Computation and Specification Models. A Comparative Study*

Egon Börger
(Università di Pisa, Italy
boerger@di.unipi.it)

September 28, 2002

Abstract

For each of the principal current models of computation and of high-level system design, we present a uniform set of transparent easily understandable descriptions, which are faithful to the basic intuitions and concepts of the investigated systems. Our main goal is to provide a mathematical basis for the technical comparison of established models of computation which can contribute to rationalize the scientific evaluation of different system specification approaches in the literature, clarifying in detail their advantages and disadvantages. As a side effect we obtain a powerful yet simple new conceptual framework for teaching the fundamentals of computation theory.

1 Introduction

The presentation of this work in the *Action Semantics* workshop started from Peter Mosses' question to compare Action Notation (AN) [45] and Abstract State Machines (ASMs) [26]. Answering that question naturally led to a broader investigation, namely a comparative analysis of current specification and computation systems in terms of ASMs.

In this paper we assume the reader to know the definition of AN and of the notion of ASM. The main difference between the Action Semantics framework and the ASM method is their different goal. Action Notation has been tailored to support the development of programming languages. The notion of ASMs has been equipped with a general purpose method for high-level hardware and software system analysis and design and its stepwise refinement to code. There is also a difference in the origin of AN and ASMs which shaped the two approaches. AN was developed aiming at enriching denotational features with practically useful operational ones. In an attempt to overcome pragmatically dissatisfactory

[0]A preliminary version has been presented under the title *Definitional Suggestions for Computation Theory* to the Dagstuhl Seminar on "Theory and Application of Abstract State Machines", Schloss Dagstuhl, March 4-8, 2002.

aspects of a purely denotational approach, primitive and composed actions were directly reflected in close correspondence to programming concepts (semantic mapping of abstract syntax trees to predefined actions) and led to a compromise between competing language development requirements, corresponding to views of the designer, the implementer and the programmer. Gurevich's foundational concern to sharpen the Church-Turing thesis [38] led to an arguably most general notion of virtual machine which became the mathematical basis of the broad-spectrum high-level ASM method for practical system design and analysis [16].

The differences in origin and goal explain also the major technical differences in the realization of AN and ASMs. Actions in AN categorize what in ASMs comes as abstract, a priori unclassified function updates and declarations. Three parameters, organized into so-called facets, serve as basis for the classification: a) different computational aspects, b) types of effect propagation of actions, and c) types of action performance. The *basic facet* covers fundamental control patterns like sequentiality, parallelism, non-determinism; data storage phenomena are dealt with in the *functional facet* in case they are transient between actions, or in the *imperative facet* if they are stable in cells; the *communicative facet* describes interactions between distributed agents; scope information is treated in the *declarative facet*. Most of these features are not directly available in ASMs, though they are definable in a natural way (see [26]). Furthermore AN aims at the generation of a tool environment from language specifications, e.g. the semantics-directed generation of interpreters, compilers, etc., whereas ASMs support a general-purpose method which covers all system design and analysis aspects. In fact ASMs have been specialized to provide an executable semantics for AN, see [6] which contains also further details on tailoring ASMs to fit the AN framework.

In the rest of this paper we use ASMs as a framework for a comparative analysis of other specification and computation systems, comprising the following ones:

- UML Diagrams for System Dynamics
- Classical Models of Computation
  - Automata: Moore-Mealy, Stream-Processing FSM, Co-Design FSM, Timed FSM, PushDown, Turing, Scott, Eilenberg, Minsky, Wegner
  - Substitution systems: Thue, Markov, Post
  - Tree computations: backtracking in logic and functional programming, context free grammars, attribute grammars, tree adjoining grammars
  - Structured and functional programming
    - Programming constructs: seq, while, case, alternate, par
    - Gödel-Herbrand computable functions: Böhm-Jacopini Theorem
    - Recursion
- Specification and Computation Models for System Design

- Executable high-level design languages: UNITY, COLD
- State-based specification languages
  - distributed (Petri Nets)
  - sequential: VDM, Z, B
- Virtual machines
- Logic-based modeling systems
  - axiomatic systems: denotational, algebraic
  - process algebras (CSP, LOTOS, etc.)

2 Motivation

Since we will use Abstract State Machines (ASMs) as modeling framework, a question to answer before proceeding is why we do not use the proof for the synchronous parallel version of the ASM thesis which claims a form of computational universality for ASMs. The general thesis, as formulated in 1985 by Gurevich in a note to the American Mathematical Society [38], reads as follows (where dynamic structures stand for what nowadays are called ASMs):

> Every computational device can be simulated by an appropriate dynamic structure—of appropriately the same size—in real time

For the synchronous parallel case of this thesis Blass and Gurevich [11] discovered postulates from which every synchronous parallel computational device could be proved to be simulatable in lock-step by an appropriate ASM. Why are we not satisfied with the ASMs constructed by this proof?

The answer has to do with the price to be paid for *proving* computational universality from abstract postulates which cover a great variety of systems. On the one side, the ASM method emphasizes to model algorithms and systems *closely and faithfully, at their level of abstraction*, laying down the essential computational ingredients completely and expressing them directly, without using any encoding which is foreign to the computational device under study. On the other side, if one looks for a mathematical argument proving from explicitly stated assumptions the computational universality of ASMs as claimed in the thesis, some generality in stating the postulates is unavoidable, to capture the huge class of data structures and of the many ways they can be used in a basic computation step, which for every proposed concrete system have to be derived (*decoded*) from the postulates.

The construction by Blass and Gurevich in op.cit., which associates to every synchronous parallel computational system an ASM simulating the system step-by-step, depends in fact on the way the abstract postulates capture the amount of computation (by every single agent) and of the communication between the synchronized agents which is allowed in a synchronous parallel computation step. The necessity to uniformly unfold arbitrary concrete basic parallel communication and computation steps from the postulates as a matter of fact

yields some encoding 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— epistemologically significant—generality of the postulates, the application of the general transformation scheme to established models of computation may yield ASMs which are more involved than necessary and may blur features which really distinguish different concrete systems.

Furthermore, postulating by an existential statement e.g. that states are appropriate equivalence classes of structures of a fixed signature (in the sense of logic), that 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 the next-step function. In addition no proof is known to include distributed systems.

Our goal is that of *naturally* modeling systems of specification and computation, based upon a careful analysis of the characteristic conceptual features of each of them. We look for ASM descriptions for each established model of computation or of high-level system design which

- for every framework directly reflect the basic intuitions and concepts, by gently capturing the basic data structures and single computation steps which characterize the investigated system,
- are formulated in a way which is uniform enough to allow explicit comparisons between the classical system models,
- include asynchronous distributed systems.

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 to be simulated faithfully and step by step by the ASM model, we provide for the full ASM thesis a strong pragmatic argument which

- avoids a sophisticated existence proof for the ASM models from abstract postulates,
- avoids decoding of concrete concepts from abstract postulates,
- avoids a sophisticated proof to establish the correctness of the ASM models.

Since despite of listening carefully to the specifics of each investigated system and of tailoring the simulating ASM models accordingly we can achieve a certain uniformity, we provide a mathematical basis for technical comparison of established system design approaches which we hope will

- contribute to rationalize the scientific evaluation of different system specification approaches, clarifying their advantages and disadvantages,
- offer a powerful yet simple framework for teaching computation theory, unraveling the basic common structure of the myriad of different machine concepts which are studied in computation theory.

3 UML Diagrams for System Dynamics

For the modeling purpose, we use a generalization of Finite State Machines (FSMs) to a class of Abstract State Machines (ASMs) which have been introduced in [15] under the name of control state ASMs and are tailored to UML diagram visualizable machines. 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**
    $rule_n$
    $ctl\_state := j_n$

In a given control state i, these machines do nothing when no condition $cond_k$ is satisfied; otherwise for every $cond_k$ which is satsfied, $rule_k$ is executed and the control state changes to j_k so that usually the conditions are supposed or guaranteed to be disjoint to avoid conflicting updates which would stop the ASM computation. Control state ASMs represent a normal form for UML activity diagrams (see[17]) from where they inherit the graphical representation of control states by circles or by (possibly named) directed arcs, to visually distinguish the control-passing role of control states from that of the update actions concerning the underlying data structure which are expressed by the ASM rules inscribed into rectangles, separated from the rule guards written into rhombs. Control state ASMs thus offer to use arbitrarily complex parallel (synchronized) data structure manipulations below the main control structure of finite state machines. The resemblance to FSMs is reflected by the following notation:

FSM(i, **if** $cond$ **then** $rule, j$) =
  if $ctl\_state = i$ **and** $cond$ **then**
    $rule$
    $ctl\_state := j$

so that the control state ASM rule above becomes the set of rules FSM(i, **if** $cond_k$ **then** $rule_k, j_k$) for $k = 1, \ldots, n$.

When writing ASMs M we will use below the distinction between functions which are *controlled* by M (meaning that they are updated by rules of M and not by the environment) and those which are *monitored* by M (i.e. updated only by the environment, but read by M) or shared (i.e. updatable and readable by both M and the environment).

This intuitive understanding of control state ASMs and of different function types suffices for most of the machines defined in this paper. Otherwise we will state what more is needed. For a detailed textbook definition of these machines and of their synchronous or asynchronous multi-agent version we refer the reader to [26].

4 Classical Models of Computation

We show here that the classical automata and substitution systems, ranging from FSMs to computationally universal systems including the structured and the functional programming approaches, are all natural variations of classes of (mostly control state) ASMs. Introducing those formalisms as ASMs, as we do in our lectures on computation theory, avoids having to redefine the semantics of such systems again and again for each variation of the underlying concept of algorithm. This generalizes the uniform semantical frame underlying Scott's definitional suggestions for automata theory [51]. In this section we suppose the reader to know the basic concepts of computation theory (see any textbook on the subject, e.g. [14]).

4.1 Automata

We model here classical automata concepts, computationally universal ones as well as restricted machines.

4.1.1 Finite State Machines

The standard FSMs, also known as Mealy automata, are control state ASMs whose rules have the following form with a dynamic input function in and a dynamic output function out:

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

in, out usually range over letters a, b, but one may also have words or other value types and also sets or sequences of input or output lines (ports), like in networks of finite automata [27].

The subclass of Moore automata is characterized by the same form of rules but with *skip* instead of the output assignment. This give rise to the generalization to *Mealy/Moore ASMs* defined in [15], a subclass of control state ASMs where the emission of output is replaced by arbitrary ASM rules:

$$\textsc{MealyAsm} = \textsc{Fsm}(i, \textbf{if } in = a \textbf{ then } rule, j).$$

MealyAsms appear for example as components of *co-design FSMs* where rules are needed to compute arbitrary combinational (external and instantaneous) functions. Co-design FSMs are used in [43] for high-level architecture design and specification and for a precise comparison of current models of computation. Other examples of MealyAsms will be shown below.

If one prefers to write FSM programs in the usual tabular form, with one entry (i, a, j, b) for every instruction "in state i reading input a, go to state j and print output b", one obtains the following guard-free Mealy FSM rule scheme for updating (ctl_state, out). The parameters $Nxtctl, Nxtout$ are the two projection functions which define the program table, mapping 'configurations' (i, a) of control state and input to the next control state j and next output b.

$$\begin{array}{l}\textsc{MealyFsm}(Nxtctl, Nxtout) = \\ \quad ctl\_state := Nxtctl(ctl\_state, in) \\ \quad out := Nxtout(ctl\_state, in)\end{array}$$

Since the input functions in are monitored, they are not updated in the rule scheme, though one can certainly make them shared, e.g. to formalize an input tape which is scanned piecemeal say from left to right.

1-way automata are turned into 2-way automata by including 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):

$$\begin{array}{l}\textsc{TwoWayFsm}(Nxtctl, Nxtout, Move) = \\ \quad ctl\_state := Nxtctl(ctl\_state, in(head)) \\ \quad out := Nxtout(ctl\_state, in(head)) \\ \quad head := head + Move(ctl\_state, in(head))\end{array}$$

Non-deterministic versions of FSMs, as well as of all the machines we consider below so that there we will only mention deterministic machine versions, are obtained by placing the rules $R_1, \ldots, R_m$ to be chosen from under the **choose** operator, obtaining ASMs with rules of the following form:

$$\textbf{choose } R \in \{R_1, \ldots, R_m\} \textbf{ in } \ R.$$

4.1.2 Stream Processing FSMs

Stream processing FSMs are a specialization of FSMs to machines which compute stream functions $S^m \to S^n$ over a data set S (typically the set $S = A^*$ of finite or $S = A^N$ 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*.

This can be captured by introducing into the MealyFsm model 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 $pref \in Prefix(ctl\_state, in)$
    choose $(c, o) \in Transition(ctl\_state, pref)$
      $ctl\_state := c$
      $out := concatenate(o, out)$
      $in := delete(pref, in)$

In [41] these machines are used to enrich the classical networks of stream processing FSMs (stream processing components communicating among each other via input/output ports) by ASM state transformations of individual components.

4.1.3 Timed Automata

In timed automata [5] 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** $c := c + time_\Delta$

4.1.4 Push Down Automata

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 are control state ASMs with rules of one of the following forms and 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** $[top(stack) = b]$
    $StackUpdate(w) \;=\; stack := push(w, [pop](stack))$

4.1.5 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 the rules of TwoWayFsm as follows (replacing *in* by *tape* and $Nxtout$ by $Write$):

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

The extension of the 1-tape Turing machine to a k-tape and to an n-dimensional TM results from data refining the 1-tape Turing *mem*ory and the related operations and functions. Register machines are a data refined instance of k-tape Turing machines ([14, Ch.AI1]).

Scott [51] and Eilenberg [31] 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 global *mem*-transforming functions. Eilenberg's *X-machines* (and similarly their stream processing version) 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):

$$\text{XMachine} = \text{Fsm}(i, \textbf{if } in = a \textbf{ then } \{out := b, mem := f(mem)\}, j)$$

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:

$$\begin{aligned}&\text{ScottMachine}(Action, Test) =\\ &\quad ctl\_state := IfThenElse(ctl\_state, Test(ctl\_state)(mem))\\ &\quad mem := Action(ctl\_state)(mem)\end{aligned}$$

4.1.6 Interacting Turing Machines

Wegner's interactive Turing machines [53] in each step can receive some input from the environment and yield output to the environment. Thus they simply extend the TuringMachine by an additional *input* parameter and an *output* action

$$\begin{aligned}&\text{TuringInteractive}(Nxtctl, Write, Move) =\\ &\quad ctl\_state := Nxtctl(ctl\_state, tape(head), input)\\ &\quad tape(head) := Write(ctl\_state, tape(head), input)\\ &\quad head := head + Move(ctl\_state, tape(head), input)\\ &\quad output(ctl\_state, tape(head), input)\end{aligned}$$

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 $(inp_1, \ldots, inp_n)$.

4.1.7 Substitution Systems

Replacement systems à la Thue, Markov, Post are Turing-like machines operating over $mem: A^*$ 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 Thue systems is captured by two selection functions.

THUESYSTEM($ReplacePair$) =
  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 THUE 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. Note that we include the condition on *match*ing already into the specification of these selection functions. Similarly by instantiating $select_{rule}(ReplacePair, mem)$ the ASM for *Post normal systems* is obtained 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.

4.2 Tree Computations

In this section we model some basic tree computation schemes including language generating grammars like context free, attribute and tree adjoining grammars. Essentially we show how the notion of tree generation and traversal using a backtracking scheme can be captured by an ASM in such a way that applying to it appropriate data refinements yields well-known logic and functional programming patterns and generative grammars (context free and attribute grammars). For the underlying refinement notion see **??**.

4.2.1 Backtracking

We define here a BACKTRACK machine which dynamically constructs a tree of alternatives and controls its traversal. When its *ctl_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 =
  if $mode = ramify$ **then**
    let $k = |alternatives(Params)|$
    let $o_1, \ldots, o_k = new(NODE)$
      $candidates(currnode) := \{o_1, \ldots, o_k\}$
      forall $1 \leq i \leq k$
        $parent(o_i) := currnode$
        $env(o_i) := i\text{-th}(alternatives(Params))$
      $mode := select$
  if $mode = select$ **then**
    if $candidates(currnode) = \emptyset$ **then** BACK **else**
      TRYNEXTCANDIDATE
      $mode := execute$
  where
    BACK =
      if $parent(currnode) = root$ **then** $mode := Stop$
      else $currnode := parent(currnode)$
    TRYNEXTCANDIDATE =
      $currnode := next(candidates(currnode))$
      DELETE$(next(candidates(currnode)), currnode)$

We show now that by pure data refinements BACKTRACK can be turned into the backtracking engine for the core of ISO Prolog [21], of IBM's constraint logic programming language CLP(R) [23], of the functional programming language Babel [19], and also for context free and for attribute grammars [42].

4.2.2 Logic Programming Engine

We data refine here BACKTRACK to the backtracking engine for Prolog by instantiating the function *alternatives* to the function $procdef(stm, pgm)$. This is a Prolog specific function which yields the sequence of clauses in *pgm* to be tried out in this order to execute the current goal *stm*; these clauses come 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 [21], 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 [23].

The functional language Babel uses the same function *next*, whereas the function *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 [19] applies the defining rules in the given order to reduce *currexp* to normal form (using narrowing, a combination of unification and reduction).

4.2.3 Context-Free and Attribute Grammars

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

In a similar way one can formulate an ASM for tree adjoining grammars, generalizing Parikh's analysis of context free languages by 'pumping' of contex free trees from *basis trees* (with terminal yield) and *recursion trees* (with terminal yield except for the root variable), see [42].

4.3 Structured Programming

In this section we model standard structured programming constructs by natural classes of ASMs. In [24] two operators **seq** and **iterate** have been defined to compose ASMs sequentially and iteratively, capturing these two notions in a black-box view which fits the synchronous parallelism of ASMs, hiding internals

of subcomputations by compressing them into one step (so that the resulting machines became known as *turbo ASMs*). This allows one to provide succinct ASMs for standard programming constructs, as we are going to illustrate by turbo ASMs for the celebrated Structured Programming Theorem of Böhm and Jacopini [12], thus showing how to combine the advantages of Gödel-Herbrand style functional and of Turing style imperative programming.

Call *Böhm-Jacopini-ASM* any ASM M which can be defined, using only **seq, while**, from ASMs whose non-controlled functions are restricted to one (a 0-ary) input function (whose value is fixed by the initial state), one (a 0-ary) output function, and the initial functions of recursion theory as static functions. The purpose of the 0-ary input function which we write in_M is to contain the number sequence which is given as input for the computation of the machine. Similarly out_M is used to receive the output of M. The *initial* functions of recursion theory are the following functions from Cartesian products of natural numbers into the set of natural numbers: $+1$, all the projection functions U_i^n, all the constant functions C_i^n and the characteristic function of the predicate $\neq 0$. The **while**-operator can be defined in the usual way from an iteration operator:

$$\textbf{while } (cond)\ R \ = \ \textbf{iterate } (\textbf{if } cond \textbf{ then } R).$$

As usual a number theoretic function $f\colon N^n \to N$ is called *computable by an ASM* M if for every n-tuple $x \in N^n$ of arguments on which f is defined, the machine *started with input* x *terminates with output* $f(x)$. By 'M started with input x' we mean that M is started in the state where all the dynamic functions different from in_M are completely undefined and where $in_M = x$. Assuming the monitored function in_M not to change its value during an M-(turbo) computation, it is natural to say that M 'terminates in a state with output' y, if in this state out_M gets updated for the first time, namely to y. In the machines F we are going to construct now by induction for every partial recursive function f, the termination state will always be the state in which the intended turbo-computation reached its final goal.

Each initial function f is computed by the machine F of only one function update which reflects the defining equation of f.

$$F \equiv out_F := f(in_F)$$

In the inductive step we construct, for every partial recursive definition of a function f from its constituent functions f_i, a machine F which mimics the standard evaluation procedure underlying that definition. We use the following macros which describe inputting from some external input source *in* to a machine F before it gets started respectively extracting the machine output upon termination of F to some external target location *out*. These macros reflect the mechanism for providing arguments and yielding values which is implicit in the standard use of functional equation systems to determine the value of a function for a given argument.

$F(in) \equiv in_F := in$ **seq** F
$out := F(in) \equiv in_F := in$ **seq** F **seq** $out := out_F$

4.3.1 Function Composition

If functions $g, h_1, \ldots, h_m$ are computed by Böhm-Jacopini-ASMs $G, H_1, \ldots, H_m$, then their composition f defined by $f(x) = g(h_1(x), \ldots, h_m(x))$ is computed by the following machine $F = \textsc{FctCompo}$ where for reasons of simplicity but without loss of generality we assume that the submachines have pairwise disjoint signatures:

$\textsc{FctCompo}(G, H_1, \ldots, H_m) =$
$\{H_1(in_F), \ldots, H_m(in_F)\}$ **seq** $out_F := G(out_{H_1}, \ldots, out_{H_m})$

Unfolding this structured program reflects the order one *has* to follow for evaluating the subterms in the defining equation for f, an order which is implicitly assumed in the equational (functional) definition. First the input is passed to the constituent functions h_i to compute their values, whereby the input functions of H_i become controlled functions of F. The parallel composition of the submachines $H_i(in_F)$ reflects that their computations are completley independent from each other; what counts and is expressed is that all of them have to terminate before the next 'functional' step is taken. That next step consists in passing the sequence of out_{H_i} as input to the constituent function g. Finally g's value on this input is computed and assigned as output to out_F.

4.3.2 Primitive Recursion

Let a function f be defined from g, h by primitive recursion:

$$f(x, 0) = g(x), \quad f(x, y + 1) = h(x, y, f(x, y))$$

and let Böhm-Jacopini-ASMs G, H be given which compute g, h. Then the following machine $F = \textsc{PrimitiveRecursion}$ computes f, composed as sequence of three submachines. The start submachine evaluates the first defining equation for f by initializing the recursor *rec* to 0 and the intermediate value *ival* to $g(x)$. The *while* submachine evaluates the second defining equation for f for increased values of the recursor as long as the input value y has not been reached. The output submachine provides the final value of *ival* as output. As in the case of simultaneous substitution, the sequentialization and iteration described here make the bare minimum on ordering computational substeps explicit which is assumed and in fact needed in the standard functional use of the defining equations for f.

$\textsc{PrimitiveRecursion}(G, H) =$ **let** $(x, y) = in_F$ **in**
$\{ival := G(x),\ rec := 0\}$ **seq**
(**while** $(rec < y)$ $\{ival := H(x, rec, ival),\ rec := rec + 1\}$) **seq**
$out_F := ival$

4.3.3 Minimalization

If f is defined from g by the μ-operator, i.e. $f(x) = \mu y(g(x,y) = 0)$, and if a Böhm-Jacopini-ASM G computing g is given, then the following machine $F = \mu$-OPERATOR computes 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.

μ-OPERATOR$(G) =$
  $\{G(in_F, 0),\ rec := 0\}$ **seq**
  (**while** $(out_G \neq 0)$ $\{G(in_F, rec + 1),\ rec := rec + 1\})$ **seq**
  $out_F := rec$

4.4 Functional Programming (Recursion)

In this section we show how to model basic functional programming constructs by a natural subclass of turbo ASMs. A black-box submachine concept for value returning turbo ASMs has been defined in [24] which abstractly models the standard imperative calling mechanism. Triggered by the question raised in [44]: 'If algorithms are machines, then which machine is the mergesort?', the definition has been applied in [13] for simultaneous calls of multiple submachines, to seamlessly integrate functional description and programming techniques into ASMs. We illustrate this by a natural model for widely used forms of recursion.

The atomic view of an entire turbo ASM computation as one step is rendered by a set $[\![R(a_1, \ldots, a_n)]\!]^A$ of updates produced through executing the turbo ASM call $R(a_1, \ldots, a_n)$ in state A. This set represents the total effect of executing the submachine R in the call state A and is defined by

$$[\![R(a_1, \ldots, a_n)]\!]^A = [\![body[a_1/x_1, \ldots, a_n/x_n]]\!]^A,$$

where the submachine R is declared by $R(x_1, \ldots, x_n) = body$. The characteristic *functional abstraction* consists in abstracting from everything in a computation except the intended input/output relation, for example when using a machine to return a value and then passing it by value to other machines S. This is easily reflected in turbo ASMs by projecting that value out of the total computational effect $[\![R(a_1, \ldots, a_n)]\!]^A$ and passing it to S via the **let**-construct. Without loss of generality we assume expected output to be stored in a reserved location **result** which the programmer can change to a location l where he wants the expected return value to be transfered. We adopt the standard notation $l \leftarrow R(a)$ to denote the turbo computation outcome $[\![R_l(a)]\!]^A$ where R_l is the result of replacing **result** in R by l, i.e. $R_l = R(l/\textbf{result})$, so that when the computation is terminated its expected value can be retrieved from the location l. We use a function *new* to provide for each submachine call a fresh location (read: a 0-ary dynamic function, the variables of programming) where to record the result of the subcomputation, given that simultaneous calls—also of the same machine

but with different parameters—may yield different results. This explains the following definition.

Definition. Let R_i, S be arbitrary turbo ASMs where R_i may come with formal parameter sequences x_i and S with formal parameter sequences y_i . We define:

let $\{y_1 = R_1(a_1), \ldots, y_n = R_n(a_n)\}$ **in** $S =$
  let $l_1, \ldots, l_n = new(FUNCTION_0)$ **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

The use of turbo ASM return values allows one to explicitly capture the abstract machine(ry) which underlies the common mathematical evaluation procedure for functional expressions, including those defined by forms of recursion. We illustrate this by the following turbo ASM definitions of Quicksort and of Mergesort which exactly mimic the usual recursive definition of the algorithms to provide as *result* a sorted version of any given list. This answers the question raised in [44]: 'If algorithms are machines, then which machine is the mergesort?'

4.4.1 Quicksort

The computation suggested by the well-known recursive equations to quicksort L proceeds as follows: *first* partition the *tail* of the list into the two sublists $tail(L)_{<head(L)}, tail(L)_{\geq head(L)}$ of elements $< head(L)$ respectively $\geq head(L)$ and quicksort these two sublists separately (independently of each other), *then concatenate* the results taking $head(L)$ between them. The fact that this description uses various auxiliary list and comparison operations is reflected by the appearance of corresponding auxiliary functions in the following turbo ASM.

QUICKSORT$(L) =$
  if$\mid L \mid \leq 1$ **then result** $:= L$ **else**
    let
      $x =$ QUICKSORT$(tail(L)_{<head(L)})$
      $y =$ QUICKSORT$(tail(L)_{\geq head(L)})$
    in result $:= concatenate(x, head(L), y)$

4.4.2 Mergesort

The computation suggested by the usual recursive equations to mergesort a given list L consists in *first* splitting it 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 $Merge$ defined below as a submachine.

MERGESORT(L) =
  if| L |≤ 1 **then result** := L **else**
    let
      x = MERGESORT($LeftHalf(L)$)
      y = MERGESORT($RightHalf(L)$)
    in result := $Merge(x, y)$

Usually also $Merge$ is defined by a recursion, suggesting the following computation scheme which is formalized by the turbo ASM below. If both lists are non-trivial, by a case distinction the smaller one of the two list heads is determined and placed as the first element of the *result* list, concatenating it with the result of a separate and independent $Merge$ operation for the two lists remaining after having removed the chosen smaller head element. The ι-operator in $\iota x(P)$ denotes the unique x with property P (if there is such an x).

MERGE(L, L') =
  if $L = \emptyset$ **or** $L' = \emptyset$ **then result** := $\iota l(l \in \{L, L'\}$ **and** $l \neq \emptyset)$
  elseif $head(L) \leq head(L')$ **then**
    let $x = Merge(tail(L), L')$ **in result** := $concatenate(head(L), x)$
  else
    let $x = Merge(L, tail(L'))$ **in result** := $concatenate(head(L'), x)$

5 System Design Models

In this section we show how to model by ASMs the basic semantical concepts of currently used high-level design languages. We use in this section also the concept of asynchronous multi-agent ASMs, roughly speaking sets of ASMs whose runs are defined by appropriately constrained partial orders to reflect the intended causal dependencies between steps of different machines. The definition can be found in [39] and in [26, Ch.6].

5.1 Executable High-Level Design Languages

We discuss here two major executable high-level design languages of the 90'ies. We relate their characteristic semantical features to ASMs, without mentioning further the important executability aspect which clearly influenced the choice of the language constructs. The languages are UNITY [28] and COLD [32].

5.1.1 UNITY

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 shared, respecting some naming conventions), conditions are typically formulated in terms of $<, =$, steps are executions of program statements which

correspond in a direct way to ASM rules. The steps are scheduled using a global clock (the 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 non-deterministic schedulers (required to respect a certain fairness condition on infinite runs). (Dijkstra's guarded commands come with the same type of non-deterministic choice of one command per step.) Like in basic ASMs, 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. Unity comes with a particular proof system, geared to extract proofs from the program text, equipped with appropriately specialized composition and refinement concepts we do not discuss here.

UnitySystem(S) =
  choose $com \in Component(S)$
    choose $rule \in Rule(com)$
      $rule$

5.1.2 COLD

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 corresponding 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.

There is an idiomatic high-level construct Mod of Cold which supports non-determinism in choosing subsets of variables to be updated by chosen values. It is modeled by the following ASM.

ColdModify(Var) =
  choose $n \in N$, **choose** $x_1, \ldots, x_n \in Var$, **choose** $v_1, \ldots, v_n \in Value$
    forall $1 \leq i \leq n$ **do** $val(x_i) := v_i$

A similar construct Use permits to choose procedures from a set $Proc$ to be called in sequence.

ColdUse($Proc$) = **choose** $n \in N$, **choose** $p_1, \ldots, p_n \in Proc$
  p_1 **seq** $\ldots$ **seq** p_n

5.2 State-based Specification Languages

For sequential state-based specification languages we discuss three representative systems: VDM [33] (denotational), Z [57] (axiomatic), and B [1] (pseudo-code). As representative distributed state-based modeling systems we relate Petri nets [47] to asynchronous ASMs.

5.2.1 VDM, Z, B

These three 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** $R \in \mathit{Operation}$ **in** R or harnessed by a scheduler **let** $R = \mathit{scheduled}(\mathit{Operation})$ **in** R; similarly for events which in event-B are allowed to happen only one per time unit.

This view points to a methodological difference between the forces which drove the development of the B method compared to that of the ASM method. Abrial's B method is the result of an engineer's bottom-up analysis: 'The ideas behind the concept of abstract machine have all been borrowed from those ideas that are behind some well-known programming features such as modules, packages, abstract data types or classes' [2, pg.175]. Also the event-B notion of basic events, which corresponds to the guarded update rules of basic ASMs, came out of the concern to 'separate assignments from scheduling'. Gurevich's concept of ASMs is the result of a logician's top-down analysis, brought to light by a mathematical investigation of the ASM thesis (and supported by an extensive experimentation with the concept, see [16], [26, Ch.10] for the historical details).

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, 3, 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. Thus defining modules becomes intimately related to inventing lemmas. 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 models 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 [40]), 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.

5.2.2 Petri Nets

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 asynchronous ASMs, associating to each transition one *agent* to execute the transition. Each single transition is modeled by a basic ASM rule of the form defined below, 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 [47] 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

5.2.3 Virtual Machines

Virtual machines by definition are machines. Typically they work over a specific set of states, appropriate to the specific purpose. Thus they 'are' particular ASMs. In fact for design or analysis purposes numerous virtual machines have been explicitly modeled as ASMs, e.g. the Warren Abstract Machine [22] and

its extensions [8, 23, 10, 9, 7], the Transputer [18], the RISC machine DLX [20], the Java Virtual Machine [52], the Neural Net (abstract data flow) Machine [25], the UPnP architecture [36]), etc.

5.3 Logic Based Modeling Systems

There is a myriad of logic-based and algebraic specification and 'declarative programming' languages and calculi, like Prolog and its numerous variants, VDM, Z, structural operational or natural semantics systems, process algebra languages like CSP, LOTOS, innumerable 'logics of programs', dynamic logics, temporal logics, rewriting logics, offering proof calculi to support verification of program properties, etc. These approaches have the pattern of logic in common: specifications are typically expressed by systems of equations (with fixpoint operators to solve equations) or of general axioms and inference rules, so that they all are exposed to the frame problem and the difficulty to control the order of inference rule applications. Most of these systems are not conceived to serve general-purpose specifications but are tailored to specific goals, the way Plotkin's *Structural Operational Semantics* [35] or Kahn's *Natural Semantics* or Mosses' Action Semantics [45] are tailored for dealing with the semantics of programming languages. Numerous of these approaches are driven by structural patterns where the syntax dictates the principles of compositionality. Since this is not the place to evaluate the advantages or disadvantages of such often stateless approaches with respect to state-based transition systems, we limit ourselves to observe that the ASM method allows one to use such logic-based design and verification techniques *where appropriate*—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 possible with ASMs. Successful projects in this direction have been reported using theorem proving systems (KIV, PVS, Isabelle) and model checkers, see e.g. [46, 37, 49, 58, 30, 29, 50, 48, 34] and [54, 55, 56] for details.

References

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

[2] J.-R. Abrial. Extending b without changin 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 theoryof timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[6] M. Anlauff, S. Chakraborty, P. Kutter, A. Pierantonio, and L. Thiele. Generating an Action Notation environment from montages descriptions. *Software Tools and Technology Transfer, Springer*, 3:431–455, 2001.

[7] C. Beierle. Formal design of an abstract machine for constraint logic programming. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 377–382, Elsevier, Amsterdam, the Netherlands, 1994.

[8] C. Beierle and E. Börger. Correctness proof for the WAM with types. In E. Börger, G. Jäger, H. Kleine Büning, and M. M. Richter, editors, *Computer Science Logic*, volume 626 of *LNCS*, pages 15–34. Springer-Verlag, 1992.

[9] 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.

[10] 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.

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

[12] 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.

[13] T. Bolognesi and E. Börger. Remarks on turbo asms for computing functional equations and recursion schemes. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003*, volume xxx of *LNCS*. Springer, 2003.

[14] 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.

[15] 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.

[16] 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.

[17] E. Börger, A. Cavarra, and E. Riccobene. An ASM semantics for UML activity diagrams. In T. Rus, editor, *Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000 Proceedings*, volume 1816 of *LNCS*, pages 293–308. Springer-Verlag, 2000.

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

[19] 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.

[20] E. Börger and S. Mazzanti. A practical method for rigorously controllable hardware design. In J. P. Bowen, M. B. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *LNCS*, pages 151–187. Springer-Verlag, 1997.

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

[22] 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.

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

[24] 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.

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

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

[27] A. Brüggemann, L. Priese, D. Rödding, and R. Schätz. Modular decomposition of automata. In E. Börger, G. Hasenjäger, and D. Rödding, editors, *Logic and Machines: Decision Problems and Complexity*, number 171 in LNCS, pages 198–236. Springer, 1984.

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

[29] A. Dold. A formal representation of Abstract State Machines using PVS. Verifix Technical Report Ulm/6.2, Universität Ulm, July 1998.

[30] A. Dold, T. Gaul, V. Vialard, and W. Zimmermann. ASM-based mechanized verification of compiler back-ends. In U. Glässer and P. Schmitt, editors, *Proceedings of the Fifth International Workshop on Abstract State Machines*, pages 50–67. Magdeburg University, 1998.

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

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

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

[34] A. Gargantini and E. Riccobene. Encoding Abstract State Machines in PVS. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 303–322. Springer-Verlag, 2000.

[35] G.D.Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN19, Department of Computer Science at University of Aarhus, 1981.

[36] 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.

[37] W. Goerigk, A. Dold, T. Gaul, G. Goos, A. Heberle, F. W. von Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Ruess, and W. Zimmermann. Compiler correctness and implementation verification: The verifix approach. In P. Fritzson, editor, *International Conference on Compiler Construction*, volume Proceedings of the Poster Session of CC'96, IDA Technical Report LiTH-IDA-R-96-12, Linköping/Sweden, 1996.

[38] Y. Gurevich. A new thesis. *Abstracts, American Mathematical Society*, page 317, August 1985.

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

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

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

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

[43] 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.

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

[45] P. D. Mosses. *Action Semantics*. Cambridge University Press, 1992.

[46] C. Pusch. Verification of compiler correctness for the WAM. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *LNCS*, pages 347–362. Springer-Verlag, 1996.

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

[48] G. Schellhorn. *Verifikation abstrakter Zustandsmaschinen*. PhD thesis, Universität Ulm, 1999.

[49] G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM case study. *J. of Universal Computer Science*, 3(4):377–413, 1997.

[50] G. Schellhorn and W. Ahrendt. The wam case study: Verifying compiler correctness for prolog with kiv. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*. Kluwer, 1998.

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

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

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

[54] K. Winter. Model checking for Abstract State Machines. *J. of Universal Computer Science*, 3(5):689–701, 1997.

[55] K. Winter. Towards a methodology for model checking ASM: Lessons learned from the flash case study. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 341–360. Springer-Verlag, 2000.

[56] K. Winter. Automated checking of control tables. E-mail to E. Börger, December 24, 2001.

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

[58] W. Zimmerman and T. Gaul. On the construction of correct compiler backends: An ASM approach. *J. of Universal Computer Science*, 3(5):504–567, 1997.