Egon Börger (Pisa)

Contributions of the ASM method to program verification

and some future challenges

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

Origin: Gurevich's Idea of Sharpening Turing's Thesis

- *1985* Notices of the Am.Math.Soc. 6(4):317 (TR UMich 1984)
  First, we adapt Turing's thesis to the case when only devices with bounded resources are considered. Second, we define a more general kind of abstract computational device, called dynamic structures, and put forward the following new thesis: Every computational device can be simulated by an appropriate dynamic structure – of appropriately the same size – in real time; a uniform family of computational devices can be uniformly simulated by an appropriate family of dynamic structures in real time. In particular, every sequential computational device can be simulated by an appropriate sequential dynamic structure.
- *1995*: Definition of Evolving Algebras (Lipari Guide, OUP Book)
- *2000*: Proof of the sequential ASM thesis from three natural postulates (ACM Trans.Comp.Logic 1(1):77-111).
  – sequential ASMs correspond to quantifier-free interpretations in logic

1990-1992: Recognizing the Practical Relevance of ASMs

- *formulate* ASM ground models (read: system blueprints capturing requirements) & relevant properties in traditional math. terms, free from formalization concerns for a specific logic lg or proof calculus
- *validate*, experimentally, ground model properties and behavior by simulation, performing experiments as systematic attempts to
  - "falsify" the model in the Popperian sense against the to-be-encoded piece of reality
  - "inspect" ground model to check completeness and correctness wrt requirements
  - "check" model behavior by runtime verification and analysis, e.g. testing characteristic sets of scenarios
- *verify*, mathematically, desired ground model properties (e.g. consistency, resolving conflicting objectives in requirements)
- *refine* ground models in a mathematically verifiable way to compilable code via hierarchies of ASM refinements reflecting design details

ASM Ground Models and their Provably Correct Refinements

- ISO-PROLOG (method adopted for C, Cobol, Smalltalk, C++, ...)
  - horizontal refinements: Prolog core and built-in layers
    - adding *constraints*: Colmerauer's Prolog III and IBM's CLP(R)
    - adding *polymorphic types*: IBM's Protos-L (FACJ '96)
    - adding *functional* programming: Babel (Madrid)
    - adding *declarative* programming: Gödel (Lloyd/Hill)
    - adding *oo-features*: OO-Prolog (B. Müller, Oldenburg)
    - adding *parallelism*: Parlog, Concurrent Prolog, Guarded Horn Clauses, Pandora
  - vertical refinements:
    - Prolog2WAM: 12 refinement steps, correctness proofs KIV-verified
    - *reused* for CLP(R)2CLAM and Protos-L2PAM *models/proofs*
- ITU-T SDL-2000, OASIS BPEL4WS (2006), ECMA C# (TCS'05), IEEE VHDL93 refined to pictorial extension PHDL, to analog VHDL/Verilog (Toshiba 1998/9), SystemC and SpecC (2001)

Starting 1992: Practicability Test of ASMs beyond Progg Lgs

Experimenting with ASM models beyond language interpreters

- Architecture Design and VMs (method enhanced in Teich's architecture and compiler co-generation project since 2000)
  - *APE100*: programmer's view ground model refined to VLSI-implemented microprocessor with pipelining/VLIW parallelism
  - *DLX* one-instruction-at-a-time RISC processor ground model refined by standard pipelining methods: structural, data, control hazards
  - *PVM* (Oak Nat Lab): distributed ground model at C-interface with characteristic event handling and message-passing interface
  - *Transputer*: 14 refinement steps linking Occam programmers' ground model to instruction set architecture (Computer J. 1996)
    - Enhanced in German Verifix project for DEC-Alpha processor family and compiler back-ends based on realistic intermediate lgs
- Protocols: authentication, cryptography, cache-coherence, routing-layers for mobile networks, group-membership, etc.

Starting 1994: ASMs for Industrial System Development (a)

- From ASM ground models via refinement to control software:
  - *Steam Boiler* Dagstuhl Seminar 1995 & LNCS 1165 with goal to "contribute to a realistic comparison, from the point of view of practicality for applications under industrial constraints, of the major techniques which are currently available for formally supported specification, design, and verification of large programs and complex systems"
    - ASM ground model, checkable to capture the requirements
    - stepwise refined to C++ code controlling FZI simulator, each intermediate model reflecting some design decision
  - *Production Cell*: ASM ground model refined to C++ code to support
    - code inspection (Dagstuhl Seminar 1997), changes, maintenance (modularity, structural similarity of ground model and code)
    - standard verification (PVS) and validation (model checking) methods proving correctness, safety, performance, liveness, etc.

Starting 1994: ASMs in Industrial System Development (b)

- *Requirements Elicitation and Analysis*: Light control Dagstuhl Seminar 1999 & JUCS 6(7), 2000: ASM method efficient to capture, validate and document requirements by mix of rigorous, explicit ("formal") and interpretable implicit ("informal") language elements
- *Code Generation from ASM model*: reengineering project Falko (May'98-March'99) creating first prototypical ASM based industrial development environment supporting seamless flow from ASM ground model definition to compilable code (using Schmid's compiler to C++), including high-level testing (using Del Castillo's Workbench) and code maintenance
- *Modeling Web Services*: ASM ground model for Virtual Provider refined to Semantic Web Service Discovery model (SAP 2005/6)
- *Modeling patterns* for sw, communication, refinements: in progress
- *Integration of tool support* for validation, verification, documentation, maintenance: ..., AsmL (2000), CoreAsm (2006)

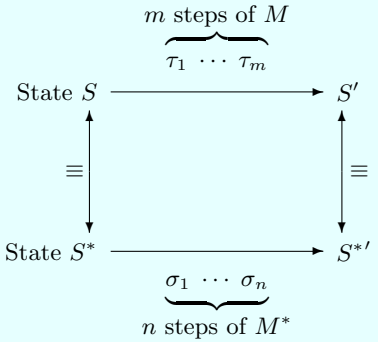
Refinement for Management of Design Decisions

Needed: generalization of classical refinement method (Wirth/Dijkstra)

- to cope with the "explosion of 'derived requirements' (the requirements for a particular design solution) caused by the complexity of the solution process" and encountered "when moving from requirements to design" (Glass 2003, Fact 26)
- to check and document by correctness proofs the design decisions taken in linking through various levels of abstraction the system architect's view (at the abstraction level of a blueprint) to the programmer's view (at the level of detail of compilable code)
  - *split checking complex detailed properties* into a series of simpler checks of more abstract properties and their correct refinement
  - provide systematic *rigorous system development documentation*, including behavioral information and needed internal interfaces by state-based abstractions

Foundation of Design Decisions: ASM Refinement Method

- allows one to systematically separate, structure and document orthogonal design decisions, relating different system aspects and system architect's to programmer's views
- supports
  - cost-effective system maintenance and management of system changes
  - piecemeal system validation and verification techniques



With an equivalence notion $\equiv$ between data in locations of interest in corresponding states.

The parameters for defining an ASM refinement step

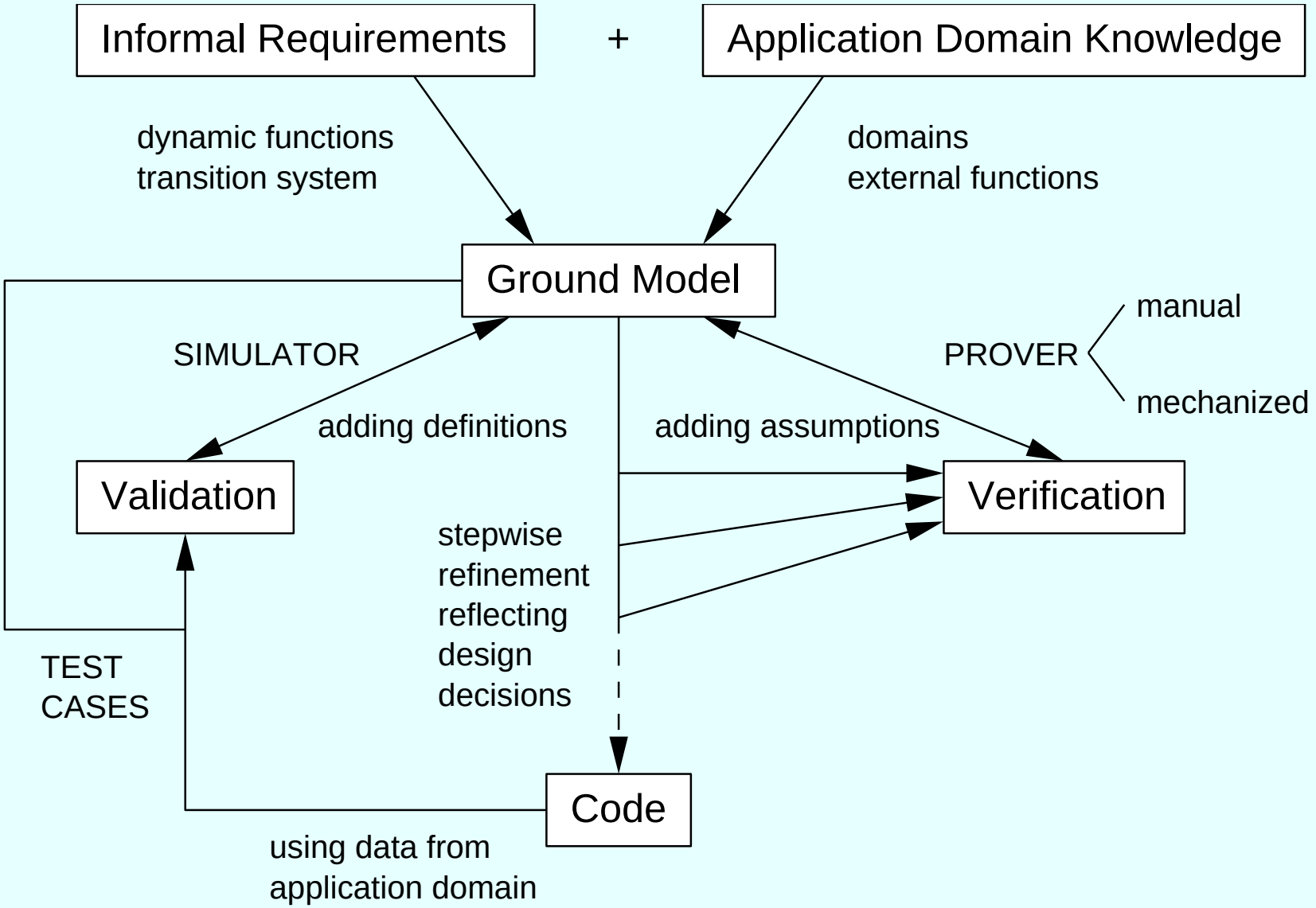
- a notion of *refined state*
- a notion of *states of interest* and of *correspondence* between M-states S and M^*-states S^* of interest, including usually initial/final states (if there are any)
- a notion of abstract *computation segments* $\tau_1, \ldots, \tau_m$, where each τ_i represents a single M-step, and of corresponding refined computation segments $\sigma_1, \ldots, \sigma_n$, of single M^*-steps σ_j, which in given runs lead from corresponding states of interest to (usually the next) corresponding states of interest (the resulting diagrams are called (m, n)-diagrams and the refinements (m, n)-refinements)
- a notion of *locations of interest* and of *corresponding locations*, i.e. pairs of (possibly sets of) locations one wants to relate in corresponding states
- a notion of *equivalence* $\equiv$ of the data in the locations of interest

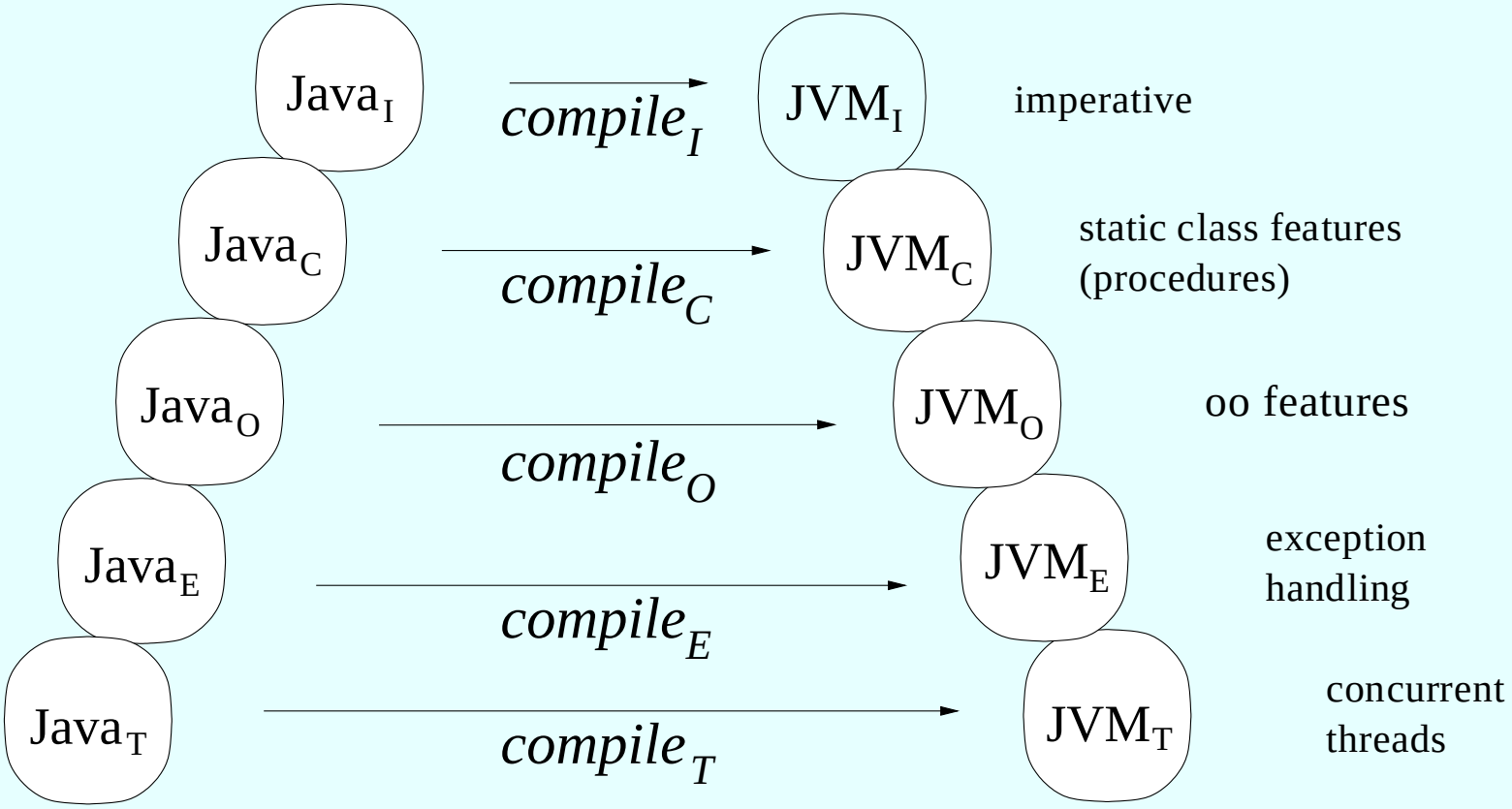
ASM Method: Resumé

Supports, within a single *precise yet simple conceptual framework*, and uniformly integrates the following activities/techniques:

- the major **software life cycle activities**, linking in a controllable way the two ends of the development of complex software systems:
  - **requirements capture** by constructing rigorous ground models
  - **architectural and component design** bridging the gap between specification and code by *piecemeal, systematically documented detailing* via stepwise refinement of models to code
  - **documentation** for *inspection*, *reuse*, *maintenance* providing, via intermediate models and their analysis, explicit descriptions of *software structure* and major *design decisions*
- the principal **modeling and analysis techniques**
  - dynamic (*operational*) and static (*declarative*) descriptions
  - **validation** (simulation) and **verification** (proof) methods *at any desired level of detail*

Models and methods in the ASM-based development process

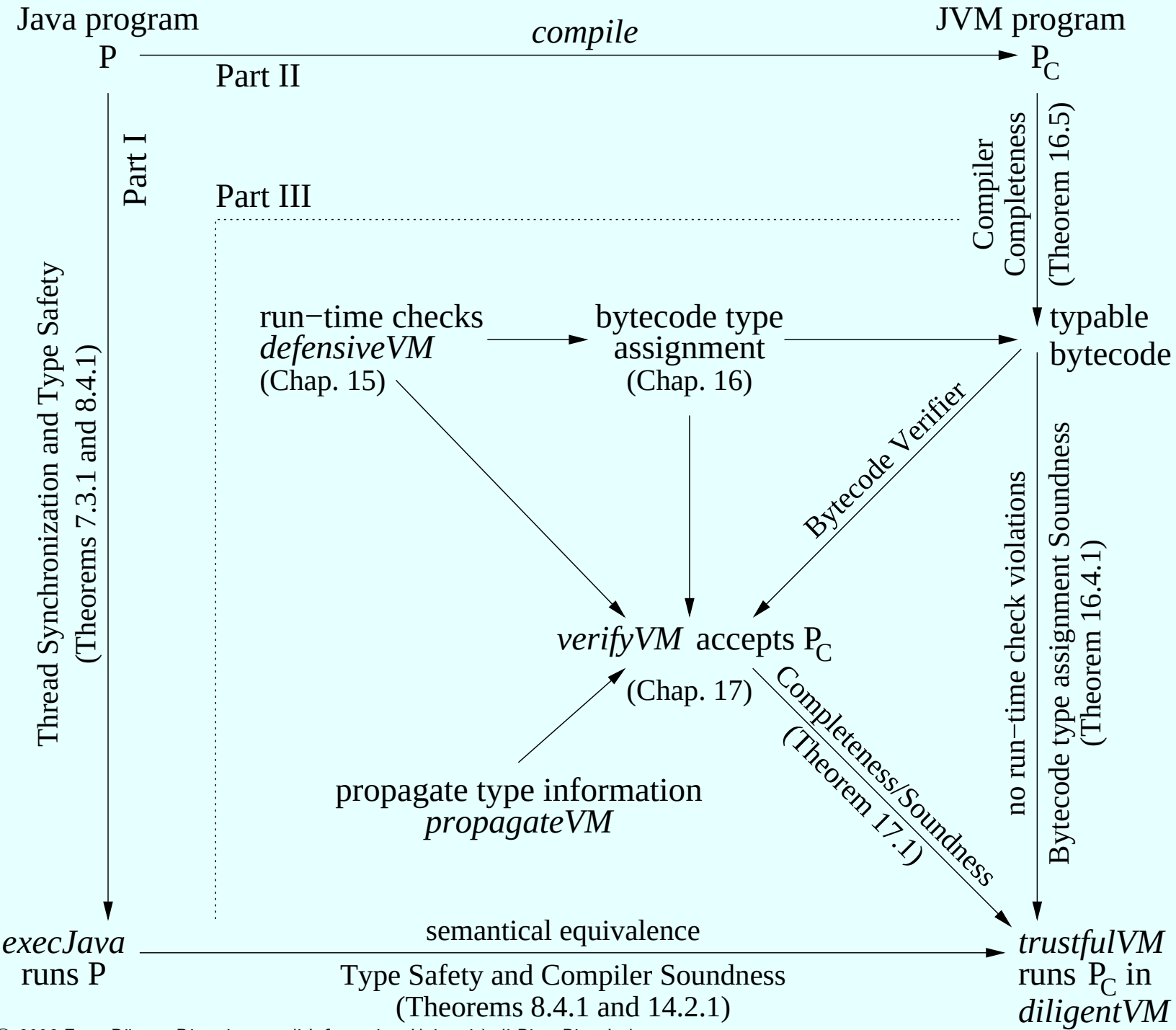




Case Study: ASM Modeling, Validation, Verification

R. Stärk, J. Schmid, E.Börger: Java and the JVM. Springer 2001.

- horizontal refinement: Java/JVM ground models (reused for C#/CLR)
- vertical refinement: via compilation and decomposition of JVM into trustful, defensive, diligent (separating bytecode type assignment from bytecode verification), dynamic (loading) machine



Mechanical Verification Technology Transfer Challenge

Starting from the structured and high-level ASM definition of Java and of its implementation on the Java Virtual Machine

Verify: Theorem. Under explicitly stated conditions, any well-formed and well-typed Java program:

- upon correct compilation
- passes the verifier
- is executed on the JVM
  - without violating any run-time checks
  - correctly wrt Java source pgm semantics

in a way that can be applied by language developers, e.g. reused for language extensions: C#, ...

Hoare's Logic-Based Verified Sw Grand Challenge Vision

- "Construct a program verifier that would use *logical proof* to give an *automatic check* of the correctness of programs submitted to it"
- "program verifiers...transform a program and its specification into verification conditions that can be discharged by the logical tools"
- "The criterion of correctness is specified by types, assertions, and other redundant annotations associated with the code of the program."

But programming is more than producing annotated program texts!

- T. Hoare: *The Verifying Compiler*: A Grand Challenge for Computing Research. J. ACM 50.1 (2003) 63-69
- T. Hoare and J. Misra: *Verified software*: theories, tools, experiments. Vision of a Grand Challenge project (manuscript, July 2005)
- Proc. IFIP Working Conference on the *Program Verifier* Challenge, ETH Zürich, October 2005. Springer LNCS (in print)

Context where compilable code verification makes sense

Compilable programs are

- *software representations* of computer-based systems, written *for mechanical elaboration* by machines, not by humans
- result of two development activities exercised mainly by humans and not captured by a verifying compiler
  - turning the requirements into *ground models*, defining application centric system meaning abstractly and precisely, prior to coding
    - calls for validation, by pragmatic reasoning and experiments, of the application-domain-based correctness (semantical foundation)
  - linking ground models to compilable code by a series of *refinements*, which introduce step by step the details *resulting from the design decisions* for the implementation
    - calls for mathematical (not limited to logical) verification of refinement correctness (foundation of design decisions)

3 Epistemological Ground Model Attributes

- precise at appropriate level of detailing, excluding undesired ambiguities: rigorous, but not more than needed (*informal accuracy*)
  - semantically well founded (basis for verification and validation)
  - overcoming infinite regress via
    - *inspection* by domain experts establishing completeness & correctness wrt empirical interpretation in intended requirements
    - *domain-specific reasoning* checking properties (e.g. consistency)
- minimal (abstract): abstracting from design details
- simple and concise to be understandable & acceptable as contract by domain experts/system designers
  - using abstractions that "directly" reflect the structure of the represented real-world phenomenon without extraneous encoding
  - solving communication pbl: mediate bw software designers & domain experts or real-world appl. domain & world of models

NB. These properties can be obtained with ASM models.

ASMs for Foundation of Ground Models & Design Decisions

- Compilable programs, though often considered as the true definition of the system they represent, seldom *ground the design in reality*
  - needs investigation of correspondence between extra-logical theoretical terms and their empirical interpretation (Carnap)
- ASM ground models are math. application-centric system blueprints
  - realizing justifiably correct transition from nat-lg descriptions to formulations of mathematical nature that
    - represent the algorithmic content (what Brooks calls "conceptual construct" or "essence") of the software contract, based upon a clear empirical interpretation of the used concepts
- ASM refinements provide a math. link from ground model construction to verification of compilable code by verifying compiler

This calls for lifting Hoare's verified sw challenge from program verification to a discipline of verifiable system development

Challenges for Verified Software Project

- *refinement generator challenge*: define practical model refinement schemes (refinement patterns), which capture established programming knowledge, together with justifications of their correctness—to turn model properties into software interface assertions comprising behavioral component aspects
  - to be used where run-time features are crucial for a satisfactory semantically founded correctness notion for code
- *refinement verifier challenge*: enhance current logical or computer-based verification systems by means to prove the correctness of ASM refinement steps, enhancing work done with KIV/PVS and exploiting link from Event-B to a class of ASMs
- *refinement validator challenge*: link ground model refinements to tools for generation and comparison of corresponding test runs of abstract and refined machines (e.g. relating system and unit level test)

Challenges for Verified Software Project (Cont'd)

- *ground model pattern* challenge: collect patterns of frequently occurring model schemes, raising the level of abstraction of popular programming and design patterns
- *runtime verifier challenge*: instrument high-level model execution tools (e.g. interpreters for ASMs or event-B models or model checkers for TLA+ models) to monitor the truth of selected properties at runtime (as done in AsmL), enabling in particular the exploration of ground models to detect undesired or hidden effects or missing behavior
- *re-engineering method challenge*: define methods to extract ground models from legacy code as basis for analysis or re-implementation
- *system certification milestone*: integrate ground model validation and analysis into industrial system certification processes, e.g. formulating the technical content of software reliability for embedded systems
- *verified compiler challenge*: verify the verifying compiler itself
  – extending the Verifix and related work with Coq etc.

References on ASM Method

- E. Börger The *ASM Ground Model Method* as a foundation of requirements engineering. LNCS 2772 (2003) 145-160
- E. Börger The *ASM Refinement Method*. Formal Aspects of Computing 15 (2003) 237-257
- E. Börger and R. F. Stärk: Abstract State Machines. Springer 2003. pp.X+438. http://www.di.unipi.it/AsmBook/
- E. Börger: Linking Content Definition and Analysis to What the Compiler Can Verify.
  Proc. IFIP Working Conference on the *Program Verifier* Challenge, ETH Zürich, October 2005. Springer LNCS (to appear)