

Ford Motor Company  
General Motors Corporation  
Motorola Automotive and Industrial Electronics Group

---

# *SmartVehicle* Challenge Problems

---

Embedded Software Analysis and Generation

---

In support of the University California Open  
Experimental Platform for DARPA – MoBIES  
DARPA Contract F33615-01-C-1841.

---

# Embedded Software Analysis and Generation Challenge Problems

This document presents the challenge problems in the area of embedded software analysis and generation for the *SmartVehicle* Open Experimental Platform (OEP) provided by The University of California at Berkeley. The challenge problems are divided into 4 sub-areas presented in order of importance in the subsequent sections. The challenge problems are not specified to the greatest level of detail to allow the MoBIES Phase I developers more flexibility to apply their own frameworks and methodologies where possible. The challenge problems represent Ford's immediate needs in the area of embedded software development. All Phase I developers are welcome to propose and solve related problems in addition to these challenge problems. Although the problems are stated within the context of MATLAB®/Simulink®/Stateflow® modeling environment, Ford remains open to exploring the solutions in other modeling domains as well.

## 1 Automated Unit Test Vector Generation

### 1.1 Motivation

The code that is placed in a vehicle needs to be of high quality. This is becoming increasingly more important as new technologies are making the software more complex. Traditional methods for testing the software, however, are labor intensive, sensitive to the person doing the work, often not repeatable, and usually ad hoc.

Recent advances in *computer-aided control system design (CACSD)* tools such as MATLAB®/Simulink®/Stateflow® have allowed control algorithms to be developed rapidly as simulation models. The use of CACSD tools offers many benefits. First, the control algorithm models can be used as precise specifications for the embedded system code [1]. Second, the use of these models offers a very good opportunity for automating the generation of unit test vectors [2].

With always decreasing resources (time and people), what is needed is a tool that can quickly generate high quality test inputs in a repeatable manner based on the CACSD model. These test cases then need to be executed on the model and the code and the results compared. This should prove to a high degree of confidence that the code, which could be hand written or generated from an automatic code generation tool, matches the behaviors specified in the model. This will in turn enhance the quality of the code by eliminating implementation errors, shortening the overall development time, and lowering overall costs.

## 1.2 Test Vector Generation Challenge Problems

For all of the following problems, the input test vectors must be generated for the overall model even when the test target is only a single block in the *SmartVehicle* model. Furthermore, the input test vectors must be generated starting from time  $t = 0$  for the original initial conditions of the model, i.e. no arbitrary reset of initial conditions is allowed. The simulation may be terminated as soon as the desired test criterion is achieved. The set of generated test vectors should also be minimal to minimize the number of tests that the reviewer must perform with these test vectors.

### 1.2.1 State Machine Coverage

For each Stateflow® block in the *SmartVehicle* model, generate a minimal set of test vectors to achieve the followings.

- Each state is entered at least once.
- Each state is exited at least once (when possible).
- Each transition is evaluated at least once.
- Each condition is evaluated to true and false at least once for each truth value.
- Each transition is taken at least once.
- A user specified path (sequence of transitions) is taken.

### 1.2.2 Discrete Logic Coverage

For each discrete logic block in the *SmartVehicle* model, generate a minimal set of test vectors to achieve the following types of coverage:

- *Block coverage*: The block is executed at least once. Any output value is acceptable.
- *Decision coverage*: Each possible output value of the block is produced at least once
- *Modified condition decision coverage (MC/DC)*: Each possible output value of the block is produced at least once. Every combination of inputs that gives a different output value with the change of a single input is examined.
- Generate all possible input combinations.

The above types of coverage are listed in the increasing order of coverage level. Thus, if test vectors have already been generated for a higher coverage level, e.g. for all possible input combinations, then there is no need to generate any more test vectors for a lower coverage level. In many cases, generating test vectors for the complete coverage (last criterion in the above list) may be prohibitive due to limited computational resources. Note: the user should be able to select the coverage level for which to generate test vectors.

### 1.2.3 Look-up Table Analysis

For each look-up table or similar block in the *SmartVehicle* model, generate a minimal set of test vectors to yield the maximum, the minimum, and at least one interpolated output value.

### 1.2.4 Block I/O Stressing

For each block in the Simulink®/Stateflow® model, generate a minimal set of test vectors to exercise all combinations of extreme the block input and output values.

### 1.2.5 Signal Stressing

For an arbitrary signal (wire) in the *SmartVehicle* model, generate a set of minimal test vectors to yield the maximum, the minimum and a singularity (NaN or Inf) in the signal value. If a singularity exists, it should be detected.

## 2 Schedulability Analysis

### 2.1 Motivation

It is difficult to determine if an algorithm will execute within given time bounds prior to implementation. Developing test input vectors to stimulate the worse case or nominal path through the code is difficult and time consuming.

Tools are needed to analyze the model for the probable worst-case path (from a CPU usage point of view) and generate a test vector that will stimulate this path. A test vector that stimulates a nominal or a user defined path through the model would also be useful.

This analysis tool will help ensure more robust designs that work the first time. This should reduce time to market and lower overall costs.

### 2.2 Schedulability Analysis Challenge Problems

#### 2.2.1 Worst Case Execution Time (WCET) Analysis

In order to perform the schedulability analysis, it is important to first obtain the worst-case execution time (WCET) for each portion of the embedded system code on the target processor.

- For each controller block in the *SmartVehicle* model, generate vectors to yield the longest, the nominal, and any user selected path along with the associated execution time on the target processor for each case. By the longest path, we mean the path that will take the longest amount of time to execute.

Although the above problem may be placed under the test vector generation challenge problem, we choose to place it in this section as it is more closely related to the schedulability analysis of the embedded system code. It is also preferable to have a timing-accurate simulator for the target hardware so that the execution time associated with the above mentioned paths could be obtained without having to resort to the real

hardware, which may not be readily accessible to the engineer who is performing the analysis.

### 2.2.2 Schedulability Analysis

- Given a target software and hardware architecture, the worst-case execution time obtained in the previous step, the *real-time operating system (RTOS)* specification, and additional timing constraints, perform a schedulability analysis to determine whether the target code meets all the timing requirements.

The description of the hardware architecture should take into account such items as cache, register swapping when an RTOS is used, and the effects of off-chip ROM or RAM. The software architecture description should include such items as preemption of one task by a task of higher priority, resource locking, and different scheduling mechanisms.

### 2.2.3 Automatic Derivation of Software Architecture

It may be the case that the embedded system developer is primarily concerned with satisfying the embedded code timing requirements on the target hardware without worrying about the specific software architecture being used. In this case, it is preferable to leave the software architecture unspecified and let the schedulability analysis tool derive a feasible architecture automatically. The challenge problem for this section is stated below.

- Given a target hardware architecture, the worst-case execution time obtained in the previous step, the RTOS specification, and additional timing constraints, perform a schedulability analysis to determine whether there exists a software architecture for the generated target code that meets all the timing requirements and automatically derive and realize one such architecture.

## 3 Automatic Code Generation

### 3.1 Motivation

Once a model is developed, it must be implemented in C code before being compiled for use in the vehicle. Currently, this translation is largely a manual, time-consuming process which, for a number of reasons, is prone to human errors and inefficiencies.

The translation phase can be difficult because of the broad range of styles and constructs available in the modeling tool. Problems occur when the modeling tool uses a construct or concept that doesn't easily translate to one available in the C language (hierarchical state machines are a good example). As well, variations in the programming styles of individuals and a high turnover rate among coders make it more difficult for "lessons learned" to be consistently applied to the code generation process.

From this it can be seen that a tool to automatically generate code from a CACSD model is needed [3][5]. Such a tool will have many benefits including: (i) consistent look and efficiency; (ii) the fact that once implemented, lessons learned are never forgotten; (iii) shorter development time; and (iv) no misinterpretations of the model.

Although commercial automatic code generation tools such as the *Real-time Workshop (RTW)* and *TargetLink* for MATLAB®/Simulink®/Stateflow® are now available [4], we still seek improvements in terms of generated code size and execution time. When designing for cost-sensitive automotive applications, limited computing resources are a reality. Additionally, target processors such as the Motorola 68332 and PowerPC 505 that are common in automotive applications are given ever increasing task loads to realize the latest control features.

### 3.2 Automatic Code Generation Challenge Problems

To assist in the code generation process, we assume that extra definition identifiers, or “tags” are added to the model. These tags contain specifications for the software detail that the core MATLAB®/Simulink®/Stateflow® model does not normally support, for example: variable names, types, data scoping, etc. These software tags are annotations that get saved with the model, but are transparent during the simulation of the model. The automatic code generation challenge problem is stated below.

**Given a Simulink®/Stateflow® model that is annotated with software tags, generate C code that has the following features:**

- Strictly conforms to the ANSI C standard, including Appendix F of the ANSI/ISO C standard.
- Preserves the functionalities and semantics of the model.
- Is most efficient in terms of ROM and RAM usage and CPU time.
- Is understandable by the user and traceable between the generated code and the source MATLAB model.
- Is independent of hardware platform, RTOS, and low level I/O drivers.

**The code generator should also provide the following features.**

- The code generation process must be completely automatic; no extra user intervention should be required to generate the code with the above named features.
- All MATLAB blocks that are usable in a discrete time domain should be supported.
- Calls to user defined functions must be supported, both in Simulink® and Stateflow®. For example, blocks such as Lookup Tables must be mapped to user defined function calls. Mapping these blocks, as well as user-defined library blocks, to their corresponding functions can be handled using specialized association files.
- The code generation tool needs to have a high degree of configurability in terms of user control over:

**Variable names** – Identifiers for variables should be user-assignable and resolved by the code generator

**Variable types** - All variables should be able to reflect any storage type, type specifier, and/or type qualifier *and* have the capability to be aliased to user defined names. Common type 'char' in standard 'C' base types is not required. Because of memory resource limitations in the automotive industry, using a larger than needed type is not acceptable.

*i.e.1.* type specifier float; type qualifier const; storage class extern; typedef customvar extern const float.

*i.e.2.* "static const volatile IDENTIFIER\_1;" ⇔ "custom\_var IDENTIFIER\_1;"

*i.e.3.* typedef F32 float; typedef custom\_var static const volatile;

**Variable scopes** – Same-name identifiers must be correctly resolved to the appropriate file or function.

*i.e.* a variable name 'counter\_value' appears in both the state machine 'A1' and the 'B1' within a model. An identification should be added to the variable name as 'SM\_A1\_counter\_value' and 'SM\_B1\_counter\_value' to define as two variables.

**Variable initializations** – Initial values may be contained in the variable declaration or in an initialization function. Initial values contained in the CACSD model may be overwritten.

**Variable location** – The file and/or function where each variable is declared or defined must be specified.

**Variable size declaration** – Variables may be scalar, vector or array of specified size.

*i.e.* var1 [3] [3]; var2 [5];

**Function partitioning** – sub-systems within the model may be demarcated into blocks translatable by the auto-code tool.

**Function prototypes** - The format of the calling interface for partitioned functions may be specified.

**File partitioning** – Functional elements within the model may be composed separately into 'C' files during code generation.

**Support for user defined C structures** – The ability to define and access elements of variables formatted in arbitrary C structures format must be supported.

**Support for calls to/from Legacy Code** – Existing 'C' code including custom data structures, must be integratable into the automatically generated code. Calls to legacy (hand code) and automatically generated code must be supported. Custom structures may include diagnostic code or lookup tables.

Existing C code, with or without custom data structures for lookup tables and diagnostic codes (for example), must be completely integratable into the automatically generated code. This requires calls to existing, as well as to automatically generated code.

**User defined comments in code** – Comment blocks or annotations in models should be implemented as appropriately located comments in the generated code.

**Target Processor** – The automatic code generation should take advantage of target processor knowledge to optimize the generated 'C' code.

- In addition to the above features, fixed-point arithmetic effects like quantization errors, saturation or overflows need to be addressed. The tool should allow the configuration of any definable data type for both fixed and floating point. For fixed point, relevant data type details require setting binary point and range as well as being able to determine scaling factors and detecting over/under flow where it occurs.

### 3.3 Example Problems

In order to aid the Phase one participants we have developed two example problems from the larger Berkeley powertrain model. These problems are stand alone collections of models and data dictionaries which would be used to generate code. We will use these examples as part of our baseline report on code generation. These models are available on the Berkeley Automotive OEP web site. (<http://vehicle.me.berkeley.edu/mobies>)

## 4 Automatic Real-time Operating System Generation

### 4.1 Motivation

For many companies including Ford, it is not desirable to commit extensive resources to coding and maintaining a proprietary scheduler. Commercial schedulers, however, often provide more features than those that are generally needed in automotive applications. This results in a significant waste of ROM, RAM, and CPU resources, which are typically very limited. Thus, a scheduler that is custom built to yield the highest efficiency in ROM, RAM, and CPU usage for each application is needed. Such a scheduler will allow the developers to concentrate their efforts on the application without having to be overly concerned with limited computing resources.

### 4.2 RTOS Generation Challenge Problem

As the topic is still largely unexplored, we loosely state the RTOS generation challenge problem as follows:

- Given a target software and hardware architecture, the worst-case execution time for the embedded system code, and additional timing constraints, generate a

custom RTOS that enables the target code to meet all the timing requirements and is the most efficient in ROM, RAM, and CPU usage.

In a sense, the above problem is the same as the software architecture generation challenge problem if one considers the RTOS as part of the software architecture. In addition, the RTOS generator must satisfy the following requirements.

- The generated RTOS must be OSEK compliant.
- The RTOS generator should be able to handle both single processor and distributed or networked architecture.

### 4.3 Baseline Information

As a guideline for the development and evaluation of the RTOS generator, we provide the following baseline information.

A typical powertrain application consists of 10 tasks. Six tasks will be periodic with a minimum sampling period of 1 to 4 msec. Other tasks will usually be multiples of this rate. Four tasks will be triggered by an external interrupt. Conceptually, up to 100 tasks may be preferred; to fit within the resource constraints (ROM, RAM, and CPU), these tasks are combined to form the above 10 tasks.

Aside from establishing a periodic task and triggering a task from an interrupt service routine, the only other RTOS feature that is generally needed is an occasional resource locking. In a typical automotive application, there are no mail messages or other explicit inter-task communication. Global variables are used to save memory and CPU. Neither is there dynamic task creation. For some applications, everything is done on a single processor while for others, a distributed solution is preferred.

The RTOS budget varies with each application, but representative numbers are:

microprocessor : 683xx (total RAM = 2-7 K, total ROM = 256 K)  
RAM : 1 K

microprocessor : PowerPC 5xx (total RAM = 24-36 K, total ROM = 256-512 K)  
ROM : 19 K  
RAM : 6 K

The most efficient commercial RTOS that we found was the SSX5 from Realogy. This kernel is a single stack kernel that supports a preemptive multitasking, but does not allow a task to suspend itself. Compared to "standard" embedded RTOS, this system has fewer features, but is still sufficient for today's powertrain applications.

<b>68332</b>	<b>SSX5</b>
Code ROM size	~3 K
RAM for 10 tasks (worst case)	732 Bytes
RAM for 10 tasks (best case)	380 Bytes
CPU @ 1000 interrupts/sec.	7.46 %
CPU @ 1500 interrupts/sec.	10.35 %
<b>PowerPC 505</b>	<b>SSX5</b>
Code ROM size	~3 K

RAM for 10 tasks (worst case)	2752 Bytes
RAM for 10 tasks (best case)	1272 Bytes
CPU @ 1000 interrupts/sec.	8.78 %
CPU @ 1500 interrupts/sec.	12.26 %

CPU numbers are for a 16MHz clock, 4 msec tick, 250 periodic tasks per second and specific hardware configuration and compiler options.

A benchmark of 13 RTOSes for the 683xx and 10 RTOSes for the PowerPC 5xx has also been presented in the July issue of the Embedded Systems Programming Magazine [6].

## 5 References

- [1] Butts, K., Toeppe, S., Ranville, S., "Specification and Testing of Automotive Powertrain Control System Software using CACSD tools," Proceedings of the 17<sup>th</sup> AIAA/IEEE/SAE Digital Avionics System Conference 1998
- [2] Toeppe, S., Ranville, S., "Model Driven Automatic Unit Testing Technology: Tool Architecture Introduction and Overview," Proceedings of the 18<sup>th</sup> AIAA/IEEE/SAE Digital Avionics System Conference 1999
- [3] Toeppe, S., Ranville, S., Bostic, D., Rzeimen, K., "Automatic Code Generation Requirements For Production Automotive Powertrain Applications," IEEE International Symposium on Computer Aided Control System Design 1999
- [4] Toeppe, S., Ranville, S., Bostic, D., Wang, Y., "Practical Validation of Model Based Code Generation for Automotive Applications," Proceedings of the 18<sup>th</sup> AIAA/IEEE/SAE Digital Avionics System Conference 1999
- [5] Toeppe, S., Ranville, S., Bostic, D., "Automating Software Specification, Design and Synthesis for Computer Aided Control System Design Tools," Proceedings of the 19<sup>th</sup> AIAA/IEEE/SAE Digital Avionics System Conference 2000
- [6] Toeppe, S., Ranville, S., "Powertrain Controller RTOS Evaluation: CPU and RAM Resource Usage Modeling and Benchmarks," Embedded Systems Conference - Europe 1999 and Embedded Systems Programming Magazine July 2000