Experiments on Semantics Based Testing of a Compiler

Anton Esin[1], Andrey Novikov[1], and Rostislav Yavorskiy[2]

[1] Faculty of Mechanics and Mathematics
Moscow State University, Moscow 119992, Russia
{esin,novikov}@lpcs.math.msu.su
[2] Steklov Mathematical Institute
Gubkina str. 8, Moscow 119991, Russia
rey@mi.ras.ru

Abstract. The paper reports our ideas and experience on model based testing of a compiler. We consider a tiny programming language that consists of the standard control statements and external method calls only. Static and dynamic semantics of the language is formalized in the Abstract State Machines specification Language (AsmL), and used to produce a test suite for the compiler. In the second half of the paper we discuss possible extensions of the work.

1 Introduction

The notion of Abstract State Machines (ASMs, formerly known as evolving algebra) was introduced to provide mathematical foundation for the modern theory of algorithms. The approach has been successfully applied for modelling various hardware and software systems (see [1, 5]). In particular, several industrial programming languages were provided with precise mathematical semantics by means of ASMs. The most recent project of this kind is a formal semantics for C# presented in [4] (see also [7, 9]). General direction of our work is to use the ASM based formal semantics of the language to measure the quality of available C# compilers. We start from a very simple fragment of the language, which consists of basic selection, iteration and jump statements and simple boolean function calls only. We use the Abstract State Machines Specification Language (AsmL) developed at Microsoft Research [2, 7] to formalize syntax, static and dynamic semantics of the fragment. Then, we use AsmL Tester Tool to produce a test suite with reasonable coverage properties.

Typically, semantics based approach to compilers testing includes the following steps:

1. *Formalize the syntax of the considered fragment L of the language to be tested.* On this stage we simply define Abstract Syntax Tree (AST) of an L-program in AsmL.
2. *Formalize static semantics of the considered sub-language L.* This is done by adding the corresponding constraints to the AsmL description of AST.

Though the syntactical restrictions on the program structure could be quite complicated, the expressive power of multi-sorted first order logic (which is a part of AsmL) is sufficient.

3. *Formalize the dynamic semantics of L-programs.* According to ASM paradigm [8], one has to define explicitly the state of a program (control flow, memory state, methods called etc.) and then to describe one step transition function.
4. Now, the model of the language could be used for testing a compiler. The first two items above (AST plus static semantics) allow us to produce syntactically correct programs, and so to test the compiler front end. Then, one can use the dynamic semantics to check whether the produced executable conforms to the original program code. On this stage one faces the following crucial issues regarding parameter generation:

   (a) *How to generate the compiler input?* The space of all syntactically correct programs is huge even for tiny language L. One needs a practical idea which programs to choose for the testing.
   (b) *How to generate input for the compiler output?* Once the L-program is given and compiled, one needs to choose a realistic set of the program parameters to check the correctness of the compiler output.

2 Testing of the basic control flow statements

2.1 Sub-language under the test

Consider a language with the following grammar:

```
Statement ::  if ( BooleanCondition ) { Statement } else { Statement }
Statement ::  while ( BooleanCondition ) { Statement }
Statement ::  { Statement Statement }
Statement ::  break;
Statement ::  goto Label;
Statement ::  Label : Statement
Statement ::  ExpressionStatement;
ExpressionStatement ::  MethodCall
BooleanExpression ::  MethodCall
MethodCall ::  Fun(Label)
```

The syntax is formalized by the appropriate definition of the abstract syntax tree of a program. The static semantics corresponds to the requirements on the program structure. For the considered fragment there are three such restrictions:

1. The label used in a goto statement should be defined.
2. A jump should be correct (it is not allowed to jump into a loop body etc.)
3. A break statement should be located within a loop body.

2.2 Dynamic semantics of the sub-language

To formalize dynamic semantics of the language we use the notion of control point. Every statement is an instance of the abstract Statement class, which contains the two fields common for all statements, namely, EntryPoint and reference to the default ReturnPoint (i.e. where to go when the job is finished).

```
abstract class Statement
  const EntryPoint as ControlPoint  // initial control point
  var ReturnPoint  as ControlPoint  // return by default
```

Below is the definition of the `if` statement. The dynamic semantics is formalized in the `StepOn()` method. It specifies control flow inside the statement and transformation of the memory state. Namely, when control comes to the entry point of the statement it is forwarded immediately to the condition entry point. Then, after computing the condition the control comes to the branching point. At this state we expect that the valu of the condition in stored in the memory (the `Result` is boolean). Depending on the value the control is forwarded to the corresponding branch. When the branch has the job completed, the control comes to the `TerminalPoint`, and so on.

```
class IfThenElse extends Statement
  Condition as BooleanExpression   // abstract syntax
  YesBranch as Statement           //
  NoBranch  as Statement           //
  BranchingPoint as ControlPoint // additional control point
  TerminalPoint  as ControlPoint // for dynamic semantics

  override StepOn()   // dynamic semantics
    require CurrentControlPoint ne null
      and then CurrentControlPoint.Owner = me
    if CurrentControlPoint = EntryPoint then
        CurrentControlPoint := Condition.EntryPoint
    elseif CurrentControlPoint = BranchingPoint then
        require Result is Boolean
        if Result = true
          CurrentControlPoint := YesBranch.EntryPoint
        else
          CurrentControlPoint := NoBranch.EntryPoint
    elseif CurrentControlPoint =  TerminalPoint then
          CurrentControlPoint := ReturnPoint
```

Here `Result` is a variable that reflects the state of the memory. Due to the simplicity of the considered fragment the memory value could be either `Boolean` or `Null`.

Although we restrict ourselves to the Boolean valued methods only, the set of all possible results of the method call is wider. The method may also halt the execution or throw an exception. The dynamic semantics of the function call is given by means of nondeterministic choice and the corresponding external function.

```
class FunctionCall extends BooleanExpression
  override StepOn()
    choose result in {True, False, Halt}
    ExternalStepOn(result)

  ExternalStepOn(res as FunctionResult)
    require CurrentControlPoint = me.EntryPoint
    match res
      True:    Result := true
               CurrentControlPoint := ReturnPoint
      False:   Result := false
               CurrentControlPoint := ReturnPoint
      Halt:    CurrentControlPoint := null
```

2.3 Generating the compiler input

Since the grammar of the considered sub-language is very simple, one can practically consider all syntactically correct programs of a given depth d. For $d = 3$ there are 2039 such programs [10] (we do not distinguish programs with identical AST).

2.4 The conformance check

One can see that state space of any program in the considered fragment is finite. Moreover, the number of the states is linear on the size of the program. From the behavioral point of view every program represents deterministic finite automaton. The input of the automaton corresponds to the sequence of the results of the subsequent function calls. So, a natural requirement on the produced test suite is to cover all reachable transitions in the automaton. We use the AsmL Test tool [1] to accomplish the task. Consider the following example.

```
static void TestCase(){
   L: if ( Fun("L1") ){
        L2: goto L;
      } else{
        L3: while( Fun("L31") ){
              L32: Fun("L32");
} } }
```

The string argument of the function indicates the location of the function call in the program. It is used to observe the actual path of the computation. As it is explained above, any run of the program is determined by the consequent values returned by `Fun()`. For the considered example the produced sequences are:

```
[TEST_SEQ]
tff     L1L1L31
fth     L1L31L32
fttf    L1L31L32L31
ftth    L1L31L32L31
```

```
ftttfh  L1L31L32L31L32L31
h       L1
tth     L1L1L1
[TEST_SEQ_ENDS]
```

Here `t` stands for `true`, `f` for `false`, `h` for `halt`. The left column is used to implement `Fun()`, the right one indicates the expected output.

3 Testing with large programs

The test suite described in the previous section has the following nice property: it has a test case for every combination of three control flow statements. Its weak point is the size of the program. Only few of the produced programs have more than dozen lines of code.

The next our task is to use the produced cases as a basis for more complicated tests.

3.1 Iteration of a program

The idea is quite simple. We take any program, remove a leaf statement and put on its place a copy of the initial program. In such a way we can get arbitrarily large programs with very simple (periodic) structure. Theoretically, such an iteration of a correct program AST is always a correct program AST. On the other hand, it is naturally expected that every particular compiler has limits.

3.2 Computing tests for the iteration of a program

It turns out that test sequences for the iteration of a program could be explicitly computed from the test sequences for the initial program. That gives us the following testing agenda:

1. Take a program from the depth 3 complete test suite described above. Iterate it n times for $n = 2, 3, 4, \ldots$ etc.
2. Find the largest n for which the result is still accepted by the compiler under the test (in our experience such a number always exists).
3. Compute the test sequences for the resulted program and use them to check the executable.

This approach gives us test suite made of programs located on the edge of the compiler limits.

4 Extensions of the considered sub-language

4.1 Exceptions

First of all we extend the grammar with the following line:

```
Statement ::  try { Statement } catch { Statement } finally { Statement }
```

In general, that changes almost nothing. Still, every program corresponds to a finite automaton of the restricted size, so we can produce and completely test all the programs of depth 3, and also to experiment with the idea of program iteration described above.

4.2 Boolean arithmetic

In order to model programs that use complex expressions (e.g. boolean combinations of conditions) one needs to improve the memory model. In the simplest case the memory was modelled by the single variable `Result`. Now it becomes a stack of bounded size (from the theoretical point of view it is equivalent to a fixed set of boolean variables).

4.3 Static integer fields, arithmetical operators and simple assignments

The next reasonable extension is to add integer variables and basics of arithmetic. More precisely, we add static (integer) variables declarations, external integer functions, standard arithmetical operators, equality and inequality relations (and their boolean combinations), and the assignment statement.

It follows immediately that the state space of a typical program is too large to cover it completely (like we have done for programs in the initially considered fragment). So we need to choose a reasonable coverage criteria for the test sequences [3, 7]. Given a fixed set $\{P_1, \dots, P_m\}$ of boolean conditions on the program variables we consider the following equivalence relation on the program states: $s_1 \approx s_2$ iff $P_i(s_1) \equiv P_i(s_2)$ for all $i \in \{1, \dots, m\}$ and

$$\texttt{CurrentControlPoint}(s_1) = \texttt{CurrentControlPoint}(s_2).$$

Then, the goal is to find the set of program parameters such that each of the equivalence classes is visited at least once, and all possible transitions between the hyper-states are covered [7, 11].

The latest results of this project could be found at [10].

References

1. Abstract State Machines academic web page `http://www.eecs.umich.edu/gasm/`
2. AsmL at Microsoft Research. `http://www.research.microsoft.com/fse/asml`
3. T. Ball. *A Theory of Predicate-Complete Test Coverage and Generation.* Microsoft Research Technical Report MSR-TR-2004-28, April 2004.
4. E. Boerger, G. Fruja, V. Gervasi, R. Stark, *A High-Level Modular Definition of the Semantics of* C# . Theoretical Computer Science, 2004, pp. 1-35 (to appear).
5. P. Cutter and A. Pierantonio. *Montages: specifications of Realistic Programming Languages.* Journal of Universal Computer Science, 3(5) pp. 416–442, 1997.

6. C# language specification. Standard ECMA-334, 2nd edition, December 2002.
7. W. Grieskamp, Yu. Gurevich, W. Schulte, and M. Veanes. *Generating Finite State Machines from Abstract State Machines.* In ISSTA 2002, International Symposium on Software Testing and Analysis, July 2002.
8. Yu. Gurevich. *Evolving algebras 1993: Lipari Guide.* In E.Boerger, editor, Specification and Validation Methods, pp. 9–36, Oxford University Press, 1995.
9. A. Kalinov, A. Kossatchev, A. Petrenko, M. Posypkin, V. Shiskov. *Coverage-driven Automated Compiler Test Suite Generation.* ENTCS, 2003.
10. Modante research project. `http://www.modante-research.narod.ru/projects.html`
11. M. Veanes and R. Yavorsky. Combined Algorithm for Approximating a Finite State Abstraction of a Large System. In SCESM 2003, 2nd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, May 2003, pp. 86-91.