Abstract

Model-based development is increasingly becoming the method of choice for developing embedded systems for applications in automotive and aerospace industries. It relies on tool-suites consisting of a variety of model-processing tools like simulators, model-translators and code-generators. The correctness of these tools used in the development process is a key requirement for safety critical applications. This paper proposes a novel testing methodology for the rigorous verification of model processing tools.

The proposed methodology takes as input the syntactic and semantic meta-model of a modeling language, expressed in the form of inference rules. Using a coverage criteria over this meta-model, it generates test-models, and test-inputs for these test-models. Apart from testing the syntactic aspects of the translation, our method aims at testing subtle semantic interactions of the modeling language that are potentially mistranslated by the model-processing tools.

We illustrate the methodology with a simple prototypical process calculus. We also report on the experiments carried out with Stateflow, a variant of hierarchical state-machines implemented in the Matlab/Simulink tool-suite.

1. Introduction

A key requirement for high integrity applications in automotive and aerospace domains is the rigorous verification of the tools used in the development process, over and above the rigorous verification of the models and the low-level code (generated or developed) for the applications. Model based development, as popularized by companies like The Mathworks [16], Telelogic I-Logix [11], etc., has become

the method of choice for developing such high integrity embedded systems. It makes use of high level modeling languages like Simulink/Stateflow, UML/StateMate, etc., and is tool-intensive, employing a suite of tools like simulators, translators and code-generators, collectively referred to as model-processors. This paper focuses on the problem of verifying the model-processors. In particular, it proposes a method for rigorously testing these tools.

Model-processors are complex software, the design of which requires thorough understanding and careful implementation of the syntactic and semantic aspects of the modeling language. The correctness of model-processors demands that the semantic behavior be equivalent between the input model and the corresponding translated code, in the case of translators, and the observed behavior, in the case of simulators. The inputs and outputs of model-processors are models having complex syntactic and semantic structures, making it difficult to apply traditional testing techniques which typically deal with simpler input and output domains.

The testing methodology proposed in this paper is based on the observation that model-processing tools preserve the behavior of the model, while translating it from the modeling language to the implementation language. In other words, a model-processor encodes the semantics of the modeling language into the implementation language. Testing a model-processor is about ensuring that, given a model and an implementation of the model (which is the output of the model-processor), the semantics of the model is captured faithfully in its implementation. Therefore, testing of model-processors primarily requires a specification of the semantics of the modeling language.

The proposed methodology takes as input the formal meta-model of the modeling language being processed by the tool. This formal meta-model includes the specification of the semantics of the language, in addition to the syntactic information. Based on a test specification that identifies the behaviors to be tested, our methodology generates a set of test-models for testing the model-processor. These models

*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.
are fed to the model-processor, which will interpret them or generate the corresponding target code. In addition, the methodology also generates a set of test-inputs and the corresponding set of expected outputs for these test-models. This can be used for checking the equivalence of the test-models and their corresponding simulations/translations.

A unique aspect of our methodology is the use of a uniform framework of inference rules for specifying both syntax and semantics. This enables the generation of test-models for both syntactic and semantic aspects of the modeling language. Another novel aspect is the integrated generation of test-inputs for the generated test-models.

Using a prototype implementation of the proposed methodology, we have generated test-models to exercise simulators and code-generators for Stateflow, a well-known graphical language of the Matlab/Simulink tool-suite that is extensively used in the development of embedded systems. During this experimentation, we came across some discrepancies which were traced back to errors [1] in the definitions of some of the semantic rules given in [8].

Our methodology is applicable in situations where a formal verification of the model-processor is not possible: for example, if the source code of the model-processor is not available, or in cases where the current verification technology does not scale to the complexity of the model-processor. Another situation where our methodology can be applied is for checking multiple implementations of the same model-processing tool: for instance, many vendors re-implement Rhapsody state-chart code generation, since the standard code-generator may not satisfy the in-house coding standards. In extreme cases, vendors even change the standard semantics of modeling formalisms to suit their deployment environment: for example, changing the execution semantics of state-charts from being event-triggered to being polled. In this case, our methodology provides a means for generating test-cases which are customized to the particular semantic scenario.

The rest of the paper is organized as follows: Section 2 discusses the relation of this work to the existing literature. Section 3 gives an overview of our testing methodology. Section 4 illustrates our methodology with an example. The details of our methodology are presented in Section 5. Section 6 gives some implementation details, followed by discussion and conclusion in Section 7.

2. Related Work

Software testing has been an important area of research and a large body of literature exists; see [5] for a comprehensive, but dated, survey in this area. While the testing of conventional programs has probably matured, resulting in well-accepted methodologies, testing of compilers and model-processors is still an order of magnitude more complex and less developed.

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 [7, 15]. Use of this approach in an industrial context is however still infeasible due to the complexity of the modeling languages and also the effort required for the use of current generation theorem proving tools [6]. Recently, translation validation [4, 9, 20–22] 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 the target code against the source model. 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.

Methods proposed in the literature for testing language processing tools include white-box coverage testing [3, 23], benchmark testing and grammar-based testing [2, 13, 17]. However, these approaches have known drawbacks. White-box coverage testing requires the availability of the source code of the tool, which is most often not the case for third-party tools. In contrast, our approach requires only the specification of the model-processor to be tested – i.e., the syntax and semantics of the modeling language. Compiling a benchmark suite requires a large investment, and moreover it is difficult to give an objective assessment of the quality of the benchmark suite. Grammar-based testing deals with only 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-processing tool.

Although we incorporate some of the techniques of grammar based testing in our method, our focus is on semantics. We not only generate test-models, but also generate specific inputs to them that test subtle semantic interactions, which would otherwise require impractically deep syntactic coverage of the grammar to be generated.

3. Overview of the Proposed Methodology

The proposed methodology is an attempt to formalize an expert tester who designs inputs for a model-processor under test (MPUT). The inputs to the MPUT are models,
which we call test-models, that conform to the meta-model accepted by the MPUT. The tester has a complete understanding of the syntax and semantics of the modeling language. He/she generates a set of test-models such that it has complete coverage over all aspects of syntax and semantics. More often, a human tester focuses on the semantic subtleties of the meta-model and accordingly chooses the test-models so that possible errors in the implementation are uncovered.

Figure 1 summarizes our approach. As shown in the figure, our approach assumes that the meta-model of the modeling language is available. It is expressed as a set of inference rules that captures not only the syntactic specification of the models, but more importantly, their semantic behaviors. The main advantage of inference rules is that they are capable of representing a wide range of syntactic and semantic definitions. In particular, they can represent context-free grammars. Also, the well-formedness conditions – sometimes referred to as the static semantics of the language – which are typically not expressible in terms of context-free grammars can be expressed using the side-conditions of the inference rules. Similarly, both small-step SOS [19] semantics and big-step natural semantics are expressible in the form of inference rules.

We use inference rules to obtain inference trees, by instantiating the rules in a tree structure such that a premise of one rule is the conclusion of another. Inference trees are the primary objects of interest in our methodology, as they represent a syntactic or semantic derivation. Our aim is to explore such interesting derivations and generate test-cases from these derivations.

The second input to our approach is the test specification, which identifies the different syntactic and semantic scenarios that are to be tested – it is the mechanism for identifying an interesting set of inference trees. It typically states some coverage criteria such as the coverage of certain syntactic and semantic rules, their combinations etc. We have so far considered fairly simple test specifications identifying a set of inference rules to be exercised, and the depth to which exploration of possible inference trees will be carried out in the generation process.

The central component of the proposed approach is the test-model generator (TMG). Given the meta-model of a modeling language and a test specification, the TMG generates a set of syntactically valid model instances. The fact that the meta-model includes the semantic rules of the modeling language makes it possible to examine whether certain semantic subtleties in the language have been correctly dealt with by the MPUT. The TMG first constructs an inference tree representing the required semantic scenario and then derives a test-model that is capable of exhibiting the scenario. The TMG also derives a test-input for the test-model that will drive the model to exhibit that particular scenario.

We refer to the pair of a test-model and its test-input as a test-case. The TMG generates test-cases by solving the constraints extracted from the inference tree.

The obtained test-cases can be exercised by a test harness to check whether the model-processor works correctly. For instance, one could take the test-model and run it through the model-processor to obtain an output-model. This output-model can then be simulated on the test-inputs to see whether the behavior of the output-model matches that of the input test-model.

Internally, TMG consists of three components as illustrated in Figure 2: an inference tree generator (ITG), a constraint generator (CG) and a constraint solver (CS).

ITG ITG takes as inputs the meta-model and a test specification, and exhaustively generates all possible inference trees. It essentially performs combinatorial exploration of the possible inference trees. Combinatorial coverage is controlled by the test specification, which can be as simple as the depth of inference trees to be generated. More sophisticated test specifications are possible, but we do not consider them in this paper.

CG Inference trees represent specific syntactic and semantic scenarios that are to be exercised. CG extracts the syntactic and semantic constraints on test-models and test-inputs, in order to exhibit the behavior described by the inference tree.

The selection of a logical system for expressing the constraints is also an important part of the design of
4. A Simple Illustration

We illustrate the ideas behind the Test-Model Generator (TMG) by considering a simple language of processes, styled after CCS [18]. The formal meta-model of the language is described by the inference rules in Figure 3 and Figure 4.

### Figure 3. Syntax of process language

<table>
<thead>
<tr>
<th>Inference Rule</th>
<th>Formulas</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Ax-Proc)</td>
<td>⊢ ∅ : Proc</td>
</tr>
<tr>
<td></td>
<td>⊢ S : Proc ⊢ a : Action</td>
</tr>
<tr>
<td></td>
<td>⊢ a.S : Proc</td>
</tr>
<tr>
<td>(Ax-ACT)</td>
<td>⊢ S₁ : Proc ⊢ S₂ : Proc</td>
</tr>
<tr>
<td></td>
<td>⊢ S₁</td>
</tr>
<tr>
<td>(PARALLEL)</td>
<td>⊢ S₁ + S₂ : Proc</td>
</tr>
<tr>
<td>(CHOICE)</td>
<td>⊢ S₁ + S₂ : Proc</td>
</tr>
<tr>
<td>(SEQ AX)</td>
<td>⊢ S₁ a S₂ : Proc</td>
</tr>
<tr>
<td></td>
<td>⊢ S₁ a L S₂ : Proc</td>
</tr>
</tbody>
</table>

### Figure 4. Semantics of process language

This language is a simple language whose syntax can be represented as a context-free grammar, and this simplicity is reflected in the inference rules in Figure 3. The language is modeled using two syntactic sorts: Action and Proc, which represent the syntactic categories of the language.

The semantic rules, shown in Figure 4, are slightly more complex. It includes the specification of the sort ActSeq representing the semantic domain of the process language. The inference rules include the traditional single-step transition (given by a →, where a is an action) and the transitive closure of the single-step transition (given by (L) ↠, where L is a sequence of actions). Transitive closure of the single-step transition is needed by our implementation to generate sequences of actions as test-inputs for processes.

<table>
<thead>
<tr>
<th>Inference Rule</th>
<th>Formulas</th>
</tr>
</thead>
<tbody>
<tr>
<td>(EMPTY SEQ)</td>
<td>⊢ ϵ : ActSeq</td>
</tr>
<tr>
<td>(ACTSEQ)</td>
<td>⊢ a : Action ⊢ L : ActSeq</td>
</tr>
<tr>
<td></td>
<td>⊢ a.L : ActSeq</td>
</tr>
<tr>
<td>(ACT)</td>
<td>⊢ a.S₁ a S₁</td>
</tr>
<tr>
<td>(PAR)</td>
<td>⊢ S₁</td>
</tr>
<tr>
<td>(CHOICEL)</td>
<td>⊢ S₁ a S₁’</td>
</tr>
<tr>
<td></td>
<td>⊢ S₁ + S₂ a S₁ + S₂</td>
</tr>
<tr>
<td>(CHOICER)</td>
<td>⊢ S₂ a S₂’</td>
</tr>
<tr>
<td>(SEQ AX)</td>
<td>⊢ S₁ a S₂’</td>
</tr>
<tr>
<td></td>
<td>⊢ S₁ a L S₂’</td>
</tr>
</tbody>
</table>

The primary input to the Inference Tree Generator (ITG) is the meta-model consisting of the inference rules for both syntax and semantics. For the test specification, we choose...
the depth to which the meta-model has to be explored as 4, and the specification of the test-cases to be extracted as variables S₁ and L in judgements of the form \( \vdash S_1 \rightarrow L \). Here, S₁ denotes the test-process and L denotes the test-action-sequence for S₁.

Given these inputs, ITG generates inference trees having depth less than or equal to 4, with judgements of the form \( \vdash \_ \rightarrow \_ \) as the conclusion of the root. An example inference tree of depth 4 that will be generated is:

```

<table>
<thead>
<tr>
<th>Act</th>
<th>( \vdash P_1 \xrightarrow{A_4} P'_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHOICER</td>
<td>( \vdash P_3 \xrightarrow{A_4} P'_3 )</td>
</tr>
<tr>
<td>SEQ</td>
<td>( \vdash P_2 \xrightarrow{A_4} P'_2 )</td>
</tr>
<tr>
<td>SEQAX</td>
<td>( \vdash P_4 \xrightarrow{A_4} P'_4 )</td>
</tr>
</tbody>
</table>

\( \vdash P_5 \xrightarrow{L_5} P'_5 \) |

\( \vdash P_6 \xrightarrow{L_6} P'_6 \) |
```

This inference tree is a justification for the fact that process \( P_6 \) can perform a sequence \( L_6 \) of actions, by firing CHOICEL rule followed by CHOICER rule. Note that the terms \( P_1 \), \( A_4 \) and \( L_1 \) are variables that need to be instantiated to obtain a ground inference tree. In this sense, the generated inference tree is a template which can be exhibited by many different processes. Now, generating a test-case requires the calculation of a particular process (test-model) that can exhibit the behavior specified by the given inference tree, and a particular sequence of actions (test-input) that can be performed by this process, while exhibiting the required behavior.

The generation of a test-case is achieved in two steps:

- Generation of a constraint from the inference tree; and
- Solving the constraint to obtain a test-case.

Translation of the generated inference tree into a constraint is performed by the Constraint Generator (CG). The language we use for expressing constraints is a multi-sorted first-order logic with equality and some extra-logical constants. In the above example, these constants are:

- Function symbols corresponding to the process constructors ('.', '||', '+', '\emptyset'),
- Function symbols for action-sequence concatenation ('.') and an empty action-sequence ('\emptyset'),

The following (simplified) constraint \( \text{TestCase}(P_6, L_6) \), with variables \( P_6 \) and \( L_6 \) representing the test-process and the test-action-sequence respectively, is generated by CG:

\[
\exists P_1, P'_1, P''_1, Q_1 : \text{Proc}; A_1, A_3 : \text{Action}; L_5 : \text{ActSeq} \mid
P_6 = P_1 + Q_1 \land P'_1 = A_1, P''_1 \land Q_1 = A_3, P''_1 \land
L_6 = A_1, L_5 \land L_5 = A_3, \epsilon
\]

The test-process and test-action-sequence are then calculated by solving the first-order formula.

If the generated constraint is satisfiable, we obtain an assignment for the free-variables, assigning terms to the free-variables. In the above example, an assignment calculated by the solver for the variables \( P_6 \) and \( L_6 \) is

\[
P_6 = a_1, \emptyset + a_0, \emptyset
\]

\[
L_6 = a_1, a_0, \epsilon
\]

In summary, each inference tree generated by the ITG will give rise to a constraint and each satisfiable constraint will give rise to an assignment. These assignments can then be used as test-cases to test a simulator of the process language. This can be done by simulating the process \( P_6 \) and ensuring that it can perform the sequence of actions \( L_6 \).

5. Test-Model Generator

In this section, we present the details of the approach illustrated in Section 4.

5.1. Meta-Model

Meta-model is the primary input to the Test-Model Generator. It consists of syntactic and semantic definitions expressed using terms, predicates, judgements and inference rules.

Term Terms represent the basic syntactic and semantic values of the modeling-language. We assume that they are taken from a multi-sorted algebraic signature which specifies the constructors used in the terms. We also assume a supply of sorted variables which can be used in the terms.

For instance, in our example process language, we have used the sorts Proc and Action for representing syntactic terms, and the sort ActSeq for semantic terms. We have also used constructors such as '.', '+', '||' and '\emptyset' for constructing processes; '\epsilon' and '.' for constructing action sequences\(^1\).

Predicate Predicates represent the side-conditions on the inference rules. They express constraints on sets of (sorted) terms from the meta-model.

The choice of a logical formalism for expressing the predicates is very much influenced by the language used for expressing and solving the constraints generated from the inference trees. We assume that the predicates are expressed in a multi-sorted first-order logical language with equality and some extra-logical constants.

\(^1\)The same constructor '.' is used in both syntactic and semantic terms without fear of confusion.
Judgement  A judgement is of the form $\Gamma \vdash \Delta$ where $\Gamma$ is a sequence of (sorted) terms. A meta-model will typically contain different kinds of judgements, distinguished by a signature consisting of the sequence of sorts which identifies the sorts of the argument terms that the given judgement can be applied to.

In Section 4, we have represented the judgements of our example process language in a simplified form. However, in our implementation, these judgements are formally represented using a number of different judgement kinds:

- $\vdash^\text{wf}_\text{Proc}$ is used to represent well-formed process terms in the syntax, and takes terms of sort Proc as argument.
- $\vdash^\text{wf}_\text{Act}$ is used to represent well-formed action terms in the syntax, and takes terms of sort Action as argument.
- $\vdash^\text{wf}_\text{ActSeq}$ is used to represent well-formed action-sequence terms in the semantics, and takes terms of sort ActSeq as argument.
- $\vdash$ is used to represent a single-step execution of processes, and takes as argument a 3-tuple of terms Proc, Action and Proc respectively.
- $\vdash$ is used to represent multi-step execution of processes, and takes as argument a 3-tuple of terms Proc, ActSeq and Proc respectively.

For example, the judgement $\vdash P_6 : \text{Proc}$ in the example from Section 4 will be formally represented as:

$\vdash P_6 : \text{Proc}, L_6 : \text{ActSeq}, P'_6 : \text{Proc}$

Inference Rule  Inference rules formalize the deductions that can be applied to judgements. In the literature, inference rules are typically expressed using a pattern-matching idiom. As part of this idiom, premises and conclusion of the rule are expressed as judgements over sorted terms containing variables, and the commonalities between the terms express the implicit side-conditions. For example, in the rule SEQ of Figure 4, the premises are

$\vdash S_1 : \text{Proc}, a : \text{Action}, S'_1 : \text{Proc}$

$\vdash S'_1 : \text{Proc}, L : \text{ActSeq}, S''_1 : \text{Proc}$

The pattern identified by $S'_1$ in fact expresses an implicit side-condition that equates the third argument of the first premise and the first argument of the second premise. Similar implicit identifications can occur between the premise judgements and the conclusion judgement. Another use of the pattern-matching idiom is to constrain the terms to have a certain form. For example, the conclusion

$\vdash S_1 || S_2 : \text{Proc}, a : \text{Action}, S'_1 || S'_2 : \text{Proc}$

of the PAR rule in Figure 4 expresses an implicit side-condition that the first and the third arguments of the conclusion will be terms constructed using the `||` constructor.

We formally represent an inference rule as a tuple $(P, C, S)$, where

- $P$ is a (possibly empty) sequence of kinds of the premise judgements
- $C$ is the kind of the unique conclusion judgement
- $S$ is a predicate representing the side-condition of the rule

The side-condition of an inference rule expresses some relation between the terms appearing in the premise and conclusion judgements. We formally represent it as a function that takes judgements of the kinds specified by $P$ and $C$ as arguments, and returns a predicate relating the terms appearing in the argument judgements.

For example, the rule ACT in Figure 4 is represented as the tuple $(P, C, S)$ where

$P \equiv \epsilon$

$C \equiv \vdash$

$S \equiv \lambda J \cdot (X_1 = A_1, X_2)$

where $J$ is the judgement $\vdash X_1 : \text{Proc}, A_1 : \text{Action}, X_2 : \text{Proc}$

5.2 Test Specification

The second input to the Test-Model Generator is the test specification. Abstractly, the purpose of the test specification is to specify syntactic structures and semantic behaviors that have to be tested. Concretely, in our current implementation, this is specified using two pieces of information:

- The maximum depth of the inference trees to be generated
- The judgement kind(s) and the argument positions within the judgement(s), for extracting test-models and test-inputs

For the process language in Section 4, we have considered a depth of 4. For the second part of the test specification, we identified the judgement kind $\vdash$, and further identified its first argument (of sort Proc) as a test-model, and its second argument (of sort ActSeq) as a test-input.

5.3 Generating the Inference Trees (ITG)

An inference tree is generally taken to be a tree of judgements, where each judgement is labeled by the inference
rule used to derive this judgement. An inference tree also satisfies various conditions which ensure that the judgement in each node can indeed be derived by applying the inference rule labeling the node to the judgements in the children nodes.

Our concrete representation of inference trees is dual to the typical representation given above. We take an inference tree to be a tree whose nodes are inference rule instances. Each node (rule instance) contains a sequence of judgements corresponding to the premise judgements of the rule, and a unique judgement corresponding to the conclusion judgement of the rule. Also, the node is annotated with a predicate obtained by applying the side-condition function to the premise and conclusion judgements of the rule instance. Each node is also associated with another predicate called link predicate, obtained by unifying the conclusion judgement of the node and the corresponding premise judgement of its parent node. We also note that the judgements in the inference trees may contain variables and not just ground terms.

From this representation of inference trees, we observe that the trees can be represented as algebraic terms with judgement kinds interpreted as sorts and inference rules interpreted as constructors. For instance, in the case of the process language in Section 4, corresponding to the judgement kinds \( \vdash \) and \( \leftrightarrow \), we have sorts Trans and Exec. Now, corresponding to the inference rule SEQ in Figure 4, we have a constructor:

\[
\text{SeqTerm} : \text{Trans} \times \text{Exec} \rightarrow \text{Exec}
\]

which is obtained by interpreting the kinds of the premise judgements of the rule as the sorts of the argument terms of the derived constructor, and the kind of the unique conclusion judgement of the rule as the sort of the term obtained by applying the derived constructor to arguments having the derived sorts.

Using this algebraic signature, we can generate a set of terms. We can use well-known techniques for the generation of terms from grammars [13, 17]. The term generation process uses the depth specified as part of the test specification to limit the depth of the terms generated. As mentioned in Section 5.2, test specification also identifies a judgement kind for extracting the test-cases. This judgement kind will be associated with a derived sort and the term generation process generates only those terms whose sort is same as this derived sort. This ensures that the kind of the conclusion judgement in the root node of the generated inference tree will be the one identified by the test specification.

It then remains to convert these terms into inference trees by mapping each constructor in the term to a fresh instance of the corresponding inference rule, and generating the corresponding side-condition predicates and link predicates. By a fresh instance of an inference rule, we mean that judgements with fresh variables as arguments are used for the premises and the conclusion of the inference rule.

### 5.4. Generating the Constraints (CG)

Given an inference tree, the side-condition predicates and the link predicates give the conditions for the validity of the inference tree. These conditions are expressed over the variables occurring in the tree.

The root of every generated inference tree is a rule instance whose conclusion judgement has the same kind as that given in the test specification. Using the test specification, we can therefore identify the variables \( P \) and \( L \) occurring in the conclusion judgement, which correspond to the test-model and test-input that we wish to generate.

We generate the constraint as a predicate \( \text{TestCase}(P, L) \) by taking the conjunction of the side-condition predicates and the link-predicates occurring in the inference tree, and then existentially quantifying all free variables (other than \( P \) and \( L \)) which occur in the resulting predicate.

### 5.5. Solving the Constraints (CS)

Generating a test-case requires solving the predicates generated by the Constraint Generator. We would like to use off-the-shelf solvers for this step, and in our current implementation we use the Alloy tool [12] for solving the predicates. Using off-the-shelf solvers requires the translation of predicates from the language that is used in the meta-model to the language used by the solvers. We have encapsulated this translation process and the actual invocation of the solver into the Constraint Solver (CS) module.

We have implemented a simplistic translation of multi-sorted first-order logic formulae into the Alloy specification language. The models for the formulae are algebraic terms, which also can be expressed in the Alloy language. If a formula is satisfiable, then a solution to the formula is given as an assignment to the free-variables in the formula. On the other hand, if the formula is unsatisfiable, it indicates that the corresponding inference tree is an infeasible inference tree for our meta-model.

### 6. Implementation and Evaluation

We have implemented an initial prototype of the work presented in this paper and are able to generate test-models from the given meta-model rules. The choice of Alloy for solving the constraints was made based on our familiarity with the language and the tool. It performs bounded satisfiability checking for solving the predicates. The Alloy tool also provides a Java API which can be used for optimizing the constraint solving process for a collection of predicates:
it helps in checking whether a model generated as the solution for one predicate also satisfies other predicates in that collection – in this case, the expensive step of satisfiability checking for the satisfied predicates can be elided. This also reduces the number of generated test-cases, by sharing the same test-case among multiple predicates.

For the process language in Section 4, we have generated test-cases with different values for the depth of inference trees. The example test-case given in Section 4 illustrates one of the interesting test-cases generated. The results of the test-case generation for our process language have been summarized in Table 1. The failures indicated in the table are inference trees for which we were unable to solve the generated constraints. Some of these inference trees are truly infeasible and therefore result in unsatisfiable constraints. Another reason for failure is the fact that we use a bounded model-checker to check for satisfiability, and no solution is possible within the bound that the solver uses.

Currently, our tool ignores the unsatisfiable constraints and proceeds with test-model generation from the remaining constraints. Additionally, the tool will give information about the constraints (or inference trees) for which it failed to generate test-models. This information will assist us in evaluating the performance of the tool. For instance, in the case of trees with depth 3, we have manually verified that all the failures reported by the tool are indeed the result of infeasible trees. For example, the inference tree

<table>
<thead>
<tr>
<th>ACT</th>
<th>ACT</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHOICEL</td>
<td>SEQAX</td>
</tr>
</tbody>
</table>

is an infeasible inference tree in the meta-model of the process language introduced in Section 4.

We have encoded the syntax and semantics of a subset of Stateflow in our framework, and have generated test-cases using the syntax part. This is summarized in Table 2. For instance, we generated 1926 inference trees for depth 4, of which 1019 were failures. From the remaining 907 satisfiable inference trees, we have generated 108 test-cases. The reduction in the number of test-cases, compared to the number of satisfiable inference trees, is because of the fact that some test-cases satisfy the constraints of more than one inference tree.

The semantics part of Stateflow is far more complex, and we are in the process of experimenting with test-case generation for the same. Our experimentation is based on the formalization presented in [8].

To give a glimpse of the complexity of the Stateflow semantics, we show an example inference tree that we generated in Appendix A. This inference tree gives a derivation for the sequence of actions executed by a Stateflow machine in active state \( s_0 \). When the machine receives the event \( e_1 \) and the condition \( c_1 \) evaluates to True, it will execute the condition-action \( a_3 \) of the transition \( t_1 \), followed by the exit-action \( a_2 \) of the state \( s_0 \), followed by the transition-action \( a_4 \) of \( t_1 \), and finally followed by the entry-action \( a_5 \) of the state \( s_1 \).

We generated the constraint for this inference tree and obtained a test-case by solving it. The test-case consisted of the Stateflow model shown in Figure 5 and the sequence of actions: \( a_3, a_2, a_4, a_5 \).

During this experiment, we came across 5 discrepancies, which were traced back to errors [1] in the definitions of some of the semantic rules in [8], which we corrected in our encoding of the meta-model.

![Figure 5. The generated Stateflow test-model](image-url)

### Table 1. Test generation for process language

<table>
<thead>
<tr>
<th>Depth</th>
<th>No. of inference trees</th>
<th>Failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>225</td>
<td>183</td>
</tr>
</tbody>
</table>

### Table 2. Test generation for Stateflow syntax

<table>
<thead>
<tr>
<th>Depth</th>
<th>No. of inference trees</th>
<th>Failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>178</td>
<td>58</td>
</tr>
<tr>
<td>4</td>
<td>1926</td>
<td>1019</td>
</tr>
</tbody>
</table>

7. Conclusion and Future Work

We have presented a rigorous approach for testing model-processors. The test-models are generated by constructing inference trees using the meta-model of the modeling language that includes both syntactic and semantic definitions. This helps in generating test-models that cover not only the syntactic rules, but also the semantic subtleties of the language, which is a novel aspect of our methodology. We have also implemented a prototype tool that uses Alloy for solving the constraints.

There are a number of directions for further work in this area. First and foremost, we are validating the ideas pre-
sented in this paper on industrial code-generators. Also, from the industrial applicability point of view, we are working on improving the flexibility of the tool to generate different kinds of test-models. The presentation in this paper concentrates on generating a test-model and a sequence based test-input, and we have chosen meta-models that express these artifacts directly. In real applications, the kind of test-artifact to be generated for a given modeling language will depend very much on the functionality of the model processing tool being tested. We are investigating ways of parameterizing our tool to allow the generation of different test-artifacts without changing the basic meta-model provided for a modeling language.

We are also addressing the scalability issues of the methodology. The current bottleneck is the constraint solving process. The constraints generated from the semantic part of the Stateflow meta-model result in fairly large Alloy formulae, some of which take a very long time to solve. We have used a very simplistic translation of predicates into the Alloy language, and we are investigating how this translation can be improved. Also, Alloy may not be the most suitable language for solving constraints over algebraic terms. We are investigating whether other constraint solving systems could be more useful for our purposes. On a related note, we are also investigating whether a different language for expressing the constraints, for instance set-constraints [10], could be used effectively in our methodology.

Finally, we are also attacking the problem of scalability from an application specific perspective, and are investigating means of improving the test specification using application specific criteria. The hope is that application specific criteria would reduce the combinatorial explosion in the generation of inference trees. The challenge in this is to find a generic means of expressing these application specific criteria.

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 and suggestions.

References

[1] Correspondence with Grégoire Hamon.
A. Example Inference Tree for Stateflow

\[
t_0 \ ::= \ (e_0, \varnothing, \varnothing, s_2, s_0)
\]
\[
t_1 \ ::= \ (e_1, c_1, a_3, a_4, s_2, s_1)
\]
\[
s_0 \ ::= \ ((a_0, a_1, a_2), \emptyset r(\emptyset, s_2, s_0, \emptyset_T, \{\}), \emptyset_T, t_1, \{\})
\]
\[
s_1 \ ::= \ ((a_5, a_6, a_7), \emptyset r(\emptyset, s_2, s_1, \emptyset_T, \{\}), \emptyset_T, T_2, \{\})
\]

\[
R_1 \ ::= \ e = e_1, \ e, D_0 \vdash c_1 \rightarrow true \quad e, D_0 \vdash a_3 \leftrightarrow D_1
\]
\[
e, D_0 \vdash t_1 \xrightarrow{D_1} Fire(s_2, s_1, a_4) \quad \text{(t-FIRE)}
\]

\[
R_2 \ ::= \ R_1
\]
\[
e, D_0, J \vdash t_1, T_1 \xrightarrow{D_1} Fire(s_2, s_1, a_4) \quad \text{(T-FIRE)}
\]

\[
R_3 \ ::= \ R_2
\]
\[
e, D_1 \vdash Or(\emptyset, s_2, s_0, \emptyset_T, \{\}) \xrightarrow{D_1} Or(\emptyset, s_2, s_0, \emptyset_T, \{\}) \quad e, D_1 \vdash a_2 \leftrightarrow D_2
\]
\[
e, D_2 \vdash a_4 \leftrightarrow D_3
\]
\[
e, D_0, J \vdash s_0 \xrightarrow{D_3} s_0, Fire(s_2, s_1, \emptyset) \quad \text{(SD-FIRE)}
\]

\[
R_4 \ ::= \ R_3
\]
\[
e, D_3 \vdash a_5 \leftrightarrow D_4
\]
\[
e, D_4, \emptyset_p \vdash Or(\emptyset, s_2, s_1, \emptyset_T, \{\}) \xrightarrow{D_4} Or(\emptyset, s_2, s_1, \emptyset_T, \{\})
\]
\[
e, D_3, \emptyset_p \vdash s_1 \xrightarrow{D_4} s_1 \quad \text{(SD-INIT)}
\]

\[
(tv = \text{No}) \lor (tv = \text{End}) \quad R_3 \quad R_4 \quad \text{(OR-INT-FIRE)}
\]
\[
e, D_0, J, tv \vdash Or(s_0, s_2, t_0, \emptyset_T, \{s_0 : s_0, s_1 : s_1\}) \xrightarrow{D_4} Or(s_1, s_2, t_0, \emptyset_T, \{s_0 : s_0, s_1 : s_1\}), \text{No}
\]