Shweta et al. International Journal of Recent Research Aspects ISSN: 2349-7688, Vol. 2, Issue 4, December 2015, pp.130-134

Study of Requirement of Automated Testing

Shweta[1], Nikita Nain[2]
[1]M. Tech student, IIET (Jind)
[2]H.O.D., IIET (Jind)

Abstract: **With the ever-increasing complexity of embedded software applications, and the emergence of more and more safety critical applications, thorough validation and verification of the code is needed. To address this need, many embedded software development groups are using models and doing upfront engineering before testing on the final product. Using the old style of testing late in the development cycle resulted in very long and expensive release cycles. Ford estimated that 60% of work tasks were to correct requirements or design defects that had been released to downstream developers. With today's increasing need to get to market quickly with a safe product, this old style of testing is not adequate. Ford also used randomly generated unit test vectors, due to the lack of a commercially available tool, which only had approximately 75% coverage. Because of the need for safe systems, this level of testing is insufficient.**
This paper presents requirements for model checking and unit test generation tools so that the tools are practical in a large production environment that is typical in the automotive industry.

Keywords: Software Testing

I. INTRODUCTION

The ever increasing complexity of embedded control algorithms, the need for shorter development cycles, and the need for high quality and safety critical systems have helped move the embedded software development community towards using graphical modeling and program specifications. This modeling allows for a well-defined algorithm from which verification and validation are practical as well as provides a mechanism for a high degree of automation.

Today's tools allow for a broad spectrum of uses for the models being developed. Some of these uses include: requirements capture, algorithm specification, algorithm validation and verification, documentation, automatic code generation, automatic unit test vector generation, hardware-in-the-loop testing, rapid prototype testing, and architecture specification.

One of the biggest remaining problems is making these tools practical for the "typical" engineer working in a production environment. Most of today's tools have been used very successfully by "high end" users, such as researchers and advance groups. These high end users are typically very motivated individuals with extensive training and ample time to learn the tools and experiment with them until they work. Unfortunately, the production engineers often have neither the training nor the time to experiment with the new modeling tools. These engineers need tools that are easy to learn, intuitive, and nearly push-button to use. Also, due to their overbooked workload, these engineers need analysis tools that can work on a single model file. They do not have the time to implement and double check the same algorithm in multiple tools.

Model checking is an emerging technology for analysis of model based software designs. Model checking can also be used to automatically generate test vectors. While unit test vectors can be generated using specialized algorithms, many of the emerging automated test vector generation tools use model checking technology. These tools negate the property of interest and present it to the model checker. The counter example returned is the desired test vector, which exactly exercises the property.

Currently, the standard practice in the automotive industry is to do a significant amount of in-vehicle testing but very little upfront testing. This is a very costly manner of conducting business, and the industry is trying to move towards a virtual environment in which most testing is done early in the development process. From the software testing point of view, the implication is that any testing is better than no testing. Thus, a tool that can help with any piece of automating the model checking or unit test vector generating would be useful. However, any testing that is done needs to be nearly push-button due to schedule constraints in the production environment. In other words, a highly automated tool which does part of the testing could potentially gain widespread use, whereas a partly automated tool that does everything many not get used at all.

The rest of this paper will describe the types of model checking and unit test vectors that are of interest to the automotive industry, provide a brief overview of some of the available tools for modeling, model checking, and

generation of unit test vectors, and describe an effort to make model checking and automated unit test vector generation practical for the automotive industry. We believe this also applies to related embedded industries such as aerospace, robotics, and medical devices.

II. TYPES OF DESIRED MODEL CHECKING AND UNIT TEST VECTORS

This section presents some model checks and types of unit test vectors that would be useful.

One particular challenge for the research community is that many of the models being made, especially for automotive power train applications, contain a mixture of control and data. The data consists of mathematical equations, which often have floating point variables. Most model checking tools cannot handle such data, since the state space is too large. Some of the emerging model checking tools are finding innovative techniques to deal with this large state space and produce results both in a timely manner and within the memory available on a standard PC. One alternative to completely exploring the entire state space is to use a form of depth-first search. A second alternative is to abstract floating point variables into a few boolean conditions, for instance, replacing $x > 4.2$ with a boolean xTooLarge.

Another challenge is that the models can be quite large. At Ford, a typical powertrain application may have 5,000-10,000 diagrams.Each diagram consists of a number of "basic" blocks such as gain and sum blocks. Depending on the item under test, the test tool may only need to deal with a small piece of the total application. Some of these pieces can be quite big as well. The test tools, while utilizing a minimum of time and computer memory, will need to analyze large models.

III. "PASSIVE" MODEL CHECKS

The goal of model checking is to check that the specification is sound. One set of checks that are important can be termed "passive" checks, that is, the tester does not need to specify anything beyond the original model. They are predefined and commonly agreed upon. Some of these checks include: all states reachable, no unnecessary states, no graphical dependencies, all outcomes accounted are for, no writes before a definition, no algebraic loops, and array indexes are all within bounds. In addition to helping validate the specification, passive checks may help the practical economics, too.

Automotive applications are extremely cost sensitive. As a result adding off-chip memory is only done in exceptional cases, usually requiring the approval of someone high in the management chain. The preference is for the entire program to reside on-chip. Even though current microprocessors have more memory than their predecessors, wasting code is very undesirable. In can force the use of more chips. Consequently, identifying and removing unreachable or unnecessary states increases the efficiency of the code implementation, especially when an automatic code generation tool is used.

Some tools, such as Matlab's®[1] Stateflow™, allow for the graphical position of model elements to determine how the model executes. This is inherently dangerous when the models are also used for documentation since apparently cosmetic changes in the layout may lead to subtle behavioral changes. This type of check should be optional as it may be acceptable and even needed by certain groups.

For consistency, all outcomes of an expression should be accounted for. For example, if a function can return three values, but the specification only checks for two or has an extra check for a fourth return value, an error should be flagged. Using the traditional data flow concepts, a write to a variable should not occur before that variable is defined. Also, two writes before a use may be flagged as a warning of possible suspicious behavior.

Most tools, especially those that provide an executable specification, will flag algebraic loops before running a simulation. For those tools that do not have this built in, the model checker should perform this check. Another safety check is to ensure that array indexes are within the bounds for the given variable.

IV. "ACTIVE" MODEL CHECKS

The next category of model checks can be termed "active" tests in that user input is required. For this case, an easy-to-use GUI is needed so the tester can input the checks in an intuitive, or at least easily learned, language. Most model checking tools, such as Z or SMV, require a very specialized input format, which is unfamiliar to the typical production engineer. Model checking is more traditionally used for these kind of active tests. For example, can states NorthSouthGreen and EastWestGreen ever be active at the same time, or can variable X ever increase more than five mi/hr in one step?

For practicality, a GUI should allow the tester to create the desired checks. These tests should be able to be saved to a file for future use.

V. NIT TEST VECTORS

One way to break down the problem of testing into manageable pieces is use different coverage levels for the test vectors. The most common coverage levels are statement coverage, decision coverage, MC/DC coverage, and some form of all paths. Another coverage

[1] Reference to specific products, brands, or firms is for information purposes only; no endorsement or recommendation by the National Institute of Standards and Technology, explicit or implicit, is intended.s

type is boundary values; see definition below. A tool is needed in which the tester can select the desired coverage level and for which a minimal set of tests should be generated to accomplish the selected coverage criteria. The different coverage levels and a minimal set of test vectors have the same end purpose: identify errors in a minimal fashion. Due to time constraints, if the test engineers get multiple tests that fail for the same reason, they will probably get frustrated and stop using the tool. Coverage levels allow the tester to progressively increase the thoroughness of the testing. Hopefully the less stringent coverage levels identify major bugs. Once those are fixed, the more thorough coverage levels will find the more subtle bugs. The more thorough coverage levels typically take longer for the tools to generate the test vectors. Therefore, by starting with the lower coverage levels, the more time consuming tests can be run fewer times, saving overall testing time. Also, producing a minimal set of test vectors for a given coverage reduces time to execute tests and analyze the results of each test.

For state machines, the standard coverage measures need some redefinition. For example, statement coverage can be redefined as touching every state or using every transition between the states. The mixing of data and control flow is particularly important for state machines, as many of the transitions depend on variables which are potentially calculated outside of the state machine.

VI. MODELING TOOLS

This section outlines some of the major modeling tools that are used in production environments. The goal is to provide a feel for the modeling tools with which analysis tools need to be compatible to be successful. As stated above, analysis tools that require a new model are less likely to be successful because replacing an existing modeling tool, especially in a big organization, is unlikely due to the large amounts of learning time, training costs, tool costs, process changes, and "customized" glue code support that was needed to get the original modeling tool used. In addition production engineers are typically overworked and do not want to take the time to learn a new tool, and new tools are not trusted until they have undergone a lengthy prove-out period. These constraints apply mostly to a large production organization. Typically, the smaller the organization and the more research focused the group is, the more open they are to new modeling tools.

Matlab® is becoming the de-facto standard for modeling control algorithms, especially within the automotive powertrain area. Many major automotive companies such as Ford, GM, and Toyota appear to be using Matlab® or moving towards using it. As a result, a number of automotive suppliers are also using Matlab®, and it also appears to have significant use within the aerospace industry.

MATRIXx had some pockets of use, but with the recent acquisition by The Mathworks™ of distribution rights to MATRIXx (posting on The Mathworks™ web page dated 2/20/2001), it appears that MATRIXx may be getting phased out and no longer used.

ETAS's ASCET-SD appears to have a following with the German automotive companies and some following in the U.S. as well.

Within automotive body electronics, anything that is not powertrain, I-Logix's Statemate appears to have some use.

Rational's Rose appears to be the most popular UML tool at this time. Although popular in the pure software development community, it does not appear to be mainstream in automotive, although they seem to be starting to use UML.

There are many other tools in use, but the above tools appear to be the more popular tools with Matlab® Simulink® and Stateflow™ being the most popular.

VII. MODEL CHECKING & AUTOMATED UNIT TEST VECTOR GENERATION TOOLS

This section presents some model checking and unit test generation tools. We provide a brief description of the tools along with any of the above modeling tools to which they connect. The purpose of this is to provide a brief overview of what exists today, to show how the tool companies are trying to make their tools applicable to wider audiences, and to give a feel for the vast panorama of tools that designers already have to deal with.

ADI, Applied Dynamics International, has a tool called AUTT that will produce test vectors from a BEACON specification. The literature states: that the tool will report coverage achieved, not achieved, possibly achieved but not easily proven, overflow and underflow information, and identify dead code; coverage measures include MC/DC (and this statement and decision coverage), boundary value, table, stub, mathematical stressing, inputs and output stressing. ADI also has a tool to convert from Matlab® Simulink® and Stateflow™ to BEACON. BEACON can also generate code.

T-VEC Technologies Incorporated has a tool called T-VEC that will produce test vectors from a T-VEC specification. The literature states that T-VEC: performs automated model analysis, test vector generation (satisfies MCDC), test coverage analysis, and test driver generation to eliminate many manual and error-prone activities involved in verification and testing. T-VEC Technologies also has a tool to convert from MATRIXx to T-VEC.

Siemens has a tool called VALID that will analyze a model for consistency and also produce test vectors. VALID is intended to model the coordination between components. Their claim is that while UML allows for this type of modeling with message passing between

state machines, VALID makes this type of modeling much easier. The tool claims to: check for deadlock, livelock, reachability, and controllability; produce test vectors for event and state coverage; generate production code; generate documentation; and generate a test harness and execute the tests. The tool also allows the user to specify properties to be checked in the model. An add-on has been developed so that VALID diagrams can be imported into Rational Rose.

IAR has a tool called visualSTATE® that will analyze a visualSTATE® model. The literature claims that the tool can: check that all transitions are reachable, what states can never be exited, conflicting behavior, only explicitly defined state transitions can take place; user supplied questions of the model; simulate the model; create a prototype for testing; generate code; measure the test coverage and profile the application; perform regression testing; and create documentation.

EDAptive Computing has a tool called VectorGen™ that will produce test vectors from a Rosetta specification. They have an add-on to the Mentor Graphics Renoir product that allows for a graphical way to simplify the creation of a Rosetta specification. EDAptive is also developing their own graphical interface to allow the creation of a Rosetta specification.

Reactive Systems has a tool called Reactis that will analyze the model and also produce test vectors. The literature claims that the tool can: check the model for undefined variables, type errors, missing cases, non determinism, dead code, and deadlock; check for user supplied questions of the model; generate test vectors for decision coverage, statement coverage, and MC/DC coverage; simulate the model; generate code; and generate custom run-time monitors from the model. The models can be developed with Reactive System's proprietary notation. Their literature also states that models (or sub-components of a model) can be developed in The Mathworks® Simulink® and Stateflow™, I-Logix StateMate, Teleogics SDL, and Rational's UML state machine notation.

ATTOL has some test tools and are developing a tool that will generate test vectors from both Simulink® and MATRIXx. According to the literature, the existing tools: measure and display the code coverage; automatically generates a test harness and provides an execution environment and reporting mechanism; and an integration and validation test platform for any message-based distributed systems, including OSEK.

I-Logix has a couple of tools called Statemate and Rhapsody. I-Logix has partnered with OFFIS Systems and Consulting GmbH, a spin-off company to OFFIS, to add model checking and automatic test generation capabilities to Statemate and Rhapsody.

NIST, the U.S. National Institute of Standards and Technology, has developed some test tools that produce test vectors from an SMV specification. NIST is also working with the below Simulink®/Stateflow™ Intermediate Representation project to allow their tools to work with a Matlab® specification.

Bruce Krogh, of Carnegie Mellon University, is applying model checking and generation of test vectors to Matlab® models. One tool, SF2SMV, converts Stateflow™ models to SMV so model checking can be applied. Another tool works directly with a Simulink® and Stateflow™ model to generate test vectors. Krogh is also working with the Simulink®/ Stateflow™ Intermediate Representation project.

Simulink®/Stateflow™ Intermediate Representation

Many analysis tool companies and researchers would like to have connectivity to Matlab® Simulink® and Stateflow™. In addition, designers cannot take the time to rewrite and revalidate specifications for every tool and keep the different specifications in agreement when they change. While The Mathworks™ has kept the tool very open, deciphering some of the semantic meaning of the models is not an easy task. Moreover some semantics, such as the order of evaluation dictated by the graphical layout, are implicit and hard to reconstruct. Thus, creating a translator from Matlab® to another tool is a very time and labor-intensive activity. A group of researchers and tool users realized this issue and have formed an informal consortium to address this.

The goal of this consortium is to define an intermediate representation (IR), create a working demonstration of converting from Matlab® to the IR, and to create an API that will easily allow a tool company to develop a trnslator from the IR to their tool.

The IR will have an additional feature that should expand the capability of Matlab®. Matlab® is a very good tool for doing controls work. However, it is not as good for tasks like software design and timing analysis. Matlab® has placeholders for user defined annotations, and the IR will use these annotations to add extra information that analysis tools need. The model creator can either enter these annotations directly, or one of the analysis tools can add the annotations to the model. An example of this is as follows: 1) a controls engineer creates a control algorithm, 2) a programmer adds software implementation information, such as variable names, types, scope, and file names, as annotations 3) an automatic code generation tool produces the production level code 4) a test vector tool produces test vectors to stress the code from a chronometrics point of view 5) a timing accurate simulator uses the code and the test vectors to produce timing values that get placed back into the model as annotations 6) a timing analysis tool uses the timing annotations to determine if the model is schedulable or not.

VIII. CONCLUSIONS

There is a significant need for more upfront engineering in today's embedded software design process. Within the

automotive area, very little upfront testing has been done. With the introduction of executable modeling tools such as MLUnit this upfront testing is more feasible. It is the job of the tool vendors to make this testing technology available and practical to the end user.

Due to the constraints placed on the production engineers, principally limited time, the test tools need to be: nearly push-button to use, intuitive to learn, and connect to the tools that they are already using. Some additional characteristics that will make the test tools practical include: identify any error just once, allow the user to select the coverage level of interest, minimize the time to check the model or generate the test vectors, use the language of the production engineer, run on a standard desktop PC, generate a sequence of test vectors for testing state machines, and handle large models and models that consist of both data and control flow with a large state space.

These challenges may seem daunting for the tool vendors, but providing a partial solution is better than no testing at all. And many tools are on the verge of being practical in today's production environment.

REFERENCES

1) Butts, K., et. al., "Automotive Powertrain Control Development Using CACSD", Perspectives in Control: New Concepts and Applications, Tariq Samad (ed.), IEEE Press, 1999.

2) Butts, K., Toeppe, S., Ranville, S., "Specification and Testing of Automotive Powertrain Control System Software using CACSD tools", 1998, Proceedings of the 17th AIAA/IEEE/SAE Digital Avionics System Conference

3) Toeppe, S., Ranville, S., "Model Driven Automatic Unit Testing Technology: Tool Architecture Introduction and Overview", 1999, Proceedings of the 18th AIAA/IEEE/SAE Digital Avionics System Conference

4) Toeppe, S., Ranville, S., "An Automated Inspection Tool For a Graphical Specification and Programming Language", 1999, Quality Week Conference

5) Toeppe, S., Ranville, S., Bostic, D., Rzeimen, K., "Automatic Code Generation Requirements For Production Automotive Powertrain Applications", 1999, IEEE International Symposium on Computer Aided Control System Design

6) Toeppe, S., Ranville, S., Bostic, D., Wang, C., "Practical Validation of Model Based Code Generation for Automotive Applications", 1999, Proceedings of the 18th AIAA/IEEE/SAE Digital Avionics System Conf.

7) Patel, S., Smith, P., Sun, W., Ramanan, R., Donald, H., Toeppe, S., Ranville, S., Bostic, D., Butts, K., "CACSD in Production Development: An Engine Control Case Study", 2000, Global Powertrain Conference

8) Toeppe, S., Ranville, S., Bostic, D., "Automating Software Specification, Design and Synthesis for Computer Aided Control System Design Tools", 2000, Proceedings of the 19th AIAA/IEEE/SAE Digital Avionics System Conf.

