Model-based embedded design and control

Faster, Easier, Safer

John Milios
Sendyne Corp.
New York, NY, USA
jmilios@sendynecom

***Abstract* --- Model-based control is fast becoming the technology of choice for new, complex embedded control systems, such as those deployed at the Edge of the Internet-of-Things (IoT). The success of this technology rides on increasingly powerful MCUs, such as those based on the ARM M4, capable of handling multivariable control applications. Model-based control offers many advantages, including straightforward science-based formulation, faster development times, easier maintenance, as well as shorter/simpler redesign and update paths.**

Further, model-based control lends itself to implementation in various functional safety standard frameworks. A good example is ISO 26262, soon to be a mandatory standard for creating safe automotive systems.

In this paper we will review how model-based embedded design fits in the ISO 26262 framework as well as its advantages and disadvantages when compared to more traditional types of development. We will review some new technologies that advance model-based control, and illustrate their impact in the safety of a system. Finally, we will give an example on how these technologies can be applied in controlling the safety and performance aspects of a Li-Ion battery system.

Keywords --- model-based control; functional safety; Edge of the IoT

I. Introduction

Model based design, analytics, prediction and control is quoted as the technology of choice for modern complex systems. In this broad area of technologies the term "model-based" refers to the use of an explicit and separately identifiable model simulating the behavior of the subject system.

- In model-based development, models are used as an abstraction for the underlying hardware and as the main assets to enable concurrent and multi-platform development of complex hardware and software systems.
- In model-based analytics the model is used in an optimization loop to dynamically extract hidden process parameters through the measurements of process observable quantities.
- In model-based control the model is used to predict future behavior of the observed process as well as an input to the control algorithm in order to initiate preemptive actions.
- In model-based adaptive control, as shown in Fig. 1, the model besides providing its predictive function optimizes itself in order to represent with higher fidelity a dynamically evolving system.

As the sophistication of modern systems evolves so does the complexity of the underlying models. Models in the most general form can be described by a set of Partial Differential Equations (PDEs). These equations through discretization can be transformed to a set of Differential Algebraic Equations (DAEs). A class of simpler models can be described by a set of Ordinary Differential Equations (ODEs). Typically an analytical solution for these equations is either too complicated or more often does not exist and equations have to be solved utilizing numerical methods. This is one of the roles of the model solver. The model solver advances the model in time by finding the numerical solution that satisfies all model equations concurrently and within an acceptable error margin. There are several equation-based numerical model solvers for the desktop environment which are either general purpose (Mathematica, Matlab, etc.) or tailored to specific applications (Spice, gPROMS, etc.). While these solvers are well fitted for developing and optimizing a model in the desktop environment, their vast processing requirements deem them unsuitable for the embedded environment.

II. A Two-step Process

Porting a model from the desktop to the embedded environment is currently a two-step process. After the model has been developed and validated in the desktop environment it will

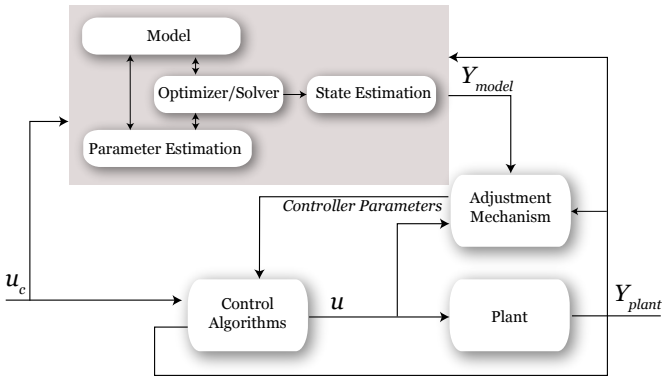

Figure 1: In model based control model predictions are utilized by the control algorithms. In adaptive control the model adapts to measurements to optimize its parameters.

have to be translated into embedded code that can run into an embedded processor. The embedded code can be either created manually if the model is simple enough or created automatically as in the case with the MathWorks Embedded Coder [1]. There are two issues that we want to point out with this process.

The first issue arises with the quality of the code produced. The State-of-the-Art today cannot handle DAEs, uses fixed time steps and employs only a limited number of algorithms. As a result the code exhibits poor performance in handling stiff problems. The automatically generated code also needs non-trivial tweaking for handling memory management issues.

The second issue is related to code validation and verification. As shown in Fig. 2, each time there is a change in the model equations the process has to be repeated, adding to the development cost. In the case of automatic code generation this extra verification step is suggested to be limited in verifying that the new code produced conforms to the model verified in the desktop environment [2].

III. One-step Process

dtSolve is a model numerical solver toolbox designed for embedded applications. The concept is to use the same model solver both on the desktop and the embedded environment making model development and deployment a one-step process. A developer will use *dtSolve* on the desktop to design and optimize an equation based model. When the model is validated then the same model with the same model solver will be ported to the embedded processor. This simplification in development was made possible by the small size of the solver, its high execution speed and the specially designed embedded memory management functions.

dtSolve is a complete model solving toolbox written in C++ and utilizing State-of-the-Art methods including Automatic Differentiation, modern DAE formulations with a clean narrow interface to the DAE solver, sparse matrix techniques, variable time-steps along with a sensitivities and optimization toolbox [3].

IV. Features of the Embedded Model Solver *dtSolve*

dtSolve is written in C++ and has been designed and tested over a period of a few years in battery cell and pack modeling. A block diagram of the *dtSolve* kernel is shown in Fig. 3. Some of the key features of the implementation are listed and elaborated upon in the following paragraphs.

A. Automatic Differentiation

Automatic Differentiation (AD) is fundamentally a simple idea (essentially "just" the chain rule of calculus), but the way it is implemented can significantly affect the performance of a model solver. Techniques for implementation of AD are now well-developed and widely accepted in the modeling community [4].

Briefly, AD is a mixed symbolic numeric technique that provides an accurate numerical value of a derivative without finite-differencing. One approach to AD is an off-line technique [5] in which the user formulates a subroutine to compute a residual vector. The source code for this subroutine is then submitted to a compiler-like tool which returns source code for the computation of the residual and its associated Jacobian matrix or gradient.

The computation implemented by the derivative code is typically more sophisticated and efficient than simply applying symbolic differentiation to the residual equations. This technique can give very efficient derivative code, but does not allow run-time selection of which variables to take derivatives with respect to; *i.e.*, which variables are considered independent. For this reason, *dtSolve* instead implements an on-line scheme.

The distinction between forward and reverse AD is well described in the literature and is closely related to the so-called adjoint method for transient solvers [6]. In a nutshell, forward AD is preferred when one wants derivatives of a large number of expressions with respect to a relatively small number of independent variables. On the other hand, reverse AD is efficient when one wants derivatives of a relatively small number of expressions with respect to a large number of independent variables.

dtSolve implements forward AD because it is a bit simpler to implement and works well for typical applications of battery pack simulations.

Such an on-line, forward AD scheme is implemented in

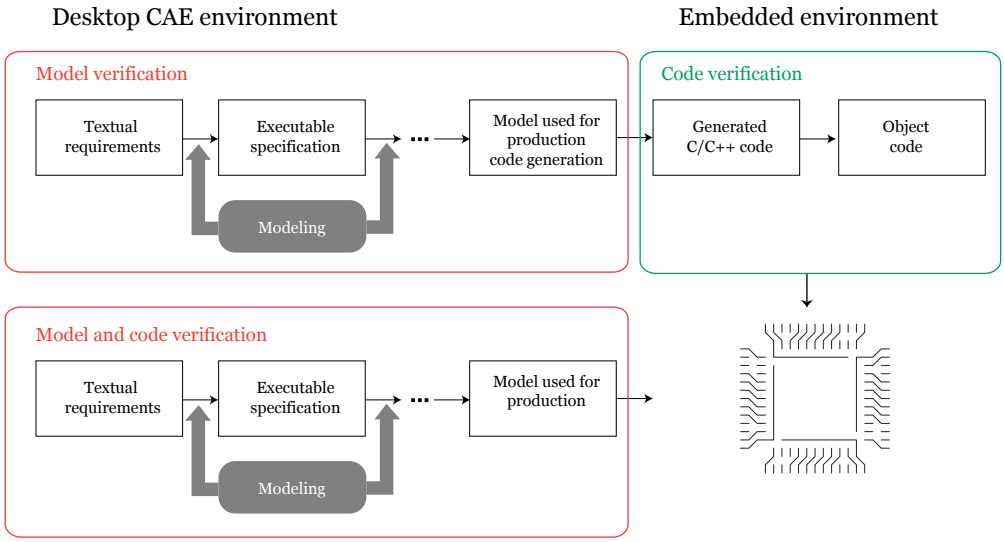

Figure 2: In the two-step process shown on top each time the model is modified the embedded code has to be verified also. The ability to embed the model along with its solver directly into the embedded processor eliminates one extra step in verification, as shown on the bottom flowchart.

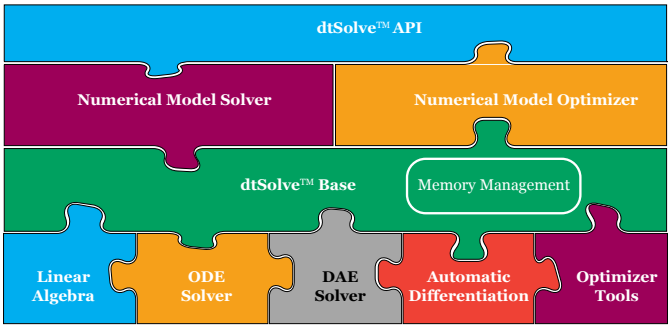

Figure 3: *dtSolve* provides all the functions of an equations-based model solver and optimizer

dtSolve with a C++ custom data type. The operator overloading feature of C++ [7] is used by this data type so that a user can formulate expressions using familiar arithmetic syntax. If needed, it is possible to obtain expression values without gradient computation. This is more efficient especially from a memory usage point of view.

For large systems, the AD scheme provides a facility to get a gradient in sparse form – *i.e.*, a list of (index,value) pairs for the gradient entries which are not symbolically zero.

The systematic use of AD throughout *dtSolve* provides several advantages over hand-coded derivatives:

- The possibility of a symbolic error in a derivative expression is avoided.
- Modeling code can be written in familiar form without the clutter of derivative computations.
- It is possible to turn on and off derivative computation at runtime for efficiency. It is also possible to nominate new independent variables "on the fly".

B. Modern DAE Formulation

Formulating a problem as a DAE – instead of a traditional ODE – simplifies the model description and can avoid the pitfalls associated with the conversion of DAEs to ODEs such as a significant increase of the model size. Modern DAE solvers are highly developed and are thoroughly competitive with ODE solvers in terms of performance [8]. For models than can be formulated directly into ODEs, *dtSolve* provides an ODE solver with sensitivities capability.

dtSolve is based on a classic DAE [9] formulation which is the preferred method for complex physical models.. Moreover, the formulation used by *dtSolve* facilitates a frequency domain analysis.

The formulation is expressed as a system of residual equations:

$$\mathbf{G}(t,\mathbf{y},\dot{\mathbf{y}},\mathbf{p})=0 \qquad (1)$$

Where,

- $\mathbf{y}(t)\in R^n$ is the state at time t
- $\mathbf{G}: R\times R^n\times R^n\to R^n$ is a smooth mapping
- $\mathbf{p}$ is a vector of model parameters

A traditional ODE can be easily represented in the form of (1) as $\dot{\mathbf{x}}=\mathbf{F}(t,\mathbf{p},\mathbf{x})$.

C. Clean, Narrow Interface to DAE Solvers

dtSolve interfaces to existing codes for transient solves, linear algebra, non-linear static solves and constraint optimization. Such codes are typically of very high quality and have been thoroughly vetted and tested by the community. Achieving such a narrow interface to external numerical packages requires some non-trivial technical details in the design of *dtSolve* which are described later in this document. Numerical packages and subroutines not needed for a particular application are removed by the compiler and linker toolchain before forming the final executable for space efficiency.

D. Sparse Matrix Techniques

dtSolve integrates sparse matrix techniques. This feature adds to the speed of model execution and allows the construction of more complicated models without sacrificing performance. A speed improvement and memory reduction of several orders of magnitude has been demonstrated for medium-sized examples.

E. Object Oriented Design

Object oriented design – using the C++ language – contributes to the modularity of *dtSolve*. For example, *dtSolve* makes extensive use of the compile-time polymorphism and template meta-programming facilities of C++. This makes it possible to transparently enable or disable *dtSolve* features depending on user inputs, such as switching between dense and sparse algebra, activating or deactivating sensitivities computation, or using AD to perform Jacobian computations. Moreover, this also facilitates the integration of *dtSolve* into larger user projects, as the model solving steps are entirely encapsulated in dedicated code entities.

F. Sensitivities and Optimization

dtSolve facilitates the computation of time-domain sensitivities [10] during a transient solution. These quantities can then be used by the optimization code for model fitting, parameter extraction, or to adapt the model in pseudo real-time.

Given a time-varying state variable $\mathbf{x}(t)$ and a system parameter $\mathbf{p}$ (*e.g.*, a resistor value in a circuit model) the sensitivity of $\mathbf{x}$ w.r.t. $\mathbf{p}$ is the time varying quantity $d\mathbf{x}/d\mathbf{p}$.

The sensitivity equations are derived from Eq. (1) by applying implicit differentiation w.r.t. the parameter $\mathbf{p}$ (in general, a vector of several parameters) [15]. With almost any solver it is possible to estimate sensitivities using finite-differencing [11], but the use of AD as described above coupled with a sensitivity-enabled DAE solver is faster, more accurate and numerically more stable.

Given a candidate compact model for a cell, an optimization process is applied by *dtSolve* to find model parameters that provide a best fit to experimental data. For example, a time-varying load current can be applied to the cell model resulting in a time-varying output voltage and *dtSolve* can optimize the model parameter values to reproduce available measurements.

dtSolve currently uses a reliable non-linear least squares solver [12] which attempts to minimize the sum-of-squares difference between the simulated and measured quantities. In addition, *dtSolve* also provides other optimization tools to perform, for example, constrained or global optimization with or without derivatives evaluation.

G. Further capabilities

The use of C++ class to describe a system of equations provides additional capabilities such as hierarchical formulations and parameter passing. For example, suppose one has defined an individual battery cell as a C++ class and wants to arrange several such cells into a pack. Given a number of rows and columns for the battery pack (which become known at run-time) it is straightforward to write two nested loops that will allocate instances of the individual cell model (using the

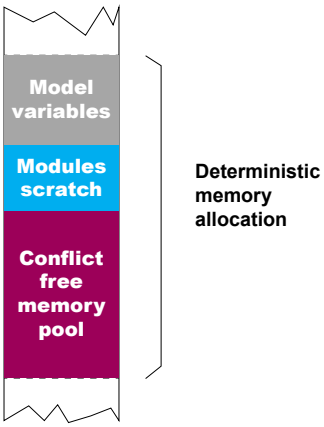

Figure 4: All memory required for solver operation is allocated during initialization and instantaniation. No dynamic memory allocation after this point in time.

new operator) and supply the interconnecting nodes required to wire the cells together.

V. Performance

dtSolve performance was benchmarked in several occasions against MATLAB Coder™ The lack of DAEs support in MATLAB Coder™ makes it impossible to perform a direct comparison based on the previous simulation using *CellMod*. In order to appreciate *dtSolve's* capability in handling stiff systems we selected the "Van der Pol" [16] circuit equation as a classical example of a stiff problem.

The "Van der Pol" oscillator equation is given by:

$$\ddot{\mathbf{x}} - \mu(1-\mathbf{x}^2)\dot{\mathbf{x}} + \mathbf{x} = 0 \quad (2)$$

Eq (2) was numerically solved by *dtSolve* as well as MATLAB Coder™ configured alternatively with the ode23 [14] and the ode45 option [15]. For the comparison the following settings were used: relative tolerance 10^{-3}, absolute tolerance 10^{-6}, no upper bound on the step size. Compilation was performed using *GCC ARM* with typical optimization flags (-O3) and the benchmark was performed on a 2.7 GHz Intel i5 processor. Eq. (2) was integrated from t=0 to t=2000s and the integration was repeated 100 times for each implementation in order to get meaningful results. The results are shown in Table 1.

TABLE 1

	MATLAB Coder™		*dtSolve*
	ode45	ode23	
Mean time per integration (ms)	474	187	2.86
# of steps	1,516,100	1,527,370	516
Speed index	1	2.5	**165**

Table 1: *Performance comparison in solving the Van der Pol equation. The speed index indicates multiples in speed of execution using as reference the slowest one (highest value is the best).*

From Table 1 it can be seen clearly the advantages of *dtSolve* in handling stiff systems. These advantages become more apparent in low power and real-time applications.

VI. Memory Management as a Safety Issue

One of the most safety critical aspects of using a model solver in an embedded environment is memory management. A model solver is handling concurrently multiple tasks that each requires its own memory area. On the desktop environment memory is aplenty but nevertheless one of the most common reasons for software crashes is fragmentation and unserved memory allocation requests. In an embedded application, memory is limited and dynamic memory allocation cannot be employed. *dtSolve* implements a patented memory allocation strategy that guarantees memory availability for all solver tasks at all times. During initialization and instantanation the program determines the memory space needed for each solving task. Through this analysis it allocates a static memory block where all model variables and the various module scratch areas reside. In addition it creates a conflict free memory pool managed by the solver itself as shown in Fig. 4.

VII. Solver and the ISO 26262

Part 8 of the ISO 26262 [17] international standard defines criteria for the qualification of software tools utilized in the development of a system. For a model based design the model solver used for the model development and subsequently for the system deployment needs to be verified for correct safety functioning. Fig. 5 illustrates the criteria used for determining the need for software tool qualification. Tool confidence levels TCL4-TCL2 require software tool qualification. For example, if

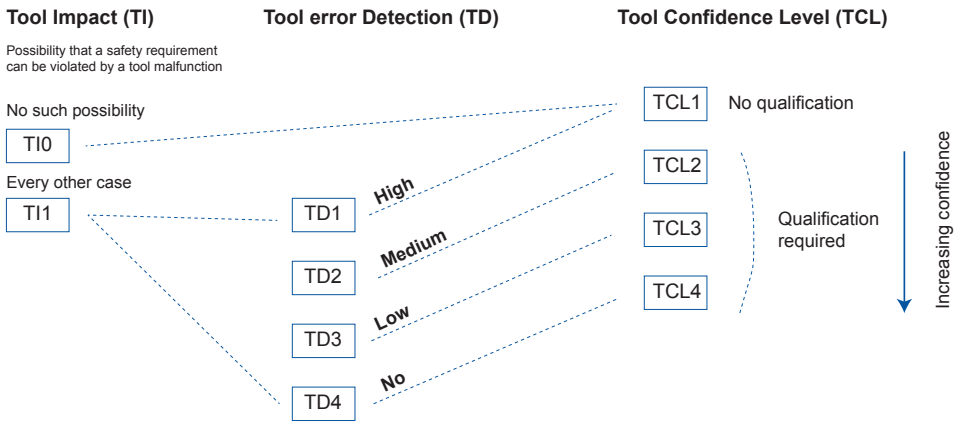

Figure 5: A software tool needs to be qualified if there is a possibility for a safety requirement violation and if the tool error detection mechanism is not high.

Qualification of software tools classified TCL3

	Methods	ASIL A	ASIL B	ASIL C	ASIL D
1a	Increased confidence from use	++	++	++	+
1b	Evaluation of the development process	++	++	++	++
1c	Validation of the software tool	+	+	+	++
1d	Development in compliance with a safety standard*	+	+	+	++

*No safety standard is fully applicable to the development of software tools. Instead, a relevant subset of requirements of the safety standard can be selected

++ Method highly recommended
+ Method recommended
o Method not applicable

Figure 6: Example of tool qualification method for TCL3 classification.

a software tool has the capability to violate a safety requirement and the same tool has low error detection mechanism, the tool is classified at TCL3 requiring qualification. The method for qualifying the software tool is described in Part 8 of the specification and it shown as an example in Fig. 6. Depending on the Automotive Safety Integrity Level (ASIL) of the system (A, B, C or D) one or more of the suggested methods has to be employed for the qualification process. The tool qualification can be part of the total system qualification process or it can be performed independently regarding the tool as a stand-alone entity. In this case the validity of this qualification needs only to be confirmed. The standard provides a detailed explanation of how each step has to be performed.

While Part 8 of the ISO 26262 can be adequate for the use of a model solver in the model development stage, much stricter requirements apply for the model deployment in a safety critical system. In this case the software embedded in the system has to follow the product development at the software level guidelines described in Part 6 of the standard.

What is interesting in this case is that it has only to be performed once. For example changing the model will not require the model solver software re-qualification. The same model solver software can also be utilized in different systems without any need for re-qualification as long as the ASIL does not exceed the originally ASIL for which the software was qualified [18].

VIII. Conclusions

Model based development and control for embedded systems is gaining acceptance as the preferred method for complex, physics-based systems. A model solver that can be embedded as is within a microcontroller, besides providing flexibility to the sophistication of utilized models, can improve the quality of the final product, simplify the qualification process as it has to be performed only once and save development costs and time.

References

[1] MathWorks, "MathWorks Embedded Coder," 2017. [Online]. Available: MathWorks Embedded Coder. [Accessed: 15-Jan-2017].

[2] E. Dillaber, L. Kendrick, W. Jin, and V. Reddy, "Pragmatic Strategies for Adopting Model-Based Design for Embedded Applications," 2010.

[3] R. Melville, N. Clauvelin, and J. Milios, "A high-performance model solver for 'in-the-loop' battery simulations," in American Control Conference (ACC), 2016, 2016, pp. 3119–3125.

[4] S. Forth, P. Hovland, E. Phipps, J. Utke, and A. Walther, Recent Advances in Algorithmic Differentiation, vol. 87. Springer, 2012.

[5] A. Griewank and others, "On automatic differentiation," Mathematical Programming: recent developments and applications, vol. 6, pp. 83–107, 1989.

[6] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland, "ADIFOR-generating derivative codes from Fortran programs," Scientific Programming, vol. 1, no. 1, pp. 11–29, 1992.

[7] Y. Cao, S. Li, L. Petzold, and R. Serban, "Adjoint sensitivity analysis for differential-algebraic equations: The adjoint DAE system and its numerical solution," SIAM Journal on Scientific Computing, vol. 24, no. 3, pp. 1076–1089, 2003.

[8] B. Stroustrup, Programming: principles and practice using C++. Pearson Education, 2014.

[9] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward, "SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers," ACM Transactions on Mathematical Software (TOMS), vol. 31, no. 3, pp. 363–396, 2005.

[10] L. R. P. Uri M. Ascher, Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations. SIAM Press, 1998.

[11] R. K. Brayton, Sensitivity and optimization. Elsevier Science Inc., 1980.

[12] J. R. R. A. Martins, "Sensitivity Analysis," 2001. [Online]. Available: http://aero-comlab.stanford.edu/jmartins/aa222/aa222sa.pdf, accessed 2/29/2014

[13] R. J. Hanson, "Linear least squares with bounds and linear constraints," SIAM Journal on scientific and statistical computing, vol. 7, no. 3, pp. 826–834, 1986.

[14] MATLAB ode23. [Online]. Available: http://www.mathworks.com/help/matlab/ref/ode23.html. [Accessed: 14-03-2016].

[15] MATLAB ode45. [Online]. Available: http://www.mathworks.com/help/matlab/ref/ode45.html. [Accessed: 14-03-2016].

[16] B. Van der Pol, "LXXXVIII. On 'relaxation-oscillations,'" The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, vol. 2, no. 11, pp. 978–992, 1926.

[17] "ISO 26262: 2011 Road vehicles - Functional safety." [Online]. Available: http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=43464. [Accessed: 15-Jan-2017].

[18] R. Bates, "Component Reuse in Safety Critical Systems." [Online]. Available: https://www.mentor.com/embedded-software/multimedia/player/ecu-component-reuse-in-iso-26262-safety-critical-systems-e02b71c7-be6d-4f47-bfe3-9f1fad1577c9. [Accessed: 16-Jan-2017].