View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Agder University Research Archive

Distributed ASM - Pitfalls and Solutions

Andreas Prinz[1] and Edel Sherratt[2]

[1] Department of ICT, University of Agder
andreas.prinz@uia.no
[2] Department of Computer Science, Aberystwyth University
eds@aber.ac.uk

Abstract. While sequential Abstract State Machines (ASM) capture the essence of sequential computation, it is not clear that this is true of distributed ASM. This paper looks at two kinds of distributed process, one based on a global state and one based on variable access. Their commonalities are extracted and conclusions for the general understanding of distributed computation are drawn, providing integration between global state and variable access.

1 Introduction

For many years, models and languages of astonishing variety and depth have been developed to describe distributed computation, and still its essence is far from understood.

Distributed Abstract State Machines (ASM) [8] are a key part of a drive to establish a distributed ASM thesis analogous to the successful sequential ASM thesis [7]. This work has not yet led to a final result, although Glausch and Reisig in [5] have established that distributed algorithms that fulfil certain criteria are captured by DASM.

This paper looks at characteristics of distributed computations and scenarios that are not fully captured by distributed ASM. Based on the work of Lamport [9], a new ASM model is proposed that captures more of these scenarios.

The paper is structured as follows. Section 2 introduces distributed ASM. Section 3 argues that they do not fully capture distributed algorithms. The Lamport model is presented in section 4. Section 5 extracts essential properties of distributed computations, and section 6 proposes how the global view can be combined with the local variables view. Section 7 concludes the paper.

2 Asynchronous Multi-agent (Distributed) ASMs

A distributed ASM (DASM) is a family of pairs $(a; \mathit{Module}(a))$ with pairwise different agents, elements of a possibly dynamic finite set *Agent*, each equipped with a sequential ASM $\mathit{Module}(a)$. Each sequential ASM provides a set of states (first order structures over the same vocabulary), a set of initial states and a state transition function which can only take into account a bounded number of elements.

© Springer-Verlag Berlin Heidelberg 2014

Definition 1 ((global) DASM run). *A partially ordered run of a DASM is a partially ordered set* $(M; \prec)$ *of moves* m *(rule applications) of its agents* $agent(m)$ *with a state function* s *satisfying the following conditions [8]:*

- *finite history: each move has only finitely many predecessors, i.e.* $\{m' \in M | m' \prec m\}$ *is finite for each* $m \in M$.
- *sequentiality of agents: for each agent* a *the set of its moves is linearly ordered, i.e.* $agent(m) = agent(m')$ *implies* $m \prec m'$ *or* $m' \prec m$.
- *coherence: each finite initial segment (downward closed subset)* I *of* $(M; \prec)$ *has an associated state* $s(I)$ *– think of it as the result of all moves in* I *– which for every maximal element* $m_{max} \in I$ *is the result of applying the state transition function of* $agent(m_{max})$ *in state* $s(I - \{m_{max}\})$.

This definition implies a global state accessible to all agents, where each agent has its own local view given by the variables read by the agent. The definition does not say how moves are to be scheduled in a run; moves can be performed in parallel, or by interleaving the moves of different agents. However, every run leads to the same end state.

Proposition 1. *All linearizations of the same finite initial segment of a DASM run have the same final state [8].*

This means that each ASM run is essentially sequential and we conclude.

Proposition 2. *If DASM runs are the most general way to look at distributed computation, then distributed computation is essentially sequential.*

3 Distributed ASM Do Not Capture Distributed Algorithms

The distributed ASM thesis is still open, because there are many distributed scenarios that are not properly captured by distributed ASM.

1. *Context switching* between threads can occur between a read and a write. In ASM, an update is performed instantaneously, which means that the state is read, the answer is computed and the result is written as a single atomic action.
2. In larger distributed systems, *inconsistent system states* are possible. With ASM, the system state is always consistent.
3. In parallel computation, two processors can *simultaneously* write the same memory location. Similarly, a write could be at the same time as a read. The ASM consistency condition [4] excludes such conflict, and a more elaborate treatement by [1] treats memory locations as proclets (active processors) in their own right, that do some computing to resolve write conflicts.
4. The meaning of distributed computations varies a lot according to the level of *atomicity* used. DASM have a fixed level of atomicity.

This brings us to the following conclusion.

Proposition 3 (Failed Distributed ASM Thesis). *DASM as defined in section 2 do not capture distributed computation; at least they do not capture the scenarios given above.*

If certain restrictions are accepted, then DASM do capture some kinds of distributed computation[5]. However, those restrictions conflict with our scenarios. A consistent *global state* cannot be assumed for a highly distributed computation. Context switching between reads and writes conflicts with the assumption of *instantaneous actions*. For *autonomicity*, the update sets of [5] introduce constraints by claiming that the input (read) and output (write) locations should be the same. This leads to the impossibility of parallel read, which is not in line with our understanding.[1] Finally it has to be noted, that the concept of DASM run as introduced in [5] is not the same as the traditional DASM run. In particular, the consistency condition is not introduced, and no proof is given that both concepts coincide. As our examples in section 4 show, these two ideas of DASM run do not coincide.

4 Sequentially Consistent Runs

A different way of looking at distributed computation was introduced by Lamport [9]. Here, a distributed execution is a set of sequential executions (one per agent), each being a sequence of reads and writes of locations. Not all Lamport executions are valid. Lamport defines sequential consistency as follows[9].

Definition 2 (sequential consistency). *Consider a computation (execution) composed of several sequential processors accessing a common memory. The computation is* sequentially consistent *iff the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

A sequentially consistent execution has at least one *witness*, which is a legal interleaving of the reads and writes. Different witnesses may yield different results.

The level of granularity is lower for Lamport reads and writes as opposed to moves for DASM. Reading and writing is implicit in the DASM model. As a contrast, Lamport does not look into the global system state.

4.1 Examples to Compare DASM and Lamport Runs

It might not be obvious how Lamport and DASM runs differ, so we give some small examples with agents A and B, and variables x and y.

[1] Please note that [5] is not altogether consistent at this place. In requirement D4 (autonomicity), the parameter values of the locations could be locations themselves. However, this is not used in the examples shown later. But if there are no locations used as parameters, D4 is trivially true. On the other hand, using locations as parameters, the notion of "same location" suddenly becomes quite advanced.

Table 1. ASM and Lamport witnesses - all examples

	No	witness ASM	witness Lamport	result
Ex1	W1	$m_A; m_B$	$\mathrm{write}(x,1); \mathrm{write}(x,2)$	x=2
	W2	$m_B; m_A$	$\mathrm{write}(x,2); \mathrm{write}(x,1)$	x=1
Ex2	W1	$m_A; m_B$	$\mathrm{read}(x,0); \mathrm{write}(x,1); \mathrm{read}(x,1); \mathrm{write}(x,2)$	x=2
	W2	$m_B; m_A$	$\mathrm{read}(x,0); \mathrm{write}(x,1); \mathrm{read}(x,1); \mathrm{write}(x,2)$	x=2
	W3	–	$\mathrm{read}(x,0); \mathrm{read}(x,1); \mathrm{write}(x,1); \mathrm{write}(x,1)$	x=1
Ex3	W1	$m_A; m_B$	$\mathrm{read}(y,0); \mathrm{write}(x,1); \mathrm{read}(x,1); \mathrm{write}(y,2)$	x=1, y=2
	W2	$m_B; m_A$	$\mathrm{read}(x,0); \mathrm{write}(y,2); \mathrm{read}(y,2); \mathrm{write}(x,1)$	x=1, y=2
	W3	–	$\mathrm{read}(x,0); \mathrm{read}(y,0); \mathrm{write}(x,1); \mathrm{write}(y,2)$	x=1, y=2
	W4	–	$\mathrm{read}(x,0); \mathrm{read}(y,0); \mathrm{write}(y,2); \mathrm{write}(x,1)$	x=1, y=2

Example Ex1 : $A : x := 1$; $B : x := 2$; initially $x = 0$.
ASM: two possible runs: {W1}, {W2}
Lamport: one possible run: {W1, W2}
Example Ex2 : $A : x := x + 1$; $B : x := x + 1$; initially $x = 0$.
ASM: two possible runs: {W1}, {W2}
Lamport: two possible runs: {W1, W2}, {W3}[2]
Example Ex3 : $A : x := y * 0 + 1$; $B : y := x * 0 + 2$; initially $x = y = 0$
ASM: one possible run: {W1, W2}
Lamport: three possible runs: {W1}, {W2}, {W3, W4}[3]

4.2 Distributed ASM Runs Are Sequentially Consistent

Since each move of a DASM run writes the same values, regardless of the linearization, it is possible to translate DASM runs into sequentially consistent Lamport runs. Please note that it is not true that each move reads the same values independent of the linearization, see the last example in the previous section.

Thus, one DASM run can produce several Lamport runs and each Lamport run of a DASM run is sequentially consistent.

5 General Properties of Distributed Computation

Distributed computation generally comprises sequential agents that work together. They may use synchronization of memory locations to coordinate their work. However, it is essential that their work has to respect causality (proper synchronization of writes with reads). When conflicts arise, then there is an underlying mechanism to handle inconsistencies between reads and writes of different agents.

[2] Observe that W3 is not possible in ASM, although it is not conflicting.

[3] The last two runs have one more witness each where the reads are swapped. As opposed to DASM, [5] would not consider W1 and W2 independent but view them as two different runs.

Property 1 (Sequentiality). *The actions of each agent are sequential.* [4]

Property 2 (Synchronization). *There are (global) memory locations where access is sequential.*

Property 3 (Causality). *It is impossible to read values before they have been written.* [5]

Property 4 (Consistency). *When two agents try to write to the same position, then one of them wins as opposed to having arbitrary outcome. In the same way, also possible conflicts between read and write are solved.*

With these requirements in mind, we will describe a local state model that captures our idea of distributed computation.

6 Localized State

We introduce a localized DASM model where a memory location can be updated by one agent, and its value can take some time before it is available to other agents. This is addressed with reference to persistent queries in [2,3], where a query is accompanied by the location where its result is to be deposited.

Definition 3 ((localized) DASM run). *A localized partially ordered run of a DASM is a partially ordered set $(M; \prec)$ of moves m (rule applications) of its agents $agent(m)$ with a state function s satisfying the following conditions:*

1. *finite history: see definition 2*
2. *sequentiality of agents: see definition 2*
3. *The (local) states of an agent before and after a move m are related using the state transition function of $agent(m)$.*
4. *The (local) state of an agent before a move is a combination of all the (local) states after the directly preceding moves. If there are no preceding moves, an initial state is used.*
5. *A combination of two (local) states is done with the following rules.*
   - *When the value of a location is the same in both states, then this value is taken.*
   - *When the value of a location is different in the two states, then the one resulting from the later move with respect to the partial order is taken.*
   - *When the value of a location is different in the two states and both values are coming from moves that are not ordered by the partial order, then an arbitrary value of the two is chosen.*

The new definition brings the following advantages, in particular related to the problems given earlier.

[4] Although this property looks innocent enough, it is rejected by the Java memory model for distributed computation [6].

[5] This property is also called no-out-of-thin-air in the context of Java.

1. *Context switching* is implicit in the new model, since the write of a move is taken into account first when the new read is done.
2. Since agents work independent of each other and each agent has its own local state, a global *inconsistent system states* is not only possible but normal.
3. Concurrent *write at the same time* onto the same memory location is possible and would result in one of the written values.
4. Reads and writes are the level of *atomicity*.
5. The new definition does not guarantee sequential consistency. Please note that all examples from section 4 will be captured in one run using the localized model. In all three cases, the moves of agents A and B can be unordered.
6. The new definition provides a higher level of abstraction than Lamport and at the same time brings less restrictions to the runs. It aligns better with the moves of ASM.

7 Summary and Conclusions

In this paper, we have shown that distributed computation is not always easy to understand and that DASM do not capture the essence of distributed computation. We have compared DASM with the Lamport model and have extracted a new model that is not sequential in the bottom. This local state model captures at least the problems indicated with the DASM model.

References

1. Blass, A., Gurevich, Y.: Abstract state machines capture parallel algorithms. ACM Transactions on Computational Logic (TOCL) 4(4), 578–651 (2003)
2. Blass, A., Gurevich, Y.: Persistent queries (2008)
3. Blass, A., Gurevich, Y.: Persistent queries in the behavioral theory of algorithms. ACM Transactions on Computational Logic (TOCL) 12(2), 1–43 (2011)
4. Börger, E., Stärk, R.: Abstract State Machines – a Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
5. Glausch, A., Reisig, W.: An ASM-characterization of a class of distributed algorithms. In: Abrial, J.-R., Glässer, U. (eds.) Rigorous Methods for Software Construction and Analysis. LNCS, vol. 5115, pp. 50–64. Springer, Heidelberg (2009)
6. Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A.: The Java language specification Java SE 7 edition (2013), `http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf`
7. Gurevich, Y.: The sequential ASM thesis.The Logic in Computer Science Column. Bulletin of European Association for Theoretical Computer Science (1999)
8. Gurevich, Y.: Evolving algebras 1993: Lipari guide. In: Börger (ed.) Specification and Validation Methods. Oxford University Press (1995)
9. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers 28(9) (September 1979)