

Verification of Model Processing Tools*

Prahladavaradan Sampath, A. C. Rajeev, K. C. Shashidhar and S. Ramesh

General Motors India Science Lab

Copyright © 2008 GM Global Technology Operations, Inc. Published by SAE International with permission.

ABSTRACT

A key requirement for the development of safety-critical systems is the correctness of the tools used in their development process. Standards such as DO-178B mandate the qualification of tools used in the software engineering process of the systems to be certified at the highest levels of criticality. On the other hand, the increasing complexity of software requires the use of methodologies such as Model Based Development (MBD) that are highly tool intensive. MBD employs a suite of tools such as model-translators, code-generators, optimizers, simulators, *etc.*, that can collectively be referred to as model-processors. A model-processor accepts a model in one language, and outputs a processed model in a possibly different language. Due to the increasing sophistication in modern modeling languages, model-processors are prone to implementation errors. Also, they are continuously evolving, resulting in differences in their behaviour across different releases. Our objective is to address the need for ensuring the correctness of model-processors before their deployment in safety-critical software engineering process.

We propose the MetaTest method for verification of model processing tools such as simulators and code-generators. MetaTest uses a *meta-model based test-case generation method* (MMBT) that generates test-cases for model-processors. This approach allows us to directly address the problem of testing the *functionality* of model-processors. We have evaluated MetaTest on some commonly used model-processors in the industry, and from our results, we find it promising to realize a rigorous testing process for such tools.

*The opinions expressed in this document are those of the authors, and do not necessarily reflect the opinions or positions of their employers, other individuals, or other organizations. Mention of commercial products does not constitute endorsement by any person or organization.

MODEL-BASED SOFTWARE DEVELOPMENT

The increasing sophistication of systems has resulted in the adoption of a number of software development methodologies to control the complexity of the software used to realize these systems. Model-based development (MBD) of software is one such method that has found wide acceptance in automotive and aviation industries.

MBD methodology introduces a number of high-level *modeling* languages for modeling different artefacts of software development. These languages are used in all stages of software development including requirements, architecture, high-level design, software partitioning and deployment. Each stage of software development demands different abstractions, and accordingly, MBD provides different languages tailored for the idiosyncrasies of each stage of software development. For example, scenario-diagrams and use-cases can be effectively used for expressing requirements; class diagrams and state-charts can be used for expressing high-level design.

One of the primary benefits of using MBD is that the decisions made at each stage of software development are captured and recorded as artefacts expressed using various modeling languages. These languages have a well-defined *formal* syntax that can be used to effectively structure the software artefacts. This is in contrast to the use of natural language based descriptions that are less structured and are also open to ambiguities. In addition, if the modeling languages have a well-defined and formal semantics, it is further possible to perform various *automatic* analyses on the artefacts, and even to cross-verify different artefacts against each other. The introduction of automation at every stage of software development is perhaps the most significant characteristic of MBD. The possibility of automation is also one of the key factors that make MBD an effective method for designing and building complex safety-critical systems.

ROLE OF TOOLS IN MBD

Automation in MBD is achieved by means of a *tool-chain* which assists and guides the developer through different stages of software development. The tools used as part of MBD range from fairly simple syntax checking tools to tools for simulating the designs to sophisticated tools for analysis and synthesis.

SYNTAX CHECKING TOOLS One of the characteristics of MBD is the use of formal syntax in the place of *natural language*, for representing the decisions made during software development. A formal syntax limits the different ways in which a particular fact can be recorded, and thereby removes the ambiguity that is inherent in a natural language. Syntax checking tools are used for ensuring that the models captured using formal syntax are syntactically well-formed. These tools can also perform some preliminary analysis such as *type-checking* on the models.

SIMULATION TOOLS Many of the modeling languages express dynamic behaviour. Models in such languages can be *symbolically* executed to determine whether the behaviour exhibited by them is indeed the intended behaviour. Such symbolic execution engines (simulators) serve a number of purposes:

- They help to validate the models by observing their behaviour under simulation
- They serve as an early prototype to demonstrate the functionality of the software
- They can be used as a test-bed for generating test-cases

ANALYSIS TOOLS The use of formally defined languages in MBD enables the use of a number of analysis tools at every step of software development. These analysis tools help the designers to perform early validation of the design choices they have made, by exploring the consequences of their choices. Such tools include model-checking engines for verifying logical properties of the software, schedulability analysis tools that can analyse the feasibility of schedules, conflict detection tools that can detect unwanted interactions between components, architecture evaluation tools, *etc.* Correctness of analysis tools is important for systems certified at the highest levels of criticality, as these tools might fail to detect errors, thereby leading to a false sense of security. Hence, standards like DO-178B require the qualification of such analysis tools.

SYNTHESIS AND CODE GENERATION TOOLS The final aim of a software system is the deployment of code on a given architecture. A striking characteristic of MBD is the use of code generation tools to automate the synthesis of low-level code from high-level design expressed in various modeling languages. The correctness of these tools is very important, as they can introduce errors that may not be detectable by other processes mandated for safety-critical certification. For this reason, standards such as DO-178B require the qualification of such tools for use in safety-critical software engineering process.

STANDARDS AND CERTIFICATION

Software has become ubiquitous and it affects every aspect of human society. It controls a vast array of devices we use in everyday life, and it also monitors and controls equipments that can directly affect human life. Software failures have been known to cause disastrous effects, a widely documented case of which is the failure of the Ariane 5 [21] causing a loss of more than USD 370 million.

Standards represent the collective wisdom and the best practices for safety-critical software development. Different industries such as transportation, medical and defence have evolved various standards that reflect their specific safety-critical needs. In particular, the aviation industry widely uses the RTCA/DO-178B [24] standard for the regulatory certification of software used in commercial aircrafts. In the automotive industry, MISRA (Motor Industry Software Reliability Association) has released a set of guidelines for automotive software. The upcoming ISO-26262 standard is also being designed for use by the automotive industry for safety-critical software.

From the perspective of model processing tools, standards mandate the certification of tools used in the development of safety-critical software. For example, DO-178B introduces the concept of *qualification* of tools. According to this standard, a development tool that produces an artefact that has a direct impact on the final product needs to be qualified, if the output of the tool would not be verified and the tool leads to elimination, automation or reduction of any of the DO-178B processes [15]. Qualification differs from *certification* in that the tools can be qualified for use only within the context of a particular project. The tool itself does not receive a stand-alone certification, and needs to be separately qualified for different projects.

RELATED WORK

In order to check the correctness of model-processors, three broad approaches are used: *translator verification*, *translation validation* and *classical testing*. Translator verification is based on the idea of formal verification of software. It involves the use of theorem proving techniques for establishing the correctness of the implementation for translating the source model to the target code [11, 17, 18]. The use of this approach in an industrial context is how-

ever still infeasible due to the complexity of the modeling languages and also the effort required for the use of current generation theorem proving tools [5].

Recently, translation validation [2, 13, 20, 22, 23, 28] has been proposed as an alternative to translator verification. The basic idea in this approach is to verify each instance of translation rather than the translator, *i.e.*, check for the *equivalence* of target code against the source model for each instance of translation. Even though for certain classes of languages this approach is more tractable than translator verification, it often requires internal tool details which are difficult to obtain in the case of third-party tools. Therefore, in an industrial context, testing based approaches remain the preferred method for verifying model-processors.

Software testing has been an important area of research and a large body of literature exists; see [3] for a broad survey in this area. Automatic test-case generation (ATG) for software testing is an active area of research resulting in a plethora of new techniques, many of which are targeted to specific requirements. Recently, many interesting ATG techniques have been developed based on fresh insights into the nature of the domain of the test-cases, and by combining ideas from static/dynamic analysis and model-checking. However, these promising techniques, for example, [7, 10, 14], do not address the ATG problem for model-processors, where the test-cases are models with rich syntactic and semantic structure.

In practice, the usual approach to gain confidence in model-processors is to manually develop a suite of benchmark test-cases. However, this requires a large investment, and moreover it is difficult to give an objective assessment of the quality of the benchmark suite. Therefore, it is advantageous to use an ATG method instead of manual development of test-suites. Below, we discuss a few methods that we are aware of in the rather sparse literature on ATG for model-processors.

Grammar-based testing [6, 16, 19, 31] deals only with those aspects of a model-processor that are based on context-free grammars or attribute grammars – mainly the syntactic constructs. None of these approaches take into account the semantics of the modeling language, which is essential for uncovering subtle semantic errors in the model-processor. Although we incorporate some ideas from grammar-based testing in our method, our focus is on semantics. We not only generate test-models, but also generate specific inputs to these test-models for testing subtle semantic interactions, *which would otherwise require impractically deep syntactic coverage of the grammar to be generated*.

The work presented in [1, 8, 29, 30] perhaps comes closest to the work presented in this paper. These efforts mainly aim to verify the correctness of the optimization rules of a code-generator. The optimization rules are modeled as graph transformation rules, and techniques such as Clas-

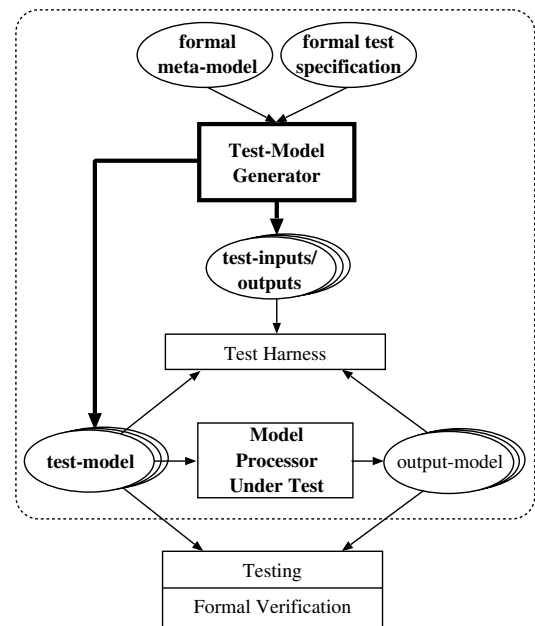


Figure 1: The MetaTest method

sification Tree Method (CTM) are used to obtain coverage of the different components of the graph transformation rules. In this sense, this work is very similar in spirit to our approach in that we also generate test-cases from a model of the semantics. The major difference between our approach and those presented in [1, 8, 29, 30] is the way in which we generate test-cases. The current literature generates test-models and then applies various testing techniques, such as model-based testing, to check the equivalence of test-models and the code generated from them. On the other hand, our method generates the associated test-inputs/outputs that can drive the test-models to exhibit specific semantic scenarios. We will elaborate on this particular point in the following sections.

THE METATEST METHOD

In this section, we present an ATG method for the verification of model-based software development tools. We will focus mainly on code generation tools, although we argue later in the paper that the method is applicable to a wider range of tools including simulators, and even analysis and verification tools. The dotted box of Figure 1 gives a schematic of the different components of the MetaTest method. The MetaTest method takes two inputs: the formal *meta-model* of the modeling language being processed, and a *test specification* that expresses the tester's intent. The formal meta-model of the modeling language specifies both the syntax and the semantics. The test specification can express different forms of exercising the syntactic and semantic rules of the meta-model, for example, by using a limiting size measure on the generated test-models. Given these inputs, the method generates a set of test-models that cover the syntactic aspects of the modeling language. The novelty of our method is