

On the Expressive Power of Unbounded-Nondeterministic Abstract State Machines

Andreas Glausch and Wolfgang Reisig

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

Abstract. Conventional computational models assume a symbolical representation of states. Gurevich’s *Abstract State Machines (ASMs)* take a more liberal position: any mathematical structure may serve as a state. In [7] Gurevich characterizes the expressive power of *sequential ASMs*: he defines the class of *sequential algorithms* by help of only a few, amazingly general requirements and proves this class to be equivalent to sequential ASMs. In [8] this result is extended to the class of *bounded-nondeterministic ASMs*.

This paper considers the general case of *unbounded-nondeterministic ASMs*: in each step, a nondeterministic ASM may choose among infinitely many alternatives. We define the class of *nondeterministic algorithms* by a conservative extension of Gurevich’s original requirements to sequential algorithms. We show that this class is equivalent to unbounded-nondeterministic ASMs.

1 Introduction

Abstract State Machines (ASMs) have been introduced as a “computational model that is more powerful and more universal than the standard computational models” by Yuri Gurevich in 1985 [12]. This is achieved by a combination of two classic notions of computer science and mathematics: *transition systems* and *structures*.

Transition systems play a fundamental role in theoretical computer science. Usually, the operational semantics of a computational model is specified in terms of a transition system, consisting of a set of *states* and a *next-state relation*. Examples are the transition systems as generated by Turing machines, λ -expressions, CCS-expressions, Petri nets, etc. . The computational model of ASMs also fits in this setting, but differs from classical computational models in its general notion of states: each state is a *structure*, a notion well-known from first-order logic.

As usual, a structure comprises a nonempty set U (its *universe*) together with finitely many functions defined over U , each with a fixed arity.¹ No additional

¹ Usually a structure also includes relations. In the theory of ASMs, relations are represented by Boolean-valued functions, thus simplifying some technicalities.

properties are required. For example, a structure may include uncomputable functions or real-valued functions such as *sin* and *log*. Even more, a function of a structure may be defined over real-world objects having no symbolic representation at all. An example from chemistry is the function of a litmus paper which maps each chemical solution to its pH-value. *Any* of these structures may serve as a state for an ASM.

This basic idea implies several benefits. *Firstly*, a computation of an ASMs may *a priori* employ mathematical concepts like real numbers, vectors, graphs, geometrical objects, etc. . In classical computational models, such concepts usually require a particular encoding. As a consequence, ASMs are well suited to model mathematical algorithms in a simple and natural way. *Secondly*, a computation of an ASMs may involve real-world objects. For instance, the course of a chemical experiment can be modelled by an ASM whose state includes the pH-function as mentioned above. *Thirdly*, the level of abstraction of an ASM may be chosen freely. Inverting a matrix, for instance, can either be considered as an elementary operation on a higher-level ASM, or as a composed operation on a lower-level ASM.

Admittedly, these benefits come at a price: as ASMs allow for any structure, the formal analysis of a given ASM may be difficult. In particular, there are no automated analysis and verification methods that would exploit particular properties of ASMs in general. In most case studies the analysis of an ASM model is carried out manually, sometimes additionally supported by a theorem prover [5].

Despite those objections, the freedom to model states by arbitrary structures make ASMs a successfully applied design and analysis methodology [5, 4]. By stepwise refinement and composition of ASMs, large-scale systems from real-world have been modelled and analyzed formally. For example, the operational semantics of the SDL-2000 standard officially is defined by an ASM model [9], and the correctness and completeness of a Java standard compiler has been proven based on a formal ASM model [11].

2 Scope and Contribution of this Paper

Classically, ASMs describe the state transitions by a simple pseudo-code like syntax. However, it has been revealed by Gurevich that ASMs may be characterized independently of any concrete syntax: in [7] Gurevich defines a class of transition systems which he called *sequential algorithms* and proves this class to be equivalent to the class of *sequential ASMs* (c.f. also [10]). Later, Blass and Gurevich identified similar characterizations for other variants of ASMs, including bounded-nondeterministic, parallel, and interactive versions [8, 3, 2, 1].

In this paper we contribute to this work by a corresponding result for the general case of *unbounded-nondeterministic ASMs* (nondeterministic ASMs for short). Nondeterministic ASMs have been introduced in [6] as “ASMs with qualified choose”. We introduce the class of *nondeterministic algorithms*, and show that this class is equivalent to nondeterministic ASMs. We exemplify the profit

of this result by proving the reversibility of nondeterministic ASMs: for each nondeterministic ASM M there exists a nondeterministic ASM M^{-1} executing M in reverse order.

The rest of this paper is organized as follows. In the next section we present the class of nondeterministic algorithms. In Sect. 4 we exemplify and formalize nondeterministic ASMs. We close this paper by theorems on the expressive power and the reversibility of nondeterministic ASMs.

3 Nondeterministic Algorithms

In [7] Gurevich characterizes the class of *sequential algorithms* by five necessary requirements which are fairly general, nevertheless simple and intuitive:

1. A sequential algorithm consists of a set of states \mathcal{S} , a set of initial states $\mathcal{J} \subseteq \mathcal{S}$, and a next-state function $\tau : \mathcal{S} \rightarrow \mathcal{S}$.
2. Each state $S \in \mathcal{S}$ is a structure.
3. τ preserves the universe of states.
4. \mathcal{S} and \mathcal{J} are closed under isomorphism, and τ preserves isomorphism.

The decisive fifth requirement to sequential algorithm is *bounded exploration*. Intuitively, the bounded exploration requirement reads:

5. A finite set of ground terms is sufficient to characterize τ .

In this section we define the class of *nondeterministic algorithms* by five likewise intuitive requirements. The first four requirements are a canonical nondeterministic extension of Gurevich's original requirements. This extension has been presented in [8] already. The fifth requirement is new: intuitively, it requires a nondeterministic algorithm to perform a bounded amount of work in each step. In the following subsections, we present the requirements and justify their reasonability.

3.1 A Nondeterministic Algorithm Describes a Transition System

As indicated in the introduction already, the operational behaviour of an algorithm specified in a particular computation model is usually given by a transition system. Hence, transition systems appear to be the most general framework to represent behaviour of algorithms. Therefore, we assume that the behaviour of every nondeterministic algorithm can naturally be represented by a nondeterministic transition system.

Requirement N1 (states and transitions) *A nondeterministic algorithm \mathcal{N} consists of*

- a set of states $S_{\mathcal{N}}$,
- a set of initial states $I_{\mathcal{N}} \subseteq S_{\mathcal{N}}$
- a next-state relation $\rightarrow_{\mathcal{N}} \subseteq S_{\mathcal{N}} \times S_{\mathcal{N}}$.

A nondeterministic algorithm yields *sequential runs* in the obvious way: a sequential run is a sequence $S_0S_1S_2\dots$ of states where consecutive states conform to the next-state relation, i.e. $S_i \rightarrow_{\mathcal{N}} S_{i+1}$ for all indices i . A pair $(S, S') \in \rightarrow_{\mathcal{N}}$ is a *step of \mathcal{N}* .

3.2 A State of a Nondeterministic Algorithm is a Structure

As emphasized by Tarski in the early 1950ies, mathematical structures are general enough to faithfully describe any static mathematical entity on any level of abstraction. Consequently, it is legitimate to assume that every state of every algorithm can be described naturally by a structure. The second requirement will formalize this idea.

As usual, a *signature* Σ is used to address the functions of a structure: Σ consists of finitely many function symbols $\mathbf{f}_1, \dots, \mathbf{f}_k$, each \mathbf{f}_i with its arity n_i . A structure S is a Σ -*structure* if S defines for each n -ary function symbol \mathbf{f} a unique n -ary function \mathbf{f}_S , called the *interpretation of \mathbf{f} in S* .

As an algorithm always has a finite representation, an algorithm addresses only a finite set of functions. Hence, a single signature (with a finite set of symbols) suffices for all states. This leads to the second requirement:

Requirement N2 (abstract state) *For a nondeterministic algorithm \mathcal{N} holds: all states in $S_{\mathcal{N}}$ are structures over the same signature $\Sigma_{\mathcal{N}}$.*

Due to this requirement, we use the notions *state* and *structure* interchangeably in this paper. As a running example throughout this paper we consider the following structure Q , with universe $U = \{1, 2, 3\}$, consisting of two 0-ary functions \mathbf{a}_Q and \mathbf{b}_Q , and two unary functions \mathbf{v}_Q and \mathbf{next}_Q :

$$\begin{array}{llll} \mathbf{a}_Q = 1 & \mathbf{b}_Q = 2 & \mathbf{v}_Q(1) = 1 & \mathbf{next}_Q(1) = 2 \\ & & \mathbf{v}_Q(2) = 2 & \mathbf{next}_Q(2) = 3 \\ & & \mathbf{v}_Q(3) = 3 & \mathbf{next}_Q(3) = 1 \end{array}$$

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

3.3 Steps of a Nondeterministic Algorithm Preserve the Universe

The foundation of a state S is its universe U . In addition, S defines relationships between the elements in U in terms of functions over U . In this sense, the elements of U may be considered as atomic objects the state S is built upon. An algorithm cannot decompose, destroy, or create elements of U . It may only update the relationships between the elements of U .

As an example consider the Euclidian algorithm which computes the greatest common divisor of two given integers. The states of the Euclidian algorithm are built over the universe of all integers. A computation of the Euclidian algorithm does not add or remove integers from this universe. It merely computes new

relationships like “3 is the greatest common divisor of 12 and 27”. Consequently, the third requirement reads:

Requirement N3 (universe preservation) *For a nondeterministic algorithm \mathcal{N} holds: for each step (S, S') of \mathcal{N} , S and S' have the same universe.*

3.4 Steps of a Nondeterministic Algorithm Preserve Isomorphisms

As usual, we denote a bijective mapping i between the universes U_R and U_S of two Σ -structures R and S as an *isomorphism between R and S* (written $i : R \rightarrow S$) iff

$$i(\mathbf{f}_R(u_1, \dots, u_n)) = \mathbf{f}_S(i(u_1), \dots, i(u_n))$$

for all n -ary function symbols \mathbf{f} of Σ and all $u_1, \dots, u_n \in U_R$. That is, the structure S is derived from R by bijectively replacing the elements in U_R by the elements in U_S . Intuitively, isomorphic structures only differ in their concrete representation of elements, whereas the functions of both structures are essentially the same.

For an algorithm the concrete representation of elements should not matter. For example, the Euclidean algorithm computes the greatest common divisor regardless whether the integers are represented by some transistors on a chip or by ink on a paper. In general, an algorithm should not distinguish isomorphic states. It should compute isomorphic next-states at isomorphic states. This insight is formalized by the fourth requirement:

Requirement N4 (isomorphism preservation) *For a nondeterministic algorithm \mathcal{N} holds:*

- $\mathcal{S}_{\mathcal{N}}$ and $\mathcal{J}_{\mathcal{N}}$ are closed under isomorphism,
- for every two states $R, S \in \mathcal{S}_{\mathcal{N}}$, and for every isomorphism $i : R \rightarrow S$ holds: if $S \rightarrow_{\mathcal{N}} S'$, then there exists a state R' such that $R \rightarrow_{\mathcal{N}} R'$ and $i : R' \rightarrow S'$ is an isomorphism.

3.5 Steps of a Nondeterministic Algorithm Perform Bounded Work

The requirements N1–N4 are merely Gurevich’s classical first four requirements to sequential algorithms, adjusted to the nondeterministic case. The fifth requirement presented next is new.

A real-world processor (e.g., a computer, an organization, or a human being) executing an algorithm can obviously perform only a bounded amount of work in each step. For this reason it is quite natural to require an algorithm to limit the amount of work to be done in each step. In the following we formalize this vague idea.

According to the previous requirements, a step of a nondeterministic algorithm preserves the signature and the universe of the state. That is, for a step (S, S') , S and S' share the same function symbols and the same function arguments, and differ only in their function values. In order to represent such

differences formally, it turns out to be useful not to consider S as a collection of functions, but as a set of location-value-triples. A *location* of S consists of a n -ary function symbol \mathbf{f} and a n -ary argument tuple \bar{a} . For example $(\mathbf{v}, [1])$ is a location of the state Q (we enclose the argument tuple in square brackets for the sake of readability). Obviously, each location (\mathbf{f}, \bar{a}) of S defines a unique value $v = \mathbf{f}_S(\bar{a})$. The triple (\mathbf{f}, \bar{a}, v) represents a small component of S which we call an *atom of S* . For example, $(\mathbf{v}, [1], 1)$ is an atom of the state Q . Intuitively, the atom $(\mathbf{v}, [1], 1)$ states that “the function denoted by \mathbf{v} maps the argument tuple $[1]$ to the value 1”.

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

$$Q = \{ (\mathbf{a}, [], 1), (\mathbf{b}, [], 2), \\ (\mathbf{v}, [1], 1), (\mathbf{v}, [2], 2), (\mathbf{v}, [3], 3), \\ (\mathbf{next}, [1], 2), (\mathbf{next}, [2], 3), (\mathbf{next}, [3], 1) \}.$$

This representation of structures by atoms is not new. It corresponds to the *elementary diagram view* of S known from model theory. Calling them “updates”, Gurevich employed atoms already in [6] to describe differences between structures.

As announced above we intend to formalize the idea of “performing bounded work in each step”: only a bounded part of a state S should contribute to a step where the rest of S remains unaffected. The representation of S by a set of atoms permits a simple formalization of “a part of S ”: each subset $M \subseteq S$ is a *substate of S* . As an example, the set

$$M_Q = \{ (\mathbf{a}, [], 1), (\mathbf{v}, [2], 2) \}.$$

is a substate of Q .

Substates are used to describe steps that involve only a bounded part of a state: a *substep* changes a substate M by updating the values of the atoms in M . For instance, the pair (M_Q, M'_Q) with

$$M_Q = \{ (\mathbf{a}, [], 1), (\mathbf{v}, [2], 2) \}, \\ M'_Q = \{ (\mathbf{a}, [], 2), (\mathbf{v}, [2], 3) \}$$

is a substep which changes the substate M_Q to the substate M'_Q . In general, a substep is a pair of substates (M, M') such that the locations of the atoms of M and M' coincide (i.e. M and M' differ only in the values of their atoms).

We employ substeps to capture the “amount of work” performed by a step of \mathcal{N} : each step (S, S') is decomposed into a substep (M, M') and a substate E such that $(S, S') = (M \uplus E, M' \uplus E)$, where \uplus denotes disjoint union. Intuitively, the substep (M, M') describes the actual state change performed by the step whereas E describes the part of the state that is unaffected by the step (we use the letter “ E ” for “environment”). In this case, we call the step (S, S') a *completion of (M, M')* . Figure 1 shows a step (Q, Q') which is a completion of the substep (M_Q, M'_Q) .

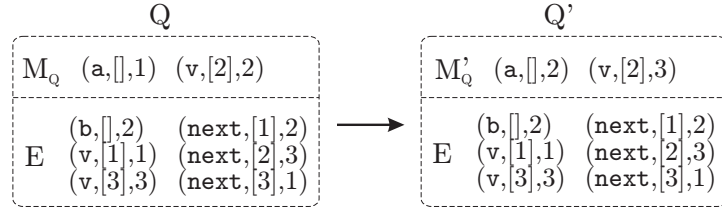


Fig. 1. A completion (Q, Q') of the substep (M_Q, M'_Q)

A “bounded amount of work” then is captured by a substep bounded in size: For a natural number k , a substep (M, M') is k -bounded iff $|M| \leq k$ (which is equivalent to $|M'| \leq k$). We are now able to formulate the final requirement stating that the steps of a nondeterministic algorithm are characterized by bounded substeps:

Requirement N5 (bounded work) *For a nondeterministic algorithm \mathcal{N} there exists a constant $k \in \mathbb{N}$ and a set \mathcal{W} of k -bounded substeps such that for all states S, S' of \mathcal{N} : (S, S') is a step of \mathcal{N} iff (S, S') is an completion of a substep in \mathcal{W} .*

As \mathcal{W} witnesses the fact that \mathcal{N} performs only a bounded amount of work in each step, we call \mathcal{W} a *bounded-work witness* of \mathcal{N} .

Though every substep in \mathcal{W} is bounded, the set \mathcal{W} itself may be infinite, even uncountably infinite. For example, consider a function symbol \mathbf{r} holding a real number: \mathcal{W} may contain uncountably many substeps that change the value of \mathbf{r} to a different real number. This points out a decisive difference to Gurevich’s original bounded-exploration requirement: a finite set of ground terms is not sufficient to characterize such rich behaviour.

We claim that the Requirements N1–N5 are intuitively necessary requirements to nondeterministic algorithms. However, we do not demand any further requirements: we call *any* entity satisfying the Requirements N1–N5 a nondeterministic algorithm.

4 Nondeterministic Abstract State Machines

In the previous section we introduced the class of nondeterministic algorithms in a purely semantical and declarative way. This may appear strange, as algorithms usually are represented in an explicit and syntactical form, e.g. by program code or by natural language. This gives rise to the question whether a given nondeterministic algorithm can be represented in a syntactical way at all: is there a language expressive enough to describe *any* nondeterministic algorithm? In the following we answer this questions positively by presenting the operational computation model of *nondeterministic ASMs*, which has been introduced in [6] already.

We start by introducing the syntax and semantics of *nondeterministic ASM rules* which divide into four rule types: *assignment rules*, *conditional rules*, *parallel rules*, and *choice rules*. The syntax of these rules is simple and intuitive, and reminds of pseudo-code. However, in contrast to pseudo-code, nondeterministic ASM rules have a formal semantics. Nondeterministic ASM rules form the syntactical basis of nondeterministic ASMs introduced afterwards.

4.1 Assignment Rules

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

Terms are used to form *assignment rules* which update a single function value of a structure. An example built from signature Σ_Q is the assignment rule R_{assign} :

$$\mathbf{v}(\mathbf{a}) := \mathbf{next}(\mathbf{b}).$$

Executing rule R_{assign} at state Q assigns to the function symbol \mathbf{v} at the argument $a_Q = 1$ the value $\mathbf{next}(\mathbf{b})_Q = 3$. The result is a new state Q' similar to Q except for the value at location $(\mathbf{v}, [1])$.

The general form of an assignment rule A is

$$\mathbf{f}(t_1, \dots, t_n) := t'$$

where t_1, \dots, t_n and t' are Σ -terms, and \mathbf{f} is a n -ary function symbol of Σ . Applied at a state S , the assignment rule A updates the value of the function symbol \mathbf{f} at the argument $\bar{a} =_{\text{def}} (t_{1S}, \dots, t_{nS})$ by the value $v =_{\text{def}} t'_S$. This update is represented by an *update atom* (\mathbf{f}, \bar{a}, v) , where v specifies the new value of location (\mathbf{f}, \bar{a}) .

In general, ASM rules may update more than a single location of S . Multiple updates are represented by a *set* of update atoms, which is called an *update set*. The update set of the assignment rule A at state S is defined as

$$A_S =_{\text{def}} \{ (\mathbf{f}, \bar{a}, v) \}$$

with \bar{a} and v as defined above.

In general, an update set U (e.g. the update set A_S) then is applied to a state S by changing the function values according to the update atoms in U : for each update atom $(\mathbf{f}, \bar{a}, v) \in U$, the value of \mathbf{f} at argument \bar{a} is changed to v . The resulting state is denoted by $S \oplus U$. Hence, the functions of $S \oplus U$ are defined as

$$\mathbf{f}_{S \oplus U}(\bar{a}) = \begin{cases} v & , \text{ for } (\mathbf{f}, \bar{a}, v) \in U \\ \mathbf{f}_S(\bar{a}) & , \text{ otherwise} \end{cases}$$

for each n -ary function symbol \mathbf{f} and each n -ary argument tuple \bar{a} .

4.2 Conditional Rules

An assignment rule may be guarded by a condition, which is represented by a *conditional rule*. For example, the conditional rule R_{cond}

$$\text{if } (a=b) \text{ then } a:=\text{next}(a)$$

executes the assignment rule at a state S only if the condition $a=b$ holds in S .

The condition may be an arbitrary *Boolean expression*. In technical terms, a Boolean expression ϕ consists of several *term equations* of the form $t_1=t_2$ connected by the usual Boolean operations \neg , \wedge , and \vee . For a given state S , the truth value of ϕ is computed in the obvious way.

The general form of a conditional rule C is

$$\text{if } \phi \text{ then } A$$

where ϕ is a Boolean expression and A is an assignment rule. Its semantics is obvious: the assignment rule A is executed if the condition ϕ is satisfied in S . Otherwise the state S is left unchanged. Hence, the update set of C at state S is defined as

$$C_S =_{\text{def}} \begin{cases} A_S & , \text{if } \phi \text{ holds in } S \\ \emptyset & , \text{otherwise.} \end{cases}$$

For technical convenience, we assume every assignment rule as a special conditional rule whose condition holds in every state.

4.3 Parallel Rules

Several conditional rules may be executed simultaneously, which is represented by a *parallel rule*. A simple example is the parallel rule R_{par}

$$\text{par } a:=b \text{ } b:=a \text{ endpar.}$$

R_{par} simultaneously executes both assignment rules (which are special conditional rules, as explained above). Executing R_{par} at a state Q yields a new state Q' where the values of a and b are swapped.

The general form of a parallel rule P is

$$\text{par } C_1 \dots C_n \text{ endpar.}$$

where C_1, \dots, C_n are conditional rules. Executing P at a state S will result in a simultaneous execution of the updates performed by C_1, \dots, C_n . Formally, the update set of P is defined as

$$P_S =_{\text{def}} C_{1S} \cup \dots \cup C_{nS}.$$

Note that P_S may be *inconsistent*, i.e. may contain two different atoms with the same location. For example, executing the parallel rule

$$\text{par } f(x):=u \text{ } f(y):=v \text{ endpar}$$

at a state S with $x_S = y_S$ and $u_S \neq v_S$ yields an inconsistent update set. An inconsistent update set cannot be applied to a state, i.e. P yields no next-state in that case. For technical convenience, we assume every conditional rule C as a parallel rule containing only C .

4.4 Choice Rules

Choice rules allow a nondeterministic choice of an element of the universe of a state. An example of a choice rule built over the signature Σ_Q is the rule R_{choice} :

`choose v do $a := v$.`

Executed at a state S , R_{choice1} nondeterministically chooses a value v from the universe of S and assigns it to a . For example, executed at state Q , the rule R_{choice1} yields three different possible next-states: the value of v may be 1, 2, or 3, each of which yielding a different next-state. However, at a state S with an infinite universe U , R_{choice1} yields an infinite number of next-states, one for each element of U . Hence, choice rules introduce *unbounded nondeterminism*.

A slightly advanced example of a choice rule is R_{choice2} :

`choose x, y with $v(x) \neq y$ do $v(x) := y$.`

R_{choice2} assigns the nondeterministically chosen value y to the function symbol v at the nondeterministically chosen argument x . The additional condition $v(x) \neq y$ restricts the possible values for x and y . In this case, the condition ensures that the newly assigned value differs from the old one. Applied at state Q , the rule R_{choice1} yields six possible next-states.

In general terms, a choice rule introduces *quantified variables* as known from first-order logic. As usual, these variables may then be used in Σ -terms analogously to 0-ary function symbols. In order to evaluate a Σ -term t containing such variables, in addition to a state S , a variable assignment α is required: α maps each variable to a value of the universe of S . We call α a *variable assignment for S* . We write $t_{S,\alpha}$ to denote the evaluation of the term t by S and α .

Terms with variables may then be used in assignment rules. The interpretation of such an assignment rule A at a state S then requires an additional variable assignment α for S . The update set of A at state S and variable assignment α is denoted by $A_{S,\alpha}$. In a similar way, the syntax and semantics of Boolean expressions, conditional rules and parallel rules are extended to include variables. We denote the update sets of a conditional rule C and a parallel rule P by $C_{S,\alpha}$ and $P_{S,\alpha}$, respectively.

The general form of a choice rule N is

`choose x_1, \dots, x_n with ϕ do P`

where x_1, \dots, x_n are variables, ϕ is a Boolean expression, and P is a parallel rule. ϕ and P both may contain the variables x_1, \dots, x_n . For a given state S , the choice rule N first nondeterministically chooses a variable assignment α for S such that ϕ is satisfied. Then P is executed by use of the variable assignment α . Consequently, the choice rule N has the potential for infinitely many possible update sets at state S . Formally, this set of possible update sets is defined as

$$N_S =_{\text{def}} \{ P_{S,\alpha} \mid \phi \text{ is satisfied by } S \text{ and the variable assignment } \alpha \}.$$

The semantics of a choice rule N built over a signature Σ may then be given in terms of a next-state relation \rightarrow_N : for two Σ -structures S, S' holds $S \rightarrow_N S'$ iff there is a consistent update set $U \in N_S$ such that $S' = S \oplus U$.

4.5 Generalized Syntax and Semantics

The syntax of ASM rules presented above is rather restricted. In the end, a choice rule merely may contain a collection of simultaneously executed assignment rules. Such a restriction definitely would make the practical application of ASM rules impossible.

To cope with this problem, the syntax and semantics of ASM rules may be extended to allow arbitrary nesting of conditional rules, parallel rules, and choice rules. In fact, ASM rules classically are defined to allow such arbitrary nesting [6, 5]. An example of such a nested ASM rule over signature Σ_Q is the following ASM rule R_{nested} :

```

par
  if (a=v(a)) then par
    choose  $x$  do a:= $x$ 
    choose  $y$  do b:= $y$ 
  endpar
  if ( $\neg$  a=v(a)) then b:=inc(b)
endpar .

```

In this paper we stick to the more simple version of ASM rules in order to keep the technical details as low as possible. Nevertheless, this restriction is not critical. The expressive power of the ASM rules introduced in this paper does not increase by allowing arbitrary nesting: a deeply nested ASM rule like R_{nested} may be canonically transformed to a single equivalent choice rule. This fact is illustrated by an analogy from first-order logic: every first-order formula containing no universal quantifier and no negation may be transformed to an equivalent first-order formula containing only a single existential quantifier.

According to their syntax, ASM rules merely seem to be yet another programming language. But in contrast to classical programs, an ASM rule R built over a signature Σ may be executed on an *arbitrary* Σ -structure S . Therefore, as explained in the introduction, R may compute on arbitrary mathematical objects, real-world items, etc. .

4.6 Nondeterministic ASMs

A nondeterministic ASM resembles a nondeterministic algorithm except for the next-state relation which is explicitly given by a choice rule. More precisely, we define a nondeterministic ASM M to consists of

- a signature Σ_M ,
- a set of Σ_M -structures \mathcal{S}_M , closed under isomorphism (the states of M),
- a set $\mathcal{J}_M \subseteq \mathcal{S}_M$, closed under isomorphism (the initial states of M),
- a choice rule N built over the signature Σ_M .

The next-state relation of M , denoted by \rightarrow_M , is the restriction of \rightarrow_N to the states of M .

Similar to nondeterministic algorithms, a nondeterministic ASM yields sequential runs: a sequential run is a sequence $S_0 S_1 S_2 \dots$ of states of M with $S_i \rightarrow_M S_{i+1}$ for all indices i .

5 The Equivalence Theorem and the Reversibility Theorem

In this section we present the main result of this paper. The following theorem states that the class of nondeterministic algorithms (as introduced in Section 3) and the class of nondeterministic ASMs (as introduced in Section 4) are equivalent:

Theorem 1. *Nondeterministic algorithms and nondeterministic ASMs describe the same set of transition systems.*

A compact proof of this theorem is presented in the next section.

Theorem 1 confirms that the notion of nondeterministic ASMs and the notion of nondeterministic algorithms may be used interchangeably. In particular, interesting properties of nondeterministic ASMs may be identified by examining nondeterministic algorithms. As an example, the following fact may be shown: for each nondeterministic algorithm \mathcal{N} , inverting the next-state relation $\rightarrow_{\mathcal{N}}$ yields another nondeterministic algorithm. Hence, each nondeterministic algorithm is reversible. This fact is shown by verifying the Requirements N1–N5 for the reverse of \mathcal{N} . This task is quite simple, as the Requirements N1–N5 are highly symmetric w.r.t. the next-state relation $\rightarrow_{\mathcal{N}}$.

According to Theorem 1, the following theorem follows immediately:

Theorem 2. *For each nondeterministic ASM M there exists a nondeterministic ASM M^{-1} such that $\rightarrow_{(M^{-1})} = (\rightarrow_M)^{-1}$.*

Alternatively, Theorem 2 can be proven without Theorem 1 by constructing for each ASM rule a corresponding reverse rule. For instance, the reverse of the ASM rule “ $\mathbf{a} := \mathbf{next}(\mathbf{a})$ ” is the ASM rule “choose x with $\mathbf{a} = \mathbf{next}(x)$ do $\mathbf{a} := x$ ”. However, the proof of Theorem 2 by Theorem 1 is considerably simpler, as there are no syntactical constructions involved.

6 Proof of The Equivalence Theorem

In this section we present a compact version of the proof of Theorem 1. The proof divides into two parts: firstly, we show that every nondeterministic algorithm can be simulated by a nondeterministic ASM. Secondly, we show that every nondeterministic ASM represents a nondeterministic algorithm.

6.1 From Nondeterministic Algorithms to Nondeterministic ASMs

Let \mathcal{N} be a nondeterministic algorithm, and \mathcal{W} be a bounded-work witness for \mathcal{N} . We have to show that there exists a nondeterministic ASM \mathcal{M} simulating \mathcal{N} .

We start with a lemma presenting a connection between completions of sub-steps and the \oplus -operator:

Lemma 1. *Let $(M, M') \in \mathcal{W}$ and let $S, S' \in \mathcal{S}_{\mathcal{N}}$. Then (S, S') is a completion of (M, M') iff $M \subseteq S$ and $S' = S \oplus M'$.*

Proof. (\Rightarrow) Let E be a substate such that $(S, S') = (M \uplus E, M' \uplus E)$. Obviously $M \subseteq S$. Further show that $S \oplus M' = (S \setminus M) \cup M'$. As $S' = E \cup M' = (S \setminus M) \cup M'$, this implies $S' = S \oplus M'$. (\Leftarrow) For $E =_{\text{def}} S \setminus M$, show that $(S, S') = (M \uplus E, M' \uplus E)$. \square

The next lemma states that parallel rules preserve isomorphisms between states. For a parallel rule P , let $\tau_P^\alpha(S) =_{\text{def}} S \oplus P_{S,\alpha}$ denote the next state computed by P at state S and variable assignment α .

Lemma 2. *Let P be a parallel rule over a signature Σ , let S, R be Σ -structures with an isomorphism $i : S \rightarrow R$, and let α be a variable assignment for S . Then $i : \tau_P^\alpha(S) \rightarrow \tau_P^{i \circ \alpha}(R)$ also is an isomorphism.*

Proof. Is proven by examining the semantics of P . \square

The following Lemma presents the main part of the proof: for each step (S, S') of \mathcal{N} , a choice rule C can be constructed such that (S, S') is a step of C , and all other steps of C are also steps of \mathcal{N} .

Lemma 3. *Let (S, S') be a step of \mathcal{N} . Then there is a choice rule C such that $S \rightarrow_C S'$, and $R \rightarrow_C R'$ implies $R \rightarrow_{\mathcal{N}} R'$ for all states R, R' of \mathcal{N} .*

Proof. By Requirement N5, there exists a substep $(M, M') \in \mathcal{W}$ such that (S, S') is a completion of (M, M') . We use (M, M') to construct the choice rule C .

Let U_S be the universe of S and let $V \subseteq U_S$ be all elements of U occurring in M and M' . For each element $v \in V$, choose a unique variable x^v . Define the Boolean expressions ϕ and ψ by

$$\phi =_{\text{def}} \bigwedge_{v \neq w \in V} x^v \neq x^w, \quad \psi =_{\text{def}} \bigwedge_{(\mathbf{f}, [u_1, \dots, u_n], v) \in M} \mathbf{f}(x^{u_1}, \dots, x^{u_n}) = x^v.$$

Construct for each atom $(\mathbf{f}, [u_1, \dots, u_n], v) \in M'$ the assignment rule $\mathbf{f}(x^{u_1}, \dots, x^{u_n}) := x^v$. Combine all of these assignment rules to a single parallel rule P . Let v_1, \dots, v_m be the elements in V . Define the choice rule C by

$$\text{choose } x^{v_1}, \dots, x^{v_m} \text{ with } \phi \wedge \psi \text{ do } P.$$

According to requirement N5, the size of M and M' is bounded by a constant k . Consequently, the size of C also is bounded by a constant c .

We first show that $S \rightarrow_C S'$: let α be the variable assignment defined by $\alpha(x^v) = v$ for all $v \in V$. Then ϕ and ψ hold at S and α . Further it holds $P_{S,\alpha} = M'$. Therefore, $S \rightarrow_C S \oplus M'$. By Lemma 1, $S \oplus M' = S'$.

Finally, let R, R' be states of \mathcal{N} such that $R \rightarrow_C R'$. We have to show that $R \rightarrow_{\mathcal{N}} R'$. As $R \rightarrow_C R'$, there is a variable assignment β such that ϕ and ψ hold at R and β , and $R' = \tau_P^\beta(R)$. Let U_R be the universe of R .

Construct a state Q from R by bijectively replacing, for each $v \in V$, the element $\beta(x^v)$ by v , and by replacing every other element from U_R by a new element not contained in U_R . As ϕ holds, this construction is well-defined. Let

$i : R \rightarrow Q$ be the corresponding isomorphism. As ψ holds, $M \subseteq Q$. By the construction of P holds $P_{Q,i\circ\beta} = M'$.

Let $Q' =_{\text{def}} \tau_P^{i\circ\beta}(Q)$. By Lemma 2, $i : R' \rightarrow Q'$ is an isomorphism. As R, R' are states of \mathcal{N} , Requirement N4 implies that Q, Q' also are states of \mathcal{N} . As $P_{Q,i\circ\beta} = M'$ and by the definition of $\tau_P^{i\circ\beta}(Q)$, it holds $Q' = Q \oplus M'$. As $M \subseteq Q$, Lemma 1 implies that (Q, Q') is a completion of (M, M') . By Requirement N5, (Q, Q') is a step of \mathcal{N} . As $i : R \rightarrow Q$ and $i : R' \rightarrow Q'$ are isomorphisms, Requirement N4 implies that (R, R') also is a step of \mathcal{N} . \square

The last lemma states that a finite set of choice rules can be united to a single choice rule:

Lemma 4. *Let C_1, \dots, C_n be choice rules. Then there exists a single choice rule C such that $\rightarrow_C = \rightarrow_{C_1} \cup \dots \cup \rightarrow_{C_n}$. C is the union of C_1, \dots, C_n .*

Proof. For the sake of simplicity, we present C in form of a nested nondeterministic ASM rule. However, as explained in Section 4, C may be transformed to a single choice rule. The desired rule C is

```

choose  $x_0, \dots, x_n$ 
with  $\bigvee_{i \in \{1, \dots, n\}} (x_0 = x_i \wedge \bigwedge_{j \in \{1, \dots, n\}, j \neq i} x_0 \neq x_j)$ 
par
  if  $(x_0 = x_1)$  then  $C_1$ 
  ...
  if  $(x_0 = x_n)$  then  $C_n$ 
endpar.

```

\square

The final proof combines the lemmata presented above:

Proof (of Theorem 1). For all steps (S, S') , derive a choice rule $C_{(S, S')}$ by applying Lemma 3. Let \mathcal{C} be the set of all of these choice rules. By construction (see proof of Lemma 3), the size of all choice rules in \mathcal{C} is bounded by a constant c . Therefore, \mathcal{C} contains only finitely many programs (up to renaming of variable symbols).

By Lemma 4, let C be the union of all choice rules in \mathcal{C} . Then for all states S, S' of \mathcal{N} holds $S \rightarrow_{\mathcal{N}} S'$ iff $S \rightarrow_C S'$: (\Rightarrow) Let $S \rightarrow_{\mathcal{N}} S'$. By Lemma 3, $S \rightarrow_{C_{(S, S')}} S'$. By Lemma 4, $S \rightarrow_C S'$. (\Leftarrow) Let $S \rightarrow_C S'$. By Lemma 4, there is a step (R, R') with $S \rightarrow_{C_{(R, R')}} S'$. By Lemma 3, $S \rightarrow_{\mathcal{N}} S'$. \square

6.2 A Nondeterministic ASM is a Nondeterministic Algorithm

Let \mathcal{M} be a nondeterministic ASM, and let C be the choice rule of \mathcal{M} . We briefly show that \mathcal{M} constitutes a nondeterministic algorithm. This task is considerably simpler than the reverse direction presented in the previous section: we only need to verify that \mathcal{M} satisfies the requirements N1–N5.

The Requirements N1 and N2 are satisfied by the definition of nondeterministic ASMs. Requirement N3 and N4 are properties of the semantics of C that are easily verified. Requirement N5 holds due to the fact that in each step (S, S') of \mathcal{M} , the choice rule C accesses and modifies only a bounded substate of S .

7 Conclusion

The theory of ASMs suggests a comprehensive and quite general approach to the notion of “algorithm”. A number of variants of ASMs have been identified, among them sequential, nondeterministic, parallel, and interactive versions. The deeper understanding of all such classes of ASMs requires a characterization of their expressive power. This has been achieved for many of them, including sequential, parallel, and interactive versions.

In this paper we characterized the expressive power of unbounded-nondeterministic ASMs. To this end we introduced the class of nondeterministic algorithms and showed that this class is captured by unbounded-nondeterministic ASMs. Surprisingly, the definition of nondeterministic algorithms turns out to be considerably simpler than the semantics of unbounded-nondeterministic ASMs. Due to this fact, we were able to prove the reversibility of unbounded-nondeterministic ASMs without any effort. Further interesting properties, possibly more subtle than reversibility, could be proven in a similar way.

References

1. A. Blass, Y. Gurevich, D. Rosenzweig, and B. Rossman. General Interactive Small Step Algorithms. Technical Report MSR-TR-2005-113, Microsoft Research, Aug 2006.
2. A. Blass and Y. Gurevich. Ordinary Interactive Small-Step Algorithms, parts I, II, III. *ACM Trans. Comput. Logic*. to appear.
3. A. Blass and Y. Gurevich. Abstract State Machines Capture Parallel Algorithms. *ACM Trans. Comput. Logic*, 4(4):578–651, 2003.
4. 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.
5. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
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, Jul 2000.
8. Y. Gurevich and T. Yavorskaya. On Bounded Exploration and Bounded Nondeterminism. Technical Report MSR-TR-2006-07, Microsoft Research, Jan 2006.
9. ITU-T. SDL Formal Semantics Definition. ITU-T Recommendation Z.100 Annex F, International Telecommunication Union, Nov 2000.
10. W. Reisig. On Gurevich’s Theorem on Sequential Algorithms. *Acta Informatica*, 39(5):273–305, 2003.
11. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.
12. Y. Gurevich. A New Thesis. *Abstracts, American Mathematical Society*, page 317, Aug 1985.