

Automatic C to Simulink Model Converter (C2M) Tool

Smitha Kizhakkai Palakkal, Priti Ranadive, and Naveen Boggarapu
CREST, KPIT Technologies, Ltd.

Rakesh Rao
KPIT Technologies, Ltd.

Pallavi Kalyanasundaram
K.K.W.I.E.E.R, Nasik

ABSTRACT

The automotive industry today follows Model Based Development (MBD) for developing modern automotive applications. This method involves creating models for a system under design and then using tools like MATLAB/Simulink to auto-generate code for target platforms. This method is popular since maintenance of MBD based applications is simple and less time consuming as compared to maintaining hand-written application code. Thus, MBD facilitates correct designs and easy maintenance of automotive applications. However, there are legacy automotive applications that are not developed using models. It is difficult to accommodate and test any changes in such application codes since it requires extensive testing. Additionally, for application code generated from models, many a times, code is changed during testing and these changes are not reflected in the model. Hence, there is a need to convert legacy automotive application codes to models. A novel Code to Model (C2M) tool is described in the paper that can automatically convert a given code to Simulink model. We also discuss the design and few implementation challenges of this tool. The C2M tool has been tested with an automotive powertrain code. It has resulted in near 100% productivity improvement with respect to the conversion process while adhering to the MathWorks Automotive Advisory Board (MAAB) guidelines and has 95% accurate conversion rate.

CITATION: Kizhakkai Palakkal, S., Ranadive, P., Boggarapu, N., Rao, R. et al., "Automatic C to Simulink Model Converter (C2M) Tool," *SAE Int. J. Passeng. Cars – Electron. Electr. Syst.* 8(2):2015, doi:10.4271/2015-01-0164.

I. INTRODUCTION

In recent years, the design and development of automotive applications has changed from an electrical-mechanical engineering discipline to a software based electrical-mechanical engineering discipline. This new trend, known as Model Based Development, also popularly abbreviated as MBD, is a mathematical and visual method of addressing designing complex problems associated with control engineering, signal processing, and communication systems. It is used in the design and testing of many automotive applications. Hence, embedded software components are not written in programming languages like C, instead, tools like MATLAB / Simulink™ [1] are used to develop models of the system to be designed

Model based development allows the development of high-level models used for simulation in very early stages of the development process. This is important since automotive system development needs an interdisciplinary approach with software, electrical, and mechanical engineering aspects combined.

Graphical representation and simulation of such models allows engineers to find a common functional understanding early in the design phase. Thus, model-based development improves communication within development teams, with customers, or between car manufacturers and suppliers. It reduces time to market through component re-use and reduces costs by validating systems and software designs prior to implementation. MBD advantages include, using a common design environment across project teams, linking design directly with requirements, integrated testing with design to continuously identify and rectify errors, automatically generate embedded software code and reuse of designs to deploy systems across different hardware targets. One of the widely used tools for MBD is MATLAB/Simulink. Simulink tool allows the users to develop model and then can automatically generate code for target platform during deployment.

Many times developers come across a situation where they have to develop models based on existing legacy code modules and information provided in requirement documents [2]. There are also situations when developers have used tools like Simulink to automatically generate a code for target platform but later during testing, have made changes to specific modules in the code [3]. These

changes made in the code do not reflect in the original or reference models. In these situations, the process to integrate all required information and then generate a model becomes very complex and time-consuming [4]. Hence, there is a need to automate the process of code to model conversion. With this motivation, we have implemented an automatic code to model (C2M) conversion tool.

II. RELATED WORK

Many aerospace and automotive companies have invested in the past to convert code modules to models. While the basic approach remains the same, many authors have suggested key considerations for the migration process. For example, the authors of [2] discuss the importance of efficient and reliable conversion and how a structured conversion process helps. They also discuss the criteria to measure the quality of the model created after the conversion. The same key considerations are iterated in [3] along with the fact that the created model is a true representation of the code. Both [2] and [3] are based on identifying the architecture of the source code to be viewed as a complete system. They further mention considering partitioning the code into subsystems and then further partition subsystems to atomic functional units. This approach helps in unit testing by highlighting the structural dependencies in the code.

Another paper [5], proposes a framework to bring out the importance of integrating automotive control system development with engineering roles. All the above-mentioned papers address the model creation or conversion process related to optimization. However, they still do not consider a fully automated approach of converting legacy applications to executable models. Such an automatic code to model conversion tool was proposed in [6]. However, the paper does not address many aspects of automotive MBD process, for example compliance to MAAB guidelines [7]. The translation tool mentioned in [6] also does not handle functions that have pointer type arguments or that are recursive. Additionally, the tool [6] does not consider generating a data flow graph (DFG) that is an important consideration to address some of the timing and functional correctness aspects of a generated model.

Unlike the tool mentioned in [6], the C2M tool handles all the above types of C constructs apart from the regular constructs. Additionally, our tool also generates models that are MAAB compliant and integrates the Control Flow Graph (CFG) and Data Flow Graph (DFG) to generate functionally correct executable models.

III. THE C2M TOOL OVERVIEW

The C2M tool is a static analysis based tool, that converts a given C source code to a MATLAB script (.m file). This script can be executed in MATLAB to create a Simulink model file (.mdl file). The entire conversion process is automated and hence users of this tool will be able to improve productivity. [Figure 1](#) shows the overview of the C2M tool.

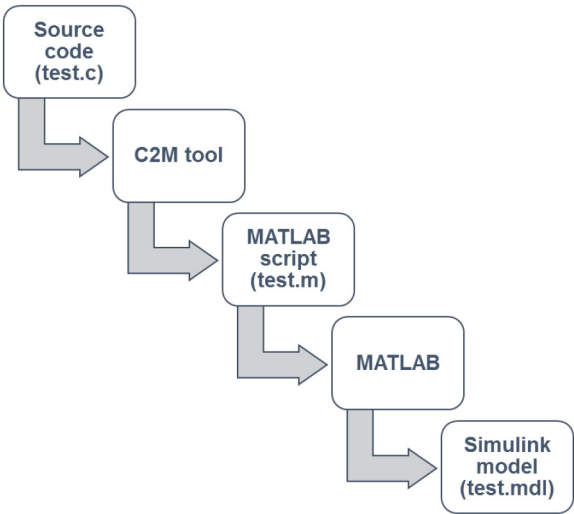


Figure 1. Overview of C2M tool

[Figure 2](#) shows the block diagram of the tool. The tool has a frontend module that includes a parser, scanner and Extensible Markup Language (XML) file generator modules. The backend module of the tool includes the control flow and data flow graphs creation modules and a MATLAB script generator module.

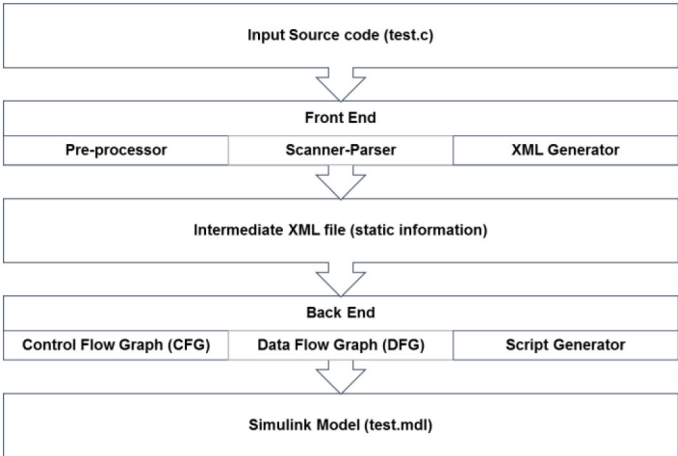


Figure 2. Block diagram of C2M tool

Pre-processor does the preprocessing of all the C files that are given to the tool as input. The pre-processor replaces constants, which are present in terms of pragmas, and resolves the dependencies in header files. It then formats the sequential C code as required by the scanner. Output of the pre-processor will be a C file (that has C code without the header file inclusion, macros, and comments) and a header file that has all the information about the included header files.

C2M tool uses an open source C Pre-Processor viz. MCPP [8] to perform the pre-processing steps. Few modifications were made to the original MCPP, in order to accomplish required functionality. The scanner forms lexical tokens [9] of the source code, and

writes it in to corresponding token files, which are the input for the Parser. The parser analyzes these tokens as per the American National Standards Institute (ANSI) C [10] [11] grammar and generates an intermediate file as output. This intermediate file (XML file) contains all the code information that is subsequently used by different back end modules. The intermediate file (XML file) stores metadata about program constructs that are present in the sequential C programs. The metadata stored in the intermediate XML file, for example, for a variable includes its definition, data type and scope, files in which the variable is declared, etc. Similarly, metadata stored for a function would include its definition, return types, caller (other functions that call the function under consideration) and callee (other functions that are called from the function under consideration) function lists, parameters, arguments, and so on. Similar information is stored for iterative, control and selection statements as well. The C2M tool uses the Minimal XML (MXML) [12] library to manage the intermediate XML file data. The MXML library consists of useful functions to perform various operations on xml file like read the file, write to the file, create a node, add / delete attributes to the node and delete the node.

The scanner-parser module of C2M tool is designed to handle most of the complex constructs / expressions present in an application. The tool performs lexical analysis of the entire code and information is stored in an intermediate file (Token File). The output of this analysis goes to parser module and the parser module considers various programming styles. The parser follows a recursive-descend strategy. The tool generates a symbol table similar to syntax analyzer in a compiler. C2M parser stores information in an intermediate file format (XML) and holds more details than Abstract Syntax Tree (AST), which is created by most of the compilers. For example, C2M tool stores information about variable line numbers along with the data type, scope etc.

An example of data contained in the XML file is as shown in [figure 3](#). The contents can be explained as follows: Values that are mentioned in double quotes viz., EX, SX, VA, FL etc. are unique identification numbers for expressions, sub-expressions, variables, files respectively. In [figure 3](#), EX98, FL1, IF97, SX101, SX103, VA100 etc. are unique Id's given by the scanner. The XML code snippet is equivalent to the C expression below:

```

if ( flag_var4 && ( (! flag_var5) || cond_var1 ) && ( cond_var2
>= cond_var3 ) && ( flag_var3 != 0 ) )
{
local_uchar1      =      glob_var4      >=      local_uchar3;
local_uchar4      =      glob_var4      <=      local_uchar5;
}

```

```
- <var_info id="VA76" name="flag_var3" file_id="FL19" parent_id="NULL"
  level_of_indr="0" num_lines="3" is_fun_pt="0">
- <var_info id="VA99" name="flag_var4" file_id="FL19" parent_id="NULL"
  level_of_indr="0" num_lines="1" is_fun_pt="0">
- <var_info id="VA100" name="flag_var5" file_id="FL19" parent_id="NULL"
  level_of_indr="0" num_lines="1" is_fun_pt="0">
- <var_info id="VA102" name="cond_var1" file_id="FL19" parent_id="NULL"
  decl_line_no="12" level_of_indr="0" num_lines="1" is_fun_pt="0">
- <var_info id="VA105" name="cond_var2" file_id="FL19" parent_id="NULL"
  decl_line_no="12" level_of_indr="0" num_lines="1" is_fun_pt="0">
- <var_info id="VA106" name="cond_var3" file_id="FL19" parent_id="NULL"
  decl_line_no="12" level_of_indr="0" num_lines="1" is_fun_pt="0">
- <var_info id="VA112" name="local_uchar1" file_id="FL19" parent_id="NULL"
  level_of_indr="0" num_lines="1" is_fun_pt="0">
- <var_info id="VA113" name="local_uchar3" file_id="FL19" parent_id="NULL"
  level_of_indr="0" num_lines="1" is_fun_pt="0">
- <var_info id="VA117" name="local_uchar4" file_id="FL19" parent_id="NULL"
  level_of_indr="0" num_lines="1" is_fun_pt="0">
- <var_info id="VA118" name="local_uchar5" file_id="FL19" parent_id="NULL"
  level_of_indr="0" num_lines="1" is_fun_pt="0">
- <exp_info id="EX98" file_id="FL1" parent_id="IF97" line_no="32" is_ass_exp="0">
  <sub_exp_info id="SX101" LHS="NULL" RHS="VA100" OPR="TOK_NEGATION" />
  <sub_exp_info id="SX103" LHS="SX101" RHS="VA102" OPR="TOK_OR_OP" />
  <sub_exp_info id="SX104" LHS="VA99" RHS="SX103" OPR="TOK_AND_OP" />
  <sub_exp_info id="SX107" LHS="VA105" RHS="VA106" OPR="TOK_RIGHT_EQUAL" />
  <sub_exp_info id="SX108" LHS="VA104" RHS="SX107" OPR="TOK_AND_OP" />
  <sub_exp_info id="SX109" LHS="SX106" RHS="NUMCONST 0" OPR="TOK_NOT_EQUAL" />
  <sub_exp_info id="SX110" LHS="SX108" RHS="SX109" OPR="TOK_AND_OP" />
</exp_info>
- <exp_info id="EX111" file_id="FL1" parent_id="IF97" line_no="33" is_ass_exp="1">
  <sub_exp_info id="SX114" LHS="VA61" RHS="VA113" OPR="TOK_RIGHT_EQUAL" />
  <sub_exp_info id="SX115" LHS="VA112" RHS="SX114" OPR="TOK_EQUAL" />
</exp_info>
- <exp_info id="EX116" file_id="FL1" parent_id="IF97" line_no="34" is_ass_exp="1">
  <sub_exp_info id="SX119" LHS="VA61" RHS="VA118" OPR="TOK_LEFT_EQUAL" />
  <sub_exp_info id="SX120" LHS="VA117" RHS="SX119" OPR="TOK_EQUAL" />
</exp_info>
```

Figure 3. Example of information stored in XML file

For example, the expression “EX98” corresponds to condition expression of if clause in sample code. “VA100” corresponds to ‘flag_var5’ and the sub-expression “SX101” corresponds to ‘! flag_var5’.

The static information in the XML file serves as the input for the C2M backend module. The backend module consists of the data and control flow graphs (DFG and CFG) generator modules. Detailed description of the DFG and CFG are given in [section IV](#). By using the DFG and CFG we can convert or represent any given source code in terms of a Simulink sub-system. For example, consider the following control statement.

```
if (A==10) {
    B = 20;
    C = 30;
}
```

The above source code can be represented with the following Simulink model, shown in Figure 4.

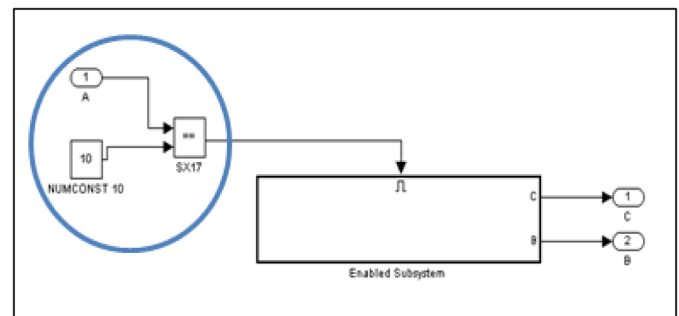


Figure 4. Model generated for the example code