Abstract State Machines for the Classroom
– The Basics –

Wolfgang Reisig

Institut für Informatik, Math.-Nat. Fakultät II, Humboldt-Universität zu Berlin,
Unter den Linden 6, DE 10099 Berlin, Germany,
`reisig@informatik.hu-berlin.de`

> *... we should have achieved a mathematical model of computation, perhaps highly abstract in contrast with the concrete nature of paper and register machines, but such that programming languages are merely executable fragments of the theory ...*

<div align="right">Robin Milner [17]</div>

Summary. Abstract State Machines (ASM) have been introduced as "a computation model that is more powerful and more universal than standard computation models" by Yuri Gurevich in 1985.

Here we provide a bunch of intuitive and motivating arguments, and characteristic examples for (the elementary version of) ASM. The intuition of ASM as a formal framework for an amazingly liberal notion of algorithms is highlighted. Generalizing variants of the fundamental "sequential small step" version of ASM are also considered.

Introduction

Many people find ASM difficult to understand. Most of them are conventionally educated computer scientists, hence occupied with a bunch of implicit or explicit assumptions and expectations on "yet another" specification language or computation model. ASM challenge some of those assumptions and expectations. It is this aspect that makes people struggle when trying to understand ASM. Would Computer Science education start out with ASM (and there are many good reasons to do so), people would conceive the basic ideas of ASM as the most simple and natural approach to the notion of "algorithm".

This paper addresses the conventionally educated computer scientist. To meet his or her implicit and explicit assumptions, the first part of this presentation addresses the intuition and foundations of ASM in great detail and various aspects. The second part then focuses technical details of the most elementary class of ASM. The third part considers various variants and extensions.

I – Intuition and Foundations of ASM

The first Section addresses the fundamental aspects that make ASM a technique, quite different from other techniques, to describe algorithms or, more generally, discrete systems. Without going into details, the central idea is highlighted, and the ASM approach is embedded into the context of first order logic and computable functions.

The second Section is devoted to some small examples. As the central idea of ASM is, to some extent, independent of concrete syntactical representations, we represent each example in a Pseudocode notation, as particularly

intuitive for the respective algorithm. The representation of such algorithms as syntactically correct ASM is postponed to Sect. 5. As we stick to a version of ASM in this paper that can be described by *transition systems*, we start the Section with this fundamental notion.

Section 3 starts with an investigation on the notion of states. The algorithms of Sect. 2 are use to exemplify how a program is applied to a state.

1 What Makes ASM so Unique?

1.1 A Basic Question

At the first glance, an abstract state machine is just a set of conditional assignment statements. Several extensions of the basis version of ASM have been suggested, including parallel, distributed and reactive ones. These concepts are likewise not too new. Some versions do with quantified variables, essentially with "$\forall x \ldots$" and "$\exists x \ldots$". Quantified variables usually don't appear in programming languages, but specification languages such as Z very well use quantification. What, then, makes ASM so unique? In what sense are ASM "a computation model that is more powerful and more universal than standard computation models", as Yuri Gurevich has written 1985 already [12]?

1.2 The Central Idea of ASM

The central and new idea of ASM is easily described: It is the systematic way of how symbols occurring in the syntactic representation of a program are related to the real world items of a state. In fact, a state of an ASM may include *any* real world objects and functions. In particular, the ASM approach does not assume a symbolic, bit level representation of all components of a state. Herein it differs from standard computation models – and most obviously to Turing Machines – where a state is a (structured) collection of symbols. The designer or user of a Turing Machine or any (more involved) programming or specification language may very well have in mind a particular meaning of a symbol. Examples of such symbols include "1", "$\in$", "init" or "millisecond". And a model usually makes little sense without this kind of interpretation. But conventional computation concentrates on the *transformation* of symbols, not dwelling too deeply on what they stand for.

1.3 ASM in the Context of Set Theory and Logic

ASM's manner to relate symbols to their interpretation is not new at all. One may read the ASM approach as a recommendation to just take seriously, what formal logic has revealed in the last century. Since Cantor's definition of a set to be "any collection into a whole M of definite and separate objects m of our intuition or our thought" [4], sets entered mathematics in a clear

and simple way. Tarski, in [21] suggested *structures*, including functions and predicates over real world items, as the most general mathematical framework. First order logic has been developed as a language to define and to analyze such structures. In close correspondence to this line of development, Gurevich suggests a further step, introducing *algorithms* over real world items.

1.4 ASM and Computable Functions

The above considerations rise the question of *implementation*: In fact, many algorithms are definable by ASM, but can not be implemented on a computer. Furthermore, many of them aren't even intended to be implemented. They rather describe procedures involving real world items. Examples include the algorithms to use a teller machine to withdraw money from one's bank account, or the procedure to press buttons on a highriser's walls and inside its lifts, in order to be transported to an upper floor. In any case, ASM is a specification language to describe the steps of dynamic, discrete systems. Those systems include in particular the systems that are implementable on a computer.

Computability theory characterizes the computable functions as a subset of *all* functions over the integers. Can the ASM-specifiable algorithms likewise be described as a subset of a potentially larger class of candidates? In fact, Gurevich in [13] provides such a characterization for the most elementary class of ASM. Intuitively formulated, a discrete system can be represented as a "sequential small step ASM", if the system exhibits global states and proceeds in steps from state to state, and if for each step $S \rightarrow S'$ holds: To derive S' from S, if suffices to explore a bounded amount of information of S. (Details follow in Sect. 6.3). For other classes of ASM similar characterizations exist or are under investigation.

1.5 The Future Role of ASM

In the above perspective, the theory of ASM contributes to the foundations of informatics as a scientific discipline. At the end of the day it may turn out that ASM (together with various, so far unknown equivalent notions) provide the adequate notion of "algorithms" (with the important subclass of the "implementable" algorithms, i.e. the computable functions).

2 What Kind of Algorithms Do ASM Cover?

2.1 Transition Systems

Classical system models of discrete systems assume global states and describe dynamic behaviour as steps

$$S \rightarrow S'$$

from a state S to its successor state, S'. We stick in this paper to systems with this kind of behaviour. In technical terms, we consider (initialized) *transition systems*:

A transition system

$$A = (\mathbf{states}, \mathbf{init}, F) \tag{1}$$

consists of a set **states** of *states*, **init** $\subseteq$ **states** of *initial states*, and a *next state function* $F : \mathbf{states} \rightarrow \mathbf{states}$.

A *run* of a transition system is a sequence

$$S_0 S_1 S_2 \ldots$$

of states S_i with S_0 an initial state and $S_i = F(S_{i-1})$ $(i = 1, 2, \ldots)$.

One may suggest to reduce the set **states** to the reachable ones, i.e. to those occurring in runs. But this set may be difficult to characterize. A a matter of convenience, it is frequently useful to allow a larger set of states.

The general framework of transition systems requires no specific properties of states. In particular, it is not required to represent all components of a state symbolically. The forthcoming examples of – admittedly quite simple – algorithms yield transition systems that dwell on this aspect.

This general version of transition systems is not new at all: In the first volume of his seminal opus [15], Don Knuth introduces the notion of *algorithms*. As a framework for the semantics of algorithms, Knuth suggests *computational methods*. A computational method is essentially what we called a transition system (2). Knuth additionally assumes *terminal states* t with $F(t) = t$, and calls a transition system A an *algorithm* if each run of A reaches a terminal state. The interesting aspect in Knuth's definition is that it comes without the requirement of F being "effective". Quoting [15, p8]: "F might involve operations that mortal man can not always perform". Knuth defines *effective* computational methods as a special case: A computational method is effective iff it is essentially equivalent to a Turing Machine or to any other mechanism for the computable functions. Nowadays, the term "algorithm" is usually used to denote what Knuth calls an "effective computational method".

As we did above already, we will use the term "transition system" instead of "computational method", and "effective transition system" instead of "effective computational method".

Transition systems have been generalized in several directions: Non-terminating computation sequences adequately describe behaviors of reactive systems; the next-state function F has been generalized to a relation $R \subseteq Q \times Q$, with computation sequences $x_0 x_1 \ldots$ where $(x_i, x_{i+1}) \in R$. This represents nondeterminism. Additionally one may require the choice of x_{i+1} to follow a stochastic distribution, or to be fair. Some system models describe a single behavior not as a sequence of states, but as a sequence of actions. The se-

quence orders the actions along a time axis. One may even replace the total order by a partial order, representing the cause-effect relation among actions.

All these generalizations of effective transition systems can be reduced to equivalent conventional effective transition systems, by reasonable notions of reduction and equivalence. Generalizations of this kind are intended to express algorithmic ideas more conveniently. They are not intended to challenge the established notion of effective computation.

We study non-effective transition systems in this paper. The reader may wonder whether there is anything interesting "beyond" the computable functions. In fact, there is an exciting proper subclass of all transition systems, called "Sequential Abstract-State Machines", that in turn properly contains the effective transition systems.

Yet, to communicate algorithms, we have to *represent* them somehow. We may allow for any kind of notation, as most intuitive for the respective algorithm.

Distributed systems do not canonically exhibit global states and steps. Consequently, transition systems don't adequately represent their behaviour. That kind of systems will shortly be glanced at in part III of this paper, together with some other extensions of the basic formalism.

The rest of this Section describes a series of algorithms, of which none is implementable on a computer, but each will turn out representable in the framework of ASM.

2.2 Set Extension

Let *augment* be a function with two arguments: The first argument is *any* item. The function *augment* then extends the set by the item. More precisely, for a set M and an item m, define

$$augment(M, m) =_{def} M \cup \{m\}. \qquad (2)$$

Now we intend to construct an algorithm that extends any given set M by *two* elements m and n, using the function *augment*. The idea is obvious: In a sequence of two steps, augment one element in each step. We write this idea down in the usual style of "pseudocode". To this end we introduce three variables X, x and y which in the initial state S_0 are valuated by M, m and n respectively. Then the algorithm

```
begin
      X := augment(X,x);
      X := augment(X,y);
end.
```
(3)

applied to S_0, terminates in a state S that valuates X by $M \cup \{m, n\}$. Notice that this algorithm can be applied to *any* set M and *any* elements m and n. A bit level representation of M, m and n is not required.

2.3 The Tangent Algorithm

In the geometrical plain assume a circle C with center p, and let q be a point outside C (cf. Fig. 1). Design an algorithm to construct one of the tangents of C through q . Such an algorithm is well-known from high school: First construct the point halfway between p and q. Call it r. Then construct a

Fig. 1. The problem of the tangent algorithm

circle D with center r, passing through p (and, by construction, through q). The two circles C and D intersect in two points. Pick out one of them; call it "s". The wanted tangent is the line through q and s. Figure 2 outlines this construction. Figure 3 shows a corresponding program.

Fig. 2. The solution of the tangent algorithm

This algorithm employs three sets of date items: POINTS, CIRCLES and LINES, and five basic operations:

$$\text{halfway: POINTS} \times \text{POINTS} \to \text{POINTS},$$
$$\text{circle: POINTS} \times \text{POINTS} \to \text{CIRCLES},$$
$$\text{intersect: CIRCLES} \times \text{CIRCLES} \to \mathcal{P}(\text{POINTS}),$$
$$\text{makeline: POINTS} \times \text{POINTS} \to \text{LINES},$$
$$\text{pick: } \mathcal{P}(\text{POINTS}) \to \text{POINTS}.$$

The tangent algorithm does not specify how points, circles and lines are represented, and how the operations produce their result. One choice was to represent a point as a pair of real numbers $\binom{x}{y}$, a circle by its center and its radius, and a line by any two points on it. In this case, the above four operations (6) can be defined by well established formulas, e.g.

$$\mathsf{halfway}\left(\begin{pmatrix} x_1 \\ y_1 \end{pmatrix}, \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}\right) = \begin{pmatrix} (x_1+x_2)/2 \\ (y_1+y_2)/2 \end{pmatrix}.$$

The choice from high school was to represent a point as a black dot on a white sheet of paper, a circle by its obvious curved line and a (straight) line by one of its finite sections. Each of the four above operations (6) can then be performed by pencil, rulers and a pair of compasses.

```
input(p, C, q);
if q outside C then
    r := halfway(p,q);
    D := circle(r,p);
    M := intersect(C,D);
    {|M| = 2}
    s := pick(M);
    l := makeline(q,s);
output(l);
```

Fig. 3. The tangent algorithm

Observe that the above algorithm likewise applies to three-dimensional points, with spheres replacing the circles.

2.4 The Bisection Algorithm

For continuous functions $f : \mathbb{R} \to \mathbb{R}$, the *bisection algorithm* approximates zeros, i.e. finds arguments x_0 such that $|\mathtt{f(x_0)}| < \varepsilon$ for some given bound ε. This algorithm starts with two real numbers a and b such that $f(a)$ and $f(b)$ are different from 0 and have different leading signs. While $|\mathtt{f(a)} - \mathtt{f(b)}| > \varepsilon$, two actions are executed: Firstly, the mean m of a and b is computed. Secondly, if $\mathtt{f(a)}$ and $\mathtt{f(m)}$ have different leading signs, a is set to m, otherwise b is set to m. Figure 4 outlines a typical step, and Fig. 5 shows this algorithm.
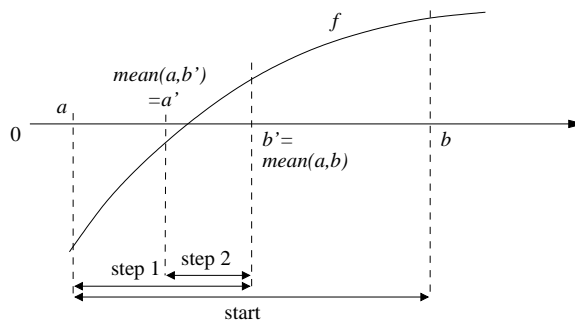
Fig. 4. Bisection step

```
while |f(a) − f(b)| < ε do
      m := mean(a,b);
      if sign(a) ≠ sign(m) then b := m
                                else a := m
```

Fig. 5. The bisection algorithm

2.5 The Halting Problem Decision Algorithm

Let $\mathcal{T}$ be the set of all Turing machines. It is well known that $\mathcal{T}$ can be enumerated, i.e. the sets $\mathcal{T}$ and $\mathbb{N}$ correspond bijectively. Let now `halt`: $\mathbb{N} \to \{0,1\}$ be defined by `halt(i) = 0` iff the i-th Turing machine terminates when applied to the empty tape. It is well known that `halt` is not computable (and this is the only reason for selecting `halt`; any other non-computable function would likewise do the job). Nevertheless, the algorithm

$$
\begin{array}{ll}
\texttt{input(i);} & \\
\texttt{b := halt(i);} & \quad(4) \\
\texttt{output(b).} &
\end{array}
$$

"computes" the function f.

2.6 A Cooking Receipt

As an – admittedly extreme – case, a cooking receipt may be considered as an algorithm, too. An example is the following receipt for Pasta Carbonara:

A: Fry the Pancetta bacon in the butter over medium-high heat until it browns.
B: In a small saucepan, heat the milk.
C: Cook the pasta until al dente. Drain well, then return pasta to pot.
D: Upon termination of A and B, add the bacon and butter to the saucepan. Stir well. Add the vinegar. Reduce heat to low and cook the sauce gently for about 15 minutes.
E: Upon termination of C and D, add the sauce, the beaten eggs, and the cheese to the pot. Stir well and serve.

The algorithm starts with three parallel branches A, B, C. A and B are single actions and C is a sequence of three actions. Upon termination, A and B trigger D. Finally, C and D trigger E.

2.7 Some General Observations

The reader may prefer a notion of "algorithm" that would exclude some of the behaviours described above, for various reasons. Certainly, none of the algorithms is implementable. For example, the bisection algorithm of Sect. 2.4 applies to any continuous function f and any real numbers a, b and ε. But only

rare cases of f, a, b and ε are representable in a real computer without causing precision problems. Yet, all of them can be handled in a formal setting. The forthcoming Part II will provide the details.

3 Representing States and Steps

In this Section we first discuss requirements for "faithful" models of algorithms. Then we see that a liberal, albeit classical framework of symbols and their interpretation yields a proper notion of states of faithful models of algorithms. Finally, we show how the symbols used in the representation of states can also be used to represent steps, and hence algorithms.

3.1 Faithful Modeling

Bound to concrete examples, we have usually a clear understanding of what an "adequate" description of an algorithm could be: It should cover all aspects one would like to emphasize and it should hide all aspects one would prefer not to mention. Formulated more precisely, a really "faithful" modeling technique represents

- each elementary object of the algorithm as an elementary object of the formal presentation,
- each elementary operation of the algorithm as an elementary operation of the formal presentation,
- each state of the algorithm as a state of the formal presentation,
- each step of the algorithm as a step of the formal presentation.

Formulated comprehensively, objects and operations, as well as states and steps of an algorithm and of its model should bijectively correspond. This is the tightest conceivable relationship of intuitive and formal presentations of algorithms.

Can this kind of faithful modeling be conceived at all? Is there a modeling technique that would achieve this goal at least for some reasonable class of algorithms? Are there some general principles to construct such models? This are the questions to be discussed in the rest of this Section.

3.2 Symbols and Their Interpretation in a State

The programs in (3) and in Figs. 3 and 5 employ symbols that stand for various items and functions. For example, X in (3) stands for any initially given set, x and y stand for any items and "augment" for a function. The algorithm is executable only after interpreting the symbol "X" by a concrete set, M, the symbols "x" and "y" by concrete items m and n and "augment" by the function *augment* that augments an element to a set. Hence, each *initial*

state of an algorithm must provide an interpretation of all symbols, except the key symbols such as *begin*, *if* etc. For example, let $\Sigma = \{X, x, y, \text{augment}\}$ be a set of symbols and let S be a state with

$$X_S = M, \; x_S = m, \; y_S = n \text{ and augment}_S = augment$$

as defined in (2).

The program (3) is applicable to this state. The first assignment statement of (3), $X := \text{augment}(X, x)$, then updates S, thus yielding a new state, S'. This state differs from S only with respect to the interpretation of X:

$$X_{S'} = X_S \cup \{m\} = M \cup \{m\}.$$

Then the second assignment statement, $X := \text{augment}(X, y)$, is executed, yielding a state S'' with

$$X_{S''} = X_{S'} \cup \{n\} = M \cup \{m\} \cup \{n\} = M \cup \{m, n\}.$$

3.3 Examples: Bisection and Tangent Algorithms Revisited

The bisection algorithm of Sect. 2.4 can be conceived according to the above schema: Based on the symbol set $\Sigma = \{a, b, \varepsilon, m, f, mean, sign, <\}$, assume an initial state S with a_S, b_S, ε_S and m_S any real numbers, $f_S : \mathbb{R} \to \mathbb{R}$ any function, $mean_S : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ with $mean_S(x, y) = (x + y)/2$, $sign_S : \mathbb{R} \to \{+, -\}$, with $sign_S(x) = +$ iff $x > 0$, and $<_S \subseteq \mathbb{R} \times \mathbb{R}$ as usual. Assuming an initial state S_0, the program of Fig. 5 generates a sequence $S_0 S_1 S_2 \ldots S_k$ of states, iteratingly updating m, a and b, with finally $|f_{S_k}(a_{S_k})| = |f_{S_0}(a_{f_k})| < \varepsilon_S$. Note that this holds for *any* real numbers a_{S_0} and ε_{S_0} and *any* unary function f_{S_0} over the real numbers.

The same procedure applies to the tangent algorithm of Sec. 2.3. Given the symbol set $\Sigma = \{C, p, q, r, s, D, M, outside, halfway, circle, intersect, pick, makeline\}$ assume an initial state S such that C_S is any circle with center p_S, and q_S a point outside C_S. Furthermore, for points a, b and circles A, B let $outside_S(a, A) = true$ if a lies outside of A, let $halfway_S(a, b)$ return the point halfway between a and b, let $circle_S(a, b)$ be the (unique!) circle, having a as its center and b on its surface, let $intersect_S(A, B)$ be the set of interesting points of A and B and $makeline_S(A, B)$ be the (unique!) line through a and b. For a set M of elements, let $pick_S(M)$ be an arbitrarily chosen element of M. The symbols q, r, s, D, M may freely be interpreted in the initial state S.

3.4 Applying a Program to a State

The above examples reveal a very simple schema: A representation P of an algorithm M consists of two kinds of symbols:

1. Key symbols such as `begin`, `:=`, `end`, `;`, `input`, `if`, `then`, `output`, `while`, `do`, `else` (in the order of their occurrence in Sect. 2),
2. constant and function symbols such as X, x, y in (3) and $C, p, r, q, \ldots$ in Fig. 3.

P can be *applied* to a *state* S, where S provides an interpretation σ_S for each constant and each function symbol σ of P. Applying P to S produces a state $P(S)$, where the interpretation of some symbols σ have been updated.

The notion of *a state of M* deserves a closer investigation: Each constant symbol and each function symbol should be interpreted by any item; virtually "everything" may serve as an interpretation. The only restriction is the *arity* of symbols: A constant symbol must be interpreted by an item, and a function symbol with arity n must be interpreted by a function of arity n. For example, in (3) the symbol "augment" has arity 2 and so every state S of this algorithm must provide a function augment$_S$ which requires two arguments. This restriction does not unduly limit the formalism: A state S (i.e. an interpretation of the constant and function symbols) that violates this restriction would spoil any attempt to define the application of P to S.

A class of such algorithms, called *sequential small step* algorithms, has been characterized by Gurevich [13]. Details will be explained in the Sect. 5.

II − The Formal Framework

The ASM approach is based on some few notions that have been identified by Tarski [21] as a most useful general conceptual basis for mathematics: The notions of *structure*, *signature*, and their combination in Σ-*structures*. Any formalism employs symbols to represent objects that in general are no symbols. Σ-structures provide the means for this kind of arguments. Section 4 presents the details. Σ-structures are the formal basis for Σ-programs, i.e. pseudocode programs over a signature Σ, as will be defined in Sect. 5, including the important subclass of sequential, small step ASM programs. Algorithms based on such programs are investigated in Sect. 6.

4 Signatures and Structures

4.1 Structures

Here we compile the algebraic prerequisits for the rest of this paper. This is merely a reminder for some elementary notions and properties, well established in the field of General Algebra.

As explained above, a state S of a program is a *structure* (sometimes also called *algebra*), consisting of

- a set U, the *universe* of S,

- finitely many *constants*, viz elements of U, and
- finitely many functions over U, shaped $\phi : U^n \to U$. n is the *arity* of ϕ.

Constants can be conceived as degenerated functions, with arity *zero*. So, a *structure S* is usually written

$$S = (U, \phi_1, \ldots, \phi_k). \tag{5}$$

With n_i the arity of the constant or function ϕ_i, the arity tuple $(n_1, \ldots, n_k)$ is the *type* of S.

4.2 Homomorphism and Isomorphism

Fundamental relationships among structures are *homomorphisms* and *isomorphisms*:

Assume two structures $R = (U_R, \psi_1, \ldots, \psi_k)$ and $S = (U_S, \phi_1, \ldots, \phi_k)$, both of the same type $(n_1, \ldots, n_k)$. Assume furthermore a mapping $h : U_R \to U_S$ such that for all $i = 1, \ldots, k$ and all $u_1, \ldots, u_{n_i} \in U_R$ holds:

$$h(\psi_i(u_1, \ldots, u_{n_i})) = \phi_i(h(u_1), \ldots, h(u_{n_i})). \tag{6}$$

Then h is a *homomorphism* from R to S, written $h : R \to S$. Figure 6 shows

$$
\begin{array}{ccc}
(u_1, \ldots, u_{n_i}) & \xrightarrow{\;\;\psi_i\;\;} & \psi_i(u_1, \ldots, u_{n_i}) \\[2pt]
\Big\downarrow{\scriptstyle h} \quad \cdots \quad \Big\downarrow{\scriptstyle h} & & \Big\downarrow{\scriptstyle h} \\[4pt]
(h(u_1), \ldots, h(u_{n_i})) & \xrightarrow{\;\;\phi_i\;\;} & \begin{array}{l} h(\psi_i(u_1, \ldots, u_{n_i})) = \\ \phi_i(h(u_1), \ldots, \phi_i(h(u_{n_i})) \end{array}
\end{array}
$$

Fig. 6. The homomorphism property

the property of an homomorphism in a diagrammatic form.

Let now R and S be structures of the same type and let $h : R \to S$ be a bijective homomorphism. Then h is called an *isomorphism*.

It is not difficult to show that the reverse function $f^{-1} : U_S \to U_R$ of an isomorphism $f : R \to S$ is again a homomorphism, $f^{-1} : S \to R$. Hence, it is reasonable to declare two structures R and S as *isomorphic*, written $R \simeq S$, if there exists an isomorphism $h : R \to S$.

4.3 Signatures and Ground Terms

The symbols occurring in a program can be collected in a *signature*. Each function symbol is associated its arity and each constant symbol is given the arity 0. A signature Σ with symbols $f_1, \ldots, f_l$ is usually written

$$\Sigma = (f_1, \ldots, f_l, a_1, \ldots, a_l) \tag{7}$$

with a_i the arity of f_i $(i = 1, \ldots, l)$. $(a_1, \ldots, a_l)$ is the *type* of Σ.

A signature Σ yields canonically the set T_Σ of *ground terms over* Σ: T_Σ is the smallest set of sequences of symbols in Σ such that

- each constant symbol in Σ is an element of T_Σ
- if $t \in \Sigma$ with arity n and if $t_1, \ldots, t_n \in T_\Sigma$ then $f(t_1, \ldots, t_n) \in T_\Sigma$.

T_Σ is apparently infinite iff Σ contains at least one constant symbol and one symbol with arity $n \geq 1$.

Ground terms typically occur on the right hand side of an assignment statement, such as in "$x := x + 1$". In the context of ASM, "x" is a constant symbol and "$+$" a function symbol of arity 2. The ground term "$+(x, 1)$" is written in the more convenient infix form "$x + 1$". First order logic employs terms with additional symbols, called *variables*. The basic version of ASM, considered here, does without variables. We will see later that ground terms in their general form may also occur as the left side of an assignment statement, i.e. an assigment may be shaped as

```
f(t_1,  ...,  t_n)  :=  ... .
```

This may be conceived as an update of an array.

4.4 Σ-Structures

A structure $S = (U, \psi_1, \ldots, \psi_k)$ of type $(n_1, \ldots, n_k)$ as in (5), "fits" to a signature $\Sigma = (f_1, \ldots, f_l, a_1, \ldots, a_l)$ as in (7) if both have the same type, i.e. if $k = l$ and $(n_1, \ldots, n_k) = (a_1, \ldots, a_l)$. In this case, S is a Σ-*structure*. The function ϕ_i is the *interpretation* of ϕ_i in S and we frequently write ϕ_i as f_{i_S}. Hence, S can be written

$$S = (U, f_{1_S}, \ldots, f_{k_S}).$$

Each term $t \in T_\Sigma$ canonically denotes an element t_S of the carrier of each Σ-structure S, defined by induction over the structure of T_Σ:

- $t_S = f_{i_S}$ if $t = f_i$ and $n_i = 0$
- $t_S = f_{i_S}(t_{1_S}, \ldots, t_{n_S})$ if $t = f(t_1, \ldots, t_n)$.

A signature Σ yields the set $str(\Sigma)$ of all Σ-*structures*. This is a rich set, including a variety of quite different structures. Vice versa, if S is a Σ-structure as well as a Σ'-structure, both signatures Σ and Σ' are identical up to bijective renaming of their symbols.

4.5 Two Lemmata on Σ-Structures

The following two Lemmata will help characterize the expressive power of ASM algorithms. The first Lemma states that the homomorphism property of Σ-structures extends to terms:

Lemma 1: *[homomorphism]*
Let Σ be a signature, let R and S be two Σ-structures, and let $h : R \rightarrow S$ be an homomorphism. Then holds $h(t_R) = t_S$ for all $t \in T_\Sigma$.

Proof: By induction over the structure of T_Σ.
First case: t is a constant symbol. Then the property holds according to the definition of homomorphism (cf. (6) in Sect. 4.2).
Second case: $t = f(t_1, \ldots, t_n)$. The inductive hypothesis implies $h(t_{i_R}) = t_{i_S}$ for $i = 1, \ldots, n$. Then, again by definition of homomorphism,

$$
\begin{aligned}
h(t_R) &= h(f(t_1, \ldots, t_n)_R) \\
&= h(f_R(t_{1_R}, \ldots, t_{n_R})) \\
&= f_S(h(t_{1_R}), \ldots, h(t_{n_R})) \\
&= f_S(t_{1_S}, \ldots, t_{n_S}) \\
&= f_S(t_1, \ldots, t_n) = t_S. \qquad \qquad \square
\end{aligned}
$$

The second lemma states that isomorphic Σ-structures can not be distinguished by help of equations over terms:

Lemma 2: *[indistinguishability]*
Let Σ be a signature, let R and S be two Σ-structures, and let $R \simeq S$. Then for all $t, u \in T_\Sigma$ holds: $t_R = u_R$ iff $t_S = u_S$.

Proof: Let $h : R \rightarrow S$ be an isomorphism. According to the above Lemma on homomorphism $\quad t_R = u_R \quad$ iff $\quad h(t_R) = h(u_R) \quad$ iff $\quad t_S = u_S$. $\qquad \square$

5 Sequential, Small Step ASM Programs

Part I provided the intuition and Sect. 4 the formal means to define syntax and semantics of a special kind of programs P: Given a signature Σ, each state S of P is just a Σ-Structure. The step-function of P, providing for each state S a successor state $P(S)$, is syntactically represented by the help of terms in T_Σ, together with some key symbols such as `if`, `then`, `:=` etc.

In this Section we define a particularly simple version of such programs. We start with assignment statements that update constants and functions. Then we proceed to sets of consistent statements and to conditional statements, and finish with "sequential, small step ASM programs". The semantics of such programs is rigorously defined in a mathematical setting.

5.1 Simple Assignment Statements

The simplest form of a program over a signature Σ is just an assignment statement, shaped

$$ f := t \tag{8} $$

with f a constant symbol in Σ and $t \in T_\Sigma$.

Applied to a Σ-structure S, (8) yields the step

$$S \xrightarrow{f:=t} S'$$

where S' updates the value of f: The constant symbol f gains t_S as a new value in S', i.e.

$$f_{S'} = t_S$$

and the semantics of all other symbols remains untouched, i.e. $g_{S'} = g_S$ for each $g \in \Sigma$, $g \neq f$. For example, with the signature

$$\Sigma = (c, f, 0, 1) \tag{9}$$

and the Σ-structure

$$S = (\mathbb{N}, 0, suc) \tag{10}$$

holds $c_S = 0$ and $f_S = suc$. The step

$$S \xrightarrow{c:=f(c)} S' \tag{11}$$

yields $c_{S'} = 1$ and $f_{S'} = suc$.

As an exercise, the reader may show that (2) updates the value of *each* term $t \in T_\Sigma$, more precisely:

$$t_S = n \text{ iff } t_{S'} = n + 1$$

for each $t \in T_\Sigma$.

5.2 Updates of Functions

The general form of assignment statements over a signature Σ is shaped

$$f(t_1, \ldots, t_n) := t, \tag{12}$$

with $f \in \Sigma$ and $t_1, \ldots, t_n, t \in T_\Sigma$, of which (8) is the special case with $n = 0$. f_S may be conceived as a n-dimensional array, to be updated for one argument tuple. A step

$$S \xrightarrow{f(t_1,\ldots,t_n):=t} S'$$

updates f_S at $(t_{1_S}, \ldots, t_{n_S})$ by t_S, yielding

$$f_{S'}(t_{1_S}, \ldots, t_{n_S}) = t_S.$$

Hence, the right hand side as well as the terms $t_1, \ldots, t_n$ denoting the arguments of the array on the left hand side are evaluated in the initial state, S. The function f remains untouched for all other arguments, i.e.

$$f_{S'}(u_1, \ldots, u_n) = f_S(u_1, \ldots, u_n)$$

for all $(u_1, \ldots, u_n) \neq (t_{1_S}, \ldots, t_{n_S})$. Likewise, the semantics of all other function symbols remains, i.e.

$$g_{S'} = g_S$$

for all $g \in \Sigma$, $g \neq f$. As an example we consider the signature Σ and the Σ-structure S as in (9) and (10). The step

$$S \xrightarrow{f(c):=c} S'$$

yields $c_S = 0$, hence with (11)

$$f_{S'}(0) = f_{S'}(c_S) = 0.$$

For all $i \geq 1$, $f_S(i)$ remains untouched, i.e.

$$f_{S'}(i) = f_S(i) = suc(i) = i + 1. \tag{13}$$

Therefore, the functions f_S and $f_{S'}$ differ only for the argument 0.

As an exercise, the reader may show that (13) updates the value of *all* terms $t \in T_\Sigma$ (except $t = c$), i.e.

$$t_{S'} = 0.$$

Summing up, in a step $S \to S'$, an assignment statement selects in S *one* constant, or *one* function at *one* argument tuple, and in S' replaces it by a new value from the universe of S. In particular, the universe of S' coincides with the universe of S.

In view of terms, an update $S \xrightarrow{f(t_1,\ldots,t_n):=t} S'$ potentially yields fresh values for all terms u which include as a subterm any term shaped $f(v_1, \ldots, v_n)$ with $(v_{1_S}, \ldots, v_{n_S}) = (t_{1_S}, \ldots, t_{n_S})$.

5.3 Consistent Assignment Statements

A step $S \to S'$ of an ASM program in general executes more than one assignment statement. This is easily achieved provided each two such assignments are *consistent*, i.e. they don't try different updates of the same constant, or of the same function at the same argument tuple. More precisely, two assignment statements $f(t_1, \ldots, t_n) := t$ and $f(u_1, \ldots, u_n) := u$ are *consistent at a state S* if

$$(t_{1_S}, \ldots, t_{n_S}) = (u_{1_S}, \ldots, u_{n_S}) \text{ implies } t_S = u_S.$$

This definition is easily generalized: A set Z of assignment statements is *consistent at a state S* if the elements of Z are pairwise consistent at S.

To define the semantics of assignment statements formally, let Σ be a signature and let Z be a set of assignment statements with terms in T_Σ. Let furthermore S be an Σ-structure and assume Z be consistent at S. Then S and Z together define a step

$$S \xrightarrow{Z} S' \qquad (14)$$

where S' is a Σ-structure, too, and the universe U of S' is identical to the universe of S. For an n-ary symbol $f \in \Sigma$ and an argument $\mathbf{u} \in U^n$ we define: In a state S, Z *updates* f_S at $\mathbf{u}$ by v in case Z includes an assignment statement shaped $f(t_1, \ldots, t_n) := t$ with $\mathbf{u} = (t_{1_S}, \ldots, t_{n_S})$, and $v = t_S$. For S' as in (14), the value of $f_{S'}(\mathbf{u})$ is now given by

$$f_{S'}(\mathbf{u}) = \begin{cases} v & \text{, in case } Z \text{ at } S \text{ updates } f_S(\mathbf{u}) \text{ by } v \\ f_S(\mathbf{u}) & \text{, otherwise.} \end{cases} \qquad (15)$$

5.4 Guards and Conditional Assignment Statements over a Signature Σ

ASM employ *conditional* assignment statements, shaped

$$\text{if } \alpha \text{ then } r, \qquad (16)$$

where r is an assignment statement and α is a boolean expression. The term α plays the role of a *guard* in (16).

For a signature Σ, the *guards over* Σ are symbol sequences such that

- for all $t, u \in T_\Sigma$, "$t = u$" is a guard over Σ and
- if α and β are guards over Σ, so are "$\alpha \wedge \beta$" and "$\neg \alpha$".

Hence, we assume each signature Σ be extended by the symbols $=, \wedge, \neg$, `true`, `false`. Each Σ-structure S is expected to interpret these symbols as usual. This implies for each guard α over Σ and each Σ-structure S,

$$\alpha_S \in \{true, false\}.$$

(16) is a *conditional assignment statement over a signature Σ* iff

- α is guard over Σ and
- r is an assignment statement over Σ, as defined in (12)

5.5 Sequential, Small Step ASM Programs and Semantics

A *sequential, small step ASM program P over a signature Σ* is a set of conditional assignment statements over Σ, as defined in Sect. 5.4. To each Σ-structure S, the program P defines a *successor structure S'*, usually written $P(S)$, by the step

$$S \xrightarrow{P} S'. \tag{17}$$

To define S', let $Z =_{def} \{r|$ ex. "if α then r" $\in P$ and $\alpha_S = true\}$. If Z is consistent at S, construct S' according to (15), otherwise let $S' = S$.

Then term "sequential" is bewildering in the face of concurrently executed statements; the term *lock step* was perhaps more intuitive. Furthermore, "small step" refers to the limited number of updates during a step: The number of updates is bounded by the number of conditional statements. Hence, the term *bounded* was perhaps more on the point. We will however follow traditions.

5.6 Simulation of Conventional Control Structures

Usual forms of programs differ from sequential small step ASM mainly w.r.t. control: Sequences, alternatives and iterations are replaced in ASM in favor of parallel execution of a set of conditional assignment statements. That ASM can simulate conventional control structures is fairly obvious; Sect. 5.7 will show some examples. Vice versa, some additional constant symbols help simulate the ASM control structure by conventional control structures, i.e. sequences, alternatives and iteration.

This kind of simulation comes however with a price: One step of an ASM program usually requires a sequence of steps in terms of conventional control structures. This price is quite high in the context of ASM, because a decisive aspect of ASM is the expressive power of their single steps, as discussed in Sect. 3.1: A sequence $S \to S'' \to S'$ of two steps from S to S' is not "as good as" the single step $S \to S'$.

5.7 Examples

Section 2 presented a couple of algorithms. We may wonder how they can be represented as ASM programs.

The set extension program of Sect. 2.2 is no ASM program at first glance: An ASM program cannot express sequential composition. This deficit is easily overcome by a well-known "trick": Extend the initial state by a fresh variable, l, and valuate l by 0 in the initial state, S_0. Reformulate (3) by

```
par     if l=0 then X := g(X,x);
        if l=0 then l := 1;
        if l=1 then X := g(X,y);
        if l=1 then l := 2
endpar.
```

The same technique can be applied to get rid of the sequential composition in the tangent algorithm in Sect. 2.3. Börger and Stärk in [3] suggested a further, elegant representation of sequential behaviour.

The bisection algorithm of Sect. 2.4, formulated as an ASM, reads

```
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)
    ∧ 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)
```

As a final 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 values of x and 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, a fresh value 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 $x_{S_0} = $ x_undef$_{S_0}$, $y_{S_0} = $ y_undef$_{S_0}$ and buffer$_{S_0} = $ b_empty$_{S_0}$, the following components define a sequential ASM program with the described behaviour:

$$
\begin{aligned}
\text{prod} &=_{def} \{ \text{ if } x=x\_undef \text{ then } x := item \} \\
\text{send} &=_{def} \text{ if } ¬(x=x\_undef) ∧ buffer=b\_empty \text{ then} \\
&\qquad\qquad \{ buffer := x; x := x\_undef \} \\
\text{rec} &=_{def} \text{ if } ¬(buffer=b\_empty) ∧ y=y\_undef \text{ then} \\
&\qquad\qquad \{ y := buffer; buffer := b\_empty \} \\
\text{cons} &=_{def} \{ \text{ if } ¬(y=y\_undef) \text{ then } y := y\_undef \}
\end{aligned}
$$

Then

$$\Gamma = \text{prod} \cup \text{send} \cup \text{rec} \cup \text{cons} \tag{18}$$

is the required sequential ASM. Its behaviour is:

$$S_0 \xrightarrow{\text{prod}} S_1 \xrightarrow{\text{send}} S_2 \xrightarrow[\text{rec}]{\text{prod}} S_3 \xrightarrow[\text{cons}]{\text{send}} S_4 \xrightarrow[\text{rec}]{\text{prod}} S_5 \quad \ldots \tag{19}$$

where each step is inscribed by the components with guards that evaluate to *true*.

6 Properties of Sequential Small Step ASM Programs

The semantics of programs as defined in Sect. 5.5 implies a series of properties of steps, to be considered here. Two of them ("steps preserve universes" and "steps respect isomorphism") are quite obvious. The third one, "exploration is bounded", is intuitively also simple, but requires a bit of formalism.

In the rest of this Section we assume a sequential small step ASM program P over a signature Σ, and a Σ-structure S.

6.1 Steps Preserve Universes

The semantics of sequential small step ASM programs, as defined in Sect. 5.5, is based on (15) and (14). There it is explicitely specified:

The universes of S and $P(S)$ coincide.

6.2 Steps Respect Isomorphism

Let R be a Σ-structure, and let $h : S \to R$ be an isomorphism. Definitions in Sect. 5.3 imply for a set Z of assignment statements, that Z is consistent at S iff Z is consistent at R. Furthermore, for each k-ary $f \in \Sigma$ and each argument tuple $(u_1, \ldots, u_k)$ for f_S, the set Z updates f_S at $(u_1, \ldots, u_k)$ by v iff Z updates f_R at $(h(u_1), \ldots, h(u_k))$ by $h(v)$. Consequently, referring to (15), $f_{P(S)}(u_1, \ldots, u_k) = v$ iff $f_{P(R)}(h(u_1), \ldots, h(u_k)) = h(v)$. Together with (17) now follows:

If $h : R \to S$ is an isomorphism, then
$h : P(R) \to P(S)$ is an isomorphism, too.

6.3 Exploration Is Bounded

To properly understand the last property, we reconsider the semantics of ASM programs, as given in (14) and (15): P only describes the *updates* of a state S, and does not care about the rest of S. In fact, the rest of S is just adopted in $P(S)$. Technically, an update of a step $S \to P(S)$ is given by three parameters: A function symbol $f \in \Sigma$, an n-tuple $\mathbf{u}$ of arguments for f_S, and the new value, v. Formulated more formally, let Σ be a signature, let f be a function symbol in Σ with arity n, let U be a universe, let $\mathbf{u} \in U^n$ and let $v \in U$. Then

$$(f, \mathbf{u}, v) \tag{20}$$

is a Σ-*update over* U. For a step $S \to P(S)$, the triple (20) may be used to indicate that $f_S(\mathbf{u})$ is updated by v.

Each step of P yields a *set* of updates. This motivates the following definition: A Σ-update $(f, \mathbf{u}, v)$ over the universe U of S is a *P-update* of S iff

$$f_S(\mathbf{u}) \neq f_{P(S)}(\mathbf{u}) = v.$$

Let

$$\triangle(P, S)$$

denote the set of all P-updates of S.

Now, let $T \subseteq T_\Sigma$ be the set of all terms occurring in P. Let R and S be two Σ-structures that T can not distinguish, i.e. for all $t \in T$

$$t_R = t_S. \tag{21}$$

Then P inevitably yields the same updates for both states:

$$\triangle(P, R) = \triangle(P, S). \tag{22}$$

7 Gurevich's Theorem on Sequential Small Step Algorithms

We are now searching for a characterization of the expressive power of sequential small step ASM. We state this problem as a question to transition systems, as considered in (1) allready.

7.1 A Question and a Partial Solution

What requirements at a transition system $A = (\mathbf{states}, \mathbf{init}, F)$ would guarantee that F can be represented as a sequential small step ASM program P as defined in Sect. 5.5?

It will turn out that essentially the properties discussed in Sect. 6 provide such a set of requirements.

The first requirement for the above question is obvious: There must exist a signature Σ such that **states** (and hence **init**) is a set of Σ-structures. The properties of sequential small step ASM, as discussed in Sect. 6, must hold for F, hence they provide another three requirements for the above question: F preserves universes, i.e. for each state $S \in$**states**, the domains of S and $F(S)$ coincide. Furthermore, F respects isomorphisms, i.e. for each $S \in$ **states** and each isomorphism $h : R \rightarrow S$, $h : F(R) \rightarrow F(S)$ is an isomorphism, too. This requires F be well defined for each R isomorphic to some $S \in$ **states**. Consequently, we require **states** be closed under isomorphism, i.e. if $S \in$ **states** and $S \simeq R$, then $R \in$ **states**. The last requirement starts out with the obvious observation that the required program P essentially does with finitely

many terms $t \in T_\Sigma$. Together with (21) and (22) this implies that there exists a finite set $T \subseteq T_\Sigma$ of terms such that two states evolve the same updates if they interpret all $t \in T$ alike.

7.2 Some Properties of Transition Systems

The above informal discussion, together with Sect. 6 is now rephrased in a more formal setting. To this end, let $A = (\textbf{states}, \textbf{init}, F)$ be a transition system.

A is *signature based* iff there exists a signature Σ such that **states** is a set of Σ-structures. If Σ is known, A is denoted as Σ-*based*.

In the rest of this Section, assume A to be Σ-*based*.

A *preserves universes* if for each $S \in \textbf{states}$, the universes of S and of $F(S)$ coincide.

A is *isomorphism closed* if for each $S \in \textbf{states}$ and each structure $R \simeq S$ holds: $R \in \textbf{states}$.

A *respects isomorphism* iff for each $S \in \textbf{states}$ and each isomorphism $h : R \to S$, $h : F(R) \to F(S)$ is an isomorphism, too.

The last property requires the following definition: For a state $S \in \textbf{states}$, an *update of A at S* is a triple $(f, \textbf{u}, v)$ with $f \in \Sigma$, $\textbf{u} \in U^k$ and $v \in U$, where k is the arity of f, U is the carrier of S and $f_S(\textbf{u}) \neq f_{F(S)}(\textbf{u}) = v$. Let $\triangle(S)$ denote the set of all updates at a state $S \in \textbf{states}$. The last property now reads:

A *bounds exploration* iff there exists a finite set $T \subseteq T_\Sigma$ of terms, such that for all $R, S \in \textbf{states}$ holds: If $t_R = t_S$ for all $t \in T_\Sigma$, then $\triangle(R) = \triangle(S)$.

A is *ASM-adapted* iff A is signature-based, A preserves universes, A is isomorphism closed, A respects isomorphism and A bounds exploration.

7.3 Gurevich's Theorem

It has been shown in Sect. 6 that the properties of Sect. 7.2 provide necessary conditions for the question of Sect. 7.1: F can be represented as a sequential small step ASM program only if A is ASM-adapted.

As an amazing and beautiful result, Gurevich in [13] has proven that this property is even sufficient! Hence,

Theorem 1 *Let $A = (\textbf{states}, \textbf{init}, F)$ be an ASM-adapted transition system. Then there exists a sequential small step ASM program P such that*

$$F = P|_{\textbf{states}}.$$

The proof of this Theorem is far from trivial. It has critically been examined in [20].

III − Extensions

Not each algorithm is sequential or small step. There are distributed, reactive, and large step algorithms. The ASM approach covers those algorithms as generalizations of the version presented in Part 3.4. We glance at some of those versions in this part.

8 Sequential Large Step ASM Algorithms

As explained in Sect. 5.5, the term "small step" refers to the limited amount of updates in each step of an ASM program P: This number is bounded by the number of assignment statements in P. Of course, there are algorithms without such a bound.

   We present an example of such an algorithm and show its representation in an extended version of ASM programs.

8.1 An Example: Node Reachability

Let G be a directed graph and let *root* be a distinguished node of G. We search for an algorithm that computes a unary predicate R on the nodes of G, to discern the nodes reachable from *root*.

   Intuitively, this algorithm operates as follows: Initially, $R(x)$ holds if and only if x ist the root. The following step is iterated until a fixpoint is reached (i.e. a state identical to its successors state): For all arcs $x \to y$ with $R(x)$ and $\neg R(y)$, extend R by y.

   The amount of updates executed in one step is unbounded: In a state S, the number of arcs $x \to y$ in G with $R(x)$ and $\neg R(y)$, is not limited. A step may even include an infinite amount of updates, in case a node has infinitely many neighbours. The algorithm can therefore not be represented by a small step ASM program.

8.2 Quantified Variables

The above node reachability algorithm can be presented with a standard technique of formal logic, viz with quantified variables. The steps of the algorithm can then be described by the program

$$
\begin{aligned}
&\textit{for all } \text{x,y } \textit{with } \text{Edge(x,y)} \land \text{R(x)} \land \neg\text{R(y)} \textit{ do}\\
&\qquad \text{R(y)} := \textit{true}.
\end{aligned}
\tag{23}
$$

(23) is a *large step* ASM program.

   An algorithm is large step not only if the amount of updates fails to be bounded, but also if the amount of involved explorations is unbounded. An

example is the following ASM program that checks whether a given graph has isolated points:

$$\texttt{if } \forall \texttt{x } \exists \texttt{y Edge(x,y) } \textit{then } \texttt{output := } \textit{false}$$
$$\textit{else } \texttt{output := } \textit{true}$$

9 Non-deterministic and Reactive ASM

So far we assume an ASM program P over a signature Σ to define a unique successor state, $P(S)$ for each state, i.e. each Σ-structure, S. This generalizes to a *set* $P(S)$ of successor states, for non-deterministic programs P. Nondeterminism can be caused by different means, considered in the sequel.

9.1 Non-deterministic Semantics

One may change the semantic rule of (15): In a state with more than one assignment statement's guard valuated to *true*, one may select one or a subset of then for execution. Though possible in principle, this idea is fairly bewildering for the reader used to the conventional approach and has therefore rarely been used.

9.2 The Operator "choose"

The *choose* operator is frequently useful. For example, let $A = \{a_1, \ldots, a_k\}$ be a set of symbols. An algorithm is to produce all symbol sequences $u \in A^*$ such that there exist $v, w \in A^*$ with

$$u = vw, \ v \neq w \ \text{ and } |v| = |w|.$$

The ASM program of Fig. 7 (with "choose" and "for all") does the job.

```
choose n,i with i<n
    choose a,b ∈ A with a≠b
        v(i) := a
        w(i) := b
    forall j<n, j≠i
        choose a,b ∈ A
            v(j) := a
            w(j) := b
```

Fig. 7. Appliance of *choose*

9.3 The Reactive Case

The last source for non-determinism is the case of the environment updating a constant f_S or a function f_S for some argument tuple $\mathbf{u}$, at a state S. This is the case of *reactive* systems.

From the perspective of a program P, a step $S \rightarrow S'$ then includes a spontaneous change of the value of f_S or $f_S(\mathbf{u})$, respectively, not caused by P. Technically, this is a non-deterministic choice from an – in general infinite – set of alternatives: An elegant method to construct reactive ASM programs. As an alternative to this kind of "inter step" interaction, an "intra step" interaction includes a system step together with a step of the environment. Details on this topic can be found e.g. in [2], [1] and many other papers published mainly by Blass and Gurevich.

9.4 Turbo Algorithms

As frequently mentioned, the faithful modeling requirement as discussed in Sect. 3.1 is sensible against the atomicity of steps. There are good reasons to squeeze more than one action, in particular communicating actions of reactive algorithms, into one step. This aspect has been addressed in many contributions including [11] and [3].

9.5 Recursive Algorithms

The quest of atomicity of actions is particularly crucial for *recursive* algorithms. A typical example was Quicksort. Faithful representation of such algorithms requires a recursive version of ASM, as introduced e.g. in [14].

10 Distributed ASM

Both small step and large step ASM algorithms describe a single *run* of an algorithm, as a sequence $S_0 S_1 \ldots$ of states S_i. This is not adequate for *distributed* algorithms. As discovered by C.A. Petri in the 1960ies already [18] and later on discussed also by Pratt [19], Lamport [16] and Gurevich [11], a run of a distributed algorithm is a partially ordered set of events, with $a > b$ iff a is causally necessary for b.

10.1 Distributed ASM programs

This gives rise to the idea of a distributed version of ASM: A *distributed ASM* is just a nonempty, finite set of ASM programs, all over the same signature, Σ. The 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.
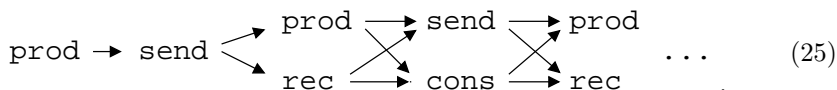
A proper definition of distributed small step ASM can be found in [10] . The general case of distributed ASM is discussed in [11].
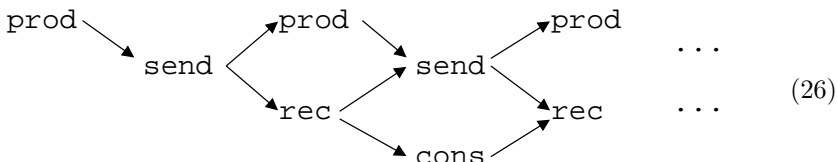
10.2 An Example of a Distributed ASM

As an example consider the producer/consumer system of Sect. 5.7: In contrast to the sequential program Γ of (18), the set

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

is a distributed ASM. Notice that (24) differs decisively from (18): 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 a notion of *distributed runs*. The order of occurrences of the component programs in the run of Γ, as depicted in (19), is

$$
\begin{array}{l}
\texttt{prod} \rightarrow \texttt{send} \quad
\begin{array}{ccc}
\texttt{prod} \rightarrow & \texttt{send} \rightarrow & \texttt{prod} \\
\texttt{rec} \rightarrow & \texttt{cons} \rightarrow & \texttt{rec}
\end{array}
\quad \ldots
\end{array} \tag{25}
$$

In contrast, the ASM program D in (24) yields a partially ordered run of occurrences of its four components, shown in (26).

$$
\begin{array}{l}
\texttt{prod} \\
\quad \texttt{send} \quad \texttt{prod} \quad \texttt{send} \quad \texttt{prod} \quad \ldots \\
\quad \quad \texttt{rec} \quad \quad \texttt{rec} \quad \ldots \\
\quad \quad \quad \texttt{cons}
\end{array} \tag{26}
$$

It is illuminating to compare the partial order of the component occurrences, as outlined in Fig. 26, with the partial order of (25): 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 (25). Figure 26 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 second production is causally before second send. This example shows that distributed ASM in fact provide a substantial generalization of sequential ASM.

11 ASM as a Specification Language

In this paper we do not even attempt to glance the huge amount of application projects of ASM. Nor do we discuss all the tools and techniques, e.g. for refinement and simulation, that support ASM as a specification language. We concentrate on one fundamental aspect only, i.e. the role of constants and functions for various purposes in an algorithm.

11.1 Static Constants and Functions

Theory does not prevent any state S to interpret the constant symbol "2" by the boolean value "true", or to interpret the function symbol "$\sqrt{}$" by the function that assigns each employee of a company his or her salary. But this is not what the reader expects. There is a number of symbols with unique world wide accepted interpretations; including the integer symbols "0", ..., "9" to construct representations of integers and rational numbers, and function symbols such as $+$, $-$, $\sqrt{}$, $^{-1}$, etc. to denote the corresponding well known functions. Some symbols have a generally agreed denotation only in distinguished communities. It is of course reasonable to inquire the initial state S of an algorithm to interpret such symbols according to their conventional denotation and never to update them. In Sect. 5.4 we remarked already the symbols for propositional logic such as "$\neg$" and "$\wedge$" must be interpreted as usual, in order to construct reasonable guards. A set of constants and functions is available this way and denoted as *static* for obvious reasons.

More generally, we denote as "static" also constants and functions that are fixed in the initial state and are never updated. They typically play the role of input to the algorithm. Typical examples are f and ε in the bisection algorithm, C and p in the tangent- and f in the halting problem decision algorithm.

Notice the generalized concept of "input" here: It may include entire functions, such as f in Fig. 5 and (4), hence, in general, infinite structures.

11.2 Constant Symbols as "Program Variables"

The non-static constants include in particular symbols which in conventional programming languages would play the role of variables: Such a symbol, x, gains some – irrelevant – value in the initial state. x is updated before being read, i.e. an assignment statement with x at the left side is executed before an assignment statement with x occurring in the term of its right side is executed. Consequently, a 0-ary symbol frequently plays the role of a program variable: A frequent source of confusion for ASM beginners. In particular the reader must not confuse this kind of constants with quantified variables as introduced for large step ASMs in Sect. 7.

11.3 Further Roles of Constants and Functions

A constant symbol occurring at the left, but never at the right side of an assignment statement, may be used as an output variable for reactive algorithms.

More generally, for the sake of clarity one may explicitly declare a constant symbols as an "output variable" in case it is assumed to be read by the environment.

12 Conclusion

This paper is intended to show that the ASM approach in deed suggests a reasonable notion of "algorithm", very adequate as a framework for the modeling of systems. Implementable systems come as the special case of states with a bit level representation for all components.

Two aspects in particular motivate the choice of ASM: Firstly, ASM fit perfectly into the framework of general algebra and logic. The notion of structure, signature, and Σ-structure are well established to describe system states in an abstract way. Computer science employs those notions in the context of Algebraic Specifications, to abstractly describe states. As a (sequential) behaviour is a sequence of states, it is very natural to describe steps in terms of Σ-structures, too.

Secondly, the definition of a sequential small step ASM as a set of simultaneously executed set of conditional statements, is very well motivated by Gurevich's Theorem, as described in Sect. 7.3.

The idea to employ mathematical structures as components of states has been advocated in [5] already: Data spaces such as stacks, trees and all forms of data structures from Algol, Lisp and Fortran, together with corresponding operations, define *virtual machines*. ASM generalizes this to *any* kind of data spaces, via algebras; [5] sticks to structures that are implementable in a canonical way.

[7] suggests to define the state of a program $\mathcal{P}$ as Σ-algebras, exactly as done in ASM. Ganzinger formally defines the semantics of $\mathcal{P}$ to be a free construct, i.e. a mapping from a set of Σ-algebras to a set of Σ-algebras. [8] expands on this idea; it may be likewise applied to ASM.

The "state-as-algebra" paradigm [8] has been a basis for various lines of research. Categorial constructs, as employed in [7] already, are likewise used in [22]. In [9], the authors show that the "state-as-algebra" paradigm is useful to describe the semantics of specification languages such as VDM, Z and B. They advocate the combination of algebraic and imperative specifications, of which ASM is an example. A further example is the "Algebraic Specification with Implicit state" approach of [6].

Modern specification techniques such as Z, TLA and FOCUS follow logic based guidelines, such as "a specification is a (huge) logical expression", "implementation (refinement) is implication", or "composition is conjunction". The ASM formalism has not been designed along these guidelines, nor does it contradict them. It might be useful to critically review those guidelines in the light of ASM.

One may very well expect a lot of related representations of algorithms to arise, in particular further variants of nondeterministic, distributed interactive and other variants, both small-step and large step, together with interesting characterizing Theorems, in analogy to Gurevich's Theorem in Sect. 7.3. On the long range, one may get used to this approach as an adequate starting point for computer sciences curricula.

13 Acknowledgements

References

1. Andreas Blass and Yuri Gurevich. Ordinary Small-Step Algorithms. *ACM Trans. Comput. Logic*, 7:2, 2006.
2. Andreas Blass, Yuri Gurevich, Dean Rosenzweig, and Benjamin Rossman. General Interactive Small-Step Algorithms. Technical report, Microsoft Research, August 2005.
3. Egon Börger and Robert Stärk. *Abstract State Machines - A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
4. Georg Cantor. *Gesammelte Abhandlungen mathematischen und philosophischen Inhalts*. Berlin: Springer-Verlag, 1932.
5. A.B. Cremers and T.N. Hibbard. Formal Modeling of Virtual Machines. *IEE on Software Engineering*, SE-4 No 5:426–436, September 1978.
6. P. Dauchy and M.-C. Gaudel. Implicit state in algebraic specifications. In *ISCORE'93*, volume No 01/93 of *Informatik-Berichte*. Universität Hannover, 1993.
7. H. Ganzinger. Programs as Transformations of Algebraic Theories. In *11. GI-Jahrestagung*, Informatik-Fachberichte 50, pages 32–41. Springer-Verlag, 1981.
8. H. Ganzinger. Denotational Semantics for Languages with Modules. In D. Bjørner, editor, *Formal Description of Programming Concepts - II*, pages 3–20. North-Holland, 1983.
9. M.-C. Gaudel and A. Zamulin. Imperative Algebraic Specifications. In *PSI'99, Novosibirsk, June 1999*, volume 1755 of *LNCS*, pages 17–39. Springer-Verlag, 2000.
10. Andreas Glausch and Wolfgang Reisig. Distributed Abstract State Machines and their Expressive Power. Technical Report 196, Humboldt-Universität zu Berlin, Institut für Informatik, Unter den Linden 6, 10099 Berlin, Germany http://www.informatik.hu-berlin.de/top, January 2006.
11. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
12. Yuri Gurevich. A new thesis. *American Mathematical Society Abstracts*, page 317, August 1985.
13. Yuri Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, Juli 2000.
14. Yuri Gurevich and Marc Spielmann. Recursive abstract state machines. *J. UCS*, 3(4):233–246, 1997.

15. Donald E. Knuth. *Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley Professional, 1973.
16. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. Reprinted in several collections, including Distributed Computing: Concepts and Implementations, McEntire et al., ed. IEEE Press, 1984.
17. Robin Milner. Software Science: From Virtual to Reality. *Bulletin of the EATCS*, (87):12–16, 2005. EATCS Award Lecture.
18. C.A. Petri. Kommunikation mit Automaten. *Schriften des Institutes für Instrumentelle Mathematik Bonn*, 1962.
19. V.R. Pratt. Modeling concurrency with partial orders. *Int. J. of Parallel Programming*, 15(1):33–71, Feb 1986.
20. Wolfgang Reisig. On Gurevich's Theorem on Sequential Algorithms. *Acta Informatica*, 39(5):273–305, 2003.
21. A. Tarski. Contributions to the theory of models I. *Indagationes Mathematicae*, 16:572–581, 1954.
22. E. Zucca. From Static to Dynamic Abstract Data-Types. In A.Szalas W.Penczek, editor, *MFCS 96*, volume 1113 of *LNCS*, pages pp 579–590, 1996.