

Lean Model-Based Development: Practical Approach

Lev Vitkin

Delphi Electronics & Safety

ABSTRACT

Development pace of new embedded projects often requires usage of model-based design process (MBD). More individuals start using MBD without previous experience with tools and new processes. Matlab/Simulink/Stateflow is a common tool that is used in control applications in automotive and aerospace industries. Because of its complexity, the tool has a steep learning curve. Therefore, it is vitally important to set the MBD environment that allows persons to develop real-life projects even without a deep knowledge of the tool. The quality of the product should not be compromised and the development time should not be extended due to the initial lack of knowledge of the tool by the developers.

The shifting to MBD leads to changes of roles and responsibilities of algorithm designers and software implementers. This shift is due to ability of creating of efficient production code by code generators. Code generators empower algorithm developers to produce production code of almost the same quality as the software engineers used to create by hand. However, the generation of that software is possible in only a well-defined MBD environment. That environment is set by those with deep knowledge of the embedded software as well as modeling and autocoding tool.

This paper describes the components of the Simulink-based MBD environment that are proven to be effective in practice. Usage of the MBD environment streamlines the development and allows its automation. That, in turn, facilitates the execution of lean MBD development process, increase the robustness of the processes, and leads to a high quality final software.

CITATION: Vitkin, L., "Lean Model-Based Development: Practical Approach," *SAE Int. J. Passeng. Cars – Electron. Electr. Syst.* 6(2):2013, doi:10.4271/2013-01-0437.

INTRODUCTION

Modeling/Autocode technology, indisputably, has improved the technical abilities of systems and software engineers job performance. The use of modeling algorithm development process improves the way requirements are gathered and validated for different ideas of algorithm design. Code generators reduce time for coding or in many cases eliminate it altogether, thus allowing going from the model directly to production quality executable code. Process of creating of production-quality floating-point software using code generation might not require from the users having deep software expertise. Therefore, the autocoding can be done by the same control engineer who develops the model. Never the less, in many current production projects the autocoding process is still performed by the software group. In many cases the original model is converted to the "software model" by software engineers for efficient generation of the code. It creates potential errors due to miss-interpretation of the original model, and the models' maintenance becomes a nightmare. The fundamental paper [1] in a field of modeling and simulation defines Model-Based Design (MBD) and

specifies best practices in applying MBD. One of these practices addresses the issue of refocusing the role of software engineers from coding to architecting the software, thus leaving coding creation to an Autocode generation tool. Having said that, the software engineer shall actively participate in the development of the model. His involvement at that stage of development would lead to efficient code generation and comply with the software architecture of the product. Furthermore, in MBD development process the role of software/ software-tool group is extended to create and maintain an automated MBD environment [2]. This environment allows algorithm engineers to create a model that leads to a generation of production code software that is not different than the one generated by the software engineer. Software engineers are still responsible for software project architecture, low-level software, integration aspects of the software, and production releases. Software engineers don't need to re-develop the code from the given models. Instead, they closely monitor the generated code by algorithm developers and make needed adjustments in MBD environment. This environment makes the models compliant with industry and company standards and guidelines. It also

enforces standardized model architectures, pre-defines the interfaces and allocates all artifacts in accordance with project file partitioning and software architecture. MBD environment allows model development and production code generation by engineers who have limited initial knowledge in the modeling/autocode tool. The use of MBD environment addresses the issue with steep and time-consuming learning process of complex modeling tool like MATLAB/Simulink/Stateflow and different code generation tools. Below we will focus on practical implementation of the components of MBD environment for MATLAB/Simulink/Stateflow based applications.

The examples in this paper use terminology of Mathworks' Embedded Coder code generator. However, there is nothing in this paper that is specific to that code generator. The same examples can be easily re-created for dSPACE's TargetLink code generator.

The snippets of the Delphi-made tools are used in this paper for illustration purpose only. These examples show one of many ways of implementation of the MBD environment components described below. The purpose of this paper is neither to promote any particular tool nor to provide a detail of these tools.

COMPONENTS OF MBD ENVIRONMENT

We follow principals of lean development as they are described in [3]. Specifically, we focus on few key-stone principals: the way to eliminate waste by “hiding” via automation of all pre-set of non-essential information from the user; amplify learning of the tool by auto-setting or auto-corrections on the model per modeling Guidelines; enabling rapid-prototyping on the target with production-quality software. The principals are implemented in practice by automated MBD environment. This environment is common for MBD projects, but flexible enough to be tuned for the given project.

We assume that the end user of the environment is the algorithm engineer developing the individual features (component models) of the project as well as complete integrated model. The algorithm engineer creates the component-model, performs initial off-line simulation and functional testing of that model. The same engineer also generates production quality code and checks the behavior of the final product on a bench or a vehicle. The code, which is produced by the algorithm engineer for his/her component, should be used “as is” in the final production software. That assumption leads to define certain criteria for a lean MBD environment.

First, the MBD environment should be flexible enough to be applied to different types of model-based projects. Furthermore, MBD environment shall be re-configurable and easily adjustable to fit to specific project framework.

Second, the process of defining properties for variable and constants should be as simple as possible. All properties of

data, which are required by code generators, should be hidden from the algorithm developer. For fulfilling this purpose, a project-tuned Data Dictionary can be used.

Third, the MBD environment should enforce the usage of modeling guidelines, both industry-wide, like MAAB Guidelines [4] and company-specific guidelines. That includes incorporation of best practices for model development and code generation.

Forth, the MBD environment shall facilitate testing of the model in either/or MIL, SIL and PIL simulations.

Fifth, the MBD environment should be heavily automated. All tools, scripts, and libraries, which are part of development process for the given project, should be readily available for the user. This automation should facilitate the execution of every step of lean MBD process. This includes a push-button solution for generation of production quality code, at least for floating-point math based software.

Finally, the MBD environment shall be intuitive to the end user. This is critical for minimizing the learning curve for persons who are new to modeling and Autocode technologies.

Let's take a closer look at these criteria.

Project Configuration

Project settings always includes the description of project's files partitioning, the location of requirements, models, test vectors, legacy and autcoded software, setting model's Configuration Parameters file in accordance with project's models objectives, etc. We can call all these artifacts of the project as “project properties”. Having information about all these properties in one place is very convenient for the developers. For that purpose, we created ProjectConfiguration.xml file. User can access/edit this file via GUI. Fig.1 illustrates our current GUI for ProjectConfiguration.xml. Via this GUI, the user selects the ProjectConfiguration.xml file that is associated with the given project. It sets the path to the folders which are linked to the project, specifies project file partitioning, and specifies the software components (SWC) of the project. Also, it sets the path to model's Configuration Parameters file. In addition, the users assign pre-defined alias names to user-chosen names for project folders. The feature “alias” in Project Configuration is a very important one: it allows using multiple scripts for project automation regardless the names of user-chosen project folders. Let's take an example of a generated code. By default, this code is placed in the folder where the model is located or in one of its sub-folders. It's common that the user wants to place generated code in different folders. For example, a code might go to the Source folder; calibrations will go to another folder, EEPROM data in the third folder, etc. Many users have their own m-scripts that copy the products of code generation into project-specific folders. m-scripts cannot be reused if the user changes name and location of destination files. Addressing this issue, we limit the pool of names that the automation scripts can use. The user can take advantage of the re-usability scripts after he/she

makes the assignment of the user-selected names of the files to the pre-defined “aliases”.

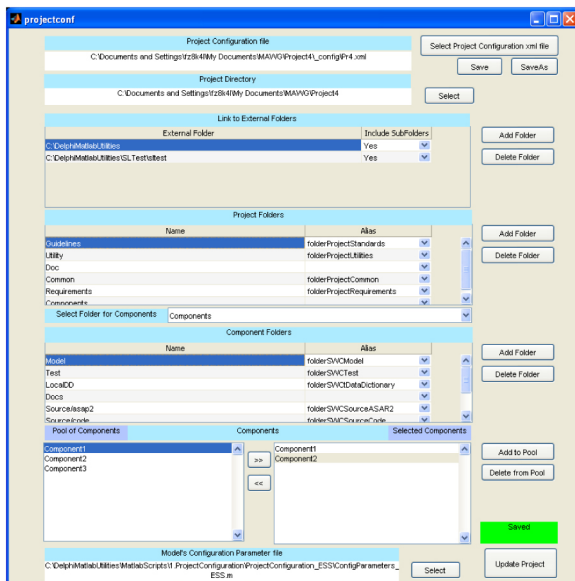


Figure 1. Example of a Project Configuration GUI.

All project-related settings are saved in a single <project>ProjectConfiguration.xml file. Upon invoking the project, that file is processed by a startup script and all data from ProjectConfiguration.xml file are available for multiple scripts and tools of MBD environment. The list of these scripts is long. Commonly used scripts are: the script which is responsible for project file partitioning, adding/deleting the folders and subfolders of the project; the script that sets Matlab paths for the current Matlab session; the script that invokes model's Configuration Parameters .m (or.mat) file; the script that invokes the compiler and builds final executable file using the list of the SWCs to be compiled and build, etc. All information that is needed for MBD scripts and tools is in a single ProjectConfiguration.xml file. That approach helps manage the project and makes easy to switch between the projects. Opening a different project means running different ProjectConfiguration.xml. And, therefore, the MBD environment is tuned toward the new active project with its own folders partitioning, links and Matlab paths. As a result, the developer doesn't waste time searching for the location of the correct scripts or use a wrong data setting in a Workspace that was placed there by previous project.

Data Dictionary

The use of Data Dictionary (DD) significantly simplifies process of defining properties of variables and constants that are required for code generation stage. The objective and criteria for DD is described in [5]. Different companies and engineering groups use all-features DDs like [5,6] or use simplified versions, one of them is shown in Fig.2 The main purpose of DD's GUI is to minimize the amount of information that the user should specify for variables and

constants based on project's specific criteria. The rest of information, which is needed for autocoding can be “pre-set” for the project and can be stored in proprietary DD files or, in our case, in xml file. That file is similar to the xml file used by Project Configuration file.

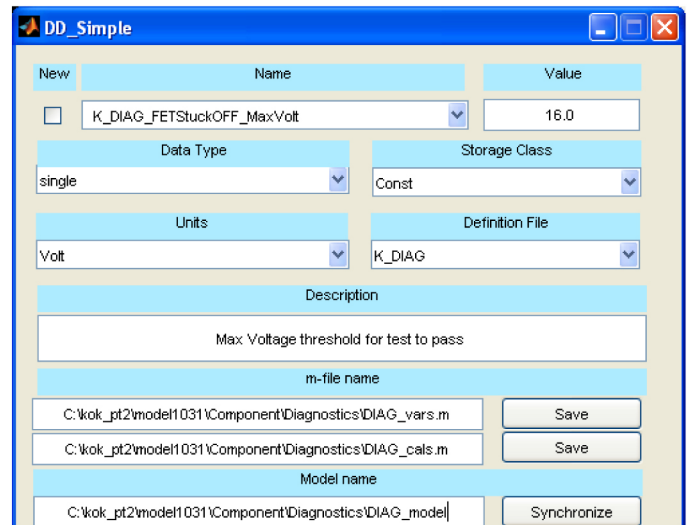


Figure 2. Example of a simplified GUI of Data Dictionary.

Let's take a look at a few different project software architectures. The first one requires that the global variables are defined in the same file that contains the c-code implementation of the model. In this case the Storage Class for model's topmost Output signals is set as Global (for Embedded Coder) with the file-owner name remaining the same as model name. The second architecture requires all global variables to be placed in a single project-wide data file. In this case the Storage Class is set as ExportToFile, or Global with the name of the file provided. The third example requires all calibration to be placed in the same file as the models. Each of the calibrations shall be supplied with #pragma statements, so the calibration is placed in the dedicated memory by compiler during build process. In this case the Custom Storage Class shall be introduced and calibrations shall be assigned to that class. Different Storage Classes lead to generation of different code. But all Classes mentioned above are irrelevant to the model developers. Thus, the use of DD addresses several major components of the lean MBD process. Once DD is configured for the project, it hides all irrelevant information from the algorithm developer, and, therefore, trims time for setting of non-value added parameters. Use of DD also simplifies setting of data properties. It therefore, assures the correct setting of data properties by the developers with limited knowledge of Autocode and Autocode-required data settings. At the same time, use of DD leads to consistency in generated code due to consistency of setting model interfaces and data. It is always a possibility that the user sets data properties bypassing DD, strait in ModelExplorer. That is why Synchronization feature