Distributed Abstract State Machines and Their Expressive Power

Andreas Glausch, Wolfgang Reisig

January 30, 2006

Gurevich's *sequential* Abstract State Machines (ASMs) are taken as a basis for the construction of *distributed* ASMs as *sets* of sequential ASMs.

A theorem on the expressive power of distributed ASM is proven in analogy to Gurevich's classical theorem on the expressive power of sequential ASM.

1 Introduction

Abstract State Machines have been introduced as a "computation model that is more powerful and more universal than the standard computation models" by Yuri Gurevich in 1985 [11]. This is achieved by adopting classical concepts from logics and universal algebra, and their conservative extension to describe sequential steps. A number of variants of ASMs evolved over time, in particular parallel, distributed, and interactive versions [6, 2, 1]. In addition to theoretical considerations, ASMs have proven their practical benefits for the specification and analysis of real-world systems [4, 3].

Gurevich introduced in [7] the class of *sequential small-step algorithms* by only a few intuitive and amazingly liberal requirements, and proved every sequential small-step algorithm to be represented by a sequential small-step ASM. "Small-step" means intuitively that the amount of change is bounded for all steps of the algorithm. [10] reexamines Gurevich's theorem, and provides some further explanations and examples. Later, the results for sequential ASMs have been extended by Blass and Gurevich to parallel and interactive versions of ASMs [2, 1].

In the "Lipari Guide" [6], Gurevich describes the central constructs of distributed ASMs. In particular, a single run of a distributed ASM is defined as a partially ordered set of actions of sequential ASMs, called *moves* in [6]. These definitions have been examined in more detail in [12], and were applied for the specification and verification of several distributed algorithms [12, 5, 8].

This paper investigates *distributed small-step ASMs.* It translates, to some extent, Gurevichs characterization of sequential small-step algorithms to *distributed* small step algorithms. It turns out that this can be achieved indeed, even as a conservative extension of the sequential case.

This paper is organized as follows: The next section reexamines the established fundamentals of Abstract State Machines and identifies *stores* as a basic concept. Calling them "updates", Gurevich employed stores in [6] already to describe the effect of assignment statements. We furthermore use them here to describe entire states.

Section 3 defines the syntax and the semantics of sequential ASMs as given in [6]. This prepares the notion of distributed ASMs, as given in Sec. 4.

Finally, Sec. 5 characterizes distributed ASMs in analogy to Gurevich's characterization of sequential ASMs. The decisive aspect is the requirement corresponding to *bounded exploration* as introduced in [7]: Actions of distributed ASMs are bounded in size. The proof of the main theorem and of a couple of preparing lemmata requires a bit of formal arguments and is confined to Sec. 6.

2 States, stores, and steps

In this section we motivate and exemplify the fundamental notions of ASMs. The most important and new aspect of this section is the decomposition of states into *stores*, presented in Sec. 2.3.

2.1 States are structures

In the computation model of ASMs, a *system state* is conceived as a *structure* S, i.e. a set U (the *universe* of S), and finitely many functions $\varphi_i : U^{n_i} \to U$ $(i = 1, \dots, k)$, each with its arity $n_i \geq 0$. From the perspective of computer science, each function φ_i may be conceived as an n_i-dimensional array (albeit, in general, with an infinite index set). The universe of a given structure S is denoted by $U(S)$.

The universe U may be infinite, even uncountable. We assume no particular properties of the functions φ_i. In particular, we do not assume any means to syntactically represent the functions of a state.

2.2 Structures have signatures

As usual, a *signature* Σ is used to address the functions of a structure: Σ comprises finitely many symbols $f_1, \dots, f_k$, each f_i with its arity n_i $(i = 1, \dots, k)$. A structure S is a Σ-*structure* if its functions φ_i correspond bijectively to symbols f_i of the same arity. In this case, φ_i is usually written as

$$f_{iS}$$

and called the *interpretation* of f_i in S. In case f_i is a symbol with arity $n_i = 0$, f_i denotes a single element $f_{iS} \in U(S)$, i.e. f_i denotes a constant.

Hence, in the sequel,

$$\textit{a state is a } \Sigma\textit{-structure.} \tag{1}$$

As an example, assume a state where we want to apply a *bisection algorithm* to find a zero x_0 of a continuous function f, i.e.

$$\text{find } x_0 \text{ such that } f(x_0) = 0.$$

The bisection algorithm works as follows: Given a continuous function f, start with two real numbers a, b such that $f(a)$ and $f(b)$ are both different from 0 and have different leading signs. Then compute the arithmetic mean m of a and b and check the value of $f(m)$: If $f(m) = 0$, we are done. Otherwise, check the leading sign of $f(m)$: if $f(m)$ and $f(a)$ have the same leading sign, set a to m, otherwise set b to m. This step is iterated until the distance of a and b drops below a bound ϵ. Figure 1 represents two steps of the bisection algorithm.

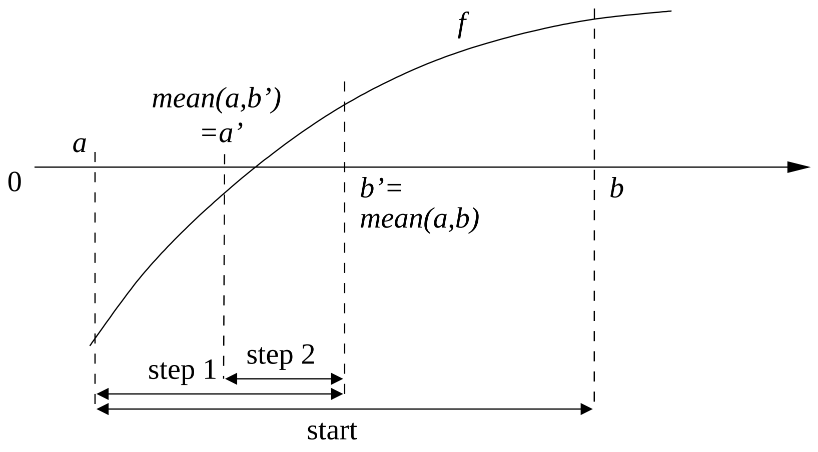

Figure 1: Two steps of the bisection algorithm.

We model the bisection algorithm by help of symbols to represent all entities required by the algorithm: The symbols `a` and `b` represent the values of a and b, `f` represents the function f, the symbol `0` represents the real number 0, the symbol `mean` represents a function to compute arithmetic mean, and the symbol `eqsign` represents a predicate checking whether two real numbers have the same leading signs. We furthermore need a symbol `stop` to represent the termination condition and a symbol `result` to represent the algorithm's result.

This yields the signature $\Sigma = (\mathtt{f}, \mathtt{a}, \mathtt{b}, \mathtt{0}, \mathtt{result}, \mathtt{true}, \mathtt{mean}, \mathtt{eqsign}, \mathtt{stop})$ with arities $(1, 0, 0, 0, 0, 0, 2, 2, 2)$, respectively. The initial state is a Σ-structure S with

- $\mathtt{f}_S$ any continuous function over $\mathbb{R}$,
- $\mathtt{a}_S, \mathtt{b}_S$ any real number such that $\mathtt{f}_S(\mathtt{a}_S)$ and $\mathtt{f}_S(\mathtt{b}_S)$ are not 0 and have different leading signs,
- $\mathtt{0}_S$ the real number 0,
- $\mathtt{mean}_S$ a function to compute the arithmetic mean of two real numbers,
- $\mathtt{true}_S$ the truth value *true*,
- $\mathtt{eqsign}_S$ a binary function on real numbers such that $\mathtt{eqsign}_S(x, y) = \mathtt{true}_S$ iff x and y have the same leading sign,

– $\texttt{stop}_S$ a binary function on real numbers such that $\texttt{stop}_S(x,y) = \texttt{true}_S$ iff $|x-y| \leq \epsilon$ for a fixed positive real number ϵ.

Later we will show how the bisection algorithm can be formalized by a sequential ASM. The decisive aspect of the ASM approach is already visible: An ASM may do with *any* functions and objects, whereas conventional programs are based on a bit level representation of data structures.

2.3 States consist of stores

Throughout this paper it turns out useful not to consider a state as a monolithic entity but rather as a collection of *stores*. A store has a *location* ("Where can something be stored?") and a *value* ("What is stored?").

A signature Σ and a set U together identify a set of locations. Each location is formed

$$l = (f, \underline{u})$$

with $f \in \Sigma$ and $\underline{u} \in U^n$, where n is the arity of f. In case f is a 0-ary function symbol, the location of f is written $(f, \_)$, with $\_$ denoting the empty tuple in U^0.

Each Σ-structure S with a universe U then defines a set $\widetilde{S}$ of stores. Each store s of $\widetilde{S}$ is shaped

$$s = (f, \underline{u}, u_0) \tag{2}$$

consisting of the location $(f, \underline{u})$ and the value $u_0 = f_S(\underline{u})$.

Hence, to each location l, the set $\widetilde{S}$ contains exactly one store, shaped (l, u). This way, $\widetilde{S}$ identifies S: To each function f_S and each argument $\underline{u}$ of f_S there exists a unique store $(f, \underline{u}, u_0)$, with $u_0 = f_S(\underline{u})$.

As an obvious yet important property of states we get from the above construction together with (1):

$$\text{States with equal universes have equal locations.} \tag{3}$$

Under the name of *updates*, Gurevich has identified stores in [6] already. He used them to describe the *change* of states as caused by steps. We extend their use to describe entire states.

2.4 Steps consist of "sufficiently similar" states

As usual for conventional models of computation, in the ASM model, too, a *step* is a pair (S, S') of states. Consequently, a (sequential) run $S_0S_1S_2\ldots$ is a (possibly infinite) sequence of steps (S_{i-1}, S_i) $(i = 1, 2, \ldots)$.

The two states of a step (S, S') are usually not entirely different. S evolves to S' by a finite amount of update [1]. This requires means to describe what remains equal and what is updated. To this end, the states S and S' must be "sufficiently similar". In

[1] *Large-step* versions of ASMs allow infinite amount of update, too. But we stick to the elementary version of small-step ASMs here.

particular, S and S' must be structures of the same signature (later on we will see that *all* states of an ASM program are structures of the same signature). Furthermore, S and S' are assumed to have the same universe. Hence, by (3), they even have the same locations. Consequently, the step from S to S' can be characterized by a set $\Delta^{\text{old}} \subseteq \widetilde{S}$ and a set Δ^{new} of stores where the locations of Δ^{old} and Δ^{new} coincide , such that

$$\widetilde{S'} = (\widetilde{S} \setminus \Delta^{\text{old}}) \cup \Delta^{\text{new}}. \tag{4}$$

As an example, a step of the bisection algorithm mentioned above updates the stores with the locations $(\texttt{a}, \_)$, $(\texttt{b}, \_)$ and $(\texttt{result}, \_)$, and retains all other stores.

3 Assignment statements and ASM programs

According to (4), the problem of defining a step reduces to the problem of characterizing a set of stores. In this section we will show how such sets can be described by the help of *ASM programs*.

3.1 Assignment statements define steps with one update

As usual, a signature yields *terms*: each 0-ary symbol is a term, and for an n-ary symbol f and given terms $t_1, \ldots, t_n$, the symbol sequence

$$f(t_1, \ldots, t_n)$$

is a term, too. Such terms are interpreted by a Σ-structure S in the usual way: For each 0-ary symbol x, the element $x_S \in U(S)$ denotes the *interpretation* of x in S. For $n \geq 1$, the *interpretation* of a term $f(t_1, \ldots, t_n)$ in S is inductively defined as

$$f(t_1, \ldots, t_n)_S \;\; =_{\text{def}} \;\; f_S(t_{1S}, \ldots, t_{nS}).$$

Terms are used to form *assignment statements*. A most simple example is the assignment statement

$$\alpha : \quad x := f(x), \tag{5}$$

where x and f are 0-ary and unary symbols in Σ, respectively. Applied to a state S, (5) updates the store

$$\alpha_S^{\text{old}} \;\; =_{\text{def}} \;\; (x, \_, x_S)$$

by the store

$$\alpha_S^{\text{new}} \;\; =_{\text{def}} \;\; (x, \_, f(x)_S).$$

Formulated more conventionally, the application of (5) yields a state S' with

$$x_{S'} = f(x)_S.$$

A slightly more involved example is the statement

$$\alpha : \quad f(x) := g(y) \tag{6}$$

where x, y are 0-ary, and f, g are unary symbols. Applied to a state S, (6) updates the store

$$\alpha_S^{\text{old}} =_{\text{def}} (f, x_S, f(x)_S)$$

by the store

$$\alpha_S^{\text{new}} =_{\text{def}} (f, x_S, g(y)_S).$$

Formulated more conventionally, this update yields a state S' with

$$f_{S'}(x_S) = g(y)_S.$$ [2]

The general form of an assignment statement α is

$$\alpha : \quad t := t' \tag{7}$$

with terms t, t' over Σ. The term t specifies the location to update, and t' specifies the new value of this location in the next state. Formally, for $t = f(t_1, \dots, t_n)$ and a state S,

$$\text{loc}_S(t) =_{\text{def}} (f, (t_{1S}, \dots, t_{nS})) \tag{8}$$

denotes the *location of* t *in* S. Applied to state S, (7) then updates the store

$$\alpha_S^{\text{old}} =_{\text{def}} (f, \text{loc}_S(t), t_S) \tag{9}$$

by the store

$$\alpha_S^{\text{new}} =_{\text{def}} (f, \text{loc}_S(t), t'_S). \tag{10}$$

Formulated more conventionally, this update yields a state S' with

$$f_{S'}(t_{1S}, \dots, t_{nS}) = t'_S.$$

α_S^{old} and α_S^{new} then define a step (S, S'), where

$$\widetilde{S'} =_{\text{def}} (\widetilde{S} \setminus \{\alpha_S^{\text{old}}\}) \cup \{\alpha_S^{\text{new}}\}. \tag{11}$$

3.2 Conditional assignment statements define steps conditionally

A *conditional assignment statement* extends an assignment statement such as (7) by a condition described as a *boolean expression*. A boolean expression is built from terms: For any two terms t_1 and t_2, $t_1 = t_2$ is a boolean expression. Such an expression is *fulfilled* in a state S iff $t_{1S} = t_{2S}$, i.e. if t_1 and t_2 are interpreted equally in S. Boolean expressions can be combined to new expressions by using the usual boolean operations, such as $\wedge$ and $\neg$: For any given boolean expressions β_1 and β_2, $\beta_1 \wedge \beta_2$ and $\neg\beta_1$ are boolean expressions, too. As usual, $\neg\beta_1$ is fulfilled in a state S iff β is *not* fulfilled in S and $\beta_1 \wedge \beta_2$ is fulfilled in S iff β_1 *and* β_2 are fulfilled in S. $S \models \beta$ is used as an abbreviation for "β is fulfilled in state S".

[2]Notice this equation does not read $f_{S'}(x_{S'}) = \dots$

A *conditional assignment statement* γ is shaped

$$\gamma : \quad \texttt{if } \beta \texttt{ then } \alpha, \tag{12}$$

where β is a boolean expression and α is an assignment statement. β is called the *guard* or the *condition* of γ. Applied to a state S where β is not fulfilled, (12) yields no update at all and S is a final state, i.e. there is no step (S, S'). We differ from the classical semantics of sequential ASMs at this point: [6] generates the step (S, S) in this case. We will justify our proposal when it comes to distributed ASMs.

Otherwise, if β is fulfilled in S, (12) updates the store α_S^{old} by the store α_S^{new}, in accordance with (9), (10) and (11). Hence, for γ as in (12),

$$\gamma_S^{\text{old}} =_{\text{def}} \alpha_S^{\text{old}}, \tag{13}$$

$$\gamma_S^{\text{new}} =_{\text{def}} \alpha_S^{\text{new}}, \tag{14}$$

$$\widetilde{S}' =_{\text{def}} (\widetilde{S} \setminus \{\gamma_S^{\text{old}}\}) \cup \{\gamma_S^{\text{new}}\} \tag{15}$$

in case β is fulfilled in S.

For convenience, an assignment statement $t := t'$ can be considered as a conditional statement whose guard is fulfilled at every state.

3.3 ASMs define steps with many updates

An *ASM program* Γ is a nonempty, finite set of conditional assignment statements. Executing Γ in a state S means to execute all statements in Γ at once. So, we generalize the notions of (13) and (14) to sets of statements: For a state S, let

$$\Gamma_S^{\text{old}} =_{\text{def}} \{ \alpha_S^{\text{old}} \mid (\texttt{if } \beta \texttt{ then } \alpha) \in \Gamma \text{ and } S \models \beta \}, \tag{16}$$

$$\Gamma_S^{\text{new}} =_{\text{def}} \{ \alpha_S^{\text{new}} \mid (\texttt{if } \beta \texttt{ then } \alpha) \in \Gamma \text{ and } S \models \beta \}. \tag{17}$$

Γ_S^{new} may be *inconsistent*, i.e. may include two stores with equal locations but different values. For example, Γ may include the assignment statements

$$f(x) := u \quad \text{and} \quad f(y) := v$$

with $x_S = y_S$ and $u_S \neq v_S$. An inconsistent set of stores cannot update S, thus for S there is no step (S, S') and S is a final state.

In case the conditions of all statements in Γ fail in S, we take S as a final state. This decision deviates again from [6], where *stuttering steps* (S, S) are suggested in this case.

In all other cases, i.e. if Γ_S^{new} is consistent and at least one condition in Γ is fulfilled, Γ defines S' by:

$$\widetilde{S}' =_{\text{def}} (\widetilde{S} \setminus \Gamma_S^{\text{old}}) \cup \Gamma_S^{\text{new}}. \tag{18}$$

Then (S, S') is a *step* of Γ.

The set of all steps of an ASM program Γ then constitutes a *sequential ASM.* The term "sequential" comes as a surprise, as the statements of Γ are executed in parallel. The reason for this is merely traditional, and contrasts with *parallel* ASMs not considered here, as well as with *distributed* ASMs, to be considered in Sec. 4.

To make life easier, we introduce the following syntactic simplification: For assignment statements $\alpha_1, \ldots, \alpha_n$,

$$\texttt{if}\ \beta\ \texttt{then}\ \{\alpha_1, \ldots, \alpha_n\} \tag{19}$$

stands for $\{\texttt{if}\ \beta\ \texttt{then}\ \alpha_1, \ldots, \texttt{if}\ \beta\ \texttt{then}\ \alpha_n\}$.

3.4 ASMs represent algorithms

We are now ready to formulate the bisection algorithm introduced in Sec. 2.2. Let

```
Γ := { if stop(a,b)=true then result:=a,
       if ¬(stop(a,b)=true) ∧ f(mean(a,b))=0 then
           result:=mean(a,b),
       if ¬(stop(a,b)=true) ∧ ¬(f(mean(a,b))=0)                    (20)
           ∧ eqsign(f(a),f(mean(a,b)))=true then a:=mean(a,b),
       if ¬(stop(a,b)=true) ∧ ¬(f(mean(a,b))=0)
           ∧ eqsign(f(b),f(mean(a,b)))=true then b:=mean(a,b) }.
```

Notice that the algorithm Γ cannot be conceived as a conventional program: No program would process *any* real numbers and *any* continuous function. Each computation of Γ in fact approximates a zero of a continuous function f in the interval (a, b). (20) is indeed a formalization of the informal description of the bisection algorithm in Sec. 2.2.

As a further example consider a system composed of four components:

`prod`: a *producer* to produce items,
`send`: a *sender* to send produced items to a buffer,
`rec` : a *receiver* to take items from the buffer,
`cons`: a *consumer* to consume items provided by the receiver.

We base the model of this system onto a signature including the 0-ary symbols `x`, `y` and `buffer`. Their value may represent items to be processed by the system. Furthermore, the value of `x` and of `y` may be undefined (represented by `x_undef` and `y_undef`, respectively), and the buffer may be empty (represented by `b_empty`).

The components interact as follows: In case the value of `x` is undefined, the value of `item` is assigned to `x` (by `prod`), then forwarded to the empty buffer (by `send`), removed from the buffer and assigned to `y` (by `rec`), and finally consumed (by `cons`).

Applied to an initial state S_0 with $\texttt{x}_{S_0} = \texttt{x\_undef}_{S_0}$, $\texttt{y}_{S_0} = \texttt{y\_undef}_{S_0}$ and $\texttt{buffer}_{S_0} = \texttt{b\_empty}_{S_0}$, the following components define a sequential ASM program with the described behaviour:

```
prod  =def  { if x=x_undef then x := item },

send  =def  if ¬(x=x_undef) ∧ buffer=b_empty then
                { buffer := x, x := x_undef },
rec  =def  if ¬(buffer=b_empty) ∧ y=y_undef then
              { y := buffer, buffer := b_empty },

cons  =def  { if ¬(y=y_undef) then y := y_undef }.
```

Then

$$\Gamma = \texttt{prod} \cup \texttt{send} \cup \texttt{rec} \cup \texttt{cons} \tag{21}$$

is the required sequential ASM. Its behaviour is:

$$S_0 \xrightarrow{\texttt{prod}} S_1 \xrightarrow{\texttt{send}} S_2 \xrightarrow[\texttt{rec}]{\texttt{prod}} S_3 \xrightarrow[\texttt{cons}]{\texttt{send}} S_4 \xrightarrow[\texttt{rec}]{\texttt{prod}} S_5 \quad \ldots \tag{22}$$

where each step is inscribed by the components with true guards. For reasons that will become clear later on, we emphasize the order of occurrences of the components, thus writing (22) as

$$\begin{tikzcd}
 & & \texttt{prod} \arrow[r] \arrow[ddr] & \texttt{send} \arrow[r] \arrow[ddr] & \texttt{prod} & \\
\texttt{prod} \arrow[r] & \texttt{send} \arrow[ur] \arrow[dr] & & & & \cdots \\
 & & \texttt{rec} \arrow[r] \arrow[uur] & \texttt{cons} \arrow[r] \arrow[uur] & \texttt{rec} &
\end{tikzcd} \tag{23}$$

4 Distributed ASMs

In this section we will introduce *distributed* ASMs as *sets* of ASM programs. The semantics of a distributed ASM will be defined by help of *actions*, which are then used to construct *distributed runs*.

4.1 A distributed ASM is a set of ASM programs

Two ASM programs Γ_1 and Γ_2 may involve disjoint stores in a state S. Then nothing prevents Γ_1 and Γ_2 to be concurrently executed.

This gives rise to the idea of a distributed version of ASMs: A *distributed ASM* is just a nonempty, finite set of ASM programs, all over the same signature, Σ. These programs are then called *components* of the distributed ASM, and every Σ-structure forms a *state* of the distributed ASM. The components may be executed concurrently in case they involve stores with separate locations.

As an example, consider the producer/consumer system of Sec. 3.4:

$$D = \{\texttt{prod}, \texttt{send}, \texttt{rec}, \texttt{cons}\} \tag{24}$$

is a distributed ASM. Notice that D differs decisively from the sequential ASM in (21): A sequential ASM is a single set of conditional assignment statements, while a distributed ASM is a family of sets of conditional assignment statements.

This implies the notion of *distributed run*, to be considered in the sequel: A distributed run of a distributed ASM is a partially ordered set of occurrences of the component programs. Unordered occurrences represent concurrent execution of the corresponding component programs.

4.2 ASM programs describe actions

To formalize the above idea, we have to specify the *stores involved* when an ASM program Γ is applied to a state S, i.e. when Γ_S^{new} is computed. The stores Γ_S^{old} as discussed in (16) and (17) are certainly involved. Furthermore, all terms and subterms occuring in the assignment statements and conditions of Γ are involved. So, for a term $t = f(t_1, \dots, t_n)$ over Σ and a Σ-structure S, we define the set of involved stores $\text{inv}_S(t)$, inductively by

$$\text{inv}_S(t) \;\; =_{\text{def}} \;\; \text{inv}_S(t_1) \cup \dots \cup \text{inv}_S(t_n) \cup \{(\text{loc}_S(t), t_S)\}. \tag{25}$$

For a boolean expression β, let T^β denote the set of terms occuring in β. Then the involved stores of β are the involved stores of all terms in T^β:

$$\text{inv}_S(\beta) \;\; =_{\text{def}} \;\; \bigcup_{t \,\in\, T^\beta} \text{inv}_S(t). \tag{26}$$

For an assignment statement

$$\alpha : \quad t := t'$$

and a state S we define

$$\text{inv}_S(\alpha) \;\; =_{\text{def}} \;\; \text{inv}_S(t) \cup \text{inv}_S(t'). \tag{27}$$

For a conditional assignment statement

$$\gamma : \quad \texttt{if } \beta \texttt{ then } \alpha$$

and a state S we define

$$\text{inv}_S(\gamma) \;\; =_{\text{def}} \;\; \text{inv}_S(\beta) \cup \text{inv}_S(\alpha). \tag{28}$$

In case β is fulfilled in S, apparently holds

$$\gamma_S^{\text{old}} \in \text{inv}_S(\gamma). \tag{29}$$

$\text{inv}_S(\gamma)$ covers all ressources needed to execute γ, i.e. to compute the truth value of the guard β and to compute γ_S^{new} in case β is fulfilled. More precisely, if β is fulfilled,

$$\text{inv}'_S(\gamma) \;\; =_{\text{def}} \;\; (\text{inv}_S(\gamma) \setminus \{\gamma_S^{\text{old}}\}) \cup \{\gamma_S^{\text{new}}\} \tag{30}$$

includes all involved stores after the execution of γ.

For an ASM program Γ we generalize the above notions by

$$\text{inv}_S(\Gamma) \;\; =_{\text{def}} \;\; \bigcup_{\gamma \in \Gamma} \text{inv}_S(\gamma). \tag{31}$$

According to (16) and (29), apparently holds

$$\Gamma_S^{\text{old}} \subseteq \text{inv}_S(\Gamma). \tag{32}$$

Hence, it makes sense to define

$$\mathrm{inv}'_S(\Gamma) \;=_{\mathrm{def}}\; (\mathrm{inv}_S(\Gamma) \setminus \Gamma_S^{\mathrm{old}}) \cup \Gamma_S^{\mathrm{new}}. \tag{33}$$

Summing up, $\mathrm{inv}_S(\Gamma)$ contains all involved stores *before* the execution of Γ, and $\mathrm{inv}'_S(\Gamma)$ contains all involved stores *after* the execution of Γ. The locations of the stores in $\mathrm{inv}_S(\Gamma)$ coincide with the locations of the stores in $\mathrm{inv}'_S(\Gamma)$.

$\mathrm{inv}_S(\Gamma)$ is in fact all one needs to compute $\mathrm{inv}'_S(\Gamma)$, as the following lemma shows:

Lemma 1. *Let Γ be a ASM program and let S, R be states of Γ such that $\mathrm{inv}_S(\Gamma) = \mathrm{inv}_R(\Gamma)$. Then $\mathrm{inv}'_S(\Gamma) = \mathrm{inv}'_R(\Gamma)$.*

The proof of this lemma is postponed to Sec. 6.

We are now ready to define the fundamental notion of an *action*: Intuitively, an action is an ASM program Γ together with a set of resources needed to execute Γ, and the result of executing Γ. Technically, let S be a state such that at least one condition of a statement in Γ is fulfilled and $\mathrm{inv}'_S(\Gamma)$ is consistent. Then the *action of Γ in state S* is:

$$\Gamma_S \;=_{\mathrm{def}}\; (\Gamma, \mathrm{inv}_S(\Gamma), \mathrm{inv}'_S(\Gamma)). \tag{34}$$

Thus an ASM program performs an action in a state S, if at least one of its assignments is executed and the assignments yield a consistent set of stores. We denote the set of all actions of Γ by $\mathrm{act}(\Gamma)$. Notice that this definition fits well to the definition of steps of an ASM program Γ in Sec. 3.3: Γ can perform an action Γ_S in a state S if and only if there is a step (S, S') of Γ.

Steps and actions of an ASM program Γ are connected even more tightly: The steps of Γ can be completely described by the actions of Γ, as the following lemma states:

Lemma 2. *Let Γ be an ASM program and S, S' be states of Γ. Then (S, S') is a step of Γ iff there is an action $(\Gamma, A, B) \in \mathrm{act}(\Gamma)$ with $A \subseteq \widetilde{S}$ and*

$$\widetilde{S'} = (\widetilde{S} \setminus A) \cup B.$$

The proof of this lemma is postponed to Sec. 6.

The set of actions of a *distributed* ASM $D = \{\Gamma_1, \ldots, \Gamma_n\}$ is then defined by

$$\mathrm{act}(D) \;=_{\mathrm{def}}\; \mathrm{act}(\Gamma_1) \cup \cdots \cup \mathrm{act}(\Gamma_n), \tag{35}$$

i.e. the actions of a distributed ASM are constituted simply by the actions of its components.

4.3 Actions have nice properties

Though almost trivial, it is worth noting that for an action (Γ, A, B), the sets A and B are finite and the locations of the stores in A coincide with the locations of the stores in B.

Of course, different states S_1, S_2 yield in general different sets $\mathrm{inv}_{S_1}(\Gamma)$ and $\mathrm{inv}_{S_2}(\Gamma)$ of involved stores. Even more, the locations of both sets of stores may differ. For example, if x is a constant symbol with $x_{S_1} = 1$ and $x_{S_2} = 2$, the set $\text{inv}_{S_1}(f(x))$ includes the store $(f, 1, f(x)_{S_1})$, hence the location $(f, 1)$, whereas $\text{inv}_{S_2}(f(x))$ includes the store $(f, 2, f(x)_{S_2})$, hence the location $(f, 2)$.

Though the locations of $\text{inv}_{S_1}(\Gamma)$ and $\text{inv}_{S_2}(\Gamma)$ are in general different, they are never disjoint: For a statement γ in Γ, γ includes at least one constant symbol x. As the involved stores of γ are contained in both $\text{inv}_{S_1}(\Gamma)$ and $\text{inv}_{S_2}(\Gamma)$, this means that both $\text{inv}_{S_1}(\Gamma)$ and $\text{inv}_{S_2}(\Gamma)$ contain a store with the location $(x, \_)$.

4.4 Distributed runs

As a technical framework for distributed runs we adopt some notions from the formalism of Petri nets [9]: Each occurrence of an action is represented by an *action atom*, which is an inscribed Petri net holding just one transition. Action atoms are then used to construct distributed runs.

A *Petri net* is a tuple

$$N = (P, T, \lessdot)$$

where P and T are sets and $\lessdot$ is a relation $\lessdot \subseteq (P \times T) \cup (T \times P)$. P is the set of *places* of N, T is the set of *transitions* of N, and $\lessdot$ is the *flow relation* of N. ${}^\bullet t$ (and $t^\bullet$) denotes the set of all $p \in P$ with $p \lessdot t$ ($t \lessdot p$, respectively).

We will only deal with a very special class of Petri nets, and slightly deviate from the conventional graphical representation.

A distributed run of a distributed ASM D constitutes a partial order of action occurrences. Each action occurrence is represented by an action atom: For an action $a = (\Gamma, A, B)$ of D, an action atom of a is a simple Petri net inscribed by Γ, A, and B. To be more precise, let $(P, \{t\}, \lessdot)$ be a Petri net and let λ be an inscription of t and of all $p \in P$ such that

- ${}^\bullet t$ and $t^\bullet$ are disjoint,
- $\lambda(t) = \Gamma$,
- λ inscribes ${}^\bullet t$ bijectively by the stores in A,
- λ inscribes $t^\bullet$ bijectively by the stores in B.

Then $N_a = (P, \{t\}, \lessdot, \lambda)$ is an *action atom of a*. Since a is an action of the distributed ASM D, N_a is also called an *action atom of D*.

Reflecting the conventional graphical representation of Petri nets, an action atom of the action (Γ, A, B) with $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_n\}$ is outlined as

$$\begin{tikzcd}
a_1 \arrow[dr] & & b_1 \\
\vdots & \boxed{\Gamma} \arrow[ur] \arrow[dr] & \vdots \\
a_n \arrow[ur] & & b_n .
\end{tikzcd}$$

As a concrete example, consider the following action of the prod-component in (24):

$$\begin{aligned}(\texttt{prod},\{&(\texttt{item},\_,\boxtimes),(\texttt{x\_undef},\_,?),(\texttt{x},\_,?)\},\\ &\{(\texttt{item},\_,\boxtimes),(\texttt{x\_undef},\_,?),(\texttt{x},\_,\boxtimes)\}).\end{aligned} \tag{36}$$

This action replaces the undefined value in x (represented by ?) by the item value (represented by $\boxtimes$). An action atom of this action is then outlined as

$$\begin{tikzcd}
\texttt{item},\boxtimes \arrow[dr] & & \texttt{item},\boxtimes \\
\texttt{x\_undef},? \arrow[r] & \boxed{\texttt{prod}} \arrow[ur] \arrow[r] \arrow[dr] & \texttt{x\_undef},? \\
\texttt{x},? \arrow[ur] & & \texttt{x},\boxtimes
\end{tikzcd}.$$

As a convention, for 0-ary symbols f we write "f, a" for the store $(f, \_, a)$.

A distributed run R of a distributed ASM D is an inscribed Petri net, composed of action atoms of D. An action of D may occur more than once in R. In this case, the action is represented by different action atoms with equal inscriptions.

Technically, R is an inscribed occurrence net: An *occurrence net* is a Petri net $N = (P, T, \lessdot)$ where

- the transitive closure of $\lessdot$, denoted by $<$, is a strict partial order on $P \cup T$,
- for each $p \in P$, there exists at most one $t \in T$ with $t \lessdot p$, and at most one $t \in T$ with $p \lessdot t$,
- for each $x \in P \cup T$, $\{\ y \mid y < x\ \}$ is finite.

For an occurrence net N, $^\circ N$ denotes the set of all $p \in P$ with no predecessor $t \lessdot p$.

A distributed run of D is an inscribed occurrence net: Let $(P, T, \lessdot)$ be an occurrence net and let λ be an inscription of all $p \in P$ and all $t \in T$ where

- for each $t \in T$, the *restriction of* R *to* t, defined as

  $$R_{|t} \ =_{\text{def}}\ ({}^\bullet t \cup t^\bullet, \{t\}, \lessdot_{|\{t\}\cup{}^\bullet t\cup t^\bullet}, \lambda_{|\{t\}\cup{}^\bullet t\cup t^\bullet}),$$

  is a action atom of D,
- there exists a state S_0 of D, called the *initial state of* R, such that λ inscribes $^\circ N$ bijectively by the stores in $\widetilde{S_0}$. (37)

Then $R = (P, T, \lessdot, \lambda)$ is is a *distributed run* of D.

Figure 2 outlines a finite distributed run of the producer/consumer system (24). First, an action of the prod-component occurs, which assigns an item to x. Next, an action of send occurs and moves the item from location x to location buffer. Subsequently, an action of prod and an action of rec occur concurrently, as both actions involve stores with separate locations: The prod-action puts an item to x again, while the rec-action moves the item from buffer to y. For every transition t, observe that the subnet constituted by t, ${}^\bullet t$ and $t^\bullet$ is indeed an action atom of (24).

In a distributed run $R = (P, T, \lessdot, \lambda)$, global states are identified by the notion of *cut*. A cut is a maximal set of unordered places with only finitely many preceding transitions. More precisely, a cut of R is a set $C \subseteq P$ such that

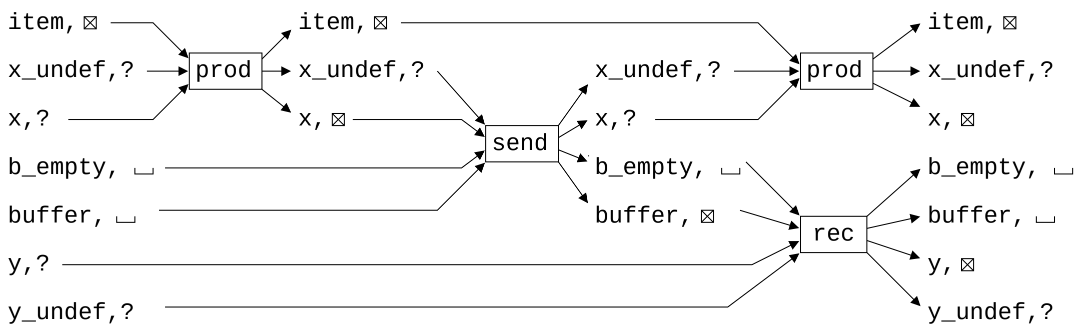

Figure 2: A distributed run of the distributed producer/consumer ASM.

- the places in C are pairwise unordered,
- C is maximal, i.e. there is no $p \in P \setminus C$ such that the places in $C \cup \{p\}$ are pairwise unordered,
- the set $T_{<C} =_{\text{def}} \{\ t \in T \mid \exists p \in C : t < p\ \}$ is finite.

According to this definition, ${}^\circ R$ is a cut of R, called the *initial cut of* R. As $T_{<C}$ is required to be finite, every cut of R can be reached from the initial cut of R by finitely many action occurrences. In Fig. 3, the dotted lines represent three different cuts of the distributed run of the producer/consumer ASM.

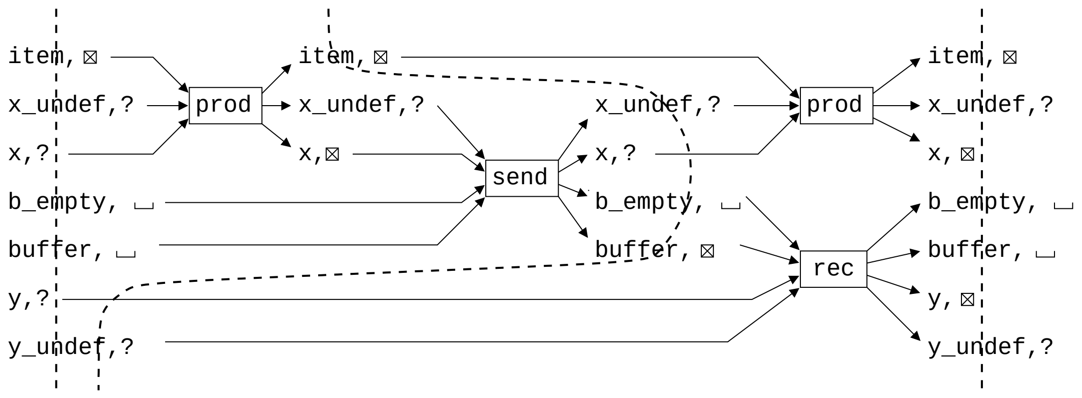

Figure 3: Three cuts in a distributed run.

The following important property holds for every cut of a distributed run:

Lemma 3. *Let D be a distributed ASM, let R be a distributed run of D, and let C be a cut of R. Then there is a state S of D, such that the places in C are inscribed bijectively by the stores in $\widetilde{S}$.*

Hence, every cut of a distributed run of D represents a state of D. Notice that the inscriptions of each cut in Fig. 3 indeed constitute a state of the producer/consumer system. The proof of this Lemma is postponed to Sec. 6.

In Sec. 4.2 we have prevented actions (and steps in Sec. 3.3, respectively) of an ASM program Γ to occur in a state S if

1. they yield an inconsistent set of stores, or
2. no condition of the statements in Γ is fulfilled in S.

Here we justify this proposal. If we admit actions to occur in such states, the resulting actions would not change the state at all. Nevertheless, such actions would involve stores and therefore could prevent the execution of other components. For example, the component `prod` in (24) is guarded by the condition $\texttt{x} = \texttt{x\_undef}$. In a state S where $\texttt{x}_S \neq \texttt{x\_undef}_S$, i.e. where an item is stored in `x`, this condition is not fulfilled. Executing `prod` in such a state would create the action

$$\begin{aligned}(\texttt{prod}, \{&(\texttt{x\_undef}, \_, ?), (\texttt{x}, \_, \boxtimes)\}, \\ &\{(\texttt{x\_undef}, \_, ?), (\texttt{x}, \_, \boxtimes)\}).\end{aligned} \tag{38}$$

`prod` could execute this action infinitely often, without changing the state at all. This would prevent execution of the component `send`, thus violating intuition.

For a distributed ASM $\{\Gamma\}$ consisting of a single ASM program Γ, the notions of distributed and sequential run coincide: According to Sec. 4.3, the stores of all actions of Γ share at least one location. As a consequence of Lemma 3, places inscribed by stores with equal locations are always ordered. Then, according to the definition of occurrence nets, the action occurrences of a distributed run of $\{\Gamma\}$ are totally ordered.

It is illuminating to compare the partial order of the component occurrences, as outlined in Fig. 4, with the partial order of (23): In fact, the latter is unnecessarily strict. This is due to the lockstep semantics of a sequential ASM: A run is a sequence of steps, and its action occurrences are unordered if they belong to the same step. This yields partial orders with a transitive non-order relation, such as (23). Figure 4 shows that, for a distributed run of a distributed ASM, non-order of action occurrences is not necessarily transitive: The second production occurs unordered with first consumption, which in turn occurs unordered to second send. But the second production is causally before the second send. This example shows that distributed ASMs in fact provide a substantial generalization of sequential ASMs.

$$\begin{tikzcd}
\texttt{prod} \arrow[dr] & & \texttt{prod} \arrow[dr] & & \texttt{prod} & \cdots \\
& \texttt{send} \arrow[ur] \arrow[dr] & & \texttt{send} \arrow[ur] \arrow[dr] & & \\
& & \texttt{rec} \arrow[ur] \arrow[dr] & & \texttt{rec} & \cdots \\
& & & \texttt{cons} \arrow[ur] & &
\end{tikzcd}$$

Figure 4: The partial order of component occurrences of a distributed run of the producer/consumer ASM.

The above semantics induces an inconvenient consequence, easily highlighted by an example. Let

$$D = \{\Gamma_1, \Gamma_2\}$$

be a distributed ASM, with

$$\begin{aligned}\Gamma_1 &= \{\texttt{x} := \texttt{1}\}\\ \Gamma_2 &= \{\texttt{y} := \texttt{1}\}.\end{aligned}$$

Intuitively, both components should be executed concurrently. But our semantics excludes concurrent access to the constant symbol $\texttt{1}$.

As another example, let

$$D' = \{\Gamma'_1, \Gamma'_2\}$$

with

$$\begin{aligned}\Gamma'_1 &= \{\texttt{x} := \texttt{f(a)}\}\\ \Gamma'_2 &= \{\texttt{y} := \texttt{f(b)}\}.\end{aligned}$$

In a state S where $\texttt{a}_S = \texttt{b}_S$, Γ'_1 and Γ'_2 are competing for the access to the location $(\texttt{f}, \texttt{a}_S)$ and cannot be executed concurrently. In every other state, Γ'_1 and Γ'_2 can be executed concurrently.

To overcome this problem, one may distinguish a subset $\Phi \subseteq \Sigma$ of the symbols in the underlying signature Σ, assuming different instances f_Γ of each $f \in \Phi$ for each component Γ of a distributed ASM D. Φ would usually cover all symbols with a standard interpretation, including e.g. symbols for the integers, the booleans and operations over integers and booleans.

This is not too artificial: Assuming the components of a distributed ASM be implemented on a computing network, each machine in fact has its own internal representation of the values and functions of those symbols.

5 A characterization of distributed ASMs

Gurevich [7] characterizes the expressive power of sequential ASMs by help of *sequential algorithms*. He formulates five necessary requirements for an algorithm to be reasonably called "sequential". 1st: A sequential algorithm $\mathcal{A}$ has a fixed signature Σ such that each state of $\mathcal{A}$ is a Σ-structure. 2nd: The states S and S' of a step (S, S') have the same universe. 3rd: The set of states (and, separately, the set of initial states) is closed under isomorphism. 4th: The steps of $\mathcal{A}$ preserve isomorphism. The decisive 5th property of a sequential algorithm is *bounded exploration*: Intuitively formulated, finite exploration requires a finite set of Σ-terms be sufficient to characterize all steps. Gurevich then proves that to each algorithm fulfilling those requirements there exists a sequential ASM with the same set of sequential runs.

In analogy to sequential algorithms, in the following subsections we define the notion of *distributed algorithm* by five axiomatic requirements. We then prove that to each distributed algorithm there exists a distributed ASM with the same set of distributed runs.

5.1 A distributed algorithm consists of actions

In [7] Gurevich introduced a sequential algorithm $\mathcal{A}$ to consist of a set $\mathcal{S}$ of Σ-algebras, the *states* of $\mathcal{A}$, and of a set of steps (S, S') with $S, S' \in \mathcal{S}$, encoded as a function $\tau_{\mathcal{A}} : \mathcal{S} \to \mathcal{S}$.

Global steps shaped (S, S') cannot be used to describe distributed behavior. Steps in a distributed algorithm occur locally and occasionally pairwise concurrent. Thus, we define the notion of *action over* Σ to describe a local step: An action a over a signature Σ is a pair

$$a = (A, B), \tag{39}$$

where A and B are sets of stores of Σ such that A as well as B is consistent, and the locations of the stores in A and in B coincide. In particular, if Γ is a sequential ASM over a signature Σ and (Γ, A, B) is an action of Γ, then (A, B) is an action over Σ. To refer to the components of an action a, let $a^{\text{old}} =_{\text{def}} A$ denote the *set of pre-stores of* a, and let $a^{\text{new}} =_{\text{def}} B$ denote the *set of post-stores of* a.

A sequential algorithm consists of a set of steps. Correspondingly, we require a distributed algorithm to consist of a set of actions. Furthermore, steps of a sequential algorithm contain only Σ-structures over a fixed signature Σ. We likewise require the actions of a distributed algorithm all to be built over the same signature. Thus, we can formulate the first requirement:

$$\boxed{\textit{A distributed algorithm } \mathcal{A} \textit{ consists of a set } \mathrm{act}(\mathcal{A}) \textit{ of actions over a fixed signature } \Sigma(\mathcal{A}).} \tag{R1}$$

The steps of a sequential algorithm generate *sequential runs*. Consequently, the actions of a distributed algorithm generate *distributed runs*. These runs are defined similarly to the runs of distributed ASMs as introduced in Sec. 4.4. Nevertheless, there is a small deviation: In contrast to distributed ASMs, an action of a distributed algorithm does not indicate an ASM program performing the action. Consequently, distributed runs of distributed algorithms do not inscribe transitions by ASM programs.

So, given an action a over a signature Σ, an *action atom* of a is an inscribed Petri net $N_a = (P, \{t\}, \lessdot, \lambda)$ where λ is an inscription of all $p \in P$ and

- $^\bullet t$ and $t^\bullet$ are disjoint,
- λ inscribes $^\bullet t$ bijectively by the stores in a^{old},
- λ inscribes $t^\bullet$ bijectively by the stores in a^{new}.

For an action a of a distributed algorithm $\mathcal{A}$, N_a is called an *action atom of* $\mathcal{A}$.

A distributed run of a distributed algorithm $\mathcal{A}$ is defined as an inscribed occurrence net $R = (P, T, \lessdot, \lambda)$ where λ is an inscription of all $p \in P$ and

- for each $t \in T$, the *restriction of* R *to* t, defined as

  $$R_{|t} \;=_{\text{def}}\; ({}^\bullet t \cup t^\bullet, \{t\}, \lessdot_{|\{t\} \cup {}^\bullet t \cup t^\bullet}, \lambda_{|{}^\bullet t \cup t^\bullet}),$$

  is a action atom of $\mathcal{A}$,

– there exists a $\Sigma(\mathcal{A})$-structure S_0, called the *initial state of* R, such that λ inscribes $^\circ N$ bijectively by the stores in $\widetilde{S_0}$.

5.2 Actions of a distributed algorithm preserve elements

In [7] sequential algorithms are required to preserve the universe of states, i.e. if (S, S') is a step of a sequential algorithm, then S and S' have the same universe. Hence, a step of a sequential algorithm does not introduce new elements into the universe.

We require a corresponding property for the actions of a distributed algorithm: For an action a, the stores of a^{new} contain only elements that are contained in the stores of a^{old} already. To formalize this requirement, we define the elements of a store $s = (f, (u_1, ..., u_n), u_0)$ as

$$E(s) \;=_{\text{def}}\; \{u_0, \ldots, u_n\}.$$

The elements of a set A of stores are then defined as

$$E(A) \;=_{\text{def}}\; \bigcup_{s \in A} E(s).$$

Informally, $E(A)$ contains all elements mentioned in A. We formulate the second requirement:

$$\boxed{\textit{For each action } a \textit{ of a distributed algorithm } \mathcal{A},\ E(a^{\text{new}}) \subseteq E(a^{\text{old}}).} \qquad \text{(R2)}$$

Thus, an action a of a distributed algorithm may use only elements already contained in the stores of a^{old} to perform the step. There is no way for an action to “magically” create a new semantical element and to insert it to a^{new}.

Due to (R2), for every distributed run R of a distributed algorithm holds: Every place inscription of R contains only elements from the universe of the initial state of R.

5.3 A distributed algorithm respects isomorphism

In [7], sequential algorithms are required to respect isomorphism between states. An isomorphism between two Σ-structures S_1, S_2 is a bijective function $\Phi : U(S_1) \to U(S_2)$ ($\Phi : S_1 \to S_2$ for short), such that

$$\Phi(f_{S_1}(u_1, \ldots, u_n)) = f_{S_2}(\Phi(u_1), \ldots, \Phi(u_n)) \qquad (40)$$

for all n-ary function symbols f in Σ and $u_1, \ldots, u_n \in U(S_1)$.

More precisely, the isomorphism requirement in [7] requires for two steps (S, S'), (R, R') and an isomorphism $\Phi : S \to R$, that Φ is an isomorphism from S' to R', too. Thus, if S_0 and S'_0 are isomorphic states of a sequential algorithm $\mathcal{A}$ with an isomorphism Φ, and

$$S_0 \to S_1 \to S_2 \to S_3 \ldots$$

is a sequential run of $\mathcal{A}$, then

$$S'_0 = \Phi(S_0) \to \Phi(S_1) \to \Phi(S_2) \to \Phi(S_3) \ldots$$

is a sequential run of $\mathcal{A}$, too. Intuitively, a sequential algorithm generates isomorphic sequential runs for isomorphic initial states. This we require in the distributed case, too: A distributed algorithm generates isomorphic distributed runs for isomorphic initial states.

To formalize this requirement, we extend any isomorphism $\Phi : S_1 \to S_2$ to act on stores $s = (f, (u_1, \ldots, u_n), u_0) \in \widetilde{S_1}$ by

$$\Phi(s) \ =_{\text{def}} \ (f, (\Phi(u_1), \ldots, \Phi(u_n)), \Phi(u_0)). \tag{41}$$

This extension is straightforward: replace every element in s by the isomorphic element according to Φ. Based on this definition, we can now formulate the third requirement:

Let $R = (P, T, \lessdot, \lambda)$ be a distributed run of a distributed algorithm $\mathcal{A}$ and let S_0 be the initial state of R. Let S_0' be a $\Sigma(\mathcal{A})$-structure isomorphic to S_0 and let $\Phi : S_0 \to S_0'$ be an isomorphism. Then $R' = (P, T, \lessdot, \Phi \circ \lambda)$ is a distributed run of $\mathcal{A}$, too. (R3)

Informally, (R3) states that a distributed algorithm creates isomorphic behaviours for isomorphic initial states. The run R' is well-defined: (R2) implies that every place inscription of R contains only elements from the universe of S_0. Therefore, Φ can be applied to any place inscription of R.

5.4 Actions of a distributed algorithm act on autonomous sets of stores

In this section, we will present a requirement for distributed algorithms, for which [7] has no counterpart. In contrast to sequential algorithms, the concept of distributed algorithms is sensitive against resources and their locations: Each action a of a distributed algorithm specifies the locations of all semantical elements involved. More precisely, a^{old} must contain a store (l, u) for each $u \in E(a^{\text{old}})$. To give an example, the set $\{(x, \_, 1), (f, 1, 2)\}$ meets this property, whereas $\{(f, 1, 2)\}$ does not. We call the set $\{(x, \_, 1), (f, 1, 2)\}$ *autonomous.*

Definitely, an algorithm may only access locations comprising of semantical elements available so far. The set of stores $\{(f, 1, 2), (f, 2, 1)\}$ violates this requirement: 1 needs to be available before the store $(f, 1, 2)$ can be accessed, and, conversely, 2 needs to be available before the store $(f, 2, 1)$ can be accessed. That is, none of both stores could have been accessed first. Consequently, we call $\{(f, 1, 2), (f, 2, 1)\}$ *not autonomous.*

The following inductive definition of *autonomous* sets of stores ensures that only locations comprising of available elements are accessed:

- $\emptyset$ is an autonomous set of stores.
- If A is an autonomous set of stores and $s = (f, (u_1, \ldots, u_n), u_0)$ is a store with $u_1, \ldots, u_n \in E(A)$, then $A \cup \{s\}$ is an autonomous set of stores. (42)

We can now formulate the fourth requirement:

For each action a of a distributed algorithm $\mathcal{A}$, a^{old} is autonomous. (R4)

Notice that a^{new} is not autonomous in general. To give an example, the ASM program $\Gamma = \{\texttt{x} := \texttt{f(x)}\}$ applied to a state S with $\texttt{x}_S = 1$ and $\texttt{f}_S(1) = 2$ yields the action

$$\begin{aligned}\Gamma_S = (\Gamma, \quad & \{(x, \_, 1), (f, 1, 2)\}, \\ & \{(x, \_, 2), (f, 1, 2)\} \quad).\end{aligned}$$

Obviously $\{(x, \_, 1), (f, 1, 2)\}$ is autonomous, whereas $\{(x, \_, 2), (f, 1, 2)\}$ is not.

5.5 Actions of a distributed algorithm are bounded

In [7] for every sequential algorithm $\mathcal{A}$ a *finite* set T of terms suffices to describe the updates of all states, i.e. to compute the next state of any given state. Thus, a sequential algorithm uses only a bounded set of resources of a state to compute the next state.

In the distributed case, all involved resources of an action a are fully contained in a^{old}, as a^{old} is required to be autonomous. It remains to require the *number* of involved resources to be bounded:

$$\boxed{\text{\textit{For a distributed algorithm } } \mathcal{A}\text{\textit{, there exists a constant } } c \in \mathbb{N} \text{ \textit{such that for each action} } a \text{ \textit{of} } \mathcal{A} \text{ \textit{holds} } |a^{\text{old}}| \leq c.} \qquad \text{(R5)}$$

Thus, for every distributed algorithm $\mathcal{A}$, the number of stores involved in an action is bounded. This requirement is rather natural, as in distributed systems an action is usually considered to be atomic. Actions that grow beyond any bound would contradict the intuition of atomicity.

5.6 A distributed ASM-Theorem

The sequential ASM-Theorem in [7] states that every sequential algorithm can be simulated by a sequential ASM. We formulate a similar theorem for distributed algorithms we defined above: Every distributed algorithm can be simulated by a distributed ASM. More precisely, for every distributed algorithm $\mathcal{A}$, there exists a distributed ASM D such that the actions of $\mathcal{A}$ and D coincide.

Technically, the actions of $\mathcal{A}$ and D differ: An action of D specifies the ASM program executing the action, whereas an action of $\mathcal{A}$ does not. We call the actions of the distributed ASM D *named* and call the actions of the distributed algorithm $\mathcal{A}$ *anonymous*. To integrate actions of $\mathcal{A}$ and D, we *anonymize* named actions by

$$\text{anon}((\Gamma, A, B)) \; =_{\text{def}} \; (A, B).$$

The function anon extends canonically to sets of named actions: Let ACT be a set of named actions, then

$$\text{anon}(\text{ACT}) \; =_{\text{def}} \; \{ \, \text{anon}(a) \mid a \in \text{ACT} \, \}.$$

The distributed ASM-Theorem then reads:

Theorem 4. *Let $\mathcal{A}$ be a distributed algorithm according to (R1), ..., (R5). Then there exists a distributed ASM D such that*

$$\mathrm{act}(\mathcal{A}) = \mathrm{anon}(\mathrm{act}(D)).$$

Thus, the actions of every distributed algorithm can be fully specified by a distributed ASM. On the other hand, it is easy to prove that every distributed ASM constitutes a distributed algorithm by verifying (R1), ..., (R5). As the actions of $\mathcal{A}$ and D coincide, it follows immediately that $\mathcal{A}$ and D generate the same set of distributed runs, up to transition inscriptions by ASM programs.

6 Proofs

This section presents the proof of Theorem 4. In the first two subsections we study equivalences between states and prove some related properties. Next, we show how an ASM program Γ can be constructed for a given action a such that a is an action of Γ. Based on this construction, Theorem 4 is proven. Finally, the last three subsections present the proofs of Lemma 1, 2 and 3.

6.1 ASM programs and their actions

As defined in (34), an ASM program Γ generates for each state S an action Γ_S. In this section we show how equivalences between states carry over to equivalences of the generated actions.

The first equivalence is induced by *isomorphism* between states. To this end, we extend isomorphisms to store sets and actions: Let S, R be isomorphic Σ-structures with an isomorphism $\Phi : U(S) \to U(R)$, and let A be a set of stores over Σ with $E(A) \subseteq U(S)$. Then

$$\Phi(A) \ =_{\mathrm{def}} \ \{ \ \Phi(s) \mid s \in A \ \}$$

is a set of stores with $E(\Phi(A)) \subseteq U(R)$. For each named action (Γ, A, B) and each anonymous action (A, B), we define

$$\begin{aligned} \Phi((\Gamma, A, B)) \ &=_{\mathrm{def}} \ (\Gamma, \Phi(A), \Phi(B)), \\ \Phi((A, B)) \ &=_{\mathrm{def}} \ (\Phi(A), \Phi(B)). \end{aligned} \tag{43}$$

Hence, all elements occurring in the actions are replaced according to the isomorphism.

An ASM program yields isomorphic actions to isomorphic states, as the following lemma shows:

Lemma 5. *Let Σ be a signature, let Γ be an ASM program and let S, R be isomorphic Σ-structures with an isomorphism $\Phi : S \to R$. Then $\Gamma_R = \Phi(\Gamma_S)$.*

Proof. As Φ is an isomorphism from S to R, $t_R = \Phi(t_S)$ for all Σ-terms t. Applying these equations to the definition of Γ_R yields $\Gamma_R = \Phi(\Gamma_S)$. □

The second equivalence is *equality of term interpretations*. Assume two states coincide on the interpretation of *every* term and *every* subterm occurring in an ASM program Γ. Then Γ yields the same actions in both states. Formally, we define the *subterm closure* $\downharpoonleft T$ of a set T of terms to be the least set such that $T \subseteq \downharpoonleft T$ and such that $f(t_1, \dots, t_n) \in \downharpoonleft T$ implies $t_1, \dots, t_n \in \downharpoonleft T$. The corresponding lemma then reads:

Lemma 6. *Let Σ be a signature, let Γ be an ASM program over Σ, and let T be the set of all terms occurring in Γ. Let S, R be Σ-structures such that $t_S = t_R$ for all $t \in \downharpoonleft T$. Then $\Gamma_S = \Gamma_R$.*

Proof. Applying the equations $t_S = t_R$ for all $t \in \downharpoonleft T$ to the definition of Γ_S yields $\Gamma_S = \Gamma_R$. □

6.2 T-equivalence

In addition to the equivalences of the preceding section, we introduce *T-equivalence* in this section, where T is a set of Σ-terms. Two Σ-structures R and S are *T-equivalent* if both R and S concordantly identify the terms in T by their interpretations:

Definition 1 (T-equivalence)**.** Let Σ be a signature, and let T be a set of Σ-terms. Let S, R be Σ-structures such that $t_S = t'_S$ iff $t_R = t'_R$ for all $t, t' \in T$. Then S and R are *T-equivalent.*

The following lemma characterizes T-equivalence in terms of isomorphism and equality:

Lemma 7. *Let Σ be a signature, and let T be a set of Σ-terms. Two Σ-structures S and R are T-equivalent iff there exists a Σ-structure Q such that S and Q are isomorphic and $t_Q = t_R$ for all $t \in T$.*

Proof. ($\Rightarrow$) Generate Q from S by replacing for all $t \in T$ the element t_S by t_R and by replacing every other element from $U(S)$ by a new element not contained in $U(S)$. As S and R are T-equivalent, this construction is well-defined. By construction of Q, S and Q are isomorphic, and $t_Q = t_R$ for all $t \in T$.

($\Leftarrow$) For all $t, t' \in T$ holds

$$\begin{aligned} t_S = t'_S &\Leftrightarrow t_Q = t'_Q \quad \text{(as } S \text{ and } Q \text{ are isomorphic)} \\ &\Leftrightarrow t_R = t'_R \quad \text{(as } t_Q = t_R \text{ for all } t \in T\text{).} \end{aligned}$$

Hence, S and R are T-equivalent. □

6.3 Deriving ASM programs from actions

We prove Theorem 4 in a constructive manner, i.e. for a given distributed algorithm, we construct a distributed ASM. The foundations of this construction are laid in this section: For an action a of a distributed algorithm, an ASM programm Γ is constructed such that a is an action of Γ.

First of all, we introduce a simplifying notation: For a set A of stores over Σ, let

$$L(A) \ =_{\text{def}}\ \{\, l \mid (l, u_0) \in A \,\} \tag{44}$$

denote the *set of locations of A*. The following lemma and its proof then provide the first step of the construction of the ASM programm Γ: From a^{old} a set T of terms is derived such that $\text{inv}_S(T) = a^{\text{old}}$ for some state S, with

$$\text{inv}_S(T) \ =_{\text{def}}\ \bigcup_{t \in T} \text{inv}_S(t). \tag{45}$$

Later, Γ will be constructed from the terms in T.

Lemma 8. *Let Σ be a signature and let A be an consistent and autonomous set of stores over Σ. Let S be a Σ-structure such that $A \subseteq \widetilde{S}$. Then there exists a finite set T of Σ-terms such that*

- *T is closed under subterms,*
- *$\text{inv}_S(T) = A$,*
- *for every $u \in E(A)$ there is a term $t^u \in T$ with $t^u_S = u$,*
- *for every $l \in L(A)$ there is a term $t^l \in T$ with $\text{loc}_S(t^l) = l$.*

Proof. We prove this by induction over A:
Base: Let $A = \emptyset$. Then $T = \emptyset$ satisfies the requirements.
Step: Let $A \neq \emptyset$. As A is autonomous, according to (42) there is an autonomous subset $A' \subset A$ such that $A = A' \cup \{(f, (u_1, \ldots, u_n), u_0)\}$ with $u_1, \ldots, u_n \in E(A')$. Let Lemma 8 hold for A'. That is, there is a set T' of Σ-terms such that

- T' is closed under subterms, (46)
- $\text{inv}_S(T') = A'$, (47)
- for every $u \in E(A')$ there is a term $t^u \in T'$ with $\text{val}_S(t^u) = u$. (48)
- for every $l \in L(A')$ there is a term $t^l \in T'$ with $\text{loc}_S(t^l) = l$. (49)

As $u_1, \ldots, u_n \in E(A')$, and by the use of (48), set

$$\begin{aligned} t &\ =_{\text{def}}\ f(t^{u_1}, \ldots, t^{u_n}), \\ T &\ =_{\text{def}}\ T' \cup \{t\}. \end{aligned}$$

As $A \subseteq \widetilde{S}$, $(f, (u_1, \ldots, u_n), u_0) \in \widetilde{S}$. Therefore,

$$\begin{aligned} t_S &= f_S(t^{u_1}_S, \ldots, t^{u_n}_S) \\ &= f_S(u_1, \ldots, u_n) && \text{(by (48))} \\ &= u_0 && \text{(as } (f, (u_1, \ldots, u_n), u_0) \in \widetilde{S}\text{).} \end{aligned} \tag{50}$$

Furthermore,

$$\begin{aligned}\mathrm{loc}_S(t) &= (f, (t_S^{u_1}, \ldots, t_S^{u_n})) \\ &= (f, (u_1, \ldots, u_n)) \qquad \text{(by (48))}. \end{aligned} \tag{51}$$

By construction of t and by (46), T is closed under subterms. We show $\mathrm{inv}_S(T) = A$:

$$\begin{aligned}\mathrm{inv}_S(T) =&\mathrm{inv}_S(T') \cup \mathrm{inv}_S(t) \\ &\qquad \text{(as } T = T' \cup \{t\} \text{ and by (45))} \\ =&\mathrm{inv}_S(T') \cup \mathrm{inv}_S(t^{u_1}) \cup \cdots \cup \mathrm{inv}_S(t^{u_n}) \cup \{(\mathrm{loc}_S(t), t_S)\} \\ &\qquad \text{(by (25))} \\ =&\mathrm{inv}_S(T') \cup \{(\mathrm{loc}_S(t), t_S)\} \\ &\qquad \text{(as } t^{u_1}, \ldots t^{u_n} \in T' \text{ and by (45))} \\ =&\mathrm{inv}_S(T') \cup \{(f, (u_1, \ldots, u_n), u_0)\} \\ &\qquad \text{(by (50) and (51))} \\ =&A' \cup \{(f, (u_1, \ldots, u_n), u_0)\} \\ &\qquad \text{(by (47))} \\ =&A. \end{aligned}$$

Now we prove that, for every $u \in E(A)$, there is a term $t^u \in T$ with $t_S^u = u$: In case $u \in E(A')$, this is true according to (48). In case $u \notin E(A')$, $u = u_0$. With $t^u =_{\mathrm{def}} t$ and by (50) holds

$$t_S^u = t_S = u_0 = u.$$

Finally, for $l \in L(A)$, we show that there is a $t^l \in T$ with $\mathrm{loc}_S(t^l) = l$: In case $l \in L(A')$, this is true according to (49). In case $l \notin E(A')$, $l = (f, (u_1, \ldots, u_n))$. With $t^l =_{\mathrm{def}} t$ and by (51) holds

$$\mathrm{loc}_S(t^l) = \mathrm{loc}_S(t) = (f, (u_1, \ldots, u_n)) = l.$$

□

The following lemma states that the involved stores of an ASM program Γ are completely determined by the terms occurring in Γ.

Lemma 9. *Let Σ be a signature, let Γ be an ASM program and let T be the set of all terms occurring in Γ. let S be a Σ-structure such that the guard of at least one statement in Γ is fulfilled. Then* $\mathrm{inv}_R(\Gamma) = \mathrm{inv}_R(T)$.

Proof. For every boolean expression β over Σ, let T^β denote the set of all terms occurring in β. By (26), $\mathrm{inv}_S(\beta) = \mathrm{inv}_S(T^\beta)$. For every assignment statement α: $t := t_0$, let

$T^\alpha = \{t, t_0\}$. By (27), $\mathrm{inv}_S(\alpha) = \mathrm{inv}_S(T^\alpha)$. Then holds:

$$\begin{aligned}
\mathrm{inv}_S(\Gamma) &= \bigcup_{(\texttt{if } \beta \texttt{ then } \alpha)\in\Gamma} \mathrm{inv}_S(\texttt{if } \beta \texttt{ then } \alpha) \\
&= \bigcup_{(\texttt{if } \beta \texttt{ then } \alpha)\in\Gamma} \mathrm{inv}_S(\beta) \cup \mathrm{inv}_S(\alpha) \qquad \text{(by (31))} \\
&= \bigcup_{(\texttt{if } \beta \texttt{ then } \alpha)\in\Gamma} \mathrm{inv}_S(T^\beta) \cup \mathrm{inv}_S(T^\alpha) \\
&= \mathrm{inv}_S\Big(\bigcup_{(\texttt{if } \beta \texttt{ then } \alpha)\in\Gamma} T^\beta \cup T^\alpha \Big) \\
&= \mathrm{inv}_S(T).
\end{aligned}$$

□

In analogy to isomorphisms between structures, we introduce *action isomorphisms* between actions: Two actions are *isomorphic* if they can be derived from each other by bijectively replacing their elements. The *set of elements* of an action a is simply defined by $E(a) =_{\mathrm{def}} E(a^{\mathrm{old}}) \cup E(a^{\mathrm{new}})$. Formally, action isomorphisms are then defined as follows:

Definition 2 (action isomorphism)**.** Let Σ be a signature and let a, b be actions over Σ. Further let $\phi : E(a) \to E(b)$ be a bijection such that $\phi(a) = b$, with $\phi(a)$ defined as in (43). Then ϕ is an *action isomorphism* between a and b, and a and b are *isomorphic*, written $a \cong b$. The *isomorphism class* of a is

$$[a] \;=_{\mathrm{def}}\; \{\, b \mid b \cong a \,\}.$$

As announced at the beginning of this section, the following lemma and the corresponding proof present, for a given action a of a distributed algorithm, the construction of an ASM program Γ such that a is an action of Γ. Furthermore, this construction ensures every other action of Γ to be isomorphic to a:

Lemma 10. *Let Σ be a signature and let a be an action over Σ such that a^{old} is autonomous and $E(a^{\mathrm{new}}) \subseteq E(a^{\mathrm{old}})$. Then there is an ASM program Γ such that* $\mathrm{anon}(\mathrm{act}(\Gamma)) = [a]$.

Proof. We start with the construction of Γ and prove $\mathrm{anon}(\mathrm{act}(\Gamma)) = [a]$ afterwards. Let S be a Σ-structure with $a^{\mathrm{old}} \subseteq \widetilde{S}$. By Lemma 8, there is a finite set T of Σ-terms such that

- T is closed under subterms,
- $\mathrm{inv}_S(T) = a^{\mathrm{old}}$, (52)
- for every $u \in E(a^{\mathrm{old}})$ there is a term $t^u \in T$ with $t^u_S = u$. (53)

– for every $l \in L(a^{\text{old}})$ there is a term $t^l \in T$ with $\text{loc}_S(t^l) = l$. (54)

As T is finite, we can construct the following boolean expression:

$$\beta = \bigwedge_{t,t' \in T} \begin{cases} (t = t') & \text{, if } t_S = t'_S \\ \neg(t = t') & \text{, otherwise} \end{cases}$$

For each store $(l, u_0) \in a^{\text{new}}$ holds: As $L(a^{\text{old}}) = L(a^{\text{new}})$, $l \in L(a^{\text{old}})$. As $E(a^{\text{new}}) \subseteq E(a^{\text{old}})$, $u_0 \in E(a^{\text{old}})$. Therefore, by the use of (53) and (54), set

$$\Gamma \;=_{\text{def}}\; \{\, \texttt{if}\ \beta\ \texttt{then}\ t^l := t^{u_0} \mid (l, u_0) \in a^{\text{new}} \,\}.$$

This construction implies the following properties:

1. $\text{anon}(\Gamma_S) = a$, (55)
2. $[a] \subseteq \text{anon}(\text{act}(\Gamma))$,
3. $\text{anon}(\text{act}(\Gamma)) \subseteq [a]$.

In the following we prove each of the these properties separately.

Proof of 1. By construction of β, S satisfies β. First, we show $\Gamma_S^{\text{new}} = a^{\text{new}}$:

$$\begin{aligned} \Gamma_S^{\text{new}} &= \{\, (t^l := t^{u_0})_S^{\text{new}} \mid (l, u_0) \in a^{\text{new}} \,\} && \text{(by (17))} \\ &= \{\, (\text{loc}_S(t^l), t_S^{u_0}) \mid (l, u_0) \in a^{\text{new}} \,\} && \text{(by (10))} \\ &= \{\, (l, u_0) \mid (l, u_0) \in a^{\text{new}} \,\} && \text{(by (53) and (54))} \\ &= a^{\text{new}}. \end{aligned}$$

Next, we show $\Gamma_S^{\text{old}} = a^{\text{old}}$:

$$\begin{aligned} \Gamma_S^{\text{old}} &= \{\, (l, u_0) \in \widetilde{S} \mid l \in L(\Gamma_S^{\text{old}}) \,\} && \text{(as } \Gamma_S^{\text{old}} \subseteq \widetilde{S} \text{ and as } \widetilde{S} \text{ is consistent)} \\ &= \{\, (l, u_0) \in \widetilde{S} \mid l \in L(\Gamma_S^{\text{new}}) \,\} && \text{(as } L(\Gamma_S^{\text{old}}) = L(\Gamma_S^{\text{new}})) \\ &= \{\, (l, u_0) \in \widetilde{S} \mid l \in L(a^{\text{new}}) \,\} && \text{(as } \Gamma_S^{\text{new}} = a^{\text{new}}) \\ &= \{\, (l, u_0) \in \widetilde{S} \mid l \in L(a^{\text{old}}) \,\} && \text{(as } L(a^{\text{old}}) = L(a^{\text{new}})) \\ &= a^{\text{old}} && \text{(as } a^{\text{old}} \subseteq \widetilde{S} \text{ and as } \widetilde{S} \text{ is consistent).} \end{aligned}$$

Finally, we show $\text{inv}_S(\Gamma) = a^{\text{old}}$:

$$\begin{aligned} \text{inv}_S(\Gamma) &= \text{inv}_S(T) && \text{(by Lemma 9)} \\ &= a^{\text{old}} && \text{(by (52)).} \end{aligned}$$

Summing up, $\Gamma_S^{\text{new}} = a^{\text{new}}$, $\Gamma_S^{\text{old}} = a^{\text{old}}$, and $\text{inv}_S(\Gamma) = a^{\text{old}}$. Therefore, we conclude

$$\text{inv}'_S(\Gamma) = (\text{inv}_S(\Gamma) \setminus \Gamma_S^{\text{old}}) \cup \Gamma_S^{\text{new}} = (a^{\text{old}} \setminus a^{\text{old}}) \cup a^{\text{new}} = a^{\text{new}}. \quad (56)$$

(52) and (56) together yield

$$\mathrm{anon}(\Gamma_S) = (\mathrm{inv}_S(\Gamma), \mathrm{inv}'_S(\Gamma)) = (a^{\mathrm{old}}, a^{\mathrm{new}}) = a.$$

Proof of 2. Let $b \in [a]$, i.e. $b \cong a$. Let ϕ be an action isomorphism from a to b. Construct from S a Σ-structure R by replacing every element $u \in E(a)$ by $\phi(u)$ and every other element from $U(S)$ by a new one not contained in $U(S)$. Then R is isomorphic to S with an isomorphism Φ such that $\phi = \Phi_{|E(a)}$. Then

$$\begin{aligned}
\mathrm{anon}(\Gamma_R) &= \mathrm{anon}(\Phi(\Gamma_S)) && \text{(by Lemma 5)}\\
&= \Phi(\mathrm{anon}(\Gamma_S)) && \\
&= \Phi(a) && \text{(by (55))}\\
&= \phi(a) && \text{(as } \phi = \Phi_{|E(a)}\text{)}\\
&= b && \text{(by Def. 2)}
\end{aligned}$$

Hence, $b \in \mathrm{anon}(\mathrm{act}(\Gamma))$.

Proof of 3. Let $b \in \mathrm{anon}(\mathrm{act}(\Gamma))$. Then there is a Σ-structure R such that $R \models \beta$ and $b = \mathrm{anon}(\Gamma_R)$. As β is satisfied by R and S, for all $t, t' \in T$ holds

$$t_R = t'_R \Leftrightarrow t_S = t'_S.$$

Hence, R and S are T-equivalent. According to Lemma 7, there is a Σ-structure Q such that S and Q are isomorphic and $t_Q = t_R$ for all $t \in T$. Let Φ be an isomorphism from S to Q. By construction of Γ, T is the set of all terms occurring in Γ. Furthermore, as T is closed under subterms, $T = \downharpoonleft T$. Then

$$\begin{aligned}
\Phi(a) &= \Phi(\mathrm{anon}(\Gamma_S)) && \text{(by (55))}\\
&= \mathrm{anon}(\Phi(\Gamma_S)) && \\
&= \mathrm{anon}(\Gamma_Q) && \text{(by Lemma 5)}\\
&= \mathrm{anon}(\Gamma_R) && \text{(as } t_Q = t_R \text{ for all } t \in \downharpoonleft T \text{ and by Lemma 6)}\\
&= b.
\end{aligned}$$

Therefore, $\Phi_{|E(a)}$ is an action isomorphism from a to b, i.e. $b \in [a]$. □

6.4 Main proof

In this section we present two important lemmata, and finally prove Theorem 4. The first lemma states that, for a distributed algorithm $\mathcal{A}$ and for an action a of $\mathcal{A}$, each action $b \cong a$ is an action of $\mathcal{A}$, too. Hence, $\mathrm{act}(\mathcal{A})$ is closed under action isomorphism.

Lemma 11. *Let $\mathcal{A}$ be a distributed algorithm and let $a \in \mathrm{act}(\mathcal{A})$. Then $[a] \subseteq \mathrm{act}(\mathcal{A})$.*

Proof. Let $b \in [a]$ and let ϕ be an action isomorphism between a and b. Let S be a state such that $a^{\mathrm{old}} \subseteq \widetilde{S}$. Let $R = (P, T, \lessdot, l)$ be a distributed run of $\mathcal{A}$ with initial state S and one occurrence of action a, i.e. $T = \{t\}$ and $R_{|t}$ is an action atom of a.

Construct a Σ-state S' from S by replacing every $u \in E(a)$ by $\phi(u)$ and every other element from $U(S)$ by a new one not contained in $U(S)$. By construction, S is isomorphic to S' with an isomorphism Φ such that $\phi = \Phi_{|E(a)}$.

According to (R3), $R' = (P, T, \lessdot, \Phi \circ l)$ is a distributed run of $\mathcal{A}$. As $R_{|t}$ is an action atom of a, $R'_{|t}$ is an action atom of $\Phi(a)$. Hence, as $\Phi(a) = \phi(a) = b$, $R'_{|t}$ is an action atom of b. Therefore, b is an action of $\mathcal{A}$, otherwise R' would not be a run of $\mathcal{A}$. □

According to (R5), actions of a distributed algorithm are bound in size. Therefore, the following lemma holds:

Lemma 12. *Let $\mathcal{A}$ be a distributed algorithm. Then* act$(\mathcal{A})$ *decomposes into finitely many equivalence classes wrt* $\cong$.

Proof. According to (R5), the size of steps in act$(\mathcal{A})$ is bounded by a constant c. Therefore, there is an upper bound $m \in \mathbb{N}$ such that $|E(a)| \le m$ for all actions $a \in \operatorname{act} \mathcal{A}$.

Let M be a set consisting of m elements. Then for every action a of $\mathcal{A}$, construct an action a_M by replacing each element in $E(a)$ by an unique element from M. By construction, $E(a_M) \subseteq M$ and $a \cong a_M$.

Let act_M denote the set of all actions b over Σ with $E(b) \subseteq M$. As M is finite, act_M is finite, too. But since every action $a \in \operatorname{act}(\mathcal{A})$ is isomorphic to $a_M \in \operatorname{act}_M$, act$(\mathcal{A})$ decomposes into finitely many isomorphism classes. □

Finally, we prove Theorem 4:

Proof of Theorem 4. According to Lemma 12, the equivalence $\cong$ decomposes act$(\mathcal{A})$ into finitely many equivalence classes $C_1, \dots, C_n$.

For $i = 1, \dots, n$, choose an action $a_i \in C_i$. According to Lemma 10, for every $i = 1, \dots, n$ there is an ASM program Γ_i such that

$$\operatorname{anon}(\operatorname{act}(\Gamma_i)) = [a_i]. \tag{57}$$

According to Lemma 11, for every equivalence class C_i, $i = 1, \dots, n$, holds $[a_i] \subseteq C_i$. As all actions in C_i are pairwise isomorphic,

$$C_i = [a_i]. \tag{58}$$

(57) and (58) yield:

$$C_i = \operatorname{anon}(\operatorname{act}(\Gamma_i)). \tag{59}$$

Thus, every equivalence class C_i is generated by an ASM program Γ_i.

For the distributed ASM $D = \{\Gamma_1, \dots, \Gamma_n\}$ then holds:

$$\begin{aligned}
\operatorname{act}(\mathcal{A}) &= C_1 \cup \dots \cup C_n \\
&= \operatorname{anon}(\operatorname{act}(\Gamma_1)) \cup \dots \cup \operatorname{anon}(\operatorname{act}(\Gamma_n)) \quad \text{(by (59))} \\
&= \operatorname{anon}(\operatorname{act}(\Gamma_1) \cup \dots \cup \operatorname{act}(\Gamma_n)) \\
&= \operatorname{anon}(\operatorname{act}(D)). \quad \text{(by (35))}
\end{aligned}$$

□

6.5 Proof of Lemma 1

Lemma 1 states that for an ASM program Γ and for a state S of Γ, $\text{inv}_S(\Gamma)$ comprises all stores needed to compute $\text{inv}'_S(\Gamma)$. More precisely, in case $\text{inv}_S(\Gamma) = \text{inv}_R(\Gamma)$ for two states S, R of Γ, $\text{inv}'_S(\Gamma) = \text{inv}'_R(\Gamma)$.

Before proving Lemma 1, we present three preparing lemmata. Intuitively, the first lemma states that the evaluation of a term is fully determined by its set of involved stores.

Lemma 13. *Let Σ be a signature, let T be a set of Σ-terms closed under subterms, and let S, R be Σ-structures such that* $\text{inv}_R(T) \subseteq \widetilde{S}$. *Then $t_S = t_R$ for all $t \in T$.*

Proof. Let $t \in T$. The proof is given by induction over the subterms in t.

Base: Let t be a 0-ary function symbol. By (25) holds $(t, \_, t_R) \in \text{inv}_R(t)$. As $\text{inv}_R(t) \subseteq \text{inv}_R(T) \subseteq \widetilde{S}$, $(t, \_, t_R) \in \widetilde{S}$. By (2), $(t, \_, t_S) \in \widetilde{S}$. As $\widetilde{S}$ is consistent, $t_S = t_R$.

Step: Let $t = f(t_1, \ldots, t_n)$ and let $t_{iR} = t_{iS}$ for all $i = 1, \ldots, n$. By (25) holds $(f, (t_{1R}, \ldots, t_{nR}), t_R) \in \text{inv}_R(t)$. As $\text{inv}_R(t) \subseteq \text{inv}_R(T) \subseteq \widetilde{S}$, $(f, (t_{1R}, \ldots, t_{nR}), t_R) \in \widetilde{S}$. As $t_{iR} = t_{iS}$ for all $i = 1, \ldots, n$, $(f, (t_{1S}, \ldots, t_{nS}), t_R) \in \widetilde{S}$. By (2), $(f, (t_{1S}, \ldots, t_{nS}), t_S) \in \widetilde{S}$. As $\widetilde{S}$ is consistent, $t_S = t_R$. □

The next lemma states that the evaluation of involved stores is invariant under subterm closure:

Lemma 14. *Let Σ be a signature, and let T be a finite set of Σ-terms. Then for every Σ-structure S holds* $\text{inv}_S(T) = \text{inv}_S(\downharpoonleft T)$.

Proof. Obviously, the lemma holds in case $T = \downharpoonleft T$. Therefore, let $T \subset \downharpoonleft T$. Let $t = f(t_1, \ldots, t_n) \in T$ such that $t_k \notin T$ for some $k \in \{1, \ldots, n\}$. By (25), $\text{inv}_S(t_k) \subseteq \text{inv}_S(t)$. As $t \in T$, $\text{inv}_S(t) \subseteq \text{inv}_S(T)$. Hence, $\text{inv}_S(t_k) \subseteq \text{inv}_S(T)$.

$$\begin{aligned} \text{inv}_S(T) &= \text{inv}_S(T) \cup \text{inv}_S(t_k) && (\text{as } \text{inv}_S(t_k) \subseteq \text{inv}_S(T)) \\ &= \text{inv}_S(T \cup \{t_k\}). && (\text{by } (45)) \end{aligned}$$

Thus, the set of involved stores does not change if the subterm t_k is added to T. Repeating this step iteratively, we gain a sequence of subterms $v_1, \ldots, v_m$ such that $\downharpoonleft T = T \cup \{v_1, \ldots, v_m\}$, and

$$\begin{aligned} \text{inv}_S(T) &= \text{inv}_S(T \cup \{v_1, \ldots, v_m\}) \\ &= \text{inv}_S(\downharpoonleft T). \end{aligned}$$

□

The next lemma states that the action of an ASM program is fully determined by the set of involved stores.

Lemma 15. *Let Σ be a signature, let Γ be an ASM program, and let S, R be Σ-structures such that* $\text{inv}_R(\Gamma) \subseteq \widetilde{S}$. *Then $\Gamma_S = \Gamma_R$.*

Proof. Let T be the set of all terms occurring in Γ. Then

$$\begin{aligned} \mathrm{inv}_R(\downharpoonleft T) &= \mathrm{inv}_R(T) && \text{(by Lemma 14)} \\ &= \mathrm{inv}_R(\Gamma) && \text{(by Lemma 9)} \\ &\subseteq \widetilde{S}. && \text{(by preconditions)} \end{aligned}$$

Then, according to Lemma 13, $t_R = t_S$ for all $t \in \downharpoonleft T$. Lemma 6 then implies $\Gamma_R = \Gamma_S$. □

The following proof shows that Lemma 1 is a corollary of Lemma 15:

Proof of Lemma 1. As $\mathrm{inv}_S(\Gamma) \subseteq \widetilde{S}$ and as $\mathrm{inv}_S(\Gamma) = \mathrm{inv}_R(\Gamma)$, $\mathrm{inv}_R(\Gamma) \subseteq \widetilde{S}$. According to Lemma 15, $\Gamma_R = \Gamma_S$. By (34), $\mathrm{inv}'_R(\Gamma) = \mathrm{inv}'_S(\Gamma)$. □

6.6 Proof of Lemma 2

Lemma 2 states that the actions of an ASM program Γ fully characterize the steps of Γ. More precisely, (S, S') is a step of Γ iff there is an action (Γ, A, B) of Γ with $A \subseteq \widetilde{S}$ and $\widetilde{S'} = (\widetilde{S} \setminus A) \cup B$.

Proof of Lemma 2. In (18), a step of Γ is defined by Γ_S^{old} and Γ_S^{new}. First we show that a step of Γ can also be described by means of $\mathrm{inv}_S(\Gamma)$ and $\mathrm{inv}'_S(\Gamma)$: According to (18), (S, S') is a step of Γ iff $\widetilde{S'} = (\widetilde{S} \setminus \Gamma_S^{\mathrm{old}}) \cup \Gamma_S^{\mathrm{new}}$. We derive

$$\begin{aligned} \widetilde{S'} &= (\widetilde{S} \setminus \Gamma_S^{\mathrm{old}}) \cup \Gamma_S^{\mathrm{new}} \\ &= (((\widetilde{S} \setminus \mathrm{inv}_S(\Gamma)) \cup \mathrm{inv}_S(\Gamma)) \setminus \Gamma_S^{\mathrm{old}}) \cup \Gamma_S^{\mathrm{new}} && \text{(as } \mathrm{inv}_S(\Gamma) \subseteq \widetilde{S}\text{)} \\ &= ((\widetilde{S} \setminus \mathrm{inv}_S(\Gamma)) \setminus \Gamma_S^{\mathrm{old}}) \cup (\mathrm{inv}_S(\Gamma) \setminus \Gamma_S^{\mathrm{old}}) \cup \Gamma_S^{\mathrm{new}} && \text{(distributivity of } \cup \text{ and } \setminus\text{)} \\ &= (\widetilde{S} \setminus \mathrm{inv}_S(\Gamma)) \cup (\mathrm{inv}_S(\Gamma) \setminus \Gamma_S^{\mathrm{old}}) \cup \Gamma_S^{\mathrm{new}} && \text{(by (32))} \\ &= (\widetilde{S} \setminus \mathrm{inv}_S(\Gamma)) \cup \mathrm{inv}'_S(\Gamma) && \text{(by (33)).} \end{aligned}$$

Hence, (S, S') is a step of Γ iff

$$\widetilde{S'} = (\widetilde{S} \setminus \mathrm{inv}_S(\Gamma)) \cup \mathrm{inv}'_S(\Gamma). \tag{60}$$

We will prove both directions of the lemma separately.

($\Rightarrow$) Let (S, S') be a step of Γ. By (34), (Γ, A, B) with $A = \mathrm{inv}_S(\Gamma)$ and $B = \mathrm{inv}'_S(\Gamma)$ is an action of Γ. According to (60), it follows $\widetilde{S'} = (\widetilde{S} \setminus \mathrm{inv}_S(\Gamma)) \cup \mathrm{inv}'_S(\Gamma) = (\widetilde{S} \setminus A) \cup B$.

($\Leftarrow$) Let (Γ, A, B) be an action of Γ with $A \subseteq \widetilde{S}$. Then there is a state R such that $A = \mathrm{inv}_R(\Gamma)$, and $B = \mathrm{inv}'_R(\Gamma)$.

As $\mathrm{inv}_R(\Gamma) = A$ and $A \subseteq \widetilde{S}$, $\mathrm{inv}_R(\Gamma) \subseteq \widetilde{S}$. According to Lemma 15, $\Gamma_R = \Gamma_S$. By (34), $A = \mathrm{inv}_R(\Gamma) = \mathrm{inv}_S(\Gamma)$ and $B = \mathrm{inv}'_R(\Gamma) = \mathrm{inv}'_S(\Gamma)$. According to (60), (S, S') with

$$\begin{aligned} \widetilde{S'} &= (\widetilde{S} \setminus \mathrm{inv}_S(\Gamma)) \cup \mathrm{inv}'_S(\Gamma) \\ &= (\widetilde{S} \setminus A) \cup B \end{aligned}$$

is a step of Γ. □

6.7 Proof of Lemma 3

Lemma 3 states that every cut of a distributed run $R = (P, T, \lessdot, \lambda)$ of a distributed ASM D is inscribed bijectively by the stores of a state S of D. Though Lemma 3 is intuitively correct, the proof is rather technical. We will only present the main steps of the proof, and leave technical details to the reader.

Proof of Lemma 3. Let C be a fixed cut of R. According to the definition of cut, $T_{<C} =_{\text{def}} \{ \ t \in T \mid \exists p \in C : t < p \ \}$ is finite. Let $t_1, \ldots, t_n$ be an ordering of the transitions in $T_{<C}$ such that $t_i < t_j$ for all $0 \leq i < j \leq n$ (i.e. $t_1, \ldots, t_n$ is a total ordering of the partially ordered set $T_{<C}$). Then there is a unique sequence of cuts $C_0, C_1, \ldots, C_n$ such that $C_0 = {}^\circ R$, $C_n = C$, ${}^\bullet t_i \subseteq C_{i-1}$ and $C_i = C_{i-1} \setminus {}^\bullet t_i \cup t_i{}^\bullet$ for $i = 1, \ldots, n$.

We prove by induction that, for $i = 0, \ldots, n$, λ inscribes the places in C_i bijectively by the stores of a Σ-structure S_i:

Base: Let S_0 be the initial state of R. Then, by the definition of distributed run, C_0 is inscribed bijectively by the stores of S_0.

Step: Let λ inscribe C_{i-1} bijectively by the stores of a Σ-structure S_{i-1}. Then

$$\begin{aligned} \lambda(C_i) &= \lambda(C_{i-1} \setminus {}^\bullet t_i \cup t_i{}^\bullet) \\ &= \lambda(C_{i-1} \setminus {}^\bullet t_i) \cup \lambda(t_i{}^\bullet) \\ &= \lambda(C_{i-1}) \setminus \lambda({}^\bullet t_i) \cup \lambda(t_i{}^\bullet) \end{aligned}$$

The last step holds, as ${}^\bullet t_i \subseteq C_{i-1}$ and as λ is injective on C_{i-1}. Therefore, since $\lambda({}^\bullet t_i)$ as well as $\lambda(t_i{}^\bullet)$ is consistent, and since the locations of $\lambda({}^\bullet t_i)$ and $\lambda(t_i{}^\bullet)$ coincide, $\lambda(C_i)$ is consistent, and the locations of $\lambda(C_{i-1})$ and $\lambda(C_i)$ coincide. Hence, the stores in $\lambda(C_i)$ constitute a Σ-structure S_i.

Finally, we have to show that λ is injective on C_i. Assume that λ is not injective on C_i. Then there are two places $p, q \in C_i$ such that $p \neq q$ and $\lambda(p) = \lambda(q)$. As $C_i = C_{i-1} \setminus {}^\bullet t_i \cup t_i{}^\bullet$, $\{p, q\} \subseteq C_{i-1} \setminus {}^\bullet t_i \cup t_i{}^\bullet$. As λ is injective on $C_{i-1} \setminus {}^\bullet t_i$ and $t_i{}^\bullet$, respectively, neither $\{p, q\} \subseteq C_{i-1}$ nor $\{p, q\} \subseteq t_i{}^\bullet$. Therefore, WLOG let $p \in C_{i-1} \setminus {}^\bullet t_i$ and let $q \in t_i{}^\bullet$. As the locations of $\lambda({}^\bullet t_i)$ and $\lambda(t_i{}^\bullet)$ coincide, there is a place $r \in {}^\bullet t_i$ such that the locations of $\lambda(r)$ and $\lambda(q)$ are same. As $\lambda(q) = \lambda(p)$, the locations of $\lambda(r)$ and $\lambda(p)$ are same. Therefore, as $r, p \in C_i$ and $r \neq p$, either $\lambda(C_i)$ is inconsistent, or λ is not injective on C_i. This contradicts the induction assumption. $\square$

7 Conclusion

Sequential small-step ASMs and distributed ASMs are important classes of ASMs in the systematic framework of ASMs, as presented e.g. in [6] and in [4]. In this work we introduced a basic version of *distributed small-step ASMs* and studied its expressive power in the style of [7].

To this end, we identified *actions* to be the fundamental building blocks of the semantics of distributed small-step ASMs. In contrast to the steps of sequential ASMs, which describe *global* state changes, actions describe *local* state changes. Therefore, in case two actions act on disjoint locations, they may occur concurrently. Consistent with the definitions in [6], a *distributed run* then represents a partial order of action occurrences.

In analogy to the sequential small-step algorithms in [7], we characterized the expressive power of distributed ASMs by the class of *distributed algorithm*. This class is defined by five axiomatic requirements, some of which are very similar to the requirements in [7]. The proof of the characterization employs some of the arguments given in [7]. Nevertheless, since distributed ASMs introduce concurrency not considered in [7], some further arguments and more technical effort is required.

Our future research concentrates on more expressive semantics of distributed ASMs than the occurrence net semantics presented in this work. In [6] Gurevich introduced a version of distributed ASMs offering advanced features like concurrent access to locations, adding and removing agents, and references to agents. Our aim is to find further variants of distributed ASMs, maybe less expressive than the version in [6], and to examine their expressive power. Furthermore, we intend to study the suitability of these variants for the specification and analysis of distributed algorithms.

References

[1] Andreas Blass and Yuri Gurevich. Ordinary Interactive Small-Step Algorithms, parts I, II, III. *ACM Trans. Comput. Logic.* to appear.

[2] Andreas Blass and Yuri Gurevich. Abstract State Machines Capture Parallel Algorithms. *ACM Trans. Comput. Logic*, 4(4):578–651, 2003.

[3] E. Börger. The Origins and the Development of the ASM Method for High Level System Design and Analysis. *Journal of Universal Computer Science*, 8(1):2–74, 2002.

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

[5] R. Eschbach. A Termination Detection Algorithm: Specification and Verification. In J. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of FM'99*, volume II of *LNCS*, pages 1720–1737. Springer-Verlag, 1999.

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

[7] Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, Juli 2000.

[8] J. Huggins. Kermit: Specification and Verification. In E. Börger, editor, *Specification and Validation Methods*, pages 247–293. Oxford University Press, 1995.

[9] Wolfgang Reisig. *Petri Nets: An Introduction.* Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[10] Wolfgang Reisig. On Gurevich's Theorem on Sequential Algorithms. *Acta Informatica*, 39(5):273–305, 2003.

[11] Y. Gurevich. A New Thesis. *Abstracts, American Mathematical Society*, page 317, August 1985.

[12] Y. Gurevich and D. Rosenzweig. Partially Ordered Runs: A Case Study. In Y. Gurevich and P. Kutter and M. Odersky and L. Thiele, editor, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 131–150. Springer-Verlag, 2000.