CoGenTe: A Tool for Code Generator Testing

A. C. Rajeev  Prahladavaradan Sampath  K. C. Shashidhar ∗  S. Ramesh
India Science Lab, General Motors Global Research and Development
Bangalore, India
{rajeev.c, p.sampath, ramesh.s}@gm.com, kodambal@iese.fhg.de

ABSTRACT
We present the CoGenTe tool for automated black-box testing of code generators. A code generator is a program that takes a model in a high-level modeling language as input, and outputs a program that captures the behaviour of the model. Thus, a code generator’s input and output are complex objects having not just syntactic structure but execution semantics, too. Hence, traditional test generation methods that only handle syntax are not effective in testing code generators. CoGenTe amends this by incorporating various coverage criteria over semantics. This enables it to generate test-cases with a higher potential of revealing subtle semantic errors in code generators. CoGenTe has uncovered such issues in widely used real-life code generators: (i) lexical analyzer generators Flex and JFlex, and (ii) The MathWorks’ simulator/code generator for Stateflow.

Categories and Subject Descriptors
D.2.4 [Software Engineering]: Software/Program Verification; D.2.5 [Software Engineering]: Testing and Debugging; D.3.4 [Programming Languages]: Processors

General Terms
Experimentation, Languages, Reliability, Verification

Keywords
Automatic Test Generation, Black-box Testing, Code Generators, Semantic Coverage Criteria

1. INTRODUCTION
Code generators play a critical role in the model-based development of complex software systems. But they are prone to implementation errors owing to the complexity of the syntax and semantics of modeling languages. Also, when the modeling language evolves (which is often the case with most domain specific ones), the related code generators have to be updated to handle the new/altered constructs. In addition, the source code of third-party code generators may not be available for inspection. These points motivate the need to rigorously validate code generators, especially when they are used in developing safety-critical software [1, 3].

CoGenTe generates test-cases to cover not only the syntactic aspects of a translation, but its complex semantic aspects, too. Syntactic coverage ensures that models containing certain syntactic constructs are accepted by a code generator. A test-case in this case is just a model; it does not include any information about the model’s behaviour. In comparison, semantic coverage ensures that certain behaviours exhibited by the models are translated correctly. Here, a test-case consists of a model, inputs to drive the model (so that it exhibits the behaviour being tested), and the corresponding outputs from the model. These inputs/outputs can be used to compare the behaviour of the model and the corresponding program generated by a code generator. This integrated generation of inputs/outputs for models is the main innovation in CoGenTe. This aspect, along with the provision of semantic coverage criteria, distinguishes CoGenTe from grammar-based testing tools (cf. Geno [9]) and related, but white-box, testing tools (cf. CESE [11]). We refer to our previous papers [13, 15, 14] for more discussion on related work.

2. CoGenTe ARCHITECTURE
CoGenTe has two components (refer to the following figure): Test Generator (TG) and Test Harness (TH). TG is a generic component that takes two inputs: (i) syntactic and semantic meta-model of a modeling language expressed using inference rules and (ii) a test specification in the form of a coverage criterion over the meta-model, and generates a test-suite that can be used to test any code generator for this language. TH encapsulates the details of how a particular code generator takes inputs and generates outputs, and is therefore specific to the code-generator-under-test (CGUT).

Since we represent semantics using inference rules, semantic coverage can be expressed in terms of coverage over these rules; behaviours appear as patterns in the inference trees built from the rules. CoGenTe employs the following coverage criteria [14]: (i) depth of inference given as an integer n (ii) rule coverage given as a set of rules (iii) rule dependency coverage given as a set of rule pairs, and (iv) behavioural coverage given as a high-level behavioural pattern.

TG has three components: Inference Tree Generator (ITG), Constraint Generator (CG) and Constraint Solver (CS). They have been implemented in Standard ML. ITG identifies a set of test-goals corresponding to the given coverage criterion, and generates –using algorithms like IPO [10]– a set of inference trees covering these test-goals [14]. CG extracts the...
side-condition predicates of the rules in a tree, and CS computes a satisfying assignment which is then used to construct a test-case [13, 14]. As part of CS, we have modules based on Alloy [8], Yices [7] and custom-built solvers. The input to TH is a test-suite generated by TG. Given a test-case, TH first represents the model in CGUT's input format. It then invokes CGUT to generate the corresponding program, and feeds the executable of this program with the inputs. The resulting outputs are compared with the expected outputs to identify the success or failure of the test-case.

3. TWO CASE STUDIES

Lexical Analyzer Generators (LAGs): A LAG takes a list $R$ of regular expressions as input, and outputs a lexical analyzer $LA$. Given a string $s$ as input, $LA$ generates a token sequence $T$ corresponding to the sub-strings of $s$ that match the regular expressions in $R$. Test-suites generated by CoGenTe, using the semantics in [15], identified issues that match the regular expressions in the lexical analyzers/verifiers.

The syntax and semantics of a language (and is independent of the actual transformations done by the tool-under-test), it can be used to test not just code generators but any model processing tool such as syntax checkers, optimizers and analyzers/verifiers.

4. CONCLUSION

We have presented the CoGenTe tool for meta-model based testing of code generators. It incorporates the lessons learned from our work on testing of code generators [13, 15, 14] into a fully automated, scalable, end-to-end tool with a variety of syntactic and semantic coverage criteria. For future work, we note that since CoGenTe requires only a specification of the syntax and semantics of a language (and is independent of the actual transformations done by the tool-under-test), it can be used to test not just code generators but any model processing tool such as syntax checkers, optimizers and analyzers/verifiers.

5. REFERENCES