Distributed Abstract State Machines

Status Report of a Doctoral Thesis

Andreas Glausch
Supervisor: Prof. Dr. Wolfgang Reisig

Humboldt-Universität zu Berlin
Institut für Informatik
glausch@informatik.hu-berlin.de

Abstract. In 1985, Gurevich introduced *Abstract State Machines* (ASMs) as a computational model more powerful and universal than classical computational models. Since then ASMs have been applied successfully both in theory and practice: On the theoretical side, ASMs evolved to a general basis for the study of different classes of algorithms and their expressive power. On the practical side, ASMs have been extended to a full-fledged design methodology, applied successfully in industry for the specification and analysis of real-world systems.
In my doctoral thesis I examine the class of *distributed ASMs* and study their expressive power. I also intend to develop basic notions of refinement and composition for distributed ASMs.

1 Introduction

Abstract State Machines (ASMs) have been introduced as a "computational model that is more powerful and more universal than the standard computational models" by Yuri Gurevich in 1985 [17]. This is achieved by a combination of two of the most elementary notions of computer science and mathematics: *transition systems* and *structures*.

The operational semantics of classical computational models like Turing machines and the λ-calculus are all defined in terms of transition systems, each of which consisting of a set of *states* and a *next-state relation*. For example, the state of a Turing machine is given by the internal control state, the head position, and the tape inscriptions. The next state is then computed according to the transition function of the Turing machine. Another example is the λ-calculus, where each state is represented by a λ-expression, and next states are computed by applying transformation rules to the λ-expression.

The computational model of ASMs fits perfectly into this setting: the operational semantics of ASMs is also defined in terms of transition systems. In contrast to classical computational models, however, ASMs employ the most general representation of states available in today's mathematics: each state of an ASM is a *structure*.

As usual, a structure comprises a set U (its *universe*) together with finitely many functions over U, each with a fixed arity. [1] No additional properties are required: for example, a state of an ASM may include uncomputable functions which in that way become available to the computation of the ASM. Even more, the state may include real-valued functions like sinus and logarithm, which frequently occurs in algorithms of numerical mathematics. Last but not least, functions of a state may be defined over real-world objects which have no symbolic representation at all. As an example, consider the function of a litmus paper which maps each chemical solution to a pH-value.

Given a state of an ASM, the next state is computed by applying an *ASM program* whose syntax reminds of pseudo code. But in contrast to pseudo code, ASM programs provide a formal semantics which is crucial for the formal reasoning about the correctness of programs.

In its most basic version, an ASM program merely consist of conditional assignment statements. For example, the euclidian algorithm is represented by the following ASM program `EUCLID`:

```
if isGreater(a,b) then a:=minus(a,b)
if isGreater(b,a) then b:=minus(b,a)
```

A state of this ASM program is a structure comprising of four functions denoted by the symbols `a`, `b`, `isGreater` and `minus`. The symbols `a`, `b` denote 0-ary functions (that is they take no arguments at all) returning natural numbers. The symbols `isGreater` and `minus` denote binary functions representing the ordering relation and the subtraction operation over natural numbers.

The ASM program `EUCLID` modifies a state by updating the values of `a` or `b` according to the assignment statements. Starting at an initial state and iterating `EUCLID` eventually reaches a state where `a` and `b` contain the greatest common divisor of the values of `a` and `b` at the initial state.

The decisive point is that the above ASM program `EUCLID` does not depend on a particular representation of natural numbers. Instead, `EUCLID` employs natural numbers and operations defined over natural numbers in a straight and natural way. In classical computational models, natural numbers need to be encoded by symbol sequences, for example by particular tape inscriptions or by particular λ-expressions.

Of course, ASMs are not bounded to natural numbers. As indicated already above, the elements and the operations employed by an ASM can be chosen freely. The freedom to model states by arbitrary functions over arbitrary universes lead to a remarkable success of ASMs as a design and analysis methodology [5, 4]. By applying stepwise refinements, large-scale systems from real-world have been modelled and analyzed formally with the help of ASMs. For example, the operational semantics of the SDL-2000 standard is defined by an ASM model [11], and the correctness and completeness of a Java standard compiler has been proven based on a formal ASM model [14].

[1] Usually a structure also includes relations. In the theory of ASMs, relations are represented by boolean-valued functions, thus simplifying some technicalities.

Due to its rigorous mathematical foundations, the theory of ASMs also evolved (and still evolves) to a theory of algorithms in general. In [9], Gurevich axiomatized the class of *sequential algorithms* by only a few, intuitively convincing and amazingly general requirements, and proved that every sequential algorithm can be simulated by a *sequential ASM*. This result is particularly surprising as sequential ASMs merely employ conditional assignment statements as shown above. Later, this result has been extended by Blass and Gurevich to various variants of ASMs, including nondeterministic, parallel, and interactive versions [3, 2, 10].

2 Goals of the Doctoral Thesis

In my doctoral thesis I examine *distributed ASMs* as they have been defined in the "Lipari Guide" [8]: a distributed ASM consists of a set of *agents* which concurrently modify the state. In the course of my doctoral thesis I am going to characterize the expressive power of distributed ASMs. I also intend to develop refinement and composition techniques for distributed ASMs in order to improve their usability for the specification and analysis of large-scale distributed systems. The rest of this section presents these goals in more detail.

As stated in the introduction already, numerous variants of ASMs have been classified during the last years, in particular sequential, nondeterministic, parallel, interactive, and distributed versions of ASMs. All of these versions share the same fundamental idea: a state of an ASM is a structure. These variants, however, differ in their expressive power by enabling a well-defined subset of the following features:

- nondeterministic steps,
- unbounded or infinite amount of change in a step,
- communication with the environment,
- concurrent steps.

In order to improve the understanding of the different versions of ASMs, much effort has been spent to characterize their expressive power formally: For each of the above variants of ASMs, except distributed ASMs, a corresponding class of *algorithm* has been axiomatized, capturing the expressive power of the variant. As a subgoal of my doctoral thesis I am going to extend this result to the class of distributed ASMs. The particular difficulty in the case of distributed ASMs their different nature of state change: state changes in a distributed ASM may occur concurrently, whereas all other variants of ASMs perform state changes in a sequential manner.

The second subgoal of my doctoral thesis is the extension of distributed ASMs by refinement and composition techniques. This goal is crucial in order to establish distributed ASMs as a means to specify and analyze distributed systems of realistic size. For ASMs with sequential behaviors, [5] presents a general notion of refinement and several composition techniques. In my doctoral thesis I intend to adapt and to extend these techniques to the case of distributed ASMs.

3 Current Results

In this section I present the main result achieved so far: We identified a subclass of distributed ASMs as defined in [8] and characterized their expressive power by a corresponding class of distributed algorithm. A detailed and comprehensive presentation of this result is given in [7]. In the following I will briefly sketch the main ideas.

Distributed ASMs, in contrast to other variants of ASMs, perform *locally bounded* and *concurrent* changes of states. For this reason, we adapt the idea of *actions* with locally bounded cause and effect, and the notion of *distributed runs*, in the tradition of Petri [12], Pratt [16], and Gurevich [8]: a distributed run is a set of action occurrences, partially ordered by causal before-after.

In order to describe actions as locally bounded changes of a state it is necessary to decompose the state into local components. A local component of a structure (representing a state) describes the value of a *single* function at a *single* argument. More precisely, a structure canonically decomposes into a set of triples $(\mathtt{f}, \bar{a}, v)$ where $\mathtt{f}$ is a symbol denoting a particular function of the structure, $\bar{a}$ is an argument tuple of the function, and v is the corresponding value of the function at argument $\bar{a}$. We call this triple *store* as it stores the value of the function denoted by $\mathtt{f}$ at the argument $\bar{a}$. Under the name of *updates*, Gurevich identified those triples already to describe changes of a state [8]. We extend their use to describe entire states.

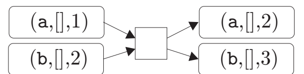
As an example, consider the following structure Q over the universe $U = \{1, 2, 3\}$, consisting of two 0-ary functions denoted by $\mathtt{a}$ and $\mathtt{b}$, and two unary functions denoted by $\mathtt{f}$ and $\mathtt{g}$:

$$\mathtt{a}() = 1 \qquad \mathtt{b}() = 2 \qquad \begin{array}{l} \mathtt{f}(1) = 2 \\ \mathtt{f}(2) = 3 \\ \mathtt{f}(3) = 1 \end{array} \qquad \begin{array}{l} \mathtt{g}(1) = 1 \\ \mathtt{g}(2) = 2 \\ \mathtt{g}(3) = 3 \end{array} .$$

This structure is identified by the following set $\widetilde{Q}$ of stores:

$$\begin{aligned} \widetilde{Q} = \{ \ & (\mathtt{a}, [], 1), (\mathtt{b}, [], 2), \\ & (\mathtt{f}, [1], 2), (\mathtt{f}, [2], 3), (\mathtt{f}, [3], 1), \\ & (\mathtt{g}, [1], 1), (\mathtt{g}, [2], 2), (\mathtt{g}, [3], 3) \ \}. \end{aligned}$$

The structure Q (represented by its set of stores $\widetilde{Q}$) can then be changed locally by modifying a *subset* of the stores in $\widetilde{Q}$. As an example, Q could be changed by replacing the stores $(\mathtt{a}, [], 1)$ and $(\mathtt{b}, [], 2)$ by the stores $(\mathtt{a}, [], 2)$ and $(\mathtt{b}, [], 3)$. This yields another structure Q' where $\mathtt{a}$ and $\mathtt{b}$ denote the values 2 and 3, respectively. We apply the graphical notation of Petri nets and outline this action by



.

Two actions modifying disjoint sets of stores do not interfere with each other, and may therefore be applied concurrently. This implies the notion of *distributed run*: a distributed run is a partially ordered set of action occurrences. We represent this partial order by an inscribed *occurrence net*, a notion well known from the theory of Petri nets (see [13]). As an example, Fig. 1 outlines a distributed run: each square (called *transition*) represents the occurrence of one action replacing the stores attached to the incoming arcs by the stores attached to the outgoing arcs. The transitive closure of the arcs then induces a partial order on the transitions, i.e. a partial order on the action occurrences.

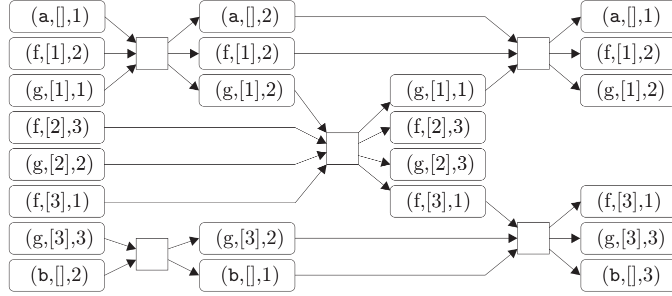


Fig. 1. A distributed run.

Based on the notions introduced so far, we defined the following class of distributed ASMs: A distributed ASM comprises

1. a set of states, where each state is a structure, and
2. a finite set of ASM programs called *agents*.

The agents concurrently modify a state of the distributed ASM by performing actions as introduced above. These actions then compose to a distributed run (as shown in Fig. 1) which represents a concrete behavior of the distributed ASM. We call this variant *basic distributed ASMs* to distinguish them from *general distributed ASMs* as defined in [8].

Basic distributed ASMs are not as expressive as general distributed ASMs: agents cannot access stores of a state concurrently, and agents cannot be added and removed during a run. Nevertheless we were able to characterize the expressive power of basic distributed ASMs by a corresponding class of distributed algorithm [7]. This has not been achieved yet for general distributed ASMs.

4 Open tasks

The results presented in the last section consider only the class of basic distributed ASMs. Currently we extend these results to general distributed ASMs,

which splits into two subtasks. *Firstly*, the operational semantics of basic distributed ASMs (and the corresponding class of distributed algorithm) needs to be generalized such that two agents may access the same store concurrently. This can be achieved by adapting the read arc concept from Petri nets [15] or, alternatively, by applying partial order concepts from trace theory [6]. *Secondly*, the static set of agents of a basic distributed ASM needs to be generalized to a dynamic set that may grow and shrink during a run of the ASM. In that way, a distributed ASM can faithfully model systems which dynamically create and suspend processes (e.g. operating systems, Java threads, ...).

The second goal of my doctoral thesis is the development of refinement and composition techniques for distributed ASM. In [5] a very general notion of refinement is presented: Given two ASMs A and B and an arbitrary equivalence between the states of A and B, A refines B if the runs of A and B respect the equivalence between the states. Furthermore, [5] presents composition concepts like sequential, iterative, recursive, and submachine composition. In the course of my doctoral thesis I intend to adapt and to extend these notions for distributed ASMs. Logic-based approaches ("implementation is implication" and "composition is conjunction") as formulated by Abadi and Lamport [1] may help to achieve this task.

References

1. M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Program. Lang. Syst.*, 17(3):507–535, 1995.
2. A. Blass and Y. Gurevich. Ordinary Interactive Small-Step Algorithms, parts I, II, III. *ACM Trans. Comput. Logic.* to appear.
3. A. Blass and Y. Gurevich. Abstract State Machines Capture Parallel Algorithms. *ACM Trans. Comput. Logic*, 4(4):578–651, 2003.
4. E. Börger. The Origins and the Development of the ASM Method for High Level System Design and Analysis. *Journal of Universal Computer Science*, 8(1):2–74, 2002.
5. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag, 2003.
6. V. Diekert. *The Book of Traces.* World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1995.
7. A. Glausch and W. Reisig. Distributed Abstract State Machines and Their Expressive Power. Informatik-Berichte 196, Humboldt-Universität zu Berlin, Jan 2006. http://www.informatik.hu-berlin.de/top/download/publications/techreport196.pdf.
8. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
9. Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, Jul 2000.
10. Y. Gurevich and T. Yavorskaya. On Bounded Exploration and Bounded Nondeterminism. Technical Report MSR-TR-2006-07, Microsoft Research, Jan 2006.
11. ITU-T. SDL Formal Semantics Definition. ITU-T Recommendation Z.100 Annex F, International Telecommunication Union, Nov 2000.

12. C.A. Petri. Non-Sequential Processes. Interner Bericht ISF–77–5, Gesellschaft für Mathematik und Datenverarbeitung, 1977.
13. W. Reisig. *Petri Nets: An Introduction.* Springer-Verlag New York, Inc., New York, NY, USA, 1985.
14. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer-Verlag, 2001.
15. W. Vogler. Partial order semantics and read arcs. *Theor. Comput. Sci.*, 286(1):33–63, 2002.
16. V.R. Pratt. Modeling Concurrency with Partial Orders. *Int. J. of Parallel Programming*, 15(1):33–71, Feb 1986.
17. Y. Gurevich. A New Thesis. *Abstracts, American Mathematical Society*, page 317, Aug 1985.