Abstract State Machines: A Unifying View of Models of Computation and of System Design Frameworks

Egon Börger

Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy, boerger@di.unipi.it

Abstract

We capture the principal models of computation and specification in the literature by a uniform set of transparent mathematical descriptions which—starting from scratch—provide the conceptual basis for a comparative study [1].

1 Introduction

Since the pioneering work of Post, Turing, Church, Gödel, Herbrand and many others to capture the intuitive notion of algorithm, and with the advent of computer programming numerous models of computation and system design frameworks have been developed to support the construction and analysis of complex algorithmic systems. In this paper we capture the principal models of computation and specification in the literature in a systematic way by a uniform set of transparent mathematical descriptions. Starting from scratch they provide the conceptual basis for a comparative investigation of different specification methods to technically clarify their merits and drawbacks. As a by-product the descriptions constitute a useful definitional framework for teaching, providing in particular a small set of definitions which unravel the basic common structure of the myriad of different machine concepts which are studied in computation theory. The goal of this paper is not to prove a theorem, but to formulate a new set of definitional suggestions for computation theory [44] which covers also system specification frameworks. The exposition is driven by systematic and not by historical considerations, for detailed historical references we refer the reader to [14] or to the update in [23, Ch.9].

[1] Helmut Schwichtenberg gewidmet, dem Freund und Kollegen seit der gemeinsamen Studienzeit in Münster

Our basis are Abstract State Machines, briefly reviewed in Section 2, which were defined in [32] and became an integral part of the powerful high-level system design and analysis method explained in the AsmBook [23]. Elaborating upon [13] we provide here for representative algorithmic systems a uniform set of simple ASM definitions which are faithful to the basic intuitions and concepts of each investigated system. The particularly natural, 'coding-free' way in which ASMs allow one to capture other computation models and thus turn out to be 'universal' contrasts with the difficulties one usually encounters when trying to reverse the simulation with a definition of ASMs in other computational frameworks. This is related to the universality claim known as 'ASM thesis' and analysed by Blass and Gurevich in [33,8].

The ASM models we construct in this paper are however different from the ones which come out of the proofs for the two special versions of the ASM thesis in op.cit. where a small number of postulates is exhibited from which every sequential or synchronous parallel computational device can be proved to be simulatable in lock-step by an appropriate ASM. The construction in [8] depends on the way the abstract postulates capture the amount of computation performed by every single agent, and of the communication between synchronized agents, which are allowed in a synchronous parallel computation step. The desire to *prove* computational universality from abstract postulates implies the necessity to first capture a) the huge class of data structures and b) the many ways they can be used in a basic computation step and then to unfold every concrete basic parallel communication and computation step from the postulates. This unavoidably yields some 'encoding' and 'decoding' overhead to guarantee, for *every* computational system which possibly could be proposed, a representation by the abstract concepts of the postulates. As side effect of this generality of the postulates, the application of Blass and Gurevich's proof scheme to established models of computation tends to yield 'abstract' machine models which are more involved than necessary and may blur features which really distinguish different concrete systems. Furthermore, *postulating by an existential statement* that 'states' are appropriate equivalence classes of structures of a fixed signature (in the sense of logic), that the evolution happens as iteration of single 'steps', that the single-step 'exploration space' is bounded (i.e. that there is a uniform bound on memory locations basic computation steps depend upon, up to isomorphism), does not by itself provide, for a given computation or specification model, a standard reference description of its characteristic states, of the objects entering a basic computation step, and of its next-step function.

The goal in this paper is that of *naturally* capturing the basic data structures and single computation steps which characterize current systems of specification and computation, and of formulating them in a way which is uniform enough to pave the way for explicit technical comparisons. By deliberately keeping the ASM model for each proposed system as close as possible to the

original usual description of the system, so that it can be recognized straightforwardly to be simulated correctly and step by step by the ASM model, we provide for the ASM thesis a strong argument which includes asynchronous distributed systems, for which no proof of the ASM thesis is known. In addition our argument avoids a sophisticated existence proof for the ASM models from abstract postulates, avoids decoding of concrete concepts from abstract postulates, and avoids involved proofs to establish the correctness of the ASM models.

We start from control state ASMs, which were defined in [12] as a natural extension of Finite State Machines (FSMs), and from the extension of basic ASMs to so-called turbo ASMs, which were defined in [21] to establish a submachine concept that fits the synchronous parallelism of ASMs and includes sequential composition and iteration. Both concepts are reviewed in Section 2. In Section 3 we investigate Turing machine like *classical models of computation* which typically appear in standard computation theory textbooks (automata, substitution systems, tree computations, structured and functional programming) and in Section 4 the major currently used *system design models* (*executable* high-level design languages like UNITY or COLD, *state-based* specification languages like Petri nets and B, *stateless* modeling approaches like process algebra, and *axiomatic* logico-algebraic design systems like Z). For a survey of the numerous ASM models which have been defined for the dynamic and/or static semantics of the major current programming languages we refer to [14] and [23, Ch.9].

2 ASMs, control state ASMs, turbo ASMs

Abstract State Machines (more precisely what in [23] are termed basic ASMs) are transition systems which transform structures of a given signature, i.e. finite sets of so called transition *rules* of form

if *Condition* **then** *Updates*.

The *Condition* under which a rule is applied is a closed predicate logic formula of the underlying signature. *Updates* is a finite set of assignments of form $f(t_1, \ldots, t_n) := t$ whose execution is to be understood as *changing* (or defining, if there was none) in parallel the value of the occurring functions f at the indicated arguments to the indicated value. More precisely, in any given structure (called 'state'), first all parameters t_i, t are evaluated to their values, say v_i, v, then the value of $f(v_1, \ldots, v_n)$ is updated to v which represents the value of $f(v_1, \ldots, v_n)$ in the next state. Such pairs of a function name f and an argument $(v_1, \ldots, v_n)$ are called *locations*, location-value pairs (loc, v) are called *updates*. Without loss of generality predicates are treated as character-

istic functions (returning *true* or *false*) and constants and individual variables as 0-ary functions. Partial functions are turned into total functions by interpreting $f(x) = undef$ with a fixed special value *undef* as $f(x)$ being undefined.

An ASM computation step in a given state consists in executing *simultaneously* all updates of all transition rules whose guard is true in the state, if these updates are consistent, in which case the result of their execution yields the *next* state. In the case of inconsistency the computation does not yield a next state. A set of updates is called *consistent* if it contains no pair of updates with the same location, i.e. no two elements $(loc, v), (loc, v')$ with $v \neq v'$.

Simultaneous execution is enhanced by the following notation to express the simultaneous execution of a rule R for each x satisfying a given condition φ (where typically x will have some free occurrences in R which are bound by the quantifier):

forall x **with** φ

R

Similarly non-determinism can be expressed by rules of the form

choose x **with** φ

R

where φ is a Boolean valued expression and R a rule. The meaning of such an ASM rule is to execute rule R with an arbitrary x chosen among those satisfying the selection property φ. If there exists no such x, nothing is done.

We freely use combinations of **where**, **let**, **if-then-else**, etc. which are easily reducible to the above basic definitions. When dealing with multi-agent systems we use sets of agents each executing its own ASM. In the case of asynchronous computations the underlying concept of run (computation) becomes that of a partially ordered set of basic computation steps of single agents. These partial orders have to satisfy three natural constraints for whose definition we refer the reader to the last section of [32].

Control State ASMs. In [12] we defined a particularly frequent class of ASMs, called *control state ASMs*, which allow one to define machines providing below the main control structure of Finite State Machines (FSMs) synchronous parallelism and the possibility to manipulate data structures. A *control state ASM* is an ASM whose rules are all of the following form:

if $ctl\_state = i$ **then**

if $cond_1$ **then**

$rule_1$

$ctl\_state := j_1$

...

if $cond_n$ **then**

$rule_n$

$ctl\_state := j_n$

The finitely many control states $ctl\_state \in \{1, \ldots, m\}$ resemble the so-called 'internal' states of Finite State Machines. In a given control state i, these machines do nothing when no condition $cond_j$ is satisfied. One can use the widespread flowchart representation for control state ASMs where the control states appear as (possibly named) circles or directed arcs. This helps to visually distinguish the role of control states—to "pass control"—from that of ASM rules, which describe the update "actions" concerning the underlying data structure and are inscribed into rectangles, often separated from the rule guards which are then written into rhombs or hexagons labeling the arcs outgoing or ingoing the control states. We will sometimes use the following abbreviation which relates the visual notation for FSMs to the usual textual description of ASMs:

FSM(i, **if** $cond$ **then** $rule$, j) =

if $ctl\_state = i$ **and** $cond$ **then**

$rule$

$ctl\_state := j$

As is usually done for FSMs one could consider non-deterministic control state ASMs, using non-determinism as mechanism to resolve possibly conflicting updates of ctl_state. For reasons of clarity we prefer to keep for control state ASMs the parallel synchronous understanding of ASMs as firing in each step every rule. One can control possible conflicts e.g. by taking care that the rule guards $cond_k$ of rules fireable in control state i are disjoint.

Turbo ASMs. In [21] a precise definition is given which justifies to use, within the context of simultaneously executed rules, an occurrence of a 'rule call' $r(x_1, \ldots, x_n)$ where an ASM rule is expected, coming with a declaration $r(x_1, \ldots, x_n) = R$. Such a rule call $r(x_1, \ldots, x_n)$ is used only when the parameters are instantiated by legal values (objects, functions, rules, whatever) so that the resulting rule has a well defined semantical meaning. In these submachine rules also 'sequential machines' P **seq** Q are allowed to occur,

defined in op.cit. for ASM rules P, Q to denote the machine which in one macro-step provides the state change which is realized by the two substeps of first applying to the given state machine P and then to the resulting state machine Q. Similarly 'iteration machines' **iterate** P may be used, denoting for an ASM P the machine which in one macro-step realizes the computation $P^n = P$ **seq** $P \ldots$ **seq** P (n times) where n is the smallest number of iterations of P which lead to a state where a further application of P does not change the state any more. Technically speaking this is the case when a rule application produces the empty update set. Note that n may be undefined ($n = \infty$). ASMs which are built from basic ASMs using rule calls, sequential and iteration submachines are called *turbo ASMs*. For a detailed definition see [21].

3 Classical Computation Models

In this section we instantiate control state ASMs to classical automata and substitution systems (see any textbook on computation theory, e.g. [11]): FSMs (finite state machines à la Moore-Mealy and their more recent extensions by stream-processing, timing conditions, co-design control features), pushdown and computation universal automata (à la Turing, Scott, Eilenberg, Minsky, Wegner), replacement systems (à la Thue, Markov, Post and context free, attribute and tree adjoining grammars for language generation). The point of these definitions is not to model those systems by ASMs, which is a rather straightforward exercise performed on some examples in the literature, but to derive them uniformly as instances (simple refinements) of a general scheme which is made explicit in the present paper. We also show how to model by turbo ASMs structured and functional programming concepts and tree computations, including general forms of recursion.

3.1 Automata

Variations of Mealy/Moore Automata. Deterministic Mealy and Moore automata are control state ASMs where every rule has the following form, with *skip* instead of the output assignment in the case of Moore automata.

$$\textsc{Fsm}(i, \textbf{if } in = a \textbf{ then } out := b, j)$$

Writing programs in the usual tabular form, where one has one entry (i, a, j, b) for every instruction "in state i reading input a, go to state j and print output b", yields the following guard-free FSM rule scheme for updating (ctl_state, out), where the parameters *Nxtctl* and *Nxtout* are the two projec-

tion functions which define the program table, mapping 'configurations' (i, a) of control state and input to the next control state j and output b.

$\textsc{MealyFsm}(Nxtctl, Nxtout) =$

$ctl\_state := Nxtctl(ctl\_state, in)$

$out := Nxtout(ctl\_state, in)$

Since the input function *in* is monitored (i.e. read but not updated by the machine), it is not updated in the rule scheme, though one could certainly make it shared to formalize an input tape which is scanned letterwise from (say) left to right (see as example the machine StreamProcessingFsm defined below). The question of 1-way or 2-way automata is a question of whether one includes into the instructions also *Moves* of the input head (say on the input tape), yielding additional updates of the *head* position and a refinement of *in* to $in(head)$ (the input portion seen by the new reading head):

$\textsc{TwoWayFsm}(Nxtctl, Nxtout, Move) =$

$ctl\_state := Nxtctl(ctl\_state, in(head))$

$out := Nxtout(ctl\_state, in(head))$

$head := head + Move(ctl\_state, in(head))$

Non-deterministic versions of FSMs, as well as of all the machines we consider below, are obtained by placing the above rules under a **choose** operator to allow choices among different $R \in Rules$, obtaining rules of the form **choose** $R \in Rule$ **in** R.

We illustrate an instance of this scheme for the extension of FSMs to machines which appears in [37]. These machines compute stream functions $S^m \to S^n$ over a data set S (typically the set $S = A^*$ of finite or $S = A^{\mathbb{N}}$ of infinite words over a given alphabet A), yielding an output stream *out* resulting from consumption of the input stream *in*. Non-deterministically in each step these automata

- read (consume) at every input port a prefix of the input stream *in*,
- produce at each output port a part of the output stream *out*,
- proceed to the next control state *ctl_state*.

To extend the MealyFsm machines to a model of these stream processing FSMs it suffices to introduce two choice-supporting functions $Prefix\colon Ctl \times S^m \to PowerSet(S^m_{fin})$, yielding sets of finite prefixes among which to choose for given control state and input stream, and $Transition\colon Ctl \times (S^m_{fin}) \to PowerSet(Ctl \times S^n_{fin})$ describing the possible choices for the next control state

and the next finite bit of output. The rule extension for stream processing FSMs is then as follows, where input consumption is formalized by deletion of the chosen prefix from the shared function *in*.

StreamProcessingFsm(*Prefix*, *Transition*) =

choose *pref* ∈ *Prefix*(*ctl_state*, *in*)

choose (*c*, *o*) ∈ *Transition*(*ctl_state*, *pref*)

ctl_state := *c*

out := *concatenate*(*o*, *out*)

in := *delete*(*pref*, *in*)

Mealy/Moore automata give rise to *Mealy/Moore ASMs* defined in [12], a subclass of control state ASMs where the emission of output is generalized to arbitrary ASM rules:

MealyAsm = Fsm(*i*, **if** *in* = *a* **then** *rule*, *j*).

MealyAsms appear as components of *co-design FSMs* where turbo ASM component rules are needed to compute arbitrary combinational (external and instantaneous) functions. Co-design FSMs are used in [40] for high-level architecture design and specification and for a rigorous comparison of current models of computation. Usually co-design FSMs come together with a global agent scheduler or with timing conditions for agents which perform durative (not only atomic) actions. We illustrate the inclusion of timing conditions by an extension of Mealy-ASMs to *timed automata* [5]. In these automata letter input comes at a real-valued occurrence time which is used in the transitions where clocks record the time difference of the current input with respect to the previous input: $time_\Delta = occurrenceTime(in) - occurrenceTime(previousIn)$. Firing of transitions may be subject to clock constraints and includes clock updates (resetting a clock or adding to it the last input time difference). Typically the constraints are about input to occur within $(<, \leq)$ or after $(>, \geq)$ a given (constant) time interval, leaving some freedom for timing runs, i.e. choosing sequences of *occurrenceTime*(*in*) to satisfy the constraints. Thus timed automata can be modeled as control state ASMs where all rules have the following form:

TimedAutomaton(*Constraint*, *Reset*) =

Fsm(*i*, **if** *TimedIn*(*a*) **then** *ClockUpdate*(*Reset*), *j*)

where

$TimedIn(a) = (in = a$ **and** $Constraint(time_\Delta) = true)$

$ClockUpdate(Reset) =$

forall $c \in Reset$ **do** $c := 0$

forall $c \notin Reset$ **do** $c := c + time_\Delta$

In pushdown automata the Mealy automaton 'reading from the input tape' and 'writing to the output tape' is extended to reading from input and/or a *stack* and writing on the *stack*. Since these machines may have control states with no input-reading or no stack-reading, pushdown automata can be defined as control state ASMs where all rules have one of the following forms with the usual meaning of the *stack* operations $push, pop$ (optional items are enclosed in []):

PushDownAutomaton =

Fsm(i, **if** $Reading(a, b)$ **then** $StackUpdate(w), j$) **where**

$Reading(a, b) = [in = a]$ **and** $[top(stack) = b]$

$StackUpdate(w) \;=\; stack := push(w, [pop](stack))$

Turing-like automata. Writing pushdown transitions in tabular form

PushDownAutomaton($Nxtctl$, $Write$) =

$ctl\_state := Nxtctl(ctl\_state, in, top(stack))$

$stack :=$

$Pop\&Push(stack, Write(ctl\_state, in, top(stack)))$

identifies the 'memory refinement' of FSM *in*put and *out*put tape to *in*put and *stack* memory. The general scheme becomes explicit with Turing machines which combine *in*put and *out*put into one *tape* memory with moving *head*. All the *Turing-like machines* we mention below are control state ASMs which in each step, placed in a certain *pos*ition of their *mem*ory, read this *mem*ory in the *env*ironment of that *pos*ition and react by updating *mem* and *pos*. Variations of these machines are due to variations of mem, pos, env, whereas their rules are all of the following form:

TuringLikeMachine(mem, pos, env) =

Fsm(i, **if** $Cond(mem(env(pos))$ **then**

update $(mem(env(pos)), pos), j)$

For the original *Turing* machines this scheme is instantiated by $mem = tape$ containing words, integer positions $pos\colon Z$ where single letters are retrieved, $env = identity$, *Write*s in the position of the *tape head*. This leads to extending TwoWayFsm as follows (replacing *in* by *tape* and *Nxtout* by *Write*):

TuringMachine(*Nxtctl*, *Write*, *Move*) =

$ctl\_state := Nxtctl(ctl\_state, tape(head))$

$tape(head) := Write(ctl\_state, tape(head))$

$head := head + Move(ctl\_state, tape(head))$

It is an exercise to extend the 1-tape Turing machine to a k-tape and to an n-dimensional TM by data refining the 1-tape Turing memory and the related operations and functions correspondingly. Also register machines [11, Ch.AI1] are instances of Turing-like machines, whether working on numbers or words. In contrast to Turing and register machines, their generalizations introduced by Scott [44] and Eilenberg [27] instead of read/write operations on words stored in a tape provide data processing for arbitrary data, residing in abstract *mem*ory, by arbitrarily complex *mem*-transforming functions. Eilenberg's *X-machines* can be modeled as instances of Mealy ASMs whose rules in addition to yielding *out*put also update *mem* via global memory functions f (one for each input and control state):

XMachine = Fsm(i,

if $in = a$ **then** $\{out := b, mem := f(mem)\}, j)$

It is an exercise to define a stream processing version of Eilenberg's X-machines, similarly to the StreamProcessingFsm above. The global *mem*ory *Action*s of *Scott machines* together with their standard *IfThenElse* control flow directed by global memory *Test* predicates yield control state ASMs consisting of rules of the following form:

ScottMachine(*Action*, *Test*) =

$ctl\_state := IfThenElse(ctl\_state, Test(ctl\_state)(mem))$

$mem := Action(ctl\_state)(mem)$

Wegner's *interactive Turing machines* [46] in each step can receive some input from the environment and yield output to environment. Thus they simply extend the TuringMachine by an additional *input* parameter and an *output* action. This description clarifies the limitations of Wegner's rather particu-

liar model for systems of interacting machines, compared with the concept of asynchronous multi-agent ASMs.

TuringInteractive($Nxtctl$, $Write$, $Move$) =

$ctl\_state := Nxtctl(ctl\_state, tape(head), input)$

$tape(head) := Write(ctl\_state, tape(head), input)$

$head := head + Move(ctl\_state, tape(head), input)$

$output(ctl\_state, tape(head), input)$

Considering the output as written on an in-out tape comes up to define

$output :=$

$concatenate(input, Out(control, tape(head), input))$

as the output action using a function Out defined by the program. Viewing the input as a combination of preceding inputs/outputs with the new user input comes up to define $input$ as a derived function

$input = combine(output, user\_input)$

depending on the current $output$ and $user\_input$. The question of single-stream versus multiple-stream interacting Turing machines (SIM/MIM) is only a question of instantiating input to a stream vector $input = (inp_1, \ldots, inp_n)$.

The alternating variation of Turing machines can be obtained by extending TuringMachine to spawn subprocesses, each executing the same program but on different tapes. An alternating TM-computation is focussed to either accept or reject the initial input tape, whereto it is permitted to also invoke TM-subcomputations and to explore whether some or all of them accept or reject their input. For this purpose to the traditional control states, which are termed *normal* and in which the given TuringMachine is executed, four new *type*s are added: control states which simply *accept* or *reject* or which accept/reject in case some/every subcomputation accepts/rejects (*existential* type) or which accept/reject in case every/some subcomputation accepts/rejects (*universal* type). When in an existential or universal control state subcomputations are created and put into *running mode* (rule AltTmSpawn below), the invoking computation turns to *idle mode* to observe whether the *yield* of the subcomputations switches from *undef* to either *accept* or *reject* and to define its own *yield* correspondingly (TmYieldExistential, TmYieldUniversal below). Different subcomputations of an alternating Turing machine, whose program is defined by the given functions $Nxtctl$, $Write$, $Move$, are distinguished by parameterizing the machine instances by

their executing agents a, obtaining TuringMachine($Nxtctl$, $Write$, $Move$)(a) from the above defined TuringMachine by replacing the dynamic functions ctl_state, $tape$, $head$ with their instances $a.ctl\_state$ and $a.tape$, $a.head$. This leads us to the following definition where for simplicity of exposition but without loss of generality we assume that in an existential or universal state, the alternating Turing machine does not print or move its head and $NxtCtl$ yields the set of possible next control states where the subcomputations are started. We use $children(a) = \{c \mid parent(c) = a\}$ and denote agents by the variable **self**.

AlternatingTm($Nxtctl$, $Write$, $Move$) =

if $type$(**self** $.ctl\_state$) = $normal$ **then**

TuringMachine($Nxtctl$, $Write$, $Move$)(**self**)

if $type$(**self** $.ctl\_state$) $\in \{existential, universal\}$ **then**

AltTmSpawn(**self**)

TmYieldExistential(**self**)

TmYieldUniversal(**self**)

if $type$(**self** $.ctl\_state$) $\in \{accept, reject\}$ **then**

$yield$(**self**) := $type$(**self** $.ctl\_state$)

AltTmSpawn(a) = **if** $a.mode = running$ **then**

let $\{j_1, \ldots, j_k\} = Nxtctl(a.ctl\_state, a.tape(a.head))$

let $a_1, \ldots, a_k = new(Agent)$ **forall** $1 \leq i \leq k$

Activate(a_i, a, j_i)

$parent(a_i) := a$

$a.mode := idle$

Activate(b, a, j) =

StartSubComp(b, a, j)

$b.mode := running$

$b.yield := undef$

StartSubComp(b, a, j) =

$b.ctl\_state := j$

forall $pos \in domain(a.tape)$ **do** $b.tape(pos) := a.tape(pos)$

$b.head := a.head$

TmYieldExistential(a) =

if $a.mode = idle$ **and** $type(a.ctl\_state) = existential$ **then**

if $\forall c \in children(a)\ yield(c) = reject$ **then**

$yield(a) := reject$

if $\exists c \in children(a)\ yield(c) = accept$ **then**

$yield(a) := accept$

TmYieldUniversal(a) =

if $a.mode = idle$ **and** $type(a.ctl\_state) = universal$ **then**

if $\forall c \in children(a)\ yield(c) = accept$ **then**

$yield(a) := accept$

if $\exists c \in children(a)\ yield(c) = reject$ **then**

$yield(a) := reject$

Substitution systems. The substitution systems à la Thue, Markov, Post are Turing-like machines operating over $mem\colon A^*$ for some finite alphabet A with a finite set of word pairs (v_i, w_i) where in each step one occurrence of a 'premisse' v_i in mem is replaced by the corresponding 'conclusion' w_i. The difference between *Thue systems* and *Markov algorithms* is that Markov algorithms have a fixed scheduling mechanism for choosing the replacement pair and for choosing the occurrence of the to be replaced v_i. In the semi-Thue ASM rule below we use $mem([p, q])$ to denote the subword of mem between the p=th and the q-th letter of mem, which *match*es v if it is identical to v. By $mem(w/[p, q])$ we denote the result of substituting w in mem for $mem([p, q])$. The non-determinism of semi-Thue systems is captured by two selection functions. For the Markov version we show how one can include the condition on *match*ing already into the specification of these selection functions.

SemiThue(*ReplacePair*) =

let $(v, w) = select_{rule}(ReplacePair)$

let $(p, q) = select_{sub}(mem)$

if $match(mem([p, q]), v)$ **then** $mem := mem(w/[p, q])$

The Markov ASM is obtained from the SemiThue ASM by a pure data refinement, instantiating $select_{rule}(ReplacePair, mem)$ to yield the first $(v, w) \in ReplacePair$ with a premise occurring in mem, and $select_{sub}(mem, v)$ to determine the leftmost occurrence of v in mem. Similarly the ASM for *Post normal systems* is obtained by instantiating $select_{rule}(ReplacePair, mem)$ to yield a pair $(v, w) \in ReplacePair$ with a premise occurring as initial subword of mem, $select_{sub}(mem)$ to determine this initial subword of mem, and by updates of mem which delete the initial subword v and copy w at the end of mem.

Language generating grammars. We extract here from the ASM for the core of Prolog in [18] a basic tree generation and traversal ASM Backtrack variations of which model other constraint logic or functional programming languages as well as context free and attribute grammars. The machine dynamically constructs a tree of alternatives and controls its traversal. When its control state (which we call here *mode*) is *ramify*, it creates as many new children nodes to be computation *candidates* for its *currnode* as there are computation *alternatives*, provides them with the necessary *env*ironment and switches to *select*ion mode. In *mode* = *select*, if at *currnode* there is no more candidate the machine Backtracks, otherwise it lets the control move to TryNextCandidate to get *execute*d. The external function *alternatives* determines the solution space depending upon its parameters and possibly the current state. The dynamic function *env* records the information every new node needs to carry out the computation determined by the alternative it is associated with. The macro Back moves *currnode* one step up in the tree, to *parent*(*currnode*), until the *root* is reached where the computation stops. TryNextCandidate moves *currnode* one step down in the tree to the *next* candidate, where *next* is a possibly dynamic choice function which determines the order for trying out the alternatives. Typically the underlying execution machine will update *mode* from *execute* to *ramify*, in case of a successful execution, or to *select* if the execution fails. This model is summarized by the following definition.

Backtrack =

Ramify

Select

RAMIFY =

if $mode = ramify$ **then**

let $k =| alternatives(Params) |$

let $o_1, \ldots, o_k = new(NODE)$

$candidates(currnode) := \{o_1, \ldots, o_k\}$

forall $1 \leq i \leq k$

$parent(o_i) := currnode$

$env(o_i) := ith(alternatives(Params))$

$mode := select$

SELECT =

if $mode = select$ **then**

if $candidates(currnode) = \emptyset$ **then** BACK

else

TRYNEXTCANDIDATE

$mode := execute$

BACK =

if $currnode = root$

then $mode := Stop$

else $currnode := parent(currnode)$

TRYNEXTCANDIDATE =

$currnode := next(candidates(currnode))$

DELETE($next(candidates(currnode))$,

$candidates(currnode)$)

By data refinements BACKTRACK can be turned into the backtracking engine for the core of ISO Prolog [18], of IBM's constraint logic programming lan-

guage CLP(R) [20], of the functional programming language Babel [17], and of context free and of attribute grammars [38] as we are going to sketch here. To obtain the backtracking engine for Prolog, we instantiate *alternatives* to the function $procdef(stm, pgm)$ yielding a sequence of clauses in *pgm*, which have to be tried out in this order to execute the current goal *stm*, together with the needed state information from *currnode*. We determine *next* as *head* function on sequences, reflecting the depth-first left-to-right tree traversal strategy of ISO Prolog. It remains to add the execution engine for Prolog specified as ASM in [18], which switches *mode* to *ramify* if the current resolution step succeeds and otherwise switches *mode* to *select*. The backtracking engine for CLP(R) is the same, one only has to extend *procdef* by an additional parameter for the current set of *constraints* for the indexing mechanism and to add the CLP(R) engine specified as ASM in [20]. The functional language Babel uses the same function *next*, whereas *alternatives* is instantiated to $fundef(currexp, pgm)$ yielding the list of defining rules provided in *pgm* for the outer function of *currexp*. The Babel execution engine specified as ASM in [17] applies the defining rules in the given order to reduce *currexp* to normal form (using narrowing, a combination of unification and reduction).

To instantiate Backtrack for context free grammars G generating leftmost derivations we define $alternatives(currnode, G)$ to yield the sequence of symbols $Y_1, \ldots, Y_k$ of the conclusion of a G-rule whose premise X labels *currnode*, so that *env* records the label of a node, either a variable X or terminal letter a. The definition of *alternatives* includes a choice between different rules $X \to w$ in G. For leftmost derivations *next* is defined as for Prolog. As machine in $mode = execute$ one can add the following rule. For nodes labeled by a variable it triggers further tree expansion, for terminal nodes it extracts the yield (concatenating the terminal letter to the word generated so far) and moves the control to the parent node to continue the derivation in $mode = select$.

Execute(G) =

if $mode = execute$ **then**

if $env(currnode) \in VAR$ **then** $mode := ramify$

else

$output := output * env(currnode)$

$currnode := parent(currnode)$

$mode := select$

For attribute grammars it suffices to extend the instantiation for context free grammars as follows. For the synthesis of the attribute $X.a$ of a node X from

its children's attributes we add to the else-clause of the BACK macro the corresponding update, e.g. $X.a := f(Y_1.a_1, \ldots, Y_k.a_k)$ where $Y_i = env(o_i)$ for children nodes o_i and $X = env(parent(currnode))$. Inheriting an attribute from the parent and siblings can be included in the update of *env* (e.g. upon node creation), extending it to update also node attributes. The attribute conditions for grammar rules are included into EXECUTE(G) as additional guard to yielding output, of the form

$$Cond(currnode.a, parent(currnode).b, siblings(currnode).c)$$

We leave as an exercise to formulate an ASM for tree adjoining grammars, generalizing Parikh's analysis of context free languages by 'pumping' of context free trees from *basis trees* (with terminal yield) and *recursion trees* (with terminal yield except for the root variable).

3.2 Structured programming and recursion

Turbo ASMs provide the conceptual ingredients of structured programming. This has been illustrated in [21] by providing a surprisingly elementary proof for a general form of the celebrated Structured Programming Theorem of Böhm and Jacopini [9], constructing by sequential iteration simple turbo ASMs to compute arbitrary computable functions, in a way which combines the advantages of Gödel-Herbrand style functional and of Turing style imperative programming. We call *Böhm-Jacopini-ASM* any turbo ASM M which can be defined, using only **seq, while**, from basic ASMs whose non-controlled functions are restricted to one (a 0-ary) input function (whose values represent the arguments on which the given function is to be computed, they are fixed by the initial state), one (a 0-ary) output function (which upon terminating the computation will be updated to the computed function value), and the initial functions of recursion theory as static functions. It suffices to define by induction for each partial recursive (definition of a computable function) f a Böhm-Jacopini-ASM F computing it. For example if f is defined from g by the μ-operator, $f(x) = \mu y(g(x, y) = 0)$, and if a Böhm-Jacopini-ASM G computing g is given, then the following machine μ-OPERATOR mimics the standard definition of the μ-operator to compute f. The start submachine computes $g(x, rec)$ for the initial recursor value 0, the iterating machine computes $g(x, rec)$ for increased values of the recursor until 0 shows up as computed value of g, in which case the reached recursor value is set as output. We use a macro $F(in)$ to describe inputting from some external input source in to a machine F before it gets started, which in the functional notation appears as argument providing mechanism. Formally $F(in)$ stands for $in_F := in$ **seq** F.

μ-OPERATOR(G) =

$G(in_F, 0)$

$rec := 0$

seq

while $out_G \neq 0$

$G(in_F, rec + 1)$

$rec := rec + 1$

seq $out_F := rec$

The examples illustrates how by the atomicity of their black-box computations, turbo ASMs allow one to reflect exactly the machinery which underlies the common mathematical use of *functional* equations to evaluate function values. Often functional programs are characterized as different from imperative ones because "rather than telling the computer what to do, they *define* what it is that the computer is to provide" (quoted from [25]). The equations which appear in the Gödel-Herbrand type definition of partial recursive functions "define what it is that the computer is to provide" only on the basis of the implicit assumptions made for the procedure to be followed for the manipulation of arguments and values during the evaluation of terms. The corresponding Böhm-Jacopini-ASMs as indicated above make this machinery explicit, exhibiting how to evaluate the subterms when using the equations, as much as needed to make the functional shorthand work correctly the way it was hardwired in our brains through training at school.

In a similar way we can answer Moschovakis' question in [41] what are the abstract machines for standard forms of recursion, like the ones used to define the Mergesort algorithm. In fact turbo ASM submachines abstractly model the standard imperative calling mechanism, which provides the key for expressing the common intuitive understanding of recursion in terms of single-agent ASM computations. It suffices to extract the desired value from the final state reached by a turbo ASM step, which describes an abstract form $l \leftarrow R(a)$ of returning values to an indicated location (0-ary function) l. Here is the formal definition whose detailed justification can be found in [15] [2]. Let R_i, S be turbo ASMs with formal parameter sequences x_i of R_i and parameters y_i of S. Define:

let $\{y_1 = R_1(a_1), \ldots, y_n = R_n(a_n)\}$ **in** $S \equiv$

[2] A similar definition is given in [30]. In [15] we also explain why we are dissatisfied with the explanation of recursion in terms of distributed ASMs which is advocated in [34].

let $l_1, \ldots, l_n = new(FUN_0)$ **in**

forall $1 \leq i \leq n$ **do** $l_i \leftarrow R_i(a_i)$ **seq**

let $y_1 = l_1, \ldots, y_n = l_n$ **in** S

This definition allows us to capture by a turbo ASM the computations suggested by systems of recursive equations, e.g. to mergesort a given list L. It consists in FIRST splitting the list into a $LeftHalf(L)$ and a $RightHalf(L)$ (if there is something to split) and mergesort these two sublists separately (independently of each other), THEN to *Merge* the two results by an auxiliary elementwise *Merge* operation. This is expressed by the following turbo ASM which besides two auxiliary functions *LeftHalf*, *RightHalf* comes with an external function *Merge* which can be defined independently as a submachine.

MERGESORT(L) =

if$\mid L \mid \leq 1$ **then result**:= L **else**

let

$x =$ MERGESORT$(LeftHalf(L))$

$y =$ MERGESORT$(RightHalf(L))$

in

result$\leftarrow$ MERGE(x, y)

Remark. Concluding the ASM modeling of classical automata and computation concepts in this section one can say that *with hindsight*, it comes as no surprise that the numerous definitions of the notion of algorithms found in the 30'ies and 40'ies of the last century, in an attempt to mathematically capture the intuitive notion of *computable function*, all turned out to be equivalent: they are all variations of control state ASMs, mostly data refinements of the TURINGLIKEMACHINE model made explicit above. This positions also the Church-Turing thesis with respect to the more general ASM thesis [33].

4 System Design Models

In this section we show how to model by ASMs the basic semantical concepts of the executable high-level design languages UNITY and COLD, of widely used sequential and distributed state-based specification languages (illustrated for sequential systems by Parnas tables and B machines, for distributed sys-

tems by Petri Nets), of dedicated virtual machines (e.g. data flow machines), and of axiomatic logic-based or stateless modeling systems (like denotational semantics, VDM, Z, and algebraic systems like process algebras).

UNITY [24]. *Unity* computations are sequences of state transitions where each step comprises the simultaneous execution of multiple conditional variable assignments, including quantified array variable assignments of form **forall** $0 \leq i < N$ **do** $a(i) := b(i)$. States are formed by variables (0-ary dynamic functions which may be updated by different agents, respecting some naming conventions), conditions are typically formulated in terms of $<, =$, steps are executions of program statements which correspond in an obvious way to basic ASM rules. This is expressed below by applying a here not furthermore specified machine EXEC to basic ASM rules, which allows one to reflect metaprogramming features (e.g. to turn 'rules' into computational objects). The steps are scheduled using a global clock (Unity system time) which synchronizes the system components for an interleaving semantics: per step one statement of one component program in the system is scheduled using nondeterministic schedulers (required to respect a certain fairness condition on infinite runs). The Unity system time is thus identified with the time for executing one step of the ASM UNITYSYSTEM below. Like in basic ASMs, also in Unity there is no further control flow. Identifying components with basic ASMs and systems with sets of components leads therefore to the following computational model for Unity systems (which is linked to a particular proof system Unity, geared to extract proofs from the program text):

UNITYSYSTEM(S) =

choose $com \in Component(S)$

choose $rule \in Rule(com)$

EXEC($rule$)

COLD [28]. In the *C*ommon *O*bject-oriented *L*anguage for *D*esign states are realized as structures, including abstract data types (ADT) linked to an underlying dynamic logic proof system which is geared to provide proofs for algebraic specifications of states and their dynamics (à la Z and VDM). Computations are sequences of state transitions (due to the execution of procedure calls, built from statements viewed as expressions with side effects) allowing synchronous parallelism of simultaneous multiple conditional variable assignments (but no explicit **forall** construct) and non-deterministic choices among variable assignments and rules (procedure invocations). Thus a Cold class (with a set of states, one initial state, and a set of transition relations) corresponds in a standard way to a control state ASM, except that different states of a same class are allowed to have different signatures. The black box view offered for sequencing and iteration is directly reflected by the correspond-

ing turbo ASM constructs, taking into account that Cold provides a separate *guard statement* for blocking evaluation of guards which is executed only (with *skip* effect) when the guard becomes true.

The machine ColdModify(*Var*) models the idiomatic high-level construct *Mod* of Cold which supports non-determinism in choosing subsets of variables to be updated by chosen values. A similar construct *Use* permits to choose procedures from a set *Proc* to be Executed in sequence.

ColdModify(*Var*) =

choose $n \in \mathbb{N}$

choose $x_1, \ldots, x_n \in Var$

choose $v_1, \ldots, v_n \in Value$

forall $1 \leq i \leq n$ **do** $val(x_i) := v_i$

ColdUse(*Proc*) =

choose $n \in \mathbb{N}$, **choose** $p_1, \ldots, p_n \in Proc$

Exec(p_1) **seq** ... **seq** Exec(p_n)

Parnas Tables. An elaborate definition has been given in [42] for the semantics of a complex classification of Parnas tables which underlies the SCR method [36]. Their semantical meaning as a special matrix notation—a 2-dimensional layout of the CASE construct—for sequential systems with finitely many (controlled or monitored) state variables can be succinctly expressed by basic ASMs, providing an easily accessible foundation for the systematic use of such tables in system engineering. *Normal tables* are used to express the assignment of a value $t_{i,j}$ to $f(x, y)$ under the i-th row and the j-th column condition (where it is assumed that for each x, y at most one pair of row and column condition is true), formally:

NormalTable = **forall** $i \leq n, j \leq m$

if $RowCond_i$ **and** $ColumnCond_j$ **then** $f(x, y) := t_{i,j}$

Inverted tables are used to assign a value t_j to $f(x, y)$ under a leading row condition and a side condition (assumed to be sufficiently disjoint as for normal tables, to prevent inconsistent ('ambiguous') function updates), formally described by the following rules (for all $i \leq n, j \leq m$):

InvertedTable(i, j) = **if** $RowCond_i(x, y)$ **then**

if $SideCond_{i,j}$ **then** $f(x, y) := t_j$

Decision tables trigger a column action t_j under a parameterized column condition, formally expressed by the following set of rules (for all $j \leq n$, where disjoint properties avoid column actions conflicts):

DecisionTable(j) =

if $\forall i \leq m\ RowCond_{i,j}(s_i)$ **then** trigger t_j

VDM,Z,B [29,47,1]. These high-level design languages share the notion of computation as sequence of state transitions given by a before-after relation, where states are formed by variables taking values in certain sets (in VDM built up from basic types by constructors) with explicitly or implicitly defined auxiliary functions and predicates. The single (in basic B sequencing-free and loop-free) transitions can be modeled in a canonical way by basic ASM rules which capture also the 'unbounded' as well as the 'bounded' choice and the parallelism B offers in terms of simultaneous ('multiple generalized') substitution. The basic scheme is determined by what Abrial calls the 'pocket calculator model' which views a machine (program) as offering a set of operations (in VDM procedures with side effects) which are callable one at a time, e.g. in the non-deterministic form **choose** $R \in$ *Operation* **in** R or harnessed by a scheduler **let** $R =$ *scheduled*(*Operation*) **in** R; similarly for events which in event-B are allowed to happen only one per time unit. The structuring mechanisms for large and refined B machines are captured by turbo ASMs, including also the machine state hiding mechanism operations typically come with: it is allowed to activate (call) an operation for certain parameters, which results in an invariant preserving state modification, but besides calling the operation and taking its result no other direct access to the state is granted. Historically, this view has led to a certain bias to functional modeling one can observe for uses of VDM.

By the logical nature of Z specifications, their before-after expressions define the entire system dynamics. In B as in the ASM method, the formulation of the system dynamics—in B by operations (in event-based B by events [2–4]), in ASMs by rules—is separated from the formulation of the static state invariants and of the dynamic run constraints, which express desired system properties one has to prove to hold through every possible state evolution. However for carrying out these proofs, in contrast to the ASM method, there is a fixed link between B and a computer assisted proof system relating syntactical program constructs to proof rules which are used to establish program invariants and dynamic constraints along with the program construction. This fits also the basically axiomatic foundation of B as of Z and VDM: VDM by a denotational semantics; Z by axiom systems formulated in (mainly first-order) logic; B by Dijkstra's weakest precondition theory, interpreted in set-theoretic mod-

els and based upon the syntactic global concept of substitution (from which local assignment $x := t$ and parallel composition are derived). Differently from Z, which due to the purely axiomatic character of Z descriptions has intrinsic problems to turn specifications into executable code (see [35]), VDM and B are geared to obtain software modules from abstract specifications via refinements which are tailored to the proof rules used for proving that the refined operations satisfy 'unchanged' properties of their abstract counterparts.

Petri Nets [43]. The general view of Petri nets is that of distributed transition systems transforming objects under given conditions. In Petri's classical instance the objects are marks on *places* ('passive net components' where objects are stored), the *transitions* ('active net components') modify objects by adding and deleting marks on the places. In modern instances (e.g. the predicate/transition nets) places are locations for objects belonging to abstract data types (read: variables taking values of given type, so that a marking becomes a variable interpretation), transitions update variables and extend domains under conditions which are described by arbitrary first-order formulae. The distributed nature of Petri nets is captured by modeling them as multi-agent asynchronous ASMs, associating to each transition one *agent* to execute the transition. Each single transition is modeled by a basic ASM rule of the following form, where pre/post-places are sequences or sets of places which participate in the 'information flow relation' (the local state change) due to the transition and *Cond* is an arbitrary first-order formula. By modeling Petri net states as ASM states we include the abstract Petri net view proposed in [43] where states are interpreted as logical predicates which are associated to places and transformed by actions.

PetriTransition =

if *Cond*(*prePlaces*) **then** *Updates*(*postPlaces*)

where

Updates(*postPlaces*) = a set of function updates

Virtual Machines. IBM's *Virtual Machine* [39] and Dijkstra's *Abstract Machine* [26] concept originated in the 60-ies of the last century as a high-level operating system abstraction, but quickly spread to hierarchical system design in general, ranging from data spaces to programming language platforms to layered software architectures, and nowadays has become ubiquitous in high-level system design. The definition of ASMs provides an explicit mathematical description of the class of machines covered by this concept and thus not surprisingly found quickly numerous applications for modeling complex virtual machines (e.g. Warren's Abstract Machine [19] and its extensions [20,7,6], the Transputer [16] and the Java Virtual Machine architecture [45], the Neural Net (abstract data flow) Machine [22], the UPnP architecture [31]).

Logico-Algebraic Design Systems. The fascinating idea to write specifications as logical formulae with computations corresponding to logical deductions, furthermore in such a way that logical conjunction corresponds to system composition and logical implication to refinement, has led to a myriad of logic and algebraic specification and 'declarative programming' languages and calculi, like Prolog and its numerous variants, VDM, Z, innumerable 'logics of programs' offering proof calculi to support verification of program properties. It is certainly an advantage that once a complete logical specification is in place, a (possibly machine supported) proof for the desired program properties provides a rather high degree of reliability (if the implementation of the prover and its handling by the operating system are correct). Unfortunately this advantage has to be paid at such a high price that despite of the longstanding world-wide research effort in this direction, logical specifications are simply not part of standard industrial software engineering practice (though they are used with success in certain well-delineated areas, among which design and verification of control systems, protocols and hardware). This is so not only for the extraordinary cost of logical formalizations of real-life software projects, due to the considerable tecnicho-mathematical skill and the time needed to carry to the end a large scale logico-algebraic design and verification project, but also for intrinsic reasons. All declarative specifications by their very logical nature are subject to the frame problem of having to describe not only the local changes, but also everything that is supposed not to change. Every logic system implies a fixed level of abstraction for design and verification. These two features lead to the rightly critisized 'formal specification explosion phenomenon' that formal specifications which come in the form of a huge logical formula or system of algebraic equations tend to become orders of magnitude larger than the executable code, making it difficult (if possible at all) to fully understand them and to derive an efficient program from them. Minor but not negligible disadvantages derive from the typical external non-determinism in inference rule applications, which does not necessarily reflect the computationally intended scheduling, and from the natural drive of logical descriptions to lead to the rather special case of purely functional specifications ('big-step semantics' with exclusive consideration of relations between initial and final states) which are not always easy to be implemented. See the characteristic view expressed in [35, pg.89], that "The most important characteristic of Z, which singles it out from every other formal method, is that it is completely independent of any idea of computation".

The ASM method allows one to use such logic-based design and verification techniques *where appropriate*—which means desired, technically feasible and cost-effective—, integrating them into the high-level but state-based, genuinely semantical and computation oriented, specification and analysis techniques which are supported by ASMs. Successful projects in this direction have been reported using theorem proving systems (KIV, PVS, Isabelle) and model checkers, see the survey paper [14] for details. See also [10] for an integration

of process-algebraic and ASM concepts.

Conclusion We hope some reader will feel challenged by the above definitions to investigate classes of ASMs whose elements reflect the structure of well known computation or system design concepts, trying to establish interesting properties which relate different models, e.g. in terms of simulations between them or of the complexity of expressing in them computational features of interest.

To appear in: Annals of Pure and Applied Logic (2004)

References

[1] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[2] J.-R. Abrial. Extending B without changing it (for developing distributed systems). In H. Habrias, editor, *1st Conference on the B Method*, number ISBN 2-906082-25-2, pages 169–190, 1996.

[3] J.-R. Abrial and L. Mussat. Specification and design of a transmission protocol by successive refinements using B. In M. Broy and B. Schieder, editors, *Mathematical Methods in Program Development*. Springer, 1996.

[4] J.-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In D. Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method*, volume 1393 of *LNCS*, pages 82–128. Springer, 1998.

[5] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[6] C. Beierle and E. Börger. Refinement of a typed WAM extension by polymorphic order-sorted types. *Formal Aspects of Computing*, 8(5):539–564, 1996.

[7] C. Beierle and E. Börger. Specification and correctness proof of a WAM extension with abstract type constraints. *Formal Aspects of Computing*, 8(4):428–462, 1996.

[8] A. Blass and Y. Gurevich. Abstract State Machines capture parallel algorithms. *ACM Transactions on Computational Logic*, 3, 2002.

[9] C. Böhm and G. Jacopini. Flow diagrams, Turing Machines, and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, 1966.

[10] T. Bolognesi and E. Börger. Abstract State Processes. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003–Advances in Theory and Applications*, volume 2589 of *LNCS*, pages 22–32. Springer, 2003.

[11] E. Börger. *Computability, Complexity, Logic (English translation of* Berechenbarkeit, Komplexität, Logik *)*, volume 128 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1989.

[12] E. Börger. High level system design and analysis using abstract state machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, number 1641 in LNCS, pages 1–43. Springer-Verlag, 1999.

[13] E. Börger. Computation and specification models. A comparative study. In P. D. Mosses, editor, *Proceedings of the Fourth International Workshop on Action Semantics*, volume NS-02-8 of *BRICS Series*, pages 107–130. Department of Computer Science at University of Aarhus, December 2002.

[14] E. Börger. The origins and the development of the ASM method for high level system design and analysis. *J. of Universal Computer Science*, 8(1):2–74, 2002.

[15] E. Börger and T. Bolognesi. Remarks on turbo ASMs for computing functional equations and recursion schemes. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003—-Advances in Theory and Applications*, volume 2589 of *LNCS*, pages 218–228. Springer-Verlag, 2003.

[16] E. Börger and I. Durdanović. Correctness of compiling Occam to Transputer code. *Computer Journal*, 39(1):52–92, 1996.

[17] E. Börger, F. J. López-Fraguas, and M. Rodríguez-Artalejo. A model for mathematical analysis of functional logic programs and their implementations. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 410–415, Elsevier, Amsterdam, the Netherlands, 1994.

[18] E. Börger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 24:249–286, 1995.

[19] E. Börger and D. Rosenzweig. The WAM — definition and compiler correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence, chapter 2, pages 20–90. North-Holland, 1995.

[20] E. Börger and R. Salamone. CLAM specification for provably correct compilation of CLP(R) programs. In E. Börger, editor, *Specification and Validation Methods*, pages 97–130. Oxford University Press, 1995.

[21] E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Proceedings of CSL 2000)*, volume 1862 of *LNCS*, pages 41–60. Springer-Verlag, 2000.

[22] E. Börger and D. Sona. A neural abstract machine. *J. of Universal Computer Science*, 7(11):1007–1024, 2001.

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

[24] K. M. Chandy and J. Misra. *Parallel Program Design. A Foundation*. Addison Wesley, 1988.

[25] M. Davis. *The Universal Computer: The Road from Leibniz to Turing*. W.W. Norton, New York, 2000.

[26] E. W. Dijkstra. Structure of the T.H.E. multiprogrammming system. *Communications of the ACM*, 11:341–346, 1968.

[27] S. Eilenberg. *Automata, Machines and Languages Vol.A*. Academic Press, 1974.

[28] L. M. G. Feijs and H. B. M. Jonkers. *Formal Specification and Design*. Cambridge University Press, 1992.

[29] J. Fitzgerald and P. G. Larsen. *Modelling Systems. Practical Tool and Techniques in Software Development*. Cambridge University Press, 1998.

[30] N. G. Fruja and R. F. Stärk. The hidden computation steps of turbo Abstract State Machines. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003–Advances in Theory and Applications*, volume 2589 of *Lecture Notes in Computer Science*, pages 244–262. Springer-Verlag, 2003.

[31] U. Glässer, Y. Gurevich, and M. Veanes. High-level executable specification of the universal plug and play architecture. In *Proceedings of 35th Hawaii International Conference on System Sciences — 2002*, pages 1–10. IEEE Computer Society Press, 2002.

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

[33] Y. Gurevich. Sequential Abstract State Machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.

[34] Y. Gurevich and M. Spielmann. Recursive Abstract State Machines. *J. of Universal Computer Science*, 3(4):233–246, 1997.

[35] J. A. Hall. Taking Z seriously. In *ZUM'97*, volume 1212 of *Springer LNCS*, 1997.

[36] C. Heitmeyer. Using SCR methods to capture, document, and verify computer system requirements. In E. Börger, B. Hörger, D. L. Parnas, and D. Rombach, editors, *Requirements Capture, Documentation, and Validation*. Dagstuhl Seminar Report 99241, 1999.

[37] J. W. Janneck. *Syntax and Semantics of Graphs*. PhD thesis, ETH Zürich, 2000.

[38] D. E. Johnson and L. S. Moss. Grammar formalisms viewed as Evolving Algebras. *Linguistics and Philosophy*, 17:537–560, 1994.

[39] E. Kohlbrenner, D. Morris, and B. Morris. The history of virtual machines. Web pages at http://cne.gmu.edu/itcore/virtualmachine/history.htm.

[40] L. Lavagno, A. Sangiovanni-Vincentelli, and E. M. Sentovitch. Models of computation for system design. In E. Börger, editor, *Architecture Design and Validation Methods*, pages 243–295. Springer, 2000.

[41] Y. N. Moschovakis. What is an algorithm? In B. Engquist and W. Schmid, editors, *Mathematics Unlimited—2001 and beyond*. Springer, 2001.

[42] D. Parnas and J. Madey. Functional documents for computer systems. *Science of Computer Programming*, 25:41–62, 1995.

[43] W. Reisig. *Elements of Distributed Algorithms*. Springer, 1998.

[44] D. Scott. Definitional suggestions for automata theory. *J. Computer and System Sciences*, 1:187–212, 1967.

[45] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.

[46] P. Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40:80–91, 1997.

[47] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, 1996.