Model Based Software Development Process For Production Applications

Chandrashekar, MS
Delphi Electronics & Safety
Bangalore, India
ms.chandrashekar@delphi.com

Susan Dong
Delphi Electronics & Safety
Shanghai, China
susan.dong@delphi.com

Alwandi, Naveen
Delphi Electronics & Safety
Bangalore, India
naveen.alwandi@delphi.com

1. ABSTRACT

Over the past decade, the automotive industry has been increasingly adopting Model Based Software Development to reduce product development time and to improve the quality of the products.

To meet the requirement of production quality software, an established process is needed for the Model Based Software Development (MBSD). This paper describes the MBSD process that has been followed for real-world production applications. The main steps in the MBSD process are explained in detail. The similarities and differences between the MBSD process and the traditional software development process are also discussed. Further it provides a quantitative and qualitative comparison of the discussed process with respect to the traditional software development.

2. INTRODUCTION

In recent times, there have been increasing pressure on the automotive OEMs and suppliers to reduce product cost, improve product quality, and reduce time to market. As embedded software is a major part for most of the automotive applications, embedded software developers shares the same pressure. In the traditional embedded software development process, the scope of reducing the cost and time to market is limited as it invariably affects the quality of the product. With model-based approach, development time could be reduced considerably not only with no adverse effect on the quality of the product, but even improves the quality of the product. [1] This is the leading reason for the automotive industry to adopt the model based software development.

The benefits of model based approach such as inherent simulation ability and possibility to verify the design early has been discussed in several papers in various conferences[1][2]. The MBSD is still relatively new in the automotive industry. An established process is needed for developing production intent software. The process presented in this paper has been applied for practical production applications. The main steps of the autocoding and testing process are illustrated in Figure 1. Several of the main steps in the MBSD process are similar as the steps in the traditional software development process. For example, fixed-point math analysis step is applicable for both autocode and handcode if an integer microcontroller is used. But there are steps that are unique to MBSD process, such as Model Preparations.

In the MBSD process, software testing is closer to the system level because the requirement comes as executable models. The autocode component-testing step is much closer to a system level testing even though it is performed during software development. In this step, an application scenario is played back as test vectors on both the model and the autocode; the outputs from the model and the code are compared to identify errors in the generated code.

In the automotive industry, any product development will incur recurring cost and non–recurring cost. Recurring cost includes the manufacturing cost while the non-recurring cost includes the product development cost. Traditionally, the effort is made to reduce the recurring cost to increase the cost feasibility of the product. In this regard, integer micro-controllers are preferred instead of floating point micro-controllers. When we use integer micro-controllers, all computationally intensive control algorithms will have to be implemented in fixed-point autocode. The MBSD process described in this paper are valid for both fixed-point autocode and floating-point autocode, except that unit-testing step is not needed for floating-point autocode, and some of the steps are much simpler for floating-point autocode.

During our software development and testing for production applications, actual engineering metrics and microprocessor metrics for fixed-point autocoding were recorded and compared with the metrics for traditional software development process. The comparison results are summarized in the paper.

3. MODEL BASED SOFTWARE DEVELOPMENT

Like the conventional software development, Model based approach has four important steps:[2]

- Model Based Design
- Model Validation
- Implementation (Autocoding)
- Autocode Testing

3.1 Model Based Design

Model based software development starts with requirements being modeled to form the design. This design is basically graphical models like Simulink® / Stateflow® models incorporating process and control logic. This step involves analyzing the requirements to decide the input and output interfaces and eventually implementing the algorithm. Most importantly some constraints are imposed on modeling by way of style guidelines. This is done to make the models more consistent in style and structured and also to make the model suitable for autocode. Along with the model, the design also consists of a data dictionary containing the list of calibration and module's interfaces. Data dictionary provides the range and resolution of the calibrations and the interfaces, which are used during implementation.

3.2 Model Validation

In this step the models created above is validated against the requirements. This validation can be done on the host by way of simulation. The host validation can be done in an interactive way using tools like Altia® or GUIDE

More vigorous way of model validation is by using a rapid prototyping environment. For example, running and testing the models in the vehicle using an emulator like dSpace's Autobox.

3.3 Implementation

The validated model is then used to generate the C – code using autocoding tools like dSpace's Targetlink®, Mathworks's E-Coder. The first step in is to make the model compatible for autocoding. This can be partially taken care during modeling with proper style guidelines. Further, the design model is modified to satisfy the target controller constraints and operating system requirements. Software information is then entered into this model to form the autocode model. Finally, autocode is generated from this model. It must be noted that autocoding is not completely automatic coding. Rather it is a graphical way of coding as it involves the manual effort in software partitioning and entering appropriate software information to obtain optimized code.

3.4 Autocode Testing

Even though autocode is automatically generated, testing is still needed because there is software properties are manually entered. During the autocode testing, autocode is tested against the model. In model-based approach, errors in autocode due to wrong interpretation of logic tend to zero as the autocode is obtained directly from the requirement model. However, errors do exist in autocode and are mainly due to manual activities like range-resolution analysis and software partitioning. Errors in autocode will be typically overflows, underflows and resolution deficiencies. Autocode testing strategies are developed to detect the above errors. In short, the testing is done to ensure that the autocode behaves as the design expects it to do. [3]

Incorporation of fixed point implementation effects mostly the autocoding and the testing steps of the above described process. Hence, these two steps are dealt in considerable detail the following sections. The process described in this paper is based on our experiences using the combination of MATLAB/Simulink/Stateflow from The Mathworks Inc and TargetLink® from dSpace. However, the same result could be applied to the code, which is generated with E-coder from The Mathworks Inc instead.

4. PROCESS OF AUTOCODING

The main steps in autocoding and testing are shown in Figure 1. All these steps except Model Preparation and Autocode Component Testing are similar to those in traditional hand code process. Each of the steps will be described in details in the following sections and compared with hand code process.

4.1 Model Preparation

The very first specification model for autocoding, needs to be configured such that the execution order and rate of execution of each component are the same as the requirement for the integrated software. Usually a model scheduler is used in the model to simulate the operating system in the integrated software such that the calling order and execution rate are the same between the model and the integrated software. This task is only performed in the autocoding process but not in the hand code process because the requirement for autocoding comes in executable models whereas the requirements for hand code are mostly in non-executable text documents.

Initial versions of the model released for autocoding usually needs to be partitioned to generate production efficient code. The criteria for model partition are coherence of functionality, and microprocessor's constraint on ROM, RAM and throughput. For instance,

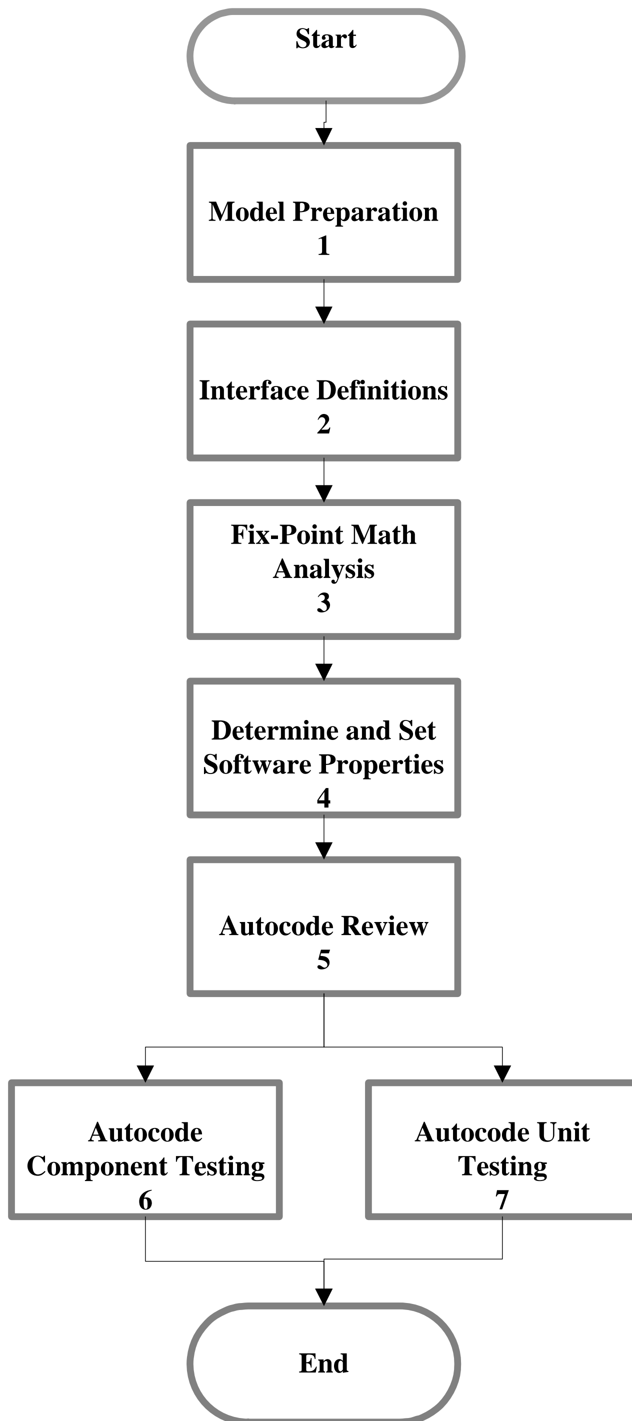

Figure 1: Process of Autocoding and Testing

if a subsystem is repeatedly used in the model, it should be made a user library block so that it can be made reusable C-functions, which will save ROM and autocoding effort. In the hand code process, this task is also performed except that it is performed during software design stage, but not as part of the structure of the requirement.

Every version of the system model for autocoding is checked and modified as per "Model Guidelines For Autocode". These guidelines are designed to ensure that the model is autocodable and will generate microprocessor resource efficient, consistent and robust code. We have developed in-house tools to automatically check some of the rules and generate a compliance report to the guidelines.

If the code generator tools used cannot generate code directly from the model, system models have to be converted to code generator models. For example, if the system model is developed with Simulink® / Stateflow®, and the code generator is TargetLink, the Simulink® / Stateflow® models will have to be converted into TargetLink models.

4.2 Interface Definition

For all the interface variables, a consistent approach similar to the hand code is followed for variable class, datatype and name. If the variable is an input to autocode modules, the software properties are set up such that the same variable class, datatype and name from the source module are referenced in the autocode module. If a variable is an output from an autocode module, the autocode module provides the declaration and definition of the variable, and the destination module should reference the same.

In Interface Definition step, the system range and resolution of each input, the ranges of interest for the outputs and required resolutions for the system outputs need to be gathered from system requirement outside of the model. This information will be used if fixed-point math analysis is needed.

4.2 Fixed-point Math Analysis

For some of the high-volume production application, it demands the use of low-cost, fixed-point micro controller units that contain extremely compact, fast, and traceable code. If math intensive algorithms have to be implemented for the fixed-point microcontroller, fixed-point analysis has to be performed to decide how many bits are needed to represent the range and how many are needed to represent the resolution of a variable. This analysis is needed regardless of whether the algorithm is implemented by an autocode-generation tool or hand-coded. The inputs for this analysis are the ranges and resolutions requirements of the systems interfaces, i.e. inputs and outputs. The analysis will consist of two parts. One part is range analysis, and the other part is resolution analysis.

Range analysis starts with the range requirements of the system inputs. Since fixed-point can only represent numbers with limited range, output range (maximum and

minimum values) of each math operation is calculated based on the maximum and minimum ranges of the inputs. The range calculation starts from the beginning of the dataflow in the model and then propagates the ranges to the end of the dataflow. The results of each range calculation will determine the number of bits needed for the integer part representation. These range calculations are performed for each math operation in the model. Autocode-generation tools, like TargetLink, or E-coder, assist in automation of this routine job. However, this process is not completely automated and requires significant time to execute.

Resolution analysis starts with the resolution requirements of the system outputs. This resolution prediction, similar to range estimation, has to be repeated for each math operation. For a generic two variable math function in equation *(1),* use Δx and Δy to represent the resolution of inputs x and y due to fixed-point representation, use Δz to represent the output change due to the input variation by the resolution amount Δx and Δy. In equation *(2),* Δx, Δy and Δz represent the exact changes in x, y and z for the function f. Δz can be estimated using Approximation by Differential Method in equation *(3)*. Assuming that the required resolution Δz for output z is either specified as system requirement or derived from the resolution requirement of the final system output, equations *(4)* and *(5)* can be used to predict what is the finest resolution needed for x and y in order to meet the required resolution Δz.

$$Z = f(x,y) \quad (1)$$

$$\Delta z = f(x+\Delta x, y+\Delta y) - f(x,y) \quad (2)$$

$$\Delta z \approx f_x'(x,y)\,\Delta x + f_y'(x,y)\,\Delta y \quad (3)$$

$$\Delta x \le \beta\,\Delta z \,/\, |\,f_x'(x,y)\,|_{max} \quad (4)$$

$$\Delta y \le (1-\beta)\,\Delta z \,/\, |\,f_y'(x,y)\,|_{max} \quad (5)$$

Where $\beta \le 1$.

For multiple variable functions, similar formulas can be derived also based on Approximation by Differential Method.

The resolution prediction starts from the end of the dataflow in the model, then propagate to the beginning of the data flow. The results of the resolution analysis determine how many bits are needed to represent the fractional part of a variable.

An example model to illustrate fixed-point math analysis is depicted in Figure 2. The inputs to the fixed-point analysis are ranges and resolutions of the interface variables as listed in Table 1. The results of the fixed-point math analysis are summarized in Table 2. For this example, the required resolutions for inputs *speed_r, distance_r,* and *accel_r* are all lower than the available system resolutions. Therefore, the required resolution for the final system output *delta_speed_squared* will be met if all the fixed-point data type are chosen based on the fixed-point analysis.

Resolution analysis could be very time consuming and complicated if there are many math operations along a data flow path. This is especially true if there are feedback loops or sophisticated filters. Sometimes, for complicated cases, resolutions are decided based on domain knowledge of the algorithm, then rely on testing to uncover errors that are caused by inadequate resolutions.

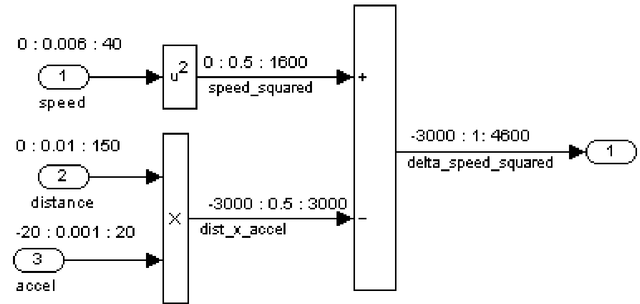

Figure 2: Example of Fixed-point Math Analysis

Table 1: Inputs To Fixed-point Math Analysis

Input, Output Variable	**Max, Min, Resolutions**
speed_max	40
speed_min	0
speed_r	0.006
distance_max	150
distance_min	0
distance_r	0.01
accel_max	20
accel_min	-20
accel_r	0.001
delta_speed_squared_r	1

Where x_r means resolution for variable x.

Table 2 : Results of Fixed-point Math Analysis

Calculation Formula	Range & Resolution
speed_squared_max = speed_max ^2	1600
speed_squared_min = speed_min^2	0
*dist_x_accel_max = distance_max * accel_min*	-3000
*dist_x_accel_min = distance_max * accel_max*	3000
speed_squared_r <= 0.5 delta_speed_squared_r	0.5
dist_x_accel_r <= 0.5 delta_speed_squared_r	0.5
*speed_r <= speed_squared_r/(2*speed_max)*	0.00625
distance_r <= 0.5 dist_x_accel_r/accel_max	0.0125
accel_r <= 0.5 dist_x_accel_r/distance_max	0.0016667

4.3 Determine and Set Software Property

The amount of software properties to be entered depends upon the type of Simulink block or objects of the Stateflow subsystem. For example, a Sum block requires three software properties to be set for code generation: the name of the block output variable, the class of the block output variable, and the data type of the block output variable. For example, the Gain block requires six properties to be set: the name of the block output, the class of the block output and the data type for both the gain output and the gain parameter. Figure 3 is a GUI for setting software properties for the "MUL" block in TargetLink, the code generator from dSPACE, Inc.

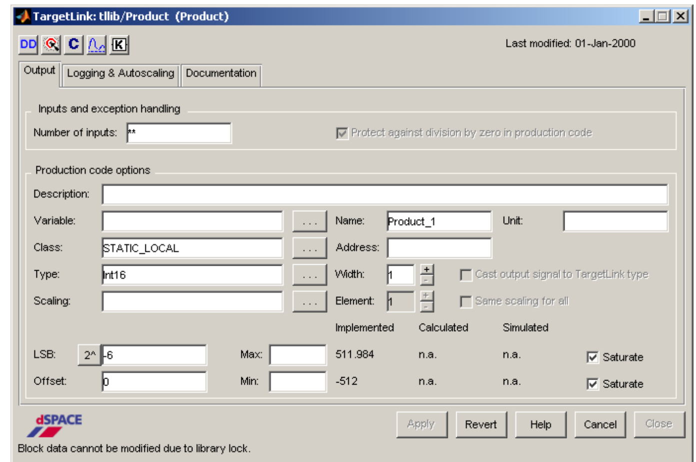

Figure 3: Software property Setting for MUL Block

To generate consistent and robust autocode, guidelines are developed for how to set software properties for input and output interfaces, math based blocks, lookup tables, and logic based blocks, constants and calibrations. For fixed-point autocode, data types are set based on the fixed-point analysis. Software properties are set to prevent overflow, underflow, and division by zero. Storage class specifiers such as automatic, static or external are determined and set for the module and function interfaces. The decisions of these properties are heavily microprocessor resource dependent.

4.4 Autocode Review

As discussed in previous sections, even in autocoding considerable amount of software properties is set manually. This is especially true for fixed-point autocode. Autocode review is to make sure that first, fix-point data types are sufficient to meet the range and resolution required; second, code is efficient in terms of microprocessor resources ROM, RAM, stack and throughput; third, interface software properties match the other modules that it interfaces with; and fourth, autocode properties are consistent.

5. PROCESS OF AUTOCODE TESTING

The MBSD process opens up new possibilities for software testing. The software specification now is in form of an executable model. The model itself is used to generate the expected outputs with given input test vectors. Then the autocode is excited with the same input test vectors and the autocode outputs are compared with the expected outputs. The differences between the two outputs will have to be less than the specified allowable delta. Otherwise root causes and resolutions would have to be identified to eliminate or reduce the allowable value. This process is illustrated in Figure 4.

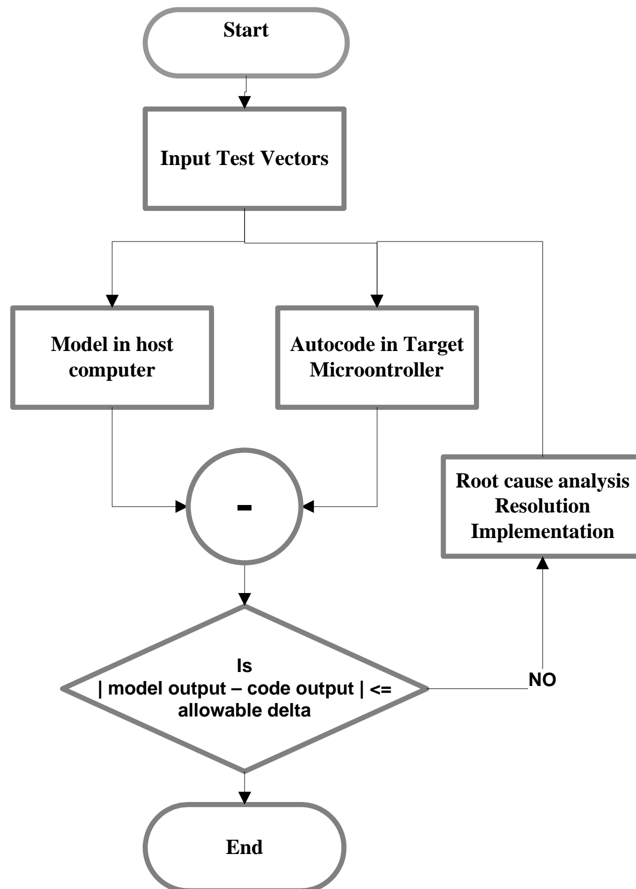

Figure 4: Autocode Testing

5.1 Autocode Component Testing

The Autocode Component Testing is done to identify differences between the component specification model and component autocode at the system level. Test vectors used for this testing are either collected during model testing or generated using a vector generation tool. Autocode is verified by comparing the outputs of the code in the target microprocessor against the floating-point model on a host computer with the same input test vectors fed to the model and autocode. This test is designed to capture errors due to wrong software properties, insufficient ranges and/or resolutions in the data types, and accumulative errors due to chains of math operations, filters and feedback loops.

5.2 Autocode Unit Testing

A unit in MBSD approach is visualized as a group of Simulink/Stateflow blocks which can be usefully tested. Our process involves identifying the unit at the lowest subsystem level with optimum number of blocks. The units are extracted from the specification model. Unit autocode is the code generated from the unit. Unit test plan is designed to verify that the unit autocode works correctly for worst-case scenarios interfaces and calibrations. Currently, the unit test plan creation is done manually.. These tests excite each math block in the unit to its extreme values. This is mainly to ensure that there will be no overflow or underflows of the fixed-point data types. The minimum and maximum output values are obtained by varying inputs and calibrations within their specification ranges or derived ranges. For logic intensive units, the input test vector profile needs to ensure complete MC/DC (Modified Condition / Decision Coverage)for the logical blocks present in the unit. Unit testing is only performed for fixed-point autocode because there is no need to check for overflow and / or underflow for floating-point autocode. [4]

6. COMPARISON WITH HANDCODING

In this section, MBSD approach with fixed-point implementation is compared with the existing hand code process. The comparison has been done under three categories: engineering effort, microprocessor resource and software quality. For the below comparison study, we have selected a mix of math intensive and control intensive modules.

6.1 Engineering Effort Comparison

For the implementation effort comparison we classified the modules as math intensive modules and logic intensive modules. Math intensive modules had a 90: 10 ratio of math versus logic while logic intensive modules had 10: 90 ratio of math versus logic.

Table 3: Effort Comparison

	Effort Reduction by MBSD	
Module Type	Implementation	Unit Testing
Math Intensive	51%	54%
Logic Intensive	49%	46%

Above Table 3 provides coding and unit testing effort comparison between hand code and autocode. The column "Effort Reduction by MBSD" gives the reduction in effort when MBSD approach is used instead of manual approach. In case of MBSD approach, the scope of wrong interpretation of logic, syntax errors is almost zero. However, even in MBSD approach fixed-point analysis, code optimization, software partition, interface definition has to be done manually. Hence there is considerable effort involved in implementation phase of MBSD approach. Our studies show the implementation effort is

around 50% less than that in traditional software development. Traditionally, hand code is generally subjected to unit testing at the module level. In MBSD approach, the expected outputs of the units under test are obtained using the model itself. Further, the input test case generation is also eased by the simulation ability of the model. Hence, the unit testing effort comparison provided by Table 1 shows a considerable reduction in effort for MBSD approach.

6.2 Critical Resource Comparison

RAM, ROM, stack and throughput are considered as critical resources for embedded systems. There has been a perception for few years now that autocode takes more memory and throughput compared to the hand code. However, over the years code generators have become more efficient such that autocode has comparable metrics even for fixed-point autocode. Our comparison studies in Table 4 were based on autocode generated for HC12 microcontroller and Cosmic complier combination.

Table 1: Critical Resource Comparison

	Ratio between Autocode and point hand code
ROM	1.00
RAM	1.03
Stack	1.09
Throughput	0.82

ROM for fixed point autocode is same as that of hand code while RAM and stack show a slight increase. In case of hand code, the fix math implementations are done using generic macros while in autocode these are done inline. So, the throughput is considerably less for autocode.

6.3 Software Quality Comparison

One of the ways for measuring software quality is using the number of peer review and testing findings. The findings are classified as functional and non-functional. Functional findings are the ones, which affect the functionality of the software while non-functional ones are those related to code efficiency, and code consistencies.

Table5: Defect Density Comparison

Defects For 1000 code lines	Reduction in errors in Autocode with respect to Hand code
Peer Review	23%
Unit Testing	42%

In Table 5 column 2 provides the percentage of reduction in functional findings in autocode with respect to hand code. It is observed that findings are considerably lower for autocode. Autocode is generated from pre–validated models and the models can be simulated on the host PC to understand the intended behavior. This substantiates the lower defect density for autocode both in peer review and unit testing. Further, majority of the code generators in the market are compliant to most of the safety standards as defined by MISRA. Hence compliance to standards is already built into the autocode.

Also, the autocode process discussed in earlier sections includes the additional “Component Testing”. This being a functional testing at the component level improves the quality to the autocode software.

7. CONCLUSION

A detailed Model Based Software Development process has been developed for real-world production programs. Several production programs have followed this process over the past few years. The same process has been used for both fixed-point and floating-point autocoding. Based on our experience as described in this paper, Model Based Software Development process scores higher than conventional hand-code process, with respect to development time and software quality. The engineering effort for fixed-point autocoding is around 50% less compared to conventional hand coding. The effort reduction should be even higher than 50% for floating-point autocode as the Fixed-Point Math Analysis and Unit Testing steps would be eliminated. MBSD provides the advantage of testing autocode with real-world scenarios, which improves the software quality considerably. During the recent few years, code generators have made significant progress in terms of generating microprocessor resource efficient code and safety standard compliant code. Based on our experience with HC12 microcontroller and Cosmic compliers, the ROM and RAM microprocessor resource usage for autocode is almost the same as handcode, and throughput is considerably less for autocode than handcode.

However, autocode development is far from push-a - button. As the code generator tools and MBSD methods are still evolving, there are areas for improvements. For instance, the manual review of autocode could be eliminated if the code generators were closer to 100% reliable. Hence the need of the hour is to have tools, which are more robust and reliable so that software engineering can derive the complete benefit of model-based approach.

8. REFERENCES

[1] "Incorporating Autocode Technology into Software Development Process", Lev Vitkin and T K Jestin, Delphi, ICSE Worshop on Software Engineering for Automotive Systems, Edinburgh, May 2004

[2] "Production software development process using modeling and auto-coding", B. C. Manjunath, Technical Center India, Delphi Electronics & Safety, 2004 Presentation in the dSpace User's conference. http://www.dspaceinc.com/ww/en/inc/company/events/usuc2004/production_software_develop.htm

[3] Scott Ranville and Dr. Paul E. Black, "Automated Testing Requirements - Automotive Perspective",

[4] http:// hissa.nist.gov/~black/Papers/autoTestReqs

[5] WAPATV.rtf

[6] "Unit Testing of Model based Embedded Software", Chandrashekar MS, Delphi, 3rd Matlab and Simulink Tech Expo

[7] The Mathworks, www.mathworks.com

[8] Reactive Systems, www.reactive-systems.com

[9] The dSpace, www.dspace.de