\right)$ (see eq (9)), the value function is the same as the total cost function, $J(x) = V(x)$. Equation (10) can be solved through value iteration, policy iteration, or linear programming. See [5-23] for discussion of solution methods. For discussion of using DP to find value functions for automotive control application, see [24-29]. The formulation of equation (11) is chosen to simplify management of constraints throughout the model and to conform to a standard form used in the optimization community, the negative null form [30].

One problem with solving (10) is that when the state space consists of continuous states, $V\left(x\right)$ is a function from one infinite set to another. Except in special cases, this requires approximation to solve. One common approach is to use linear bases to approximate the value function. Possible linear bases include the bases for multi-linear interpolation, the bases for barycentric interpolation, b-splines, and polynomials. See the appendices in [25] for a discussion of linear bases for dynamic programming. In the case where $V\left(x\right)$ is approximated by a linear basis,

$$V(x) = \Phi^T(x) \cdot w, \tag{12}$$

where

$$\Phi(x) = \begin{bmatrix} f_1(x) & f_2(x) & \cdots & f_N(x) \end{bmatrix}. \tag{13}$$

An approximate solution to (10) is found by finding the weights, w, which solve

$$\Phi^T(x) \cdot w = \min_{u \in U(x)} \left\{ c(x,u) + \alpha \cdot \Phi^T\left(f_d(x,u)\right) \cdot w \right\}. \tag{14}$$

See [5-7] for a discussion of using linear bases to form the value function.

It is important to understand that this controller is an optimal controller for the discrete time case only, when the controller updates every Δt seconds. In other uses, the controller will generally be suboptimal. Additionally, any development algorithm based on this methodology will suffer from the curse of dimensionality [31]. In other words, the time to find an optimal controller will increase geometrically with the size of the plant state space. As a point of reference, using a single commercially available PC from 2005, a five state controller was found in less than twenty four hours.

2 Controller Development

To use equations (2) through (14) to develop a controller, it is necessary to have a plant model which includes the dynamics (f), cost function (c), and constraints (g) all coupled to an integrator which can be invoked as a function call by a Control Design Algorithm (CDA). In addition, the set of states for the plant model and the set of controller actions must be specified to the CDA. For this work, a custom wrapper was developed that allowed batches of states and actions to be efficiently evaluated. Each evaluation returned the state at the next interval, the cost of operation for the interval, and the constraint activity over the interval.

To understand the structure of the equations involved in this work, consider a system consisting of a plant and a controller. Without loss of generality, assume the plant dynamics are described by ordinary differential equations

$$\dot{x} = f(x,u), \tag{15}$$

where f is a function that describes the plant dynamics. For notational simplicity consider a continuous time controller. Let the controller be a full state feedback controller implemented as a static mapping, M, from the state, x, to the action set, u:

$$u = M(x). \tag{16}$$

Assuming only a single global minimum exists, the dynamic programming equations in (9) can be directly used for the static mapping (16). The autonomous dynamics of this system are then described by the following equation

$$\dot{x} = f\left(x, \arg\min_{u \in U(x)} \left\{ c(x,u) + \alpha \cdot V\left(f_d(x,u)\right) \right\} \right) \tag{17}$$

This equation is then integrated to solve for $x(t)$,

$$x(t) = \int_0^t f\left(x(s), \arg\min_{u \in U(x)} \left\{ \begin{array}{l} c\left(x(s),u\right) \\ +\alpha \cdot V\left(f_d\left(x(s),u\right)\right) \end{array} \right\} \right) \cdot ds \tag{18}$$

where

$$x(0) = x_0 \tag{19}$$

defines the initial conditions. To evaluate f_d from (7), a nested integrator, which is independent of the primary simulation integrator, is required. This nested integrator executes in 'zero-time' from the perspective of the primary integrator. We refer to this as an embedded or nested simulation. Because the nested integrator is used inside a numeric optimization, it will potentially be called multiple times at each primary integrator evaluation. If f_d in (18) is expanded using (7), the plant dynamics function, f, from (15) occurs in two locations in

$$x(t) = \int_0^t f\left(x(s), \arg\min_{u \in U(x)} \left\{ \begin{array}{l} c\left(x(s),u\right) + \\ \alpha \cdot V\left(\int_0^{\Delta t} f\left(\zeta(\tau),u\right) \cdot d\tau, \zeta(0) = x(s) \right) \end{array} \right\} \right) \cdot ds \tag{20}$$

where

$$x(0) = x_0 \tag{21}$$

The nested copy of the plant dynamics equations, f, is referred to as the embedded or nested model. In the case where the controller is modeled as updating periodically, rather than continuously, the solution to the optimization problem is held constant between controller updates.

The equation structure in (20) and the reuse of the plant dynamics function, f, offer the ability to quickly synthesize controllers using numerical techniques. However, existing tools make the implementation of this type of model problematic. There are two primary issues in implementation. The first is execution efficiency. Few commercial tools have been developed with the goal of efficiently solving this class of equations. Secondly, several commercial modeling environments make the definition and reuse of the plant model cumbersome, requiring significant efforts during development and maintenance. Fortunately, the features of Modelica make the definition and reuse of a plant model manageable. The examples that follow have been developed in Dymola ®, however this general approach has also been used with Simulink® and AMESim®.

To systematically generate a system with an optimal controller, a model of the plant is generated. This plant model is 'wrapped' with an application programming interface (API) so a control design algorithm can determine the state space, the action space, the state at the next time step, the constraint activity, and the cost for a given state and action. This interaction between the plant model, the API and the control design algorithm is illustrated in Figure 1. The CDA queries the API to determine the structure of the state and action space. Given this structure and the configuration of the CDA, a sequence of DOEs is executed. The DOE data are used to find a solution to (10). For this work, the value function was modeled using multi-linear interpolation and a solution to (14) was found. To simplify coding, value iteration was used [5, 6] to find $V(x)$.

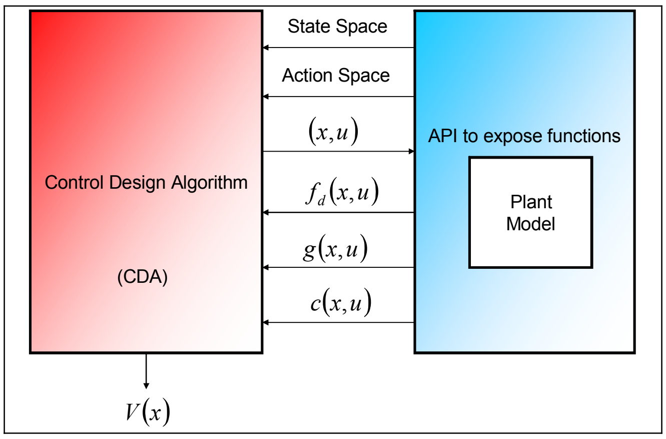

Figure 1 - Plant Model API

Once the value function is generated, the system model is formed by one of two methods. The first method is by generating a lookup table that maps the state variables to an action as in (16). The process of generating a value function, finding a mapping equivalent to (9), and realizing a controller as a mapping (or lookup table) is referred to as Indirect Model Embedded Control (IMEC). This method is appropriate for some systems. Another approach, which is more computationally expensive, is referred to as Direct Model Embedded Control (DMEC). For DMEC, the controller is realized by forming an optimization statement around an embedded copy of the plant model. This structure is illustrated in Figure 2.

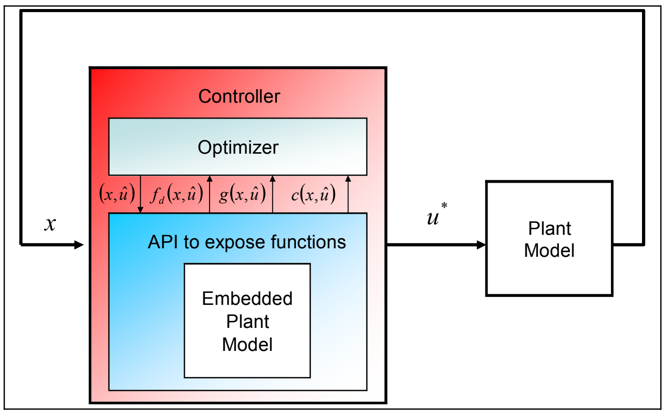

Figure 2 - Direct Model Embedded Controller Structure

To realize a Direct Model Embedded Controller (DMEC), two pieces are added to the system model. The first piece is an optimizer which solves (9). This optimizer can be as simple as a Design of Experiments (DOEs) which considers a fixed set of actions, and selects one which minimizes (9). For more sophistication, if the nature of the problem permits it, a gradient-based optimizer can be employed [30, 32, 33]. If the nature of the problem does not allow solution using these types of approaches, global solvers can be used [34-36]. Ideally, an optimization library should support both gradient and non-gradient methods for constrained optimization problems. As part of this project, libraries for performing both DOEs and gradient-based optimizations were implemented entirely in Modelica. However, there are currently no comparable commercial or public domain libraries available. The second piece required to implement a DMEC is the ability to invoke a function which efficiently initializes and simulates, over a 'short' time horizon, a set of models which are copies of the plant model with modified parameters. Because of the structure of the problem, each time the controller executes, multiple embedded simulations will execute. Depending on the nature of the action set, the number of embedded simulations may vary from as few as two embedded simulations to several thousand embedded simulations.

3 Example – Simple Engine Start

To illustrate how these concepts are used to build a controller, consider the problem of starting an internal combustion engine using an electric machine with insufficient torque to guarantee the engine completes a revolution from all possible stationary starting points. If the initial position of the engine is in a range of angles, the electric machine will stall. To simplify the modeling, let us assume the engine can be approximated using a crank slider connected to a spring. The system model, shown in Figure 3, consists of an electrical motor connected to the crank which connects through the crank slider mechanism to a piston which is subject to damping from friction. Inertia is present in the motor rotor, crankshaft and piston. The electric machine is subject to constraints on minimum and maximum torque.

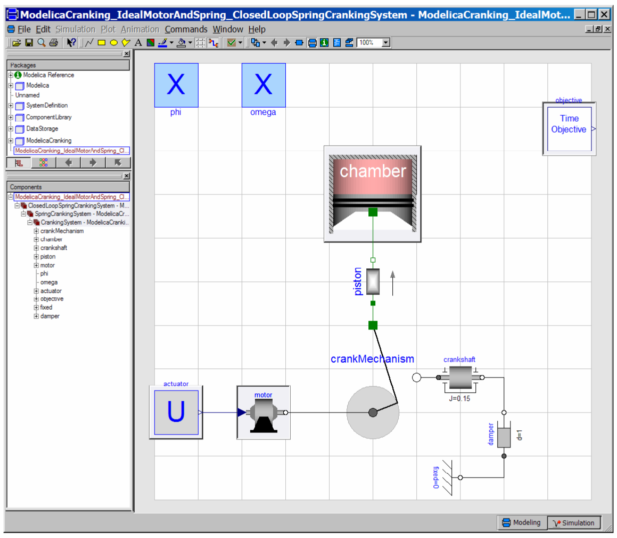
ModelicaCranking_IdealMotorAndSpring_ClosedLoopSpringCrankingSystem - ModelicaCranking_IdealMot...
File Edit Simulation Plot Animation Commands Window Help
Packages
Components
X
phi
X
omega
Time Objective
chamber
piston
crankMechanism
crankshaft
actuator
U
motor
damper
fixed
Modeling
Simulation

Figure 3 - Engine Starting Model

The objective of the control system is to ensure the engine will overcome the initial compression torque from any initial state and minimize engine start time. The total cost of operation (what is being minimized) is expressed mathematically as the total time taken to achieve a speed greater than or equal to five hundred RPM. Once this speed is achieved, the controller is deactivated and another scheme is used to manage the engine. The total cost of operation for this system is considered over an infinite time horizon and is computed as

$$J\left(x(0)\right)=\int_0^\infty \begin{Bmatrix} 0 & ,\omega(t)\geq 500 \text{ rpm} \\ 1 & ,\text{otherwise} \end{Bmatrix}\cdot dt\,. \qquad (22)$$

The instantaneous cost for this system is

$$c(x)=\begin{Bmatrix} 0 & ,\omega > 500 \text{ rpm} \\ 1 & ,\text{otherwise} \end{Bmatrix}. \qquad (23)$$

This type of cost generates a 'shortest-path' controller. The controller will minimize the total time to achieve 500 rpm. The total cost in (22) is undiscounted. Therefore the discounting factor, α, which is visible in (2) is assigned a value of one and omitted from the expression.

While it is clear that the system has exactly two states, they can be selected somewhat arbitrarily. For this example, the engine angle and engine speed were selected. With these variables as the states, the controller is represented as a static map from the engine angle and engine speed to the electric machine torque.

$$u=M\left(\omega,\theta\right) \qquad (24)$$

The feasible action set is a single real number, the motor torque, bounded by the constraints on motor torque and power. The set of feasible actions is defined by

$$U(x)=\left\{u\,\middle|\,\begin{matrix}-100\leq u\leq 100, \\ -10000\leq u\cdot\omega\leq 10000\end{matrix}\right\}. \qquad (25)$$

The value function was represented using multi-linear interpolation, see equation (12).

The plant model was implemented in Modelica. The Controller Design Algorithm (CDA) was implemented in MATLAB®. The CDA invoked function calls to a custom API, similar to Figure 1, applied to the plant model in Dymola®. The CDA solved for the weights, w, in the value function (equation (12)). This value function was used to generate an Indirect Model Embedded Controller (IMEC) and a Direct Model Embedded Controller (DMEC). The value function generated by the CDA is shown in Figure 4.

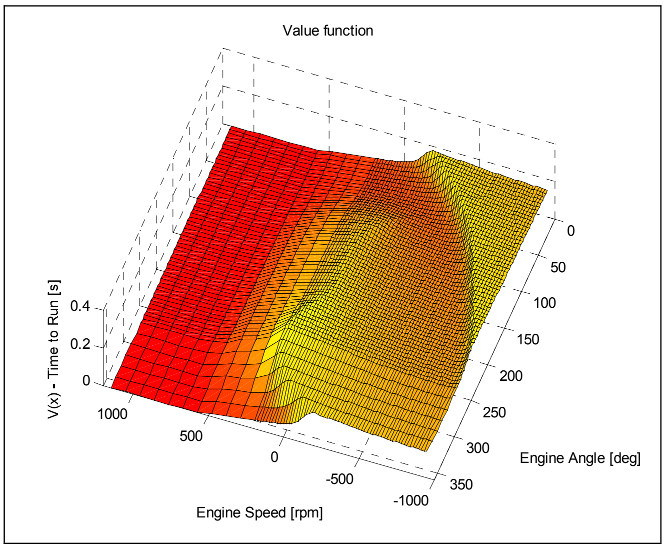

Figure 4 - Value function

The IMEC was realized as a two input lookup table with multi-linear interpolation on a regular grid. The grid points in the table were found by solving (9) using the value function generated by the CDA. This controller was implemented using standard Modelica components. The actuator commands for the IMEC controller are shown in Figure 5 as a function of engine speed and angle.

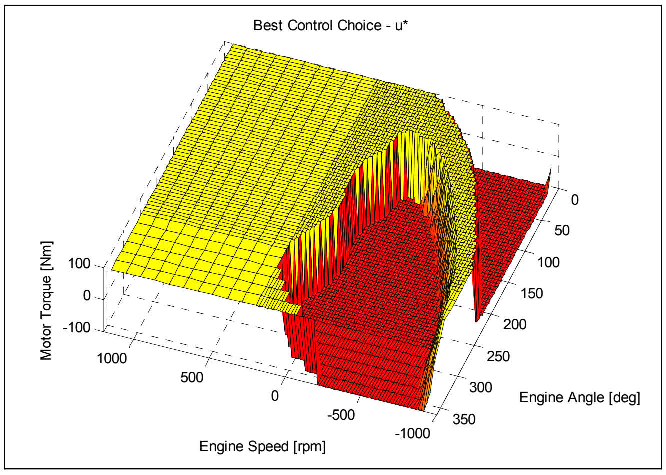

Figure 5 - IMEC control table

The DMEC was realized by wrapping a copy of the plant model with an API similar to the one used for the CDA. A DOE was used to search feasible actions. The resulting code structure is identical to Figure 2. The optimal action was chosen to minimize (9).

For both of these controllers, the problem of starting the engine from any initial condition was solved. The solution involved the counter-intuitive approach of spinning the engine backwards, then reversing direction to allow enough energy to be stored in the inertia to overcome the spring force. From a model and a control objective, an optimal controller with very complex behaviors was numerically generated in less than 10 minutes on a single PC (3GHz, 2Gb RAM). Furthermore, a similar problem with four states was solved in less than three hours. Of course the power of this approach can only be realized once a sufficient level of tool support is available so that the time required to set up the analysis is on the same order as the solution time.

3.1 Direct vs Indirect MEC

Ideally, both an IMEC and DMEC will result in identical behaviors. However, differences in approximation schemes and interpolation can results in appreciable differences. In many cases, while Indirect MEC is simpler to realize in a model, there are good reasons to implement a controller with the complexity and computational cost of a Direct MEC.

As an example, consider the previous problem. The value function, V(x), was found using the Control Design Algorithm (CDA). The IMEC controller was designed by solving for the best electric machine torque for a set of engine angles and speeds on a regular grid. For engine states which occur off this grid, multi-linear interpolation was used to calculate the control action. When the IMEC was used in an engine start simulation, if the optimal torque transitioned between positive and negative, the interpolation caused a smooth change in the torque because of the continuity imposed by interpolation.

Alternatively, consider a Direct MEC. Because of the characteristics of the dynamic programming equations and the value function, the optimal choices are either full positive or full negative torque. This results in an instantaneous, non-continuous change in torque. When plotted as in Figure 6, the difference between the control inputs and the state evolution of the system can be seen. The interpolation due to the approximation in the IMEC results in artifacts in the control actions and a slight loss of performance in the system. Mathematically this means that more detail is required to resolve $u^*(x)$, the function that we are ultimately trying to formulate, than to resolve $V(x)$.

There are cases where an IMEC is superior to a DMEC approach (*e.g.* [26] illustrates just such a case). In general, an IMEC implementation is superior when both the action set is continuous and the optimal actions are continuous. The DMEC approach is superior when either the action set is discrete or the optimal actions are not continuous with respect to the state. One example where DMEC is clearly superior is where the motor is controlled by selecting the state of a switch inverter. In this case, the action set consists of a finite set of choices for switch configuration and the optimal actions are not continuous with respect to the state.

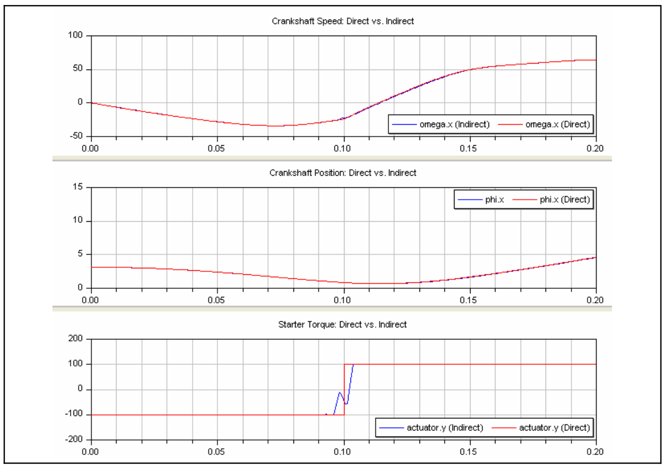

Figure 6 - Comparison of IMEC and DMEC results

4 Implementation of optimization algorithms

One of the challenges in Direct Model Embedded Control is the implementation of an optimizer. While this work was performed using a Design of Experiments (DOE) to select optimal actions, this approach becomes intractable when equality constraints and larger dimensional actions sets are considered. Towards the goal of supporting these classes of problems, a gradient-based optimizer was developed. One of the goals in developing this optimizer was to fully implement the optimizer in Modelica. By fully implementing in Modelica, all of the information used by the optimizer would be accessible for speed improvements by the compiler. Should native support for model embedding become available, all equations associated with a Direct MEC would be accessible to the compiler for speed improvement. Additionally, since the embedded simulations in a DMEC can be completely decoupled from each other, simulation tools could easily exploit the coarse grained parallelism on multi-core CPUs by running several embedded simulations concurrently when conducting searches in the optimizer (e.g. line searches and numerical gradients).

The optimizer was developed in Modelica to solve a constrained optimization problem which is generally stated in negative null form [30] as

$$\begin{Bmatrix} \min\limits_{\chi} f_{objective}(\chi) \\ s.t. \\ g_{inequalities}(\chi) \le 0 \\ h_{equalities}(\chi) = 0 \end{Bmatrix}. \qquad (26)$$

To implement a gradient optimizer, the optimizer functionality was separated from the objective function ($f_{objective}$), the inequality constraint functions ($g_{inequalities}$), and the equality constraint functions ($h_{equalities}$). The optimizer was designed under the assumption that the inequality constraint functions are all in negative null form: feasible inequality constraints are less than or equal to zero. The objective function was assumed to be a minimization objective. Since Modelica does not (yet) support the concept of methods or passing of functions as arguments, the optimizer was designed to use static inheritance. For this reason, the objective and constraint functions are replaceable functions within an optimizer package.

One feature of this library, that is not commonly available, is the ability to handle functions which are undefined over some region. The domain of the objective and constraints may not be known a priori. This occurs with MEC applications because the objective (e.g. equation (9)) and constraint functions (e.g. equation (11)) are typically evaluated using a solver. The solver may not find a solution. Hence, classical algorithms must be modified to recover from undefined evaluations.

Implementation of this capability was problematic because of the lack of numeric support for a real value which represents the concept of an undefined quantity. Either a native capability similar to Matlab's ® 'NaN', or operator overloading with the ability to extend a class from real numbers would have simplified implementation.

In this library, Modelica.Constants.inf was used to indicate that a function call was undefined. However, the language specification does not define behavior for operations (e.g. addition, subtraction, multiplication, division) on Modelica.Constants.inf. Therefore, all functions and statements which operated on variables that might be assigned a value of Modelica.Constants.inf required conditional expressions to ensure expected behavior.

While this optimization library will not be publicly released, it is available for further development. Contact the lead author for a copy.

5 Recommendations

While it is possible to realize both IMEC and DMEC controllers using Modelica 2.2, the addition of a standard optimization library and native support for embedded model simulation would vastly simplify implementation and maintenance.

Towards the goal of simplifying implementation of MEC, a recommended language improvement is the addition of a 'model simulate' function. The function would accept arguments that specify the model to simulate, the parameter values to use in each simulation, the outputs to return, and any solver specific settings. The solver should be able to be configured to solve both initialization problems and simulation problems. For efficiency in evaluation, the function should support both a scalar and vector lists of parameters. In addition to results which are associated with the model, there should be results associated with the solver. These results should be sufficient to diagnose solver failures. At a minimum, these should include the final time in the evaluation and an indication of whether the simulation successfully completed. A sample function definition along with an example invocation are shown in Figure 7.

```
function simulateModel
 input String modelName;
 input String paramNames[:];
 input String resultNames[:];
 input Real
        paramValues[:,size(paramNames,1)];
 input SettingsRecord solverSettings;
 output Real
      results[size(paramValues,1),
             size(resultNames,1)];
 …
end simulateModel;

// example call
[angle, speed, exitCondition, exitTime] =
    simulateModel(
        modelName="Library.PlantModel",
        paramNames{"w0", "theta0","u"},
        resultNames=
           {"w", "theta",
            "exitCondition", "exitTime" },
        paramValues=
              [0, 0, -100;
               1, 0, -100;
               …;
               2, 2*pi, 100],
        solverSettings =
            SettingsRecord(
                stopTime=1.0,
                fixedStep=0.1)
        );
```

Figure 7 - Model evaluation

It is important to point out that the goal is to be able to invoke such a function from within a running model and not simply as a command line analysis option. As previously mentioned, the ability to directly express such nested simulation relationships makes posing MEC problems much easier. If the MEC problem could also directly express the "optimization problem" associated with MEC then tools could also bring the underlying symbolic information to bear on efficient gradient evaluation as well.

One remaining issue for DMEC problems is the initialization of state variables in the embedded model. For DMEC problems we typically want the embedded model to start at the current state of the parent simulation. Said another way, the current values of the states in the parent simulation should be used as initial conditions in the nested simulation. Of course, it is possible using the function in Figure 7 to establish such a mapping but hopefully the language design group will consider alternatives that would be less tedious and error prone.

6 Conclusions

It is tractable to numerically synthesize near optimal (or approximately minimal) controllers for many systems. While in most cases the state feedback required for the controllers may make them impractical to deploy, they can certainly be used as prototype controllers that establish performance limits for a given design as well as provide insights into control laws for production controllers. Furthermore, this approach can easily integrate into a combined plant-controller optimization process. This can be done by making the optimal controller a function of the plant parameters. These optimal controllers can be realized as lookup tables (IMEC) or through the use of optimization and embedded models (DMEC). An algorithmic approach to controls synthesis was presented. For this paper, the IMEC and DMEC approaches were applied to an engine starting problem to generate an optimal controller in an automated fashion.

As this work has shown, Modelica is a promising technology for rapid prototyping of subsystem designs and prototype controllers. However, lack of support for 'model embedding' makes development and long term maintenance problematic because considerable work must be done to implement this embedding. Lacking any language standard, this work will always be tool specific. Furthermore, implementation of controllers which rely on optimization suffer from the lack of a standard optimization library. While an optimization library was developed for this work, it isn't practical for most users to make such an investment. By adding both language support to express the essential aspects of model embedding and optimization discussed in this paper, Modelica can evolve into a powerful technology for system development and optimization.

References

[1] H. K. Fathy,"Combined Plant and Control Optimation: Theory, Strategies, and Applications," Mechanical Engineering, University of Michigan, Ann Arbor, 2003.

[2] H. K. Fathy, P. Y. Papalambros, A. G. Ulsoy, and D. Hrovat, "Nested Plant/Controller Optimization with Application to Combined Passive/Active Automotive Suspensions."

[3] H. K. Fathy, J. A. Reyer, P. Y. Papalambros, and A. G. Ulsoy, "On the Coupling between the Plant and Controller Optimization Problems," in *American Control Conference*, Arlington, Va, 2001.

[4] P. R. Kumar and P. Variaya, *Stochastic Systems: Estimation, Identification and Adaption*. Englewood Cliffs, New Jersey: Prentice Hall, 1986.

[5] D. Bertsekas, *Dynamic Programming and Optimal Control: Vol 2*. Belmont, Mass: Athena Scientific, 1995.

[6] D. P. Bertsekas, *Dynamic Programming and Optimal Control: Vol 1*. Belmont, Mass: Athena Scientific, 1995.

[7] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-Dynamic Programming*. Belmont, Mass: Athena Scientific, 1996.

[8] M. A. Trick and S. E. Zin, "A Linear Programming Approach to Solving Stochastic Dynamic Programs," Carnegie Mellon University 1993.

[9] M. A. Trick and S. E. Zin, "Spline Approximations to Value Functions: A Linear Programming Approach," *Macroeconomic Dynamics,* pp. 255-277, 1997.

[10] D. P. de Farias and B. Van Roy, "The Linear Programming Approach to Approximate Dynamic Programming," *Operations Research,* vol. 51, pp. 850-865, November-December 2003.

[11] V. F. Farias and B. Van Roy, "Tetris: Experiments with the LP Approach to Approximate DP," 2004.

[12] D. P. de Farias,"The Linear Programming Approach to Approximate Dynamic Programming: Theory and Application," Ph.D. Dissertation, Department of Management Science and Engineering, Stanford University, Palo Alto, Ca, 2002.

[13] D. Dolgov and K. Laberteaux, "Efficient Linear Approximations to Stochastic Vehicular Collision-Avoidance Problems," in *Proceedings of the Second International Conference on Informatics in Control, Automation, and Robotics (ICINCO-05)*, 2005.

[14] G. J. Gordon, "Stable Function Approximation in Dynamic Programming," January 1995.

[15] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, Mass: MIT Press, 1999.

[16] R. Munos and A. Moore, "Barycentric Interpolators for Continuous Space and Time Reinforcement Learning," *Advances in Neural Information Processing Systems,* vol. 11, pp. 1024-1030, 1998.

[17] R. Munos and A. Moore, "Variable Resolution Discretization in Optimal Control," *Machine Learning,* vol. 1, pp. 1-24, 1999.

[18] J. M. Lee and J. H. Lee, "Approximate Dynamic Programming Strategies and Their Applicability for Process Control: A Review and Future Directions," *International Journal of Control, Automation, and Systems,* vol. 2, pp. 263-278, September 2004.

[19] D. P. de Farias and B. Van Roy, "Approximate Value Iteration with Randomized Policies," in *39th IEEE Conference on Decision and Control* Sudney, Australia, 2000.

[20] D. P. de Farias and B. Van Roy, "Approximate Value Iteration and Temporal-Difference Learning," in *IEEE 2000 Adaptive Systems for Signal Processing, Communications and Control Symposium*, 2000, pp. 48-51.

[21] B. Van Roy and J. N. Tsitsiklis, "Stable Linear Approximations to Dynamic Programming for Stochastic Control Problems with Local Transitions," *Advances in Neural Information Processing Systems,* vol. 8, 1996.

[22] P. W. Keller, S. Mannor, and D. Precup, "Automatic Basis Function Construction for Approximate Dynamic Programming and Reinforcement Learning."

[23] V. C. P. Chen, D. Ruppert, and C. A. Shoemaker, "Applying Experimental Design and Regression Splines to High Dimensional Continuous State Stochastic Dynamic Programming," *Operations Research,* vol. 47, pp. 38-53, January-February 1999.

[24] C.-C. Lin, H. Peng, and J. W. Grizzle, "A Stochastic Control Strategy for Hybrid Electric Vehicles," in *Proceedings of the 2004 American Control Conference*, 2004, pp. 4710-4715 vol. 5.

[25] E. D. Tate,"Techniques of Hybrid Electic Vehicle Controller Synthesis," Electrical Engineering: Systems, University of Michigan, Ann Arbor, Michigan, 2007.

[26] E. Tate, J. Grizzle, and H. Peng, "Shortest Path Stochastic Control for Hybrid Electric Vehicles," *Internation Journal of Robust and Nonlinear Control,* 2006.

[27] I. Kolmanovsky, I. Siverguina, and B. Lygoe, "Optimization of Powertrain Operating Policy for Feasibility Assessment and Calibration: Stochastic Dynamic Programming Approach," in *Proceedings of the American Control Conference*, Anchorage, AK, 2002, pp. 1425-1430.

[28] J.-M. Kang, I. Kolmanovsky, and J. W. Grizzle, "Approximate Dynamic Programming Solutions for Lean Burn Engine Aftertreatment," in *Proceedings of the 38th Conference on Decision & Control*, Phoenix, Arizona, 1999, pp. 1703-1708.

[29] C.-C. Lin, H. Peng, J. W. Grizzle, and J.-M. Kang, "Power Management Strategy for a Parallel Hybrid Electric Truck," *IEEE Transactions on Control Systems Technology,* vol. 11, pp. 839-849, November 2003.

[30] P. Y. Papalambros and D. J. Wilde, *Principles of Optimal Design: Models and Computation*, 2 ed. New York, New York: Cambridge University Press, 2000.

[31] J. Rust, "Using Randomization to Break the Curse of Dimensionality," 1996.

[32] S. Boyd and L. Vendenberghe, *Convex Optimization*. New York, N.Y.: Cambridge University Press, 2004.

[33] P. E. Gill, W. Murray, and M. H. Wright, *Practical Optimization*. New York, N.Y.: Academic Press, 1981.

[34] D. R. Jones, C. D. Peritunen, and B. E. Stuckman, "Lipschitzian Optimization without the Lipschitz Constant," *Journal of Optimization Theory and Applications,* vol. 79, pp. 157-181, 1993.

[35] A. J. Booker, J. Dennis, J. E. , P. D. Frank, D. B. Serafini, V. Torczon, and M. W. Trosset, "A Rigorous Framework for Optimization of Expensive Functions by Surrogates."

[36] M. J. Sasena,"Flexibility and Efficiency Enhancements for Constrained Global Design Optimization with Kriging Approximations," Mechanical Engineering, University of Michigan, Ann Arbor, 2002.