Egon Börger (Pisa)

Teaching ASMs

to practice-oriented students with limited mathematical background

References and course material can be found in the 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/

3 principles for using rigorous mathematical methods ('FMs')

- **Be problem and practice oriented** (a didactical concern)
  - *Problem orientation*. Study complex real-life systems by mathematical methods, e.g. industrial case studies, real-life architectures and programming languages, relevant and widely-used protocols
  - *Practice orientation*. Use ordinary mathematical notation and start with an intuitive "working" definition of FMs, one a practitioner can rely upon in his everyday work, without needing a PhD in logic
- **Use state-based run-time oriented abstractions**
  - *supporting the practitioners' understanding of system behavior* by succinct, purely mathematical (read: platform-independent) but intuitive operational models
  - *supporting experimentation with abstract models* instead of code, using simulation and verification tools supporting validation and verification of abstract models

3 principles for using rigorous math. methods (Cont'd)

- **Separate different concerns**
  - *Separate design from analysis* avoiding restrictions imposed by specific verification or validation methods
    - to keep the design space open
    - to keep structuring the design space open
  - *Separate different analysis types and levels* (as we are used to distinguish different design levels)
    - separate experimental validation (system simulation and testing) and mathematical verification
    - separate distinct levels of verification
      - reasoning for human inspection: proof ideas or proof sketches or complete mathematical proofs
      - mechanical design justification: interactive or automatic (theorem proving or model checking)

3 reasons to use accurate methods in practice: abstraction

- **Show the practical yield of abstractions**, namely their help to
  - *exhibit bugs and gaps in real-life systems* found through a verification attempt or a simulation that exhibits a different behavior than the expected or the real one
  - *provide maximal freedom to the implementor* without hiding relevant system features,
  - *provide precise flexible descriptions*, at the appropriate level of detailing and ready for reuse (in versioning, extensions, adaptations to different application domains)
  - *simplify* the design itself or its analysis. BUT: avoid to abstract from relevant system details.

3 reasons to use accurate methods in practice: ground models

- **Show the practical usefulness of validatable and verifiable "ground models"**
  - construct behavioral system 'blueprints' to rigorously capture the intended meaning of the informal requirements
  - use validation and verification of system behavior to exhibit, prior to coding
    - ambiguities,
    - inconsistencies,
    - incompleteness,
    - misunderstanding (by humans)

3 reasons to use accurate methods in practice: refinement

- **Show the practical role of refinement**: a systematic way to
  - *separate orthogonal design decisions* as support for a modular system development, driven by design-for-change and design-for-reuse
  - *structure the design space* by defining precise interfaces for the system decomposition (software architecture)
  - *make design decisions* and the design structure communicatable and *documentable* for later reference
  - *explain different system views* (e.g. for customer, developer, end user, maintainer) in a coherent manner
  - *port generic programs* to their instantiations in different programmming languages and on different platforms in a semantically transparent way

Basic ASMs: A Definition for the Working System Engineer

- **Transition system transforming structures** (sets of function tables representing 'abstract states') by rules of the form

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

  to be executed in parallel (abstract from unnecessary sequentialization)
- Allow also non-determinism and unbounded synchronous parallelism:

  choose x **with** $Property$ **in** $Rule$
  forall x **with** $Property$ **do** $Rule$

Asynchronous (distributed) ASMs

Generalize runs from sequences of moves of a basic ASM to **partial orders** of moves of multiple agents, each executing a 'basic' ASM, subject to a natural *coherence condition*.

Alternative Definition: (Control State) ASMs Extending FSMs

'internal states' i are generalized to structures

These structures contain in particular the 'control state' ctl which may assume as value each of the internal states $ctl = i$

transitions are generalized to synchronous parallel updates of locations

including the update of ctl. A **location** is a table entry (pair of function name and an argument), a **structure** is a set of function tables, where a function table is an association of values to each location, also called the 'interpretation of the function (name)' in the structure

Some real-life case studies

- **Control systems**: ground models for Production Cell (validated and verified), Steam Boiler (validated), Falko (train scheduling and control software at Siemens: validated), ...
- **Architectures**: verifying pipelining of DLX, APE100, architecture and compiler co-generation project (Teich), PVM, ...
- **Programming languages**: semantics and compiler verification for
  - Prolog on the WAM, Occam on the Transputer, Java on the JVM
  - VHDL, SystemC, SDL-2000, ...
- **Reuse of ASMs**: adapting models and correctness proofs
  - for Prolog/WAM to CLP(R)/CLAM and Protos-L/PAM
  - for Java to C#
- **Protocol verification**: Kermit, group membership, (mobile) network protocols, Kerberos (authentication), Needham-Schroeder (cryptographic), cache-coherence for FLASH multiprocessor, ...
- ... (See AsmBook)

Tool Support for Simulation and Testing of ASMs

- ASM engines executing certain classes of ASMs:
  - ML-based **Workbench** (G. Del Castillo, 2nd half of 90'ies), extensively used for testing purposes and user-scenario simulations in the Falko project at Siemens/Munich (5'98-3'99)
  - Gofer-based **AsmGofer** (J. Schmid, 1998-2000), used among others for experimenting with the Java/JVM models in Jbook
  - C-based **XASM** (M. Anlauff, 1998-2001), used with Montages and Teich's architecture and compiler co-generation method
  - .NET-based **AsmL** (FSE at MSR Redmond, since 2000) used in particular for ground model validation at MS
- Coding-free user-scenario simulation: provide values of external fcts from user-scenario (Exl: Falko project) (student homework!)
- Code ASMs directly in your favorite language (student homework!)
- Compile classes of ASMs, e.g. to C (Anlauff), C++ (J.Schmid), VHDL (J.Schmid), C# (FSE), ...

Tool Support for Verification of ASMs

- Interactive theorem provers
  - PVS (verified compiler back-ends: Verifix project)
  - KIV (Prolog-to-WAM compilation completely verified)
  - Isabelle (some steps of Prolog-to-WAM compilation verified)
- Model checkers (SMV model checker linked to ASM Workbench)
- Logic for ASMs (Stärk/Nanchen, implementation ongoing by extension of the KeY theorem proving environment, University of Karlsruhe)

See AsmBook Ch.8 for details

Reference: The AsmBook

E. Börger and R. F. Stärk

Abstract State Machines

Springer 2003. pp.X+438.

Slides for lectures on single chapters, themes and case studies and for an entire course on the CD coming with the book and downloadable from the **AsmBook Website** (ppt and pdf):

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