Toward Industrial Strength Abstract State Machines*

Yuri Gurevich, Wolfram Schulte, and Margus Veanes
{gurevich, schulte, margus}@microsoft.com

October 2001

Technical Report
MSR-TR-2001-98

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Abstract. A powerful practical ASM language, called AsmL, is being developed in Microsoft Research by the group on Foundations of Software Engineering. AsmL extends the language of original ASMs in a number of directions. We describe some of these extensions.

* This paper was supposed to appear in the Proceedings of the ASM'01 Workshop held in February 2001 on the Canary Islands. The Proceedings were supposed to be published in Springer Lecture Notes in Computer Science, but the publication was aborted. There are no plans for a journal publication of this article.

1 Introduction

Executable specifications in the form of abstract state machines can play an important role in software development. To realize the potential, the Foundations of Software Engineering group at Microsoft Research has developed (and continues to develop) AsmL, an ASM Language. AsmL is a powerful, practical language. It facilitates writing complex specifications and allows you to execute these specifications. It is integrated with the current Microsoft programming environment and is intended to support various useful tools. See our website [4] in this connection.

At this point (the spring of 2001), AsmL is single-threaded but is embedded into a multi-threaded runtime environment.

Conceptually AsmL is massively parallel. One state-change of an AsmL program may be a complex transaction involving numerous subprograms. This allows you to avoid unnecessary sequentialization imposed by conventional programming languages and to make your programs clearer.

To deal gracefully with numerous issues of practical execution, AsmL extends the language of original ASMs in a number of directions. Here we describe some of these extensions. Our description of these extensions is admittedly high-level but it gives main ideas. We presume that the reader is familiar with the Lipari guide [5] to abstract state machines (called evolving algebras in the guide), and we restrict attention to sequential-time ASMs. A more detailed semantical treatment of AsmL, in the spirit of the Michigan guide [6], is in preparation.

While this paper is influenced by the development of AsmL, we take a broader view. In particular, we do not adhere slavishly to the current AsmL syntax.

2 The Background Structure

By default, sets, sequences and maps are finite in this paper.

AsmL includes arithmetic and allows you to form sets, sequences, maps, tuples, etc. of your data. You may have, for example, maps from integers to sets of integers, sequences of such maps, sets of such sequences. What world do all these composite entities live in? To provide such a world, special background structures were introduced in [1]. A set-background structure was explored in [2] without using the term "background". In the rest of this section, we mention some aspects of the AsmL background structures.

Sets The background structure of AsmL includes the set-background of [2]. AsmL has symbols for the containment relation, for the empty set and for a number of set-theoretic operations. The states are closed under the set-formation operation: If s is a set of elements of a state X then s itself is an element of X; in that sense s is an *elementary set*. Think about the base set of X as follows:

it is composed of non-sets and of the sets built by repeatedly applying the set-formation operation. As in [2], we use set-comprehension expressions

```
{t(x) | x in r where φ(x)}
```

Here r is a set-valued expression so that values of r are elementary sets. For example,

```
{x*x*x | x in {1,2,3,4} where x*x < 10} = {1,8,27}
```

Tuples and Sequences The AsmL background includes tuples and sequences, so that, in any state, every tuple of elements is an element, and every sequence of elements is an element (an *elementary sequence*). We use parentheses to denote tuples: `(1,8,27,64)`. We use square brackets to denote sequences. A sequence-comprehension expression has the form

```
[t(x) | x in r where φ(x)]
```

Here r is a sequence-valued expression. For example,

```
[x*x*x | x in [3,4,3,5] where x*x < 10] = [27,27]
```

Maps The AsmL background includes maps. A map m is a unary function such that, in any state X, the domain $\mathrm{Dom}(m)$ is a set of elements of X, and every $m(x)$ is an element of X; $m(x) = \texttt{undef}$ for every x outside $\mathrm{Dom}(x)$. Any map from elements of X to elements of X is again an element of X (an *elementary map*). A built-in function Apply allows us to apply a map m to an element x. We abbreviate $\mathrm{Apply}(m, x)$ to $m(x)$.

There are only so many parentheses in a key-board. How to denote maps? Consider a map m that takes 1 to 7 and 2 to 11. The graph of m is the set $\{(1,7),(2,11)\}$. The map m itself can be denoted thus: $\{1 \mapsto 7, 2 \mapsto 11\}$. This does not mean that m is a set. This is just a convenient notation. The constituents $1 \mapsto 7$ and $2 \mapsto 11$ are called *maplets*. The notation $\{1 \mapsto 2, t \mapsto 3\}$ will generate a runtime error in a state where $t = 1$.

The map-comprehension expression has the form

```
{ t(x) ↦ t'(x) | x in r where φ(x)}
```

Here r is a set-valued expression. For example,

```
{ x ↦ x*x*x | x in {1,2,3,4} where x*x < 10} =
{ 1 ↦ 1, 2 ↦ 8, 3  ↦ 27 }
```

Reserve The above description of the AsmL background is not exhaustive. In addition, the background contains the integers for example. Think about the base set of an AsmL state as follows: it is composed of atoms (that are not sets, not sequences, etc.) and of sets, sequences, etc. built by repeatedly applying the set-forming operation, the sequence-forming operation, etc. in any order.

In a given state, an atom can be the value of an expression. Here are some example expressions: `undef, 1+1, head(s)` where `s` is a sequence variable. An atom can be a constituent of the value of an expression, for example it may be an element of the value of a set variable. The remaining atoms are indistinguishable and constitute the reserve of the state. Any permutation of the reserve gives rise to an automorphism of the state. In contrast to the Lipari, the reserve is not a naked set. For example, tuples containing arbitrary reserve and nonreserve elements exist in the state; you do not have to create them after importing the reserve elements.

3 Partial Updates of Sets and Maps

3.1 Partial Updates of Sets

There are two distinct ways to view a set. From one point of view, a set is a single unit. Accordingly, a set variable s is a nullary function whose intended values in a given state X are elements of X that are sets. An update rule $s := e$ gives rise to a total update of s; the old value of s is replaced with a new value. From the other point of view, a set is an aggregate entity; you can put elements in and out of the set. To reflect this point of view, you want to see the set variable s as a dynamic relation. Update rules $s(e_1) := \texttt{true}$, $s(e_2) := \texttt{false}$ are partial updates of s.

In the unitary approach, partial updates are problematic. If you want to insert an element, say 7, into s, you can use the update rule $s := s \cup \{7\}$. But two different insertions like that, say $s := s \cup \{7\}$ and $s := s \cup \{11\}$, contradict each other. Of course you can combine the two rules by hand, e.g., $s := s \cup \{7, 11\}$. In general, this is impractical. Different partial updates of s may come from different parts of your program.

In the aggregate approach, total updates of s are problematic. Suppose that the desired new value of s is $\{1, 2, 3\}$. The update commands $s(1) := \texttt{true}$, $s(2) := \texttt{true}$ and $s(3) := \texttt{true}$ insert the three elements into s; the problem is to ensure that s will not contain any other elements.

AsmL allows you to make total and partial updates at once. Let us worry about the legitimacy of this later. For now, let's explain how to combine a collection U of total and partial updates of the set variable s. The most interesting case is this.

- There is exactly one total update in U; it replaces the old value of s with a set a.

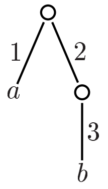
- The elements $b_1, \ldots, b_p$ that U inserts into s and the elements $c_1, \ldots, c_q$ that U removes from s are disjoint, so that $\{b_1, \ldots, b_p\} \cap \{c_1, \ldots, c_q\} = \emptyset$.

In this case, U is consistent if and only if a contains every b_i and no c_j.

3.2 Partial Updates of Maps

Let μ be a map variable. It may happen in a given state X that some $\mu(a_1)$ is again a map, and $\mu(a_1)(a_2)$ is again a map, and so on. You may want to update various maps of the form $\mu(a_1)(a_2) \cdots (a_j)$. This poses an update-consistency problem. We illustrate the problem and the solution on an example.

Let μ be a map variable with value $\{1 \mapsto a,\ 2 \mapsto \{3 \mapsto b\}\}$. μ can be pictured as a labeled tree



We update $\mu(2)$ to the map $\{5 \mapsto c\}$ via the rule

$$\mu(2) := \{5 \mapsto \mathtt{c}\}. \tag{1}$$

and we update $\mu(2)(3)$ to d via the rule

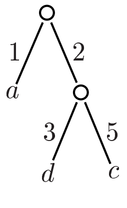
$$\mu(2)(3) := \mathtt{d}. \tag{2}$$

If both rules are fired in parallel, then the resulting update is contradictory. Indeed, let Y be the sequel of the current state X. By the first rule, the value of $\mu(2)$ in Y is the map $\{5 \mapsto c\}$ and so $\mu(2)(3) = \mathtt{undef}$, but, by the second rule, the value of $\mu(2)(3)$ in Y is d.

If, instead of rule (1), we fire the rule

$$\mu(2)(5) := \mathtt{c} \tag{3}$$

in parallel with the rule (2) then the resulting update set is consistent and the value of μ in Y is, as expected, the map



That is how AsmL deals with partial updates of maps.

3.3 Justification of Partial Updates

The Lipari guide [5] does not provide for partial updates. The development of AsmL put the issue of partial updates on the front burner. Partial updates will be legitimized in the paper [8].

The question arises whether partial updates can be justified in the Lipari framework. Yes, this can be done, but at a price. To justify partial updates in the Lipari framework, we need to present each partial update as a total update of some location. In the case of maps, one can introduce a special map registry function $\mathrm{MR}(x)$ where x is a sequence. To modify a map variable m at 1, update location $\mathrm{MR}(m,1)$. To modify $m(1)$ at 2, update $\mathrm{MR}(m,[1,2])$. To modify m itself, modify location $\mathrm{MR}(m,\mathrm{Nil})$. In this approach, m is a static function whose denotation just represents m in the state. The information about the map associated with m is give by the map registry. A similar approach can be applied to sets. We do not develop this approach in details as we are not going to adopt it.

4 Blurring the Difference Between Expressions and Rules

In AsmL, an expression may include rules so that the evaluation of the expression may produce updates. This makes programming easier and more natural. For a simple example, suppose that, for some reason, it became necessary to keep track of all arguments x passed as parameters to a function (or named expression)

```
f(x as Integer) as Integer =
   return e(x)
```

with integer arguments and integer values. To achieve the goal, introduce

```
var S as Set of Integer initially {}
```

and modify the definition of f above to

```
f(x as Integer) as integer =
   S(x) := true
   return e(x)
```

Otherwise the program remains intact. In particular, there is no need to change any code that calls f.

Update-producing expressions make modeling easier and more natural. Here is a simple and typical example. Large scale programming tends to be more and more component oriented. COM is Microsoft's Component Object Model. A COM component comes with an official interface. The interface often has an enumerator holding a sequence of integers; a special function Next allows a client to consume the sequence one integer at a time. Modeling such a component in AsmL, we have

```
  class Enumerator (s as Seq of Integer)
    var t as Seq of Integer = s
    Next() as Integer =
      if t = [] then undef
      else t := tail(t)
           return head(t)
```

Notice that a call to the Next function changes the value of t.

5 Machines

Let us first review a sorting example in the May 2001 release of AsmL. Declare a sequence variable

```
var A as Seq[Integer]
```

which holds a sequence of integers. For many purposes, AsmL treats a sequence x as a map on {0..length(x)-1}. The rule

```
swap() =
  choose i, j in {0..length(A)-1}
    where i < j and A(i) > A(j) do
    A(j) := A(i)
    A(i) := A(j)
```

chooses nondeterministically two members of the sequence which are in the wrong order and swaps them. The two assignments are executed in parallel of course. If A has been sorted, then the rule does nothing.

The rule swap can be executed repeatedly until the computation reaches the fixed point, that is the state does not change anymore. This gives rise to a little abstract state machine

```
  sort() =
    until fixpoint do
      swap()
```

The machine is an executable specification of in-space one-swap-at-a-time sorting and thus describes a family of sorting algorithms, e.g. bubble sort and quicksort. You can use sort() as a rule in your program.

```
prog1() =
  machine
    A := [7,3,8,2]
  step
    sort()
  step
    writeln(A)
```

The AsmL keyword `step` indicates sequential substitution. Thus `prog1` performs three steps: initializes `A`, sorts it, and then prints out the result. On the abstraction level of `prog1()`, `sort()` does only one step; the sorting computation is performed on a lower abstraction level. In that sense, `sort()` is a submachine of `prog1()`.

Now `prog1()` itself can be used as a submachine in a larger program, say `prog2()`. On the abstraction level of `prog2()`, machine `prog1()` performs only one step, not three. The sequence variable `A` can be declared and initialized in `prog2()` outside `prog1()`; then machine `prog1()` changes the state of `prog2()`.

The keyword `machine` is used to indicate the program itself as well as its submachines. The body of the function `sort()` is a machine but the keyword `machine` is not used there because of the keyphrase `until fixpoint do`. Machines also can return results.

Historical Remark The notion of submachine was explored in [3].

6 Exception Handling

Throwing and catching exceptions is a powerful idea of modern approach to programming. The use of exceptions allows one to write a cleaner code that emphasizes the intended functionality. A rudimentary form of exception handling appears in the Michigan guide in the form of a try-else construct. AsmL uses exception handling in a systematic way.

Here is a simplified AsmL example from one of our window-networking projects.

```
Install(parameters) =
   if something_is_wrong_with_the_parameters then
      throw e1
   else
      try
         machine
            uninstall_the_old_component
         step
            read_in_the_necessary_info
         step
         install_new_part
catch
      e2 as RuntimeException :
         write_log(e2)
         throw e3
```

The `Install` method takes a tuple (in fact a long tuple) of parameters. One of the parameters indicates which component is to be installed. First it is checked that the parameters are legal. If something is wrong with the parameters then exception `e1` is thrown. (In the actual code, different exceptions are thrown depending on the conditions that are violated.) The exception `e1` is supposed to be caught by the user who called the `Install` method.

If all the parameters are legal, the `try` block is executed: the old version, if any, of the component is removed, then the data is read in, and finally the new component is installed. Many things can go wrong during any of the three phases in which case an appropriate exception `e2` is generated and the control moves to the `catch` block. Here an entry is made into the log file and a new exception `e3` is generated for the benefit of the caller.

Every exception has a type in AsmL. It may or may not be an object. Conceptually, it is possible that several exception of the same `try` block are thrown at once. Which one should we catch? One strategy is to nondeterministically choose one of the thrown exceptions. The current implementation of AsmL is within this strategy. There are other reasonable strategies. For example, you can catch all thrown exceptions. Alternatively, you may want to catch all thrown exceptions that are most specific with respect to the subtyping discipline.

Clashing updates are of particular interest to ASMers. It is not necessary to patiently collect all updates generated by the given program during the whole one-step transaction. A `try` block around a subprogram allows us to better handle updates clashes.

Implicitly the whole program is enveloped into a `try` block with Skip as the exception handler. In this case, instead of the user exception handling, we have system exception handling; the AsmL runtime system gives you some information about what went wrong.

7 Objects

7.1 Ontology

AsmL is object oriented (OO). The problem of OO programming within ASM paradigm has been addressed earlier by Zamulin [9]. Our solution is simpler. Slightly simplified, it is this. A class C is just a dynamic universe. Objects of C are elements of that universe. Initially the class is empty. New objects are imported from the reserve. To be more precise, the dynamic universe is the *extent* of C; there are also fields and methods associated with C. And it is the identifier (ID) of an object that belongs to the dynamic universe; the object itself has its local state in addition to its ID.

Let us address here the issue of the local state of an object. Here is one natural approach. View a field F of a class C as a dynamic function on (the extent of) C. For every object o of C, $F(o)$ is a local variable of o. The local state of o is given by the tuple of the values of the local variables of o. This approach becomes problematic if the pool of classes can grow during the computation: new fields should be added to the vocabulary which is supposed to be fixed. AsmL does not have dynamic loading of classes at this point but we have to plan for it. In any case, we should be able to deal with dynamic loading of classes. To this end, we adopt a more general approach. Classes and their fields and their methods are represented as elements of the state. A special dynamic object registry OR, given (the ID of) an object o gives (the representation of) the class of o and its local state.

7.2 Object Registry and Garbage Collection

Garbage collection is an important part of modern object oriented programming. Unreachable objects can be removed. Conventional wisdom tells us that garbage collection should make no semantical difference. This is not completely true. Indeed, it should not make any semantical difference whether unreachable objects have been actually removed. On the other hand, it is important that unreachable objects *can* be removed.

In this connection, a question arises whether our object registry OR inhibits garbage collection. In particular whether OR can be used to count the number of elements in a given class. This counting would be possible if OR were a map variable, but OR is a unary dynamic function. The obvious way to extract global information from OR is to use a comprehension expression {`t(x) : x in Dom(OR) where` ϕ`(x)`} but this is syntactically wrong because OR is not an expression and so `Dom(OR)` is not an expression. In order to apply OR to the ID an object o, we need an expression that evaluates to that ID; if o is unreachable, there is no such expression.

To summarize, the use of the object registry does not inhibit garbage collection.

8 Object Initialization Problem

During one step, an AsmL program may create new objects. For example, our program may declare a class Horse

```
class Horse
   var Color as String
```

and then create a white horse Jerry.

```
var Jerry as Horse
Jerry := new Horse(‘‘White’’)
```

It is higly desirable that new objects are initialized at creation so that the field variables of the object are given some initial values. For example you may need to create a white horse, that is a horse object with the field Color set to White. Notice that initialization involves updating the object registry. These initialization updates will be executed only at the end of the current step. As a result, the initial values are not available at the current state. They will be available only at the next state but we need them in the current state. That is, in short, the initialization problem.

To present our solution of the problem, let us recall the nature of the ASM creation rule

```
create x in
   R(x)
```

An external demon (a part of our environment) chooses one of the reserve elements for us, say an element a, and then we execute the rule R with variable x set to a. Notice that any location that involves x holds `undef`. Intuitively these are new locations. Why not to allow the environment to change some of the new locations during the creation? We do just that. To create an object, we submit the initial values to the demon. The demon chooses a reserve element as the identifier of the new object and then initializes the fields of the new objects on the same occasion.

The initialization update of a new location ℓ does not compete with subsequent updates of ℓ. For example, the rule

```
c := new Horse(‘‘White’’)
c.Color := ‘‘Black’’
```

creates a new white horse and then colors it black. The horse remains white until the end of the current step, and becomes black after that.

The object initialization problem is exacerbated by the need to create groups of mutually dependant objects. For example, you may need to create at once (i) a husband object with a field indicating the wife and (ii) a wife object with a field indicating the husband. Our solution extends naturally to cover such group creations. The demon chooses a group of reserve elements and then initializes all the relevant new locations at once. For example, in the case of the family pair creation, the demon choses two reserve elements, say h and w, and then sets the wife field of h to w and sets the husband field of w to h.

9 Intra-Step Communication

There is also an important problem of intra-step communication: during the execution of one step, you call the outside world which can call you back, etc. All this takes place within one step of the ASM model. This issue will be addressed in [7].

References

1. Andreas Blass and Yuri Gurevich. Background, Reserve, and Gandy Machines. In P. Clote and H. Schwichtenberg, editors, *Proceedings of CSL'2000*, volume 1862 of *LNCS*, pages 1–17. Springer-Verlag, 2000.
2. Andreas Blass, Yuri Gurevich, and Saharon Shelah. Choiceless Polynomial Time. *Annals of Pure and Applied Logic*, 100(1–3):141–187, 1999.
3. Egon Börger and Joachim Schmid. Composition and Submachine Concepts for Sequential ASMs. In P. G. Clote and H. Schwichtenberg, editors, *Computer Science Logic*, volume 1862 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2000.
4. FSE. Foundations of Software Engineering, Microsoft Research. Website. `http://research.microsoft.com/fse/`.
5. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In Egon Boerger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
6. Y. Gurevich. May 1997 Draft of the ASM Guide. Technical Report CSE-TR-336-97, EECS Department, University of Michigan, 1997.
7. Y. Gurevich and W. Schulte. Intra-Step Interaction with Outside World (a tentative title). in preparation.
8. Y. Gurevich and N. Tillmann. Partial Updates: Exploration. to appear in the J. of Universal Computer Science, 2002.
9. A. V. Zamulin. Generic Facilities in Object-Oriented ASMs. In Y. Gurevich, P. W. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications (ASM'2000)*, volume 1912 of *Lecture Notes in Computer Science*, pages 91–111. Springer, 2000.