A Behavioural Theory of Recursive Algorithms

Egon Börger
Università di Pisa, Dipartimento di Informatica
Pisa, Italy
`boerger@di.unipi.it`

Klaus-Dieter Schewe
Zhejiang University, UIUC Institute
Haining, China
`kd.schewe@intl.zju.edu.cn`

Abstract. "What is an algorithm?" is a fundamental question of computer science. Gurevich's behavioural theory of sequential algorithms (aka the sequential ASM thesis) gives a partial answer by defining (non-deterministic) sequential algorithms axiomatically, without referring to a particular machine model or programming language, and showing that they are *captured* by (non-deterministic) sequential Abstract State Machines (nd-seq ASMs). However, recursive algorithms such as *merge-sort* are not covered by this theory, as has been pointed out by Moschovakis, who had independently developed a different framework to mathematically characterize the concept of (in particular recursive) algorithm. In this article we propose an axiomatic definition of the notion of *sequential recursive algorithm* which extends Gurevich's axioms for sequential algorithms by a Recursion Postulate and allows us to prove that sequential recursive algorithms are captured by *recursive Abstract State Machines*, an extension of nd-seq ASMs by a CALL rule. Applying this recursive ASM thesis yields a characterization of sequential recursive algorithms as finitely composed concurrent algorithms all of whose concurrent runs are partial-order runs.

Keywords: recursive algorithm, behavioural theory, abstract state machine, concurrent algorithm, partial-order run

1. Introduction

The notion of an algorithm is fundamental for computing, so it may seem surprising that there is still no commonly accepted definition. This is different for the notion of computable function that is captured by several equivalent formalisms such as Turing machines, random access machines, partial-recursive functions, λ-definable functions and many more [9]. However, as there is typically a huge gap between the abstraction level of an algorithm and the one of Turing machines, Gurevich concluded that the latter ones cannot serve as a definition for the notion of an algorithm [21]. He proposed to extend Turing's thesis to a new thesis, based on the observation that "if an abstraction level is fixed (disregarding low-level details and a possible higher-level picture) and the states of an algorithm reflect all the relevant

information, then a particular small instruction set suffices to model any algorithm, never mind how abstract, by a generalised machine very closely and faithfully".

Still it took many years from the formulation of this new thesis to the publication of the behavioural theory of sequential algorithms in [23]. In this seminal work—also known as the "sequential ASM thesis"—a *sequential algorithm* (seq-algorithm) is defined by three postulates[1]:

Sequential Time. A sequential algorithm proceeds in sequential time using states, initial states and transitions from states to successor states.

Abstract State. States are universal algebras (aka Tarski structures), i.e. functions resulting from the interpretation of a signature, i.e. a set of function symbols, over a base set.

Bounded Exploration. There exists a finite set of ground terms such that the difference between any state and its successor state is uniquely determined by the values of these terms in the state[2].

The behavioural theory further comprises the definition of *sequential Abstract State Machines* (seq-ASMs) and the proof that seq-ASMs *capture* seq-algorithms, i.e. they satisfy the postulates, and every seq-algorithm can be step-by-step simulated by a seq-ASM. As pointed out in [23, Sect.9.2] (and elaborated to a full proof in [15, Sect.7.2.2 -7.2.3]) it is easy to extend the theory to cover also bounded non-determinism, using non-deterministic sequential ASMs (nd-seq ASMs)[3].

It should be noted that the *definition* of a sequential algorithm given by Gurevich does not require a particular formalism for the *specification*. Seq-ASMs capture seq-algorithms, so they are a suitable candidate for specification[4], but they are not the only possible choice. For instance, in the light of the proofs in [23] it is not an overly difficult exercise to show that deterministic Event-B [2] or B [1] also capture seq-algorithms.

We believe that in order to obtain a commonly acceptable definition of the notion of algorithm, this distinction between an *axiomatic definition* (as by Gurevich's postulates for seq-algorithms), which does not refer to a particular language or programming style, and the *capture* by an abstract machine model (such as seq-ASMs, deterministic Event-B or others) is fundamental.

In an independent line of research, started in [29] and carried out in a series of papers (see the references in [30]) and culminating in a recent book [32], Moschovakis proposed a different mathematical definition of 'recursion' to model the mathematical properties of the intuitive notion of 'algorithm'. In [31] he questioned that algorithms can be adequately defined by abstract machines and asked how recursive algorithms like the well-known *mergesort* are covered by seq-ASMs. Although the perception that Gurevich used seq-ASMs as a definition for the general notion of algorithm—not only for its sequential

---

[1]A mathematically precise formulation of these postulates requires more care, see below, but the rough summary here will be sufficient for the Introduction.

[2]This set of terms is usually called a *bounded exploration witness*, while the difference between a state and its successor is formally given by an *update set*. Informally, bounded exploration requires that there are only finitely many terms, the interpretation of which determine how a state will be updated by the algorithm to produce the successor state.

[3]It suffices to slightly modify the sequential time and the abstract state postulates, using in particular a successor relation instead of a function and permitting the choice between finitely many rules. Gurevich uses the term 'bounded-choice nondeterministic algorithm' instead of 'nd-seq algorithm'.

[4]In particular, as pointed out in [15], rules in an ASM look very much like pseudo-code, so the appearance of ASM specifications is often close to the style, in which algorithms have been described in the past and in textbooks. The difference is of course that the semantics of ASMs is precisely defined.

instance—is a misinterpretation, unfortunately the response by Blass and Gurevich [6] to Moschovakis's criticism does not clarify the issue in a convincing way. Instead of admitting that an extended behavioural theory for recursive algorithms still needs to be developed, distributed ASMs with a semantics defined through partial-order runs [22] are claimed to be sufficient to capture recursive algorithms.[5] As Börger and Bolognesi point out in their contribution to the debate [10], a much simpler extension of seq-ASMs suffices for the specification of algorithms of the *mergesort* kind, which define recursive functions (as do Moschovakis' systems of recursive equations called 'recursive programs' [32], see the discussion in Sect.6). Furthermore, the response by Blass and Gurevich blurs the subtle distinction between the axiomatic definition and the possibility to express any algorithm on an arbitrary level of abstraction by an abstract machine. This led also to Vardi's almost cynical comment that the debate is merely about the preferred specification style (functional or imperative), which is as old as the field of programming [37].[6]

While the difficult epistemological issue concerning the definition of the general notion of algorithm has been convincingly addressed for *sequential* algorithms by Gurevich's behavioural theory, no such theory for recursive algorithms or distributed algorithms was available at the time of the debate between Moschovakis, Blass and Gurevich, Börger and Bolognesi, and Vardi. In the meantime a behavioural theory for concurrent algorithms has been developed [12]. It comprises an axiomatic definition of the notion of concurrent algorithm as a family of nd-seq algorithms indexed by agents that is subject to an additional concurrency postulate for their runs, by means of which Lamport's sequential consistency requirement is covered and generalised [25]. In a nutshell, the concurrency postulate requires that a successor state of the global state of the concurrent algorithm results from simultaneously applying update sets of finitely many agents that have been built on some previous (not necessarily the latest) states.

Using this theory of concurrency it is possible to reformulate the answer given by Blass and Gurevich to Moschovakis's question: every recursive algorithm is a concurrent algorithm with partial-order runs. Since concurrent ASMs capture concurrent algorithms (as shown in [12]), they provide a natural candidate for the specification of all concurrent algorithms, thus in particular of recursive algorithms. However, the "overkill" argument will remain, as the class of concurrent algorithms is much larger than the class of recursive algorithms.

For example, take the *mergesort* algorithm (see Sect.4.1). Every call to (a copy of) itself and every call to (a copy of) the auxiliary *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) two or more agents are active in any state. In contrast, in a concurrent algorithm all agents may be active, and they can interact in many different ways on shared locations as well as on different clocks. As a consequence, concurrent runs may become highly non-deterministic and not linearisable, whereas a sequential[7] recursive algorithm permits at most bounded non-determinism and without loss of generality several simultaneous calls can always be sequentialised.

This motivates the research we report in this article. Our objective is to develop a behavioural theory of sequential recursive algorithms. For this we propose an axiomatic definition of sequential recursive

---

[5]The definition of recursive ASMs in [24] uses a special case of this translation of recursive into distributed computations.

[6]This debate, however, is still much younger than the use of the notion of algorithm.

[7]We use here the attribute 'sequential' to emphasize that we view recursive algorithms as sequential algorithms which call sequential algorithms, so that no unbounded parallelism is allowed. The reason is that unbounded parallelism permits to define recursion, as we explain in Sect.4.

algorithms which enriches sequential algorithms by call steps, such that the parent-child relationship between caller and callee defines well-defined shared locations representing input and return parameters. We will present and motivate our axiomatisation in Section 2. In Section 3 we define recursive ASMs[8] by an appropriate extension of nd-seq ASMs with a call rule and show our main result, aka Recursive ASM Thesis:

Main Theorem. *Sequential recursive algorithms are captured by recursive ASMs.*

Section 4 is dedicated to an illustration of our theory by examples. We concentrate on *mergesort*, *quicksort* and the *sieve of Eratosthenes* for which we present recursive ASMs. We use the examples to show that the parallelism of ASMs, which is unbounded, is stronger than sequential recursion, so that there is no need to investigate parallel recursive algorithms separately from parallel algorithms.

In Section 5 we report an application of the recursive ASM thesis.[9] We return to the observation by Blass and Gurevich—though not explicitly stated in [6]—that sequential recursive algorithms are linked to concurrent algorithms with partial-order runs. We first show that indeed the runs of a sequential recursive algorithm (read: of a recursive ASM) are definable by partial-order runs (Theorem 5.2), which comes at no surprise. The amazing second discovery was that also a converse relation holds, namely if all runs of a finitely composed concurrent algorithm (read: of a concurrent ASM $\mathcal{C}$ which consists only of instances of a bounded number of nd-seq ASMs) are definable by partial-order runs, then this algorithm is equivalent to a recursive ASM (Theorem 5.3). This relativizes the overkill argument.[10]

Theorem 5.3 can be strengthened if the given concurrent ASM is static, i.e. with a fixed set of agents with fixed associated programs. Such concurrent ASMs are equivalent to nd-seq ASMs (Theorem 5.4). An interesting corollary of this theorem concerns the Process Rewrite Systems investigated in [27]. They form the most general and most expressive set in a hierarchy of classes of rewrite systems which can be used to model certain state-based concurrent systems and are classified in [27] by their expressiveness. Furthermore, the Process Rewrite Systems are shown in [27] to strictly extend Petri nets (by subroutines that can return a value to their caller), but still to have a decidable reachability problem. As a corollary of Theorem 5.4 it turns out that for each Process Rewrite System, its partial-order runs can be simulated by runs of a nd-seq ASM (Corollary 5.1).

Finally, in Section 6 we embed our work into a larger picture of related work on behavioural theories, and in Section 7 we present a brief summary and outlook on further research.

2.   Axiomatisation of Recursive Algorithms

A decisive feature of a recursive algorithm is that it calls itself, or more precisely a copy (we also say an instance) of itself. If we consider mutual recursion, then this becomes slightly more general, as there is a finite family of algorithms calling (copies of) each other. Therefore, providing copies of algorithms and enabling calls will be essential for the intended definition of the notion of recursive algorithm, whereas otherwise we rely on Gurevich's axiomatic definition of sequential algorithms. Furthermore,

---

[8]Since by definition *recursive* ASMs are extensions of nd-seq ASMs, to obtain a short name we skip the two attributes 'non-deterministic' and 'sequential'.

[9]A preliminary version of the result presented in Section 5 appeared in [14].

[10]In fact, it shows that, roughly speaking, finitely composed concurrent algorithms with partial-order runs are indeed the sequential recursive algorithms, and the response given in [6] may be seen as the result of ingenious serendipity. However, arbitrary concurrent algorithms as discussed in [12] are a much wider class of algorithms.

there may be several simultaneous calls, which give rise to non-determinism,[11] as these simultaneously called copies may run sequentially in one order or the other, or in parallel or even asynchronously. However, there is no interaction between simultaneously called algorithms, which implies that the mentioned execution latitude already covers all choices.

2.1. Non-deterministic Sequential Algorithms

In this section we recall the axiomatic definition of non-deterministic sequential algorithms.

Definition 2.1. A *non-deterministic sequential algorithm* (for short: *nd-seq algorithm*) is defined by the branching time, abstract state and bounded exploration postulates 1, 2 and 3 in [23], paper to which we refer for the motivation for these axioms.

Postulate 1. (Branching Time Postulate)
An nd-seq algorithm $\mathcal{A}$ comprises a set $\mathcal{S}$, elements of which are called *states*, a subset $\mathcal{I} \subseteq \mathcal{S}$, elements of which are called *initial states*, and a *one-step transition relation* $\tau \subseteq \mathcal{S} \times \mathcal{S}$.[12] Whenever $\tau(S, S')$ holds, the state S' is called a *successor state* of the state S and we say that the algorithm performs a step in S to yield S'.

Though Postulate 1 only gives a necessary condition for nd-seq algorithms and in particular leaves open what states are, one can already derive some consequences from it such as the notions of *run*, *final state* and *behavioural equivalence*.

Definition 2.2. Let $\mathcal{A}$ be a nd-seq algorithm with states $\mathcal{S}$, intial states $\mathcal{I}$ and transition relation τ. A *run* of $\mathcal{A}$ is a sequence $S_0, S_1, S_2, \ldots$ with $S_i \in \mathcal{S}$ for all i and $S_0 \in \mathcal{I}$ such that $\tau(S_i, S_{i+1})$ holds for all i.

Often S_i is called a *final state* of a run $S_0, S_1, S_2, \ldots$ of $\mathcal{A}$ (and the run is called *terminated* in this state) if $S_j = S_i$ holds for all $j \geq i$. But sometimes it is more convenient to use a dynamic termination predicate whose negation guards the execution of the algorithm $\mathcal{A}$ and which is set to true by $\mathcal{A}$ when $\mathcal{A}$ reaches a state one wants to consider as final.

States are postulated to be *universal algebras* (aka *Tarski structures*), which capture all desirable structures that appear in mathematics and computing.

Definition 2.3. A *signature* Σ is a finite set of function symbols, and each $f \in \Sigma$ is associated with an *arity* $\mathrm{ar}(f) \in \mathbb{N}$. A *structure* over Σ comprises a *base set*[13] B and an *interpretation* of the function symbols $f \in \Sigma$ by functions $f_B : B^{\mathrm{ar}(f)} \to B$. An *isomorphism* σ between two structures is given by a bijective mapping $\sigma : B \to B'$ between the base sets that is extended to the functions by $\sigma(f_B)(\sigma(a_1), \ldots, \sigma(a_n)) = \sigma(f_B(a_1, \ldots, a_n))$ for all $a_i \in B$ and $n = \mathrm{ar}(f)$.

Postulate 2. (Abstract State Postulate)
Each nd-seq algorithm $\mathcal{A}$ comprises a *signature* Σ such that

---

[11]The presence of this non-determinism in recursive algorithms has also been observed in Moschovakis' criticism [31], e.g. *mergesort* calls two copies of itself, each sorting one half of the list of given elements.

[12]For deterministic algorithms τ is a function.

[13]For convenience to capture partial functions it is tacitly assumed that base sets contain a constant *undef* and that each isomorphism σ maps *undef* to itself.

(i) Each state $S \in \mathcal{S}$ of $\mathcal{A}$ is a structure over Σ.

(ii) The sets $\mathcal{S}$ and $\mathcal{I}$ of states and initial states, respectively, are both closed under isomorphisms.

(iii) Whenever $\tau(S, S')$ holds, then the states S and S' have the same base set.

(iv) Whenever $\tau(S, S')$ holds and σ is an isomorphism defined on S, then also $\tau(\sigma(S), \sigma(S'))$ holds.

In the following we write f_S to denote the interpretation of the function symbol $f \in \Sigma$ in the state S. Though we still have only necessary conditions for nd-seq algorithms, one can define further notions that are important for the development of the theory.

Definition 2.4. A *location* of the nd-seq algorithm $\mathcal{A}$ is a pair $\ell = (f, (a_1, \ldots, a_n))$ with a function symbol $f \in \Sigma$ of arity n and all $a_i \in B$. If B is the base set of state S and $f_S(a_1, \ldots, a_n) = a_0$ holds, then a_0 is called the *value* of the location ℓ in state S.

We write $\mathrm{val}_S(\ell)$ for the value of the location ℓ in state S. The *evaluation function* val can be extended to ground terms in a straightforward way.

Definition 2.5. The *set of ground terms* over the signature Σ is the smallest set $\mathbb{T}$ such that $f(t_1, \ldots, t_n) \in \mathbb{T}$ holds for all $f \in \Sigma$ with $\mathrm{ar}(f) = n$ and $t_1, \ldots, t_n \in \mathbb{T}$[14]. The *value* $\mathrm{val}_S(t)$ of a term $t = f(t_1, \ldots, t_n) \in \mathbb{T}$ in a state S is defined by $\mathrm{val}_S(t) = f_S(\mathrm{val}_S(t_1), \ldots, \mathrm{val}_S(t_n))$.

With the notions of location and value one can further define updates and their result on states[15].

Definition 2.6. An *update* of an nd-seq algorithm $\mathcal{A}$ in state S is a pair (ℓ, v) with a location ℓ and a value $v \in B$, where B is the base set of S. An update (ℓ, v) is *trivial* iff $\mathrm{val}_S(\ell) = v$ holds. An *update set* is a set of updates. An update set Δ in state S is *consistent* iff $(\ell, v_1), (\ell, v_2) \in \Delta$ implies $v_1 = v_2$, i.e. there can be at most one non-trivial update of a location ℓ in a consistent update set. If Δ is a consistent[16] update set in state S, then $S + \Delta$ denotes the unique state S' with $\mathrm{val}_{S'}(\ell) = \begin{cases} v & \text{if } (\ell, v) \in \Delta \\ \mathrm{val}_S(\ell) & \text{otherwise} \end{cases}$.

Considering the locations, where a state S and a successor state S' differ, gives us the following well-known fact (see [23]).

Fact 2.1. If $\tau(S, S')$ holds, then there exists a unique minimal consistent update set Δ with $S + \Delta = S'$.[17]

We use the notation $\Delta(S, S')$ for the consistent update set that is defined by $\tau(S, S')$. We further write $\mathbf{\Delta}(S)$ for the set of all such update sets defined in state S, i.e. $\mathbf{\Delta}(S) = \{\Delta(S, S') \mid \tau(S, S')\}$.

The third postulate concerns bounded exploration. It is motivated by the simple observation that any algorithm requires a finite representation, which implies that only finitely many ground terms may appear in the representation, and these must then already determine the successor state—for a more detailed discussion see [23]—or the successor states in the case of non-determinism. Formally, this requires a notion of *coincidence* for a set of ground terms in different states.

---

[14]Clearly, for the special case $n = 0$ we get $f() \in \mathbb{T}$. Instead of $f()$ we usually write simply f.

[15]Note that update sets as we use them are merely differences of states.

[16]Otherwise, usually the term $S + \Delta$ used to define the successor state is considered as not defined. An alternative is to extend this definition letting $S + \Delta = S$, if Δ is inconsistent.

[17]The conclusion is true for any given pair (S, S') of states, independently of the relation $\tau(S, S')$.

Definition 2.7. Let $T \subseteq \mathbb{T}$ be a set of ground terms for a nd-seq algorithm $\mathcal{A}$. Two states S_1 and S_2 with the same base set B *coincide* on T iff $\text{val}_{S_1}(t) = \text{val}_{S_2}(t)$ holds for all terms $t \in T$.

Postulate 3. (Bounded Exploration Postulate)
Each nd-seq algorithm $\mathcal{A}$ comprises a finite set of ground terms $W \subseteq \mathbb{T}$ such that whenever two states S_1 and S_2 coincide on W the corresponding sets of update sets for S_1 and S_2 are equal, i.e. we have $\Delta(S_1) = \Delta(S_2)$. The set W is called a *bounded exploration witness*.

Bounded exploration witnesses are not unique. In particular, the defining property remains valid, if W is extended by finitely many terms. Therefore, without loss of generality we may tacitly assume that a bounded exploration witness W is always closed under subterms. We then call the elements of W *critical terms*. If t is a critical term, then its value $\text{val}_S(t)$ in a state S is called a *critical value*. This gives rise to the following well-known fact.

Fact 2.2. The set $\Delta(S)$ of update sets of an nd-seq algorithm $\mathcal{A}$ in a state S is finite, and every update set $\Delta(S, S') \in \Delta(S)$ is also finite.

For a proof we first need to show that in every update $((f, (v_1, \ldots, v_n)), v_0)$ in an update set $\Delta(S, S')$ the values v_i are critical [23]. As W is finite, there are only finitely many critical values, and we can only build finite update sets $\Delta(S, S')$ and only finitely many sets of update sets with these. We will use such arguments later in Section 3 to show that recursive algorithms are captured by recursive ASMs, and dispense with giving more details here.

2.2. Recursion Postulate

As remarked initially, an essential property of any recursive algorithm is the ability to perform call steps, i.e. to trigger an instance of a given algorithm (maybe of itself) and remain waiting until the callee has computed an output for the given input. We make this explicit by extending the postulate on the one-step transition relation τ of nd-seq algorithms by characteristic conditions for a call step (see Postulate 4 below).

Furthermore, it seems to be characteristic for runs of recursive algorithms that in a given state, the caller may issue in one step more than one call, though only finitely many, of callees which perform their subcomputations independently of each other. For an example see the *sort* rule in the *mergesort* algorithm in Section 4. The resulting 'asynchronous parallelism' implies that the states in runs of a recursive algorithm are built over the union of the signatures of the calling and the called algorithms.

The independency condition for 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. Below we make the term *instance of an algorithm* more precise to capture the needed encapsulation of subcomputations. This must be coupled with an appropriate input/output relation between the input provided by the caller and the output computed by the callee for this input, which will be captured by a *call relationship* in Definition 2.9.

This explains the following definition of an i/o-algorithm as nd-seq algorithm with call steps and distinguished function symbols for input and output.

Definition 2.8. An *algorithm with input and output* (for short: *i/o-algorithm*) is an nd-seq algorithm whose one-step transition relation τ may comprise call steps satisfying the Call Step Postulate 4 formulated below and whose signature Σ is the disjoint union of three subsets

$$\Sigma = \Sigma_{in} \cup \Sigma_{loc} \cup \Sigma_{out}$$

containing respectively input, local and output function symbols that satisfy the input/output assumption defined below.

Function symbols in Σ_{in}, Σ_{out} and Σ_{loc}, respectively, are called *input*, *output* and *local* function symbols. Correspondingly, locations with function symbol in Σ_{in}, Σ_{out} and Σ_{loc}, respectively, are called *input*, *output* and *local locations*. We include into input resp. output locations also variables which appear as input resp. output parameters of calls, although they are not function symbols.[18]

The following assumption on input/output locations of i/o-algorithms is not strictly needed, but it can always be arranged and it eases the development of the theory.

Input/Output Assumption for i/o algorithms $\mathcal{A}$:

(i) Input locations of $\mathcal{A}$ are only read by $\mathcal{A}$, but never updated by $\mathcal{A}$. Formally, this implies that if (ℓ, v) is an update in an update set $\Delta(S, S')$ of $\mathcal{A}$ in any state S, then the function symbol f in ℓ is not in Σ_{in} of $\mathcal{A}$.

(ii) Output locations of $\mathcal{A}$ are never read by $\mathcal{A}$, but can be written by $\mathcal{A}$. This can be formalised by requiring that if W is a bounded exploration witness, then for any term $f(t_1, \ldots, t_n) \in W$ we have $f \notin \Sigma_{out}$.

(iii) Any initial state of $\mathcal{A}$ only depends on its input locations, so we may assume that $val_{S_0}(\ell) = undef$ holds in every initial state S_0 of $\mathcal{A}$ for all output and local locations ℓ. This assumption guarantees that when an i/o-algorithm is called, its run is initialized by the given input, which reflects the common intuition using input and output.

In a call relationship we call the caller the *parent* and the callee the *child* algorithm. Intuitively,

a) the parent algorithm is able to update input locations of the child algorithm, which determines the child's initial state;

b) when the child algorithm is called, control is handed over to it until it reaches a final state, in which state the parent takes back control and is able to read the output locations of the child;

c) the two algorithms have no other common locations.

Therefore we define:

Definition 2.9. 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:

- $\Sigma_{in}^{\mathcal{A}^c} \subseteq \Sigma^{\mathcal{A}^p}$. Furthermore, $\mathcal{A}^p$ may update input locations of $\mathcal{A}^c$, but never reads these locations. Formally this implies that for a bounded exploration witness W of $\mathcal{A}^p$ and any term $f(t_1, \ldots, t_n) \in W$ we have $f \notin \Sigma_{in}^{\mathcal{A}^c}$.

---

[18]This is only a terminological convention, it does not change Definition 2.4. Formally one could treat these variables as 0-ary functions, one per call.

- $\Sigma_{out}^{\mathcal{A}^c} \subseteq \Sigma^{\mathcal{A}^p}$. Furthermore, $\mathcal{A}^p$ may read but never updates output locations of $\mathcal{A}^c$, so we have that for any update in an update set $\Delta(S, S')$ in any state S of $\mathcal{A}^p$, its function symbol is not in $\Sigma_{out}^{\mathcal{A}^c}$.
- $\Sigma_{loc}^{\mathcal{A}^c} \cap \Sigma^{\mathcal{A}^p} = \emptyset$ (no other common locations).

Postulate 4. (Call Step Postulate)
When an i/o-algorithm p—the caller, viewed as parent algorithm—calls a finite number of i/o-algorithms $c_1, \dots, c_n$—the callees, viewed as child algorithms $CalledBy(p)$—a call relationship (denoted as $CalledBy(p)$) 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 2.10. A *sequential recursive algorithm* $\mathcal{R}$ is a finite set of i/o-algorithms—i.e. satisfying the branching time, abstract state, bounded exploration and call step postulates 1, 2, 3 and 4—one of which is distinguished as *main* algorithm. The elements of $\mathcal{R}$ are also called components of $\mathcal{R}$.

Differently from runs of a nd-seq algorithm as defined by Definition 2.2, 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 5 below. In this postulate we refer to *Active* and not *Waiting* instances of components, which are defined as follows:

Definition 2.11. To be *Active* resp. *Waiting* in a state S is defined as follows:

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

Called collects the instances of algorithms that are called during the run. $CalledBy(p)$ denotes the subset of *Called* which contains all the children called by 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 5. (Recursive Run Postulate)
For a sequential recursive algorithm $\mathcal{R}$ with main component *main* a *recursive run* is a sequence $S_0, S_1, S_2, \dots$ of states together with a sequence $C_0, C_1, C_2, \dots$ of sets of instances of components of $\mathcal{R}$ which satisfy the following constraints concerning the recursive run and bounded call tree branching:

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 in S_i *Active* and not *Waiting* instances of components of $\mathcal{R}$,

- 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.[19] 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.

To capture the required independence of callee computations we now describe a way to make the concept of an instance of an algorithm and its computation more precise. The idea is to use for each callee a different state space, with the required connection between caller and callee through input and output terms. One can define an instance of an algorithm $\mathcal{A}$ by adding a label a, which we invite the reader to view as an agent executing the instance $\mathcal{A}_a = (a, \mathcal{A})$ of $\mathcal{A}$. The label a can be used as environment parameter for the evaluation $val_S(t, a)$ of a term t in state S with the given environment. This yields different functions $f_a, f_{a'}$ as interpretation of the same function symbol f for different agents a, a', 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.[20]

This allows us to make the meaning of 'activating a fresh instance of a callee' in the Call Step Postulate more precise by using as fresh instance of a child algorithm $\mathcal{A}$ called by p an instance $\mathcal{A}_c$ with a new label c, where the interpretation f_c of each input or output function f satisfies $f_c = f_p$ during the run of $\mathcal{A}_c$. Note that by the call relationship constraint in the Call Step Postulate, input/output function symbols are in the signature of both the parent and the child algorithm. This provides the ground for the 'asynchronous parallelism' of independent subcomputations in the run constraint of the recursive run postulate. In fact, when a state S' is obtained from state S by one step of each of finitely many *Active* and not *Waiting* i/o-algorithms $q_1, \ldots, q_k$, this means that for each $j \in \{1, \ldots, k\}$ the one-step transition relation holds for the corresponding state restrictions, namely $\tau_{q_j}(res(S, \Sigma^{q_j}), res(S', \Sigma^{q_j}))$ where $res(S, \Sigma)$ denotes the restriction of state S to the signature Σ.

With the above definitions one can make the Call Step Postulate more explicit by saying that if $\mathcal{A}_p$ calls $\mathcal{A}^1, \ldots, \mathcal{A}^n$ in a state S so that as a result $\tau_{\mathcal{A}_p}(S, S')$ holds[21], then for fresh instances $\mathcal{A}^i_{c_i}$ of $\mathcal{A}^i$ with input locations $input_i$ ($1 \leq i \leq n$) the following holds:

In S' the following is true **forall** $1 \leq i \leq n$:
$\quad \mathcal{A}^i_{c_i} \in Called$ **and** $\mathcal{A}^i_{c_i} \in CalledBy(\mathcal{A}_p)$ **and** $Initialized(\mathcal{A}^i_{c_i}, input_i)$
$\quad\quad$ **and** $Waiting(\mathcal{A}_p)$[22]

The predicate $Initialized(\mathcal{A}^i_{c_i}, input_i)$ expresses that the restriction $res(S', \Sigma^{\mathcal{A}^i_{c_i}})$ of S' to the signature of $\mathcal{A}^i_{c_i}$ is an initial state of $\mathcal{A}^i_{c_i}$ determined by $input_i$, so that $\mathcal{A}^i_{c_i}$ is ready to start its computation.

---

[19]Our reviewers worried here about synchronous, asynchronous and interleaving executions. Due to the independence of all the instances, whether they are executed in parallel, asynchronously or interleaved does not matter here. Note that C_i is a not furthermore restricted finite *subset* of all in S_i *Active* and not *Waiting* (completely independent) instances of components of $\mathcal{R}$. If C_i contains more than one instance, the instances in this C_i and only these are synchronized (independently for each i), but also interleaving (where each C_i is a singleton set) and asynchronous execution are possible. We impose no constraint at all on how (e.g. by an external scheduling mechanism) the sets C_i are determined.

[20]The idea underlies the definition of ambient ASMs we will use in the following. It allows one to classify certain f as ambient-dependent functions, whereby the algorithm instances become context-aware. For the technical details we refer to the definition in the textbook [11, Ch.4.1].

[21]To simplify the presentation we adopt a slight abuse of notation, writing $\tau_{\mathcal{A}}(S, S')$ with the global states S, S' even where $\tau_{\mathcal{A}}$ really holds for their restriction to the sub-signature of the concrete algorithm $\mathcal{A}$.

[22]Except the trivial case that all $\mathcal{A}^i_{c_i}$ when *Called* in S' are already *Terminated*.

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. An example of the former case is given by *mergesort*, whereas an example for the latter case is given by the recursive *sieve of Eratosthenes* algorithm discussed in [31] and used in Section 4 to illustrate our definitions.

3. Capture of Recursive Algorithms

We now proceed with the second step of our behavioural theory, the definition of an abstract machine model—these will be recursive ASMs, an extension of sequential ASMs—and the proof of the main theorem that the runs of this model capture the runs of sequential recursive algorithms (Theorems 3.1 and 3.2).

3.1. Recursive Abstract State Machines

As common with ASMs let Σ be a signature and let $\mathcal{U}$ be a universe of values. In addition, we assume a *background structure* comprising at least truth values and their connectives as well as the operations on them. Values defined by the background are assumed to be elements of $\mathcal{U}$. Then (ground) terms over Σ are built in the usual way (using also the operations from the background), and they are interpreted taking $\mathcal{U}$ as base set—for details we refer to the standard definitions of ASMs [15]. This defines the set of states of *recursive ASM rules* we are going to define now syntactically. We proceed by induction, adding to the usual rules of non-deterministic sequential (nd-seq) ASMs (which we repeat here for the sake of completeness) named rules which can be called.[23] We use an arbitrary set $\mathcal{N}$ of names for named rules.

Assignment. If $t_0, \ldots, t_n$ are terms over the signature Σ and $f \in \Sigma$ is a function symbol of arity n, then
$$f(t_1, \ldots, t_n) := t_0 \text{ is a recursive ASM rule.}$$

Branching. If φ is a Boolean term over the signature Σ and r is a recursive ASM rule, then also IF φ THEN r is a recursive ASM rule.

Bounded Parallelism. If $r_1, \ldots, r_n$ are recursive ASM rules, then also their parallel composition, denoted PAR $r_1 \| \ldots \| r_n$ is a recursive ASM rule.

Bounded Choice. If $r_1, \ldots, r_n$ are recursive ASM rules, then also the non-deterministic choice among them, denoted CHOOSE $r_1 \mid \cdots \mid r_n$ is a recursive ASM rule.

---

[23]The terse definition here avoids complicated syntax. We tacitly permit parentheses to be used in rules when needed.

Let. If r is a recursive ASM rule and t is a term and x is a variable, then `LET` $x = t$ `IN` r is also a recursive ASM rule.

Call. Let $t_0, \ldots, t_n$ be terms where the outermost function symbol of t_0 is different from the outermost function symbol of t_i for every $i \neq 0$. Let $N \in \mathcal{N}$ be the name of a rule of arity n, declared by $N(x_1, \ldots, x_n) = r$, where r is a recursive ASM rule all of whose free variables are contained in $\{x_0, \ldots, x_n\}$. Then $t_0 \leftarrow N(t_1, \ldots, t_n)$ is a recursive ASM rule.

Definition 3.1. A recursive ASM rule of form $t_0 \leftarrow N(t_1, \ldots, t_n)$ is called a *named i/o-rule* or simply i/o-rule.

The same way a sequential recursive algorithm consists of finitely many i/o-algorithms, a recursive ASM $\mathcal{R}$ consists of finitely many recursive ASM rules, also called component (or component ASM) of $\mathcal{R}$.

Definition 3.2. A *recursive Abstract State Machine* (rec-ASM) $\mathcal{R}$ consists of a finite set of recursive ASM rules, one of which is declared to be the main rule.

For the signature Σ of recursive ASM rules we use the notation $\Sigma_{in} \cup \Sigma_{loc} \cup \Sigma_{out}$ for the split of Σ into the disjoint union of input, output and local functions. For named i/o-rules $t_0 \leftarrow N(t_1, \ldots, t_n)$ the outermost function symbol of t_0 is declared as an element of Σ_{out} and for each t_i the outermost function symbol of t_i is declared as an element of Σ_{in} ($i = 1, \ldots, n$). In the definition of the semantics of a named i/o-rule we will take care that the input/output assumption and the call relationship defined in Section 2.2 for i/o-algorithms are satisfied by named i/o-rules.

Sequential and recursive ASMs differ in their run concept, analogously to the difference between runs of an nd-seq algorithm and of a sequential recursive algorithm. A sequential ASM is a 'mono-agent' machine: it consists of just one rule[24] and in a sequential run this very same rule is applied in each step—by an execution agent that normally remains unmentioned. This changes with recursive ASMs which are 'multi-agent' machines. They consist of a set of independent rules, multiple instances of which (even of a same rule) may be called to be executed independently (for an example see the *sort* rule in Sect. 4). We capture this by associating an execution agent $ag(r)$ with each rule r so that each agent can execute its rule instance independently of the other agents, in its own copy of the state space (i.e. instances of states over the signature of the executed rule), taking into account the call relationship between caller and callee (see below).

Therefore every single step of a recursive ASM $\mathcal{R}$ may involve the execution of one step by each of finitely many *Active* and not *Waiting* agents a which execute in their state space the rule $pgm(a)$ they are associated (we also say *equipped*) with. To describe this separation of state spaces of different agents (in particular if they execute the same program), we define *instances of a rule* r by ambient ASMs of form **amb** a **in** r with agents a (see below for details). The following definition paraphrases the run constraint in the Recursive Run Postulate 5.

Definition 3.3. A *recursive run* of a recursive ASM $\mathcal{R}$ is a sequence $S_0, S_1, S_2, \ldots$ of states together with a sequence $A_0, A_1, A_2, \ldots$ of subsets of *Agent*, where each $a \in$ *Agent* is equipped with a $pgm(a)$ that is an instance **amb** a **in** r of a rule $r \in \mathcal{R}$, such that the following holds:

---

[24]For notational convenience, this rule is often spelled out as a set of rules, however these rules are always executed together, in parallel.

- A_0 is a singleton set $A_0 = \{a_0\}$, which in S_0 equals the set *Agent*, and its agent a_0 is equipped with $pgm(a_0) = (\mathbf{amb}\ a_0\ \mathbf{in}\ main)$.
- A_i is a finite set of in S_i *Active* and not *Waiting* agents. We define (see Definition 2.11):

  Active(a) (in state S) **iff** $a \in$ *Agent* **and not** *Terminated*$(pgm(a))$ (in S)
  Waiting(a) (in state S) **iff forsome** $a' \in$ *CalledBy*(a) *Active*(a') (in S)
- S_{i+1} is obtained from S_i in one $\mathcal{R}$-step by performing for each agent $a \in A_i$ one step of $pgm(a)$.[25]

To complete the definition of recursive ASM runs, which extends the notion of runs of sequential ASMs, it suffices (besides explaining ambient ASMs) to add a definition for what it means semantically to apply a named i/o-rule. Using the ASM framework this boils down to extend the inductive definition of the update sets computed by sequential ASMs in a given state by defining the update sets computed by named i/o-rules.

A detailed definition of ambient ASMs can be found in [11, Ch.4.1]. Here it suffices to say that using $\mathbf{amb}\ a\ \mathbf{in}\ r$ as instance of a called rule r permits to isolate the state space of agent a from that of other agents, namely by evaluating terms t in state S considering also the agent parameter a, using $val_S(t, a)$ instead of $val_S(t)$. To establish the call relationship we require below the following: when a recursive ASM rule r, executed by a parent agent p, calls a rule q to be executed by a child agent c, then the input/output functions f of q are also functions in r and are interpreted there in the state space of p the same way as in the state space of c.

For the sake of completeness we repeat the definition of update sets for sequential ASM rules from [22] and extend it for named i/o-rules. Rules r of sequential ASMs do not change neither the set *Agent* nor the *pgm* function, so *Agent* and *pgm* do not appear in the definition of $\Delta_r(S)$.[26] i/o-rules are the only rules which involve also introducing a new element a into *Agent* (with a value assigned to $pgm(a)$) and a state initialization corresponding to the provided input, so that a executes its instance of the called rule.

- If r is an assignment rule $f(t_1, \ldots, t_n) := t_0$, then let $v_i = val_S(t_i)$.
  We define $\Delta_r(S) = \{\{((f, (v_1, \ldots, v_n)), v_0)\}\}$.
- If r is a branching rule IF φ THEN r', then let v be the truth value $val_S(\varphi)$. We define $\Delta_r(S) = \Delta_{r'}(S)$ for $v = \mathbf{true}$ and $\Delta_r(S) = \{\emptyset\}$ otherwise.
- If r is a parallel composition rule PAR $r_1 \| \ldots \| r_n$,[27] then we define $\Delta_r(S) = \{\Delta_1 \cup \cdots \cup \Delta_n \mid \Delta_i \in \Delta_{r_i}(S)\}$.
- If r is a bounded choice rule CHOOSE $r_1 \mid \cdots \mid r_n$, then we define $\Delta_r(S) = \Delta_{r_1}(S) \cup \cdots \cup \Delta_{r_n}(S)$.
- If r is a let rule LET $x = t$ IN r', then let $v = val_S(t)$, and define $\Delta_r(S) = \{[v/x].\Delta \mid [v/x].\Delta \in \Delta_{r'}(S)\}$.

Now consider the case that r is a call rule $t_0 \leftarrow N(t_1, \ldots, t_n)$. In this case let $t_0 = f(t'_1, \ldots, t'_k)$, and let $N(x_1, \ldots, x_n) = q$ be the declaration of the rule named N, with all free variables of q among $x_0, x_1, \ldots, x_n$.

In the call tree, the caller program r plays the role of the parent of the called child program that will be executed by a new agent c. The child program is an instance q_c of q with the outer function symbols of t_i for $1 \leq i \leq n$ classified as denoting input functions (which are not read by the caller program) and with the outer function symbol f of t_0 classified as denoting an output function (which is not updated

---

[25]If one wants to stick to interleaving executions, it suffices to determine A_i as singleton sets.
[26]Note that $\Delta_r(S)$ defines the set of update sets by which rule r changes state S into a successor state S'.
[27]Parallel composition rules are also written by displaying the components r_i vertically, omitting PAR and $\|$.

by the caller program).[28] The first two of the call relationship conditions are purely syntactical and can be assumed (without loss of generality) for caller and callee programs. The third condition is satisfied, because each local function symbol f of arity n is implicitly turned in a program instance into an $(n+1)$-ary function symbol, namely by adding the additional agent as environment parameter for the evaluation of terms with f as leading function symbol. Therefore, each local function of the callee is different from each local function of the caller, and to execute the call rule means to create a new agent c,[29] which is *CalledBy* the agent **self** that executes the call, to equip c with the fresh program instance q_c and INITIALIZE its state by the values of t_i, t'_j. This makes the callee ready to run and puts the caller into *Waiting* mode, in the sense defined by Definition 2.11 (except the trivial case that q_c is already *Terminated* when *Called* so that it will not be executed).

In other words we define $\Delta_r(S)$ as the singleton set containing the update set computed in state S by the following ASM, a rule we denote by $\text{CALL}(t_0 \leftarrow N(t_1, \ldots, t_n))$ which interprets the named i/o-rule $t_0 \leftarrow N(t_1, \ldots, t_n)$.

> **Definition 3.4.** $\text{CALL}(t_0 \leftarrow N(t_1, \ldots, t_n)) =$
>   let $N(x_1, \ldots, x_n) = q$ // declaration of N
>   let $v_1 = t_1, \ldots, v_n = t_n$ // input evaluation $\text{val}_S(t_i, \textbf{self})$ by caller
>   let $t_0 = f(t'_1, \ldots, t'_k)$
>   let $v'_1 = t'_1, \ldots, v'_k = t'_k$ // output location evaluation $\text{val}_S(t'_i, \textbf{self})$ by caller
>   let $c = \textbf{new } (Agent)$
> $pgm(c) := \textbf{amb } c \textbf{ in } q$ // equip callee with its program instance
> $\text{INSERT}(c, CalledBy(\textbf{self}))$
> $\text{INITIALIZE}(q_c, v_1/x_1, \ldots, v_n/x_n, f(v'_1, \ldots, v'_k)/x_o)$
> $CalledBy(c) := \emptyset$

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

Theorem 3.1. Each recursive ASM $\mathcal{M}$ defines a sequential recursive algorithm $\mathcal{M}_a$ (in the sense of Definition 2.10) such that the recursive runs of $\mathcal{M}$ can be step-for-step simulated by the runs of $\mathcal{M}_a$.

Proof. Let $\mathcal{M}$ be a recursive ASM. First of all we have to show that it satisfies the postulates in Definition 2.10 whereby it is a sequential recursive algorithm.

Each rule r belonging to $\mathcal{M}$, including named i/o-rules, is associated with a signature Σ^r given by the function symbols that appear in the rule or in the rule body if the rule is a named rule. This together with the agents c in **amb** c **in** r, defines the states (as sets of states of signature Σ^r, one per $r \in \mathcal{M}$) and gives the satisfaction of the abstract state postulate 2.

---

[28]The input parameters and the output location parameters are passed by value, so that the involved i/o-function symbols can be considered as belonging to the signature of caller and callee.

[29]The function **new** is assumed to yield for each invocation a fresh element, pairwise different ones for parallel invocations. One can define such a function also by an **import** construct which operates on a (possibly infinite) special *Reserve* set and comes with an additional constraint on the **par** construct to guarantee that parallel imports yield pairwise different fresh elements, see [15, 2.4.4]. See also the next footnote to Lemma 3.1.

The satisfaction of the branching time postulate 1 is an immediate consequence of the fact that for every state *S*, applying any recursive ASM rule in *S*, including named i/o-rules, yields a successor state.

For the satisfaction of the bounded exploration postulate 3 we exploit that by [23], each sequential ASM (i.e. without named i/o-rules) which appears in $\mathcal{M}$ is an nd-seq algorithm and thus satisfies the bounded exploration postulate. To extend this to the rules of $\mathcal{M}$, for each (of the finitely many) named i/o-rule we take every bounded exploration witness which appears in the rule body (including the parameters). By Definition 3.4 these witnesses determine the update sets yielded by the named i/o-rule in any given state.

By the definition of recursive ASM runs (Definition 3.3) and of the effect of a call rule step (Definition 3.4), the call step postulate 4 is satisfied by every recursive ASM.

As to the recursive run postulate 5, the run constraint is satisfied by the definition of recursive ASM runs (Definition 3.3). The bounded call tree branching constraint is satisfied, because there are only finitely many named i/o-rules in each of the finitely many rules $r \in \mathcal{M}$.

It remains to show that the recursive runs of $\mathcal{M}$ can be step-for-step simulated by corresponding runs of the sequential recursive algorithm $\mathcal{M}_a$ that is induced by this interpretation of $\mathcal{M}$. This follows from the two run characterizations in Postulate 5 and Definition 3.3 and from the fact that the successor relation of $\mathcal{M}_a$ is defined by the update sets which are yielded by the rules of $\mathcal{M}$ and define also the successor relation of $\mathcal{M}$. □

3.2. The Characterisation Theorem

We now show the converse of Theorem 3.1. The proof largely follows the ideas underlying the proof of the sequential ASM thesis in [23].

Theorem 3.2. For each sequential recursive algorithm $\mathcal{R}$ in the sense of Definition 2.10 there exists a recursive Abstract State Machine which is equivalent to $\mathcal{R}$ with respect to recursive runs (in the sense of Definition 3.3 and Postulate 5).

Proof. Let $\mathcal{R}$ denote any sequential recursive algorithm. Then for each state *S* and a successor state S' in a recursive run of $\mathcal{R}$ we obtain by Fact 2.1 an update set $\Delta(S, S') \in \mathbf{\Delta}(S)$. According to the Recursive Run Postulate 5 each such state transition is defined by one step of each of finitely many *Active* and not *Waiting* i/o-algorithms $\mathcal{A}_i$. Each of these i/o-algorithms is a fresh instance $\mathcal{A}_i$ of some component of $\mathcal{R}$. In particular, by the freshness and the independence condition in the Call Step Postulate 4, the instances $\mathcal{A}_i$ have disjoint signatures Σ_i and yield subruns with states $\mathrm{res}(S, \Sigma_i)$ and update sets $\Delta(\mathrm{res}(S, \Sigma_i), \mathrm{res}(S', \Sigma_i))$.

Consider now any such fresh instance $\mathcal{A}_i$ of a component $\mathcal{A} \in \mathcal{R}$. All function symbols used by $\mathcal{A}_i$ in its states and update sets are copies of function symbols of $\mathcal{A}$, labelled to ensure the freshness condition of the instance. Removing these labels we obtain for any state *S* of $\mathcal{A}$ and successor state S' of $\mathcal{A}$ pairs $(S, \Delta_{\mathcal{A}}(S, S'))$ with $\Delta_{\mathcal{A}}(S, S') \in \mathbf{\Delta}_{\mathcal{A}}(S)$. Let $D_{\mathcal{A}}$ be the set of all pairs $(S, \mathbf{\Delta}_{\mathcal{A}}(S))$ obtained this way, for any state *S* in the given recursive run of $\mathcal{R}$.

To complete the proof of the theorem it therefore suffices to show that the result of any $\mathcal{A}$-step in the given run, namely to apply an update set $\Delta_{\mathcal{A}}(S, S') \in \mathbf{\Delta}_{\mathcal{A}}(S)$ to a state *S*, can be described as result of a step of a recursive ASM rule $r_{\mathcal{A}}$, namely to apply to *S* an update set in $\mathbf{\Delta}_{r_{\mathcal{A}}}(S)$. This is established by the following Lemma 3.1. □

Lemma 3.1. For each $\mathcal{A}$ there exists a recursive ASM rule $r_{\mathcal{A}}$ with $\mathbf{\Delta}_{r_{\mathcal{A}}}(S) = \mathbf{\Delta}_{\mathcal{A}}(S)$ for all states S appearing in $D_{\mathcal{A}}$.

Proof. We choose a fixed bounded exploration witness $W_{\mathcal{A}}$ for each $\mathcal{A} \in \mathcal{R}$.

First we show that the argument values of any location ℓ in an update of $\mathcal{A}$ in any state S are critical values in S. The proof uses the same argument as in [23, Lemma 6.2].

To show the property consider an arbitrary update set $\Delta_{\mathcal{A}}(S, S') \in \mathbf{\Delta}_{\mathcal{A}}(S)$ and let $(\ell, v_0) \in \Delta_{\mathcal{A}}(S, S')$ be an update at location $\ell = (f, (v_1, \ldots, v_n))$. We show that the assumption that v_i is not a critical value leads to a contradiction.

If v_i is not a critical value, one can create a new structure $\hat{S}$ by swapping v_i with a fresh value w not appearing in S (e.g. w taken from the *Reserve* set)[30], so $\hat{S}$ is a state of $\mathcal{A}$. As v_i is assumed to not being critical, we must have $val_S(t) = val_{\hat{S}}(t)$ for all terms $t \in W_{\mathcal{A}}$. Therefore it follows from the bounded exploration postulate that $\mathbf{\Delta}_{\mathcal{A}}(S) = \mathbf{\Delta}_{\mathcal{A}}(\hat{S})$. This implies that the update $((f, (v_1, \ldots, v_n)), v_0)$ appears in at least one update set in $\mathbf{\Delta}_{\mathcal{A}}(\hat{S})$ (since $(\ell, v_0) \in \Delta_{\mathcal{A}}(S, S') \in \mathbf{\Delta}_{\mathcal{A}}(S)$), contradicting the fact that v_i does not occur in $\hat{S}$ and thus cannot occur in an update set created in this state.

Furthermore, for each pair $(S, \mathbf{\Delta}_{\mathcal{A}}(S)) \in D_{\mathcal{A}}$ we have a *recursion depth* function which indicates the maximal nesting of recursive calls performed starting in state S to compute the value of an output location in a possible successor state S'. The function is defined inductively as follows (induction on the call tree), taking the maximum over all successor states S' of S and over all updates leading from S to S'.

- rdepth$_{\mathcal{A}}(S) = \max\{\text{rdepth}_{\mathcal{A}}(S, S')) \mid \Delta_{\mathcal{A}}(S, S') \in \mathbf{\Delta}_{\mathcal{A}}(S)\}$
- rdepth$_{\mathcal{A}}(S, S')) = \max\{\text{depth}(\ell, v) \mid (\ell, v) \in \Delta_{\mathcal{A}}(S, S')\}$
- depth(ℓ, v) (with $\ell = (f, (v_1, \ldots, v_n))$) defined as follows:

  - Case 1: f is an output function symbol of a terminating recursive subcomputation started in S and leading to S'.
    Formally this means that for some callee $\mathcal{A}^c$ just activated in the run by $\mathcal{A}$, the restriction res(S, Σ^c) of S to the signature of $\mathcal{A}^c$ is an initial state S_0^c of a terminating run $S_0^c, S_1^c, \ldots, S_k^c$ of the callee, during which $\mathcal{A}$ remains waiting, and such that

    * the callee receives in state S the input from the caller, expressed by the equation res$(S_0^c, \Sigma_{in}^c) = $ res(S, Σ_{in}^c),
    * the caller receives in state S' the callee's output in the callee's final state, formally res$(S', \Sigma_{out}^c) = $ res(S_k^c, Σ_{out}^c) for the final state S_k^c for $\mathcal{A}^c$

    and $f \in \Sigma_{out}^c$.
    Then the depth of the update is defined as successor of the maximal recursion depth between any two successive states in the subcomputation, formalized by the equation depth$(\ell, v) = \max\{\text{rdepth}_{\mathcal{A}^c}(S_i^c, S_{i+1}^c) \mid 0 \leq i \leq k-1\} + 1$.
  - Case 2: Otherwise. Then we define depth$(\ell, v) = 0$.

We now proceed by a case distinction for rdepth$_{\mathcal{A}}(S)$.

---

[30]The concept of a *Reserve* set is sketched in [22] and [23]. *Reserve* can be described as a (possibly infinite) subset of the base set with elements that are distinguished from the other (the *Active*) base set elements by not being allowed in any state to be in the domain and range of any basic function of the state (except for the characteristic function of *Reserve*). For a formalization of the *Reserve Condition* algorithms are required to satisfy see [15, Sect.2.4.4], in particular the rules in Table 2.3, p.77. Using such *Reserve* sets permits to restrict the bounded exploration postulate to states with same base set.

Case 1: $\mathrm{rdepth}_{\mathcal{A}}(S)$ is defined for all states $S \in \mathcal{S}_{\mathcal{A}}$. In this case we proceed by induction over $d = \max\{\mathrm{rdepth}_{\mathcal{A}}(S) \mid S \in \mathcal{S}_{\mathcal{A}}\}$. The base case is de facto the proof of the non-deterministic sequential ASM thesis.

Induction Base: Let $d = 0$. First we construct for every state S with $\mathrm{rdepth}_{\mathcal{A}}(S) = 0$ a (sequential ASM) rule $r_{\mathcal{A},S}$ whose application to S yields the updates sets defined for S by $\mathcal{A}$, formally such that $\Delta_{r_{\mathcal{A},S}}(S) = \Delta_{\mathcal{A}}(S)$, and the same for all states $\hat{S}$ that are 'similar' to S (as defined below) wrt the bounded exploration witness $W_{\mathcal{A}}$.

To show this let S be a state with $\mathrm{rdepth}_{\mathcal{A}}(S) = 0$ and let S' be any successor state, resulting from S by applying the updates in $\Delta_{\mathcal{A}}(S, S') \in \Delta_{\mathcal{A}}(S)$. Consider any such update $(\ell, v_0) \in \Delta_{\mathcal{A}}(S, S')$ at location $\ell = (f, (v_1, \ldots, v_n))$. As all v_i are critical values and there is no child with $f \in \Sigma_{out}^c$, there exist terms $t_i \in W_{\mathcal{A}}$ with $\mathrm{val}_S(t_i) = v_i$. Thus, the assignment rule $f(t_1, \ldots, t_n) := t_0$ yields the given update in state S and the parallel composition $r_{\mathcal{A},S,S'}$ of all these assignment rules for every update in $\Delta_{\mathcal{A}}(S, S')$ yields in S the update set $\Delta_{\mathcal{A}}(S, S')$. Therefore the bounded choice composition of these rules for all successor states S' defines a rule $r_{\mathcal{A},S}$ with $\Delta_{r_{\mathcal{A},S}}(S) = \Delta_{\mathcal{A}}(S)$.

Next we extend $\Delta_{r_{\mathcal{A},S}}(S) = \Delta_{\mathcal{A}}(S)$ from S to all states which are 'similar' to S with respect to the bounded exploration witness $W_{\mathcal{A}}$.[31]

(i) First, we show the equation $\Delta_{r_{\mathcal{A},S}}(\hat{S}) = \Delta_{\mathcal{A}}(\hat{S})$ for every state $\hat{S}$ which coincides with S on $W_{\mathcal{A}}$. In fact, if S and $\hat{S}$ coincide on $W_{\mathcal{A}}$, then $\Delta_{\mathcal{A}}(S) = \Delta_{r_{\mathcal{A},S}}(S) = \Delta_{r_{\mathcal{A},S}}(\hat{S})$ holds because the rule $r_{\mathcal{A},S}$ only uses terms in $W_{\mathcal{A}}$, which have the same values in S and $\hat{S}$. We further have $\Delta_{\mathcal{A}}(S) = \Delta_{\mathcal{A}}(\hat{S})$ due to the bounded exploration postulate. These equations together give $\Delta_{r_{\mathcal{A},S}}(\hat{S}) = \Delta_{\mathcal{A}}(\hat{S})$.

(ii) Second, we show that $\Delta_{r_{\mathcal{A},S}}(S_1) = \Delta_{\mathcal{A}}(S_1)$ carries over to isomorphic states S_2. Let S_1, S_2 be isomorphic states for which $\Delta_{r_{\mathcal{A},S}}(S_1) = \Delta_{\mathcal{A}}(S_1)$ is true. Let ζ be the isomorphism with $\zeta S_2 = S_1$. Then we have $\zeta \Delta_{r_{\mathcal{A},S}}(S_2) = \Delta_{r_{\mathcal{A},S}}(S_1)$ (because all ASMs satisfy the Abstract State Postulate) and also $\zeta \Delta_{\mathcal{A}}(S_2) = \Delta_{\mathcal{A}}(S_1)$ (by the Abstract State Postulate). These equations together give $\zeta \Delta_{r_{\mathcal{A},S}}(S_2) = \zeta \Delta_{\mathcal{A}}(S_2)$ and hence also $\Delta_{r_{\mathcal{A},S}}(S_2) = \Delta_{\mathcal{A}}(S_2)$.

(iii) Third, we conclude from (i) and (ii) that the equation $\Delta_{r_{\mathcal{A},S}}(\hat{S}) = \Delta_{\mathcal{A}}(\hat{S})$ holds for every state $\hat{S}$ that is $W_{\mathcal{A}}$-similar to S. We define states S_1, S_2 to be $W_{\mathcal{A}}$-*similar* iff they identify the same critical terms, i.e. formally iff $\mathrm{val}_{S_1}(t_1) = \mathrm{val}_{S_1}(t_2) \Leftrightarrow \mathrm{val}_{S_2}(t_1) = \mathrm{val}_{S_2}(t_2)$ holds for all terms $t_1, t_2 \in W_{\mathcal{A}}$. No let $\hat{S}$ be any state that is $W_{\mathcal{A}}$-similar to S. We can assume without loss of generality that S and $\hat{S}$ are disjoint.

In fact, if this is not the case consider a state $\ddot{S}$ isomorphic to $\hat{S}$, in which each value that appears also in S is replaced by a fresh one. Then $\ddot{S}$ is disjoint from S and by construction $W_{\mathcal{A}}$-similar to $\hat{S}$, hence also $W_{\mathcal{A}}$-similar to S.

Now define a structure S^* isomorphic to $\hat{S}$ by replacing $\mathrm{val}_{\hat{S}}(t)$ by $\mathrm{val}_S(t)$ for all $t \in W_{\mathcal{A}}$. The definition of S^* is consistent because since S and $\hat{S}$ are $W_{\mathcal{A}}$-similar, $\mathrm{val}_S(t_1) = \mathrm{val}_S(t_2) \Leftrightarrow \mathrm{val}_{\hat{S}}(t_1) = \mathrm{val}_{\hat{S}}(t_2)$ holds for all terms $t_1, t_2 \in W_{\mathcal{A}}$. Since S and S^* coincide on $W_{\mathcal{A}}$, we obtain by (i) that $\Delta_{r_{\mathcal{A},S}}(S^*) = \Delta_{\mathcal{A}}(S^*)$ which by (ii) implies $\Delta_{r_{\mathcal{A},S}}(\hat{S}) = \Delta_{\mathcal{A}}(\hat{S})$.

To complete the proof for the induction base we exploit that $W_{\mathcal{A}}$ is finite, hence there are only finitely many partitions of $W_{\mathcal{A}}$ and only finitely many $W_{\mathcal{A}}$-similarity classes $[S_i]_{W_{\mathcal{A}}}$ (say $i = 1, \ldots, m$). For each such class we define a formula φ_i such that for each state S this formula evaluates in S to true if and only

---

[31] These cases are captured in Lemmata 6.7, 6.8. and 6.9 in [23]

if $S \in [S_i]_{W_{\mathcal{A}}}$ holds. φ_i formalizes the similarity type of S_i by the conjunction of all equations $t_1 = t_2$ with $val_{S_i}(t_1) = val_{S_i}(t_2)$ and all inequalities $t_1 \neq t_2$ with $val_{S_i}(t_1) \neq val_{S_i}(t_2)$ for all critical terms $t_1, t_2 \in W_{\mathcal{A}}$. Then we can define the rule $r_{\mathcal{A}}$ as follows:

$$\texttt{PAR}\,(\texttt{IF } \varphi_1 \texttt{ THEN } r_{\mathcal{A},S_1}) \parallel \ldots \parallel (\texttt{IF } \varphi_m \texttt{ THEN } r_{\mathcal{A},S_m})$$

Induction Step: Let $d > 0$. For a state S with $\mathrm{rdepth}_{\mathcal{A}}(S) = 0$ we proceed as in the base case to construct a rule $r_{\mathcal{A},S}$ such that $\boldsymbol{\Delta}_{r_{\mathcal{A},S}}(\hat{S}) = \boldsymbol{\Delta}_{\mathcal{A}}(\hat{S})$ holds for all states $\hat{S}$ that are $W_{\mathcal{A}}$-similar to S. For the case that $\mathrm{rdepth}_{\mathcal{A}}(S) > 0$ we construct such a rule as follows.

Let $(\ell, v_0) \in \Delta_{\mathcal{A}}(S, S') \in \boldsymbol{\Delta}_{\mathcal{A}}(S)$ be any update at location $\ell = (f, (v_1, \ldots, v_n))$. Then we have two cases:

(i) If there is no child algorithm $\mathcal{A}^c$ with $f \in \Sigma^c_{out}$, then we argue as in the base case, i.e. as all v_i are critical values, there exist terms $t_i \in W_{\mathcal{A}}$ with $val_S(t_i) = v_i$, and thus the assignment rule $f(t_1, \ldots, t_n) := t_0$ produces the given update in state S.

(ii) If we have $f \in \Sigma^c_{out}$ for some child algorithm $\mathcal{A}^c$, i.e. f is an output function symbol of an algorithm that is called by $\mathcal{A}$, then according to our assumption on call relationships, locations with such function symbols never appear in an update set created by $\mathcal{A}$ itself, so the update results from a final state of a run of $\mathcal{A}^c$. Let this run be $S_0^c, S_1^c, \ldots$ with final state S_k^c.
Then we must have $\max\{\mathrm{rdepth}_{\mathcal{A}^c}(S) \mid S \in \mathcal{S}_{\mathcal{A}^c}\} \leq d - 1$, so we can apply the induction hypothesis. That is, there exists an ASM rule $r_{\mathcal{A}^c}$ with $\boldsymbol{\Delta}_{r_{\mathcal{A}^c}}(S_i^c) = \boldsymbol{\Delta}_{\mathcal{A}^c}(S_i^c)$ for all $0 \leq i \leq k$.

- As the values v_i for $i = 1, \ldots, n$ are critical in S, we find terms $\hat{t}_i \in W_{\mathcal{A}}$ with $val_S(\hat{t}_i) = v_i$.
- As v_0 results from an update made by $\mathcal{A}^c$, it is critical in S_ℓ^c for some $\ell < k$, so there must exist a term $t_0 \in W_{\mathcal{A}^c}$ with $val_{S_\ell^c}(t_0) = v_0$.
- The initial state S_0^c is defined by values v_i' ($i = 1, \ldots, m$) of input locations, which are critical for S_0^c, which gives rise to terms $t_i' \in W_{\mathcal{A}^c}$ with $val_{S_0^c}(t_i') = v_i'$.
- As the input values v_i' for $\mathcal{A}^c$ have been produced by updates made by $\mathcal{A}$ we further find terms $t_i \in W_{\mathcal{A}}$ with $val_S(t_i) = v_i'$.

Using a name N for the rule of $\mathcal{A}^c$ we obtain a named rule $f(\hat{t}_1, \ldots, \hat{t}_n) \leftarrow N(t_1', \ldots, t_m')$. Furthermore, according to the definition of update sets produced by call rules, we see that the call $t_0 \leftarrow N(t_1, \ldots, t_m)$ produces the update $((f, (v_1, \ldots, v_n)), v_0)$.

Again the parallel composition of all the assignment and call rules (for every update in $\Delta_{\mathcal{A}}(S, S')$) yields in S the update set $\Delta_{\mathcal{A}}(S, S')$. The bounded choice composition of these rules for all successor states S' defines a rule $r_{\mathcal{A},S}$ with $\boldsymbol{\Delta}_{r_{\mathcal{A},S}}(S) = \boldsymbol{\Delta}_{\mathcal{A}}(S)$.

Using again the same arguments as in (i), (ii), and (iii) for the base case we get $\boldsymbol{\Delta}_{r_{\mathcal{A},S}}(\hat{S}) = \boldsymbol{\Delta}_{\mathcal{A}}(\hat{S})$ for all states $\hat{S}$ that are $W_{\mathcal{A}}$-similar to S, and exploiting the finiteness of bounded exploration witnesses we obtain again the rule $r_{\mathcal{A}}$ with the required property, i.e. $\boldsymbol{\Delta}_{r_{\mathcal{A}}}(S) = \boldsymbol{\Delta}_{\mathcal{A}}(S)$ for all states S.

Case 2: Assume that $\mathrm{rdepth}_{\mathcal{A}}(S)$ is not defined for all states $S \in \mathcal{S}_{\mathcal{A}}$.

For states S for which $\mathrm{rdepth}_{\mathcal{A}}(S)$ is defined we use the same construction as above to obtain a rule $r_{\mathcal{A},S}$ with $\boldsymbol{\Delta}_{r_{\mathcal{A},S}}(S) = \boldsymbol{\Delta}_{\mathcal{A}}(S)$.

Now take a state S, for which $\mathrm{rdepth}_{\mathcal{A}}(S)$ is not defined. Then there exists a child algorithm $\mathcal{A}^c$ with an initial state S_0^c initiated by $\mathcal{A}$, but no run starting in S_0^c leads to a final state. Then the input values

define terms t_i and t_i' for $i = 1, \ldots, m$ in the same way as for the construction of the call rule above. We can use arbitrary critical terms t_0, $\hat{t}_i$ ($i = 1, \ldots, n$) and a named rule $f(\hat{t}_1, \ldots, \hat{t}_n) \leftarrow N(t_1', \ldots, t_m')$ as before, then the call rule $t_0 \leftarrow N(t_1, \ldots, t_m)$ will lead to the run of $\mathcal{A}^c$ without final state.

The rule $r_\mathcal{A}$ with the required property, i.e. $\boldsymbol{\Delta}_{r_\mathcal{A}}(S) = \boldsymbol{\Delta}_\mathcal{A}(S)$ for all states S, then results by applying again the same arguments as above. $\qquad\square$

4. Examples

We now present three simple examples of sequential recursive algorithms, *mergesort*, *quicksort* and the *sieve of Eratosthenes*. These algorithms will be specified by recursive ASMs, which we use to illustrate the concepts in our axiomatisation. Furthermore, we show that sequential recursion can already be expressed by ASMs, as these support unbounded parallelelism. We illustrate this for the first two selected algorithms. This shows that unbounded parallelism, which is a decisive feature of ASMs, is much stronger than sequential recursion, and there is no need to separately investigate recursive parallel algorithms.

4.1. Mergesort

We first give a specification of a recursive ASM comprising two named rules *sort* (the main rule) and *merge*.

```
sorted_list ← sort(unsorted_list) =
      IF sorted_list = undef THEN
      LET n = length(unsorted_list)
      IN   PAR
            (IF n ≤ 1 THEN sorted_list := unsorted_list ) ‖
            (IF n > 1 ∧ sorted_list₁ = undef ∧ sorted_list₂ = undef
            THEN LET list₁ = IL.length(L) = ⌊n/2⌋ ∧ ∃L′.concat(L, L′) = unsorted_list
                  IN    LET list₂ = IL′.concat(list₁, L′) = unsorted_list
                        IN   PAR
                              (sorted_list₁ ← sort(list₁) ‖
                              sorted_list₂ ← sort(list₂)) ‖
            (IF n > 1 ∧ sorted_list₁ ≠ undef ∧ sorted_list₂ ≠ undef
            THEN sorted_list ← merge(sorted_list₁,sorted_list₂))
```

Here we used terms of the form $\mathbf{I}x.\varphi$ with a variable x and a formula φ, in which x is free to denote *the unique value x satisfying φ*[32].

```
merged_list ← merge(inlist₁,inlist₂) =
```

---

[32]In the cases above we could have used equivalently $@x.\varphi$ denoting an *arbitrary value x satisfying φ*, but emphasising that the value exists and is indeed unique makes the specification clearer. Both kinds of terms were originally introduced by David Hilbert using ϵ instead of $@$ and ι instead of $\mathbf{I}$. The notation $@$ reflects the common use of ANY ($@$) in rigorous methods and $\mathbf{I}$ comes from Fourman's formalisation of higher-order intuitionistic logic. We use it here for notational convenience only, to succintly define the split of a list into a left half and a right half.

```
IF merged_list = undef THEN
PAR
(IF inlist₁ = [] THEN merged_list := inlist₂) ‖
(IF inlist₂ = [] THEN merged_list := inlist₁) ‖
(IF inlist₁ ≠ [] ∧ inlist₂ ≠ []
 THEN LET x₁ = head(inlist₁) IN
        LET restlist₁ = tail(inlist₁) IN
        LET x₂ = head(inlist₂) IN
        LET restlist₂ = tail(inlist₂) IN
            PAR
            (IF x₁ ≤ x₂ ∧ merged_restlist = undef
             THEN merged_restlist ← merge(restlist₁, inlist₂)) ‖
            (IF x₁ > x₂ ∧ merged_restlist = undef
             THEN merged_restlist ← merge(inlist₁, restlist₂)) ‖
            (IF x₁ ≤ x₂ ∧ merged_restlist ≠ undef
             THEN merged_list := concat([x₁],merged_restlist)) ‖
            (IF x₁ > x₂ ∧ merged_restlist ≠ undef
             THEN merged_list := concat([x₂],merged_restlist))
```

The root of the call tree is labelled by *sort*. Every node labelled by *sort* has three children labelled by instances of *sort*, *sort* and *merge*, respectively. A node labelled by *merge* has two children, both labelled by *merge*.

Since in a run of a sequential recursive algorithm, in each step only finitely many algorithms are executed at the same time and these do not stand in an ancestor relationship, we can in fact instead of using copies of the algorithms use directly copies of the locations and run all these part-algorithms in parallel. They can only make a step, if their input has been defined. The parallelism is then unbounded, but in each step only finitely many parallel branches are executed. We present such a specification using parallel ASMs for the mergesort algorithm, using sequences of indeces $0, 1, 2$ for the parameters. The original input we are interested in is *unsorted_list*([ ]), the computed output of interest is *sorted_list*([ ]).[33]

```
forall I ∈ {0, 1, 2}* do
  if sorted_list(I) = undef and unsorted_list(I) ≠ undef then
    let n = length(unsorted_list(I))
    if n ≤ 1 then sorted_list(I) := unsorted_list(I)
    if n > 1 then
      SPLITLIST(I)
      if SortedSubLists(I) then
        if SubListsToBeMerged(I)
          then MERGE(I, sorted_list(I0), sorted_list(I1))
          else OUTPUTSORTINGRESULT(I)
```

The three submachines are defined as follows:

---

[33] We use the vertical notation of r_i instead of PAR $r_1 \| \ldots \| r_n$.

$\text{SPLITLIST}(I) =$
  if $unsorted\_list(I0) = undef = unsorted\_list(I1)$ **then**
    let (l_0, l_1) **with**
      $length(l_0) = \lfloor \frac{n}{2} \rfloor$ **and** $unsorted\_list(I) = concat(l_0, l_1)$ **in**
        $unsorted\_list(I0) := l_0$
        $unsorted\_list(I1) := l_1$
where
  $SortedSubLists(I)$ **iff**
    $sorted\_list(I0) \neq undef$ **and** $sorted\_list(I1) \neq undef$
  $SubListsToBeMerged(I)$ **iff** $merged\_list(I) = undef$
  $\text{OUTPUTSORTINGRESULT}(I) = (sorted\_list(I) := merged\_list(I))$

$\text{MERGE}(I, l_1, l_2) =$
  if $l_1 = [\,]$ **then** $merged\_list(I) := l_2$
  else if $l_2 = [\,]$ **then** $merged\_list(I) := l_1$
    else
      $\text{INITIALIZEINPUTLISTS}(I, l_1, l_2)$
      $\text{INITIALIZERESTLISTS}(I, l_1, l_2)$
      $\text{ONEMERGESTEP}(I)$
where
  $\text{INITIALIZEINPUTLISTS}(I, l_1, l_2) =$
    if $inlist_1(I) = inlist_2(I) = undef$ **THEN**
      $inlist_1(I) := l_1$
      $inlist_2(I) := l_2$
  $\text{INITIALIZERESTLISTS}(I, l_1, l_2) =$
    if $restlist_1(I) = restlist_2(I) = undef$ **then**
      $restlist_1(I) := tail(l_1)$
      $restlist_2(I) := tail(l_2)$
  $\text{ONEMERGESTEP}(I) =$
    if $inlist_1(I) \neq [\,] \neq inlist_2(I)$ **and** $restlist_1(I) \neq undef \neq restlist_2(I)$ **then**
      let $x_1 = head(inlist_1(I)), x_2 = head(inlist_2(I))$
        if $x_1 \leq x_2$ **then**
          if $merged\_list(I2) = undef$
            then
              $inlist_1(I2) := restlist_1(I)$
              $inlist_2(I2) := inlist_2(I)$
            else $merged\_list(I) := concat([x_1], merged\_list(I2))$
        if $x_1 > x_2$ **then**
          if $merged\_list(I2) = undef$
            then
              $inlist_1(I2) := inlist_1(I)$
              $inlist_2(I2) := restlist_2(I)$
            else $merged\_list(I) := concat([x_2], merged\_list(I2))$

Remark. Note that in this way every sequential recursive algorithm can be captured by an ASM as defined in [15], i.e. exploiting `FORALL` rules. This shows that the concept of unbounded parallelism that is decisive for the practicality of ASMs and is theoretically founded by the behavioural theory of parallel algorithms [19] covers the needs of sequential recursive algorithms.[34] However, parallel algorithms are a much wider class than sequential recursive algorithms.[35] We dispense with discussing this any further here. It further means that the concept of recursion extends the class of sequential algorithms, but an extension of the class of parallel algorithms by means of recursion is meaningless.

Furthermore, instead of using a single parallel ASM with the rule above we can also define agents a_I for all indices $I \in \{0, 1, 2\}^*$, each associated with an ASM $\mathcal{M}_I$ using a rule r_I defined as above without the outermost `FORALL`. The effect on runs is that for the concurrent runs of the concurrent ASM $\{(a_I, \mathcal{M}_I) \mid I \in \{0, 1, 2\}^*\}$ the individual agents operate asynchronously, which permits additional runs. However, these runs only reflect the combination of runs of the individual agent machines at different paces as already mentioned in the introduction. This will be exploited in Section 5.

4.2. Quicksort

Again we first give a specification of a sequential recursive ASM with a single named i/o-rule *qsort*.

> sorted_list $\leftarrow$ *qsort*(unsorted_list) =
> PAR (IF unsorted_list = [] THEN sorted_list := []) ||
> (IF unsorted_list $\neq$ [] $\wedge$ sorted_list$_1$ = *undef* $\wedge$ sorted_list$_2$ = *undef*
> THEN LET $x =$ head(unsorted_list) IN
> LET unsorted_list$_1$ = sublist[$y \mid y < x$](unsorted_list) IN
> LET unsorted_list$_2$ = sublist[$y \mid y > x$](unsorted_list) IN
> PAR (sorted_list$_1 \leftarrow$ *qsort*(unsorted_list$_1$)) ||
> (sorted_list$_2 \leftarrow$ *qsort*(unsorted_list$_2$)))
> (IF unsorted_list $\neq$ [] $\wedge$ sorted_list$_1 \neq$ *undef* $\wedge$ sorted_list$_2 \neq$ *undef*
> THEN sorted_list := concat(sorted_list$_1$, concat(head(unsorted_list), sorted_list$_2$)))

In this case all nodes of the call tree are labelled by instances of the one i/o-algorithm *qsort*. The following ASM rule shows how to capture the algorithm by a parallel ASM.

> FORALL $I \in \{0, 1\}^*$ DO
> PAR (IF unsorted_list(I) = [ ] THEN sorted_list(I) := [ ]) ||
> (IF unsorted_list(I) $\neq$ [ ] $\wedge$ unsorted_list(I) $\neq$ *undef*
> THEN PAR
> (IF unsorted_list($I0$) = *undef* $\wedge$ unsorted_list($I1$) = *undef*

---

[34] A reviewer objected that we cannot describe 'recursive algorithms with unbounded number of callees'. In fact one cannot do this with sequential recursive ASM because by their sequentiality they satisfy the Bounded Exploration and the Bounded Call Tree Branching Postulates. But one can do it using ASMs with unbounded `FORALL`. Similarly, the same reviewer's question how to define 'recursive symmetry-braking algorithms' can be answered by using ASMs with unbounded choice, which are not any more sequential algorithms.

[35] This explains why in the behavioural theories developed so far the emphasis was more on parallel than on recursive algorithms. Nonetheless, the ASM rule above expressing the sequential recursive algorithm is somehow easier to read than the rule for the behaviourally equivalent parallel ASM, because in the latter the copies of locations are made explicit.

$$\text{THEN LET } x = \text{head}(\text{unsorted\_list}(I)) \text{ IN}$$
$$\text{PAR} \quad (\text{unsorted\_list}(I0) := \text{sublist}[y \mid y < x](\text{unsorted\_list}(I))) \parallel$$
$$(\text{unsorted\_list}(I1) := \text{sublist}[y \mid y > x](\text{unsorted\_list}(I)))) \parallel$$
$$(\text{IF sorted\_list}(I0) \neq \textit{undef} \wedge \text{sorted\_list}(I1) \neq \textit{undef}$$
$$\text{THEN sorted\_list}(I) := \text{concat}(\text{sorted\_list}(I0),$$
$$\text{concat}(\text{head}(\text{unsorted\_list}(I)), \text{sorted\_list}(I1)))))$$

As for *mergesort*, one can define a concurrent ASM $\{(a_I, \mathcal{M}_I) \mid I \in \{0,1\}^*\}$, where the rule r_I of the machine $\mathcal{M}_I$ is defined as above without the outermost FORALL.

4.3. Sieve of Eratosthenes

The *sieve of Eratosthenes* algorithm computes all prime numbers. Starting from the set $\{x \in \mathbb{N} \mid x \geq 2\}$ as the start sieve, the smallest number of the sieve is added to the output (or printed)—so the output will be infinite—and all elements of the sieve that are divisable by the number added to the output are removed from the sieve. The following is a simple recursive ASM rule for this algorithm.

$$\leftarrow \textit{eratosthenes}(\text{sieve}) =$$
$$\text{LET} \quad p = \min(\text{sieve}) \text{ IN}$$
$$\text{LET} \quad \text{reduced\_sieve} = \{x \in \text{sieve} \mid p \nmid x\} \text{ IN}$$
$$\text{PAR} \quad \leftarrow \textit{eratosthenes}(\text{reduced\_sieve}) \parallel \text{out\_prime} := p$$

More generally, if the input sieve is any subset of $\mathbb{N}$, then the algorithm will return all numbers in $\{x \in \text{sieve} \mid \forall y \in \text{sieve} \ (y \mid x \Rightarrow y = x)\}$ (via values assigned to out_prime). None of the calls in this algorithm will reach a final state.

The crucial problem with this algorithm is not only the infinite input and output (both can in principle be handled using streams), but the fact that the involved computation of reduced_sieve requires the availability of all elements of sieve, i.e. in general infinitely many. A remedy is the following rule, which keeps sieve constant, but uses all previously determined output values in a set divisors (initially $\emptyset$) to determine the next number in the output, which requires only the investigation of an initial segment of sieve.

$$\leftarrow \textit{eratos}(\text{divisors},\text{sieve}) =$$
$$\text{LET} \quad p = \min(\{x \in \text{sieve} \mid \forall d \in \text{divisors} \ \ d \nmid x\}) \text{ IN}$$
$$\text{PAR} \quad \leftarrow \textit{eratos}(\text{divisors} \cup \{p\}, \text{sieve}) \parallel \text{out\_prime} := p$$

It is an easy exercise to describe this algorithm by a parallel or a concurrent ASM.

5. Recursive Algorithms and Partial-Order Runs

In their response to Moschovakis' criticism [31] Blass and Gurevich argued in [6] with the capture of sequential recursive algorithms by what is known as *distributed ASMs*. The semantics of distributed ASMs has been defined by *partial-order runs* [22], which later have been recognized as too restrictive and have been replaced in [12] by a rather comprehensive definition of concurrency. In this section we investigate the relationship between distributed ASMs and recursive algorithms/ASMs, as defined in Section 2. We will show (a precise version of) the following theorem.

Theorem 5.1. Characterization of partial-order runs. Recursive algorithms are exactly those finitely-composed concurrent algorithms $\mathcal{C}$ with nd-seq components such that all concurrent $\mathcal{C}$-runs are definable by partial-order runs.

The decisive notions used in this theorem such as partial-order runs and finitely-composed concurrent algorithms will be formally introduced below, together with the corresponding concepts for concurrent ASMs. The theorem fortifies the argument that partial-order runs are too weak a concept to serve as a semantic foundation for concurrent algorithms[36]. The latter aspect has been overcome by the definition of concurrent ASM runs and the corresponding concurrent ASM thesis in [12].

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 their nd-seq programs, one can simulate them even by a non-deterministic sequential algorithm (see Theorem 5.4). An interesting example of this special case are partial-order runs of Process Rewrite Systems [27] (see Corollary 5.1).

To prove the theorem we use the characterization of runs of sequential recursive algorithms as recursive runs of recursive ASMs (Theorem 3.2) and of runs of concurrent algorithms as concurrent ASM runs (see [12]).

5.1. Partial-Order Runs

Syntactically, a concurrent algorithm $\mathcal{C}$ is defined as a family of algorithms $alg(a)$, each associated with ('indexed by') an agent $a \in Agent$ (see [22], [12]) that executes the algorithm in concurrent runs. These algorithms are often assumed to be (possibly non-deterministic) sequential algorithms, though this restriction is not necessary in general. In our case here this restriction is, however, important, as we have seen in Section 4 that without this restriction, permitting unbounded **forall** and **choose** constructs, we obtain algorithms far more powerful than the recursive ones. Sometimes it is also assumed that the *Agent* set is finite, a special case we consider in Sect. 5.3.

In a concurrent run, as in recursive runs, different agents can be associated dynamically with different instances of the same algorithm. Therefore, when relating concurrent runs of a concurrent algorithm $\mathcal{C}$ to recursive runs of a sequential recursive algorithm—which by Definition 2.10 is a finite set of i/o-algorithms—we need to finitely compose $\mathcal{C}$, namely by a finite set of nd-seq components of which each $alg(a) \in \mathcal{C}$ is an instance.

In a concurrent run as defined in [12], 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 $alg(a)$-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 the successor state S_{i+1}, and that a *move* starts in S_j and contributes to updating

---

[36] As our discussion in the previous section shows, even the concept of unbounded parallelism, by means of which synchronous parallelism is captured, is stronger than recursion.

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 5.1. A *concurrent ASM* is defined as a family $\mathcal{C}$ of ASMs asm_a (also called programs and written $pgm(a)$) 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. Below we consider the special cases that *Agent* is a static finite set (see Section 5.3) or a dynamic set all of whose members are equipped however with instances of a fixed (static) finite set of programs (see Definition 5.3). In Definition 5.2 below the set of *Agent*s is fixed by the set of *M*oves.

For the reason explained above, in the following we restrict our attention to concurrent ASMs in which each component asm_a is an nd-seq ASM.

In [22] Gurevich defined the notion of *partial-order run* of concurrent algorithms.[37] The partial order is defined on the set of single moves of the agents which execute the individual algorithms. For a nd-seq algorithm $\mathcal{A}$, to make one *move* means to perform one step in a state S, as defined by the Branching Time Postulate 1, applying a set $\Delta(S, S') \in \mathbf{\Delta}(S)$ of updates to S.

Definition 5.2. Let $\mathcal{C} = \{(a, \text{alg}(a))\}_{a \in Agent}$ be a concurrent algorithm, in which each $\text{alg}(a)$ 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 $\text{alg}(a)$ ($a \in Agent$), a function $\text{ag} : M \rightarrow Agent$ 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 $\text{ag}(m) = \text{ag}(m')$ we either have $m \leq m'$ or $m' \leq m$.

coherence. For each finite initial segment $M' \subseteq M$ (i.e. 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 $\mathcal{A}_i$ such that for each maximum element $m \in M'$ the state $\sigma(M')$ is the result of applying m to $\sigma(M' - \{m\})$.

In order to characterise recursive ASM runs in terms of partial-order runs of a concurrent ASM $\mathcal{C}$ several restrictions have to be made. First of all the component ASMs must be nd-seq ASMs, an assumption we already justified above. Second, the component machines asm_a can only be copies (read: instances) of finitely many different ASMs, which we will call the program base of $\mathcal{C}$.[38] Third, runs must

---

[37]Note that in [22] Gurevich actually uses the wording *distributed algorithm* instead of concurrent algorithm, whereas we prefer to stick with the notation from [12]. One reason for this is that distribution requires also to discuss (physical) notations and messaging (as handled in [13]), whereas concurrency abstracts from this.

[38]This reflects the stipulation in [22] that in concurrent ASMs the agents are equipped with instances of programs which are taken from 'a finite indexed set of single-agent programs' (p.31).

be started by executing a distinguished main component.[39] We capture these restrictions by the notion of *finitely composed* concurrent ASM.

Definition 5.3. A concurrent ASM $\mathcal{C}$ is *finitely composed* iff (i) and (ii) hold:

(i) There exists a finite set $\mathcal{B}$ of nd-seq ASMs 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 $\mathcal{C}$ this implies that they start with a minimal move which consists in executing the program $\text{asm}(a_0) = $ **amb** a_0 **in** *main*.

(iii) Any program base component may contain rules of form **let** $a = $ **new** (*Agent*) **in** r.[40] 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.

We are now ready to more precisely state and prove the first part of Theorem 5.1. It should come as no surprise; it provides the justification for the argumentation by Blass and Gurevich in [6].

Theorem 5.2. For every recursive ASM $\mathcal{R}$ one can construct an equivalent finitely composed concurrent ASM $\mathcal{C}_{\mathcal{R}}$ with nd-seq ASM components such that 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

$$r^* = \textbf{if } \textit{Active}(r) \textbf{ and not } \textit{Waiting}(r) \textbf{ then } r.$$

That is, 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 is needed, because in every step of a recursive run of $\mathcal{R}$ only *Active* and not *Waiting* rules are executed.

In doing so we use for each call rule $r \in \mathcal{R}$ in the **then** part of r^* instead of $t_0 \leftarrow N(t_1, \ldots, t_n)$ its interpretation by the ASM rule $\text{CALL}(t_0 \leftarrow N(t_1, \ldots, t_n))$ defined in Sect. 3.1. The definition of r^* obviously guarantees the behavioral equivalence of $\mathcal{R}$ and $\mathcal{C}_{\mathcal{R}}$: in each run step the same *Active* and not *Waiting* rules r respectively r^* and their agents are selected for their simultaneous execution. Remember that, by the definition of $\text{CALL}(i/o\text{-}rule)$, 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 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 $\mathcal{R}$.

---

[39]The restriction to one component is equivalent to, but notationally simplifies, the requirement stated in [22, 6.2, p.31] for concurrent ASM runs that in initial states there are only finitely many agents, each equipped with a program.

[40]This reflects the stipulation for concurrent ASMs in [22] that 'An agent a can *make a move* at S by firing Prog(a) ... and change S accordingly. As part of the move, a may create new agents' (p.32), which then may contribute by their moves to the run in which they were created.

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 constraint on finite history and 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. $\qquad\square$

Note that the key argument in the proof exploits the fact that 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.

5.2. Capture of Partial-Order Runs

While Theorem 5.2 is not surprising, we will now show the less obvious converse of Theorem 5.1. The fact that (by Definition 3.2) a recursive ASM $\mathcal{R}$ is a finite set of recursive ASM rules, starts its runs with a main program and uses during any run only instances of its rules implies that if $\mathcal{R}$ simulates a concurrent

ASM, this concurrent ASM must be finitely composed (as assumed in [22, Sect. 6] for the definition of partial-order runs) and must use only instances of its finitely many nd-seq components.

Theorem 5.3. 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 $\mathcal{R}_{\mathcal{C}}$ which simulates $\mathcal{C}$, i.e. such that for each concurrent run of $\mathcal{C}$ there is an equivalent recursive run of $\mathcal{R}_{\mathcal{C}}$.[41]

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 $pgm(a) = \textbf{amb } a \textbf{ in } r$, an instance of a rule r in the program base of $\mathcal{C}$. To execute the concurrent step by 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 $\textbf{amb } a \in r$ and makes its program immediately *Terminated*.

To achieve this, we refine the recursive CALL machine of Definition 3.4 to a recursive ASM $\textsc{Call}^*$ by adding to INITIALIZE the update *Terminated*($\textbf{amb } c \textbf{ in } q$) := *false*. When $\textsc{Call}^*$ is applied to $out_r \leftarrow \textsc{OneStep}_r(in_r)$, the update of *Terminated* makes the delegate *Active* so that it can make a step to execute $\textbf{amb } c \textbf{ in } \textsc{OneStep}_r$. $\textsc{OneStep}_r$ is defined to perform $\textbf{amb } caller(c) \textbf{ in } r$ and immediately terminate (by setting *Terminated* to true). For ease of exposition we add in Definition 3.4 also the update $caller(c) := \textbf{self}$, 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:

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

Note that each caller step step $\textbf{amb } a \textbf{ in } \quad out_r \leftarrow \textsc{OneStep}_r(in_r)$ in an $\mathcal{R}_{\mathcal{C}}$-run is by definition equivalent to the machine $\textbf{amb } a \textbf{ in } \textsc{Call}^*(out_r \leftarrow \textsc{OneStep}_r(in_r))$ and triggers the execution of the delegate program $\textbf{amb } c \textbf{ in } \textsc{OneStep}_r$ (where $a = caller(c)$, which triggers $\textbf{amb } c \textbf{ in amb } a \textbf{ in } r$ (by definition). Furthermore, since the innermost ambient binding counts, this machine is equivalent to the simulated $\mathcal{C}$-run step $\textbf{amb } a \textbf{ in } r$.

Thus the recursive $\mathcal{R}_{\mathcal{C}}$-run which simulates $(S_0, A_0), (S_1, A_1), \ldots$ starts by Definition 5.3 in S_0[42] with program $\textbf{amb } a_0 \textbf{ in } out_{main} \leftarrow \textsc{OneStep}_{main}(in_{main})$. Let

---

[41]Equivalence via an inverse simulation of every recursive $\mathcal{R}_{\mathcal{C}}$-run by a concurrent $\mathcal{C}$-run can be obtained 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.

[42]For the sake of notational simplicity we disregard the auxiliary locations of $\mathcal{R}_{\mathcal{C}}$.

$A_i = \{a_{i_1}, \ldots, a_{i_k}\} \subseteq$ *Agent* for some i_j and k depending on i

where $a_{i_j} = ag(m_{i_j}) \in M_{i+1} \setminus M_i$ **and** $pgm(a_{i_j}) = $ **amb** a_{i_j} **in** r_{i_j} (**forall** $1 \leq j \leq k$)

We use the same agents a_{i_j} for A_i in the $\mathcal{R_C}$-run, but with $out_{r_{i_j}} \leftarrow \text{ONESTEP}_{r_{i_j}}(in_{r_{i_j}})$ as program. 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 $\mathcal{R_C}$ 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 $\mathcal{R_C}$ run) a new agent a is created, it is made not *Waiting* (by initializing *CalledBy*$(a) := \emptyset$).

$\square$

Remark. Consider an $\mathcal{R_C}$-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 5.3 heavily depends on the prerequisite that $\mathcal{C}$ only has partial-order runs. With general concurrent runs as defined in [12] the construction would not be possible.[43] The rather strong conclusion from this is that the class of sequential recursive algorithms is already rather powerful, as it captures all attempts to capture asynchronous parallelism by a formalism that includes some form of a lockstep application of updates defined by several agents. However, true asynchronous behaviour only results, if it is possible that steps by different agents can be started as well as terminated in an independent way, which is covered by the theory of concurrency in [12].

5.3. Finite Static Concurrent ASMs with partial-order runs

In this section we consider the special case of static finite concurrent ASMs, which means by Definition 5.1 a static set of pairs $(a, pgm(a))$. These ASMs have fixed finite sets of agents and programs and a fixed association of each program with an executing agent, so that there is no rule instantiation with new agents which could be created during a run. Therefore one can define global states as the union of the component states and the functions $\sigma(I)$ associated with the po-runs yield for every finite initial segment I as value the global state 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 runs of nd-seq ASMs, as we are going to show in this section. This theorem illustrates the rather special character the axiomatic coherence condition imposes on partial-order runs.

---

[43]The other prerequisites in Theorem 5.3 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 [12], it was not intended in Gurevich's definition of partial-order runs.

Theorem 5.4. For each finite static concurrent ASM $\mathcal{C} = \{(a_j, r_j) \mid 1 \leq j \leq n\}$ with nd-seq component ASMs r_j 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.

Corollary 5.1. The partial-order runs of every Process Rewrite System [27] can be simulated by runs of a non-deterministic sequential ASM.

Proof. For the states S_i of a given concurrent run of $\mathcal{C}$ let $\sigma(M_i)$ be the state associated with an initial segment M_i of a 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 the fixed finite set of rules in $\mathcal{C}$ to execute them in parallel. Formally:

$$\mathcal{M}_\mathcal{C} = \textbf{choose } \text{ALLRULESOF}(J_1) \mid \cdots \mid \text{ALLRULESOF}(J_m)$$
$$\textbf{where}$$
$$\text{ALLRULESOF}(\{j_1, \ldots, j_k\}) =$$
$$\quad r_{j_1}$$
$$\quad \ldots$$
$$\quad r_{j_k}$$
$$\{J_1, \ldots, J_m\} = \{J' \neq \emptyset \mid J' \subseteq J\} \text{ // the non-empty subsets of } J$$
$$J = \{i \mid 1 \leq i \leq n\} \text{ // the fixed set of rule indices}$$
$$m = 2^{|J|} - 1$$

To complete the proof it suffices to show the following lemma. $\qquad\square$

Lemma 5.1. 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 the nd-seq ASM $\mathcal{M}_\mathcal{C}$ is a linearised run of $\mathcal{C}$ we proceed by induction to construct the underlying 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 set $J' = \{j_1, \ldots, j_k\} \subseteq J$. By induction we have $S_i = \sigma(M_i)$ for some initial segment of a partial-order run $(M, \leq)$. Since ALLRULESOF(J') is a parallel composition, S_{i+1} results from applying the union of update sets $\Delta_{j_l} \in \mathbf{\Delta}_{r_{j_l}}$ for $l = 1, \ldots, k$ to S_i. Each Δ_{j_l} defines a move m_{j_l} of some $ag(m_{j_l}) = a_{j_l}$, move which finishes in state S_i. We now have two cases:

(i) The moves m_{j_l} with $1 \leq l \leq k$ 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 $M_{i+1} = M_i \cup \{m_{j_1}, \ldots, m_{j_k}\}$ becomes an initial segment and $S_{i+1} = \sigma(M_i)$ holds.

(ii) If the moves m_{j_l} with $1 \leq l \leq k$ are not pairwise independent, the union of the corresponding update sets is inconsistent, hence the run terminates in state S_i.

To show the converse we proceed analogously. If we have a linearized run of states $S_i = \sigma(M_i)$ for all $i \geq 1$, then S_{i+1} results from S_i by applying in parallel all moves in $M_{i+1} \setminus M_i$. Applying a move m means to apply an update set produced by some rule r_j of $\mathcal{C}$ 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 $\Delta = \bigcup_{j \in J'} \Delta_j$ for some non-empty $J' = \{j_1, \ldots, j_k\} \subseteq J$, where each Δ_{j_l} is an update set produced by r_{j_l} (for $1 \leq l \leq k$), i.e. Δ is an update set produced by $\text{ALLRULESOF}(J')$, which implies that the linearised run $S_0, S_1, \ldots$ is a run of $\mathcal{M}_\mathcal{C}$. $\qquad\square$

6. Related Work

The behavioural theory developed in this article contributes to answer the fundamental epistemological question "What is an algorithm?". It has been inspired by Gurevich's behavioural theory of sequential algorithms [23], the ur-instance of a behavioural theory, and motivated by Moschovakis' claim that recursive algorithms, which obviously cannot be modeled 'closely and faithfully' by sequential ASMs, can be 'directly expressed' by systems of recursive equations (called 'recursive programs') [32, p.100].

In [32, p.99] Moschovakis places "the basic foundational problem of *defining algorithms...* outside the scope of this book" and treats *recursive algorithms* as "faithfully expressed" (ibid. p.101) by syntactically well defined *recursive programs* (read: systems of recursive equations) which permit to compute (partial) functions from auxiliary (in ASM terminology background) functions in whatever given structures. Besides restricting the attention to algorithms which compute (partial) functions as least fixed point of a system of equations, such a *definition* is fundamentally different from the behavioural theory approach to *capture* recursive algorithms by a class of abstract machines which can be shown to satisfy an a priori given precise, axiomatic, programming language independent characterization of recursion. Furthermore, we use a class of machines which provide a general framework to characterize besides sequential or recursive algorithms also other classes of algorithms, e.g. parallel, interactive or reflective algorithms (see below), which are deliberately left out in [32].

Nevertheless, every function which is computable by a recursive program in the sense of Moschovakis can be computed (in the standard meaning of the term) by a recursive ASM. This can be shown easily, for example by using Moschovakis' 'recursive machine' [32, Sect.2D], an abstract machine which is considered by its author as "one of the classical implementations of recursion" (ibid.p.74).[44] Alternatively one can use sequential recursive ASMs to describe the fixed point construction for systems of recursive equations. Apparently it can also be shown that vice versa, every function which is computable by a recursive ASM can be computed by a recursive program in the sense of Moschovakis (use an induction on the recursion depth of recursive ASMs, with nd-seq ASMs at the basis of the induction).

As the sequential ASM thesis shows, the notion of sequential algorithm includes a form of bounded parallelism, which is a priori defined by the algorithm and does not depend on the actual state.[45] However, parallel algorithms, e.g. for graph inversion or leader election, require unbounded parallelism. A behavioural theory of synchronous parallel algorithms has been first approached by Blass and Gurevich [7, 8], but different from the sequential thesis the theory was not accepted, not even by the ASM community despite its inherent proof that ASMs [15] capture parallel algorithms. One reason is that the axiomatic definition exploits non-logical concepts such as mailbox, display and ken, whereas the

---

[44] As pointed out in [6] and also in [10], using the set of equations of a recursive program to compute a concrete function value still requires a determination of control, i.e. in which way the recursion equations are to be applied to yield the desired result value, a feature which is considered in [32] as implementation detail.

[45] Note that by their definition, Moschovakis' recursive programs satisfy the bounded exploration postulate and their non-deterministic version [32, Sect. 2E] is carefully restricted to bounded choice.

sequential thesis only used logical concepts such as structures and sets of terms[46].

In [19] an alternative behavioural theory of synchronous parallel algorithms (aka "simplified parallel ASM thesis") was developed. It was inspired by previous research on a behavioural theory for non-deterministic database transformations [35]. Largely following the careful motivation in [7] it was first conjectured in [36] that it should be sufficient to generalise bounded exploration witnesses to sets of multiset comprehension terms[47] and to make assumptions about background domains, constructors and operations for truth values, records and finite multisets explicit[48]. The formal proof of the simplified ASM thesis in [19] requires among others an investigation in finite model theory.

At the same time another behavioural theory of parallel algorithms was developed in [16], which is independent from the simplified parallel ASM thesis[49], but refers also to previous work by Blass and Gurevich. It is debatable, whether the criticism of the defining postulates by Blass and Gurevich also applies to this work; a thorough comparison with the simplified parallel ASM thesis has not yet been conducted.

There have been many attempts to capture asynchronous parallelism, as marked in theories of concurrency as well as distribution (see [26] for a collection of many distributed or concurrent algorithms). Commonly known approaches are among others the actor model [3], Process Algebras [4], Petri nets [5], high-level Petri nets [20], and trace theory [28]. Gurevich's axiomatic definition of partial-order runs [22] tries to reduce the problem to families of sequential algorithms, but the theory is too strict. As shown in [12] it is easy to find concurrent algorithms that satisfy sequential consistency [25], but their runs are not partial-order runs. One problem is that the requirements for partial-order runs always lead to linearisability.

The lack of a convincing definition of asynchronous parallel algorithms was overcome by the work on concurrent algorithms in [12], in which a concurrent algorithm is defined by a family of agents, each equipped with a sequential algorithm, possibly with shared locations. While each individual sequential algorithm in the family is defined by the postulates for sequential algorithms[50], the family as a whole is subject to a concurrency postulate requiring that in a concurrent run, a successor state of the global state of the concurrent algorithm results from simultaneously applying update sets of finitely many agents that have been built on some previous states (not necessarily the current one). The theory shows that concurrent algorithms are captured by concurrent ASMs. Given the fact that in concurrent algorithms, in particular in case of distribution, message passing between agents is more common than shared locations, it has further been shown in [13] that message passing can be captured by regarding mailboxes as shared locations, which leads to communicating concurrent ASMs capturing concurrent algorithms with message passing. In [17] it has been shown how the popular bulk synchronous parallel bridging model can be captured by a specialised behavioural theory that builds on top of the concurrent ASM thesis in [12].

Recently, there is an increased interest in distributed adaptive systems. Adaptivity refers to the ability of an algorithm to modify itself, which is known as linguistic reflection. A behavioural theory of

---

[46]Even the background, that is left implicit in the sequential thesis, only refers to truth values and operations on them.

[47]The rationale behind this conjecture is that in a particular state the multiset comprehension terms give rise to multisets, and selecting one value out each of these multisets defines the proclets used by Blass and Gurevich.

[48]The latter aspect was already part of the thesis by Blass and Gurevich.

[49]Apparently, authors of [19] and [16] seemed not to be aware of each others' research.

[50]A remark in [12] states that the restriction to sequential algorithms is not really needed. An extension to concurrent algorithms covering families of parallel algorithms is handled in [34].

reflective sequential algorithms has been developed in [33].[51] Again the key aspect is the generalisation of bounded exploration witnesses, which for reflective algorithms comprise terms that can be evaluated to terms and these to values in the base set, so coincidence after double evaluation is required for the equality of update sets in states. The integration of the behavioural theories for parallelism, concurrency and reflection has been sketched in [34], but a more detailed presentation of the combined theory still has to be written up.

7. Conclusion

The main contribution of this article is a behavioural theory of sequential recursive algorithms, providing a) a purely logical definition of this notion, which is independent from any particular abstract machine or programming model, b) a natural extension of nd-seq ASMs to recursive ASMs, and c) a proof that recursive ASMs capture sequential recursive algorithms. The resulting recursive ASM thesis shows (together with the sequential and the concurrent ASM thesis) that the class of recursive algorithms is strictly larger and more expressive than the class of nd-seq algorithms and strictly smaller and less expressive than the class of concurrent algorithms.

As an application of this theory we add to the observation in [22]—namely that recursive algorithms give rise to partial-order runs—a proof that conversely, every finitely composed concurrent algorithm with only partial-order runs is equivalent to a recursive algorithm. This corrects the criticism formulated in [10] that the answer given by Blass and Gurevich in [6] is an "overkill", as partial-order runs capture only a restricted concept of concurrency. On the other hand, it underlines also the need for a much more general theory of concurrent algorithms, which is provided by the behavioural theory of concurrent algorithms [12].

The debate about "What is an algorithm?" is not yet finished, as only an integration of all partial behavioural theories of sequential, recursive, parallel, concurrent, reflective, etc. algorithms[52] will provide a final answer to the question. A comprehensive definition of the notion of algorithm will require all the known particular classes of algorithms to be integrated in such a way that the specific subclasses arise as special cases of the general definition. However, given that in all the behavioural theories we mentioned above the postulates always concern sequential or branching time, abstract state, background and bounded exploration, the perspective of such an integration looks promising. We invite the reader to contribute to this endeavor.

References

[1] J.-R. Abrial. *The B-book - Assigning programs to meanings*. Cambridge University Press, 1996.

[2] J.-R. Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[3] G. Agha. *A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass., 1986.

[4] J. Bergstra, A. Ponse, and S. Smolka. *Handbook of Process Algebra*. Elsevierl, 2001.

---

[51]A preliminary version of this theory appeared in [18].

[52]This list is not yet complete, as in most of the related work mentioned above the aspect of non-determinism as well as randomness is not yet included. However, non-determinism is covered in ASMs [15] and is crucial for their applications in system design and analysis.

[5] E. Best. *Semantics of sequential and parallel programs*. Prentice Hall, 1996.

[6] A. Blass and Y. Gurevich. Algorithms vs. machines. *Bulletin of the EATCS*, 77:96–119, 2002.

[7] A. Blass and Y. Gurevich. Abstract State Machines capture parallel algorithms. *ACM Trans. Computational Logic*, 4(4):578–651, 2003.

[8] A. Blass and Y. Gurevich. Abstract State Machines capture parallel algorithms: Correction and extension. *ACM Trans. Comp. Logic*, 9(3), 2008.

[9] E. Börger. *Computability, Complexity, Logic*, volume 128 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1989.

[10] 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.

[11] E. Börger and A. Raschke. *Modeling Companion for Software Practitioners*. Springer, 2018.

[12] E. Börger and K.-D. Schewe. Concurrent Abstract State Machines. *Acta Informatica*, 53(5):469–492, 2016.

[13] E. Börger and K.-D. Schewe. Communication in Abstract State Machines. *J. Univ. Comp. Sci.*, 23(2):129–145, 2017.

[14] E. Börger and K.-D. Schewe. A characterization of distributed ASMs with partial-order runs. In A. Raschke, D. Mery, and F. Houdek, editors, *Rigorous State-Based Methods (Proc. ABZ 2020)*, volume 12071 of *LNCS*, pages 79–92. Springer, 2020.

[15] E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.

[16] N. Dershowitz and E. Falkovich-Derzhavetz. On the parallel computation thesis. *Logic Journal of the IGPL*, 24(3):346–374, 2016.

[17] F. Ferrarotti, S. González, and K.-D. Schewe. BSP abstract state machines capture bulk synchronous parallel computations. *Sci. Comput. Program.*, 184, 2019.

[18] F. Ferrarotti, K.-D. Schewe, and L. Tec. A behavioural theory for reflective sequential algorithms. In A. K. Petrenko and A. Voronkov, editors, *Perspectives of System Informatics – 11th International Andrei P. Ershov Informatics Conference (PSI 2017)*, volume 10742 of *LNCS*, pages 117–131. Springer, 2017.

[19] F. Ferrarotti, K.-D. Schewe, L. Tec, and Q. Wang. A new thesis concerning synchronised parallel computing – simplified parallel ASM thesis. *Theor. Comp. Sci.*, 649:25–53, 2016.

[20] H. J. Genrich and K. Lautenbach. System modelling with high-level Petri nets. *Theoretical Computer Science*, 13:109–136, 1981.

[21] Y. Gurevich. A new thesis. *Abstracts, American Mathematical Society*, 6(4):317, August 1985.

[22] Y. Gurevich. Evolving algebras 1993: Lipari Guide. In E.Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[23] Y. Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comp. Logic*, 1(1):77–111, 2000.

[24] Y. Gurevich and M. Spielmann. Recursive Abstract State Machines. *J. UCS*, 3(4):233–246, 1997.

[25] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[26] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[27] R. Mayr. Process Rewrite Systems. *Information and Computation*, 156:264–286, 1999.

[28] A. Mazurkiewicz. Trace theory. volume 255 of *LNCS*, pages 279–324. Springer, 1987.

[29] Y. N. Moschovakis. Abstract recursion as a foundation of the theory of algorithms. In E. Börger, W. Oberschelp, M. Richter, B. Schinzel, and W.Thomas, editors, *Computation and Proof Theory*, volume 1104 of *Lecture Notes in Mathematics*, page 289364. Springer, 1984.

[30] Y. N. Moschovakis. On founding the theory of algorithms. In H. G. Dales and G. Oliveri, editors, *Truth in mathematics*, pages 71–104. Clarendon Press, 1998.

[31] Y. N. Moschovakis. What is a algorithm? In B. Engquist and W. Schmid, editors, *Mathematics Unlimited – 2001 and Beyond*, pages 919–936. Springer, 2001.

[32] Y. N. Moschovakis. *Abstract Recursion and Intrinsic Complexity*. Cambridge University Press, 2019. Lecture Notes in Logic.

[33] K.-D. Schewe and F. Ferrarotti. Behavioural theory of reflective algorithms I: Reflective sequential algorithms, 2018. submitted for publication, available at http://arxiv.org/abs/2001.01873.

[34] K.-D. Schewe, F. Ferrarotti, L. Tec, Q. Wang, and W. An. Evolving concurrent systems – behavioural theory and logic. In *Proceedings of the Australasian Computer Science Week Multiconference (ACSW 2017)*, pages 77:1–77:10. ACM, 2017.

[35] K.-D. Schewe and Q. Wang. A customised ASM thesis for database transformations. *Acta Cybernetica*, 19(4):765–805, 2010.

[36] K.-D. Schewe and Q. Wang. A simplified parallel ASM thesis. In J. Derrick et al., editors, *Abstract State Machines, Alloy, B, VDM, and Z (ABZ 2012)*, volume 7316 of *LNCS*, pages 341–344. Springer, 2012.

[37] M. Y. Vardi. What is an algorithm? *Commun. ACM*, 55(3):5, 2012.