Issues in Software Testing with Model Checkers

Vadim Okun Paul E. Black
National Institute of Standards and Technology
Gaithersburg, MD 20899
{vokun1,paul.black}@nist.gov

I. Generating software tests

Most software developers consider formal methods too hard and tedious to use in practice. Instead of using formal methods, developers test software. Model checking is a "light-weight" formal method to check the truth (or falsity) of statements. We use the SMV model checker as part of a highly automated test generation tool, which we hope will motivate practitioners to use formal methods more. For instance, an organization is more likely to expend the considerable effort to develop a formal specification if, with a little extra effort, it can also get tests. In this paper we present some approaches to use model checkers to generate tests.

Model checking is being applied to test generation and coverage evaluation [3], [4], [7]. In both uses, one first decides on a notion of what properties of a design must be exercised to constitute thorough testing. This notion leads to test criteria.

One applies the chosen test criteria to the specification to derive test requirements, i.e., a set of individual properties to be tested, represented as temporal logic formulas [2]. To generate tests, the requirements must be negative requirements, that is, they are considered satisfied if the corresponding formulas are inconsistent with the state machine. They must also be of a form that a single counterexample demonstrates the inconsistency (exhaustive enumeration is needed to show inconsistency of an existential requirement). For instance, if the criterion is state coverage, the negative requirements are that the machine is never in state 1, never in state 2, etc.

When the model checker finds an inconsistent formula, it produces a counterexample. Again, for state coverage, a counterexample gives stimulus to put the machine in state 1 (if it is reachable), another to put the machine in state 2, etc. Counterexamples are automatically turned into executable tests.

An alternative approach is developing a special tool, based on an existing model checker, to generate counterexamples that have properties, such as fault visibility (Section V), especially useful for test generation. This tool could benefit from most of the technology of existing model checkers. We are not yet pursuing this line of research due to lack of resources.

II. Abstraction for testing

Since complete detailed designs are typically too big to check, abstractions, or reductions, are used. Abstractions for test generation can use a different soundness rule [1] than for property checking. Informally, counterexamples generated from the reduced specification must be valid traces in the original specification. Of course, reduction details must be kept to turn counterexamples into tests. Different test requirements may call for different reductions.

One such sound reduction, called "finite focus" [1], reduces a large or infinite domain to a small subset of values. These values can be indicated by an analyst according to their testing importance. This reduction mechanically modifies both the state machine and test requirements.

In addition to using abstractions, we often start with a high-level design.

III. Higher level specifications

SMV's description language is too low level for wide-spread use. A popular system must get state machines from higher level descriptions such as MATLAB stateflows, SCR, HOL, or UML state diagrams.

Theorem provers and model checkers complement each other in description and analysis tasks. Static (or functional) aspects of a system are best described and analyzed with a theorem prover, while a model checker is well suited for dealing with dynamic (or behavioral) parts.

HOL provides a higher level of language constructs than does SMV. A proposed test generation framework [8] starts with a system model in HOL, mechanically converts a part of the model to SMV, generates test cases for the static (HOL) and dynamic (SMV) parts separately, and integrates the tests.

Portions of an HOL specification of a secure operating system model were converted to SMV using a prototype translator tool [8]. We also generated tests from the SMV model automatically. In the future, we hope to generate test cases from HOL specifications and integrate the test sets from HOL and SMV.

IV. Deriving logic constraints

Mutation adequacy [5] is a test criterion that naturally yields negative requirements. The specification-based mutation criterion [2] requires tests to distinguish between the original state machine description and its mutants, that is, ones that differ from the original by exactly one syntactic change. Consider the following fragment of a state machine description in SMV.

```
next(state) := case
    state = ready & req : busy;
    ...
esac;
```

One possible mutation is negating a boolean variable, as in

```
state = ready & !req : busy;
```

The specification-based mutation scheme in [2] expresses the state machine in temporal logic, then systematically applies small changes to the temporal logic expressions yielding a set of mutant expressions. The model checker then finds counterexamples that detect inconsistent mutants. The mechanical process of deriving temporal logic formulas from the state machine description is called *reflection*. A possible reflection for the above SMV fragment is

```
AG (state=ready & req -> AX state=busy)
```

This form of reflection, called *direct reflection*, is straightforward to derive. Suppose the SMV state machine description has the following case statement:

next(x) := case ... b_i : v_i; ... esac;

b_i and v_i are called *guard* and *target*, respectively.

If the guards are a partition and the targets are pairwise disjoint, a tighter reflection is possible:

AG (($b_i \rightarrow$ AX (x in v_i)) & (!$b_i \rightarrow$ AX !(x in v_i)))

For instance, if the mutation is to b_i to form b_i', the mutant formula is

AG ((b_i' $\rightarrow$ AX (x in v_i)) & (!b_i' $\rightarrow$ AX !(x in v_i)))

Moreover, as shown in [2], when the resulting counterexample includes an additional step, the clause

AG ($b_i \leftrightarrow b_i$')

is a satisfactory implementation for mutations to b_i.

There are transformations to recast the guards to be a partition and ways to cope with targets that are not pairwise disjoint.

V. Fault visibility

To detect an implementation error, a test case must cause an internal fault to propagate to a visible output. Consider the following fragment of a state machine description

```
next(t) := case ... f(i) : v; ... esac;
next(o) := case ... g(t) : w; ... esac;
```

In the example, `i` is an input variable, `t` is an intermediate variable, `o` is an output variable. Suppose that mutation replaced the formula `f(i)` with `f'(i)`. In the case of direct reflection, the corresponding mutant formula is

```
AG (f'(i) -> t = v)
```

Often, the model checker will find a counterexample that will show inconsistency in the intermediate variable `t` but not in the output variable `o`. Such a test is of little value.

We proposed two methods [6] to guarantee that tests cause detectable output failures. The first method, *in-line expansion*, uses only the reflections of the transition relation involving output variables. In these temporal logic formulas, any internal variable is replaced in-line with a copy of its transition relation. This substitution is repeated until the formulas are comprised exclusively of input and output variables, hence the model checker finds counterexamples that affect the outputs. For the above example, the mutant formula is

```
AG (f'(i) -> AX (g(v) -> AX o = w))
```

Since only input and output appear, the model checker finds counterexamples that affect the output. The method may lead to an exponential increase in the number or size of logical formulas.

The second method, *state machine duplication*, duplicates the state machine and combines the two machines ensuring that the duplicate always takes the same transitions as the original. The next step is to mutate the duplicate, then assert that the visible outputs of the original and the mutant are identical over the combined state machine. If the mutant has an observable fault, the model checker will produce a counterexample leading to the state where the original and the mutant differ in an output value.

Of course, duplication of the state machine increases the size of the state space. Dependency analysis by slicing is one way to improve scalability. Our experiments suggest that both in-line expansion and state machine duplication methods are very effective for generating black-box tests.

VI. Conclusions

We believe that there are benefits of applying model checking to software testing. While some issues raised in this paper are specific to test generation, others have much in common with the more mainstream uses of model checkers. The opportunities for future work include devising new abstraction techniques geared toward test generation, integration with higher-level languages, and developing a counterexample generator that guarantees propagation of faults to the visible outputs.

References

[1] P. Ammann and P. E. Black. Abstracting formal specifications to generate software tests via model checking. In *Proc. 18th Digital Avionics Systems Conference (DASC99)*, volume 2, page 10.A.6. IEEE, Oct 1999. Also NIST IR 6405.

[2] P. Ammann, P. E. Black, and W. Ding. Model checkers in software testing. Technical Report NIST-IR-6777, National Institute of Standards and Technology, February 2002.

[3] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54. IEEE Computer Society, December 1998.

[4] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model-checking. In *Proc. 1996 SPIN Workshop*, Rutgers, NJ, Aug 1996. Also WVU TR #NASA-IVV-96-022.

[5] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Comp*, 11(4):34–41, 1978.

[6] V. Okun, P. E. Black, and Y. Yesha. Testing with model checker: Insuring fault visibility. In *Proc. 2002 WSEAS International Conference on System Science, Applied Mathematics and Computer Science*, Oct 2002.

[7] S. Rayadurgam and M. P.E. Heimdahl. Coverage based test-case generation using model checkers. In 8^{th} *Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, Apr. 2001.

[8] D. Zhou and P. E. Black. Translating HOL to specifications for the model checker SMV. In *TPHOLs 2001*, pages 400–415, Sep 2001.