

Design Verification of Automotive Controller Models

A. C. Rajeev
MathWorks India Pvt. Ltd.

Swarup Mohalik
Intel Technology India Pvt. Ltd.

S. Ramesh
General Motors Company

ABSTRACT

Model-Based Development processes in the automotive industry typically use high-level modeling languages to build the reference models of embedded controllers. One can use formal verification tools to exhaustively verify these design models against their requirements, ensuring high quality models and a reduction in the cost and effort of functional testing. However, there is a gap, in terms of processes and tools, between the informal requirements and the formal specifications required by the verification tools. In this paper, we propose an approach that tries to bridge this gap by (i) identifying the verifiable requirements through a categorization process, (ii) providing a set of templates to easily express the verifiable requirements, and (iii) generating monitors that can be used as specifications in design verification tools. We demonstrate our approach using the Simulink Design Verifier tool for design verification of Simulink/Stateflow models.

CITATION: Rajeev, A., Mohalik, S. and Ramesh, S., "Design Verification of Automotive Controller Models," *SAE Int. J. Passeng. Cars – Electron. Electr. Syst.* 6(2):2013, doi:10.4271/2013-01-0428.

INTRODUCTION

Model-Based Development (MBD) is a popular methodology for dealing with the complexities of embedded software development in automotive and aerospace industries. In MBD, first a high-level model of the software is developed (using a domain-specific modeling language) from requirements. The C implementation of the software is then auto-generated from this model, using code generators. This workflow allows the engineers to focus on high-level designs rather than low-level programming details.

A variety of verification and validation (V&V) activities may be carried out at different stages of software development cycle. These include requirements analysis, requirements based testing, model based testing, etc. These activities aim to identify as many bugs as possible early in the development cycle, as the cost of fixing a bug increases exponentially towards the latter part of the cycle.

A very useful V&V technique applied to design models is *design verification*, which formally verifies if a given model satisfies a given property. As shown in Figure 1, a design verification tool takes a model and a property as inputs, and checks whether the model satisfies the property on *all*

possible input sequences. If a property violation is detected, the tool generates an input sequence (called counter-example) that can be used to simulate the model for demonstrating the violation. These tools typically use exhaustive mathematical analysis to identify property violations. Hence, the obtained results are comprehensive unlike traditional testing based tools.

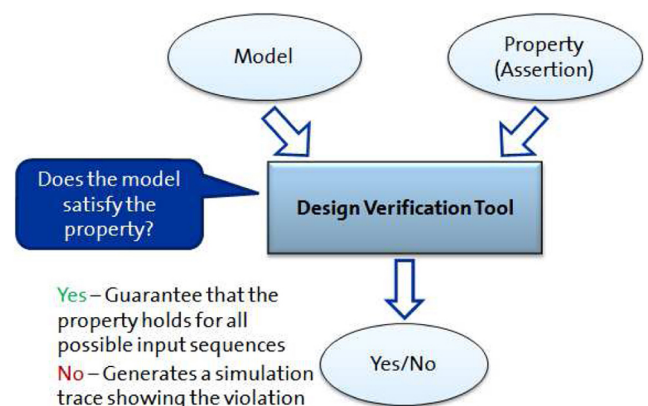


Figure 1. Design verification

Unfortunately, though a number of commercial design verification tools are available, their usage is not fully automated; a certain amount of manual preprocessing is necessary to use them. For instance, many of the tools support only a subset of the constructs available in the modeling language. Hence, a model containing unsupported constructs has to be made compatible with the tool by replacing such constructs with equivalent (or nearly equivalent) supported constructs. Another step is the identification of properties to be verified and their specification in a format that the tool can understand. In fact, the success of design verification relies significantly on this step.

In this paper, we present a systematic approach to reduce the chances of manual error while identifying and formally specifying the properties. We demonstrate this approach using the Simulink Design Verifier [1] tool for Simulink/Stateflow [2,3] models.

The rest of this paper is organized as follows. Section 2 presents the proposed approach, and Section 3 discusses its application to Simulink Design Verifier. Section 4 gives a brief review of the related work. Finally, Section 5 concludes the paper.

IDENTIFICATION AND FORMAL SPECIFICATION OF PROPERTIES

Design verification tools are capable of checking a wide range of properties. These properties can be broadly classified into two sets:

1. Model independent. These are generic properties that can be checked on any model. Examples in this class include (i) identifying the unreachable parts (similar to dead code) of the model, (ii) computing the lower and upper bounds on the values of the variables in the model, (iii) computing the possible values of the control inputs of Multiport Switch blocks in the model, etc.

2. Model dependent. These are properties that are specific to each model. They can be *functional* or *temporal*, and are typically based on the requirements of the system being modeled. An example functional property is *Pump is turned ON when the command becomes TRUE*, and an example temporal property is *Pump remains ON for at least 30s before going to OFF* [4]. Some model dependent properties represent *design decisions* taken by the design engineer. For instance, the design engineer might add a transient state (i.e., a state that is active for only a very short duration) between two states representing two different modes of a feature. This is to add a reasonable delay while switching from one mode to the other. During design verification, we would like to ensure that these additional states are really transient (i.e., they do not persist for more than a short duration).

We will now take a closer look at model dependent properties. As mentioned above, the majority of such

properties are obtained from system requirements. Requirements are usually written in an informal natural language like English, and have to be formalized before being verified on a model. However, all requirements cannot be formalized or verified on the model. In this context, we propose a categorization of requirements for the purpose of design verification:

1. Requirements that can be formalized and can be verified on the model. This category is the one that is useful in design verification.

2. Requirements that can be formalized but cannot be verified on the model. For instance, this may be a requirement that talks about the interaction between the given model and other models. The following requirement is an example: *When feature f is reset, the value of x shall be initialized as follows: if HMI is a dial then use the dial position, else if HMI has buttons then use the value in RAM if it is consistent, else use the calibrated default value.* Even though this requirement can be easily formalized, we cannot verify it on the model of feature f as HMI is not modeled as part of f.

3. Requirements that cannot be formalized but can be verified on the model. These include requirements that talk about particular implementation steps instead of the intention behind these steps. For example, consider the following requirement: *Command is computed using a lookup table CmdCmp_XY.* Since we are not informed about the function represented by the lookup table CmdCmp_XY, this requirement cannot be formalized. But we can verify by inspecting the model that the command is indeed computed using the suggested lookup table.

4. Requirements that cannot be formalized and cannot be verified on the model. This may be due to the requirement being vague or ambiguous. An example is the following requirement: *Feature f shall allow properly executed control transfer between modes M1 and M2.* Here, the term ‘properly executed’ is not well defined.

Note that the above categorization of requirements is on the lines of [5].

Given the requirements and the model of a system, the first step towards design verification is to categorize the requirements as discussed above. The next step is to formalize all requirements in Category-1. This is done by representing them in the specification format of the particular design verification tool being used. If the specification format is complex, this step can be tiresome and error prone - the specified property may end up being different from the intended one. Hence, a simple specification format is to be preferred over a complex one [4].

Our approach to handle this issue is to provide a set of easy-to-use templates (similar in spirit to the patterns provided by Embedded Validator [6] design verification tool) for specifying properties. The properties thus specified are automatically parsed and translated into the format required