Mapping automata
Simple abstract state machines

Mapping Automata

Simple Abstract State Machines

Jörn W. Janneck
Philipp W. Kutter

TIK Report 49
June 1998

Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology (ETH) Zurich

Contents

1 Introduction

Gurevich's Abstract State Machines (GASM) as defined in [4, 5] are an elegant model of computation and provide a powerful, abstract specification method for a broad variety of specification tasks. Since they also have a rather straightforward operational semantics, they can serve as a prototyping or (combined with stepwise refinement techniques) even as an implementation base for computational systems.

The original idea of GASMs [3] was to elaborate on what Gurevich calls the implicit Turing's thesis: every algorithm is simulated by an appropriate Turing machine.[1] Unfortunately the operational semantics of algorithms given by Turing machines is often not modeling the algorithm on a natural abstraction level. While an algorithm executes just one step, the simulating Turing machine typically performs a long sequence of steps. The GASM thesis [4] is that any algorithm can be modeled at its natural abstraction level by an appropriate GASM. In short, this is achieved by combining a very flexible notion of state, i.e. Tarski's notion of structure [9], with the possibility to specify freely how much is done in one step.

Aware of the problem that neither applicability to all kinds of algorithms nor appropriateness of the reachable abstraction levels can be proven, one tried to give evidence by working out a large number of case studies [1, 6]. The variety of application areas covered by these case studies was only possible because GASM are not endowed with a specific type system. For most of the single problems, a suitable and useful type system can be thought of [11, 2], but one general solution to enhance typing of GASM has not been found. The fact that the definition of GASM has not been changed over time finally became one of the major strengths of the existing GASM-work.

In [5] Gurevich claims that in dynamic situations, it is convenient to view a state as a kind of memory that maps locations to values. A location is a pair of an r-ary function name and an r-tuple of elements. Such a memory is partitioned in different areas each consisting of the locations belonging to one function. We believe it is often more appropriate to view a state as a collection of objects, each associated with a mapping from attributes to values. Furthermore, we unify the notions of attribute, value, and object. This allows to model a large number of commonly used data structures, e.g. records with pointer attributes, arrays with dynamic length, stacks, or hash-tables.

For the moment we restrict our interest to completely untyped object systems. Such systems can be modeled with a Tarski structure having only one binary function, encoding the objects and their associated mapping. We fix the name of this function to σ. *Mapping Automaton*(MA), is a name for the combination of the above explained object-view on state with GASM whose vocabulary contains only the binary σ and a set of static constants.

In this paper we define and investigate MA as a mathematical object, by adopting the definition of GASM over mapping-structures to the MA view, i.e. the σ function is made part of the formal definition of MA states. Finally we give a formal mapping from GASM to MA.

The motivation for this work is threefold. First we want to make the MA view explicit in a formal way. Second the MA and the mapping from GASM to MA serve as implementation base for a GASM interpreter written in Java [8]. And finally the definition of MA simplifies the syntactic aspect as well as the structure of a state by removing the concept of 'signature'.

Removing signature and the induced structure from the specification language and the state, respectively, makes state and specification completely orthogonal, only connected by an interpretation of the basic syntactic constants. These constants play the role of syntax (vocabulary), which are independent from the structure of the semantics (elements, and the interpretation of σ).

In effect, any specification may be interpreted in any state (that has certain basic properties, such as being 'big' enough to allow sufficiently many objects to be allocated), which in turn means that different specifications may be interpreted on the same state.

We believe that this will allow us to compose specifications much easier than was possible in GASM, an interesting aspect of this improved compositionality possibly being the easy integration of object-based constructs into the concept with a view of making it a practical specification and prototyping method in such environments [7].

In the next section, the used static structures are described, then MA are defined formally. In section 4 the definition of transition rules is adopted to MA. In the last section the mapping from GASM to MA

[1] The implicit Turing thesis is used in the informal proof of Turing's thesis [10]

is formalized.

2 Static structures

Before we present MA as describing the dynamic transition from one state to the next, we first make precise our notion of state. For MA, this notion is completely independent of any syntactical concepts and indeed of the existence of any MA defined for it.

2.1 Abstract structure of the state

Our intuitive concept of state is that of a structure between objects of a set. This set, the set of all admissible objects that may ever occur in the computation to be modeled, we will subsequently call our *universe* $\mathcal{U}$. We will not make any assumptions about its nature, except that it be big enough (cf. section 4.5 for details on this) and contain a special element $\perp$. We will refer to the elements of $\mathcal{U}$ as *objects*.

Given such as universe we can now define our concept of state as follows: Intuitively, we may think of a state as a mapping σ, that assigns each element of $\mathcal{U}$ a unary function over $\mathcal{U}$. Many common data structures can be directly conceptualized in this way: records (mapping field names to field values), arrays (indices to values), hashtables (keys to values), etc. Of course, higher arities may be modeled by successive application of unary functions or with tuples.[2]

Alternatively, and equivalently, a state may be regarded as a mapping of pairs of objects to objects, i.e. as a two dimensional square table with objects as entries. Formally,

Definition 2.1. State space. Given a universe $\mathcal{U}$, we define the state space of $\mathcal{U}$ to be

$$\Sigma = \mathcal{U} \times \mathcal{U}$$

Note that the equation

$$(\mathcal{U} \times \mathcal{U}) \longrightarrow \mathcal{U} = \mathcal{U} \longrightarrow \mathcal{U}^{\mathcal{U}}$$

supports the alternative views of the state as either a square table populated by objects or a mapping of objects to mappings.

Since these are two equivalent manners of speaking, we will freely alternate between these two conceptions of a state, talking about a mapping associated with an object, or equivalently refer to an object as being an index to a row in the state table (assuming here and in the following that a row corresponds to a mapping).

2.2 Locations and updates

The structure of such a state is changed in one atomic action by a set of pointwise *updates*, which specify a *location* to be set to a new *value*. However, MA locations are somewhat simpler than those in GASM, since they basically specify a place in the two-dimensional position in the state table, i.e. they are a pair of objects.

Definition 2.2. Location and update. Given a universe $\mathcal{U}$, a location is a pair in $\mathcal{U}$, the set of all locations is $\Lambda = \mathcal{U} \times \mathcal{U}$. An update is a pair consisting of a location and an element in $\mathcal{U}$, the set of all updates is thus defined as $\mathbf{U} = \Lambda \times \mathcal{U}$.

Applying a set of such updates results in a new state, with the entries in the square table changed to the values given in the update set:

Definition 2.3. Application of update set. Given a state $\sigma \in \Sigma$ and an update set $\mathbf{u} \subset \mathbf{U}$, applying $\mathbf{u}$ to σ yields the successor state σ' – symbolically $\sigma \xrightarrow{\mathbf{u}} \sigma'$ – that is defined as follows:

$$\sigma'\ a\ b = \begin{cases} v & ((a,b),v) \in \mathbf{u} \\ \sigma\ a\ b & \text{otherwise} \end{cases}$$

[2] See also the discussion in section 5.2 for more details.

Clearly, the above definition only yields a well-defined function if the update set contains at most one new value for a given location. This condition is called *consistency*.

Definition 2.4. Consistency. An update set $\mathbf{u}$ is called consistent, iff

$$\forall(\lambda_1, v_1), (\lambda_2, v_2) \in \mathbf{u} : \lambda_1 = \lambda_2 \Longrightarrow v_1 = v_2$$

In the following, we assume an update set to be consistent. Since there are several possible ways of defining the effects of the application of inconsistent update sets, each with its respective merits and drawbacks, we will not commit ourselves to one particular version and choose to leave this point open for further discussion.

3 Mapping automata

Mapping Automata (MA) describe the evolution of a state as defined above. Although its structure differs slightly from GASM, where it is an algebra of a given signature, the evolution is still described by a rule, that computes an update set for a given state and the application of this update set to the state it was computed for, resulting in the successor state.

Formally, we define MA as follows:

Definition 3.1. Mapping automaton. A mapping automaton is a pair $(\mathbf{C}, \mathbf{R})$, with $\mathbf{C} = \{c_i\}$ a set of *constant symbols* and $\mathbf{R}$ a *rule*.

The constant symbols c_i are similar in function to the signature in GASM in that they serve as anchor points for interpretation and also term evaluation, as will be seen below.[3]

Such an MA is related to some state universe by an *interpretation* as follows:

Definition 3.2. Interpretation. Given a universe $\mathcal{U}$ and a mapping automaton $\mathcal{M} = (\mathbf{C}, \mathbf{R})$, we call a function $\mathcal{I} : \mathbf{C} \longrightarrow \mathcal{U}$ an interpretation of $\mathcal{M}$.

Without going into the details of how such a rule may be described (this will be the task of section 4, this is what it *does*: Given an interpretation, it computes an update set from some state. Formally,

Definition 3.3. Rule. Given an MA and an interpretation of its constant symbols, its rule $\mathbf{R}$ maps states to update sets:

$$\mathbf{R} : \Sigma \longrightarrow \mathbf{U}$$

Now we can make precise the 'dynamics' of an MA, by defining a *run* starting from some state σ:

Definition 3.4. Run. A run of an MA $(\mathbf{C}, \mathbf{R})$ starting from some initial state σ is a sequence $(\sigma_i)_{i \in \mathbb{N}}$ such that

- $\sigma_0 = \sigma$
- $\sigma_i \overset{\mathbf{R}(\sigma_i)}{\longrightarrow} \sigma_{i+1}$

Of course, a run terminates iff ex k such that $\sigma_i = \sigma_{i+1}$ for all $i > k$.

4 A rule language and its denotation

In the following we will suggest a notation for MA rules, which parallels the one suggested for GASM in [5]. Following [5], we will give the denotation of each construction in our notation in terms of the update set that it represents given an interpretation and a state – according to definition 3.3. First, however, we will develop the notion of *term*, which are basic constituents in most rule constructs.

[3] In fact, as will become clear in section 4, these symbols not only serve as constants, but also as the namespace for quantified and other variables. However, since the interpretation $\mathcal{I}$ is never updated during the execution of an MA, and since even when some variable binding shadows a constant in the scope of a rule, this at least is not destructively modified in its scope, we will stick to this name.

4.1 Terms

Terms are some kind of syntactic structure that we use to refer to objects of the universe. Some objects of the universe we can refer to directly using constant symbols and an interpretation of them. For others we form compound terms and use the state. Therefore, we will define the evaluation in a given state $\sigma \in \Sigma$ and under some interpretation $\mathcal{I}$.

MA terms are very simple structures:[4] They are either constant symbols, or pairs of terms. The latter can be intuitively thought of as signifying the application of the mapping that is bound to the value of the first term to the value of the second - which is the intuition that is responsible for the name of mapping automata.[5] Since we also need a basic predicate testing for the equality (i.e. identity) of two objects, this is also a term.

Definition 4.1. Terms. Let $\mathbf{C}$ be a set on constant symbols. Then the set of all terms $\mathcal{T}_\mathbf{C}$ of $\mathbf{C}$ is defined to be the smallest set such that

- $\mathcal{C} \subset \mathcal{T}_\mathbf{C}$
- $\mathbf{s}, \mathbf{t} \in \mathcal{T}_\mathbf{C} \Longrightarrow \langle \mathbf{s}\ \mathbf{t} \rangle \in \mathcal{T}_\mathbf{C}$
- $\mathbf{s}, \mathbf{t} \in \mathcal{T}_\mathbf{C} \Longrightarrow \mathbf{s} = \mathbf{t} \in \mathcal{T}_\mathbf{C}$

They are assigned a value in a given state in a most straightforward way: constants are mapped to their interpretation, while pairs are evaluated by applying the map associated with the first element to the value of the second, or, equivalently, simply applying the state σ to the pair of values of the two terms. The identity test is $\bot$ if the two terms to not yield the same object. If they do, however, this test must produce some other element, which we will call $\top$ here, but which has no special significance other than being different from $\bot$.

Definition 4.2. Term evaluation. Given a set of constant symbols $\mathbf{C}$. Then we define the *value* $val_{\sigma,\mathcal{I}}[t]$ of a term t in a state $\sigma \in \Sigma$ under interpretation $\mathcal{I}$ recursively as follows:

$$
\begin{aligned}
val_{\sigma,\mathcal{I}}[c] &= \mathcal{I}(c) \qquad \text{for} \quad c \in \mathbf{C} \\
val_{\sigma,\mathcal{I}}[\langle s\ t \rangle] &=_\sigma val_{\sigma,\mathcal{I}}[s]\ val_{\sigma,\mathcal{I}}[t] \\
val_{\sigma,\mathcal{I}}[s = t] &= \begin{cases} \top & val_{\sigma,\mathcal{I}}[s] = val_{\sigma,\mathcal{I}}[t] \\ \bot & \text{otherwise} \end{cases}
\end{aligned}
$$

4.2 Basic rules constructs

Now we will outline a few basic rule constructs and give their meaning by the rule they denote.

The skip construct

$$skip$$

has no effect on the state. Its denotation is accordingly the empty set for any state:

$$Den_\mathcal{I}[skip](\sigma) =_{def} \emptyset$$

The most fundamental non-empty rule construct is the single atomic update, which we denote as

$$t_1\ t_2 := t$$

Given a state σ, it denotes an update set consisting of one update:

$$Den_\mathcal{I}[t_1\ t_2 := t](\sigma) =_{def} \{((val_{\sigma,\mathcal{I}}[t_1], val_{\sigma,\mathcal{I}}[t_2]), val_{\sigma,\mathcal{I}}[t])\}$$

[4] However, see. section 4.3.2 for an extension that complicates things somewhat.

[5] Making application left-associative, one can write the term $\langle\langle a\ b\rangle\ c\rangle$ in the more familiar for $a\ b\ c$.

The conditional rule construct decides which of two rules to fire according to the value of a term:

$$\textit{if } t \textit{ then } R_1 \textit{ else } R_2 \textit{ endif}$$

Its denotation is therefore:

$$Den_{\mathcal{I}}[\textit{if } t \textit{ then } R_1 \textit{ else } R_2 \textit{ endif}](\sigma) =_{def} \begin{cases} Den_{\mathcal{I}}[R_1](\sigma) & val_{\sigma,\mathcal{I}}[t] \neq \bot \\ Den_{\mathcal{I}}[R_2](\sigma) & \text{otherwise} \end{cases}$$

We also define the parallel composition of two rule descriptions, written as[6]

$$R_1 \; R_2$$

Its denotation is simply the union of the update sets:

$$Den_{\mathcal{I}}[\; R_1 \; R_2](\sigma) =_{def} Den_{\mathcal{I}}[R_1](\sigma) \cup Den_{\mathcal{I}}[R_2](\sigma)$$

4.3 First-order extensions

As shown in [5], one can add first-order constructs to describe both rules and terms. We will start with rule constructs and then turn to first-order terms.

4.3.1 Do-forall rule

The do-forall rule construction allows to compute the update set of a rule description R with some constant symbol bound to each element of some set. Its syntax is as follows:

$$\textit{do forall } c \textit{ in } s \; : \; R \textit{ enddo}$$

c is a constant symbol, R a rule description, and s specifies the set the elements which c will be bound to in R.

Clearly, we must somehow restrict the sets that may thus be iterated upon, not only for practical reasons.[7] We choose to restrict s to constructions of the form $dom\ t$ or $ran\ t$, where t is any term. These then denote the domain and range, respectively, of the mapping associated with the value of t.[8]

Definition 4.3. Domain and range of mappings. Given an $a \in \mathcal{U}$, we define its domain and range (equivalently the domain and range of the mapping associated with it) as

$$dom_\sigma\ a =_{def} \{x \in \mathcal{U} \mid \sigma\ a\ x \neq \bot\}$$
$$ran_\sigma\ a =_{def} \{x \in \mathcal{U} \setminus \{\bot\} \mid \exists y \in \mathcal{U} : \sigma\ a\ y = x\}$$

With this, the denotation of the above set constructions becomes

$$Set_{\sigma,\mathcal{I}}[dom\ t] =_{def} dom_\sigma\ val_{\sigma,\mathcal{I}}[t]$$
$$Set_{\sigma,\mathcal{I}}[ran\ t] =_{def} ran_\sigma\ val_{\sigma,\mathcal{I}}[t]$$

Now we can define the denotation of the do-forall rule construct as the union of all updates resulting from the body for each individual element of the specified set bound to the constant symbol:

$$Den_{\mathcal{I}}[\textit{do forall } c \textit{ in } s \; : \; R \textit{ enddo}](\sigma) =_{def} \bigcup_{a \in Set_{\sigma,\mathcal{I}}[s]} Den_{\mathcal{I}[c \mapsto a]}[R](\sigma)$$

[6] Since at this point we have no notion of *blocks* as in [5], we need no *do in-parallel* syntax that except for inconsistencies, this rule notation is otherwise equivalent to.

[7] From a theoretical point of view, allowing, a rule to iterate on, say, $\mathcal{U}$ would potentially make the entire universe accessible, and thus the reserve empty – see section 4.5 for details.

[8] Further constructions might be useful here and *harmless* in the sense discussed in the previous footnote, such as a range of integers (if these are available) etc. However, without making any assumptions about the structure of $\mathcal{U}$, the above seem to be most natural.

4.3.2 First-order terms

First-order terms extend the definitions of the set $\mathcal{T}_{\mathbf{C}}$ of terms for a set of constant symbols $\mathbf{C}$ (see definition 4.1 by the following clauses, assuming $S =_{def} \{dom\ t \mid t \in \mathcal{T}_{\mathbf{C}}\} \cup \{ran\ t \mid t \in \mathcal{T}_{\mathbf{C}}\}$ the set of set-expressions:

- $c \in \mathbf{C} \wedge s \in S \wedge t \in \mathcal{T}_{\mathbf{C}} \Longrightarrow (forall\ c\ in\ s\ :\ t) \in \mathcal{T}_{\mathbf{C}}$
- $c \in \mathbf{C} \wedge s \in S \wedge t \in \mathcal{T}_{\mathbf{C}} \Longrightarrow (exists\ c\ in\ s\ :\ t) \in \mathcal{T}_{\mathbf{C}}$

The forall-term evaluates to $\top$ iff t evaluates to something else than $\perp$ for all elements of the set denoted by s bound to the symbol c, and to $\perp$ otherwise. The exists-term is $\perp$ if t is $\perp$ for all elements of that set, and $\top$ otherwise. Binding an object to a constant symbol c is tantamount to changing the interpretation at point c to this new value, which we will write as $\mathcal{I}[c \mapsto a]$.

$$val_{\sigma,\mathcal{I}}[(forall\ c\ in\ s\ :\ t)] =_{def} \begin{cases} \top & \forall a \in Set_{\sigma,\mathcal{I}}[s] : val_{\sigma,\mathcal{I}[c\mapsto a]}[t] \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

$$val_{\sigma,\mathcal{I}}[(exists\ c\ in\ s\ :\ t)] =_{def} \begin{cases} \top & \exists a \in Set_{\sigma,\mathcal{I}}[s] : val_{\sigma,\mathcal{I}[c\mapsto a]}[t] \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

4.4 Nondeterministic rules

The basic nondeterministic construction is

$$choose\ c\ in\ s\ :\ R\ endchoose$$

Intuitively, this nondeterministically selects one of the values in the set denoted by s, binds it to c and evaluates R. In order to capture this intuition we must introduce a nondeterministic denotation $NDen_{\mathcal{I}}[R](\sigma)$ of a rule description R, which is a *set* of alternative update sets. For the choose-construct above, its (nondeterministic) denotation would be as follows:

$$NDen_{\mathcal{I}}[choose\ c\ in\ s\ :\ R\ endchoose](\sigma) =_{def} \begin{cases} \{\emptyset\} & Set_{\sigma,\mathcal{I}}[s] = \emptyset \\ \bigcup_{a \in Set_{\sigma,\mathcal{I}}[s]} NDen_{\mathcal{I}[c\mapsto a]}[R](\sigma) & \text{otherwise} \end{cases}$$

Of course, we now have to give nondeterministic denotations for the other rule constructs as well, which can be done as follows:

$$NDen_{\mathcal{I}}[skip](\sigma) =_{def} \{Den_{\mathcal{I}}[skip](\sigma)\}$$

$$NDen_{\mathcal{I}}[(t_1, t_2) := t](\sigma) =_{def} \{Den_{\mathcal{I}}[(t_1, t_2) := t](\sigma)\}$$

$$NDen_{\mathcal{I}}[if\ t\ then\ R_1\ else\ R_2\ endif](\sigma) =_{def} \begin{cases} NDen_{\mathcal{I}}[R_1](\sigma) & val_{\sigma,\mathcal{I}}[t] \neq \perp \\ NDen_{\mathcal{I}}[R_2](\sigma) & \text{otherwise} \end{cases}$$

$$NDen_{\mathcal{I}}[\ R_1\ R_2](\sigma) =_{def} \{d_1 \cup d_2 \mid d_1 \in NDen_{\mathcal{I}}[R_1](\sigma) \wedge d_2 \in NDen_{\mathcal{I}}[R_2](\sigma)\}$$

$$NDen_{\mathcal{I}}[do\ forall\ c\ in\ s\ :\ R\ enddo](\sigma) =_{def} \left\{ \bigcup_{a \in Set_{\sigma,\mathcal{I}}[s]} d_a \mid d_a \in NDen_{\mathcal{I}[c\mapsto a]}[R](\sigma) \right\}$$

Except for the do-forall case (and the parallel composition case, which can be considered a special case of the former), the nondeterministic denotation is very similar to the deterministic case, except that we talk about a set of update sets. For the do-forall construct, one has to consider all combinations of nondeterministic choices at each instance of the rule and build the union over these.

The notion of a run is of course also affected by non-deterministic constructions. If a rule yields a set of update sets instead of just one, a non-deterministic run then is defined like this:

Definition 4.4. Non-deterministic run. A non-deterministic run of an MA $(\mathbf{C}, \mathbf{R})$ starting from some initial state σ is a sequence $(\sigma_i)_{i\in\mathbb{N}}$ such that

- $\sigma_0 = \sigma$
- $\sigma_i \xrightarrow{\mathbf{u}} \sigma_{i+1}$ such that $\mathbf{u} \in \mathbf{R}(\sigma_i)$

4.5 Creating new objects

Even though the universe is a static collection of objects, in specifications we often wish to refer to hitherto *unused* or fresh objects. Therefore, instead of creating new objects and extending the universe itself, we make objects that have so far been unaccessible to the MA *accessible* by picking them from a part of the universe that we could not refer to. This part, which we will make more precise below, is called our *reserve*.

4.5.1 Accessibility and allocation

We will define the set of all objects $\mathcal{U}_{\sigma,\mathcal{I}}$ (or just $\mathcal{U}_\sigma$ if the interpretation is understood) that a rule can refer to and depend on in a given state σ under and interpretation $\mathcal{I}$. The definition will inductively include all elements that can be reached by the constructions of the language, starting from the elemnts which are the interpretation of the constant symbols:

Definition 4.5. Accessibility. Given constant symbols $\mathbf{C}$, we define the set $\mathcal{U}_{\sigma,\mathcal{I}}$ of all accessible elemnts of $\mathcal{U}$ in state σ under interpretation $\mathcal{I}$ to be the smallest set such that:

- $\forall c \in \mathbf{C} : \mathcal{I}\ c \in \mathcal{U}_{\sigma,\mathcal{I}}$
- $a, b \in \mathcal{U}_{\sigma,\mathcal{I}} \Longrightarrow \sigma\ a\ b \in \mathcal{U}_{\sigma,\mathcal{I}}$
- $a \in \mathcal{U}_{\sigma,\mathcal{I}} \Longrightarrow dom_\sigma\ a \subset \mathcal{U}_{\sigma,\mathcal{I}}$
- $a \in \mathcal{U}_{\sigma,\mathcal{I}} \Longrightarrow ran_\sigma\ a \subset \mathcal{U}_{\sigma,\mathcal{I}}$

Clearly, the result of any rule cannot depend on any object and its surrounding structure that is not in $\mathcal{U}_{\sigma,\mathcal{I}}$. In this sense, the accessibility criterion is similar to the rules that govern garbage collection in programming language implementations.[9]

So in any state σ and interpretation $\mathcal{I}$, we can only talk about the accessible objects in $\mathcal{U}_{\sigma,\mathcal{I}}$. If we allow arbitrary 'construction' of new objects (as we do in the rule language in section 4), we have to provide a sufficiently large universe so that we can guarantee that we can recruit new objects from the hitherto 'unused' (i.e. irrelevant) portion of the universe, which we will call our *reserve*:

Definition 4.6. Reserve. The set $\mathcal{R} = \mathcal{U} \setminus \mathcal{U}_{\sigma,\mathcal{I}}$ is called the reserve (of state σ).

The requirement for a meaningful execution of an MA is therefore that its reserve be non-empty in any reachable state. Clearly, this rules out constructions that allow iteration and updates on the entire universe, such as

$$do\ forall\ x\ in\ \mathcal{U}\ :\ c(x) := c\ enddo$$

If c is a constant symbol interpreted as any non-$\perp$ value, applying the denotation of this rule to any state leads to a state where the entire universe becomes accessible.

Of course, the notion of accessibility is strongly connected to the constructions of the rule notation. If some constructs do not occur in a given MA, we may adapt the accessibility definition accordingly. This

[9] However, this definition of global accessibility is far too loose for many practical applications to be used as a basis for storage allocation. Consider for example a situation where $\mathbf{C}$ is the set of all integer numerals, all strings, and all identifiers. A useful interpretation will supposedly map all these infinitely many symbols to infinitely many different objects, which thus become globally accessible, while any sensible implementation will only create those number objects as they are needed during the computation process. It might make sense, therefore, to restrict the globally accessible objects for a given MA to those which can be reached by terms formulated only in constant symbols actually occurring in the MA rules. We will not further elaborate this point here.

is of particular importance when we restrict the language by imposing some kind of static structuring on the rules – then the set of visible elements in this kind of automaton may be quite different from the one we must assume for general MA. See section 5.2 for an example and an application of this principle.

4.5.2 The import-rule

Constructing the reserve in the above way allows us to give meaning to the notion of *importing* new or fresh elements into our visible part of the universe. The basic rule to pick an object from the reserve looks like this:

$$\mathit{import}\ c\ R\ \mathit{endimport}$$

This rule actually does three things: it first picks an element from the reserve, binds it to the symbol c and then executes the rule body R in the new context, i.e. in an interpretation that is identical to $\mathcal{I}$ except at point c, which is mapped to the new object instead. If we call the new object chosen from the reserve a, we can write the new interpretation as $\mathcal{I}[c \mapsto a]$, and the deterministic and non-deterministic denotation, respectively, then become

$$Den_{\mathcal{I}}[\mathit{import}\ c\ R\ \mathit{endimport}](\sigma) =_{def} Den_{\mathcal{I}[c\mapsto a]}[R](\sigma) \qquad a \in \mathcal{R}$$

$$NDen_{\mathcal{I}}[\mathit{import}\ c\ R\ \mathit{endimport}](\sigma) =_{def} NDen_{\mathcal{I}[c\mapsto a]}[R](\sigma) \qquad a \in \mathcal{R}$$

As in [5] we assume that different imports choose different reserve elements. Furthermore, we assume that for any new element a, $\sigma\ a\ x = \bot$ for all $x \in \mathcal{U}$. Note also, that the new object does not automatically become a member of $\mathcal{U}_{\sigma',\mathcal{I}}$: although it is in $\mathcal{U}_{\sigma,\mathcal{I}[c\mapsto a]}$, the rule body has to manipulate the state so that it can be accessed outside the rule in the next state.

5 Comparison to traditional ASMs

In this section we will first shed some light on what we perceive as one of the basic differences between MA and GASM, and then proceed to show their fundamental equivalence (as far as computational expressibility and level of abstraction are concerned). This will serve to document our claim that MA are basically a slightly different way of doing very similar things.

5.1 State and automata

A key difference between traditional ASMs and MA is the relation between a state (and the set of all states) and the automaton: A GASM state is always a state *of a vocabulary*, i.e. a signature containing some function names of various arities that impose a certain structure on the state. Also, an ASM operating meaningfully on this state must in a sense 'know' about this structure, i.e. share its vocabulary.

In MA, the situation is somewhat simpler. First, the a state can be meaningfully defined without any recourse to syntactical elements such as function names, or their MA-counterparts, constant symbols. A state is a simple structure imposed on the elements of some universe, indeed, there need not even be an MA, constant symbols, or any other syntactical conventions to be able to talk about a state.

However, when we want to refer to particular parts of such a structure, say, individual objects, we must have a way of identifying them so we can investigate the structure 'around' them. It was felt that the most straightforward way of doing this was to simply give them names, i.e. to provide a set of names and a mapping between these names and their denotations.

These names and their interpretation, however, to not in any way introduce a structure into the system – unlike function names of various fixed arities.[10] They are basically a flat collection of distinguishable identifications of elements in the universe. The structure, therefore, is completely separated from the naming.

This separation of concerns, leaving structure to the state and naming to the automaton (and its interpretation) that describes the evolution of such a structure, can be leveraged in various ways. For

[10] Of course, the names themselves become structured by the way they relate to the different or identical elements of the universe.

instance, there is no problem in applying several automata (each with its own interpretation and even different sets of constant symbols) to the same state - concurrently, independently, alternatively. This can be used to promote a much higher degree of compositionality of automata.

When composing a specification of a set of automata, it might make sense to require them to share the same set of constant symbols. For GASM, sharing the same signature over a large number of automata would seem like a somewhat unnatural requirement, and possibly even involve a good deal of renaming, prefixing, etc. to actually make it work, but for MA this might be a sensible choice for the standard case: for instance, a conceivable set of constant symbols could consist of all identifiers plus all representations of some primitive data types, such as numbers and strings.

5.2 Equivalence of MA and traditional ASM

In this section we show how to map a GASM into an MA and vice versa. The translation from MA to GASM is already given by the fact that MA are defined as a GASM with a special kind of structure. The translation from GASM into MA allows to use the MA tool for GASM tool support, since the translation does not change the abstraction level. In fact the translation deals only with some semantical details, e.g. the adaption of the different views on boolean and relations, and the modeling of n-ary functions with tuples.

Before we start describing the translation between GASM to MA we remember the different ways booleans and partial functions are treated. In GASM booleans are modeled by two distinct elements *true* and *false* and partial functions are modeled by mapping to a third element *undef*. The carrier set of each GASM needs thus at least three distinct elements, *true*, *false*, and *undef*. Differently, in MA exist only two distinct elements, called bottom $\bot$ and top $\top$. $\bot$ is used for partial functions, and as interpretation of false, *true* is represented by $\top$ or any other element in the carrier set. Both GASM and MA are not strict.

Mapping a GASM state into an MA state. In general the universe $\mathcal{U}$ of objects in a MA consist of at least two elements, one denoted by $\bot$ and the other by $\top$. Since the GASM super-universe S contains at least three elements (*true*, *false*, and *undef*) we need to start with a $\mathcal{U}$ containing a third element. The set of constant symbols $\mathbf{C}$ of an MA modeling a GASM contains at least the three constants *true*, *false*, and *undef*, and each interpretation $\mathcal{I}$ maps *undef* to the element $\bot$, *true* to the element *top*, and *false* to the third default element in $\mathcal{U}$. We will no more make a difference between the symbols { *undef*, *true*, *false* } and the tree objects representing them, and for our convenience.

Tuples are modeled in MA by free generated elements with a static mapping as follows:

- the associated mapping of the 0-ary tuple () is given by:

$$\langle () \ t \rangle \equiv (t)$$

  where (t) is the free generated one-tuple.

- the associated mapping of a one-tuple is given by:

$$\langle (t_1) t_2 \rangle \equiv (t_1, t_2)$$

  where (t_1, t_2) is a free generated two-tuple.

- for each n ≥ 1 the mapping of an n-tuple is given by:

$$\langle (t_1, \ldots, t_n) t_{n+1} \rangle \equiv (t_1, \ldots, t_n, t_{n+1})$$

If mapping a concrete GASM A into a MA B, all elements of S are included into $\mathcal{U}$ and all symbols of the vocabulary of A are included into the constant symbols $\mathbf{C}$ of B, and for each of them a new element being its interpretation is included into $\mathcal{U}$. In other words, $\mathcal{U}$ consists of the disjoint union of $\{\bot, \top, \mathit{false}\}$, the super-universe S, the elements interpreting the GASM functions, and the above introduced tuples.

We need to make a case distinction between functions and relations in GASM. The interpretation of each n-ary *function* f in structure A, i.e. f^A, is reflected in $B's$ interpretation of σ, i.e. σ^B:

$$(f^A(o_1,\ldots,o_n) = o_0) \Leftrightarrow (\sigma^B\ \mathcal{I}(f)\ (o_1,\ldots,o_n) = o_0)$$

An n-ary *relation* r in a GASM is returning either *true* or *false*. To make everything fit together we reflect the interpretation of each r as follows:

$$\begin{array}{rcl} (r^A(o_1,\ldots,o_n) = \mathit{false}) & \Leftrightarrow & (\sigma^B\ \mathcal{I}(r)\ (o_1,\ldots,o_n) = \bot) \\ & \wedge & \\ (r^A(o_1,\ldots,o_n) = \mathit{true}) & \Leftrightarrow & (\sigma^B\ \mathcal{I}(r)\ (o_1,\ldots,o_n) = \top) \end{array}$$

Now we need two different wrappings. One is needed to get back the original *true*,*false* results of a relational term. The second is needed to map such results back into the $\bot,\top$ model in MA.

Lets thus assume two constants W_1 and W_2 such that:

$$\begin{array}{rcll} \langle W_1\ \bot\rangle & \equiv & \mathit{false} & \\ \langle W_1\ x\rangle & \equiv & x, & \mathit{where}\ x \neq \bot \\ \langle W_2\ \mathit{false}\rangle & \equiv & \bot & \\ \langle W_2\ x\rangle & \equiv & x, & \mathit{where}\ x \neq \mathit{false} \end{array}$$

For equality, the usual MA equality can be used, the logical operations in GASM are mapped into MA like normal binary relations.

Remark on reachability of course the mappings associated with the tuples and the wrappings W_1 and W_2 must be excluded from the definition of reachability.

Mapping a GASM rule into an MA rule We define now a transformation $\mathcal{T}$ from GASM rules to MA rules. For notational convenience we leave away the $\langle$ and $\rangle$ whenever the situation is clear.

Terms For all function symbols f, the subterms must be transformed:

$$\mathcal{T}[f(t_1,\ldots,t_n)] =_{def} f(\mathcal{T}[t_1],\ldots,\mathcal{T}[t_n])$$

For all relation symbols r, in addition the term is wrapped with W_1:

$$\mathcal{T}[r(t_1,\ldots,t_n)] =_{def} \langle W_1\ \langle r\ (\mathcal{T}[t_1],\ldots,\mathcal{T}[t_n])\rangle\rangle$$

Updates For all function symbols f, the subterms must be transformed::

$$\mathcal{T}[f(t_1,\ldots,t_n) := t_0] =_{def} \mathcal{T}[f(t_1,\ldots,t_n)] := \mathcal{T}[t_0]$$

For all relation symbols r, in addition the righ-hand-side is wrapped with W_2:

$$\mathcal{T}[r(t_1,\ldots,t_n) := t_0] =_{def} \mathcal{T}[r(t_1,\ldots,t_n)] := \langle W_2\ \mathcal{T}[t_0]\rangle$$

Conditional

$$\mathcal{T}[\mathit{if}\ c\ \mathit{then}\ R_1\ \mathit{else}\ R_2\ \mathit{endif}] =_{def} \mathit{if}\ \langle W_2\ c\rangle\ \mathit{then}\ \mathcal{T}[R_1]\ \mathit{else}\ \mathcal{T}[R_2]\ \mathit{endif}$$

Do forall

$$T[do\ forall\ i\ in\ I \quad Rule\ enddo]$$
$$=_{def}$$
$$do\ forall\ i\ in\ dom\ I \quad \mathcal{T}[Rule]\ enddo$$

Choose

$$T[choose\ i\ in\ I \quad Rule\ endchoose]$$
$$=_{def}$$
$$choose\ i\ in\ dom\ I \quad \mathcal{T}[Rule]\ endchoose$$

References

[1] E. Börger and J.K. Huggins. Abstract state machines 1988-1998: A commented asm bibliography. *EATCS Bulletin*, (64):105–127, 1998.

[2] G. Del Castillo, Y. Gurevich, and K. Stroetmann. Typed abstract state machines. *J.UCS*, 1998. to appear.

[3] Y. Gurevich. Logic and the challenge of computer science. In E. Böerger, editor, *Current Trends in Theoretical Computer Science*, pages 1 – 57. CS Press, 1988.

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

[5] Y. Gurevich. May 1997 draft of the asm guide. Technical Report CSE-TR-336-97, EECS Dept., University of Michigan, 1997.

[6] J.K. Huggins. Abstract state machines web page. http://www.eecs.umich.edu/gasm.

[7] J.W. Janneck. Object-based mapping automata - reference manual. Technical report, Institute TIK, ETH Zürich.

[8] J.W. Janneck. Object-based mapping automata home page. http://www.tik.ee.ethz.ch/ janneck/OMA.

[9] A. Tarsky. Der wahrheitsbegriff in den formalisierten sprachen. *Studia Philosophica*, (1):261–405, 1936. English translation in A. Tarsky. Logic, Semantics, Methamathematics. Oxford University Press.

[10] A.M. Turing. On computable numbers with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, (12):230–265, 1937. correction, *ibid*, No. 13(1937), 511–516.

[11] A. Zamulin. Typed Gurevich Machines Revisited. Joint CS & IIS Bulletin, Computer Science, 1997.