Egon Börger (Pisa) & Alexander Raschke (Ulm)

The Role of Modeling

for Design and Analysis of Software-Based Systems

Università di Pisa, Dipartimento di Informatica, boerger@di.unipi.it
Universität Ulm, Abteilung Informatik alexander.raschke@uni-ulm.de

See Ch.1 of Modeling Companion
http://modelingbook.informatik.uni-ulm.de

The two ends of software-based system development

A *reliable* method for the development of software-based systems has to

bridge the gap

- bw human understanding and formulation of real-world problems
  - which involves application domain experts and system users
- and deployment of their solutions by code-executing machines
  - which involves experts in computing (design engineers, programmers)

The gap derives from the fact that

- software and code executing computer are only part of the system
- the other parts constitute the environment on which software and computer depend and which they affect
  - technical equipment, sensors, actors, information systems, users, etc.

The gap is bw requirements and code

- Requirement documents are *descriptions of real-world phenomena*
  - typically written by domain experts *for system design experts* (usually not knowledgeable in the application domain)
  - in natural language, interspersed with diagrams, tables, formulae, etc.
  - possibly ambiguous, incomplete, inconsistent
- Compilable programs are *software representations* of real-word items and actions, written *for mechanical elaboration* by machines (symbol manipulation) and therefore coming with every needed implementation detail (technical precision, completeness, consistency).
- How can (informal) requirements and (formal) code, the latter written to satisfy the former, be linked in a way to controllably

  guarantee that the code does what the requirements describe?
- How can the link between requirements and code be reliably *preserved during maintenance* (when requirements may change)?

What is needed to 'bridge the gap'

- accurately formulate the desired software behavior from the real-world *system perspective*
  - adequately relating the relevant real-world system elements and their behavior to abstract concepts (requirements capture)
    - to 'ground' the abstract model in the reality it represents (*ground model correctness* problem)
- provide the guarantee (*implementation correctness problem*) that the ground model correctness is preserved through the *design steps* taken
  - to modify the high-level requirements model (e. g. for exploration purposes or to correct misunderstandings)
  - to transform it to machine executable code

The two concerns are of different nature (see more below):

- Ground Model Concern: application-domain focussed (epistemological)
- Refinement Concern: machine-focussed (mathematical)

The three Ground Model constituents

To correctly understand and formulate the real-word problem in the context where the software will be executed, involves three descriptions:

- a precise, complete, consistent description of the requirements
  - providing a description of the desired system behavior at the level of abstraction and rigor of the application domain
  - at an application-problem-determined level of detailing (*minimality*)
- a software specification: precise abstract description of the behavior that is expected for the sw when it is executed in the system env
- a description of those domain assumptions the system designers can rely upon to justify that the spec satisfies the requirements
  - this justification is about a behavioral *correctness property*:
    - that under the domain assumptions, the spec will behave as system component the way the requirements demand

The epistemological role of Ground Models

Goal: to reach a common understanding by humans—experts of different fields—of some to-be-implemented behavior in the real-world

- requirements and domain assumptions
  - are under responsibility of application domain experts
  - must be made explicit for the sw spec
- sw spec stands bw requs and sw: a 'blueprint' ('contract' for coding)
  - must be formulated in a language all parties involved understand, prior to coding (*communication problem*)
    - cannot be a programming or formal logic language

      'Nearly all the serious accidents in which sw has been involved in the past 20 years can be traced to reqs flaws, not coding errors.'(Leveson 2012)
  - must support any form of checkable explanations to justify the 'correctness' of the spec (*verification/validation method problem*)
    - cannot be restricted to machine-verified proofs

Characteristic properties of Ground Model specs

Using a sw spec as 'blueprint' ('golden model'), to evaluate/debug before accepting it as 'contract for coding' sw experts can rely upon

- by reasoning about its properties (verification)
- by observing its behavior through experiments (validation by testing)

requires four basic properties of ground model specs:

- *precise at the level of abstraction and rigor of the problem* and of the application domain it belongs to ('informal accuracy')
  - i.e. unambiguous (in particular in its domain knowledge related features) to be correctly understandable by the sw designer
- *correct*: model elements adequately convey the meaning of what they stand for in the real world and reliably convey original intentions
- *complete*: contain every behaviorally relevant feature
- *consistent*: conflicting objectives (which may be present in the original requirements) are resolved

Epistemological character of ground model precision

To be understandable for experts of different fields—solving the communication problem—the ground model lg must permit to

- calibrate the degree of precision (the abstraction level) of descriptions to any given application-domain problem, accurately expressing
  - *any kind of objects* in the real-world with their properties/relations
    - items which constitute arbitrary system 'states'
  - *any actions* to change the set of objects or to modify some property or relation among them
    - actions which constitute arbitrary 'state changes'

i.e. embrace a most general notion of state and state change so that domain experts and software experts can reach a common understanding of what system states and state changes are

> 'The extra communication step between the engineer [read: the domain expert] and the software developer is the source of the most serious problems with software today.'(Leveson 2012)

Epistemological basis of ground model justification

- Ground models are *conceptual* models which relate *real-world* features to linguistic elements.
- Appropriateness of the association of real-world objects/relations with model elements cannot be proved by mathematical means.

  Leibniz: proportio quaedam inter characteres et res ... est fundamentum veritatis

Model inspection can help: reviewing of the blueprint (not of code!)

- performed in cooperation by application-domain experts and sw experts
- providing *evidence* that the *direct correspondence* between calibrated model and real-world elements is the desired one (i.e. adequate)
  - in particular establishing correctness and completeness

NB. Issue is not 'declarative/operational', but calibration of precision

Ground model justification needs verification and validation

Ground model inspection

- resembles traditional code inspection but
  - happens at a higher level of abstraction
  - involves, besides sw designers and programmers, application domain
  - helps to detect conceptual (not only programming) mistakes
- involves two complementary analysis techniques:
  - *verification*: using mathematical reasoning, based upon requirements assumptions, to establish desired run properties
  - *validation* by repeatable experiments (simulation, testing, running scenarios), aimed at confirming/falsifying predicted model behavior

Both techniques require that specs are executable, conceptually and mechanically (supported by machines)

- contrary to widely held view on purely declarative, not executable specs
- supporting test oracles and exploratory design development

Refinement Concern: Management of Design Decisions

Refinement: a general methodological principle to manage complexity

- piecemeal de-/composing a system into/from its constituent parts
- which can be treated separately and (re-) combined
  - the way mathematicians prove theorems from lemmas

Here this is applied to manage system design decisions:

- formulate, check and document each design decision taken in linking, through various levels of abstraction, the system architect's view (blueprint level) to the programmer's view (compilable code level)
  - *split complex behavior/properties* into a set (often a series) of simpler actions/properties, proving the correctness of the decomposition

A practical method should allow the designer to

accurately link data and operations at whatever levels of abstraction

to link the system lifecycle activities, from requirements capture to system maintenance, in an organic, effectively maintainable way.

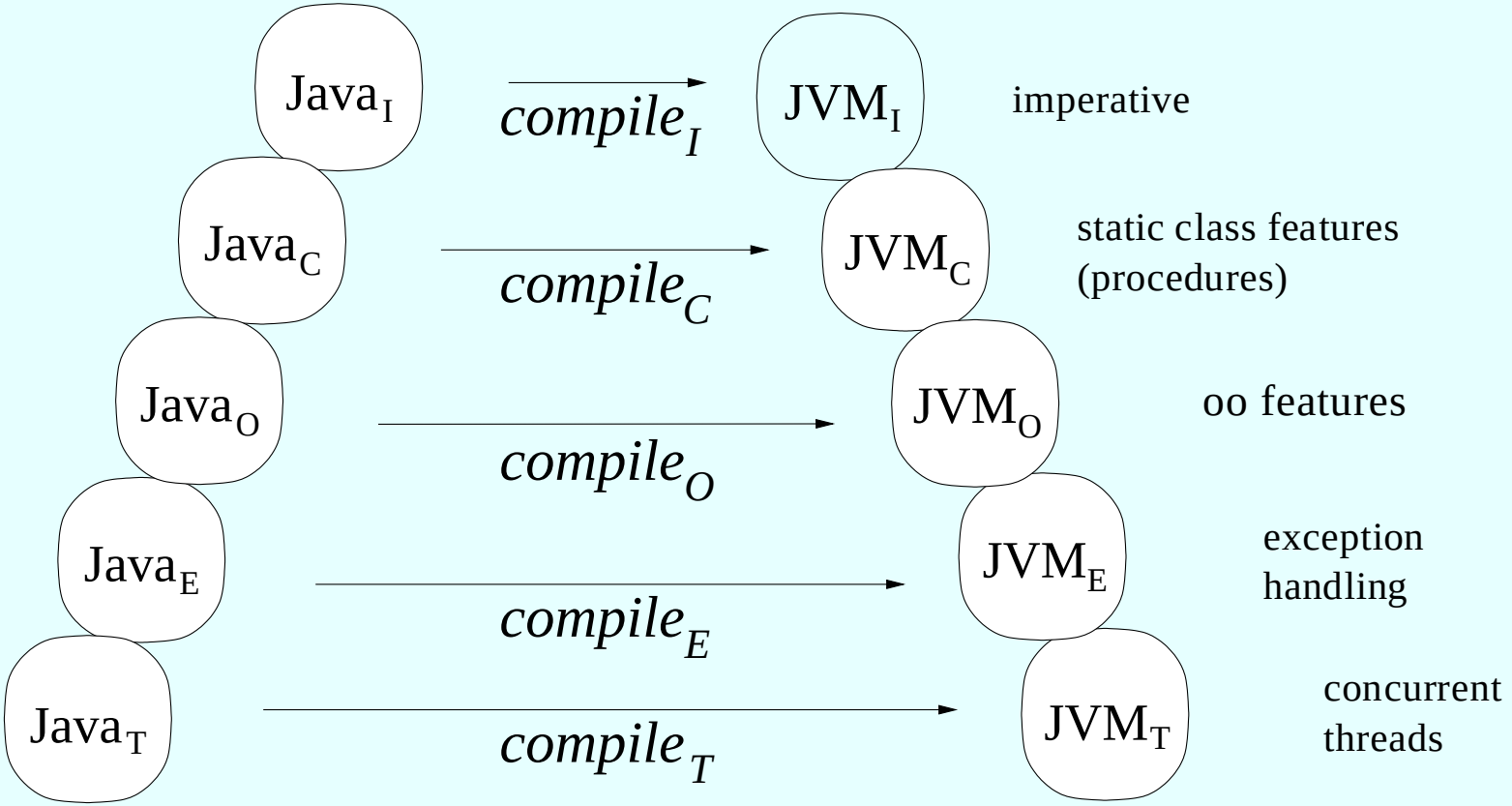
Using refinement for separation of concerns

Refinement supports the separation of concerns:

- horizontal refinements permit to accurately introduce piecemeal extensions and adaptations to changing requs or envs
  - supporting *design for change* and system *maintenance*
- vertical refinements permit to stepwise introduce more and more details implementing model elements (domains, functions, rules)
  - supporting *design for reuse* and development of design patterns
- If the models are of mathematical nature, besides modeling one can *prove refinement correctness* (from appropriate assumptions)
  - to be useful for design and development practice, proofs should be possible at every desired degree of detail:
    - proof sketches
    - detailed proofs, for humans to read
    - machine checkable/generated proofs

    each kind of verification coming with its merits and cost

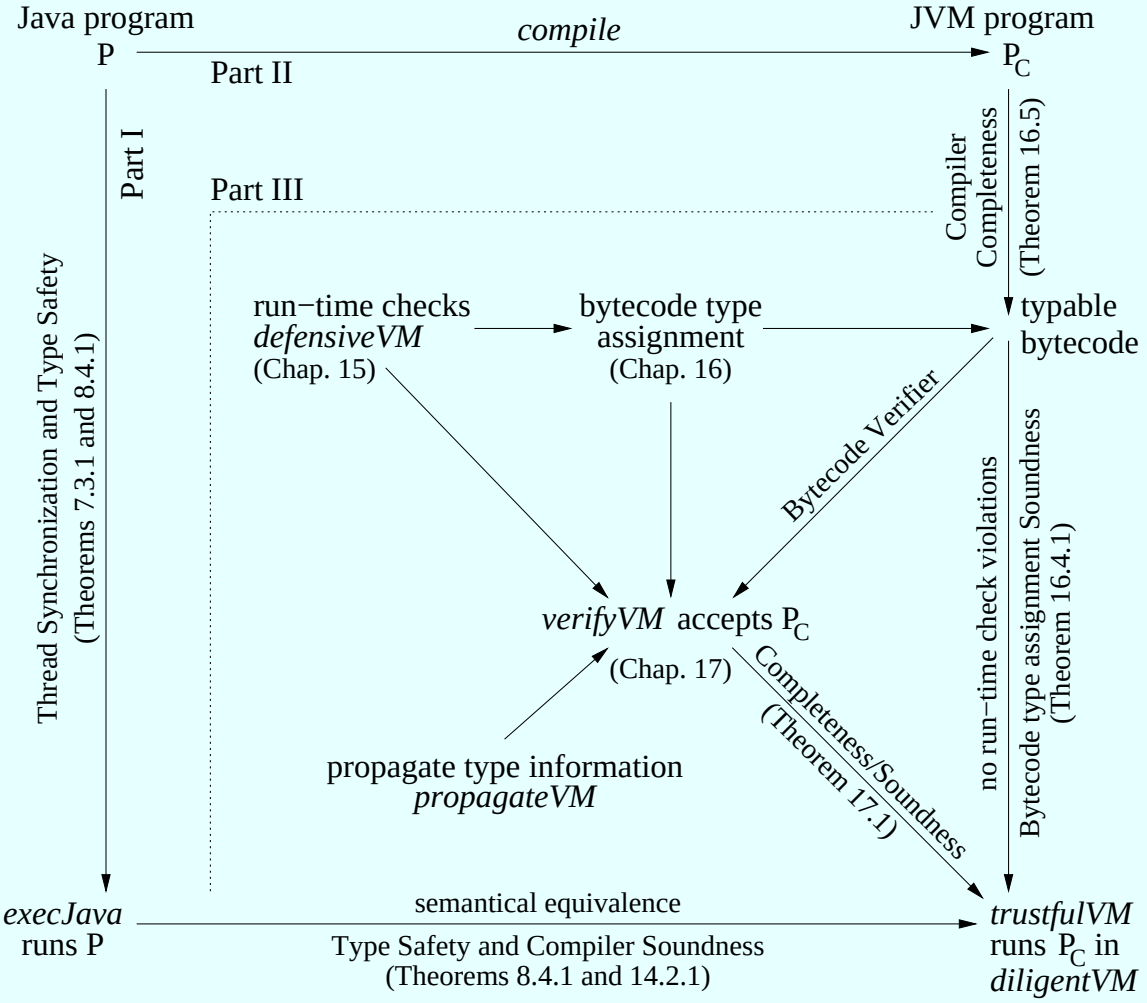
Horizontal refinement examples for Java/JVM models



1

[1] This and the next figure are from JBook, © 2001 Springer-Verlag Berlin Heidelberg, reused with permission.

Vertical refinement examples for Java/JVM models



Why ASMs are appropriate as ground models

Abstract State Machines (ASMs) are finite sets of rules of form

if $condition$ **then** $action$

- ASM language satisfies the fundamental ground model lg properties:
  - *general direct expressivity*: any rigorously defined $condition$ and $action$ involving any objects/properties are allowed
  - *easy comprehension*: rules follow a common scheme to describe an action to be taken when some condition is satisfied
  - *unambiguous definition*: state transforming effect of such rules has a simple precise definition which yields a rigorous notion of run
- ASMs are *executable* (conceptually and machine-supported) and thus can be validated by experiments
- ASMs are *mathematical objects* and thus can be analyzed by mathematical methods

Variety of real-life ASM ground models (1)

- **industrial standards**: ground models for the standards of
  - OMG for BPMN (1.0/2.0): Börger,Thalheim,Sörensen 2007-11
  - OASIS for BPEL: Farahbod et al. ASM'04 and IJBPMI 1 (2006)
  - ECMA for C#: Börger, Fruja, Gervasi, Stärk: TCS 336 (2006)
  - ITU-T for SDL-2000: Glässer, Prinz et al. 1998–2003
  - IEEE for VHDL93: Müller, Glässer, Börger:1994–1995
  - ISO for Prolog: Börger, Rosenzweig: 1991–1995
- **design, reengineering, testing of industrial systems**:
  - railway & mobile telephony network component sw (at Siemens)
  - fire detection system sw (in German coal mines)
  - implementation of behavioral interface specifications on the .NET platform and conformence test of COM components (at Microsoft)
  - business systems interacting with intelligent devices (at SAP)
  - compiler testing and test case generation tools

Variety of ASM ground models and their refinements (2)

- **programmming languages**: definition/analysis of semantics & implementation for major real-life lgs, e.g.
  - SystemC
  - Java/JVM (including bytecode verifier) see *JBook*
  - C#
  - domain-specific languages used at UBS

  including machine verification of compilation schemes (Prolog2WAM using KIV) & compiler back-ends (DFG projects using PVS)
- 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 workflows, business processes, web services (SAP, Metasonic)

Why ASMs support practical stepwise system development

ASMs come with a general refinement concept which directly follows the arguably most general, abstract concept of state and state transforming actions such that

- practicing domain experts & system designers can use ASM refinements to successively
  - detail (stepwise implement)
  - in a controllably correct (i.e. verifiable) manner

  model abstractions down to executable code

As a consequence, the ASM method

- is NOT a special-purpose but a wide-spectrum method
  - assisting system engineers in every aspect and at any level of abstraction of an effectively controllable construction of reliable computer-based systems

Characteristics of the ASM Method

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* (change management) 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*

ASM Analysis Techniques (Validation and Verification)

Practitioner supported to analyze ASM models by reasoning and experimentation at the appropriate degree of detail, separating

- orthogonal design decisions and complementary methods: abstract operational vs declarative/functional/axiomatic, state- vs event-based
- design from analysis (definition from proof)
- validation (by simulation) from verification (by reasoning)
  - e.g. ASM Workbench (ML-based, DelCastillo 2000), AsmGofer (Gofer-based, Schmid 1999), XASM (C-based, Anlauff 2001), AsmL (.NET-based, MSR 2001), CoreASM (since 2005), Asmeta
- verification levels (degrees of detail)
  - reasoning for human inspection (design justification)
  - rule based reasoning systems (e.g. Stärk's Logic for ASMs)
  - interactive proof systems, e.g. KIV, PVS, Isabelle, AsmPTP
  - automatic tools: model checkers, automatic theorem provers

6 fundamental questions for building ground models

The ASM method leads to ask the following 6 questions when building a ground model as 'model of the system's intended behaviour'

- called *golden model* in the International Technology Roadmap for Semiconductors (2005)

1. Who are the system agents and what are their relations? WIn particular, wat is the relation between the *system* and its *environment*?

2. What are the system states?

- What are the domains of objects and what are the functions, predicates and relations defined on them? (object-oriented approach to system design)
- What are the *static* and the *dynamic* parts (including input/output) of states?

6 fundamental questions for building ground models (Cont'd)

3. How and by which transitions do system states evolve?

- Under which conditions (*guards*) do the state transitions (*actions*) of single agents happen and what is their effect on the state?
- What is supposed to happen if those conditions are not satisfied? Which forms of *erroneous use* are to be foreseen and which *exception handling* mechanisms should be installed to catch them? What are the desired *robustness* features?
- How are the transitions of different agents related? How are the *internal* actions of agents related to *external* actions of the environment?

6 fundamental questions for building ground models (Cont'd)

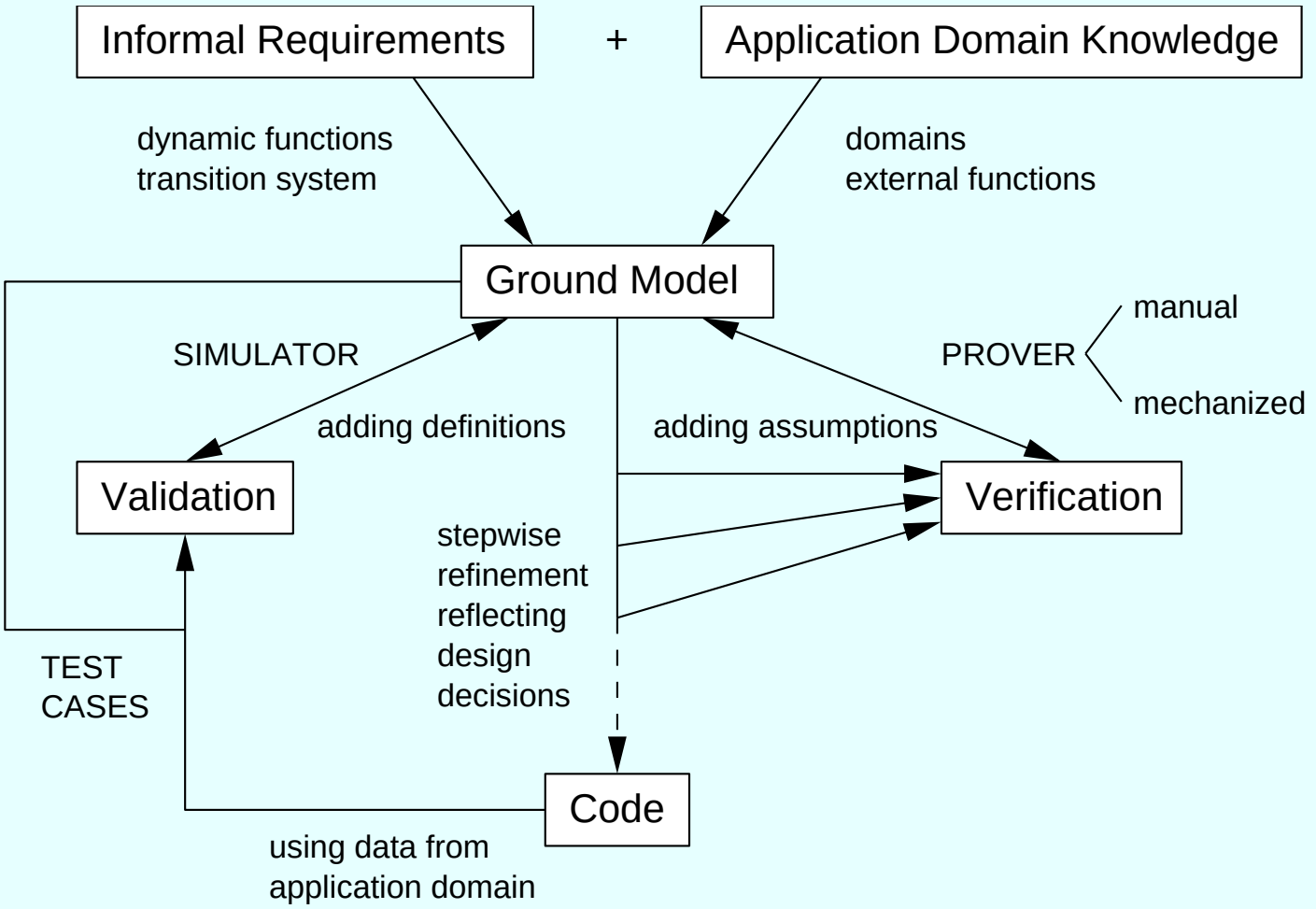
4. What is the initialization of the system and who provides it? Are there termination conditions and, if yes, how are they determined? What is the relation between initialization/termination and input/output?

5. Is the system description complete and consistent?

6. What are the system assumptions and what are the desired system properties?

- At the level of transitions this question can be formulated and dealt with in terms of pre-/postconditions (assume/guarantee scheme)

Models and methods in an ASM-based development process



2

[2] (Figure from AsmBook, © 2003 Springer-Verlag Berlin Heidelberg, reused with permission

References

E. Börger: Construction and Analysis of Ground Models and their Refinements as a Foundation for Validating Computer Based Systems.

- Formal Aspects of Computing J. 19 (2007), 225-241

E. Börger: *The ASM* Refinement *Method*

- Formal Aspects of Computing 15 (2003), 237-257

Refinement papers by G. Schellhorn in J.UCS 2001, 2008, TCS 2005, ENTCS 2008, LNCS 5238

E. Börger and A. Raschke: Modeling Companion for Software Practitioners. Springer 2018
http://modelingbook.informatik.uni-ulm.de

E. Börger and R. F. Stärk: Abstract State Machines Springer 2003. pp.X+438. See http://www.di.unipi.it/AsmBook/

R. Stärk, J. Schmid, E. Börger: Java and the Java Virtual Machine. Definition, Verification, Validation. Springer 2001.

Copyright Notice