A Characterization of Distributed ASMs with Partial-order Runs

Egon Börger[1], Klaus-Dieter Schewe[2]

[1] Università di Pisa, Dipartimento di Informatica, Pisa, Italy, boerger@di.unipi.it
[2] Zhejiang University, UIUC Institute, Haining, China, kdschewe@acm.org

Abstract. To overcome the practical limitations of partial-order runs of 'distributed ASMs' (Abstract State Machines) proposed by Gurevich, we have defined a concept of concurrent runs of multi-agent ASMs and could show that concurrent ASMs capture a natural language-independent axiomatic definition of concurrent algorithms, thus generalising Gurevich's seminal 'Sequential ASM Thesis' from sequential to concurrent algorithms. However, we remained intrigued by the fact that Blass and Gurevich used partial-order runs of distributed ASMs to explain runs of sequential recursive algorithms. We discovered that also the inverse simulation holds: for every distributed ASM with partial order runs, these runs can be described by runs of a sequential recursive algorithm. This surprising result clarifies the difference in expressivity between partial-order and concurrent runs.

1 Introduction

In [8, Sect.2-3] the concept of sequential Abstract State Machines (*seq-ASMs*) has been defined for which the 'Sequential ASM Thesis' [7]—to capture the intuitive notion of sequential algorithm—could be proved from three natural postulates, see [9]. In [8, Sect.6] the concept of sequential ASM runs is extended by *partial-order runs* of a specific class of multi-agent ASMs called *distributed ASMs*. However, contrary to the great variety of successful applications of sequential ASMs, the use of distributed ASMs with partial-order runs turned out to be impractical to adequately model concurrent systems. It has been replaced in [4] by a language-independent axiomatic characterization of concurrent runs, adding a fourth postulate (on the intuitive meaning of concurrency), together with a definition of concurrent ASMs, based upon which the Sequential ASM Thesis and its proof could be generalized to a Concurrent ASM Thesis—to capture the proposed intuitive notion of concurrent algorithms.

In reaction to some scepticism expressed in [13], whether recursive algorithms can be adequately defined by ASMs, partial-order runs of *distributed ASMs* have been used in [1] to simulate the computations of recursive algorithms.[3] For a long time we have been intrigued by this proposal, since on the one side, a simple

[3] Already the definition of recursive ASMs in [10] uses a special case of this translation of recursive into *distributed* computations.

sequential extension of ASMs suffices for the specification of recursive algorithms (see for example [2]), on the other side partial-order runs of *distributed ASMs* turned out to be impractical for modeling truly concurrent systems (see [4]).

In Sect. 3 we review Gurevich's description of *distributed ASMs* with partial-order runs and analyse the proof that the runs of recursive algorithms can be defined as partial-order runs of distributed ASMs. The analysis reveals that the *distributed ASMs* used to define recursive runs by partial-order runs are finitely composed concurrent ASMs with non-deterministic sequential (nd-seq) components (see the definition in Sect.3). In Sect.4 we show the surprising discovery that also the inverse relation holds, namely: for every finitely composed concurrent algorithm with nd-seq components, if its concurrent runs are definable by partial-order runs, then the algorithm can be simulated by a recursive algorithm. This establishes the main result of this paper.

Theorem 1.1 (Main Theorem). *Recursive algorithms are behaviourally equivalent to finitely composed concurrent algorithms $\mathcal{C}$ with nd-seq components such that all concurrent $\mathcal{C}$-runs are definable by partial-order runs.*[4]

The equivalence of runs of recursive ASMs and of partial-order runs of distributed ASMs makes it explicit in which sense concurrent ASM runs as characterized in [4] are more expressive than the 'partial-order runs of distributed ASMs' proposed in [8, Sect.6].

We will also show that if the concurrent runs are restricted further to partial-order runs of a concurrent algorithm with a fixed finite number of agents and fixed non-deterministic sequential (nd-seq) programs, one can simulate them even by a non-deterministic sequential algorithm. An interesting example of this special case are partial-order runs of Petri nets and more generally of Mayr's Process Rewrite Systems [12].

For the proofs we use an axiomatic characterization of recursive algorithms as sequential algorithms enriched by call steps,[5] such that the parent-child relationship between caller and callee defines well-defined shared locations representing input and return parameters. This characterization is reviewed in Sect.2 and is taken from [5] where it appears as Recursion Postulate and is added to Gurevich's three postulates for sequential ASMs [9] as basis for the proof of an ASM thesis for recursive ASMs.

We assume the knowledge of [8], [9] and [4] and use without further explanations standard textbook notations for ASMs, including ambient ASMs [3, Ch.4.1].

[4] We call $\mathcal{R}$ behaviourally equivalent to $\mathcal{C}$ if each $r \in \mathcal{R}$ can be simulated by a $c \in \mathcal{C}$ and vice versa.

[5] To emphasize the sequential nature of recursive algorithms we sometimes use the term 'sequential recursive algorithm'. See [5] for the technical reason for this naming policy.

2 The Recursion Postulate

We start with a characteristic example to illustrate the intuitive idea of recursion which guided the formulation of the recursion postulate below.[6] Take the *mergesort* algorithm, which consists of a main algorithm *sort* and an auxiliary algorithm *merge*. Every call to (a copy, we also say an instance of) *sort* and every call to (an instance of) the *merge* algorithm could give rise to a new agent. However, these agents only interact by passing input parameters and return values, but otherwise operate on disjoint sets of locations. In addition, a calling agent always waits to receive return values, which implies that only one or (in case of parallel calls) a finite number of agents are active in any state.

If one considers mutual recursion, then this becomes slightly more general, as there is a finite family of algorithms calling (instances of) each other. Furthermore, there may be several simultaneous calls. E.g. in *mergesort*, *sort* calls two copies of itself, each sorting one half of the list of given elements. Such simultaneously called copies may run sequentially in one order or the other, in parallel or even asynchronously. This give rise to non-deterministic execution of multiple sequential algorithms.

Therefore, for a characterization of recursive algorithms and their computations we can rely on the capture of non-deterministic sequential algorithms by non-deterministic sequential ASMs.[7] Thus, to axiomatically define recursive algorithms and their runs it suffices to add to the three postulates for nd-seq algorithms a Call Step Postulate and a Recursive Run Postulate defined below, which together form the **Recursion Postulate**.

To characterize the input/output relation between the input provided by the caller in a call step and the output computed by the callee for this input we use the ASM function classification from [6] to distinguish between *input*, *output* and *local* (also called controlled) function symbols in the signature, the union of pairwise disjoint sets Σ_{in}, Σ_{out} and Σ_{loc} respectively. We call any nd-seq algorithm which comes with such a signature and also satisfies the Call Step Postulate below an *algorithm with input and output* (for short: *i/o-algorithm*). We can then define (sequential) recursive algorithms syntactically as collections of i/o-algorithms.

Definition 2.1. A *recursive algorithm* $\mathcal{R}$ is a finite set of i/o-algorithms with one distinguished *main* algorithm. The elements of $\mathcal{R}$ are called components of $\mathcal{R}$.

The independency condition for (possibly parallel) computations of different instances of the given algorithms requires that for different calls, in particular for different calls of the same algorithm, the state spaces of the triggered subcomputations are separated from each other. This encapsulation of subcomputations can be made precise by the concept of ambient algorithms where each *instance of*

[6] For a detailed analysis see [5].

[7] The proof for the Sequential ASM Thesis is easily extended from deterministic to non-deterministic algorithms, see [9, Sect.9.2].

an algorithm has a unique context parameter for its functions, e.g. its executing agent (see [3, Ch.4.1]), and is started in an initial state that only depends on its input locations.[8]

Now we are ready to formulate the postulate for call steps. In Sect.3.4 we formalize this postulate by an ASM $\textsc{Call}(t_0 \leftarrow N(t_1, \ldots, t_n))$ (see Definition 3.4 and its refinement in Sect.4).

Postulate 1 (Call Step Postulate) When an i/o-algorithm p—the caller, viewed as parent algorithm—calls a finite number of i/o-algorithms $c_1, \ldots, c_n$—the callees, viewed as child algorithms $\mathit{CalledBy}(p)$—a *call relationship* (denoted as $\mathit{CalledBy}(p)$) holds between the caller and each callee. The caller activates a fresh instance of each callee c_i so that they can start their computations. These computations are independent of each other and the caller remains waiting—i.e. performs no step—until every callee has terminated its computation (read: has reached a final state). For each callee, the initial state of its computation is determined only by the input passed by the caller; the only other interaction of the callee with the caller is to return in its final state an output to p.

Definition 2.2. A *call relationship* holds for (instances of) two i/o-algorithms $\mathcal{A}^p$ (parent) and $\mathcal{A}^c$ (child) if and only if they satisfy the following conditions on their function classification:

- $\Sigma_{in}^{\mathcal{A}^c} \subseteq \Sigma^{\mathcal{A}^p}$ so that the parent algorithm is able to update input locations of the child algorithm. Furthermore, $\mathcal{A}^p$ never reads the input locations of $\mathcal{A}^c$.
- $\Sigma_{out}^{\mathcal{A}^c} \subseteq \Sigma^{\mathcal{A}^p}$ so that the parent algorithm can read the output locations of the child algorithm. Furthermore, $\mathcal{A}^p$ never updates output locations of $\mathcal{A}^c$.
- $\Sigma_{loc}^{\mathcal{A}^c} \cap \Sigma^{\mathcal{A}^p} = \emptyset$ (no other common locations).

Differently from runs of a nd-seq algorithm, where in each state at most one step of the nd-seq algorithm is performed, in a recursive run a sequential recursive algorithm $\mathcal{R}$ can perform in one step simultaneously one step of each of finitely many not terminated and not waiting called instances of its i/o-algorithms. This is expressed by the Recursive Run Postulate. In this postulate we refer to *Active* and not *Waiting* instances of components, which are defined as follows:

Definition 2.3. To be *Active* resp. *Waiting* in a state S is defined as follows:

$\mathit{Active}(q)$ **iff** $q \in \mathit{Called}$ **and not** $\mathit{Terminated}(q)$
$\mathit{Waiting}(p)$ **iff forsome** $c \in \mathit{CalledBy}(p)$ $\mathit{Active}(c)$
$\mathit{Called} = \{\mathit{main}\} \cup \bigcup_p \mathit{CalledBy}(p)$

[8] More precisely, one can define an instance of an algorithm $\mathcal{A}$ by adding a parameter a, say for an agent executing the instance $\mathcal{A}_a = (a, \mathcal{A})$ of $\mathcal{A}$. a can be used as environment parameter for the evaluation $\mathit{val}_S(t, a)$ of a term t in state S with the given environment. This yields for different agents a, a' different functions $f_a, f_{a'}$ as interpretation of the same function symbol f, so that the run-time interpretations of a common signature element f can be made to differ for different agents, due to different inputs which determine their initial states.

Called collects the instances of algorithms that are called during the run. The subset of *Called* which contains all the children called by p is denoted by $CalledBy(p)$. $Called = \{main\}$ and $CalledBy(p) = \emptyset$ are true in the initial state S_0, for each i/o-algorithm $p \in \mathcal{R}$. In particular, in S_0 the original component *main* is considered to not be $CalledBy(p)$, for any p.

Postulate 2 (Recursive Run Postulate) For a sequential recursive algorithm $\mathcal{R}$ with main component *main* a recursive run is a sequence $S_0, S_1, S_2, \ldots$ of states[9] together with a sequence $C_0, C_1, C_2, \ldots$ of sets of instances of components of $\mathcal{R}$ which satisfy the following constraints:

Recursive run constraint.

- C_0 is the singleton set $C_0 = \{main\}$, i.e. every run starts with *main*,
- every C_i is a finite set of instances of components of $\mathcal{R}$ which are *Active* and not *Waiting* in state S_i,
- every S_{i+1} is obtained in one $\mathcal{R}$-step by performing in S_i simultaneously one step of each i/o-algorithm in C_i. Such an $\mathcal{R}$-step is also called a *recursive step* of $\mathcal{R}$.

Bounded call tree branching. There is a fixed natural number $m > 0$, depending only on $\mathcal{R}$, which in every $\mathcal{R}$-run bounds the number of callees which can be called by a call step.

Remark (on Call Trees). If in a recursive $\mathcal{R}$-run the main algorithm calls some i/o-algorithms, this call creates a finitely branched call tree whose nodes are labeled by the instances of the i/o-algorithms involved, with active and not waiting algorithms labeling the leaves and with the main (the parent) algorithm labeling the root of the tree and becoming waiting. When the algorithm at a leaf makes a call, this extends the tree correspondingly. When the algorithm at a child of a node has terminated its computation, we delete the child from the tree. The leaves of this (dynamic) call tree are labeled by the active not waiting algorithms in the run. When the main algorithm terminates, the call tree is reduced again to the root labeled by the initially called main algorithm.

Usually, it is expected that for recursive $\mathcal{R}$-runs each called i/o-algorithm reaches a final state, but in general it is not excluded that this is not the case.

In [5] the reader can find a definition of recursive ASMs together with a proof that they capture (are equivalent to) recursive algorithms as characterized by the Recursion Postulate. Here we use the postulate as a basis for the proof that recursive algorithms are captured by 'distributed ASMs with partial-order runs', as defined in [8].

[9] For the sake of simplicity we take a state as union of the states of the component instances in the run, in other words as state over the union of the individual signatures.

3 Recursive ASMs are distributed ASMs with partial-order runs

Syntactically, a multi-agent (also called concurrent) algorithm $\mathcal{C}$ is defined as a family of algorithms $alg(a)$, each associated with ('indexed by') an agent $a \in Agent$ that executes the algorithm in a run. Each $(a, alg(a))$ resp. $alg(a)$ is called a component resp. (component) program of $\mathcal{C}$. This applies to distributed ASMs [8] as well as to recursive or concurrent algorithms and ASMs [4],[5].

To investigate the simulation of recursive runs by partial-order runs of distributed ASMs (Sect. 3.4) we must explain what are finitely composed concurrent (Gurevich's 'distributed') algorithms (Sect.3.1) and partial-order resp. concurrent runs (Sect.3.2 resp.3.3).

3.1 Finitely composed concurrent algorithms

For recursive algorithms various restrictions on the syntactical definition of multi-agent algorithms have to be made most of which appear also for distributed ASMs in [8, Sect.6].

First of all, although the components $alg(a)$ of concurrent algorithms are not necessarily sequential algorithms, to simulate specific concurrent algorithms by recursive ones, which are defined as families of nd-seq algorithms, we must restrict our attention to concurrent algorithms with sequential (though possibly non-deterministic) components.[10]

Second, for distributed ASMs it is stipulated in [8, p.31] that the agents are equipped with instances of programs which are taken from 'a finite indexed set of single-agent programs'. This leads to what we call finitely composed concurrent algorithms or ASMs $\mathcal{C}$ where the components can only be copies (read: instances) of finitely many different nd-seq algorithms or ASMs, which we will call the program base of $\mathcal{C}$.

Third, for distributed ASMs it is stipulated in [8, 6.2, p.31] that in initial states there are only finitely many agents, each equipped with a program. We reflect this by the (simplifying but equivalent) condition that the runs of a finitely composed concurrent algorithm or ASM must be started by executing a distinguished main component.

Fourth, for distributed ASMs it is stipulated in [8, p.32] that 'An agent a can *make a move* at S by firing $\mathrm{Prog}(a)$ ... and change S accordingly. As part of the move, a may create new agents', which then may contribute by their moves to the run in which they were created. For this purpose we use the **new** function.

We summarize these constraints for distributed ASMs by the notion of *finitely composed* concurrent algorithms (read: concurrent ASMs).

Definition 3.1. A concurrent algorithm $\mathcal{C}$ is *finitely composed* iff (i)-(iii) hold:

[10] In fact it is shown in [5] that permitting the unbounded **forall** and **choose** constructs results in algorithms far more powerful than the recursive ones.

(i) There exists a finite set $\mathcal{B}$ of nd-seq algorithms such that each $\mathcal{C}$-program is of form **amb** a **in** r for some program $r \in \mathcal{B}$—call $\mathcal{B}$ the *program base* of $\mathcal{C}$.
(ii) There exists a distinguished agent a_0 which is the only one *Active* in any initial state. Formally this means that in every initial state of a $\mathcal{C}$-run, $Agent = \{a_0\}$ holds. We denote by *main* the component in $\mathcal{B}$ of which a_0 executes an instance. For partial-order runs of $\mathcal{C}$ defined below this implies that they start with a minimal move which consists in executing the program $\mathrm{asm}(a_0) =$ **amb** a_0 **in** *main*.
(iii) Each program in $\mathcal{B}$ may contain rules of form **let** $a =$ **new** $(Agent)$ **in** r. Together with (ii) this implies that every agent, except the distinguished a_0, before making a move in a run must have been created in the run.

$\mathcal{C}$ is called *finite* iff *Agent* is finite.

3.2 Partial-order runs

In [8] Gurevich defined (for distributed algorithms) the notion of *partial-order run* by a partial order on the set of single moves of the agents which execute the component algorithms. For a nd-seq algorithm $\mathcal{A}$, to make one *move* means to perform one step in a state S.

Definition 3.2. Let $\mathcal{C} = \{(a, \mathrm{alg}(a))\}_{a \in Agent}$ be a concurrent algorithm, in which each $\mathrm{alg}(a)$ is an nd-seq algorithm. A *partial-order run* for $\mathcal{C}$ is defined by a set M of moves of instances of the algorithms $\mathrm{alg}(a)$ $(a \in Agent)$, a function $\mathrm{ag} : M \to Agent$ assigning to each move the agent performing the move, a partial order $\leq$ on M, and an initial segment function σ such that the following conditions are satisfied:

finite history. For each move $m \in M$ its history $\{m' \mid m' \leq m\}$ is finite.
sequentiality of agents. The moves of each agent are ordered, i.e. for any two moves m and m' of one agent $\mathrm{ag}(m) = \mathrm{ag}(m')$ we either have $m \leq m'$ or $m' \leq m$.
coherence. For each finite initial segment $M' \subseteq M$ (i.e. such that for $m \in M'$ and $m' \leq m$ we also have $m' \in M'$) there exists a state $\sigma(M')$ over the combined signatures of the algorithms $(a, \mathrm{alg}(a))$ such that for each maximum element $m \in M'$ the state $\sigma(M')$ is the result of applying m to $\sigma(M' - \{m\})$.

3.3 Concurrent runs

In a concurrent run as defined in [4], multiple agents with different clocks may contribute by their single moves to define the successor state of a state. Therefore, when a successor state S_{i+1} of a state S_i is obtained by applying to S_i multiple update sets U_a with agents a in a finite set $Agent_i \subseteq Agent$, each U_a is required to have been computed by $a \in Agent_i$ in a preceding state S_j, i.e. with $j \leq i$. It is possible that $j < i$ holds so that for different agents different $\mathrm{alg}(a)$-execution speeds (and purely local subruns to compute U_a) can be taken into account.

This can be considered as resulting from a separation of a step of an nd-seq algorithm alg(a) into a *read step*—which reads location values in a state S_j—followed by a *write step* which applies the update set U_a computed on the basis of the values read in S_j to a later state S_i ($i \geq j$). We say that a contributes to updating the state S_i to its successor state S_{i+1}, and that a *move* starts in S_j and contributes to updating S_i (i.e. it finishes in S_{i+1}). This is formally expressed by the following definition of concurrent ASMs and their runs.

Definition 3.3. Let $\mathcal{C}$ be a concurrent algorithm of component algorithms $pgm(a)$ (read: ASM rules) with associated agents $a \in Agent$. A *concurrent run* of $\mathcal{C}$ is defined as a sequence $S_0, S_1, \ldots$ of states together with a sequence $A_0, A_1, \ldots$ of finite subsets of $Agent$, such that S_0 is an initial state and each S_{i+1} is obtained from S_i by applying to it the updates computed by the agents in A_i, where each $a \in A_i$ computes its update set U_a on the basis of the location values (including the input and shared locations) read in some preceding state S_j (i.e. with $j \leq i$) depending on a.

Remark. In this definition we deliberately permit the set of *Agent*s to be infinite or dynamic and potentially infinite, growing or shrinking in a run. In Definition 3.2 above, the set of *Agent*s is fixed by the set M of moves.

3.4 Simulation of recursive by partial-order runs

We are now ready to specify recursive algorithms by *distributed ASMs*, following the thought proposed in [1]. For the sake of precision and simplicity we formulate the construction in terms of ASMs; due to the characterization theorems in [5] and[4] this implies no loss of generality.

Theorem 3.1. *Every recursive ASM $\mathcal{R}$ can be simulated by a finitely composed concurrent ASM $\mathcal{C}_\mathcal{R}$ with nd-seq ASM components for which every concurrent run of $\mathcal{C}_\mathcal{R}$ is definable by a partial-order run.*

Proof. Let $\mathcal{R}$ be a recursive ASM given with distinguished program *main*. We define a finitely composed concurrent ASM $\mathcal{C}_\mathcal{R}$ with program base $\{r^* \mid r \in \mathcal{R}\}$, where r^* is defined as

$r^* =$ **if** $Active(r)$ **and not** $Waiting(r)$ **then** r.

In doing so, for each call rule $r = t_0 \leftarrow N(t_1, \ldots, t_n)$ in $\mathcal{R}$ we use for its translation the following ASM $\textsc{Call}(t_0 \leftarrow N(t_1, \ldots, t_n))$, which rigorously defines the behavioral interpretation of the call rule r (for details see [5]):

Definition 3.4. $\textsc{Call}(t_0 \leftarrow N(t_1, \ldots, t_n)) =$
let $N(x_1, \ldots, x_n) = q$ // declaration of N
let $v_1 = t_1, \ldots, v_n = t_n$ // input evaluation $\mathrm{val}_S(t_i, \mathbf{self})$ by caller
let $t_0 = f(t'_1, \ldots, t'_k)$
let $v'_1 = t'_1, \ldots, v'_k = t'_k$
let $c =$ **new** $(Agent)$

$pgm(c) :=$ **amb** c **in** q // equip callee with its program instance
INSERT(c, *CalledBy*(**self**))
INITIALIZE($q_c, v_1/x_1, \ldots, v_n/x_n, f(v'_1, \ldots, v'_k)/x_o$)
$CalledBy(c) := \emptyset$

Note that the call is a call-by-value and that $(f, (v'_1, \ldots, v'_k))$ denotes the output location whose value the caller expects to be updated by the callee with the return value.

By definition, r^* can only contribute a non-empty update set to form a state S_{i+1} in a concurrent run, if r is *Active* and not *Waiting*; this reflects that by the recursive run postulate, in every step of a recursive run of $\mathcal{R}$ only *Active* and not *Waiting* rules are executed.

The definition of r^* obviously guarantees that $\mathcal{C}_\mathcal{R}$ simulates $\mathcal{R}$ step by step: in each run step the same *Active* and not *Waiting* rules r respectively r^* and their agents are selected for their simultaneous execution and their rules perform the same state change.

Note that by definition 3.4 of CALL(*i/o-rule*), each agent operates in its own state space so that the view of an agent's step as read-step followed by a write-step is equivalent to the atomic view of this step. Note also that in a concurrent run of $\mathcal{C}_\mathcal{R}$ the *Agent* set is dynamic, in fact it grows with each execution of a call rule, together with the number of instances of $\mathcal{R}$-components executed during a recursive run of $\mathcal{R}$.

It remains to define every concurrent run $(S_0, A_0), (S_1, A_1), \ldots$ of $\mathcal{C}_\mathcal{R}$ by a partial-order run. For this we define an order on the set M of moves made during a concurrent run, showing that it satisfies the constraints on finite history and the sequentiality of agents, and then relate each state S_i of the run to the state computed by the set M_i of moves performed to compute S_i (from S_0), showing that M_i is a finite initial segment of M and that the associated state $\sigma(M_i)$ equals S_i and satisfies the coherence condition.

Each successor state S_{i+1} in a concurrent run of $\mathcal{C}_\mathcal{R}$ is the result of applying to S_i the write steps of finitely many moves of agents in A_i. This defines the function ag, which associates agents with moves, and the finite set M_i of all moves finished in a state belonging to the initial run segment $[S_0, \ldots, S_i]$. Let $M = \cup_i M_i$. The partial order $\leq$ on M is defined by $m < m'$ iff move m contributes to update some state S_i (read: finishes in S_i) and move m' starts reading in a later state S_j with $i + 1 \leq j$. Thus, by definition, M_i is an initial segment of M.

To prove the finite history condition, consider any $m' \in M$ and let S_j be the state in which it is started. There are only finitely many earlier states $S_0, \ldots, S_{j-1}$, and in each of them only finitely many moves m can be finished, contributing to update S_{j-1} or an earlier state.

The condition on the sequentiality of the agents follows directly from the definition of the order relation $\leq$ and from the fact that in a concurrent run, for every move $m = (read_m, write_m)$ executed by an agent, this agent performs no other move between the $read_m$-step and the corresponding $write_m$-step in the run.

This leaves us to define the function σ for finite initial segments $M' \subseteq M$ and to show the coherence property. We define $\sigma(M')$ as result of the application of the moves in M' in any total order extending the partial order $\leq$. For the initial state S_0 we have $\sigma(\emptyset) = S_0$. This implies the definability claim $S_i = \sigma(M_i)$.

The definition of σ is consistent for the following reason. Whenever two moves $m \neq m'$ are incomparable, then either they both start in the same state or say m starts earlier than m'. But m' also starts earlier than m finishes. This is only possible for agents $ag(m) = a$ and $ag(m') = a'$ whose programs $pgm(a), pgm(a')$ are not in an ancestor relationship in the call tree. Therefore these programs have disjoint signatures, so that the moves m and m' could be applied in any order with the same resulting state change.

To prove the coherence property let M' be a finite initial segment, and let $M'' = M' \setminus M'_{\max}$, where $M'_{\max}$ is the set of all maximal elements of M'. Then $\sigma(M')$ is the result of applying simultaneously all moves $m \in M'_{\max}$ to $\sigma(M'')$, and the order in which the maximum moves are applied is irrelevant. This implies in particular the desired coherence property. □

The key argument in the proof exploits the Recursion Postulate whereby for recursive runs of $\mathcal{R}$, the runs of different agents are initiated by calls and concern different state spaces with pairwise disjoint signatures, due to the function parameterization by agents, unless $pgm(a')$ is a child (or a descendant) of $pgm(a)$, in which case the relationship between the signatures is defined by the call relationship. Independent moves can be guaranteed in full generality only for algorithms with disjoint signatures.

4 Distributed ASMs with partial-order runs are recursive ASMs

While Theorem 3.1 is not surprising, we will now show its less obvious inverse.

Theorem 4.1. *For each finitely composed concurrent ASM $\mathcal{C}$ with program base $\{r_i \mid i \in I\}$ of nd-seq ASMs such that all its concurrent runs are definable by partial-order runs, one can construct a recursive ASM $\mathcal{R}_\mathcal{C}$ such that each concurrent run of $\mathcal{C}$ can be simulated by a recursive run of $\mathcal{R}_\mathcal{C}$.*[11]

Proof. Let a concurrent $\mathcal{C}$-run $(S_0, A_0), (S_1, A_1), \ldots$ be given. If it is definable by a partial-order run $(M, \leq, ag, pgm, \sigma)$, the transition from $S_i = \sigma(M_i)$ to S_{i+1} is performed in one concurrent step by parallel independent moves $m \in M_{i+1} \setminus M_i$, where M_i is the set of moves which contributed to transform S_0 into S_i. Let $m \in M_{i+1} \setminus M_i$ be a move performed by an agent $a = ag(m)$ with rule $pgm(a) =$ **amb** a **in** r, an instance of a rule r in the program base of $\mathcal{C}$. To execute the

[11] One obtains even the behavioral equivalence via an inverse simulation of every recursive $\mathcal{R}_\mathcal{C}$-run by a concurrent $\mathcal{C}$-run if the delegates of $\mathcal{C}$-agents, called in the recursive run to perform the step of their *caller* in the concurrent run, act in an 'eager' way. See the remark at the end of the proof.

concurrent step by means of steps of a recursive ASM $\mathcal{R}_\mathcal{C}$, we simulate each of its moves m by letting agent a act in the $\mathcal{R}_\mathcal{C}$-run as *caller* of a named rule $out_r \leftarrow \textsc{OneStep}_r(in_r)$. The callee agent c acts as delegate for one step of a: it executes **amb** $a \in r$ and makes its program immediately *Terminated*.

To achieve this, we refine the CALL machine defined in Definition 3.4 such that upon calling $out_r \leftarrow \textsc{OneStep}_r(in_r)$, the delegate c created by the call becomes *Active* so that it can make a step to execute **amb** c **in** $\textsc{OneStep}_r$. It suffices to add to the component INITIALIZE the update *Terminated*(**amb** c **in** q) := *false*, which makes c *Active*. $\textsc{OneStep}_r$ is defined to perform **amb** *caller*(c) **in** r and to terminate immediately (by setting *Terminated* to true). For ease of exposition we add to Definition 3.4 also the update *caller*(c) :=**self**, to distinguish agents in the concurrent run—the *caller*s of $\textsc{OneStep}_r$-machines—from the delegates each of which simulates one step of its *caller* and immediately terminates its life cycle.

It remains to determine the input and output for calling $\textsc{OneStep}_r$. For the input we exploit the existence of a bounded exploration witness W_r for r. All updates produced in a single step are determined by the values of W_r in the state, in which the call is launched. So W_r defines the input terms of the called rule $\textsc{OneStep}_r$, combined in in_r. Analogously, a single step of r provides updates to finitely many locations that are determined by terms appearing in the rule, which defines out_r.

We summarize the explanations by the following definition:

$\mathcal{R}_\mathcal{C} = \{out_r \leftarrow \textsc{OneStep}_r(in_r) \mid r \in$ program base of $\mathcal{C}\}$
$\textsc{OneStep}_r =$
  amb *caller*(**self**) **in** r // the delegate executes the step of its *caller*
  Terminated(*pgm*(**self**)) := *true* // ... and immediates stops

Note that by the refined Definition 3.4, $out_r \leftarrow \textsc{OneStep}_r(in_r)$ triggers the execution of the delegate program **amb** c **in** $\textsc{OneStep}_r$. Let $a = caller(c)$. By definition, **amb** c **in** $\textsc{OneStep}_r$ triggers **amb** c **in amb** a **in** r. Furthermore, since the innermost ambient binding counts, this machine is equivalent to the simulated machine **amb** a **in** r, as was to be shown.

Thus the recursive $\mathcal{R}_\mathcal{C}$-run which simulates $(S_0, A_0), (S_1, A_1), \ldots$ starts by Definition 3.1 in S_0 with program **amb** a_0 **in** $in_{main} \leftarrow \textsc{OneStep}_{main}(out_{main})$. For the sake of notational simplicity we disregard the auxiliary locations of $\mathcal{R}_\mathcal{C}$. Let

$A_i = \{a_{i_1}, \ldots, a_{i_k}\} \subseteq Agent$ for some i_j and k depending on i
where forall $1 \leq j \leq k$
  $a_{i_j} = ag(m_{i_j}) \in M_{i+1} \setminus M_i$ **and** $pgm(a_{i_j}) =$ **amb** a_{i_j} **in** r_{i_j}

We use the same agents a_{i_j} for A_i in the $\mathcal{R}_\mathcal{C}$-run, but with program $out_{r_{i_j}} \leftarrow \textsc{OneStep}_{r_{i_j}}(in_{r_{i_j}})$. Their step in the recursive run leads to a state S_i' where all callers a_{i_j} are *Waiting* and the newly created delegates c_{i_j} are *Active* and not *Waiting*. So we can choose them for the set A_i' of agents which perform the next $\mathcal{R}_\mathcal{C}$ step, whereby

- all rules r_{i_j} are performed simultaneously (as in the given concurrent run step), in the ambient of $caller(c_{i_j}) = a_{i_j}$ thus leading as desired to the state S_{i+1},
- the delegates make their program *Terminated*, whereby their callers a_{i_j} become again not *Waiting* and thereby ready to take part in the next step of the concurrent run. We assume for this that whenever in the $\mathcal{C}$-run (not in the $\mathcal{R}_\mathcal{C}$ run) a new agent a is created, it is made not *Waiting* (by initializing $CalledBy(a) := \emptyset$).

□

Remark. Consider an $\mathcal{R}_\mathcal{C}$-run where each recursive step of the concurrent caller agents in A_i, which call each some OneStep program, alternates with a recursive step of all—the just called—delegates whose program is not yet *Terminated*. Then this run is equivalent to a corresponding concurrent $\mathcal{C}$-run.

Note that Theorem 4.1 heavily depends on the prerequisite that $\mathcal{C}$ only has partial-order runs.[12] With general concurrent runs as defined in [4] the construction would not be possible.

4.1 Partial Order Runs of Petri Nets

The semantics of Petri nets actually defines a rather special case of partial-order runs, namely runs one can describe even by a nd-seq ASM, as we show in this section.

A Petri net comes with a finite number of transition rules, each of which can be described by a nd-seq ASM (see [6, p.297]). The special character of the computational Petri net model is due to the fact that during the runs, only exactly these rules are used. In other words there is a fixed association of each rule with an executing agent; there is no rule instantiation with new agents which could be created during a run. Therefore the states are the global markings of the net. The functions $\sigma(I)$ associated with the po-runs of the net yield for every finite initial segment I as value the global marking obtained by firing the rules in I.

For this particular kind of concurrent ASMs with partial-order runs one can define the concurrent runs by nd-seq ASMs, as we are going to show in this section.

Theorem 4.2. *For each finite concurrent ASM $\mathcal{C} = \{(a_i, r_i) \mid 1 \leq i \leq n\}$ with nd-seq ASMs r_i such that all its concurrent runs are definable by partial-order runs one can construct a nd-seq ASM $\mathcal{M}_\mathcal{C}$ such that the concurrent runs of $\mathcal{C}$ and the runs of $\mathcal{M}_\mathcal{C}$ are equivalent.*

[12] The other prerequisites in Theorem 4.1 appear to be rather natural. Unbounded runs can only result, if in a single step arbitrarily many new agents are created. Also, infinitely many different rules associated with the agents are only possible, if new agents are created and added during a concurrent run. Though this is captured in the general theory of concurrency in [4], it was not intended in Gurevich's definition of partial-order runs.

Corollary 4.1. *Partial-order Petri net runs can be simulated by runs of a non-deterministic sequential ASM.*[13]

Proof. We relate the states S_i of a given concurrent run of $\mathcal{C}$ to the states $\sigma(M_i)$ associated with initial segments M_i of a given corresponding partial order run $(M, \leq, ag, pgm, \sigma)$, where each step leading from S_i to S_{i+1} consists of pairwise incomparable moves in $M_{i+1} \setminus M_i$. We call such a sequence $S_0, S_1, \ldots$ of states a *linearised run* of $\mathcal{C}$. For $i > 0$ the initial segments M_i are non empty.

The linearized runs of $\mathcal{C}$ can be characterized as runs of a nd-seq ASM $\mathcal{M}_\mathcal{C}$: in each step this machine chooses one of finitely many non-empty subsets of rules in $\mathcal{C}$ to execute them in parallel. Formally:

$\mathcal{M}_\mathcal{C} =$ **choose** AllRulesOf$(I_1) \mid \cdots \mid$ AllRulesOf(I_n)
where
AllRulesOf$(\{i_1, \ldots, i_k\}) =$
r_{i_1}
$\ldots$
r_{i_k}
$\{I_1, \ldots, I_n\} = \{I' \neq \emptyset \mid I' \subseteq I\}$ // the non-empty subsets of I
$n = 2^{|I|} - 1$

To complete the proof it suffices to show the following lemma. □

Lemma 4.1. *The linearised runs of $\mathcal{C}$ are exactly the runs of $\mathcal{M}_\mathcal{C}$.*

Proof. To show that each run $S_0, S_1, \ldots$ of $\mathcal{M}_\mathcal{C}$ is a linearised run of $\mathcal{C}$ we proceed by induction to construct the partial-order run $(M, \leq)$ with its finite initial segments M_i. For the initial state $S_0 = \sigma(\emptyset)$ there is nothing to show, so let S_{i+1} result from S_i by applying an update set produced by AllRulesOf(J) for some non-empty $J \subseteq I$. By induction we have $S_i = \sigma(M_i)$ for some initial segment of a partial-order run $(M, \leq)$. As AllRulesOf(J) is a parallel composition, S_{i+1} results from applying the union of update sets $\Delta_{i_j} \in \Delta_{r_{i_j}}$ for $j = 1, \ldots, |J|$ to S_i. Each Δ_{i_j} defines a move m_{i_j} of some $\mathrm{ag}(m_{i_j}) = a_{i_j}$, move which finishes in state S_i. We now have two cases:

(i) The moves m_{i_j} with $j \in J$ are pairwise independent, i.e. their application in any order produces the same new state. Then $(M, \leq)$ can be extended with these moves such that $M_{i+1} = M_i \cup \{m_{i_j} \mid j \in J\}$ becomes an initial segment and $S_{i+1} = \sigma(M_i)$ holds.
(ii) If the moves m_{i_j} with $j \in J$ are not pairwise independent, the union of the corresponding update sets is inconsistent, hence the run terminates in state S_i.

[13] We thank Wolf Zimmermann for pointing out that the argument applies more generally to Mayr's Process Rewrite Systems [12]. They have been used in [11] to verify protocols for services which may rise exceptions.

To show the converse we proceed analogously. If we have $S_i = \sigma(M_i)$ for all $i \geq 1$, then S_{i+1} results from S_i by applying in parallel all moves in $M_{i+1} - M_i$. Applying a move m means to apply an update set produced by some rule $r_j \in \mathcal{C}$ (namely the rule $pgm(ag(m))$) in state S_i, and applying several update sets in parallel means to apply their union Δ, which then must be consistent. So we have $S_{i+1} = S_i + \Delta$ with $\Delta = \bigcup_{j \in J} \Delta_{i_j}$ for some J, where each Δ_{i_j} is an update set produced by r_{i_j}, i.e. Δ is an update set produced by AllRulesOf(J), which implies that the linearised run $S_0, S_1, \ldots$ is a run of $\mathcal{M}_\mathcal{C}$. □

For the corollary it suffices to note that each Petri net transition can be described by a nd-seq ASM (see [6, p.297]). The functions $\sigma(I)$ associated with the po-runs yield the global marking obtained by firing the rules in I.

5 Conclusions

While Gurevich's Sequential ASM Thesis [9] provides an elegant and satisfactory mathematical definition of the notion of sequential algorithm plus a proof that sequential algorithms are captured by sequential ASMs, this theory does not capture recursive algorithms. It lacks an appropriate call concept. In fact, in an attempt to solve this problem Blass and Gurevich in [1] invoked the notion of partial-order runs of 'distributed ASMs', which has been proposed in [8] as a concurrency concept for ASMs. We showed in this paper that these 'distributed ASMs' are finitely composed ASMs whose partial-order runs characterize (are equivalent to) recursive runs. Thus, partial-order runs of distributed ASMs do not capture the concept of concurrent algorithms (but see [4]).

References

1. A. Blass and Y. Gurevich. Algorithms vs. machines. *Bulletin of the EATCS*, 77:96–119, 2002.
2. E. Börger and T. Bolognesi. Remarks on turbo ASMs for functional equations and recursion schemes. In E. Börger et al., editors, *Abstract State Machines, Advances in Theory and Practice, 10th International Workshop (ASM 2003)*, volume 2589 of *Lecture Notes in Computer Science*, pages 218–228. Springer, 2003.
3. E. Börger and A. Raschke. *Modeling Companion for Software Practitioners*. Springer, 2018.
4. E. Börger and K.-D. Schewe. Concurrent Abstract State Machines. *Acta Informatica*, 53(5):469–492, 2016.
5. E. Börger and K.-D. Schewe. A behavioural theory of recursive algorithms, 2020. In preparation. A preliminary version is available at http://arxiv.org/abs/2001.01862.
6. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
7. Y. Gurevich. A new thesis. *Abstracts, American Mathematical Society*, 6(4):317, August 1985.
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 Trans. Comp. Logic*, 1(1):77–111, 2000.
10. Y. Gurevich and M. Spielmann. Recursive Abstract State Machines. *J. UCS*, 3(4):233–246, 1997.
11. C. Heike, W. Zimmermann, and A. Both. On expanding protocol conformance checking to exception handling. *Service Oriented Computing and Applications*, 8(4):299–322, 2014.
12. R. Mayr. Process Rewrite Systems. *Information and Computation*, 156:264–286, 1999.
13. Y. N. Moschovakis. What is a algorithm? In B. Engquist and W. Schmid, editors, *Mathematics Unlimited – 2001 and Beyond*, pages 919–936. Springer, 2001.