Egon Börger (Pisa)

A uniform approach to teach the principles of

. computation

. programming

. system design and analysis

Università di Pisa, Dipartimento di Informatica, I-56127 Pisa, Italy
boerger@di.unipi.it

What are the basic concepts?

- **virtual machines** for any kind of computational system, whether stand-alone or cooperating: most general (universal) ‘architecture’
  – *states*: data structures for such machines and data sharing structures for their cooperation
- **programs** (algorithms) for most general virtual machines
  – *control structures*
  – *communication means*
  – *runs*, sequential or distributed, formed by executing instructions for state-transforming or communication steps
- **basic properties** like functionality, computational power, memory or time complexity, etc. What are the languages to appropriately express these properties?
- **basic means of analysis**: experimental validation and mathematical verification to establish properties of computational systems

The divide-and-conquer principle (separation of concerns)

- **stepwise refinement**: piecemeal introduction of design and verfication details, identifying orthogonal system elements
- **separation of design from analysis**
- **separation of different analysis types and levels**
  - separation of experimental *validation* (system simulation and testing) from mathematical *verification*
  - *distinction between verification levels* and the characteristic concerns each of it comes with, e.g.
    - reasoning for human inspection: proof ideas/sketches or completely carried out detailed proofs
    - rule-based reasoning systems: mechanical inferences, operated by humans or as computerized systems, interactively or automatically
  - separation of *static program analysis* from a *run-time-based analysis* of dynamic program properties (runtime verfication)

Starting with ASMs: Why?

- ASMs represent a **most general definition of VMs**, namely transition systems transforming structures, as evidenced by
  - over ten years of experience with modeling and analysing outstanding real-life virtual machines in terms of ASMs
  - Gurevich's ASM thesis, a resource-bound-aware generalization of the thesis of Church and Turing, and its proof from basic postulates
- ASMs provide a framework for a theoretically well-founded, coherent and uniform **practical combination of abstract operational descriptions with functional and axiomatic definitions**
  - eventually overcoming an alleged (unjustified and destructive) dichotomy between declarative and operational design elements
- ASMs based upon **three fundamental computational features**
  - conditional update (IF $Cond$ THEN $f(t_1, \ldots, t_n) := t$)
  - parallelism (simultaneous updates, **forall**-construct)
  - nondeterminism (**choose**-construct)

Classical automata as variations of ASMs

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

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

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

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

$\quad \textsc{MealyFsm}(in(head), out, Nxtctl, Nxtout)$

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

$\textsc{TuringMachine}(tape, Nxtctl, Write, Move, head) =$

$\quad \textsc{TwoWayFsm}(tape, tape(head), Nxtctl, Write, Move, head)$

$\textsc{TuringInteractive}(tape, Nxtctl, Write, Move, head, input) =$

$\quad \textsc{TuringMachine}(tape, Nxtctl_{input}, Write_{input}, Move_{input}, head)$

$\quad \textsc{Output}(input, ctl\_state, tape(head))$

Making modes explicit in control state ASMs

$$\textsc{Fsm}(i, \textbf{if } cond \textbf{ then } rule, j) =$$
$$\quad \textbf{if } ctl\_state = i \textbf{ and } cond \textbf{ then}$$
$$\qquad rule$$
$$\qquad ctl\_state := j$$

Written in this way, $\textsc{MealyFsm}$ can be defined as a set of control state ASM rules of the following form:

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

Classical automata as variations of control state ASMs

$$\textsc{TimedAutomatonInstr}(i, a, Reset, j) =$$

$$\quad \textsc{Fsm}(i, \textbf{if } Reading(a) \textbf{ then } ClockUpdate(Reset), j) \textbf{ where}$$

$$\quad\quad Reading(a) = (in = a \textbf{ and } Constraint(time_\Delta) = true)$$

$$\quad\quad ClockUpdate(Reset) =$$

$$\quad\quad\quad \textbf{forall } c \in Reset \textbf{ do } c := 0$$

$$\quad\quad\quad \textbf{forall } c \notin Reset \textbf{ do } c := c + time_\Delta$$

$$\textsc{PushDownAutomatonInstr}(i, a, b, w, j) =$$

$$\quad \textsc{Fsm}(i, \textbf{if } Reading(a, b) \textbf{ then } StackUpdate(w), j) \textbf{ where}$$

$$\quad\quad Reading(a, b) = [in = a] \textbf{ and } [top(stack) = b]$$

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

$\textsc{TmLikeInstr}(mem, pos, env) =$ (Thue,Post,Wang,Minsky,...)

$$\quad \textsc{Fsm}(i, \textbf{if } ReadingCond \textbf{ then } \textsc{Update}(mem(env(pos)), pos), j)$$

$$\quad\quad \textbf{where } ReadingCond = Condition(mem(env(pos)))$$

Two advanced examples

$\textsc{PetriTransition} =$

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

where

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

$\textsc{AlternatingTm}(tape, Nxtctl, Write, Move, head) =$

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

$\textsc{TuringMachine}(tape, Nxtctl, Write, Move, head)(\mathbf{self})$

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

$\textsc{AltTmSpawn}(\mathbf{self})$

$\textsc{TmYieldExistential}(\mathbf{self})$

$\textsc{TmYieldUniversal}(\mathbf{self})$

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

$yield(\mathbf{self}) := type(\mathbf{self}\ .ctl\_state)$

Spawning submachine of alternating TMs

$\textsc{AltTmSpawn}(a) = \textbf{if } a.mode = running \textbf{ then}$

$\quad \textbf{forall } j \in Nxtctl(a.ctl\_state, a.tape(a.head)) \textbf{ do}$

$\quad\quad \textbf{let } b = new(Agent) \textbf{ in}$

$\quad\quad\quad \textsc{Activate}(b, a, j),\ parent(b) := a$

$\quad a.mode := idle$

$\textsc{Activate}(b, a, j) =$

$\quad b.mode := running,\ b.yield := undef,\ b.ctl\_state := j$

$\quad \textsc{CopyTapeProgram}(b, a)$

$\textsc{CopyTapeProgram}(b, a) =$

$\quad \textbf{forall } pos \in domain(a.tape) \textbf{ do } b.tape(pos) := a.tape(pos)$

$\quad b.head := a.head$

$\quad b.Nxtctl := a.Nxtctl,\ b.Write := a.Write$

$\quad b.Move := a.Move,\ b.type := a.type$

Existential and Universal submachines of alternating TMs

$\text{TmYieldExistential}(a) =$

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

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

      $yield(a) := reject$

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

      $yield(a) := accept$

$\text{TmYieldUniversal}(a) =$

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

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

      $yield(a) := accept$

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

      $yield(a) := reject$

Turbo ASMs and submachines

Turbo ASMs defined via **seq, while** from basic ASMs with only

- one non-controlled (a 0-ary **in**-put) function; its value is fixed by the initial state
- one (a 0-ary) **out**-put function
- as static fcts only the initial fcts of recursion theory U_i^n, C_i^n, S

The definition (see Börger/Schmid CSL'2000) matches the *synchrony hypothesis* of synchronous programming languages (a sequence of micro-steps makes up an instantaneous program reaction). It yields a succinct treatment of recursive functions.

Turbo submachines $R(x_1, \ldots, x_n) = body$ with meaning:

$$Yield(R(a_1, \ldots, a_n) = Yield(body(a_1, \ldots, a_n))$$

This definition allows one to abstractly introduce the concepts of encapsulation, hiding, local state, error handling, returning values, recursion, etc.

Turbo ASMs computing recursive functions

$\textsc{FctCompo}(G, H_1, \ldots, H_m) =$

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

$\textsc{PrimitiveRecursion}(G, H) =$ **let** $(x, y) = in_F$ **in**

$\quad \{ival := G(x),\ rec := 0\}$ **seq**

$\quad ($**while** $(rec < y)\ \{ival := H(x, rec, ival),\ rec := rec + 1\})$ **seq**

$\quad out_F := ival$

$\textsc{MuOperator}(G) = \{G(in_F, 0),\ rec := 0\}$ **seq**

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

$\quad out_F := rec$

where

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

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

Halting problem for any computation-universal language

Assume the following closure properties for a language L:

- *sequential composition*: $P, Q \in L$ implies $(P \textbf{ seq } Q) \in L$
- *iteration*: $P \in L$ implies $(\textbf{while } b = 1 \ P) \in L$ for boolean valued b
- *calling*: $P \in L$ implies $\textbf{Call } P(in) \in L$ for input variable in
- L-programs can have program text as input

Then for every $h \in L$ the following program $\textsc{Diag}$ with input/output variables in, out is an L-program:

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

Therefore there is no $h \in L$ computing the *Halt* predicate for L-programs since otherwise:

$$Halt(\textsc{Diag}, \textsc{Diag}) \text{ iff } \textbf{not } Halt(\textsc{Diag}, \textsc{Diag})$$

- *Halt(p,in)* iff p started with input in eventually terminates
- *p computes H* iff $Halt(p, in)$ **and** $out = H(in)$ upon termination for every input in

ASM models for programming languages/constructs

The literature contains **ASM models for** languages of **every major programming paradigm**:

- object-oriented: Java, C#, C++, Oberon
- design languages: BPEL (web services), SDL2000 (telecommunication), SystemC, VHDL'93, PHDL, Analog VHDL, Verilog (hardware)
- imperative: C, Modula-2, Cobol
- parallel: PVM, Occam, Parlog, Concurrent Prolog, Guarded Horn Clauses, Pandora, CHAM, etc.
- functional: Standard ML, Babel
- logical: Prolog, Prolog III, Protos-L, Gödel, CLP(R), etc.

The ASM models for Java/C# come as **hierarchy of submodels** (for imperative, procedural, object-oriented, exception handling, concurrency, etc. features) **isolating single programmg constructs** and **instruction patterns** one can describe independently of each other

ASM models for high-level system design constructs

The literature contains ASM models for the core or fundamental constructs of

- some executable high-level design languages, e.g. UNITY, COLD
- some widely used state-based specification languages, e.g. B, SCR (Parnas tables), Petri nets
- numerous dedicated real-life virtual machines

Exploiting ASMs for high-level system design and analysis

The ASM method naturally supports and uniformly links

- within a single *precise yet simple conceptual framework*

the major activities occuring during the software life cycle

- **requirements capture** by constructing rigorous *ground models*, i.e. accurate concise high-level system blueprints (contracts)
- **architectural and component design** bridging the gap between specification and code by *piecemeal, systematically documented detailing* of abstract models via intermediate models to code
- **validation** of models by their tool-supported *simulation*
- **verification** of model properties by tool-supported *proof techniques*
- **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*

Variety of applications of ASMs (1)

- **industrial standards**: *ground models* for the standards of
  - OASIS for Business Process Execution Language for Web Services
  - ECMA for C#
  - ITU-T for SDL-2000
  - IEEE for VHDL93
  - ISO for Prolog
- **design, reengineering, testing of industrial systems**:
  - railway and mobile telephony network component software at Siemens
  - fire detection system in German coal mines
  - implementation of behavioral interface specifications on the .NET platform and conformence test of COM components at Microsoft
  - compiler testing and test case generation tools

Variety of applications of ASMs (2)

- **programmming languages**: definition and analysis of the semantics and the implementation for the major real-life programmming languages, among many others for example
  - SystemC
  - Java/JVM (including bytecode verifier) and C#
  - domain-specific languages used at the Union Bank of Switzerland

  including the verification of numerous compilation schemes and compiler back-ends
- architectural design: verification (e.g. of pipelining schemes or of VHDL-based hardware design at Siemens), architecture/compiler co-exploration
- protocols: for authentication, cryptography, cache-coherence, routing-layers for distributed mobile ad hoc networks, group-membership etc.
- modeling e-commerce and web services

References and course material: AsmBook

E. Börger and R. F. Stärk: Abstract State Machines

Springer 2003. pp.X+438.

Slides for courses on single chapters, themes and case studies are to be found in ppt and pdf format on the CD coming with the book and are also downloadable from the website:

http://www.di.unipi.it/AsmBook/

Hello World