Behaviour Directed Testing of Auto-code Generators

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

General Motors R&D - India Science Lab
Bangalore
{p.sampath, rajeev.c, ramesh.s, shashidhar.kc}@gm.com

Abstract

This paper addresses the problem of testing auto-code generators. Auto-code generators take as input a model in a certain modeling language, and produce as output a program that captures the execution semantics of the input-model. We focus on the problem of test specification for the purpose of automatically generating a test-suite. We propose a novel technique for test specification based on the execution behaviour of models. We also propose an algorithm that uses such a behavioural test specification for directing test-case generation towards very specific behavioural patterns that we would like to exercise. We have implemented this technique, and have applied it for generating test-cases for a Stateflow auto-code generator.

1. Introduction

Auto-code generators play a critical role in addressing the increasing complexity of modern software engineering. This is especially true in the case of model-based software engineering, in which models are constructed in high-level domain-specific languages, and code is automatically generated from these models. Examples of some commonly used auto-code generators are The MathWorks Real-Time Workshop [17] and Rhapsody UML code generator [23]. Standard *nix tools like flex lexical analyzer generator [20] and yacc parser generator [33] are, in essence, also auto-code generators. Auto-code generators take as input a model in a language with specific syntax and semantics, and output a program that implements the semantics of the input-model. This paper addresses the problem of verifying the correctness of such auto-code generators.

The two main approaches for the verification of auto-code generators are formal verification and testing. Formal verification methods can give very strong guarantees about the correctness of auto-code generators. It requires mathematically proving the assertion:

\[ \forall m : \text{model}, \forall i : \text{input} \]
\[ \text{ModelExec}(m, i) \equiv \text{CodeExec} (\text{CodeGen}(m), i), \]

where, ModelExec and CodeExec are the execution semantics of model and code respectively, CodeGen denotes the code generated by the auto-code generator to be verified, and \( \equiv \) compares the outputs generated by a model with the outputs generated by the corresponding code. The double quantification over the space of models and the space of inputs makes formal verification a challenging task.

A simpler approach to formal verification is the translation validation approach [21, 19, 2, 11, 27]. Here, proof of the equivalence between a model and the generated code is attempted separately for each invocation of the code generator, rather than directly verifying the code generator itself. Given a model \( m \), this approach attempts to prove the assertion:

\[ \forall i : \text{input} \]
\[ \text{ModelExec}(m, i) \equiv \text{CodeExec} (\text{CodeGen}(m), i). \]

This approach has been shown to be effective for finite-state models, but is extremely difficult for complex modeling languages involving non-trivial data types. Also, this approach is not easily applicable to situations where the correspondence between a model and the generated code is not straightforward. One such example is flex [20], where the model is a sequence of regular expressions, while the generated code is heavily table-based.

In view of the above problems, we are interested in a testing based approach for verifying auto-code generators. Program testing takes the pragmatic approach of verifying that the program under test works correctly for a small subset of inputs. When the subset of inputs is chosen carefully so as to meet some coverage criteria, it gives a degree of confidence that the program works correctly on all possible inputs. Various coverage criteria have been proposed
Testing auto-code generators amounts to establishing

\[
\text{ModelExec}(m, i) \equiv \text{CodeExec(CodeGen}(m), i)
\]

for a subset of models and inputs. Note that a test-case for an auto-code generator includes besides the model \(m\), a set of inputs to \(m\) and the corresponding expected outputs from \(m\), which can be used for establishing the equivalence of \(m\) and the corresponding code\(^1\).

Our focus in this paper is on automatic generation of test-cases for auto-code generators. This is a complex task for many reasons. First, a careful selection of models having the desirable coverage over the syntactic and semantic structure of the modeling language needs to be done; coverage over modeling language is quite different from the coverage notions over programs, used in testing literature. Besides the models, appropriate inputs/outputs required for testing the equivalence of models and code need to be generated. Further, access to the internals of auto-code generators cannot be assumed, as they are usually third-party tools with no available source code.

In earlier work \([24, 25]\), we have presented a framework for testing auto-code generators. One input to this framework is a meta-model consisting of the syntax and semantics of the modeling language handled by the auto-code generator. Another input to the framework is a test specification. The test specification permits some form of syntactic or semantic specification of models, describing the modeling language constructs to be used in the models, usage of particular combination of constructs, depth of syntactic or semantic derivation trees, etc. Grammar based ATG techniques \([18, 30, 4, 15]\) were extended to generate test-cases from a given meta-model and test specification. We have used our framework for generating test-cases for non-trivial auto-code generators such as flex \([25]\).

Our experience with the framework indicated that the test specifications employed were inadequate. Test specifications such as the size of models, the depth of derivation trees etc. are at a low level and result in a large number of test-cases, many of which are redundant. Further, while testing auto-code generators for complex modeling languages, one needs to check whether certain high level behaviours and behavioural patterns of the models coded in the language are correctly translated by the auto-code generator. For example, while testing a Stateflow auto-code generator, we may like to generate test-cases that exercise the following behavioural pattern: execute a during action, followed by three condition actions, followed by an exit action, followed by a transition action, followed by an entry action. There is currently no method/tool that can take such a kind of behavioural specification and generate appropriate test-cases.

The aim of this paper is to propose a behavioural test specification language that enables the description of such high-level behavioural patterns, and to develop a behaviour-directed test-case generation method for auto-code generators. A behavioural test specification is a set of action sequences, given by a finite-state automaton over action symbols. These action symbols correspond to specific semantic behaviours. We also develop a test-case generation algorithm that takes as input an automaton over action symbols, and identifies one or more models and inputs that result in the execution of action sequences accepted by the automaton. Our experiments with this algorithm have been very encouraging.

In summary, the contributions of this paper are:

- We have formulated a novel test specification based on the behaviour of models coded in a modeling language: action sequences.
- We propose an algorithm for behaviour-directed test-case generation, based on action sequence specifications.
- We have implemented this test specification technique in our framework for testing auto-code generators. We discuss some results of applying our technique for generating test-cases for a Stateflow auto-code generator.

The rest of the paper is organized as follows: Section 2 provides background for the overall methodology of testing auto-code generators, using test specifications. Section 3 introduces behavioural test specification with illustrations, and details the formalization of such test specifications. Section 4 presents the algorithm for generating test-cases from a behavioural specification. Section 5 reports on the implementation of the algorithm and the experiments we carried out. Section 6 puts our work in the context of the literature, and Section 7 concludes the paper with some directions for future work.

2. Testing Auto-code Generators

In \([24]\) we proposed a methodology for testing auto-code generators, that is summarized in Algorithm 1. It takes as inputs

1. a meta-model consisting of the grammar for syntax and the inference system for semantics, and
2. a test specification that denotes the intent of the tester

\(^1\)One could imagine an intermediate approach in which we select a subset of models, but check the equivalence of a model and the auto-generated code using formal verification techniques.
Algorithm 1 has two major parts: test-case generation and test-case execution. Test-case generation consists of two main procedures:

- **generateTestGoals**: This function takes as inputs the meta-model and the test specification, and generates a collection of test-goals.

- **generateTestCases**: This is the core of the test-case generation process. Given a test-goal, it generates a set of models, inputs for these models and the corresponding expected outputs from these models.

```plaintext
// Generating the test-suite
1 goals ← generateTestGoals(meta-model, testspec);
2 foreach p ∈ goals do
3    set of (model, input, output) ← generateTestCases(p, meta-model);
4    add the set of (model, input, output) to testSuite;
5 end
// Executing the test-suite
6 foreach (model, input, output) ∈ testSuite do
7    code ← GenerateCode(model);
8    output' ← Execute(code, input);
9    if oracle(output, output') then pass else fail
10 end
```

**Algorithm 1**: Testing auto-code generators

### 2.1. Meta-model

This consists of both syntactic and semantic meta-models. The syntactic part $\Sigma$ is the context-free grammar of the modeling language, expressed as a tuple

$$\Sigma \equiv (N, T, S, \rho)$$

where $N$ and $T$ are the non-terminals and terminals respectively, $S \in N$ is the start non-terminal and $\rho$ is the set of production rules. These rules are of the form $n \Rightarrow (N \cup T)^*$, where $n \in N$.

The semantic meta-model is given as an inference system represented by a tuple

$$\Delta \equiv (\Sigma, J, R, \delta)$$

where $\Sigma$ is the context-free grammar for syntax (as explained above), $J$ is the set of judgement-types that represent the different kinds of conclusions that can be derived from the semantics, and $R \in J$ is the judgement-type of the root of the derivations. $\delta$ is a set of axioms and inference rules that capture the semantics of the modeling language. Inference rules are of the form

$$\frac{P_1 \ P_2 \ P_3 \ cond}{C}$$

where each $P_i$ is a premise judgement, $C$ is the conclusion judgement and $cond$ is a side-condition imposing various constraints on premises and conclusion. Axioms are special kinds of inference rules that do not have any premise.

### 2.2. Test Specification

This input to Algorithm 1 helps in guiding the search for test-cases. In prior work [24, 25], we used two kinds of test specifications: syntax based test specifications which include the well known grammar-based coverage criteria such as rule coverage and rule dependence coverage [14]; and semantics based test specifications which include the semantic counterparts of rule coverage and rule dependence coverage.

#### 2.3. Test-case

A test-case in our framework is a tuple

$$\langle M, s, i, o \rangle$$

where $M$ is a model, $s$ and $i$ are inputs to the model ($s$ is the test-initialization input and $i$ is the actual input), and $o$ is the expected output of the model. We assume that the model and the code accept inputs and produce outputs in the same format. Otherwise, functions for translating the inputs (and outputs) from one format to the other will have to be included as part of the test-case.

### 3. Behavioural Test Specification

We propose a new test specification language that enables a tester to directly specify the behaviours and behavioural patterns of the models that he/she would like to
check for correct translation by an auto-code generator. We first motivate and illustrate such test specifications informally with examples, and then present their formalization.

Consider the Stateflow [17] modeling language, which is a variant of hierarchical state-machines as implemented in the Matlab/Simulink tool suite. It is used quite extensively for developing control algorithms, in conjunction with Simulink.

A Stateflow model may perform different types of actions during execution: entry, exit and during actions by states; condition and transition actions by transitions. These actions are executed in some specific order; e.g., entry action of a state is executed before those of its children states. It is often found that bugs in implementations of auto-code generators cause the execution of actions in wrong order. We would like to identify such bugs by testing. The proposed test specification language enables the tester to specify the order of execution of actions. Our test-case generation algorithm extracts models and inputs exhibiting such behaviour. For example, we might want to test generating models and inputs would follow similar lines as above. A Stateflow model that can generate this sequence is given in Figure 1b.

Now, instead of a single (entry), suppose the given sequence is \(\langle\text{entry}\rangle\langle\text{entry}\rangle\langle\text{entry}\rangle\). Our reasoning for generating models and inputs would follow similar lines as above. A Stateflow model that can generate this sequence is given in Figure 1b.

Note that there may be no test-case that satisfies a behavioural specification. For example, consider the specification \(\langle\text{during}\rangle\langle\text{cond}\rangle\langle\text{cond}\rangle\langle\text{exit}\rangle\langle\text{trans}\rangle\langle\text{trans}\rangle\langle\text{entry}\rangle\). This sequence contains two consecutive (trans) actions. Such a sequence is not possible, given the semantics of Stateflow as per version V4.1. Hence, our algorithm will not generate any test-case. Now, if the above sequence is modified to \(\langle\text{during}\rangle\langle\text{cond}\rangle\langle\text{cond}\rangle\langle\text{exit}\rangle\langle\text{trans}\rangle\langle\text{entry}\rangle\), our algorithm will generate a set of test-cases. One of them consists of the model in Figure 1c (Note that the transition from \(s_2\) to \(s_3\) has two segments connected by a junction; otherwise, two consecutive (cond) actions cannot be generated), and an input event \(e_2\). Our method also generates a test-initialization input event \(e_1\) to awaken the chart and drive it to a configuration where the state \(s_2\) is active. Now, on providing the input event \(e_2\), the expected output sequence of actions is \(\langle a_2\ a_10\ a_12\ a_6\ a_13\ a_7\rangle\).

One could have multiple sets of action symbols for a single modeling language. For example, we might want to test in detail the behaviour of transitions in Stateflow models. In this case, the set of action symbols may be \(\langle\text{evt-match}\rangle\) and \(\langle\text{evt-no-match}\rangle\) representing the comparison of an input event with the event labeling a transition: \(\langle\text{cond-true}\rangle\).
and \(\langle\text{cond-false}\rangle\) representing the evaluation of a condition; \(\langle\text{cond-act}\rangle\) and \(\langle\text{trans-act}\rangle\) representing the execution of condition and transition actions respectively; \(\langle\text{fail}\rangle\) and \(\langle\text{back-track}\rangle\) representing a transition segment failing (because of event not matching or condition evaluating to false) and back-tracking (because of a terminal junction), respectively. Now, a sequence \(\langle\text{evt-match}\rangle\langle\text{cond-true}\rangle\langle\text{cond-act}\rangle\langle\text{event}\rangle\langle\text{back-track}\rangle\) \(\langle\text{cond-true}\rangle\langle\text{back-track}\rangle\) \(\langle\text{cond-true}\rangle\langle\text{back-track}\rangle\) \(\langle\text{cond-false}\rangle\) \(\langle\text{back-track}\rangle\) \(\langle\text{fail}\rangle\) and back-tracking (because of event not matching or condition evaluating to false) and back-tracking (because of a terminal junction), respectively. Now, a sequence \(\langle\text{evt-match}\rangle\langle\text{cond-true}\rangle\langle\text{cond-act}\rangle\langle\text{event}\rangle\langle\text{back-track}\rangle\) \(\langle\text{cond-true}\rangle\langle\text{back-track}\rangle\) \(\langle\text{cond-true}\rangle\langle\text{back-track}\rangle\) \(\langle\text{cond-false}\rangle\) \(\langle\text{back-track}\rangle\) \(\langle\text{fail}\rangle\) and back-tracking (because of event not matching or condition evaluating to false) and back-tracking (because of a terminal junction).

As can be seen from the above illustrations, the reasoning for generating test-cases is quite non-trivial. Performing these reasoning steps manually is very error prone. In the sequel, we formalize behavioural test specification and then present an algorithm for automating the test-case generation process.

3.1. Formalizing Behavioural Test Specification

A behavioural test specification is given as a finite-state automaton that accepts a set of strings over a behavioural alphabet. The use of automata for test specifications is common in the testing literature [12]. An automaton specifies a regular language over its alphabet. It also provides a number of structural cues for identifying the test-goals: state coverage, transition coverage, etc. In this paper, we consider the coverage of final states of the specification automaton.

A behavioural alphabet \(\mathcal{B}\) consists of symbols for various semantic actions of the models in a modeling language, which must be translated faithfully by the auto-code generator. In this paper, we consider a behavioural alphabet consisting of 5 symbols for the different types of actions that can be executed by a Stateflow model: \(\langle\text{entry}\rangle\), \(\langle\text{during}\rangle\), \(\langle\text{exit}\rangle\), \(\langle\text{cond}\rangle\) and \(\langle\text{trans}\rangle\).

The link between a behavioural alphabet and the semantics of the modeling language is provided by a behavioural grammar. A behavioural grammar is a context-free grammar given by

\[
\Sigma^t = (J, \mathcal{B}, R, \rho^t)
\]

where the non-terminals are the judgement-types \(J\) in the semantic meta-model, the terminals are the behavioural alphabet \(\mathcal{B}\), and the source non-terminal is the root judgement-type \(R\) in \(\Delta\). The production rules \(\rho^t\) give the rules for deriving different behavioural sequences. They are derived from the set of inference rules \(\delta\) in \(\Delta\). Each inference rule \(r_{inf} \in \delta\) gives rise to a production rule \(r_{grm} \in \rho^t\). The LHS non-terminal of \(r_{grm}\) is the judgement-type of the conclusion of \(r_{inf}\). The RHS of \(r_{grm}\) is a string over terminals and non-terminals, such that the non-terminals correspond to the judgement-types of the premises of \(r_{inf}\).

For example, consider an inference rule in the semantic meta-model of Stateflow [22]:

\[
\begin{array}{cccccc}
\text{TransFail} & \text{TransFail} & \text{StateExit} & \text{StateEntry} & (\text{OR-Dur}) \\
\text{StateDur} & \\
\end{array}
\]

This rule represents the behaviour of an OR-state whose external and inner transitions failed (first two premises), and one child state exited and another one entered by firing some transition (third and fourth premises). In this case, the OR-state will execute its \(\langle\text{during}\rangle\) action. The corresponding production rule in \(\Sigma^t\) is:

\[
\begin{array}{cccc}
\text{StateDur} & \Rightarrow & \text{TransFail} & \langle\text{during}\rangle & \text{TransFail} \\
& & \text{StateExit} & \langle\text{trans}\rangle & \text{StateEntry} \\
\end{array}
\]

where \(\langle\text{during}\rangle\) and \(\langle\text{trans}\rangle\) are terminals from the behavioural alphabet. This production rule captures the fact that the inference rule OR-Dur corresponds to executing the actions of the failed external transition\(^2\) (first TransFail non-terminal), followed by the \(\langle\text{during}\rangle\) action of the OR-state, followed by the actions of the failed inner transition (second TransFail non-terminal), followed by the exit actions of the exited states (StateExit non-terminal), followed by the \(\langle\text{trans}\rangle\) action of the successful transition, followed by the exit actions of the entered states (StateEntry non-terminal).

Behavioural alphabet and behavioural grammar would be specified as part of the design of the test generation framework for an auto-code generator. Note that in general, such a framework would contain multiple behavioural alphabet-behavioural grammar pairs, giving a tester the flexibility to test different aspects of code generation. The design of behavioural alphabet and behavioural grammar needs thorough knowledge of the semantics of the modeling language and the encoding of semantics using inference rules in our framework. But once such a behavioural alphabet is designed, it can be used very effectively by a tester not having deep knowledge of semantics or inference rule formalisms. Application domain knowledge will enable the tester to provide behavioural test specifications that will give sufficient confidence in the use of the auto-code generator under test in that particular domain.

Note that the behavioural grammar abstracts out the side-conditions of inference rules, since it represents these rules as context-free grammar production rules. Hence, each valid inference tree can be represented by a derivation in \(\Sigma^t\); but, the converse does not necessarily hold. We will elaborate on the use of behavioural grammars in Section 4.

\(^2\)Though the transition failed, some segments in the transition may have fired, executing their \(\langle\text{cond}\rangle\) actions.
4. Test-case Generation

In this section we describe an algorithm for generating test-cases from a behavioural test specification. This algorithm takes as inputs the meta-model and a test-goal, and generates a set of test-cases (generateTestCases in Algorithm 1). Our algorithm is generic and does not depend on the specifics of any modeling language. All language specific information is captured as part of the meta-model that is given as input to the algorithm. In order to do this, we extend the meta-model $\Delta$ by including $\Sigma^2$: $\Delta \equiv (\Sigma, J, R, \delta, \Sigma^2)$. A behavioural test specification is given as a finite-state automaton $A$ over the behavioural alphabet. Test-goals in this case can be obtained by enumerating the final states of the automaton, i.e., a test-goal is a pair $(A, f)$ consisting of the automaton and a final state. Given a meta-model and a test-goal, the test generation algorithm has to search for valid semantic derivations that give rise to the behavioural strings identified by the test-goal. The test-case generation algorithm is presented in Algorithm 2.

**Input**: Behavioural test-goal $(A, f)$
**Input**: Meta-model $\Delta \equiv (\Sigma, J, R, \delta, \Sigma^2)$
**Output**: Test-cases

```latex
\begin{algorithm}
1. testSuite $\leftarrow \emptyset$;
2. init $\leftarrow$ Init($A$);
3. $A^* \leftarrow \text{pre}^*(\Sigma^1, A)$;
4. if there is no edge $\text{init} \xrightarrow{R} f$ in $A^*$ then
5. \hspace{1cm} return $\emptyset$;
6. end
7. derivTrees $\leftarrow \text{getDerivationTrees}(\Sigma^1, A^*, \text{init}, f, R)$;
8. foreach $t \in$ derivTrees do
9. \hspace{1cm} infTree $\leftarrow \text{GrammarToInference}(t, \Delta)$
10. \hspace{1cm} if isValid(infTree) then
11. \hspace{1.5cm} testSuite $\leftarrow$ testSuite $\cup$ SemToSyn(infTree);
12. end
13. end
14. return testSuite;
\end{algorithm}
```

Algorithm 2: generateTestCases

We first pre-process the specification automaton $A$ using the $\text{pre}^*$ algorithm [3], on line 3. The essence of $\text{pre}^*$ algorithm is as follows: given a regular set $L$ of strings over the terminals of a context-free grammar, its predecessors (i.e., strings from which some string of $L$ can be derived by repeated application of production rules of the context-free grammar) also form a regular set [3]. Note that predecessors are strings over the non-terminals and terminals of the context-free grammar. Given a context-free grammar $\Sigma^3$ and an automaton $A$ recognizing a regular set $L$ of strings over the terminals of $\Sigma^3$, the $\text{pre}^*$ algorithm constructs an automaton $A^*$ that recognizes the predecessors (with respect to the production rules of $\Sigma^3$) of $L$. The transitions in $A^*$ are labeled by terminals and non-terminals of $\Sigma^3$. The necessary condition for a string $s \in L$ to be derivable by the production rules of $\Sigma^3$ is the existence of a transition in $A^*$, labeled by $R$ (the start non-terminal of $\Sigma^3$), from the initial state of $A^*$ to the final state accepting $s$. Also, we can use the structure of $A^*$ to obtain the derivation of $s$.

Now, Algorithm 2 has three main steps that will be elaborated in the following sections:

- **getDerivationTrees** This generates the set of derivations in the behavioural grammar $\Sigma^2$ that can give rise to some behavioural string identified by $(A, f)$.

- **GrammarToInference** Here we generate a set of semantic inference trees, using the fact that the behavioural grammar approximates the inference rules. Note that we then retain only valid inference trees.

- **SemToSyn** This is used to generate test-cases from valid inference trees.

4.1. Generating Grammar Derivations

Algorithm 3 (getDerivationTrees) can be used for generating grammar derivations. The inputs to this algorithm are the behavioural grammar $\Sigma^1$, $\text{pre}^*$ automaton $A^*$, a pair of states in $A^*$, and a non-terminal $S$ in $\Sigma^3$ to be used as the root non-terminal of the resultant derivation.

The overall idea of getDerivationTrees algorithm is to identify all derivations that can result in a string labeling a path between the given pair of states $q_1$ and $q_2$ in $A^*$. The algorithm first identifies all production rules in $\Sigma^3$ whose LHS non-terminal is $S$ (line 3). For each such production $S \Rightarrow \beta$, line 5 identifies the set of paths labeled $\beta$, between $q_1$ and $q_2$. Note that in general there will be multiple such paths, representing different stages in the derivation of $\beta$ using the $\Sigma^2$ grammar. For each such path, the algorithm then identifies the edges labeled by non-terminals in $\Sigma^2$ (NTEdges on line 7), and recursively calculates (line 10) the sub-derivation trees (subtrees) for these non-terminals. These sub-derivation trees are then extended with the production rule $S \Rightarrow \beta$ as the root derivation (line 12).

4.2. Generating Semantic Inference Trees

The next step is to use GrammarToInference to generate semantic inference trees from the grammar deriva-
tion trees. A derivation tree in $\Sigma^d$ is a tree whose internal nodes are judgement-types and leaf nodes are behavioural alphabets. Each derivation step is according to a production rule of $\Sigma^d$. We have shown in Section 3.1 that $\Sigma^d$ is an approximation of semantic inference rules. In particular, the production rules of $\Sigma^d$ are in one-to-one correspondence with the inference rules in the semantic meta-model. Therefore, given a grammar derivation, GrammarToInference can construct an inference tree by using this correspondence. For example, given the grammar derivation of a behavioural string $b_1 b_2 b_3 b_4 b_5$ (shown as a tree):

\[
\begin{array}{c}
\frac{b_1}{J_1} (\text{Axiom 1}) \\
\frac{b_2 b_3}{J_2} (\text{Axiom 2}) \\
\frac{b_4}{J_1} (\text{Axiom 1}) \\
\frac{J_3}{J_4} (\text{Rule 1}) \\
\end{array}
\]

we can construct an inference tree of the form

\[
\begin{array}{c}
\frac{\text{Axiom 2}}{} \\
\frac{\text{Axiom 1}}{} \\
\frac{\text{Rule 1}}{} \\
\end{array}
\]

The inference trees created from the derivation trees are not necessarily valid: the constraints specified by the side-conditions of the inference rules in the inference tree have to be checked for satisfiability. If these side-conditions are unsatisfiable, the grammar derivation does not correspond to a valid semantic derivation.

4.3. Generating Test-cases

The final step in the method is to generate test-cases from valid inference trees. The function SemToSyn collects the side-conditions in a given inference tree. These side-conditions express constraints on the variables in the judgements of the inference rules. Since these variables represent various syntactic elements in the modeling language, a satisfying assignment to them (obtained by solving the constraints) can be used to construct a model and its input/output sequence. In principle, this is a generic technique and can be applied for any language and meta-model.

In our current tool, however, we have an implementation of SemToSyn for each language that we have considered as case-study. The main reason behind this is that in our experiments with off-the-shelf satisfiability solvers, we were unable to use language specific information to speed-up the constraint solving. This introduced scalability issues when dealing with large case-studies such as Stateflow. Our custom implementations of SemToSyn efficiently calculates models, inputs and expected outputs from semantic inference trees, by making use of language specific information about syntax and semantics.

5. Implementation and Experimentation

We have developed a framework for testing auto-code generators, following Algorithm 1. As part of the framework, we have implemented the algorithms for test-case generation using syntactic test specifications, semantic test specifications and behavioural test specifications.

We have considered a subset of the Stateflow language for test-case generation experiments. This subset is comparable to the one presented in [10]. However, we have identified a number of issues [6] with the formalization of Stateflow semantics in [10], and so we base our work on an independent formalization [22]. We have omitted our formalization of Stateflow meta-model from this paper due to space constraints. The non-triviality of our meta-model is evident from its complexity: the syntactic meta-model has 11 production rules; the semantic meta-model has 12 judgement-types and 52 inference rules.

The test-case generation experiments were carried out on a laptop with Intel Centrino Duo processor (2.2 GHz) and 2GB RAM, running Windows XP.

The syntactic test specifications included rule coverage (which ensures that every production rule in the syntactic meta-model is exercised at least once by the test-cases) and rule dependence coverage (which ensures that all possible executions of a production rule in the syntactic meta-model are exercised at least once by the test-cases).
dependencies between the production rules are exercised). Note that a test-case for syntactic specification consists of just a model, and does not include any information about its behaviour (input/output sequence). The main purpose of syntax based testing is to ensure that all syntactic constructs in the modeling language are accepted by the auto-code generator.

The experimentation with semantic test specifications generated a large number of test-cases. Fairly aggressive heuristics were used to prune down the search space of the algorithms, to generate results in a reasonable amount of time. We were unable to achieve 100% coverage of the test-goals for inference rule coverage and inference rule dependence coverage. However, the generated test-cases were of good quality, exercising corner cases in the semantics: a number of test-cases looked synthetic and were representing some unusual usages of the Stateflow language. The results related to syntactic and semantic test specifications are summarized in Table 1.

As opposed to syntactic and semantic specifications, test-case generation was extremely fast with behavioural specification. The main reason is that behavioural specification considerably limits the search in the semantic space.

Our tool is able to handle fairly long sequences of action-types as test specifications, resulting in test-cases that are considerably more complex than those generated by semantic specifications. For example, consider the following sequence of length 16: 

\[((\text{during}) \langle \text{cond} \rangle \langle \text{cond} \rangle \langle \text{exit} \rangle \langle \text{trans} \rangle \langle \text{entry} \rangle \langle \text{cond} \rangle \langle \text{cond} \rangle \langle \text{exit} \rangle \langle \text{trans} \rangle \langle \text{entry} \rangle \langle \text{trans} \rangle \langle \text{entry} \rangle \langle \text{entry} \rangle \langle \text{trans} \rangle \langle \text{entry} \rangle \langle \text{entry} \rangle \langle \text{trans} \rangle \langle \text{entry} \rangle \langle \text{entry} \rangle \langle \text{trans} \rangle \langle \text{entry} \rangle \langle \text{entry} \rangle \langle \text{trans} \rangle \langle \text{entry} \rangle \langle \text{entry} \rangle)\]

We generated 6912 test-cases in 1s, for this specification. One of the test-cases consisted of the model in Figure 2a. This is a non-trivial one with hierarchical depth 3, and consists of both OR-states and AND-states. The test-initialization input event c1 will drive this model to a configuration in which the states s5, s7 and s9 are active. Now, on providing the input event c2, the model will execute the actions \langle a2 a5 a3 a5 a3 a5 a3 a12 a16 a8 a3 a2 a34 a22 a11 a3 a7 a37 a36 a28 \rangle. Here, all the three transitions fire and the new set of active states consists of s6, s8 and s10.

An interesting application of behavioural specification is in the regression testing of auto-code generators. For example, consider the change in the semantics of transitions, between Stateflow Versions V4.0 (R12) and V4.1 (R12.1) [28]. Version V4.0 (R12) required the execution of transition actions of all the transition segments in a chain of segments connected by junctions, when such a transition fires. This behaviour changed in Version V4.1 (R12.1), and required the execution of the transition action of only the last segment. Since we have formalized the semantics of Version V4.1, the action sequence \langle \langle \text{during} \rangle \langle \text{cond} \rangle \langle \text{cond} \rangle \langle \text{exit} \rangle \langle \text{trans} \rangle \langle \text{trans} \rangle \langle \text{entry} \rangle \rangle \rangle that models the behaviour of a transition chain of length two in Version V4.0 (note the double occurrences of \langle \text{cond} \rangle and \langle \text{trans} \rangle) fails to produce any test-case, indicating that such a behaviour is not possible with respect to the semantics of Version V4.1. However, modifying the action sequence to \langle \langle \text{during} \rangle \langle \text{cond} \rangle \langle \text{cond} \rangle \langle \text{exit} \rangle \langle \text{trans} \rangle \langle \text{entry} \rangle \rangle \rangle immediately generates 48 test-cases. The directed nature of behavioural specification makes it a very powerful tool for test-case generation for regression testing of auto-code generators.

Behavioural specifications can also be used to enhance other test specification methods, by directing the search performed by the test-case generation algorithm. This can be used effectively to combat combinatorial explosion problems. For example, one test-goal that we are unable to cover in reasonable time using inference rule coverage based test generation is the inference rule that describes an AND-state with children states, all executing their during actions. This is mainly because of combinatorial explosion. However, by augmenting this test-goal with a behavioural sequence such as \langle \langle \text{during} \rangle \langle \text{during} \rangle \langle \text{during} \rangle \langle \text{during} \rangle \rangle, we were able to generate 288 test-cases. One of them is shown in Figure 2b. The test-initialization input e1 will make the states s2, s3 and s4 active. Now the actual input e2 will result in all the states executing their during actions, and hence the output action sequence is \langle a2 a5 a8 a11 \rangle.

### Table 1: Test-case generation for Stateflow using grammar-based test specifications.

<table>
<thead>
<tr>
<th>Test specification</th>
<th>No. of test-goals</th>
<th>No. of covered test-goals</th>
<th>% coverage</th>
<th>No. of test-cases</th>
<th>Time taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>Production rule</td>
<td>11</td>
<td>11</td>
<td>100</td>
<td>41</td>
<td>1s</td>
</tr>
<tr>
<td>Production rule dependence</td>
<td>44</td>
<td>38</td>
<td>86.36</td>
<td>191</td>
<td>9s</td>
</tr>
<tr>
<td>Inference rule</td>
<td>52</td>
<td>47</td>
<td>90.38</td>
<td>282</td>
<td>11s</td>
</tr>
<tr>
<td>Inference rule dependence</td>
<td>704</td>
<td>506</td>
<td>71.88</td>
<td>2393</td>
<td>90s</td>
</tr>
</tbody>
</table>

6. Related Work

In the recent past, 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 [5, 13, 26, 32, 31, 9]. However, the ATG problem for auto-code generators distinguishes itself because of the fact that test-cases are programs (or models) with rich syntactic and semantic structure. In the following, we discuss ATG methods that are applicable, either directly or potentially, for auto-code
White-box and grey-box coverage testing [1, 8, 29] have been used for testing auto-code generators. They rely on the knowledge of transformation rules implemented in the auto-code generator to be tested. However, this requires either the availability of the source code of the tool or at least the underlying implementation details, which is most often not the case for third-party tools. In contrast to this, our approach requires only the specification of the auto-code generator to be tested, i.e., the syntax and semantics of the modeling language.

A number of techniques have been proposed in the literature on specification-based testing and, in particular, its instance, grammar-based testing [18, 30, 4, 15]. Grammar-based testing deals only with those aspects of an auto-code generator 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 auto-code generator. Although we incorporate some ideas from grammar-based testing in our method, our focus is on semantics. We not only generate models, but also generate specific inputs to these models for testing subtle semantic interactions, which would otherwise require impractically deep syntactic coverage of the grammar to be generated.

Another related work is that of [7], which focuses on testing a refactoring engine. This work presents an imperative language for test specification. Here the semantics of transformations is encapsulated into test oracles, and is not used for test-case generation. In comparison, our test specifications are declarative and describe the behaviours to be tested. Our algorithm is essentially semantics directed, and generates test-cases that can detect subtle semantic differences between models and the generated code.

Recently, Majumdar and Xu have proposed directed test generation using symbolic grammars [16]. This is a white-box approach and performs symbolic execution of the code to generate test-cases. The novelty here is the use of a symbolic grammar of the input to direct the symbolic execution of the code. We also introduce the notion of a symbolic grammar; however, we use it to represent the outputs of models rather than inputs. This allows us to formulate very interesting test specifications based on the behaviour of models, which is not possible in [16].

### 7. Conclusion and Further Work

We have proposed a novel test specification for verifying auto-code generators, and an algorithm for test-case generation based on such specifications. We have also demonstrated our technique by applying it on a significant subset of the Stateflow modeling language. The proposed method is robust and scalable, and is applicable to auto-code generators for sophisticated modeling languages.

We found the technique of using behaviours of models as test specification to be a powerful one, in addition to being intuitive. We have presented a few interesting applications of this kind of test specification. We intend to pursue investigations regarding more applications. We also plan to improve the expressive power of the behavioural specification language.

We are also working on simplifying the effort required for formulating the meta-model of a modeling language, as this currently requires a deep knowledge of the modeling language, and also, considerable expertise in the inference rule formalism used by us.
Acknowledgments

We would like to thank Srihari Sukumaran and the anonymous referees for their constructive comments and suggestions.

References

[6] Correspondence with Grégoire Hamon.