Model Based Testing and Abstract Interpretation in the Railway Signaling Context

Model Based Testing and Abstract Interpretation in the Railway Signaling Context

Daniele Grasso, Alessandro Fantechi
Department of Computer Engineering
University of Florence
Florence, Italy
grasso.dan@gmail.com, fantechi@dsi.unifi.it

Alessio Ferrari, Carlo Becheri, Stefano Bacherini
General Electric Transportation Systems
Florence, Italy
alessio.ferrari@ge.com, carlo.becheri@ge.com
stefano.bacherini@ge.com

***Abstract*—This article presents the experience of a railway signaling manufacturer in introducing the technologies of model based testing and abstract interpretation as part of its development process. Preliminary results show the better performance of these techniques with respect to the previously employed structural coverage based testing.**

***Keywords*-abstract interpretation; model based testing; safety-critical; railway signaling;**

I. Introduction

General Electric Transportation Systems (GETS) develops embedded platforms for railway signaling systems. The safety-critical nature of these applications makes the verification and validation activities extremely crucial to ensure dependability of the products and prevent failures. It is well known that the role played by the software in embedded systems is constantly growing, and railway applications are not immune to this trend. In the company, the size of the software for a single project has increased by four times within the last two years, while the hardware has remained basically stable. Alike many other safety-critical industries, GETS has adopted the Model Based Development (MBD) technology in an effort to deal with the growing scale of its applications. MBD practices consist in developing abstract models of the system and automatically generate code from these models. GETS employed MBD first for the development of prototypes [1], and afterward for requirements formalization and code synthesis [2].

Traditionally, unit testing is the main technique adopted to detect errors in the code before integration[1]. With unit testing the code is exercized by executing it and ensuring that its behaviour is compliant to the requirements. Unit testing activities normally require a high cost, and, on the other hand, they do not ensure that the software is completely free from errors, since exploring all the possible behaviours of the code is practically unfeasible by means of testing only.

[1]Extensive testing is performed on the integrated system before the actual deployment, but we consider system testing out of the scope of this paper.

The novel development context driven by MBD practices has opened the door to a new verification process to replace traditional unit testing and ensure higher code safety and cost effectiveness at the same time. In this article we present the experience of the manufacturer in adopting model based testing and abstract interpretation technologies to address this goal.

II. Model Based Testing

Model Based Development (MBD) is a software development approach where the fundamental artifacts are models. Before getting into hand crafted code, the developer has to produce one or more abstract specification of the system in the form of models. Given this specification, software tools can provide simulation of the model behaviour and automatic code generation, this allowing a notable improvement for the process productivity.

Model based testing makes use of the behavioural part of these models, normally state transition systems or finite state automata, to produce tests for the Implementation Under Test (IUT), which is the actual software derived from the models. Though the effort of building abstract models and derive tests is not negligible, recent studies have shown how this approach allows a greater effectiveness in terms of errors detection with respect to the traditional approaches [3].

III. Abstract Interpretation

Abstract interpretation is a particular static analysis method that allows to infer dynamic properties of the code and to detect runtime errors and faulty states of the program without executing the code. The theory beyond this technology was presented by P. Cousot and R. Cousot [4] in the 70s. The core idea of the theory is to define some approximation of the semantics of a program to obtain an abstract semantics. Formal proof of the program can be done at this different level of abstraction in which irrelevant details are removed to reduce the complexity of the verification process. The method defines an overapproximation of all the program reachable states in order to check all the possible

program runs. If a property is satisfied for the analysed set then it is satisfied for the real domain of the program, that represents a subset of the one verified. As one can infer from the theory, tools for abstract interpretation may lead to false positives, caused by the analysis of runs that do not belong to the real domain of the code, and normally these situations have to be checked manually.

IV. Unit Testing Process

The proposed process for unit testing is based on two different phases, namely model based testing and abstract interpretation. With the first phase we address the objective of ensuring that the software is compliant with the requirements given for the code unit. The second phase aims at detecting those runtime errors which may derive from buffer overflow, dereferenced pointers, and other data-flow related deficiencies of the implementation.

A. Phase 1: Model Based Testing

GETS adopted the Simulink/Stateflow platform to introduce MBD within its development process. This is a widely used tool-suite for the modeling and the simulation of control systems.

The first phase of the verification process is described in Figure 1. First, a behavioural model for the functional unit is derived from the unit requirements, which are the functional requirements apportioned to the unit itself. The model is represented through Stateflow, which is a graphical tool implementing a variant of Harel's hierarchical statecharts, a well suited language for representing behavioural models. The Simulink environment supports the execution of Stateflow models, this allowing the possibility of actually running the specification and verifing its behaviour.

In order to assess that the specification is compliant with the expected behaviour of the system, a test suite is manually derived for the Stateflow model, with a set of tests defined according to the requirement coverage criterium: for each unit requirement a unit test is provided in the form of an input data sequence, in order to verify that the requirement has been correctly translated into the Statefow formalism through simulation and observation of the output.

From the Stateflow model, C source code is automatically generated by means of Real Time Workshop (RTW) Embedded Coder. A tool called Test Observer has been developed to automatically translate the unit tests manually provided for the Statefow model into test scenarios for the generated code, which represents our IUT. The tool registers the test execution during the simulation in terms of input and output Simulink *time-series* (Simulink data objects made of (*time*, *value*) tuples for each variable), and directly translates the time-series into given input/expected output matrixes for the generated module. The generated code consists of a single file having a unique interface function for each module. Another tool, called Test Integrator, creates a main file embedding the registered given input, the expected output and the model generated code. For each test case the tool produces an executable file that checks if at all times the current output equals the expected output. If the execution is performed without errors, it can be stated that, for the given unit test, the generated code is consistent with the Stateflow specification.

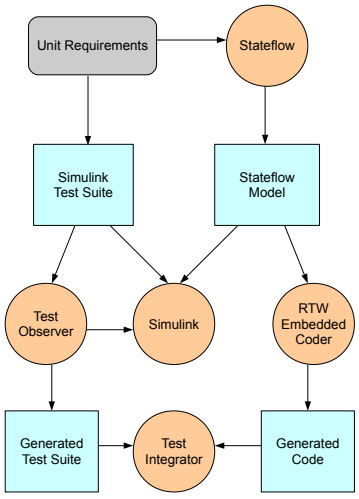

Figure 1. Model Based Testing process

B. Phase 2: Abstract Interpretation

Abstract interpretation is performed by means of the PolySpace tool [5]. PolySpace works on C code and produces its results concerning possible runtime errors through chromatic marks on the code:

- *green* if the statement can never lead to a runtime error;
- *orange* if the statement can produce an error;
- *red* if the statement leads to a runtime error in every analysed runs;
- *grey* if the statement is not reachable.

Usually, the critical issue in using PolySpace is the analysis of the high number of orange warnings caused by over-approximation. GETS has adopted a two step process (see Figure 2) in order to significantly reduce the orange checks that have to be manually reviewed. With the first step the code is quickly verified using a large overapproximation set. In the second step a finer approximation set is applied using the information obtained from the previous step.

The first step is useful to detect systematic runtime errors (red) and unrechable statements (grey). Since no constraints are given in this analysis step, the set-up time spent is negligible. On the other hand results are not selective enough about the orange warnings, and, in order to define the

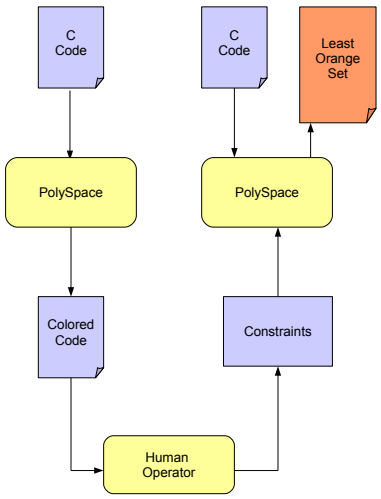

Figure 2. Static Analysis Process

constraints for the subsequent step, each orange has to be associated to the cause that could have produced it. An analyst with a minimum proficiency with the tool can easily evaluate the orange marks and quickly define the classes of causes they belong to, although in this step it is still difficult and time consuming to identify the oranges which are actually false positives. The identified classes represent input constraints to be given to the tool to restrict the analysed abstract domain of the program. Examples of input constraints are interleaving of function calls and range of program variables.

The second verification step, performed with restrictive settings, allows a finer approximation of the real domain of the program and a reduction of the number of false positives. The analyst can quickly check the small number of false positives and in the end is able to state that the code is free from runtime errors.

It should be noticed that the use of two verification steps does not produce a high overhead. Our experience, as shown by the results given in the next section, confirms that the review performed on the first phase is simplified by the fact that the generated code is characterized by a limited number of orange classes, while the results obtained with the second verification normally give a low number of warnings.

V. Case Study

The approach described has been experimented in the verification phase of a project concerning an Automatic Train Protection (ATP) system developed by GETS in 2008. ATP systems are embedded platforms aimed to control the train speed according to the wayside signals and brake the train in case of SPAD (Signal Passed At Danger), which is known to be a common cause of railway accidents.

Behavioural models of the logic of the system have been defined by means of 21 Stateflow models, each one formalizing a specific functionality. The code generated from these models is about 150 KLOC in total in which some functions reach a value of cyclomatic complexity of 60 and show more than 700 paths[2].

For each Stateflow model, unit test cases have been provided according to the requirement coverage criterium. The overall test-suite consists of 327 test-cases covering the 100% of the modeled functional requirements and part of their negative cases. With the test-suite provided, we have been able to detect 42 errors at model level, mostly related to the misinterpretation of the natural language requirements. However, the execution of the Test Observer did not found any discrepancy between the model behaviour and the generated code behaviour. Information on structural decision coverage has been provided for both the Stateflow specification and the generated code. The decision coverage resulted in being around 97% for each model, and in most cases the model coverage and the code coverage were coinciding.

After ensuring the compliance of the implementation with the requirements through the model based testing phase, the generated code is analysed through abstract interpretation to enforce runtime error detection.

Results on the abstract interpretation phase are reported for a representative set of project modules.

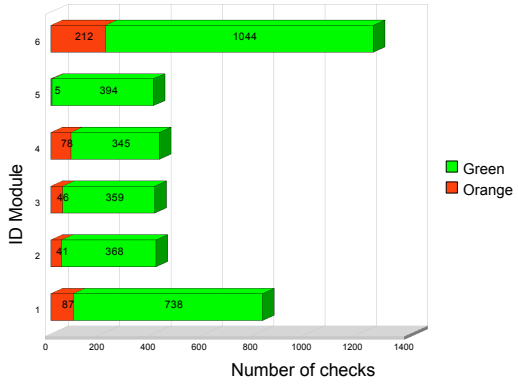

Figure 3. Results of the first PolySpace verification step

As shown in Figure 3, no systematic error (red) has been detected during the first PolySpace verification. Nevertheless, there is a relevant amount of orange marks for which it is not possible to decide if they actually represent faulty states of the program. These orange warnings have been classified according to the kind of approximation that supposedly produced them. Manual analysis of the first results has detected only two classes of causes of oranges: wrong interleaving of function calls and automatic initialization of

[2]Evaluated following boundary-interior method [6]

global variables and input function parameters (Figure 4).

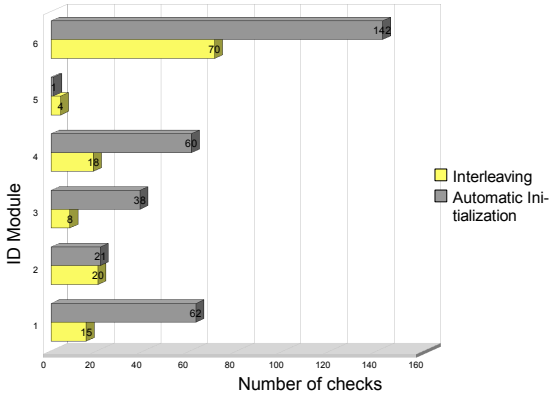

Figure 4. Orange classes associated to the approximations

The analysis of these causes has determined the constraints for the second PolySpace verification. This step produced only a few orange warnings, as shown in Figure 5.

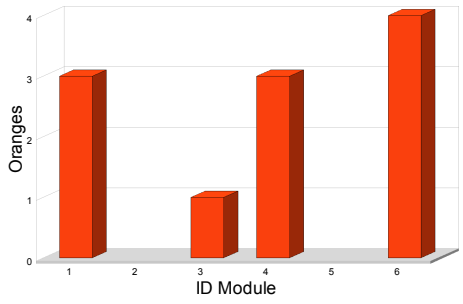

Figure 5. Results of the second PolySpace verification step

The remaining orange marks are due to complex interactions of variables that cannot be constrained by finer approximation bounds. However, an analyst with a sufficient knowledge of the actual meaning of the variables can quickly check if the warnings are false positives or not.
Table I compares the verification cost of the considered project with the effort spent for traditional structural testing of code (according to 100% boundary interior path coverage) in a previous project of comparable size in terms of modules.

Verification Process	**Modules**	**Paths**	**Hours**
Structural Testing	19	2274	**728**
MBT + Abs. Int.	21	>8000	**227** (162 + 75)

Table I
COMPARISON OF VERIFICATION COST

Our first results show that the new approach reduces the verification cost of 70%, even with code having a higher complexity in terms of path number. At the same time we obtain a verification accuracy that can not be achieved with traditional testing.

VI. CONCLUSION

This paper presented the experience of a railway signaling manufactuer in introducing the tecnologies of model based testing and abstract interpretation as part of its development process. The definition of the new approach required a considerable effort of the team for understanding the technologies and merging them with the previously established development process. According to the results obtained on a pilot project, the new approach allows to significantly reduce the verification cost in spite of the growing complexity of the code, and therefore the effort of change actually paid off.
The actual strength of our strategy is the abstract interpretation phase: since the code is not executed but formally analysed the approach allows to fully explore the state space of the program that is a prohibitive goal for traditional testing. At the same time, this technology determines the exact statement in which an error occurs. Instead, traditional testing entails an expensive report analysis to manually find the statement that has triggered the uncorrect output.
The company is currently investigating strategies for extending model based testing from unit level to system integration level. Concerning abstract interpretation, we have already defined guidelines for using PolySpace at this level, but a further analysis is required in order to completely settle the approach.

REFERENCES

[1] S. Bacherini, et al., *A Story about Formal Methods Adoption by a Railway Signaling Manufacturer*, FM 2006. LNCS, 4025/2006. Hamilton, Canada, 2006.

[2] A. Ferrari, et al., *Modeling Guidelines for Code Generation in the Railway Signaling Context*, Proceedings of 1st NASA Formal Methods Symphosium (NFM). Moffet Field, CA, U.S.A., 2009.

[3] A. Pretschner, et al., *One evaluation of model-based testing and its automation*, Proceedings of the 27th International Conference on Software Engineering (ICSE). St. Louis, MO, U.S.A., 2005.

[4] P. Cousot, R. Cousot, *Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints*, Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming language. Los Angeles, CA, U.S.A., 1977

[5] A. Deutsch, *Static verification of dynamic properties*, PolySpace White Paper, 2004

[6] W. E. Howden, *Methodology for the generation of program test data*, IEEE Trans. Comput., vol. C-24, no. 5, pp 554-559, May 1975