Testing-based compiler validation for synchronous languages

Pierre-Loïc Garoche[1] and Falk Howar[2] Temesghen Kahsai[2]
and Xavier Thirioux[3]

[1] ONERA
[2] NASA Ames / CMU
[3] IRIT

Abstract. In this paper we present a novel lightweight approach to validate compilers for synchronous languages. Instead of verifying a compiler for all input programs or providing a fixed suite of regression tests, we extend the compiler to generate a test-suite with high behavioral coverage and geared towards discovery of faults for every compiled artifact. We have implemented and evaluated our approach using a compiler from Lustre to C.

1 Introduction

In the safety critical domain it is common to verify (safety) properties of systems. Usually proofs for these properties are established at the level of source code or formal models. Source code and/or models are compiled to executables for some target platform. This compilation may invalidate already established verification results. It is thus of utmost importance to have a *trustworthy compilation process*. Existing approaches to trusted compilation fall into two categories. Either they aim at verifying the compiler itself (e.g., [7]), or they aim at validating the compiled output using a verified validator (e.g., [8]). Both exist in weaker variants, where verification is replaced by testing. There exists a body of work on generating test suites for verifying the correctness of a compiler (c.f., [3]). Testing the correctness of a compiled artifact is usually done by some form of specification-based testing (e.g., [9]).

The more rigorous approaches come at a high cost. Establishing the correctness of a compiler takes a lot of effort. Developing and verifying a validator is not less of an effort. Also, to be successful, a shared semantic basis is needed between the source and the target language. Testing the correctness of a compiler is difficult because the set of potential input programs to a compiler is potentially infinite and hard to sample in an automated fashion. Specification-based testing, on the other hand, is well understood and cheap (compared to the other approaches). It will, however, in many cases not uncover errors in a compiler: test-suites are geared towards finding violations of a specification and not towards uncovering faults in the translation of a program.

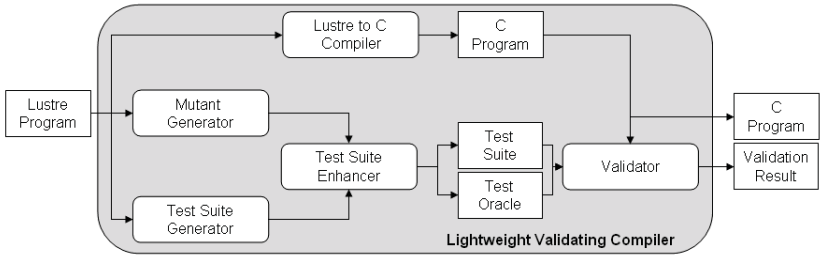

Fig. 1. Schematic view of a testing-based validating compiler from Lustre to C.

In this paper, we present a *lightweight* approach to compiler validation. We build our validating compiler upon specification-based testing, which we augment with a method for generating test cases targeting potential bugs in a compiler. Fig. 1 illustrates an overview of the developed framework. The lightweight validating compiler from Lustre to C consists of five main components: A modular *compiler* from Lustre to C, a *test suite generator* for Lustre programs, a grammar-based *mutant generator* for Lustre programs, a *test suite enhancer* – extends test suites with test cases for killing mutants – and a *validator*, which will execute the test suite on the compiled C program. The validator will use the input Lustre program as a test oracle. Test results are provided as output along with the compiled C program.

The central idea of this approach is twofold. On one hand, we verify that the compiler did not introduce any difference between the source program (Lustre) and the compiled version (C). We do this by generating automatically test suites from the source program (Lustre) based on MC/DC coverage criterion. On the other hand, we use mutations of the source program to simulate bugs in the compiler, and that test cases that differentiate mutations from the original program are likely to uncover errors in the translation of this program. We have implemented a prototypical version of such a lightweight validating compiler from Lustre to C. In this paper, we discuss the different components of our approach and present some results from a preliminary evaluation of our technique.

Outline. The paper is structured as follows: The next section, introduces the synchronous language Lustre and outlines our coverage-based test synthesis using bounded model checking (BMC). Section 3 presents how we reinforce test suites using mutants. Finally, Section 4 presents our preliminary experimental evaluations.

2 MC/DC Test Suites for Lustre Programs

Synchronous languages are a class of languages proposed for the design of "reactive systems" – systems that maintain a permanent interaction with a physical environment. Such languages are based on the theory of synchronous time, in which the system and its environment are considered to both view time with

some "abstract" universal clock. In order to simplify reasoning about such systems, outputs are usually considered to be calculated instantly [1]. In this paper, we will concentrate on Lustre [4]. Lustre combines each data stream with an associated clock as a means to discretize time. The overall system is considered to have a universal clock that represents the smallest time span the system is able to distinguish, with additional, coarser-grained, user-defined clocks. Lustre programs and subprograms are expressed in terms of *nodes*. Nodes directly model subsystems in a modular fashion, with an externally visible set of inputs and outputs. A *node* can be seen as a mapping of a finite set of input streams (in the form of a tuple) to a finite set of output streams (also expressed as a tuple). At each instant t, the node takes in the values of its input streams and returns the values of its output streams. Operationally, a node has a cyclic behavior: at each cycle t, it takes as input the value of each input stream at position or instant t, and returns the value of each output stream at instant t. Lustre nodes have a limited form of memory in that, when computing the output values they can also look at input and output values from previous instants, up to a finite limit statically determined by the program itself. Figure 2 describes a simple Lustre program: a node that every four computation steps activates its output signal, starting at the third step. The `reset` input reinitializes this counter.

```
node counter(reset: bool) returns (active: bool);
var a, b: bool;
let
  a = false -> (not reset and not (pre b));
  b = false -> (not reset and pre a);
  active = a and b;
tel
```

Fig. 2. A simple Lustre program.

Lustre programs can be compiled to main stream languages such as C or Java. Whereas initial compilation schemes of Lustre were computing a global automaton of the system [4], the approach of [2] relies on an object-like compilation of the program: each Lustre node call is seen as an instance of the generic declaration of the node. Our compiler from Lustre to C follows the latter approach.

A traditional technique to verify safety properties of synchronous languages is to use *SMT-based model checking* [6]. Such technique requires a predicate $\mathtt{M}[\tilde{s}, \tilde{in}, \tilde{s'}, \tilde{out}]$[4] describing the relationship between input flows $\tilde{in}$, output flows $\tilde{out}$ as well as internal states $\tilde{s}$ for the model M it represents. It also requires a predicate over initial states $\mathtt{M}_{init}[\tilde{s}]$ as well as a condition $\mathtt{C}[\tilde{s}, \tilde{in}, \tilde{out}]$ we are trying to meet at some time. A valid finite trace of length n for M would satisfy the following expression:

$$\mathtt{M}_{init}[\tilde{s}_0] \wedge \bigwedge_{i=0}^{n-1} \mathtt{M}[\tilde{s}_i, \tilde{in}_i, \tilde{s}_{i+1}, \tilde{out}_i] \wedge \mathtt{C}[\tilde{s}_n, \tilde{in}_n, \tilde{out}_n]$$

[4] We refer to the traditional definition of transition system in model checking techniques. A detailed description of a transition system for Lustre programs can be found in [5].

A satisfiability check using an SMT solver over this expression for a given n will produce a set of values for $\tilde{in}_i$, $\tilde{s}_i$ and $\tilde{out}_i$ for $i \in [0..n]$. In practice, tools unroll the transition relation one step at a time trying to meet the specific C condition. This can be done efficiently with an SMT solver by reusing previously computed states. We denote by $\mathsf{bmc}(M_{init}, M, C)$ such a typical BMC algorithm.

We generate test suites using Modified Condition/Decision Coverage (MC/DC) coverage criterion. The latter has been used as a test adequacy metrics for decades specially when testing critical software. We express MC/DC criteria as a predicates $\mathtt{C}[\tilde{s}, \tilde{in}, \tilde{out}]$ and use BMC to find test cases that satisfy these predicates. From a decision $P(c_1, \ldots, c_n)$ where the c_i's are a set of atomic conditions over the variables $\tilde{s}$, $\tilde{in}$ and $\tilde{out}$, we have to exert the value of each condition c_i with respect to the global truth value of P, the other conditions $c_{j\neq i}$ being left untouched. Precisely, we have to find two test cases for which, in the last element of the trace, c_i is respectively assigned to $False$ and $True$.
Remark: Bounded model checking may not be able to find a test case for some condition within an acceptable time limit[5]. In such cases, we conclude that the generated test suite does not reach the MC/DC coverage.

3 Reinforcing Test Suites via Mutation Testing

In the following, we denote by a *mutant* a mutated model or implementation where a single mutation has been introduced. The considered mutation, which is grammar based, does not change the control flow graph or the structure of the semantics but could either: (i) perform arithmetic, relational or boolean operator replacement; or (ii) introduce additional delay (*pre* operator in Lustre)[6], or (iii) negate boolean variables or expressions; or (iv) replace constants. Such generation of mutants has been implemented as an extension of our Lustre to C compiler. Once mutants are generated and the coverage-based test suite is computed, we can evaluate the number of mutants killed by the test suite. This evaluation is performed at the binary level, once the C code has been obtained from the compilation of the mutant. In this setting, the original Lustre model acts as an oracle, i.e., a reference implementation. Any test that shows a difference between a run of the original model compiled and a mutation of it, allows to kill this mutant.

In the literature, mutants are mainly used to evaluate the quality of a test suite. In our case, the motivation is different, we aim at providing the user with a test suite related to its input model. This test suite covers the model behavior in order to show that the compiler doesn't introduce bugs. We conjecture that a test suite achieving a good coverage of the code but unable to kill many mutants would not certify that the compiler did a good job. We thus introduce new tests to kill the un-killed (or resistant) mutants by the initial MC/DC-based test suite. Figure 3 illustrates the procedure to generate new test cases that allow to kill

[5] In our experiments the timeout for BMC was set to 100 secs.
[6] Note that, introducing additional delay could produce a program with initialization issues.

previously un-killed mutants. If the call to BMC (Line 3) does not terminate within the timeout (100 sec in our experiments), we don't introduce a new test case.

```
(1) proc genNewTest(M_init, M, M'_init, M') ≡
(2)   M'' := gen_mcdc_conds(M_init, M);
(3)   test := bmc(M_init[x̃ ∧ M'_init[ỹ],
(4)               M[x_{k-1}, in_{k-1}, x̃_k, out_{k-1}] ∧ M'[y_{k-1}, in_{k-1}, ỹ_k, out'_{k-1}],
(5)               ¬(out = out'))
(6)   print test
```

Fig. 3. A procedure to introduce new test cases in order to kill previously un-killed mutants.

4 Experimental Evaluation

We have implemented a prototypical version of the lightweight validating compiler from Lustre to C using the PKind model checker [6] and have performed a preliminary evaluation[7]. We ran the lightweight validating compiler on a set of 330 Lustre benchmarks. For every benchmark, we use MC/DC conditions to generate basic test suites. We then automatically generate a set of mutants (160 on average) for each benchmark.

Test suite generated via BMC guided by MC/DC conditions were able to achieve 100% MC/DC coverage on 10% of the overall benchmark. On the remaining 90% benchmarks, 87% of the MC/DC conditions could be satisfied, while 13% could not be satisfied. On average, test cases generated using BMC guided by the MC/DC conditions were able to kill 25.12% of the generated mutants (with a standard derivation of 0.26). Test cases generated using the procedure highlighted in Figure 3 increased the performance of these basic test suites by 56% (std. dev. of 1.06). In absolute terms, the combined test suites killed 34% of the mutants (std. dev. of 0.31). Figure 4 shows a view on this results. For every experiment we show the percentages of mutants killed by MC/DC generated test cases and additional mutants killed by test cases generated using genNewTest. The data set was sorted by the overall number of killed mutants.

Considering these results, the number of mutants that could not be killed is strikingly high. We believe that this is due to many behaviorally equivalent mutants being generated. This is supported by the relatively high number of cases for which we could not satisfy MC/DC conditions, indicating the existence of dead code in the examples. This will have to be substantiated in a future investigation.

[7] The prototypical implementation, benchmarks and results can be found at `https://bitbucket.org/lememta/nfm-14`.

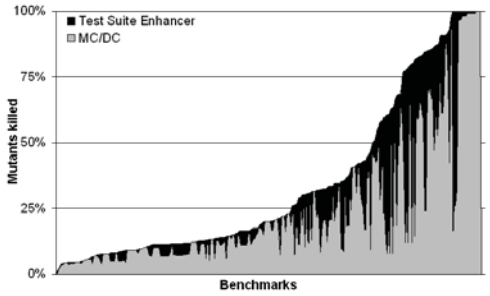

Fig. 4. Killed mutants per Benchmark. Ordered by percentage of mutants killed.

5 Future Work

As a next step we plan to assess the fault finding capabilities of the generated test suites and compare these to other methods for generating test suites (e.g., random testing). A more recent work by Whalen et. al extended MC/DC with a notion of *observability* (OMC/DC) [9]. Our approach is orthogonal to such, in principle any coverage criterion can be used to generate the initial test case. We plan to integrate the OMC/DC technique in our validating compiler. Moreover, we plan to perform experiments with seeded bugs in the Lustre to C compiler to confirm that the mutations that we selected on Lustre programs mimic the effects of potential bugs in a compiler. We also plan to investigate how the validation results can be quantified and provide an estimate for the trustworthiness of the compiler. Finally, we plan to extend this work to object oriented languages.

References

1. Benveniste, A., Berry, G.: The synchronous approach to reactive and real-time systems. In: Proceedings of the IEEE. pp. 1270–1282 (1991)
2. Biernacki, D., Colaço, J.L., Hamon, G., Pouzet, M.: Clock-directed modular code generation for synchronous data-flow languages. In: Flautner, K., Regehr, J. (eds.) LCTES. pp. 121–130. ACM (2008)
3. Boujarwah, A., Saleh, K.: Compiler test case generation methods: a survey and assessment. Information and Software Technology 39(9), 617 – 625 (1997)
4. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: Lustre: A declarative language for programming synchronous systems. In: POPL-87. pp. 178–188. ACM Press (1987)
5. Hagen, G., Tinelli, C.: Scaling up the formal verification of Lustre programs with SMT-based techniques. In: FMCAD-2008. pp. 109–117. IEEE (2008)
6. Kahsai, T., Tinelli, C.: PKind: a parallel k-induction based model checker. In: PDMC. EPTCS, vol. 72, pp. 55–62 (2011)
7. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (2009)
8. Necula, G.C.: Translation validation for an optimizing compiler. SIGPLAN Not. 35(5), 83–94 (May 2000)
9. Whalen, M., Gay, G., You, D., Heimdahl, M.P.E., Staats, M.: Observable modified condition/decision coverage. In: ICSE 2013. pp. 102–111. IEEE Press (2013)