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.

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 impossibly 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 Classification 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
that, in order to cover the semantic aspects of the modeling language, it generates test-models and the associated test-inputs/outputs for the test-models to exercise the specific behaviours described by the test specification. This integrated generation of test-inputs/outputs for the test-models has considerable advantages over checking for the equivalence of test-models and the generated code, using model-based testing or formal verification (the boxes outside the dotted box in Figure 1).

META MODEL The meta-model of a modeling language consists of a formal description of the syntax and semantics of the language. MetaTest provides facilities for representing the syntax of modeling languages in the form of context-free grammars. The assumption of a context-free grammar seems to be a reasonable one – even in the case of essentially graphical modeling languages such as Statecharts, we can model their XMI representation as a context-free grammar. MetaTest also provides facilities to represent the semantics of a modeling language using inference rules. Inference rules are a generic formalism for representing behaviour, and have been used extensively to describe the operational semantics of programming languages such as Java [9], and modeling languages such as UML Statecharts [4] and Stateflow [12].

Example As a running example, let us consider the problem of testing a code-generator for statecharts. All variants of statecharts, such as UML Statecharts and Stateflow, support the notion of state-hierarchy and also the notions of entry, during and exit actions for states. A translation of statecharts into code should preserve the sequence of execution of the various actions\(^1\). A meta-model for statecharts would include the syntax of statecharts and also a semantics that specifies the sequence in which various actions are executed. An example of such a meta-model for Stateflow is presented in [12].

ROLE OF A META-MODEL IN METATEST Traditional ATG methods take as input a program in some language and generate inputs that exercise the functionality of the program, where the functionality is given by an independent specification. Furthermore, traditional testing uses a number of coverage criteria such as coverage of specification\(^2\) (where the specification has a formal structure), coverage of code, etc.

For MetaTest method, the program under test is a tool such as a code-generator that takes as input a model and generates as output a program. The specification of a code-generator is that

\[
\text{the specified semantics of the input model is preserved in the output code}
\]

\(^1\)For sake of simplicity, we ignore transition actions for the moment.

\(^2\)Referred to in the testing literature as functional coverage [3].

This is a very crucial observation, and is one of the corner-stones of the MetaTest method. It clarifies the need for a semantic meta-model for testing tools such as code-generators: without such information, we would be able to generate syntactically correct inputs for the code-generator, but would be unable to test whether the code-generator implements its functionality of preserving the specified semantics of the input models. By employing the MetaTest method, we can test the code-generator by verifying that all forms of \textit{semantic structures} in the modeling language are preserved by the translation. In other words, we can achieve coverage of the functionality of the code-generator by achieving coverage of the semantic structures in the modeling language. This is analogous to specification coverage in traditional testing.

TEST SPECIFICATION A test specification for MetaTest identifies the parts of the meta-model that should be exercised by test-cases. It consists of various coverage criteria on the syntactic and semantic parts of a meta-model. Given a test specification, MetaTest generates a collection of test-purposes, which are then solved for generating the test-cases.

Coverage of Syntax Recall that we represent the syntax of a modeling language using context-free grammar. There are a number of coverage criteria that have been studied in the literature for context-free grammars [16, 19]. MetaTest uses some of the techniques presented in these works for generating input-models for code-generators.

The main purpose of syntax based testing is to ensure that all syntactic constructs in the modeling language are accepted as inputs to the code-generator. This is a minimum requirement for the correctness of the code-generator. A test-case for syntactic coverage consists of just an input-model, and does not include any information about the behaviour of the model.

Example In the case of a code-generator for statecharts, some of the syntactic structures that can be covered by the test specification include:

\begin{itemize}
\item Coverage of all the syntactic elements of statecharts: AND states, XOR states, state-hierarchy, transitions, junctions, etc.
\item Coverage of particular types of combinations of the syntactic elements: an AND super-state containing an XOR super-state or vice versa, a transition having an XOR super-state as source and an AND super-state as target, etc.
\end{itemize}

As an illustration, consider the coverage of transitions with all possible combinations of source and target state types. The four possible combinations of having an XOR super-
state and a basic state as the source and target of a transition are given in Table 1. Further combinations could include AND super-states, different forms of junctions, etc., as the source and target.

Coverage of Semantics Coverage of the semantics of a modeling language is the main innovation of the MetaTest method. We use inference rules to express the semantics of modeling languages. Therefore, particular semantic behaviours will be reflected as patterns in the inference trees built using these inference rules. Therefore, coverage of semantics can be expressed in terms of various forms of coverage of inference trees.

MetaTest employs a number of coverage criteria over the inference trees. At the simplest level, MetaTest uses the depth of inference as a coverage criteria. In this case, the coverage criteria is specified as an integer, say \( n \), and all non-isomorphic inference trees of depth at-most \( n \) are used to generate test-cases. This form of test-coverage is fairly robust, and can reveal a number of subtle errors in code-generators [27]. However, this coverage criteria suffers from combinatorial explosion, and can become infeasible for even small values of \( n \).

MetaTest also provides more refined coverage criteria to explore the space of behaviours. One such criterion is the rule coverage, which ensures that every rule in the semantic meta-model is exercised at least once by the test-cases. This coverage criteria is analogous to statement coverage in the traditional testing of programs.

Another related coverage criterion is the rule dependency coverage, which ensures that all possible dependencies between the rules are exercised by the test-cases. A dependency of rule \( r_1 \) on rule \( r_2 \) is considered to be covered by an inference tree that contains an application of \( r_2 \) as a descendant of an application of \( r_1 \). This coverage criteria is analogous to data-flow coverage in traditional program testing.

The coverage criteria defined above are expressed in terms of the structure of the rules in the semantic meta-model. Test-cases are generated by exercising various patterns that appear in the meta-model. This justifies our claim that MetaTest is a meta-model based testing method, in contrast to the model based testing methods that explore patterns occurring in a model.

### Table 1: Example test-models obtained by syntactic coverage

<table>
<thead>
<tr>
<th>Source state</th>
<th>Target state</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic</td>
<td>basic</td>
</tr>
<tr>
<td>basic</td>
<td>XOR</td>
</tr>
<tr>
<td>XOR</td>
<td>basic</td>
</tr>
<tr>
<td>XOR</td>
<td>XOR</td>
</tr>
</tbody>
</table>

Example In the case of a code-generator for state-charts, some of the semantic structures that can be covered by the test specification include:

- Coverage of all the inference rules used to express the semantics of statecharts
- Coverage of particular types of combinations of inference rules
- The set of action-types (entry, during and exit) that can be executed by a state

As an illustration, the coverage of action-types could include a state-machine having a state that would execute both its during and exit actions. For covering this requirement, MetaTest method would generate a state-machine as in Figure 2, a test-input sequence \( \langle e_2 \rangle \) and a test-output sequence \( \langle a_1, a_2 \rangle \). Note that MetaTest also generates the sequence \( \langle e_1 \rangle \) as a test-setup sequence that will be executed to put the test-model into a particular configuration before executing the test-input sequence.

TEST-CASE GENERATION Based on a test specification in the form of syntactic or semantic coverage criteria, MetaTest generates a set of test-purposes that ensures that the coverage criteria are satisfied. Test-case generation is performed by generating and solving constraints that are derived from the test-purposes. Our current implementation of MetaTest uses off-the-shelf constraint solvers, such as Yices [25], for multi-sorted equational logic.

Test-Models Test models are generated by solving the constraints derived from both syntactic coverage criteria and semantic coverage criteria. These test-models can be fed as input to the code-generator.

Test-Sequences A test-case for semantic coverage includes not only a test-model, but also a test-sequence.

\(^3\)We are using Stateflow semantics for this example.
in the form of inputs to exercise the model and the corresponding expected outputs. These test-sequences are generated by solving the constraints obtained from inference trees representing semantic behaviour patterns.

The generation of test-sequences in addition to test-models appears to be unique to the MetaTest methodology. Other related methods for testing code-generators, such as [29,30], generate code from test-models and then check for equivalence between the models and the generated code, using model-based testing. It is also possible to perform formal equivalence checking between test-models and code, using techniques such as translation validation [20, 23]. This is shown in Figure 1 by the box “Testing/Formal Verification”.

In comparison to model-based testing, the MetaTest method has the advantage of generating directed tests that test very specific behaviours of the generated test-models. Model-based testing would typically use fairly generic coverage criteria to test the equivalence of the test-model and the generated code, and there is a possibility that the behaviour from which the test-model is actually generated may go untested. Also, model-based testing would require a large number of test-cases to check the equivalence of the model and the code. At the other end of the spectrum, formal equivalence checking of the test-model and the generated code is guaranteed to identify discrepancies, but such techniques are typically very difficult to apply and may require expert human assistance. In comparison, MetaTest needs to execute only a single test-case for each pair of test-model and its generated code. This is a very important consideration for the scalability of MetaTest for complex code-generators.

TEST EXECUTION AND ANALYSIS MetaTest also supports test-execution and analysis of the test results. This functionality is represented by the “Test Harness” block in Figure 1. The test-execution framework has to be sufficiently generic to be able to test a number of different kinds of model processing tools like code-generators, simulators, etc. The main steps performed by the test-execution framework while testing a code-generator are:

- Process a test-model generated by MetaTest, using the code-generator
- Create an executable from the output of the code-generator
- Feed the executable with the test-input generated by MetaTest
- Observe the outputs of the executable, and compare them with the expected outputs generated by MetaTest

There are a number of complex engineering issues to be considered in test-execution. These include:

- Converting the test-cases from an internal representation used by MetaTest into a format that can be accepted by the model processing tool. This includes translating both test-models and test-inputs/outputs
- Feeding test-inputs to the executable generated from the test-model
- Observing the output behaviour of the executable

Example Consider the statechart in Figure 2. In order to execute this test-case (consisting of a test-model, a test-setup sequence and a test-input/output sequence), MetaTest will provide the test-model as input to the code-generator and will create an executable from the generated code. It then runs the executable and provides as input, the test-setup sequence $\langle e_1 \rangle$ followed by the test-input sequence $\langle e_2 \rangle$, and observes whether the executable produces the action sequence $\langle a_1, a_2 \rangle$. If the observation matches the expected output sequence, then the test is considered to be passed.

DISCUSSION

In this paper, we have presented an automatic test-case generation method for model-processors. The running example we have used is that of code-generators for state-charts, which are widely employed in model-based engineering.

The essence of our method is the use of intended semantics of the modeling languages. In a sense, the intended semantics represents a specification of the functionality of the model processing tool. It is abstract, and is even independent of the actual transformations performed by the model-processor to generate the output. Note also that the intended semantics depends on the model processing tool being verified. For illustration, in the case of statechart code-generators, the intended semantics of a statechart is the sequence of actions performed in response to an input event, while in a given state, i.e., it is a function that takes as arguments the current state and the input event, and produces as output a sequence of actions. However, from the perspective of a tool that generates an inheritance hierarchy of classes from the given statechart, the intended semantics could just be a partial order extracted from the structure of the statechart, which represents the dependencies between the states in the statechart. Similar arguments can be made for other modeling languages and model processing tools.

The MetaTest method is robust and can be applied to a number of modeling languages and model processing tools. It is also scalable and applicable to complex model processing tools. We have, in previous work [26, 27], demonstrated the applicability of this method to verify lexical analyser generators, which are quite complex model processing tools in their own right. We are also applying this method for verifying code-generators and simulators for statechart languages. The running example in this pa-
per gives a flavour of the results we have obtained in this case.

CONCLUSION

Model-based development methods are increasingly being used for engineering complex safety-critical software. On one hand, the tools used as part of MBD are continuously evolving and increasing in sophistication, and correspondingly, the verification of such tools is becoming increasingly difficult. On the other hand, the need for such sophisticated tools is also increasing in the domain of safety-critical software. There is a need for new methods and techniques for addressing this verification challenge.

Current practice of engineering safety-critical software also relies heavily on standards such as DO-178B, and these standards demand very rigorous qualification of the tools used in the engineering process. It is not currently possible to certify a development tool once and for all, and use it as part of various safety-critical engineering efforts. One of the reasons for this is the inability of current verification techniques to convincingly verify a model processing tool. The method presented in this paper can form the basis of such a convincing verification process for model processing tools.

ACKNOWLEDGMENTS

We would like to thank the members of our group at General Motors India Science Lab for their valuable feedback on this work. We would also like to thank the anonymous referees for their constructive comments.

REFERENCES


