

An ASM-Characterization of a Class of Distributed Algorithms

Andreas Glausch and Wolfgang Reisig

Humboldt-Universität zu Berlin
Institut für Informatik
{glausch,reisig}@informatik.hu-berlin.de

Abstract. Conventional computation models restrict to particular data structures to represent states of a computation, e.g. natural numbers, sequences, stacks, etc. Gurevich's *Abstract State Machines (ASMs)* take a more liberal position: *any* first-order structure may serve as a state. In [7] Gurevich characterizes the expressive power of *sequential ASMs*: he defines the class of *sequential algorithms* by means of only a few, amazingly general requirements and proves this class to be equivalent to sequential ASMs.

In this paper we generalize Gurevich's result to *distributed algorithms*: we define a class of distributed algorithms by likewise general requirements and show that this class is covered by a distributed computation model based on sequential ASMs.

1 Introduction

Conventional computation models include a distinguished notion of *states*. For example, a state of a Turing machine is captured by the head position, the internal control state, and the tape inscription. Another example is the λ -calculus where each state is represented by a λ -expression. In both cases the representation of states is based on a restricted set of data structures: alphabets, natural numbers, and sequences over alphabets.

Gurevich's Abstract State Machines (ASMs) [6] feature a more liberal representation of states: each state is a *structure*, a notion well known from first-order logic. As usual, a structure comprises a nonempty set U (its *universe*) together with finitely many functions defined over U , each with a fixed arity. No additional properties are required: *any* structure may serve as a state of an ASM. For example, a state may include uncomputable functions or real-valued functions such as *sin* and *log*.

As a consequence, a computation of an ASM may *a priori* employ mathematical concepts such as real numbers, vectors, graphs, geometrical objects, etc. (Of course, this also holds for classical data structures such as stacks, lists, and queues.) Conventional computation models such as Turing machines usually require a particular encoding in order to represent such concepts, if possible at all.

The freedom to include arbitrary data structures as part of the state of an ASM allows for a natural and flexible modeling of the states of a system. For this reason ASMs have been extended to a successfully applied design and analysis methodology [4]. By use of stepwise refinement and composition of ASMs, large-scale systems from real-world have been modeled and analyzed formally. For example, the operational semantics of the SDL-2000 standard officially is defined by an ASM model [9], and the correctness and completeness of a Java standard compiler has been proven based on a formal ASM model [13].

2 Scope and Contribution of this Paper

Classically, ASMs employ a simple pseudo-code like syntax to describe state changes [6]. Gurevich revealed that ASMs may also be characterized independently of any concrete syntax: In [7] Gurevich defines the class of *sequential algorithms* and proves this class to be equivalent to the class of *sequential ASMs* (c.f. also [12]). Blass, Gurevich, and others identified further, more general classes of algorithms – including *nondeterministic*, *parallel*, and *interactive* versions – and showed that each class is equivalent to a corresponding class of ASMs [8, 1–3].

In this paper we contribute to this work by a corresponding result for *distributed algorithms*: we define a class of distributed algorithms by five simple and general requirements and show that this class is equivalent to a distributed computation model based on ASMs.

Distributed algorithms significantly differ from the aforementioned variants of algorithms: a sequential/nondeterministic/parallel/interactive algorithm computes for each state S a successor state S' , i.e. state changes occur sequentially ordered and may involve the complete state S . By contrast, a distributed algorithm performs *concurrent* and *locally bounded* changes of the state. In order to represent such state changes we adapt the idea of *actions* with locally limited cause and effect, and the notion of *distributed runs*, in the tradition of Petri [10], Pratt [14], and Gurevich [6]: a distributed run is a set of action occurrences, partially ordered by causal dependencies.

The variant of distributed algorithms examined in this paper still has some limitations. For the sake of simplicity, we restrict attention to communication via a shared state, and do not consider message passing explicitly. Furthermore, we do not consider dynamic instantiation or disposing of agents. The main contribution of this paper is a first step towards an ASM-based theory of distributed algorithms, leaving much room for generalizations.

The rest of this paper is organized as follows: Section 3 recalls elementary notions on structures and introduces the general framework of *actions* and *distributed runs*. Section 4 defines the notion of *distributed algorithms*. Section 5 introduces *distributed ASMs*, an operational computation model for distributed algorithms. Finally, we show in Sect. 6 that distributed ASMs are expressive enough to capture every distributed algorithm as discussed in Sect. 4.

3 The Basic Framework

In this section we motivate and exemplify the basic notions employed in the rest of this paper. We recall some basic notions on structures, and introduce the concept of actions and distributed runs.

As usual, a *signature* Σ is used to address the functions of a structure: Σ consists of finitely many functions symbols $\mathbf{f}_1, \dots, \mathbf{f}_k$ and determines for each \mathbf{f}_i an arity $n_i \in \mathbb{N}$. A structure S consisting of the functions f_1, \dots, f_n is a Σ -*structure* if the arity of f_i is n_i for $i = 1, \dots, n$, i.e. the arities of the symbols in Σ and the arities of the functions in S coincide. In this case the function f_i is the *interpretation* of the symbol \mathbf{f}_i in S , denoted by \mathbf{f}_{iS} .

As already indicated in the introduction, each state of an ASM is a structure. We therefore use the notions structure and state interchangeably. As a running example throughout this paper we consider the following structure Q , with universe $U = \{1, 2, 3\}$, consisting of two 0-ary functions \mathbf{a}_Q and \mathbf{b}_Q , and two unary functions \mathbf{inc}_Q and \mathbf{val}_Q :

$$\begin{array}{llll} & & \mathbf{inc}_Q(1) = 2 & \mathbf{val}_Q(1) = 1 \\ \mathbf{a}_Q = 1 & \mathbf{b}_Q = 2 & \mathbf{inc}_Q(2) = 3 & \mathbf{val}_Q(2) = 2 \\ & & \mathbf{inc}_Q(3) = 1 & \mathbf{val}_Q(3) = 3. \end{array}$$

We represent a signature by a sequence of function symbols followed by a sequence of their respective arities. Obviously, the signature of Q is $\Sigma_Q = (\mathbf{a}, \mathbf{b}, \mathbf{inc}, \mathbf{val}, 0, 0, 1, 1)$.

In order to represent distributed computation on a Σ -structure S , we require means to describe locally bounded and concurrent changes of S . It turns out useful not to consider S as a monolithic entity but as a set of Σ -*molecules*, each of which consisting of a location and a value. A *location* of S consists of a n -ary function symbol \mathbf{f} and a n -ary argument tuple \bar{a} over the universe of S . For example $(\mathbf{val}, [1])$ is a location of the structure Q (we enclose the argument tuple in square brackets for the sake of readability). Obviously, each location (\mathbf{f}, \bar{a}) of S defines a unique value $v = \mathbf{f}_S(\bar{a})$. The triple (\mathbf{f}, \bar{a}, v) is a Σ -*molecule* of S (or simply *molecule* if this causes no confusion). For example, $(\mathbf{val}, [1], 1)$ is a molecule of the structure Q . Intuitively, this molecule states that “the function denoted by \mathbf{val} maps the argument tuple $[1]$ to the value 1”.

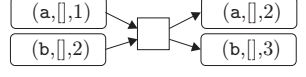
A structure S is completely described by its set of molecules. For example, the above structure Q is represented by the following set of molecules:

$$\begin{aligned} Q = \{ & (\mathbf{a}, [], 1), (\mathbf{b}, [], 2), \\ & (\mathbf{inc}, [1], 2), (\mathbf{inc}, [2], 3), (\mathbf{inc}, [3], 1), \\ & (\mathbf{val}, [1], 1), (\mathbf{val}, [2], 2), (\mathbf{val}, [3], 3) \}. \end{aligned}$$

Calling them “updates”, Gurevich employed molecules already in [6] to describe differences between structures.

A structure S is changed locally by applying an *action* which replaces some of the molecules of S . For example, the above structure Q can be changed by replacing the molecules $(\mathbf{a}, [], 1)$ and $(\mathbf{b}, [], 2)$ by the molecules $(\mathbf{a}, [], 2)$ and $(\mathbf{b}, [], 3)$.

This yields another structure Q' , interpreting the symbols **a** and **b** by 2 and 3, respectively. We apply the graphical notation of Petri nets and outline this action by



In general, a Σ -*action* (or simply *action* if Σ is clear from the context) is a pair $a = (a^{\text{in}}, a^{\text{out}})$, where a^{in} and a^{out} are sets of Σ -molecules such that

- the locations of the molecules in a^{in} and a^{out} coincide, (1)
- both a^{in} and a^{out} are *consistent*: a set of molecules is consistent if it (2)
does not contain two different molecules with the same location.

Hence, an action a only modifies the values of the molecules, but not their locations (1), and both a^{in} and a^{out} determine at most one value for each location (2).

An action a then performs a *step* $S \xrightarrow{a} S'$ by replacing the molecules a^{in} by the molecules a^{out} , thus changing the state S to a new state S' . More precisely, for two states S and S' we write $S \xrightarrow{a} S'$ to denote that

$$- a^{\text{in}} \subseteq S, \tag{3}$$

$$- S' = (S \setminus a^{\text{in}}) \cup a^{\text{out}}, \tag{4}$$

$$- S \text{ and } S' \text{ have the same universe.} \tag{5}$$

That is, a^{in} constitutes the *local cause* of a (3), whereas a^{out} constitutes the *local effect* of a (4). Furthermore, actions are required to preserve the universe of the state (5).

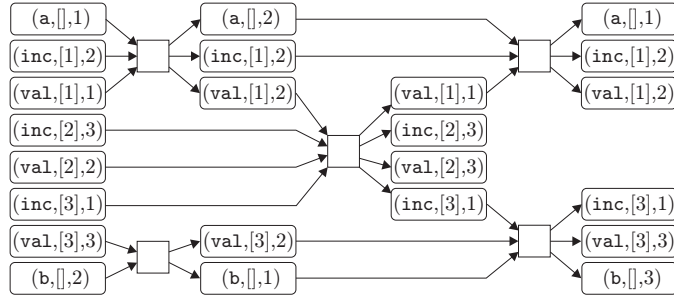


Fig. 1. The distributed run R

Two actions replacing disjoint molecules 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 [11]). Figure 1 outlines an example of a distributed run R : each square (called *transition*) represents the occurrence of one action replacing the molecules attached to the incoming arcs by the molecules attached to the outgoing arcs. An important property of occurrence nets is that each molecule is attached to at most one incoming and at most one outgoing arc. Consequently, each molecule is accessed resp. modified by at most one action, i.e. molecules cannot be accessed or modified concurrently. The transitive closure of the arcs then induces a partial order on the transitions, i.e. a partial order on the action occurrences.

In Fig. 1, the rounded rectangles inscribed by the molecules are called *places*. Observe that the leftmost places in R hold the molecules of the structure Q , which is the *initial state* of this run. Further (global) states of this run are identified by the notion of *cut*. A cut C is a maximal set of unordered places of R such that only finitely many transitions precede the places in C . Figure 2 shows three different cuts, C_1 , C_2 , and C_3 , each of which represented by a dashed line. The inscriptions of the places in a cut C then always constitute a Σ -structure S , as exemplified in Fig. 2. This structure S is the global state of the run after the occurrence of all actions preceding the places in C . Note that the cuts C_1 and C_2 in Fig. 2 represent a *step*, as only a single action occurs between C_1 and C_2 .

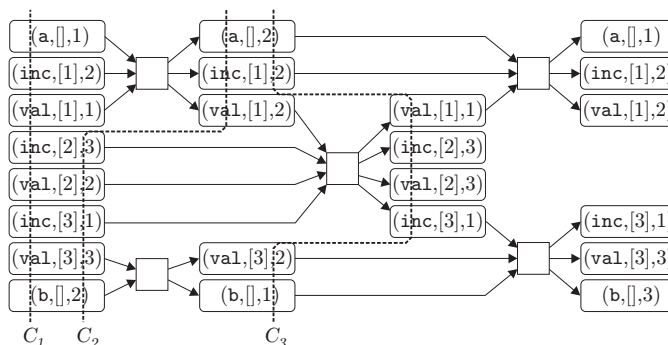


Fig. 2. Three different cuts C_1 , C_2 , and C_3 of the distributed run R

In general, a *distributed run* over a signature Σ is an occurrence net with places inscribed by molecules such that

- the molecules at the places without an incoming arc constitute a Σ -structure S_0 , (the *initial state* of the run)
- all molecules contain only elements of the universe of S_0 ,
- each transition t represents an action a : the places at the incoming/outgoing arcs of t are inscribed by the molecules in $a^{\text{in}}/a^{\text{out}}$.

Due to lack of space, we skip the (somewhat technical) formal definition of distributed runs and cuts, and rely on their intuitive graphical notation as shown in Fig. 1. For the technical details, we refer to [5].

4 Distributed Algorithms

Based on the framework introduced in the previous section, this section defines the class of *distributed algorithms* by five requirements which are fairly general, nevertheless simple and intuitive. We briefly discuss the reasonability for each of these requirements. In addition, for the ASM expert, we briefly present for each requirement its relationship to Gurevich's original requirements to sequential algorithms [7].

4.1 A State of a Distributed Algorithm is a Structure

As emphasized by Tarski already, mathematical structures are general enough to faithfully describe any static mathematical entity on any level of abstraction. Consequently, it is legitimate to assume that every state of an algorithm can also be described naturally by a structure. As an algorithm should always have a finite representation, a single signature (with a finite set of symbols) suffices for all states. Furthermore, in a computational framework, it is common to mark some of the states as initial. This leads to the following first requirement:

Requirement D1 (state requirement) *A distributed algorithm \mathcal{D} determines a non-empty set $\mathcal{S}_{\mathcal{D}}$ of states and a non-empty set $\mathcal{I}_{\mathcal{D}} \subseteq \mathcal{S}_{\mathcal{D}}$ of initial states. All states in $\mathcal{S}_{\mathcal{D}}$ are structures over the same signature $\Sigma_{\mathcal{D}}$.*

This requirement reflects Gurevich's *abstract state* requirement, where each state of a sequential algorithm is required to be a Σ -structure.

4.2 Distributed Algorithms Perform Actions

According to common intuition about distributed computing, state changes in a distributed algorithm occur locally bounded and occasionally concurrent. The concept of *actions* as introduced in Sect. 3 captures such state changes in a natural and general way. Therefore, the state changes of a distributed algorithm \mathcal{D} can always be described naturally by a set $\mathcal{A}_{\mathcal{D}}$ of actions.

Of course, applying an action of \mathcal{D} to a state of \mathcal{D} should always yield a state of \mathcal{D} again. Furthermore, for technical reasons, we require every action of \mathcal{D} be executable in at least one state of \mathcal{D} .

Requirement D2 (action requirement) *A distributed algorithm \mathcal{D} determines a set $\mathcal{A}_{\mathcal{D}}$ of actions over signature $\Sigma_{\mathcal{D}}$ such that for each action $a \in \mathcal{A}_{\mathcal{D}}$ holds:*

- if $S \in \mathcal{S}_{\mathcal{D}}$ and $S \xrightarrow{a} S'$ then $S' \in \mathcal{S}_{\mathcal{D}}$,
- there is a state $S \in \mathcal{S}_{\mathcal{D}}$ such that $S \xrightarrow{a} S'$.

For a state $S \in \mathcal{S}_{\mathcal{D}}$ and an action $a \in \mathcal{A}_{\mathcal{D}}$, $S \xrightarrow{a} S'$ is a *step of \mathcal{D}* . A *distributed run of \mathcal{D}* is a distributed run R (see Sect. 3) such that the initial state of R is an initial state of \mathcal{D} and each action occurring in R is an action of \mathcal{D} .

Requirement D2 is an adaption of Gurevich’s *sequential time* requirement, which demands each sequential algorithm to determine a set of global steps, represented as a next-state function τ . We replace global steps by local actions here.

4.3 Distributed Algorithms Respect Isomorphism

As usual, a bijective mapping $i : U_R \rightarrow U_S$ between the universes of two Σ -structures R and S is an *isomorphism between R and S* (written $i : R \rightarrow S$) iff $i(\mathbf{f}_R(u_1, \dots, u_n)) = \mathbf{f}_S(i(u_1), \dots, i(u_n))$ for all n -ary function symbols \mathbf{f} in Σ and all $u_1, \dots, u_n \in U_R$. Isomorphic structures only differ in the concrete representation of the elements of the universe, whereas the functions of both structures are essentially the same.

For an algorithm the concrete representation of elements should be inessential. For example, the Euclidean algorithm computes the greatest common divisor regardless whether the integers are represented by transistor states on a chip or by ink on a paper. In general, a distributed algorithm should not distinguish isomorphic states, i.e. should behave “isomorphic” at isomorphic states.

More precisely, if an action a can occur in a state R isomorphic to a state S , a corresponding isomorphic action can occur in S . To formalize this, we extend any isomorphism $i : R \rightarrow S$ canonically to molecules, sets of molecules, and actions. The third requirement then reads:

Requirement D3 (isomorphism requirement) *The sets of states $\mathcal{S}_{\mathcal{D}}$ and $\mathcal{J}_{\mathcal{D}}$ of a distributed algorithm \mathcal{D} both are closed under isomorphism. Furthermore, if $R \xrightarrow{a} R'$ is a step of \mathcal{D} and $i : R \rightarrow S$ is an isomorphism, then there exists also a step $S \xrightarrow{i(a)} S'$ of \mathcal{D} .*

This requirement is an adoption of a part of Gurevich’s *abstract state* requirement which demands the next-state function τ to map isomorphic states to isomorphic next-states.

4.4 Actions of Distributed Algorithms Operate Autonomously

The next requirement can be formulated simply and intuitively convincing: each action a of a distributed algorithm only uses elements that a has access to, i.e. a operates autonomously on data. In other words, an action of a distributed algorithm does not obtain elements from nowhere.

It remains to clarify the above meaning of *use* and *have access*. In technical terms, an action a uses an element x if x occurs in an argument tuple in one of the molecules of a (where it is used to access a location), or if x occurs as a value in one of the molecules in a^{out} (where it is used as a newly assigned value).

As the locations of a^{in} and a^{out} are identical, we define: a uses x iff there is a molecule $(\mathbf{f}, [u_1, \dots, u_n], v) \in a^{\text{out}}$ such that $x \in \{u_1, \dots, u_n, v\}$.

Furthermore, a has access to the value of a molecule $m \in a^{\text{in}}$ if a has access to the elements in the argument tuple of m . This leads to the following inductive definition:

- for each molecule $(\mathbf{x}, [], v) \in a^{\text{in}}$, a has access to v ,
- if a has access to u_1, \dots, u_n and $(\mathbf{f}, [u_1, \dots, u_n], v) \in a^{\text{in}}$ for $n \geq 1$, then a has access to v .

The fourth requirement is now quite obvious:

Requirement D4 (autonomy requirement) *For each action a of a distributed algorithm \mathcal{D} , a has access to each element that is used by a .*

As D4 relies heavily on the notion of action introduced in this paper, there is no direct counterpart in Gurevich’s requirements. However, D4 can be seen as a part of the *bounded exploration* requirement where ground terms are used to characterize the next-state function of sequential algorithms: the inductive evaluation of ground terms corresponds to the above, inductive definition of “having access to”.

4.5 Actions of Distributed Algorithms are Bounded

A real-world processor (e.g., a computer, an organization, or a human being) executing an algorithm can obviously perform only a bounded amount of work in each step. For this reason it is quite natural to require the actions of a distributed algorithm to be bounded in size. This idea leads to the fifth and last requirement:

Requirement D5 (bounded-actions requirement) *For a distributed algorithm \mathcal{D} there exists a constant $c \in \mathbb{N}$ such that for each action a of \mathcal{D} holds $|a^{\text{in}}| \leq c$ (which is equivalent to $|a^{\text{out}}| \leq c$).*

Similar to D4, this requirement can be seen as a part of Gurevich’s *bounded exploration* requirement which demands the next-state function τ to be characterized by a *finite* (hence, a bounded) set of ground terms.

We do not demand any further requirements: we call *any* entity satisfying the Requirements D1–D5 a distributed algorithm.

5 Distributed Abstract State Machines

In the previous section we introduced the class of distributed algorithms in a purely semantical and declarative way. But usually algorithms are represented in an explicit and syntactical form, e.g. by program code or by natural language. This gives rise to the question whether a given distributed algorithm can be represented in a syntactical way at all. In this section we answer this questions

positively by presenting the computation model of *distributed ASMs*, which is based on sequential ASMs [6].

The version of distributed ASMs we introduce here is a special case of the version presented in the Lipari Guide [6]. There, a distributed ASM includes a set of *agents*, each of which is executing an *ASM program*. The agent set may grow and shrink during computation, thus allowing dynamic instantiation and disposing of agents. By contrast, we consider only a fixed set of agents, where each agent is identified by the program it executes. Furthermore, in [6] Gurevich introduces a highly general notion of distributed run, which allows several agents to access the same location of a state concurrently. As discussed in Sect. 3, concurrent access is not possible in our version of distributed runs.

Despite those restrictions, our version of distributed ASMs is justified by the following fact: *Any* distributed algorithm as discussed in the previous section can be represented by one of our distributed ASMs. We will prove this fact in the upcoming Sect. 6.

5.1 Assignment statements

As usual, a signature Σ yields Σ -terms: each 0-ary symbol from Σ is a Σ -term, and for an n -ary symbol $\mathbf{f} \in \Sigma$ and given Σ -terms t_1, \dots, t_n , the symbol sequence $\mathbf{f}(t_1, \dots, t_n)$ is a Σ -term, too. Such terms are *evaluated* by a Σ -structure S in the usual way: for each 0-ary symbol \mathbf{a} , the element \mathbf{a}_S denotes the *evaluation* of \mathbf{a} in S , and for a term $\mathbf{f}(t_1, \dots, t_n)$, its evaluation in S is defined inductively by $\mathbf{f}(t_1, \dots, t_n)_S =_{\text{def}} \mathbf{f}_S(t_{1S}, \dots, t_{nS})$.

Terms are used to form *assignment statements*. A simple example built from signature Σ_Q is the statement $\mathbf{val}(\mathbf{a}) := \mathbf{inc}(\mathbf{b})$. Executing this assignment statement in state Q updates the value of function \mathbf{val}_Q at argument $\mathbf{a}_Q = 1$ by the value of $\mathbf{inc}(\mathbf{b})_Q = 3$, i.e. replaces the molecule $(\mathbf{val}, [1], 1)$ by the molecule $(\mathbf{val}, [1], 3)$.

The general form of an assignment statement α is $t := t'$, where t, t' are Σ -terms. Applied to a state S , α updates the location specified by t by the value of t' . More precisely, for $t = \mathbf{f}(t_1, \dots, t_n)$, the *location of t in S* is

$$\text{loc}_S(t) =_{\text{def}} (\mathbf{f}, [t_{1S}, \dots, t_{nS}]).$$

Then α replaces the molecule set α_S^{old} by the molecule set α_S^{new} , with

$$\begin{aligned} \alpha_S^{\text{old}} &=_{\text{def}} \{ (\text{loc}_S(t), t_S) \} \\ \alpha_S^{\text{new}} &=_{\text{def}} \{ (\text{loc}_S(t), t'_S) \}. \end{aligned}$$

That is, α_S^{old} and α_S^{new} contain the modified molecule *before* and *after* executing the assignment statement α , respectively.

5.2 Assignment Statements Generate Actions

Given an assignment statement α and a corresponding state S , it is tempting to regard $(\alpha_S^{\text{old}}, \alpha_S^{\text{new}})$ as the *action* executed by α in state S . Unfortunately, this

does not work, as this action would only consider the molecules *modified* by α , but not the molecules *accessed* by α .

In the distributed case, accessed molecules are significant: two assignment statements obviously cannot be executed concurrently if one of them modifies a location accessed by the other one. As a simple example, consider the two assignment statements $\mathbf{a}:=\mathbf{b}$ and $\mathbf{b}:=\mathbf{a}$.

Hence, the action generated by α needs to consider all molecules involved, i.e. all molecules that are modified or accessed. To formalize this idea, for a Σ -term $t = \mathbf{f}(t_1, \dots, t_n)$ and a Σ -structure S , we define the *set of involved molecules* t_S^{in} inductively as

$$t_S^{\text{in}} =_{\text{def}} t_1^{\text{in}} \cup \dots \cup t_n^{\text{in}} \cup \{ (\text{loc}_S(t), t_S) \}.$$

For an assignment statement $\alpha : t := t'$, the set of involved molecules then is defined as

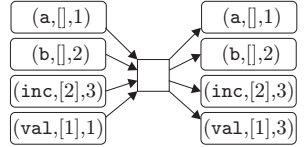
$$\alpha_S^{\text{in}} =_{\text{def}} t_S^{\text{in}} \cup t'_S{}^{\text{in}}.$$

More precisely, the set α_S^{in} contains all involved molecules *before* the execution of α . Correspondingly, the set

$$\alpha_S^{\text{out}} =_{\text{def}} (\alpha_S^{\text{in}} \setminus \alpha_S^{\text{old}}) \cup \alpha_S^{\text{new}}.$$

contains all involved molecules *after* the execution of α . Notice the different meanings of the superscripts: $^{\text{in}}$ and $^{\text{out}}$ denote *all* involved molecules before and after the execution of α , whereas $^{\text{old}}$ and $^{\text{new}}$ denote only the *modified* molecules before and after the execution of α .

The notion of *action of α* now is quite obvious: for a given state S , the action of α in state S is defined by $\alpha_S =_{\text{def}} (\alpha_S^{\text{in}}, \alpha_S^{\text{out}})$. As an example, executing the assignment statement $\mathbf{val}(\mathbf{a}) := \mathbf{inc}(\mathbf{b})$ in state Q yields the action



5.3 Guarded Assignment Statements

An assignment statement α may furthermore be guarded by a *Boolean expression* β . In that way, α is executed only in states that satisfy β .

In technical terms, a Boolean expression β consists of several *term equations* of the form $t_1 = t_2$ connected by the usual Boolean connectives \neg , \wedge , and \vee . For a given state S , the truth value of β is computed in the obvious way, where $S \models \beta$ denotes that β holds in S . For an assignment statement α , **if β then α** is a *guarded assignment statement*, γ . For technical convenience, every assignment statement as introduced in the previous subsections is conceived as a guarded assignment statement whose guard holds in every state.

In addition to an ordinary assignment statement, a guarded assignment statement γ involves further molecules to evaluate the truth value of the guard β . Formally, let T^β denote the set of all terms occurring in β . For a state S satisfying β , the action of γ then is defined by $\gamma_S =_{\text{def}} (\gamma_S^{\text{in}}, \gamma_S^{\text{out}})$ with

$$\gamma_S^{\text{in}} =_{\text{def}} \alpha_S^{\text{in}} \cup \bigcup_{t \in T^\beta} t_S^{\text{in}} \quad , \quad \gamma_S^{\text{out}} =_{\text{def}} (\gamma_S^{\text{in}} \setminus \alpha_S^{\text{old}}) \cup \alpha_S^{\text{new}}.$$

5.4 Sequential ASM Programs

Guarded assignment statements can be executed in parallel. In general, a finite, non-empty set of guarded assignment statements $\Gamma = \{\gamma_1, \dots, \gamma_n\}$ is a *sequential ASM program*. At a given state S , Γ executes simultaneously all assignment statements whose guards hold in S . More precisely, Γ replaces the molecules

$$\Gamma_S^{\text{old}} =_{\text{def}} \bigcup \{ \alpha_S^{\text{old}} \mid (\text{if } \beta \text{ then } \alpha) \in \Gamma \text{ and } S \models \beta \}$$

by the molecules

$$\Gamma_S^{\text{new}} =_{\text{def}} \bigcup \{ \alpha_S^{\text{new}} \mid (\text{if } \beta \text{ then } \alpha) \in \Gamma \text{ and } S \models \beta \}.$$

Note that Γ_S^{new} may be *inconsistent*, i.e. may contain two different molecules with the same location. For example, executing the sequential ASM program $\Gamma = \{\mathbf{f}(\mathbf{x}) := \mathbf{u}, \mathbf{f}(\mathbf{y}) := \mathbf{v}\}$ in a state S with $\mathbf{x}_S = \mathbf{y}_S$ and $\mathbf{u}_S \neq \mathbf{v}_S$ yields an inconsistent Γ_S^{new} . As an inconsistent set of molecules cannot update a state, Γ executes no action in that case.

For a state S such that Γ_S^{new} is non-empty (i.e. at least one guard is satisfied) and consistent, Γ performs the action $\Gamma_S =_{\text{def}} (\Gamma_S^{\text{in}}, \Gamma_S^{\text{out}})$ with

$$\Gamma_S^{\text{in}} =_{\text{def}} \bigcup_{\gamma \in \Gamma} \gamma_S^{\text{in}} \quad , \quad \Gamma_S^{\text{out}} =_{\text{def}} (\Gamma_S^{\text{in}} \setminus \Gamma_S^{\text{old}}) \cup \Gamma_S^{\text{new}}.$$

In this case, Γ_S is called an *action of Γ* . The set of all actions of Γ is denoted by \mathcal{A}_Γ .

As an important property, no two actions from \mathcal{A}_Γ can be executed concurrently. The reason is that all actions in \mathcal{A}_Γ share the locations of the 0-ary function symbols occurring in Γ . As a consequence, the actions of a sequential ASM program Γ always occur totally ordered in a distributed run.

5.5 Distributed Abstract State Machines

In this section we finally introduce *distributed ASMs*. A distributed ASM Δ specifies a set of sequential ASM programs, the *components* of Δ . These components concurrently change the state of Δ by performing actions as introduced above. More precisely, a *distributed ASM* Δ consist of

- a signature Σ_Δ ,

- a non-empty set \mathcal{S}_Δ of Σ_Δ -structures closed under isomorphism (the *states* of Δ),
- a non-empty set $\mathcal{J}_\Delta \subseteq \mathcal{S}_\Delta$ closed under isomorphism (the *initial states* of Δ),
- a finite, non-empty set \mathcal{P}_Δ of sequential ASM programs (the *components* of Δ), all built over signature Σ_Δ .

Based on the notions introduced so far, the operational semantics of Δ is easy to define: the actions of Δ are constituted by the actions of the components of Δ , i.e. the *set of actions of Δ* is defined as

$$\mathcal{A}_\Delta =_{\text{def}} \bigcup_{\Gamma \in \mathcal{P}_\Delta} \mathcal{A}_\Gamma.$$

A *distributed run of Δ* then is a distributed run R such that the initial state of R is an initial state of Δ and every action occurring in R is an action of Δ .

As an example, consider the following two sequential ASM programs built over signature Σ_Q :

$$\begin{aligned} A &= \{ \text{val}(a) := \text{inc}(\text{val}(a)) , a := \text{inc}(a) \}, \\ B &= \{ \text{if } (b = \text{val}(b)) \text{ then } b := \text{inc}(b) \}. \end{aligned}$$

Intuitively, A subsequently increases the values of the function val , with a as the argument counter, whereas B increases the counter b as long as the values of b and $\text{val}(b)$ coincide.

Figure 3 shows a distributed run of the components A and B at the initial state Q . For the sake of clarity, each occurring action is inscribed by the program causing the action.

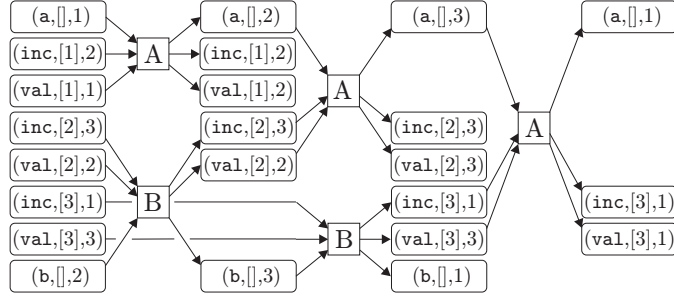


Fig. 3. A distributed run of the components A and B

6 The Distributed ASM Theorem

In this section we present the main result of this paper: every distributed algorithm as introduced in Sect. 4 can be represented by a distributed ASM as introduced in Sect. 5. More precisely, the following theorem holds:

Theorem 1. *Let \mathcal{D} be a distributed algorithm according to D1–D5. Then there exists a distributed ASM Δ such that the distributed runs of \mathcal{D} and Δ coincide.*

The reverse of Theorem 1 also holds: it is easy to prove that every distributed ASM constitutes a distributed algorithm by verifying the requirements D1–D5. Hence, the expressive power of distributed algorithms and distributed ASMs coincide.

In the rest of this paper we outline the proof of Theorem 1. We present the most important lemmata and provide for each lemma an idea of the proof. Additionally, we show for each of the requirements D1–D5 its applications in the course of the proof. A much more detailed proof of Theorem 1 (based on slightly different notations) is given in [5].

The main idea of the proof is the following: according to D2, a distributed algorithm \mathcal{D} specifies a set of actions $\mathcal{A}_{\mathcal{D}}$. We then decompose $\mathcal{A}_{\mathcal{D}}$ into finitely many equivalence classes and represent each of these classes by a sequential ASM program.

The equivalence relation that decomposes $\mathcal{A}_{\mathcal{D}}$ is *action isomorphism*: two actions a and b are *isomorphic* if b can be derived from a by bijectively replacing the elements of a . Formally, the *set of elements* of an action a is defined as

$$E(a) =_{\text{def}} \{ u_1, \dots, u_n, v \mid (\mathbf{f}, [u_1, \dots, u_n], v) \in a^{\text{in}} \cup a^{\text{out}} \}.$$

For two actions a and b , extend every bijective function $p : E(a) \rightarrow E(b)$ canonically to the molecules of a and to a itself. Then $p(a)$ denotes the action derived from a by replacing all elements according to p . The function p then is an *action isomorphism from a to b* iff $p(a) = b$. In case there is an action isomorphism from a to b , a and b are *isomorphic*, written $a \cong b$. We denote by $[a] =_{\text{def}} \{ b \mid b \cong a \}$ the *isomorphism class of a* .

The relation \cong is an equivalence relation between actions, i.e. \cong decomposes $\mathcal{A}_{\mathcal{D}}$ into disjoint subsets. Furthermore, the following lemma holds:

Lemma 1. *Let \mathcal{D} be a distributed algorithm. Then \cong decomposes $\mathcal{A}_{\mathcal{D}}$ into finitely many disjoint subsets.*

Idea of proof. This lemma holds due to D5: for each action $a \in \mathcal{A}_{\mathcal{D}}$, $|a^{\text{in}}|$ and $|a^{\text{out}}|$ are bounded by a constant $c \in \mathbb{N}$. As a consequence, there exists a *finite* set M such that every action $a \in \mathcal{A}_{\mathcal{D}}$ is isomorphic to an action a_M whose molecules contain only elements from M . But as M is finite, there can be only finitely many different actions a_M . As each equivalence class of $\mathcal{A}_{\mathcal{D}}$ is represented by an action a_M , $\mathcal{A}_{\mathcal{D}}$ contains only finitely many equivalence classes. \square

The next lemma states that $\mathcal{A}_{\mathcal{D}}$ is closed under action isomorphism: for each action a of \mathcal{D} , each action b isomorphic to a is also an action of \mathcal{D} .

Lemma 2. *Let \mathcal{D} be a distributed algorithm and let $a \in \mathcal{A}_{\mathcal{D}}$. Then $[a] \subseteq \mathcal{A}_{\mathcal{D}}$.*

Idea of proof. According to D2, there is a step $R \xrightarrow{a} R'$ of \mathcal{D} . Let b be an action isomorphic to a with an action isomorphism $p : a \rightarrow b$. Based on p , a state S and an isomorphism $i : R \rightarrow S$ can be constructed such that $p \subseteq i$. According to D3, there is a step $S \xrightarrow{i(a)} S'$ of \mathcal{D} . As $i(a) = p(a) = b$, b is an action of \mathcal{D} . \square

The following lemma states that the molecules of an action $a \in \mathcal{A}_{\mathcal{D}}$ may be characterized by a finite set of terms T .

Lemma 3. *Let $S \xrightarrow{a} S'$ be a step of \mathcal{D} . Then there exists a finite set T of Σ -terms such that*

- $a^{\text{in}} = \bigcup_{t \in T} t_S^{\text{in}}$,
- for each molecule $(l, v) \in a^{\text{out}}$ there are terms $t^l, t^v \in T$ such that $\text{loc}_S(t^l) = l$ and $t_S^v = v$.

Idea of proof. According to D4, for every molecule $(\mathbf{f}, [u_1, \dots, u_n], v) \in a^{\text{out}}$, a has access to the elements u_1, \dots, u_n and v . Then, for each $x \in \{u_1, \dots, u_n, v\}$, a term t^x along the inductive definition of “having access to” is constructed such that $t_S^x = x$. The set of all terms constructed in this way then constitutes T .

Let $(l, v) \in a^{\text{out}}$ with $l = (\mathbf{f}, [u_1, \dots, u_n])$. Then for the term $t^l =_{\text{def}} \mathbf{f}(t^{u_1}, \dots, t^{u_n})$ holds $\text{loc}_S(t^l) = l$, and for the term t^v holds $t_S^v = v$. \square

Based on the set T of terms obtained in Lemma 3, a sequential ASM program Γ is constructed which performs all actions in the isomorphism class of a :

Lemma 4. *Let a be an action of \mathcal{D} . Then there exists a sequential ASM program Γ such that $\mathcal{A}_{\Gamma} = [a]$.*

Idea of proof. Let $S \xrightarrow{a} S'$ be a step of \mathcal{D} and let T be as in Lemma 3. By use of Lemma 3, for each molecule $(l, v) \in a^{\text{out}}$ construct the assignment statement $t^l := t^v$. The set of all assignment statements obtained in this way is an ASM program Γ^0 with $\Gamma_S^0 = a$. The demanded ASM program Γ is constructed from Γ^0 by guarding the assignment statements in Γ^0 by the Boolean expression $\bigwedge_{t, t' \in T, t_S = t'_S} (t = t') \wedge \bigwedge_{t, t' \in T, t_S \neq t'_S} \neg(t = t')$. Due to this guard, Γ only executes actions isomorphic to a . \square

The previous lemmata then are composed in the main proof of Theorem 1:

Idea of proof (of Theorem 1). According to Lemma 1, the equivalence relation \cong decomposes $\mathcal{A}_{\mathcal{D}}$ into finitely many disjoint subsets C_1, \dots, C_n . According to Lemma 2, each C_i is closed under isomorphism. Then, according to Lemma 4, for each C_i there exists a sequential ASM program Γ_i such that $C_i = \mathcal{A}_{\Gamma_i}$. For the distributed ASM Δ with $\mathcal{S}_{\Delta} = \mathcal{S}_{\mathcal{D}}$, $\mathcal{J}_{\Delta} = \mathcal{J}_{\mathcal{D}}$, and $\mathcal{P}_{\Delta} = \{\Gamma_1, \dots, \Gamma_n\}$ then holds

$$\mathcal{A}_{\mathcal{D}} = C_1 \cup \dots \cup C_n = \mathcal{A}_{\Gamma_1} \cup \dots \cup \mathcal{A}_{\Gamma_n} = \mathcal{A}_{\Delta}.$$

As all states, initial states, and actions of \mathcal{D} and Δ coincide, the distributed runs of \mathcal{D} and Δ are the same. \square

7 Conclusion

The theory of ASMs suggests a comprehensive and quite general approach to the notion of “algorithm”. A number of variants of ASMs have been identified, among them sequential, interactive, parallel, and distributed versions. The

deeper understanding of all such classes of ASMs requires a characterization of their expressive power. This has been achieved for many of them, including sequential, parallel, and interactive versions.

In this paper we characterized a class of distributed algorithms with bounded actions. As suggested by Gurevich in [6], and in the tradition of Petri and Pratt, we define distributed runs as sets of action occurrences, partially ordered by causal dependencies. We furthermore showed that this class of distributed algorithm is captured by the operational computation model of distributed ASMs, which is based on sequential ASMs [7].

We intend to extend the result of this paper to more general variants of distributed ASMs. As discussed in Sect. 5, in [6] a version of distributed ASMs is introduced which offers advanced features such as concurrent access to locations and dynamic instantiation of new agents. We will examine how the requirements D1 – D5 can be generalized in order to capture those features.

References

1. Andreas Blass and Yuri Gurevich. Abstract State Machines Capture Parallel Algorithms. *ACM Trans. Comput. Logic*, 4(4):578–651, 2003.
2. Andreas Blass and Yuri Gurevich. Ordinary Interactive Small-Step Algorithms, parts I, II, III. *ACM Trans. Comput. Logic*, 2006.
3. Andreas Blass, Yuri Gurevich, Dean Rosenzweig, and Benjamin Rossman. General Interactive Small Step Algorithms. Technical Report MSR-TR-2005-113, Microsoft Research, Aug 2006.
4. Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
5. Andreas Glausch and Wolfgang Reisig. Distributed Abstract State Machines and Their Expressive Power. Informatik-Berichte 196, Humboldt-Universität zu Berlin, January 2006.
6. Yuri Gurevich. Evolving Algebras 1993: Lipari Guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
7. Yuri Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, Jul 2000.
8. Yuri Gurevich and Tatiana Yavorskaya. On Bounded Exploration and Bounded Nondeterminism. Technical Report MSR-TR-2006-07, Microsoft Research, Jan 2006.
9. ITU-T. SDL Formal Semantics Definition. ITU-T Recommendation Z.100 Annex F, International Telecommunication Union, Nov 2000.
10. Carl Adam Petri. Non-Sequential Processes. Interner Bericht ISF-77-5, Gesellschaft für Mathematik und Datenverarbeitung, 1977.
11. Wolfgang Reisig. *Petri Nets: An Introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.
12. Wolfgang Reisig. On Gurevich’s Theorem on Sequential Algorithms. *Acta Informatica*, 39(5):273–305, 2003.
13. Robert Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.
14. Vaughan Pratt. Modeling Concurrency with Partial Orders. *Int. J. of Parallel Programming*, 15(1):33–71, Feb 1986.