A practice-oriented course on the principles of computation, programming and system design and analysis

Egon Börger

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

Abstract. We propose a simple foundation for a practice-oriented undergraduate course that links seamlessly computation theory to principles and methods for high-level computer-based system development and analysis. Starting from the fundamental notion of virtual machine computations, which is phrased for both synchronous and asynchronous systems in terms of Abstract State Machines, the course covers in a uniform way the basics of algorithms (sequential and distributed computations) and formal languages (grammars and automata) as well as the computational content of the major programming paradigms and high-level system design principles. The course constitutes a basis for advanced courses on algorithms and their complexity as well as on rigorous methods for requirements capture and for practical hardware/software design and analysis.

We outline here a successful use one can make of Abstract State Machines (ASMs) as a unifying conceptual ground for a practice-oriented undergraduate course on computation theory which covers classical models of computation—algorithms (undecidability and complexity) and formal languages (grammars and automata of the Chomsky hierarchy)—, but also the principles of programming constructs and of high-level design and analysis of computer-based systems. The fundamental notion of virtual machine computations and the ways to investigate them taught in this course seem to appeal to practice-oriented students and provide a basis for advanced courses on the use of rigorous methods for requirements capture and for the design and the analysis of real-life hardware/software systems, as illustrated in the experience reports [13,15].

In Section 1 we outline the fundamental questions and concepts addressed in the proposed course and introduce the basic notion of Abstract State Machines, a rigorous form of virtual machines which allows one to treat in a uniform way computation theory and system design issues. In Section 2 we explain how to uniformly present the variety of classical formal languages (Chomsky hierarchy grammars and equations for recursive functions) and the corresponding automata, by appropriately instantiating basic ASMs. This naturally leads to a coding-free undecidability proof, on half a page with five lines of proof starting from scratch, for the termination problem of any computation-universal language, including any programmming language that satisfies a few natural closure properties, and to a discussion of the Church-Turing thesis, strengthened by ASMs to capture also resource bound considerations. In Section 3 we propose using ASMs to define the computational meaning of the major sequential and concurrent programmming constructs appearing in imperative, object-oriented, functional or logic programs. In Section 4 we indicate how the rigorous-pseudo-code character of ASMs can be exploited to use these machines as conceptual frame for advanced courses on accurate high-level modeling and analysis (validation and verification) of complex real-life computer-based systems.

In this paper we can only give an outline of the themes for the course so that we refer for an elaboration of the technical details to the literature. Except for the undecidability proof, we also restrict our attention on the definitional aspects for which we suggest a novel approach, which allows the lecturer to a) define only once the notion of (mono-agent or synchronous or asynchronous multi-agent virtual machine) computation, which covers all the variations one encounters in the literature, and to b) directly deploy the involved fundamental theoretical concepts for challenging practical problems the working computer scientist has to face in his professional life, covering in a uniform way two things:

- The investigation of the basic concepts from the theory of computation and programming through the well-known classical theorems. The textbook [9] is still a good source for many simple proofs.
- Teaching practical system design principles through building and analysing rigorous high-level models. The AsmBook [30] contains some real-life case studies.

We do not discuss any purely didactical concern or techniques how to teach the course. The AsmBook [30] shows a way how to teach ASMs. It includes a CD with a set of slide decks, in pdf and ppt format, for lecturers of introductory and advanced courses. That material can also be downloaded from the AsmBook website at `http://www.di.unipi.it/AsmBook/`. As to the level of the course, it depends on the depth chosen. The course fits wherever an introductory course on computability or automata or more generally computation theory is taught. As to the size of the course, it may vary between 20 and 50 hours, depending again on the depth into which the lecturer wishes and can afford to go in treating the proposed arguments.

1 The Basic Questions and Concepts

The proposed course is centered around the following four basic questions any conceptual framework of computation theory and of computer-based system design and analysis has to address.

- **What are the basic virtual machines** needed to perform any kind of computation (universal virtual machine concept), whether stand-alone or in cooperation involving distributed computations?
  - *What are the states* (data structures) of such a general-purpose concept of virtual machines? What are the data sharing structures to express the cooperation of these virtual machines?
- **What are the basic programs** (algorithms) for most general virtual machines?
  - *What are the basic control structures* provided by the programs of such machines?
  - *What are the basic communication means* provided by the programs of such machines?
  - *What are the runs*, sequential or distributed, formed by executing instructions for state-transforming or communication steps of such machines?
- **What are the basic properties** of such machines and their programs, like their functionality (required or achieved), their computational power, their memory or time complexity, etc? What are the languages to appropriately express these properties?
- **What are the basic means of analysis** to establish, in terms of experimental validation and of mathematical verification, desired properties for running programs on such machines?

In dealing with these questions we advocate to separate from the very beginning different concerns. This creates in the student a firm *divide-and-conquer* attitude that encourages an approach to system design and analysis which uses systematically the piecemeal introduction of design and verification details (so-called *stepwise refinement* method) and thereby enables the future practitioner to adopt for each development step the appropriate one among the multitude of available definition and proof methods. The three major types of activities we suggest to teach to distinguish are the following ones:

- *Separate design from analysis.* For conceptually simple or small systems as the ones used in classical computation theory, definition and analysis means are often intertwined with advantage, e.g. when a recursive definition of some concept is coupled with recursively defined deduction rules to prove properties for that concept in a calculational manner. Also systems that are tailored to successfully deal with particular classes of to-be-verified programs take advantage from a tight link between means of program development and proof techniques to check program properties; a good example for this is the B-method [2]. However, the complexity of real-life computer-based systems makes it appropriate to generally separate design from analysis. This helps to not restrict the design space or its structuring into components by proof principles which are coupled to the design framework in a fixed a priori defined way, as happens for example with the refinement concept in the B-method.

- *Separate different analysis types and levels.* Such a stratification principle is widely accepted in mathematics. It also applies to system analysis where it can be instantiated in the following way.
  - Separate experimental *validation* (system simulation and testing) from mathematical *verification*.
  - *Distinguish verification levels* and the characteristic concerns each of it comes with. Each verification layer has an established degree of to-be-provided detail, formulated in an appropriate language. E.g. reasoning for human inspection (design justification by mathematical proofs) requires other features than using rule-based reasoning systems (mechanical design justification). Mathematical proofs may come in the form of proof ideas or proof sketches or as completely carried out detailed proofs. Formalized proofs are based on inference calculi which may be operated by humans or as computerized systems, where one should distinguish interactive systems (theorem proving systems like PVS, HOL, Isabelle, KIV) from automatic tools (model checkers and theorem provers of the Otter type). Another distinction comes through separating static program analysis from a run-time-based analysis of dynamic program properties (so called runtime verification). For each verification level the lecturer finds in the literature case studies, coming from different domains (hardware, programming languags, protocols, embedded systems), showing how to use ASMs to reason about the specification a the given level of abstraction.

1.1 Abstract State Machines

An important reason for starting the course with the notion of Abstract State Machines, whose definition is outlined in this section, is that these machines represent a most general definition of Virtual Machines. This has become clear from over ten years of experience with modelling and analysing outstanding real-life virtual machines and is theoretically underpinned by the ASM thesis, a resource-bound-aware generalization of the thesis of Church and Turing (see the next section). In addition ASMs provide a framework for a theoretically well-founded, coherent and uniform **practical combination of abstract operational descriptions with functional and axiomatic definitions**, thus eventually overcoming an alleged, though unjustified and in fact destructive, dichotomy between declarative and operational design elements which has been advocated for the last thirty years in the literature and in teaching.

Formally, ASMs can be presented as transition systems which transform structures, thus involving two ingredients, namely notions of abstract *state* and of single *computation* step.
Abstract system states are represented by structures, understood the way they appear in Tarski's semantical foundation of classical logic[1], given as domains of objects coming with predicates (attributes) and functions defined on them. If this definition of structures cannot be assumed to be known from an introduction into classical logic, structures can equivalently be viewed as sets of values residing in abstract memory units, so-called *locations*, which are organized into tables. Here a table is an association of a value to each table entry. Each table entry is a location, consisting of a table (or function) name and an argument. These tables are nothing else than what logicians call the interpretation of a function.

This general concept of structures incorporates truly 'abstract data types', which may be defined in many satisfactory ways, not only by equations or by logical axioms, but also operationally by programs or rules producing dynamic sets of updates of locations, as explained now. In fact ASMs represent a form of "pseudo-code over abstract data", virtual machine programs whose instructions are guarded function updates, structure transforming "rules" of the form[2]

if *Condition* **then** $f(t_1, \ldots, t_n) := t$

[1] which is directly related to the object-oriented understanding of classes and their instances

[2] This definition of machine "instructions" combines the traditional distinction between branching (test) instructions and action instructions (see for example Scott's definition of abstract machine programs in [59]) and avoids, for reasons explained below, to name instructions by labels which support the interruption of the standard sequential instruction execution by branching.

Also the auxiliary functions and predicates, which appear in the expressions t_i, t and thus are part of the system states, can be given purely functional or axiomatic or whatever other form of satisfactory definitions. This is supported by a classification of functions into basic and derived. Derived functions are those whose definition in terms of basic functions is fixed and may be given separately, e.g. in some other part ("module" or "class") of the model to be built. An orthogonal classification which supports this combination of declarative and operational features is the distinction between static and dynamic functions. The further classification of dynamic functions with respect to a given (machine executing) agent into controlled (readable and writable), monitored (readable), output (writable) and shared functions supports to distinguish between the roles different 'agents' (e.g. the system and its environment) play in using (providing or updating the values of) dynamic functions. A particularly important class of monitored functions are selection functions, for which also a special notation is provided (see below). Monitored and shared functions also represent a rather general mechanism to specify communication types between different agents, executing each a basic ASM. For details see the AsmBook [30, Ch.2.2.3].
ASM computations are understood as for traditional transition systems, except that the rules of ASMs are executed in parallel[3] so that the students learn from the very beginning to avoid, as long as they are concerned with building a high-level system model, to sequentialize independent actions. The definition of further control structures (like sequential execution or iteration) can be added where needed for a concrete implementation, some standard examples will be discussed in Section 2 in connection with the concept of structured programming. For *asynchronous multi-agent ASMs* it suffices to generalize runs from sequences (linear orders) of transition system moves of just one basic ASM to *partial orders* of moves of multiple agents, each executing a basic ASM, subject to a natural *coherence condition*, see [30, Def.6.1.1].
Non-determinism as incorporated in selection functions has also an explicit standard notation, namely **choose** x **with** ϕ **do** *rule* and

```
choose $x$ with $\phi$
  rule
```

standing for the rule to execute *rule* for one element x, which is arbitrarily chosen among those satisfying the selection criterion ϕ.
Synchronous parallelism, already present in the execution of ASM rules, is extended by a standard notation, namely **forall** x **with** ϕ **do** *rule* and

```
forall $x$ with $\phi$
  rule
```

standing for the execution of *rule* for every element x satisfying the property ϕ.
Control state ASMs, introduced in [10], are an alternative way to define basic ASMs as an extension of finite state machines (FSMs), if for some reason that concept is already known to the students. This comes up to enrich the notion of state and state transition: the *internal states* i become part of general *structures*[4] and the transitions are generalized to guarded synchronous parallel updates of those structures, in addition to updating what is now called more specifically control state i or *mode*. In this perspective, synchronous ASMs are given by sets of locally synchronous and globally asynchronous control state ASMs. To make this generalization of FSMs to control state ASMs transparent, we use the notation FSM(i, **if** *cond* **then** *rule*, j) for the following rule:

```
if $ctl\_state = i$ and cond then
  rule
  $ctl\_state := j$
```

[3] More precisely: to execute one step of an ASM in a given state S determine all the fireable rules in S (s.t. *Condition* is true in S), compute all expressions t_i, t in S occuring in the updates $f(t_1, \ldots, t_n) := t$ of those rules and then perform simultaneously all these location updates.

[4] This departs from the unstructured notion of states in the above-mentioned Scott machines, see [59].

The above working definition refers only to basic intuitions from programmming practice. It suffices for most of the course material discussed below. A formalized definition of the semantics of basic or asynchronous ASMs is obviously needed for detailed proof verifications within the fixed syntax of some logic or to understand special features of implementations of ASMs. Such a definition can be given and is available in textbook form in terms of a first-order-logic-based derivation system in [30, Ch.2].

The use of ASMs on the one side allows one to explain in a uniform way the classical models of computation and the semantics of the basic programming concepts. Furthermore, these very same ASMs can be taught to support describing system behavior by succinct, purely mathematical (read: platform-independent) but intuitive operational models, which the practitioner can use for experimentation by running executable versions and for rigorous analaysis. How to present this in a course is what we are going to explain in the following sections.

2 Algorithms: Computability and Undecidability

In this section we outline how in terms of ASMs one can introduce in a uniform manner all the classical models of computation that are most frequently adopted in computation theory courses, namely automata that characterize (generate and accept) languages of the Chomsky hierarchy. We show how simple structured ASMs, defined in terms of sequentialization and iteration of components, can be used to compute the recursive functions (Structured Programmming Theorem). We indicate a coding-free five-lines undecidability proof for the termination problem of programs of any programming language satisfying some natural closure properties and point to a resource-bound-aware generalization of the Church-Turing thesis in terms of ASMs. We have experienced that this core material can be covered in two weeks (6-8 lecture and exercise hours). Details for most of the material and further references can be found in the papers [16,29] and in chapters 7 and 4 of the AsmBook [30].

2.1 Automata and Grammars

We outline here how to uniformly define, in terms of simple ASMs, the classical automata that are related to the grammars forming the Chomsky hierarchy. We concentrate upon deterministic machine versions, from which the non-deterministic counterparts are obtained by governing the *Rules* to select from by a **choose** operator in the form **choose** $R \in Rules$ **in** R.

Finite Automata. Deterministic Mealy and Moore automata can be introduced as control state ASMs, with one *out*put function and a monitored *in*put function, where every rule has the following form (in the case of Moore automata one has *skip* instead of the output assignment):

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

If one prefers to write 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", one obtains the following guard-free FSM rule scheme for updating $(\mathit{ctl\_state}, \mathit{out})$, where the parameters *Nxtctl* and *Nxtout* are the two projection functions which define the program table, mapping 'configurations' (i, a) of the current control state and the currently read *in*put to the next control state j and *out*put b.

```
$\textsc{MealyFsm}(\mathit{in}, \mathit{out}, \mathit{Nxtctl}, \mathit{Nxtout}) =$
  $\mathit{ctl\_state} := \mathit{Nxtctl}(\mathit{ctl\_state}, \mathit{in})$
  $\mathit{out} := \mathit{Nxtout}(\mathit{ctl\_state}, \mathit{in})$
```

We like to discuss important specializations of FSMs through exercises where the students are asked to formalize variations of the above scheme. We give here three simple examples, more can be found in [16]. To formalize an input tape which is scanned piecemeal it suffices to change the monitored function *in* into a shared one which is supposed to be initialized by the environment and is at each step updated by the rule. To obtain *2-way automata* it suffices to include into the instructions also *Moves* (of the position *head*) of a reading device on the input tape *in*—so that *in*(*head*) represents the currently read part of the input tape *in*—and to add updates of the *head* position.

```
TwoWayFsm($in, out, Nxtctl, Nxtout, Move, head$) =
  MealyFsm($in(head), out, Nxtctl, Nxtout$)
  $head := head + Move(ctl\_state, in(head))$
```

In *timed automata* the 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: $time_{\Delta} = occurrenceTime(in) - occurrenceTime(previousIn)$. The firing of transitions may be subject to clock constraints[5] and includes clock updates (resetting a clock or adding to it the last input time difference). Thus timed automata can be defined as specialized FSMs with rules of the following form:

```
Fsm($i$, if $Reading(a)$ then $ClockUpdate(Reset), j$)
where
  $Reading(a) = (in = a$ and $Constraint(time_{\Delta}) = true)$
  $ClockUpdate(Reset) =$
    forall $c \in Reset$ do $c := 0$
    forall $c \notin Reset$ do $c := c + time_{\Delta}$
```

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*. In the following formulation of the form of PDA-rules, the optional input-reading or stack-reading are enclosed in []; the meaning of the *stack* operations *push*, *pop* is the usual one.

```
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. The Turing machine combines readable *in*put and writable *out*put of a two-way FSM into one read/write *tape* memory, identifying *in* and *out* with *tape*. This is a good reason to rename the function *Nxtout* of the TwoWayFsm to *Write*.

```
TuringMachine($tape, Nxtctl, Write, Move, head$) =
  TwoWayFsm($tape, tape(head), Nxtctl, Write, Move, head$)
```

Wegner's *interactive Turing machines* [66] in each step can additionally receive some environmental *input* and yield *output* to the environment. So they are an extension of the TuringMachine by an additional *input* parameter of the program table functions *Nxtctl*, *Write*, *Move* and by an additional output action[6]. The output action may consist in writing the output on an in-out tape; the input can be a combination of preceding inputs/outputs with the new user input, it also may be a stream vector $input = (inp_1, \ldots, inp_n)$ (so-called multiple-stream machines).

[5] 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.

[6] When introducing additional parameters we write f_p for the function defined by $f_p(x) = f(p, x)$.

```
TuringInteractive($tape$, $Nxtctl$, $Write$, $Move$, $head$, $input$) =
  TuringMachine($tape$, $Nxtctl_{input}$, $Write_{input}$, $Move_{input}$, $head$)
  Output($input$, $ctl\_state$, $tape(head)$)
```

Numerous other variations of *Turing-like machines* appear in the literature, e.g. computationally equivalent ones like the k-tape or the n-dimensional Turing machines, the machines of Wang, Minsky, Sheperdson and Sturgis, Scott, Eilenberg, the substitution systems of Thue, Post, Markov, etc., but also weaker machines like the linear bounded Turing machines. Their definitions can be covered by exercises where the students are asked to appropriately instantiate the following scheme of which also the above classical TuringMachine is a specialization. The characteristic feature of every TuringLikeMachine is that in each step, placed in a certain *pos*ition of its *mem*ory, it reads this *mem*ory in the *env*ironment of that *pos*ition (which may be requested to satisfy a certain *Cond*ition) and reacts by updating *mem* and *pos*. Therefore the rules of each TuringLikeMachine(mem, pos, env) are all of the following form:

```
Fsm($i$, **if** $ReadingCond$ **then** Update $(mem(env(pos)), pos), j$)
  **where** $ReadingCond = Condition(mem(env(pos)))$
```

Details of how to instantiate this scheme to the classical machine or substitution systems can be found in [16]. For Chomsky grammars see also Section 3. As example we illustrate how the standard Turing machine is extended to alternating Turing machines, namely by adding new types of control states whose role is to spawn trees for subcomputations, which upon termination are accepted or rejected. The *existential* and *universal* control states play the role of tree roots where subcomputations are spawned; they differ in the way their yield is collected upon termination to either accept or reject the spawned subcomputations. Directly *accept*ing or *reject*ing control states appear at the leaves of such subcomputations. Different subcomputations of an alternating Turing machine, whose program is defined by the given functions *Nxtctl*, *Write*, *Move* used by all subcomputations, are distinguished by parameterizing the machine instances by their executing agents a, obtaining TuringMachine(a) from the standard TuringMachine by replacing the dynamic functions *ctl_state*, *tape*, *head* with their instances $a.ctl\_state$ and $a.tape$, $a.head$. For the details of the new submachines see [16].

```
AlternatingTm($tape$, $Nxtctl$, $Write$, $Move$, $head$) =
  **if** $type($**self**$.ctl\_state) = normal$ **then**
    TuringMachine($tape$, $Nxtctl$, $Write$, $Move$, $head$)(**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**$.ctl\_state)$
```

We conclude with a short discussion of Petri nets. In their most general understanding they are an instance of multi-agent asynchronous ASMs, namely 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. Technically speaking, 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.

```
PetriTransition =
**if** $Cond(prePlaces)$ **then** $Updates(postPlaces)$
**where**
  $Updates(postPlaces) =$ a set of function updates
```

2.2 Structured ASMs for Recursive Functions

No computation theory course should miss a discussion of the notion of recursion, independently of its realizations in programming languages. This is a place to exploit the elegance of purely functional equational definitions, characterizing primitive recursive and general recursive functions in the Gödel-Herbrand style. We however replace the tedious still widely used programming of the Turing machine to compute recursive functions by the introduction of so-called *turbo ASMs* defined in [29]. These machines solve in a natural way the problem to incorporate into the basic synchronous parallel computation model of basic ASMs the fundamental control structures for sequential execution and iteration (as well as of submachines). This provides a simple proof for Böhm-Jacopini's Structured Programming Theorem and more importantly a programming-language-independent general framework to discuss imperative and functional programmming concepts like composition, general recursion and procedure calls, parameterization, naming, encapsulation and hiding, local state, returning values, error handling, etc. We illustrate this by three very simple but characteristic turbo ASMs, namely to compute the composition and the minimalization operator for recursive functions and the classical recursive quicksort algorithm. Every lecturer will make up more examples tailored to his audience and taste. More details can be found in [17,41] or in chapter 4 of the AsmBook [30]. See also the interesting recent comparison of the transition from basic ASMs to turbo ASMs to the transition from FORTRAN/ALGOL58 to ALGOL60 in [53].

For computing recursive functions by turbo ASMs one can follow the standard way to compute them by structured programs for the register machine or Rödding's register operators, see [9, pages 19-23]. The turbo ASMs M we need can be defined, using only the composition operators **seq, while** defined in [29], from basic 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 0-ary input function in_M contains the number sequence which is given as the input for the computation of the machine, out_M receives the computed function value as output of M. If functions $g, h_1, \dots, h_m$ are computed by turbo ASMs $G, H_1, \dots, H_m$, then their composition f defined by $f(x) = g(h_1(x), \dots, h_m(x))$ is computed by the following machine $F = \textsc{FctCompo}$, where we write $out := F(in)$ as abbreviation for $in_F := in$ **seq** F **seq** $out := out_F$, similarly $F(in)$ for $in_F := in$ **seq** F:[7]

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

The formula for this structured program makes the order explicit in which the subterms in the defining equation for f have to be evaluated. 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 completely independent from each other, though all of them have to terminate before the next "functional" step is taken, consisting in passing the sequence of out_{H_i} as input to the constituent function g. Finally the value of g on this input is computed and assigned as output to out_F.

In the same way, if f is defined from g by minimalization, i.e. $f(x) = \mu y(g(x, y) = 0)$, and if a turbo ASM G computing g is given, then the following machine $F = \textsc{MuOperator}$ computes f. The start submachine computes $g(x, rec)$ for the initial recursor value 0, and the iterating machine computes $g(x, rec)$ for increased values of the recursor until for the first time 0 shows up as computed value of g, in which case the reached recursor value is set as output.

```
$\textsc{MuOperator}(G) = \{G(in_F, 0),\ rec := 0\}$ **seq**
  (**while** $(out_G \neq 0)$ $\{G(in_F, rec + 1),\ rec := rec + 1\}$) **seq**
    $out_F := rec$
```

[7] The set denotes the rules of an ASM which are to be executed in parallel.

The turbo ASM below for Quicksort follows its well-known recursive definition: FIRST partition the *tail* of the list L 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 independently of each other, THEN *concatenate* the results taking $head(L)$ between them.

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

The structuring principles of structured programming are directly reflected by the turbo ASM operators **seq** and **while**. Also more sophisticated structuring principles are supported by ASMs. We mention here the decomposition of systems into components [14], the instantiation of parameterized submachines [29] and the organization of classes into an inheritance hierarchy by a subclass (compatibility) relation as formalized for Java and the Java Virtual Machine in [62].

2.3 Undecidability and Church-Turing Thesis

In this section we show the general undecidability proof we use, starting from scratch, using half a page and five lines of proof, for the termination problem of any class of universal programs, as a preparation for the epistemological discussion of the intrinsic boundaries of the notion of virtual machine computation as well as of its wide range (Church-Turing thesis and its generalization by ASMs).

Undecidability. Consider any programming language L that satisifies the following four closure properties (which are known to suffice to be computationally universal, see for example the above turbo ASMs computing recursive functions)[8]:

- L provides a notion of *sequential execution*, for definiteness say in the form of an operator **seq** such that $P, Q \in L$ implies P **seq** $Q \in L$.
- L provides a notion of program *iteration*, for definiteness say $P \in L$ implies **while** $b = 1$ $P \in L$, where b is a program variable with boolean values 0,1.
- L provides a notion of *calling* a program for given input, for definiteness say in the form that $P \in L$ implies **Call** $P(in) \in L$, where in is a program input variable.
- L permits program text as input for programs[9].

We denote for the given L as usual a) by *Halt(p,in)* that program p started with input in terminates, and b) by 'p computes H' that for every input in, *Halt(p,in)* and upon termination the output variable, say out, satisfies $out = 1$ if $H(in)$ and $out = 0$ otherwise. We prove by contradiction that there is no L-program h that computes the *Halt* predicate for L-programs. In fact, otherwise by the above listed closure properties the following program $\textsc{Diag}$ with input variable in and output variable out would be an L-program (draw the flowchart diagram visualizing the diagonal argument):

$$\textsc{Diag} = \textbf{Call } h(in, in) \textbf{ seq } (\textbf{while } out = 1 \;\; \textbf{Call } h(in, in))$$

Due to its definition and to the definition of the *Halt*ing property, this program would satisfy the contradictory property that $Halt(\textsc{Diag},\textsc{Diag})$ is true if and only if $Halt(\textsc{Diag},\textsc{Diag})$ is not true.

Church-Turing Thesis. After having shown the above undecidability proof, we link this result to the ASMs for standard machines, algorithms, programming constructs and virtual machine or general system design models to motivate the discussion of what became known as ASM Thesis, stating roughly that for any algorithm (in the intuitive sense of the word) an ASM can be defined which simulates this algorithm with the same number of steps (up to a constant factor). This thesis

[8] We thank Francesco Romani for having pointed out an oversight in an earlier version of this argument

[9] This condition is of technical nature: it allows one to avoid the discussion of ways of coding of program text as input for other programs.

generalizes the Church-Turing Thesis and provides a chance for the lecturer to attract students who have a mind for epistemological questions and are not afraid of mathematical reasoning. In fact for the case of so-called sequential and synchronous parallel ASMs a proof for the thesis can be given from a small number of postulates, as shown in [46,8] (for the sequential case one may wish to also consult the expositions in [30, Ch.7.2] and [55]).

3 Principles of Programming Languages

The literature offers a rich variety of ASM models the lecturer can choose from for every major programming language paradigm, whether logical, functional, imperative, object-oriented, with or without parallelism. This includes the complete definition (together with a mathematical analysis[10]) of real-life programmming languages and their implementations, like Java [62], C# [21], SDL-2000 [49,45] and the (forthcoming OASIS standard for the) Business Process Execution Language for Web Services BPEL4WS [39,65,38]. It also includes modeling various forms of parallelism and thread handling, for example the ones in Occam [20] or C# [61].

Instead of focussing on a particular language, an alternative is to define an ASM to interpret high-level object-oriented programmming constructs, structured into layered modules of by and large orthogonal language features for an *imperative core* (related to sequential control by while programs, built from statements and expressions over the simple types), *static class features* (realizing procedural abstraction with class initialization and global (module) variables), *object-orientation* (with class instances, instance methods, inheritance), *exception handling*, *concurrency*, *delegates* (together with events, properties, indexers, attributes), *unsafe* code with *pointer arithmetic*. In a next step one can then instantiate that interpreter to one for a particular language, as done in [31] to concretely compare Java and C#, distilling their similarities and differences.

As a small illustrative example we extract here from the ASM for the core of Prolog in [26] a basic tree generation and traversal ASM BACKTRACK. This machine yields plenty of modeling and refinement exercises, namely to define by variations of the model the core of ISO Prolog [19] and of its various extensions (e.g. IBM's Protos-L [7,6] and constraint logic programming language CLP(R) [28]), of a functional programming language like Babel [23], of context free and of attribute grammars [50], etc. For details see [16].

The machine dynamically constructs a tree of alternatives and controls its traversal. In control state $mode = ramify$, 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 (static or monitored) 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.

```
BACKTRACK =
  RAMIFY
  SELECT
```

[10] The lecturer who speaks to students with interest in verification can expand here as much as he wants. For example, the detailed mathematical analysis of Java/JVM in [62] includes proofs that Java is type-safe, that the compiler is correct and complete and that the bytecode verifier is complete and sound. Also a mechanical verification of such ASM-based mathematical proofs can be presented, see for example the KIV-verification reported in [56] for the ASM-based correctness proof in [27] for the Prolog-to-WAM implementation.

```
RAMIFY =
  if $mode = ramify$ then
    let $k = |alternatives(Params)|$
    let $o_1, \dots, o_k = new(NODE)$
      $candidates(currnode) := \{o_1, \dots, o_k\}$
      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($next(candidates(currnode))$,
    $candidates(currnode)$)
```

The above mentioned exercises consist in finding instantiations of the here not furthermore specified functions *alternatives*, *next* and of the underlying system *env*ironment and execution engine EXECUTE. Instantiating the ASM to one describing tree adjoining grammars generalizes 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).

4 High-Level System Design and Analysis

In the very same way we have illustrated in Section 2 ASMs to capture classical models of computation, one can show that they can also be used to model the basic semantical concepts of executable high-level design languages (like UNITY and COLD), of widely used state-based specification languages (e.g. B [2] or SCR [47,48]), of dedicated virtual machines as well as of axiomatic logic-based or stateless modeling systems. The reader can find details in [16].

This naturally leads to continue the course with advanced courses on practical system engineering methods. We have described in [13,15] a program for teaching the ASM system design and analysis method, which over the last decade has been elaborated upon the basis of ASMs as rigorous notion of virtual machine and which within a single, precise yet simple, conceptual framework naturally supports and uniformly links the major activities which occur during the typical software life cycle, namely:

- **Requirements capture** by constructing satisfactory *ground models*, i.e. accurate high-level system blueprints, serving as precise contract and formulated in a language which is understood by all stakeholders (see [11]).
- **Detailed design** by *stepwise refinement*, bridging the gap between specification and code design by piecemeal, systematically documented detailing of abstract models down to executable code (see [12]). This includes refinement steps which lead from a high-level ASM to an executable and therebey mechanically validatable ASM.

- **Validation** of models by their *simulation*, based upon the notion of ASM *run* and supported by numerous tools to execute ASMs (*ASM Workbench* [34], *AsmGofer* [57], C-based *XASM* [3], .NET-executable *AsmL* engine [40]).
- **Verification** of model properties by proof techniques, also tool supported, e.g. by KIV [56] or PVS [36,42] or Stärk's theorem prover [60] or model checkers [67,35,44].
- **Documentation** for *inspection*, *reuse* and *maintenance* by providing, through the intermediate models and their analysis, explicit descriptions of the software structure and of the major design decisions.

The lecturer can choose among a great variety of ASM-based modeling and analysis projects in such different areas as:

- industrial standardization projects: the above mentioned models for the forthcoming OASIS standard for BPEL [65], the ECMA standard for C# [21], the ITU-T standard for SDL-2000 [45], the IEEE standard for the hardware desing language VHDL93 [22], the ISO-Prolog standard [19],
- programmming languages: definition and analysis of the semantics and the implementation for the major real-life programmming languages, e.g. SystemC [54], Java and its implementation on the Java Virtual Machine [62], domain-specific languages used at the Union Bank of Switzerland [52], etc.
- architectural design: verification (e.g. of pipelining schemes [24] or of VHDL-based hardware design at Siemens [58, Ch.2]), architecture/compiler co-exploration [63,64],
- reengineering and design of industrial control systems: software projects at Siemens related to railway [18,25] and mobile telephony network components [33], debugger specification at Microsoft [4],
- protocols: for authentication, cryptography, cache-coherence, routing-layers for distributed mobile ad hoc networks, group-membership etc., focussed on verification,
- verification of compilation schemes and compiler back-ends [27,20,37,62],
- modeling e-commerce [1] and web services [39],
- simulation and testing: fire detection system in coal mines [32], simulation of railway scenarios at Siemens [25], implementation of behavioral interface specifications on the .NET platform and conformence test of COM components at Microsoft [5], compiler testing [51], test case generation [43].

The lecturer may also use instead the AsmBook [30] and the teaching material on the accompanying CD. The book introduces into the ASM method and illustrates it by textbook examples, which are extracted from the above listed real-life case studies and industrial applications.

References

1. S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. In *Proc. 17th ACM Sympos. Principles of Database Systems (PODS 1998)*, pages 179–187. ACM Press, 1998.
2. J.-R. Abrial. *The B-Book*. Cambridge University Press, Cambridge, 1996.
3. M. Anlauff and P. Kutter. Xasm Open Source. Web pages at `http://www.xasm.org/`, 2001.
4. M. Barnett, E. Börger, Y. Gurevich, W. Schulte, and M. Veanes. Using Abstract State Machines at Microsoft: A case study. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 367–380. Springer-Verlag, 2000.
5. M. Barnett and W. Schulte. Contracts, components and their runtime verification on the .NET platform. *J. Systems and Software*, Special Issue on Component-Based Software Engineering, 2002, to appear.
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 Trans. Computational Logic*, 2002.
9. 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.
10. 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)*, volume 1641 of *Lecture Notes in Computer Science*, pages 1–43. Springer-Verlag, 1999.
11. E. Börger. The ASM ground model method as a foundation of requirements engineering. In N.Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 145–160. Springer-Verlag, 2003.
12. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.
13. E. Börger. Teaching ASMs to practice-oriented students. In *Teaching Formal Methods Workshop*, pages 5–12. Oxford Brookes University, 2003.
14. E. Börger. Linking architectural and component level system views by abstract state machines. In C. Grimm, editor, *Languages for System Specification and Verification*, CHDL, pages 247–269. Kluwer, 2004.
15. E. Börger. Modeling with Abstract State Machines: A support for accurate system design and analysis. In B. Rumpe and W. Hesse, editors, *Modellierung 2004*, volume P-45 of *GI-Edition Lecture Notes in Informatics*, pages 235–239. Springer-Verlag, 2004.
16. E. Börger. Abstract State Machines: A unifying view of models of computation and of system design frameworks. *Annals of Pure and Applied Logic*, 2004, to appear.
17. 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 *Lecture Notes in Computer Science*, pages 218–228. Springer-Verlag, 2003.
18. E. Börger, H. Busch, J. Cuellar, P. Päppinghaus, E. Tiden, and I. Wildgruber. Konzept einer hierarchischen Erweiterung von EURIS. Siemens ZFE T SE 1 Internal Report BBCPTW91-1 (pp. 1–43), Summer 1996.
19. E. Börger and K. Dässler. Prolog: DIN papers for discussion. ISO/IEC JTCI SC22 WG17 Prolog Standardization Document 58, National Physical Laboratory, Middlesex, England, 1990.
20. E. Börger and I. Durdanović. Correctness of compiling Occam to Transputer code. *Computer Journal*, 39(1):52–92, 1996.
21. E. Börger, G. Fruja, V. Gervasi, and R. Stärk. A high-level modular definition of the semantics of C#. *Theoretical Computer Science*, 2004.
22. E. Börger, U. Glässer, and W. Müller. The semantics of behavioral VHDL'93 descriptions. In *EURO-DAC'94. European Design Automation Conference with EURO-VHDL'94*, pages 500–505, Los Alamitos, California, 1994. IEEE Computer Society Press.
23. 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, 1994.
24. 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 *Lecture Notes in Computer Science*, pages 151–187. Springer-Verlag, 1997.
25. E. Börger, P. Päppinghaus, and J. Schmid. Report on a practical application of ASMs in software design. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 361–366. Springer-Verlag, 2000.
26. E. Börger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 24:249–286, 1995.
27. 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*, volume 11 of *Studies in Computer Science and Artificial Intelligence*, chapter 2, pages 20–90. North-Holland, 1995.
28. 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.
29. 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 *Lecture Notes in Computer Science*, pages 41–60. Springer-Verlag, 2000.

30. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
31. E. Börger and R. F. Stärk. Exploiting abstraction for specification reuse. The Java/C# case study. In M. Bonsangue, editor, *Proc. FMCO'03*, Lecture Notes in Computer Science. Springer, 2004.
32. W. Burgard, A. B. Cremers, D. Fox, M. Heidelbach, A. M. Kappel, and S. Lüttringhaus-Kappel. Knowledge-enhanced CO-monitoring in coal mines. In *Proc. Int. Conf. on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA-AIE)*, pages 511–521, Fukuoka, Japan, 4–7 June 1996. .
33. G. D. Castillo and P. Päppinghaus. Designing software for internet telephony: experiences in an industrial development process. In A. Blass, E. Börger, and Y. Gurevich, editors, *Theory and Applications of Abstract State Machines*, Schloss Dagstuhl, Int. Conf. and Research Center for Computer Science, 2002.
34. G. Del Castillo. *The ASM Workbench. A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models*. PhD thesis, Universität Paderborn, Germany, 2001.
35. G. Del Castillo and K. Winter. Model checking support for the ASM high-level language. In S. Graf and M. Schwartzbach, editors, *Proc. 6th Int. Conf. TACAS 2000*, volume 1785 of *Lecture Notes in Computer Science*, pages 331–346. Springer-Verlag, 2000.
36. A. Dold. A formal representation of Abstract State Machines using PVS. Verifix Technical Report Ulm/6.2, Universität Ulm, Germany, July 1998.
37. A. Dold, T. Gaul, V. Vialard, and W. Zimmermann. ASM-based mechanized verification of compiler back-ends. In U. Glässer and P. Schmitt, editors, *Proc. 5th Int. Workshop on Abstract State Machines*, pages 50–67. Magdeburg University, 1998.
38. D. Fahland. Ein Ansatz einer Formalen Semantik der Business Process Execution Language for Web Services mit Abstract State Machines. Master's thesis, Humboldt-Universität zu Berlin, June 2004.
39. R. Farahbod, U. Glässer, and M. Vajihollahi. Specification and validation of the business process execution language for web services. In W. Zimmermann and B. Thalheim, editors, *Abstract Sate Machines 2004*, volume 3052 of *Lecture Notes in Computer Science*, pages 78–94. Springer-Verlag, 2004.
40. Foundations of Software Engineering Group, Microsoft Research. AsmL. Web pages at http://research.microsoft.com/foundations/AsmL/, 2001.
41. 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.
42. 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 *Lecture Notes in Computer Science*, pages 303–322. Springer-Verlag, 2000.
43. A. Gargantini and E. Riccobene. Using Spin to generate tests from ASM specifications. 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 263–277. Springer-Verlag, 2003.
44. A. Gawanmeh, S. Tahar, and K. Winter. Interfacing ASMs with the MDG tool. 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 278–292. Springer-Verlag, 2003.
45. U. Glässer, R. Gotzhein, and A. Prinz. Formal semantics of SDL-2000: Status and perspectives. *Computer Networks*, 42(3):343–358, June 2003.
46. Y. Gurevich. Sequential Abstract State Machines capture sequential algorithms. *ACM Trans. Computational Logic*, 1(1):77–111, July 2000.
47. 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 No. 99241, Schloss Dagstuhl, Int. Conf. and Research Center for Computer Science, 1999.
48. C. Heitmeyer. Software cost reduction. In J. J. Marciniak, editor, *Enc. of Software Engineering*. 2nd edition, 2002.
49. ITU-T. SDL formal semantics definition. ITU-T Recommendation Z.100 Annex F, International Telecommunication Union, November 2000.
50. D. E. Johnson and L. S. Moss. Grammar formalisms viewed as Evolving Algebras. *Linguistics and Philosophy*, 17:537–560, 1994.

51. A. Kalinov, A. Kossatchev, A. Petrenko, M. Posypkin, and V. Shishkov. Using ASM specifications for compiler testing. 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*, page 415. Springer-Verlag, 2003.
52. P. Kutter, D. Schweizer, and L. Thiele. Integrating domain specific language design in the software life cycle. In *Proc. Int. Workshop on Current Trends in Applied Formal Methods*, volume 1641 of *Lecture Notes in Computer Science*, pages 196–212. Springer-Verlag, 1998.
53. H. Langmaack. An ALGLO-view on TURBO ASM. In W. Zimmermann and B. Thalheim, editors, *Abstract Sate Machines 2004*, volume 3052 of *Lecture Notes in Computer Science*, pages 20–37. Springer-Verlag, 2004.
54. W. Mueller, J. Ruf, and W. Rosenstiel. An ASM-based semantics of systemC simulation. In W. Mueller, J. Ruf, and W. Rosenstiel, editors, *SystemC - Methodologies and Applications*, pages 97–126. Kluwer Academic Publishers, 2003.
55. W. Reisig. On Gurevich's theorem on sequential algorithms. *Acta Informatica*, 39(5):273–305, 2003.
56. G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM case study. *J. Universal Computer Science*, 3(4):377–413, 1997.
57. J. Schmid. Executing ASM specifications with AsmGofer. Web pages at `http://www.tydo.de/AsmGofer`.
58. J. Schmid. *Refinement and Implementation Techniques for Abstract State Machines*. PhD thesis, University of Ulm, Germany, 2002.
59. D. Scott. Definitional suggestions for automata theory. *J. Computer and System Sciences*, 1:187–212, 1967.
60. R. F. Stärk. Formal verification of the C# thread model. Department of Computer Science, ETH Zürich, 2004.
61. R. F. Stärk and E. Börger. An ASM specification of C# threads and the .NET memory model. In W. Zimmermann and B. Thalheim, editors, *Abstract State Machines 2004*, volume 3052 of *Lecture Notes in Computer Science*, pages 38–60. Springer-Verlag, 2004.
62. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001. .
63. J. Teich. Project Buildabong at University of Paderborn. `http://www-date.upb.de/RESEARCH/BUILDABONG/buildabong.html`, 2001.
64. J. Teich, R. Weper, D. Fischer, and S. Trinkert. A joint architecture/compiler design environment for ASIPs. In *Proc. Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES2000)*, pages 26–33, San Jose, CA, USA, November 2000. ACM Press.
65. M. Vajihollahi. High level specification and validation of the business process execution language for web services. Master's thesis, School of Computing Science at Simon Fraser University, March 2004.
66. P. Wegner. Why interaction is more powerful than algorithms. *Commun. ACM*, 40:80–91, 1997.
67. K. Winter. Model checking for Abstract State Machines. *J. Universal Computer Science*, 3(5):689–701, 1997.