Applying formal semantics of an object-oriented language to program invariant checking

Andrey Novikov[1] and Rostislav Yavorskiy[2] ⋆

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

Abstract. We consider a specific kind of Abstract State Machines. It is shown how the machines can be used to provide a low-level formal semantics for a tiny object-oriented language, including control flow operators, object creation and field manipulation. Then the decidability result is established for checking invariants of programs corresponding to that class of ASMs.

1 Introduction

The Abstract State Machines (ASM) methodology (see the fundamental definition [12] and the contemporary book [6]) has been widely used to provide a formal semantics for programming languages (see [5], [7], [8] and the ASM Internet site [2] for an extensive collection of links).

We consider a specific class of ASMs and use it to provide low-level dynamic semantics for a part of pure object-oriented language. As an example, we show how to describe the basic constructions of C# [9] in terms of our machine. However, the language part that we model is the same for all OO languages. Namely, we consider semantics for the core (in-procedure) language, object-related operators and method calls.

Adapting the result of [13], we prove the following. Let φ be a property of the state. Under certain restrictions on φ and transition τ, the formula $\varphi \rightarrow \bigcirc_\tau \varphi$ is decidable. Here $\bigcirc_\tau$ is the temporal logic operator "valid in the next state after the transition τ". As a consequence, we obtain a class of program invariants that can be automatically checked.

Section 2 presents the computational model itself. In section 3 we describe the semantics for a tiny object-oriented language. The decidability result is presented in section 4. Section 5 concludes the previous results and formulates the invariant-checking theorem. In section 6 we discuss the ongoing work.

⋆ Partially supported by grants of President of Russia for Leading Scientific Schools, Russian Foundation for Basic Research, Russian Science Support Foundation, and Microsoft Research.

2 Computational model

Our model is a specific class of the Abstract State Machines [12], [8]. An ASM is a pair $< S, \tau >$, where S is a set of states and $\tau : S \to S$ is a transition function.

This chapter defines the computation model, describing its state and transitions.

State

The first step to describe an ASM is to define its state. A state is an algebraic structure, i.e. a set of universes and a number of functions over them. These functions are dynamic, i.e. they may be changed with the transitions.

We enumerate the universes used in our model, and give a brief description how they are used to build an object-oriented language semantics. Detailed dynamic semantics is given in the next section.

- *ObjectID*. This universe is a countable set with equation. It may be thought of as the set of all integers. In our semantics definition, it is used as an enumerator of all objects and variables that appear in the program. This universe includes special object **null**.
- *Integer* — the set of all integer numbers. This is used to store values of the object fields.
- *ControlPoints*. This is a finite set representing points of program execution. The execution process will include changing the control point on each step.

The following dynamic functions are considered in the model:

- $IntMember_i : \mathit{ObjectID} \to Integer \cup \{Undef\}$. Retrieves i-th integer-type member of an object. The members are enumerated according to the class definition. Almost all of these functions always return $Undef$.
- $ObjMember_i : \mathit{ObjectID} \to \mathit{ObjectID} \cup \{Undef\}$. Retrieves i-th class-type member of an object. Like in the previous case, almost all of these functions always return $Undef$.
- $CurrentCP : ControlPoints$. This nullary function (a variable) indicates the current control point at each moment.
- $Result : Boolean$. This variable stores the result of the last boolean function call. This register is used for operating with control-transmitting instructions.

Transition

Let us denote by S the set of all states projected on a single object. That is,

$$\begin{aligned} S = {} & ControlPoints \times \mathit{ObjectID} \times \ldots \\ & \times \mathit{ObjectID} \times Integer \times \ldots \times Integer \times Boolean \end{aligned}$$

The *Object* factors stand for values of $Member_i$ functions on the given object, and *Integer* factors stand for $Value$s of its integer-type fields. The *Boolean* field is the value of the *Result* register.

The only transition in our system is the function $StepOn()$ which moves $CurrentCP$ and possibly redefines other dynamic functions. We assume that the transition modifies only dynamic functions on one object, as this is sufficient for the semantics construction. Additionally, a transition may change the $Result$ variable.

Formally, the transition is the following:

```
procedure StepOn()
    choose NextState in S
        where δ(NextState, CurrentState)
      CurrentCP := ControlPoint(NextState)
      ObjMember_1(Self) := ObjMember_1(NextState)
      ...
      ObjMember_m(Self) := ObjMember_m(NextState)
      IntMember_1(Self) := IntMember_1(NextState)
      ...
      IntMember_n(Self) := IntMember_n(NextState)
```

Here the right parts are the corresponding values of the $NextState$, and δ is a first-order formula in the signature $(+,\geq,\cdot const)$ over integer-type members of S. The $CurrentState \in S$ is the set of initial values of dynamic functions on the object $Self$.

3 Semantics for a simple object-oriented language

This section presents a way to convert a program in an OO language to a program for our machine. We consider a language with:

- integer constants
- control flow operators like **if, while, break**
- linear arithmetic: $+, \cdot const$
- inequation predicates $\geq, >, =$
- class definitions, members and methods
- object creation and manipulation

We do not write down the full conversion pattern precisely, but provide the essential ideas of this conversion. For simplicity, we illustrate the ideas of this section with translation of the C# language constructs. However, everything is correct for Java and C++ also.

3.1 Core language

Here we discuss how to interpret core operators of the C# language in terms of our machine. Operators like **if, while-break** are considered here (the assignment operator is handled in the next subsection).

We illustrate the idea with an example of the **while** operator.

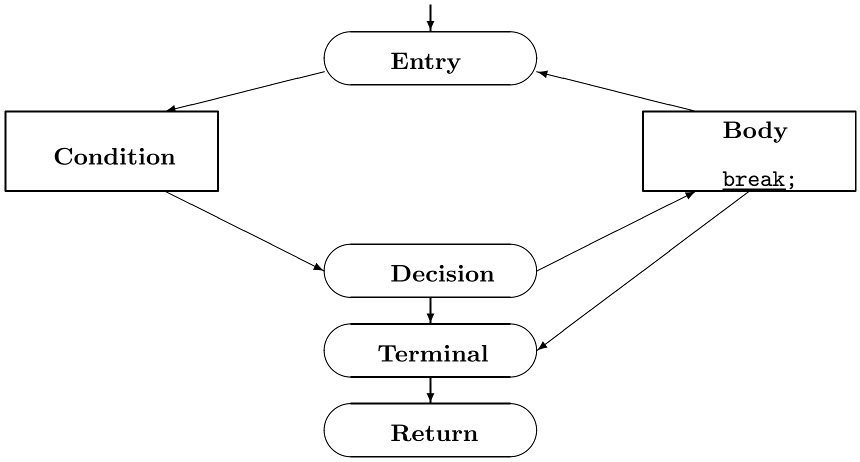

Fig. 1. Control flow graph for the **while** operator

Consider the control flow graph at Figure 1. For the **while** loop, we introduce four control points: **Entry Point, Decision Point, Terminal Point** and **Return Point**. Logically, the latter is the entry point for the next operator. The boxes **Condition** and **Body** represent graphs for the loop condition and body. The arrows depict possible control transitions. Note that the arrow between the **Body** and the **Terminal Point** is used when the `break` operation is executed.

Let us see how it works. We are using the syntax of the AsmL language [1]. Here is the part of the `StepOn()` procedure responsible for working with **while** loops.

```
procedure StepOn()
  if CurrentControlPoint = EntryPoint then
    CurrentControlPoint := Condition.EntryPoint

  elseif CurrentControlPoint = DecisionPoint then
    if Result = true then
      CurrentControlPoint := Body.EntryPoint
    else
      CurrentControlPoint := TerminalPoint

  elseif CurrentControlPoint = TerminalPoint then
      CurrentControlPoint := ReturnPoint
```

Note that the *Result* register is initialized while proceeding the **Condition** part of code.

Remark. An alike idea can be also used for describing semantics for exception handling. What we have to do is to properly define the **Return Point** for the

throw operator. It should be the entry point for the nearest exception handler that is catching exceptions of suitable type. Implicit calls of the **throw** operator (as for **div** `0`) can be made explicit.

Suppose now that we have a piece of code. The part of the semantics defined by now is an automaton switching between control points.

3.2 Object-related operators

This subsection describes basic object-related operators such as object creation and operating with fields. The method calls are discussed in the next subsection.

Object creation. Consider the following code in C#:

```
class SampleClass{
  int r;
  SampleClass obj;
}

...
  SampleClass sampleObj = new SampleClass(5, null);
```

The semantics of the above is the following. We take fresh $o \in \mathit{ObjectID}$ and set $IntMember_1(o) = 5,\ ObjMember_1(o) = \mathbf{null}$.

Field manipulation. Let us extend the previous example:

```
...
   sampleObj.obj = sampleObj;
```

This assignment corresponds to the transition $ObjMember_1(o) := o$.

3.3 Method calls

We treat a method call as a creation of a new object with its own `StepOn()`. This approach allows us to deal well with recursion and brings no additional complexity to the model.

Suppose we have a method for our `sampeClass`

```
class SampleClass{
  ...
  void Increase(int s){
    r += s;
  }
}
```

We create an additional class `SampleClassIncrease` with the single method `Run()`. The local variables of the original method `Increase` are treated as fields of `SampleClassIncrease`.

```
class SampleClassIncrease{
  int s;
  Object owner;
  void Run(){
    owner.r += s;
  }
}
```

Now we treat a call to the `Increase` procedure as a creation of a new object of type `SampleClassIncrease`. The Run method of this object is converted to a number of control points and the execution is controlled by the standard `StepOn()` function as defined above.

4 Single-step decidability

We explore here the decidability property of a single transition as defined in section 2.

Let us denote $f(o) \equiv \mathit{CurrentState}(o), \quad o \in \mathit{ObjectID}$. That means that $f(o)$ is a record of all current properties of the object o.

Then, let σ be a signature over integer numbers, including addition and $\geq$. Finally, denote the transition by τ.

We are interested in the properties of the following form:

$$\varphi \to \bigcirc_\tau \varphi.$$

Here $\bigcirc_\tau$ denotes the temporal operator "valid in the next state after transition τ", and the formula φ is of the following form:

$$\varphi \rightleftharpoons \forall o_1 \ldots o_s \in \mathit{ObjectID} \quad (\mathit{Diff}_s(o_1, \ldots, o_s) \to \psi(f(o_1), \ldots, f(o_s))),$$

where ψ is a first-order formula over σ, and the predicate Diff_k evaluates to true iff all of its k arguments are different.

By adapting the proof in [13], we obtain the following theorem. We use here that the first-order theory of linear functions and inequality over integer numbers is decidable.

Theorem 1 *The relation $\varphi \to \bigcirc_\tau \varphi$ is decidable.*

5 Conclusion

This section applies the decidability result to invariant checking in realistic programs.

Consider a piece of code in the programming language defined in the beginning of section 3. Consider a set of formulas $f_i(obj_1, \ldots, obj_n)$ containing linear operations on values of object members in the program (or members of members, etc.). Let now $\varphi(f_1, \ldots, f_n)$ be a first-order formula with inequations and linear operations. Then we have the following theorem:

Theorem 2 *In the above conditions, boolean formulas $\varphi(f_1, \ldots, f_n)$ can be checked for being invariants.*

To check the invariant, we have to check the property $\varphi \to \bigcirc_\tau \varphi$ for all types of transition used in our program (this is made statically, and the operation is finite since the program is finite).

Remark. Any computation with multiplication and linear operations can be reduced to a computation with linear operations through looping. However, this reduction changes the notion of a step, and so affects the class of invariants being checked.

However, the limitation on linear nature of the invariants being checked is crucial.

6 Ongoing work

We are currently working on detailed proofs of the theorems 1 and 2, and estimation of the computational complexity of the deciding algorithm. The semantics itself can be applied to compiler correctness testing (see [11] for details).

The future work on the topic is to programmatically implement the algorithm and check its feasibility with real-life examples.

The up to date information is available at our Internet site [14].

7 Acknowledgements

We gratefully acknowledge Yuri Gurevich and Wolfram Schulte for valuable comments on general direction of our work, and Egon Boerger for his advice on the resources available. We also thank Andrey Chepovskiy and Vladimir Krupski for their critical, though encouraging comments on our results.

We give special thanks to Nikita Mamardashvili for co-authoring the general idea of this work. Thanks to Anton Esin for regular discussions of our activity.

We are also thankful to anonymous referees for their comments on possible refinement of this work.

References

1. AsmL: The Abstract State Machine Language. Reference Manual. Modeled Computation LLC, 2002.
   http://research.microsoft.com/fse/asml/
2. Abstract State Machines academic web page, http://www.eecs.umich.edu/gasm/
3. T. Ball. *Abstraction-guided test generation: a case study,* November 2003.
4. T. Ball, S.Rajamani. *Checking temporal properties of software with boolean programs.*

5. S. Cater, J. Huggins. *An ASM Dynamic Semantics for Standard ML* in *Abstract State Machines – ASM 2000, International Workshop on Abstract State Machines*, Monte Verita, Switzerland, Local Proceedings, TIK-Report 87, Swiss Federal Institute of Technology (ETH) Zurich, March 2000
6. E. Boerger, R. Staerk. *Abstract State Machines. A Method for High-Level System Design and Analysis*, Springer-Verlag, 2003.
7. E. Boerger, N. Fruha, V.Gervasi, R. Staerk. *A High-Level Modular Definition of the Semantics of C#*, Theoretical Computer Science, 2004, pp.1-35.
8. E. Boerger, J. Shmid, R. Staerk. *Java and the Java Virtual Machine: Definition, Verification, Validation*, Springer-Verlag, 2001.
9. *C# language specification. Standard ECMA-334, Second Edition,* http://www.ecma-international.org/publications/files/ecma-st/ECMA-334.pdf
10. J. Alves-Foss. *Formal Syntax and Semantics of Java,* volume 1523 of *Lecture Notes in Computer Science.* Springer-Verlag, 1998.
11. A. Esin, A.Novikov, R.Yavoskiy. *Experiments on Semantics-based Testing of a Compiler*, http://modante-research.narod.ru/papers/comptest04.pdf
12. Y. Gurevich. *Evolving Algebras 1993: Lipari Guide* in *Specification and Validation Methods* Ed. E. Boerger, Oxford University Press, 1995, 9–36.
13. Y. Gurevich, R. Yavorskiy. *Observations on Decidability of Transitions,* in Proc. ASM 2004, Springer LNCS 3052, 163-168.
14. Modante Research website, http://modante-research.narod.ru