Automatic Code Generation: Facilitating New Teaching Opportunities in Engineering Education

Pieter J. Mosterman
The MathWorks, Inc.,
Natick, MA 01760
pieter.mosterman@mathworks.com

Abstract - **Model-Based Design in industry relies heavily on automatic code generation technology. The need to obtain code for different configurations such as rapid prototyping, hardware-in-the-loop simulation, and processor-in-the-loop simulation introduces extensive code generation from high-level models and to be cost effective and competitive requires the automation of this code generation. This paper shows some of the issues that embedded systems engineers have to negotiate as a consequence of this. It indicates that there is a shift in required skills from the algorithmic to the architectural level. As such, automatic code generation technology provides an opportunity in academia to better prepare students to be successful in the field of embedded control system design by including such architectural aspects in the curriculum, while shifting focus of algorithm design from creation to inspection.**

Index Terms - Engineering education; model-based design; automatic code generation

INTRODUCTION

In its role of preparing the future workforce, a critical aspect of engineering education is to provide students with the skills and knowledge required to be successful in industry as well as appealing from a recruiting perspective.

As such, it is important for academia to be aware of the needs and practices of industry and to recognize and act upon paradigm shifts as they occur. Conversely, industry has a responsibility to make the characteristics of prospective employees that are required are well known. As a corollary to that, tool vendors should educate academia as well as industry on the benefits and usages of their products.

An important shift in industrial practice that has profound implications with regards to the required set of skills of employees is the adoption of Model-Based Design. The pervasive access to computing power originally was mostly exploited by automating support tasks in the design process such as word processing and data storage. This led to the popularity of office software and posed a need for engineering education to include training in word processing, spread sheet operation, and database handling in the curriculum.

More recently, computing power has taken on a more integral role in system design by automating part of the required effort. As this has been facilitated by the use of computational models, modeling has become an increasingly important aspect of the required skill set of the engineering workforce. In addition, technologies that leverage the availability of models enable reaping the benefits of computational models in addressing the challenge of system design.

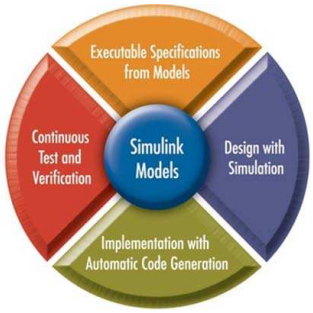

FIGURE 1
MODEL-BASED DESIGN.

Figure 1 illustrates the central position of computational models in Model-Based Design. An industrial-strength modeling environment such as Simulink® [1] supports the different technologies that comprise a successful Model-Based Design:

- Executable specifications allow immediate feedback on the behavior of a specification, as opposed to documented behavior that often is misinterpreted.
- Simulation in the design supports a quicker search of the design space as opposed to constructing hardware prototypes.
- Automatic code generation reduces the tedious and error-prone stage of translating a design into a specification for the software engineers and manually writing the computer code accordingly.
- Test and verification can be performed in a much earlier stage in the design as a computational model is available with access to all internal variables, including those that may be difficult to obtain on a hardware prototype.

This paper concentrates on how automatic generation technology is reshaping the design of embedded systems and how this potentially impacts education. It furthermore discusses possibilities in education to take advantage of such

technology to provide students with experiences that would otherwise have required prohibitively extensive programming assignments.

The following section provides the context for automatic code generation technology by giving an overview of the typical approach to embedded control system design. The next section then illustrates automatic code generation in detail based on an electronic throttle control system. The effects that automatic code generation may have on education are then discussed and the final section presents the conclusions of this work.

AN INDUSTRIAL DESIGN APPROACH

The complexity of embedded systems in terms of scale has dramatically increased over the past decades. In particular, the now prolific use of embedded computing power has enabled functionality much beyond what could be achieved with purely analog equipment. For example, in the automotive business the key differentiator between models has become features such as power seats, central door lock, park assist, built-in audio and video systems, which all rely on embedded computers. This trend is projected to continue, in particular when safety-critical systems such as braking and steering are implemented in embedded processors.

In addition, in order to stay competitive, the time-to-market of new automobile models has significantly reduced and to manage this confluence of factors successfully systematic design methods are essential.

A typical design approach in the automotive as well as the aerospace industry is 'V' shaped, as illustrated in Figure 2. The approach starts at the top left with documenting the requirements for the system to be designed. These requirements are then processed into a set of formal specifications that constitute the foundation for a set of models of the desired system. Simulation allows prompt feedback on the designed system behavior and changes can be made to the model in case undesired behavior is present.

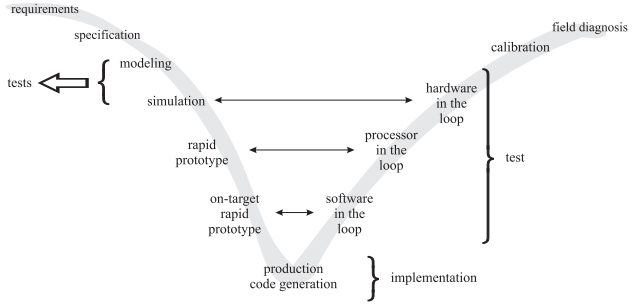

FIGURE 2
AN INDUSTRIAL DESIGN APPROACH.

Once a model is designed that behaves according to the requirements, a preliminary validation of feasibility is performed by producing a rapid prototype that includes some of the non-functional, and, therefore, not modeled, characteristics. These characteristics include response time, jitter, and delay behavior, and may require changes to the original design. On-target rapid prototyping is then employed to ensure appropriate behavior using the actual microprocessor that is envisioned in the production version of the designed system. This stage validates characteristics such as the compiler that is used, fixed-point effects, and interaction with other systems.

Finally, when all prototyping shows that a feasible and correct design has been arrived at, the production code is produced. This is highly optimized code that will be used in the high-volume product, and is rigorously verified, validated, tested, and tuned while moving up the right-hand stroke of the 'V' in Figure 2. A number of different configurations such as software-in-the-loop, processor-in-the-loop, and hardware-in-the-loop are used to this end [2].

The successful execution of the above sketched approach is predicated upon the availability of automatic code generation facilities as different forms of code have to be obtained for the different stages such as rapid prototyping, on-target rapid prototyping, production code, and processor-in-the-loop simulation. Consequently, automatic code generation is an important, if not critical, technology for the design of embedded systems and familiarity with it is a necessity for the engineers active in this domain.

AN ELECTRONIC THROTTLE CONTROL SYSTEM

To study the skills required in working with automatic code generation facilities, let us concentrate on the electronic throttle control as found in modern automobiles.

I. Control Structure

Figure 3 shows a paradigmatic structure of the throttle control. The discrete event state machine `Throttle Sequencer` is used to schedule the execution of the `Throttle Position Controller` task. Such explicit execution control requires an imperative model of computation as embraced by the dashed/dotted function-call connection from the `Throttle Sequencer` to the `Throttle Position Controller`. The `Throttle Sequencer` executes the `Throttle Position Controller` in a nominal mode of behavior. When failures occur and during startup and shutdown, execution of the `Throttle Position Controller` may be suspended.

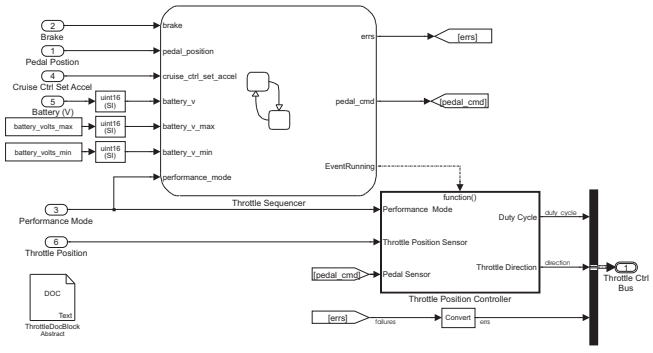

FIGURE 3
THROTTLE CONTROL.

The `Throttle Position Controller` embodies the low level feedback control to position the throttle according to a setpoint. In this example, position, integral, and derivative (PID) control is employed. This type of control takes in the difference between the setpoint and actual position of the throttle and computes a control force as a weighed sum of this difference, the time integral of the difference and the time derivative of the difference.

II. Automatic Code Generation

In Figure 4, a model of the position and integral computation of the PID control is shown. When code is automatically generated for this model, the following code fragment represents the computations of the `P Gain` and `PI_Sum` blocks:

```
/* Gain: '<S1>/P Gain' */
rtb_PI_Sum = rtu_Err * localP->PGain_Gain;

/* Sum: '<S1>/PI\_Sum' */
rtb_PI_Sum += rtb_Saturation;
```

Inspection of the code to review whether it properly reflects the model in Figure 4 is aided by the generated comments that state which block is responsible for each line of code. An HTML version of the code indeed hyperlinks block references such as `'<S1>/P Gain'` directly back to the original Simulink model.

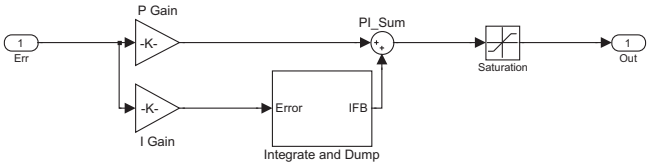

FIGURE 4
THE POSITION AND INTEGRAL PID CONTROL PART.

The above code fragment shows the code generated for the `P Gain` and `PI_Sum` blocks. Notice the need to understand the use of standardized data structures to store input variables, parameters, and block variables, here prefixed by `rtu_`, `localP`, and `rtb_`, respectively, but in general these prefixes can be customized.

In typical automotive applications, to save on stack space, function arguments are defined as globals and not explicitly passed in through the argument list. This is a significant departure from the focus in education on the use of local variables because of the increased modularity, which renders it less error prone.

Automatic code generation can be configured by subsystem options to generate local variables that are passed into a function by means of an argument list. For the PID control in Figure 4, if the PID functionality is selected to be a *reusable function*, the following function header is automatically generated:

```
void PI_Ctrl(real_T rtu_Err, rtB_PI_Ctrl *localB,
 rtDW_PI_Ctrl *localDW, rtP_PI_Ctrl *localP)
```

The corresponding data structures are defined in the accompanying header file.

III. Code Optimization

To obtain code with a smaller memory footprint and lower computational complexity, the user may choose to apply optimizations such as *expression folding* to remove superfluous variables. For the model in Figure 4, this results in the functionality of the `P Gain` and `PI_Sum` blocks to be combined into one line of code:

```
/* Sum: '<S1>/PI\_Sum' incorporates:
 *  Gain: '<S1>/P Gain'
 */
rtb_PI_Sum = rtu_Err * localP->PGain_Gain +
  rtsaturate_U0DataInY0Container;
```

IV. Fixed-Point Code

Another important aspect of control systems in industry is their implementation using fixed-point computations because fixed-point microprocessors are less expensive than the floating-point counterpart. Some control functionality may be entirely implemented in fixed point, whereas in other cases, a floating-point implementation is employed with a fixed-point back-up system. In the latter case, this results in optimal behavior in nominal modes of operation, while in the face of faults, a rudimentary fixed-point control prevents total failure.

The PID control in Figure 4 can be transformed into a fixed-point version by specifying the desired fixed-point data type. The polymorphic behavior of the model elements such as the gain and sum blocks causes the fixed-point functionality to be implemented.

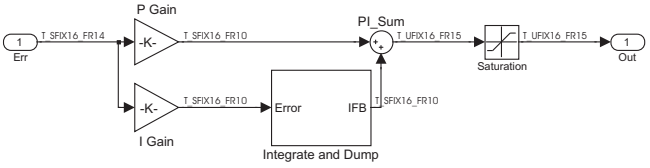

FIGURE 5
PID CONTROL IN FIXED POINT.

In Figure 5 a fixed-point version of the PID control in Figure 4 is shown. The fixed point data types are shown on the signal connections as, for example, `T_SFIX16_FR14`, which is shown as the fixed-point data type of the input signal of the `Err` port. To determine the fixed-point characteristics of this signal, its details can be requested at the MATLAB® [3] prompt

```
>> T_SFIX16_FR14

T_SFIX16_FR14 =


    DataTypeMode: 'Fixed-point: binary point scaling'
          Signed: true
      WordLength: 16
```

```
 FractionLength: 14
        IsAlias: true
     HeaderFile: 'eng_ctrl_prj_types.h'
    Description: ''
```

This shows that the signal value is signed and stored in a 16 bit word where 14 bits are used to represent the fraction of the signal value. Notice that this data type is mapped onto a signed short in the `eng_ctrl_prj_types.h` header file:

```
typedef signed   short T_SFIX16_FR14;
```

A fixed-point version of the PID control code that includes *expression folding* can now be automatically generated again. The part that reflects the `PI_Sum` and `P Gain` blocks takes the form:

```
    /* Sum: '<S1>/PI\_Sum' incorporates:
     *  Gain: '<S1>/P Gain'
     */
    rtb_PI_Sum =
      (uint16_T)((uint32_T)((rtu_Err *
        rtp_P_Gain >> 14) << 5U) +
      (uint32_T)((uint16_T)tmp_a << 5U));
```

Here the left and right bit shift operators, `<<` and `>>`, respectively, are used for optimal multiplication and addition in fixed point.

V. Including Legacy Code

Embedded software has been extensively used for decades in industry such as automotive and aerospace. Therefore, automatically generated code often must be integrated with existing code, so-called *legacy code*, that has proven its quality and value.

Importing Interface Definitions

In legacy code, interface structures are typically specified in header files. These header files can be read by Simulink to import data structures that are defined in legacy code with which the automatically generated code needs to integrate. For example, in the `eng_ctrl_prj_types.h` header file, the throttle control data structure `T_THROTTLE_CTRL` is defined to be:

```
typedef struct {
  T_UFIX16_FR15 duty_cycle;
  T_BOOLEAN     direction;
  T_BOOLEAN     errs[3];
} T_THROTTLE_CTRL;
```

The elements that it contains are the `duty_cycle` which determines the force with which the throttle is moved, the `direction` which captures the direction of movement of the throttle, and the `errs` which is a three-dimensional signal of the proportional, integral, derivative errors.

After importing this structure definition from the header file, it becomes accessible in the MATLAB workspace as any regular MATLAB expression:

```
>> T_THROTTLE_CTRL

T_THROTTLE_CTRL =

Simulink.Bus
    Description: ''
     HeaderFile: 'eng_ctrl_prj_types.h'
       Elements: [3x1 Simulink.BusElement]
```

This shows the header file that contains the structure source and viewing one of the elements in the Simulink Bus provides the detailed information of the signal characteristics:

```
>> T_THROTTLE_CTRL.Elements(1)

ans =

Simulink.BusElement
            Name: 'duty_cycle'
        DataType: 'T_UFIX16_FR15'
      Complexity: 'real'
      Dimensions: 1
    SamplingMode: 'Sample based'
      SampleTime: -1
```

This shows the `duty_cycle` bus element to be a fixed-point signal (unsigned 16 bit word with a 15 bit fraction) that is real (i.e., not a 'complex'), a scalar, and sampled with inherited sample time.

Importing Algorithms

In addition to interface definitions, entire algorithms implemented in legacy code may have to be integrated into automatically generated code.

To illustrate, consider calibration tables as used in automotive applications to determine control setpoints such as throttle opening and spark advance. Such calibration tables are essential in the behavior of the automobile and require a tremendous amount of effort to achieve optimal behavior in different modes of operation (warm up, high performance, economy, etc.).

Particular aspects in the use of calibration tables are the interpolation algorithms that are employed to compute throttle opening and spark advance setpoint values for input values in between grid points of the calibration table. For example, the following code may be employed for interpolation in 1-dimensional look-up tables:

```
#include "LookUpTables.h"
T_UFIX16_FR15 LookUpTable1D_Interp(
    const T_UFIX16_FR13  InputValue,
    const T_UFIX16_FR13 *InputMap,
    const T_UFIX16_FR15 *OutputMap,
    const T_UINT16       MapLength)
{
  T_SINT16 idx = 0;
  T_SINT16 idx_n1 = 0;

  if (InputValue <= InputMap[0])
      return( OutputMap[0]);
  if (InputValue >= InputMap[MapLength-1])
      return( OutputMap[MapLength-1]);

  /* Search for the Element in the InputMap Above the Input Value*/
  while ( (idx < (MapLength-1)) && (InputMap[idx] < InputValue) )
    idx++;

  return ( (T_UINT16)
           ((T_SINT16) OutputMap[idx\_n1] +
            (T_SINT32) ((2\^3)*
            ((T_SINT32) (InputValue     - InputMap[idx_n1] )   *
             (T_SINT32) (OutputMap[idx] - OutputMap[idx_n1])
```

```
            )) /
              (T_SINT32) (InputMap[idx]  - InputMap[idx_n1])    )
           );
}
```

Such algorithms tend to be tried and tested in the field, and have proven themselves for many years in products. Therefore, rather than re-designing them, the legacy code may have to be reused.

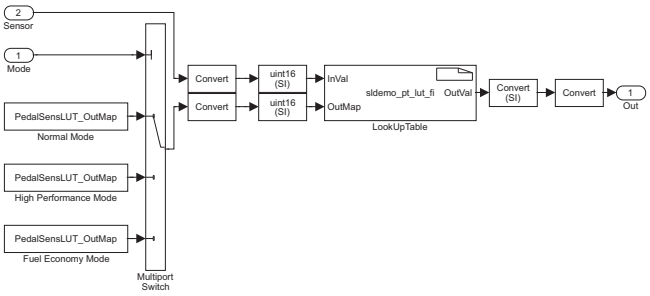

FIGURE 6
INCLUDING LOOK-UP TABLE LEGACY CODE.

In Figure 6, the look-up table code is integrated into a part of the throttle controller that computes the setpoint for the PID control that governs the throttle opening. Depending on the `Mode`, different calibration tables may be employed; one for `Normal Mode`, one for `High Performance Mode`, and one for `Fuel Economy Mode`. In order to properly integrate the legacy look-up table code, data type conversions may be required so the fixed-point legacy code can be integrated with a floating-point version of the overall control as well as a fixed-point version. The look-up table algorithm is then wrapped into an S-function interface, where it constitutes the `mdlOutput` function.

Other options of integrating legacy code exist. For example, the code can be explicitly included in a specific function that is called during execution or it can be called using the Stateflow [4] action language. Details of this integration will not be given here but can be found in the documentation of the products mentioned.

AUTOMATIC CODE GENERATION AND EDUCATION

Automatic code generation technology is enabling some important opportunities in education.

I. The Changing Skill Set

The example of the electronic throttle control system has illustrated the automatic code generation technology and how it is increasingly being employed in industry. To provide students with the skills to be attractive hires in the domain of embedded control system design and for them to be successful, it is imperative to familiarize students with the aspects of automatic code generation as sketched for this electronic throttle control system.

The most conspicuous change in required skills for software engineers in the embedded control system domain is probably the shift in focus from the algorithmic to the architectural aspect such as interfaces and file dependencies. The pervasiveness of architectural notions in throttle control system description represents how important they are in industry. Rather than designing very specific control algorithms in computer code such as C and C++, software engineers increasingly rely on the automatic code generators to produce these algorithms. Instead, attention has shifted to integrating the automatically generated code into legacy code by importing legacy data structures, memory maps, generating the correct function-call argument lists, and reading data files with, for example, parameter values.

The algorithmic side of the software engineering effort is increasingly handled by automatic code generators, and, therefore, the ability for software engineers to produce this code is shifting to the ability to review the automatically generated code. Intimate syntactic knowledge to generate C and C++ code is becoming less important than understanding how optimization algorithms affect the expression formulation.

Furthermore, the range of algorithms that need to be dealt with is increasing as it is not limited to only fixed point, but ranges from floating point non-optimized code all the way to fixed-point code with highly optimized bit shift operations to implement modeled functionality.

II. Eliminating the Tediousness

Another important effect of automatic code generation technology in education is that it removes much of the tedious work to implement a control system. Removing the need for students to manually produce the code enables experimentation as hardware-in-the-loop simulation and rapid prototyping.

This allows students to experiment with real-world effects such as noise, signal conversion lag-times, sample-time overruns, and dynamic scheduling effects. Furthermore, it allows students to design high-level control strategies and see their effect in real-world operation.

CONCLUSIONS

This paper has given an overview of Model-Based Design in industry and discussed the use of automatic code generation as an enabling technology. An electronic throttle control system has been used to illustrate a number of aspects of the use of automatic code generation.

To provide engineering students with the skill set necessary to be successful in the field of embedded control systems, it is important to familiarize and educate them on the technology of automatic code generation. This has been illustrated to require a shift from detailed knowledge of algorithm design in C and C++ to knowledge at a more architectural level.

At the algorithmic level, required skills have shifted to being able to inspect code, understand optimizations, and being able to verify low-level fixed-point bit operations.

Finally, it has briefly been indicated how automatic code generation may assume a prominent position in teaching real-world phenomena in embedded control systems such as timing

constraints and noise effects, because real-world implementations can be obtained without the tedious manual coding process.

ACKNOWLEDGMENT

The author would like to acknowledge Mark Corless for his help in designing the electronic throttle control system.

REFERENCES

[1] Simulink, *Using Simuilnk,* The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA, 2004

[2] Mosterman, P. J., S. Prahbu and T. Erkkinen, " An industrial embedded control system design process", *Proceedings of The Inaugural CDEN Design Conference,* CD-ROM, July 2004

[3] MATLAB, *The Language of Technical Computing*, The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA, 2004

[4] Stateflow, *Stateflow User's Guide,* The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA, 2004