Egon Börger (Pisa)

Why Programming Must be Supported by Modeling

Slides prepared for a talk at the ISoLA18 conference track

Towards a Unified View of Modeling and Programming

See paper *Why Programming Must be Supported by Modeling and How* in: T. Margaria and B. Steffen (Eds): ISoLA 2018. Springer LNCS 11244, pg. 1-22.

https://doi.org/10.1007/978-3-030-03418-4_6

boerger@di.unipi.it

The problem: two ends of sw-based-system development

A *reliable* method for the development of a *software-based-system* must

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)

NB. In a software-based-system

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

Where the pbl becomes tangible: requirements capture

- 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 do change)?

How the problem can be solved

Step 1. Turn the requirements into a ground model.

Ground models are 'blueprints', descriptions which must provide a common, correct, objectively checkable understanding of the intended system behavior by all parties involved.

This requires the descriptions to be:

- *precise*, formulated in accurate application-domain terms (not code)
- *complete* and minimal
  - containing each element which is relevant for the behavior of the intended system, avoiding what is needed only for its implementation
- *correct* wrt the intensions for the system
- *consistent* (an internal model property)

Step 2. Transform the ground model in a *correctness preserving* way into code.

Step 1. Characteristics of ground models

A common, correct and checkable understanding of system behavior by

- application domain experts, who provide reqs for desired behavior
- sw engineers, who provide implemented behavior

requires 3 properties for 'blueprints': to be

- formulated in a common language all parties involved understand (*communication problem*)

  '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)
- objectively checkable for its correctness (*evidence problem*)
  - to provide evidence that model elements adequately convey the meaning of what they stand for in the real world and reliably express the intended real-life system behavior
- executable, conceptually or by machines, to be experimentally falsifiable in the Popperian sense (*validation problem*)

(1) Communication problem: calls for precise sublg of nat lg

Understandability by all stakeholders requires basic ground model lg to permit to calibrate the degree of precision (the abstraction level) of descriptions to any given application-domain problem:

- express intended system behavior without encoding, *directly* in terms of
  - *any kind of objects* in the real-world with their properties/relations
    - items which constitute arbitrary system 'states'
  - *any state changing actions*, concurrently performed by any agents
- using rigorous application domain concepts

i.e. embrace a most general, accurate notion of state and state change:

$$\textbf{in } situation \textbf{ do } action$$

> '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)

(2) Evidence problem: of epistemological character

- Ground models are *conceptual* models which relate *real-world* features directly 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 (evidence problem)

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

- performed in cooperation by application-domain experts and sw experts
- supported by experiments with model executions
- providing *evidence* that the *direct correspondence* between calibrated model and real-world elements is the desired one (i.e. adequate)
  - thus establishing 'correctness' wrt intentions

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

(3) Validation problem: requires a run concept for models

Ground model justification by inspection

- resembles traditional code inspection but
  - happens at a higher level of abstraction
  - involves both sw and application domain experts
  - 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 for the model
  - *validation* by repeatable experiments (simulation, testing, running scenarios), aimed at confirming/falsifying predicted model behavior

Both techniques require that models are executable, can be run conceptually and/or mechanically (supported by machines)

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

Step 2. Transforming ground models correctly to code

Appropriate combination of three basic approaches:

- direct programming, typically of once-only applications
  - using ground model as spec
  - correctness by code inspection against the ground model
  - easened if pgg lg (e.g. DSL) offers modeling concepts
- compilation, typically where requirements keep changing
  - write a compiler for a class of to-be-expected ground models
  - compiler correctnes implies code correctness from ground model correctness
    - established once, requiring verification expertise
- stepwise refinements of models, to manage complexity
  - piecemeal de-/composing models into/from simpler constituent parts
  - which can be treated separately and (re-) combined

Needed: a practical use of refinement methods

- to manage (formulate, justify, document) system design decisions
  - turning abstract behavior/properties into (a set, often a series of) more detailed (implementable) actions/properties
  - proving/testing the correctness of the (de-)composition

  thus linking, typically through various levels of abstraction, the system architect's view (blueprint level) to the programmer's view (compilable code level)
- to support 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

Why ASMs are appropriate as ground models

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

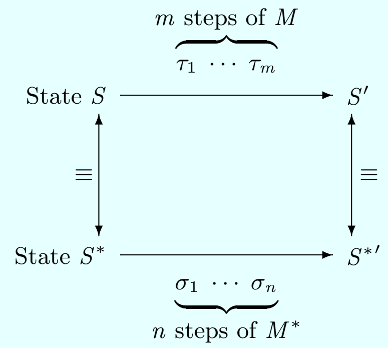
$$\textbf{if } condition \textbf{ then } action$$

- ASM language satisfies the fundamental ground model lg properties:
  - *direct & general expressivity*: any rigorously defined $condition$ and $action$ involving any objects/properties are allowed
    - comprising functional, axiomatic, operational means of desription
  - *general intelligibility*: rules follow a common intuitive scheme to describe an action to be taken when some condition is satisfied
  - *unambiguous definition*: state transforming effect of such rules has a precise yet simple definition, supporting an intuitive 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

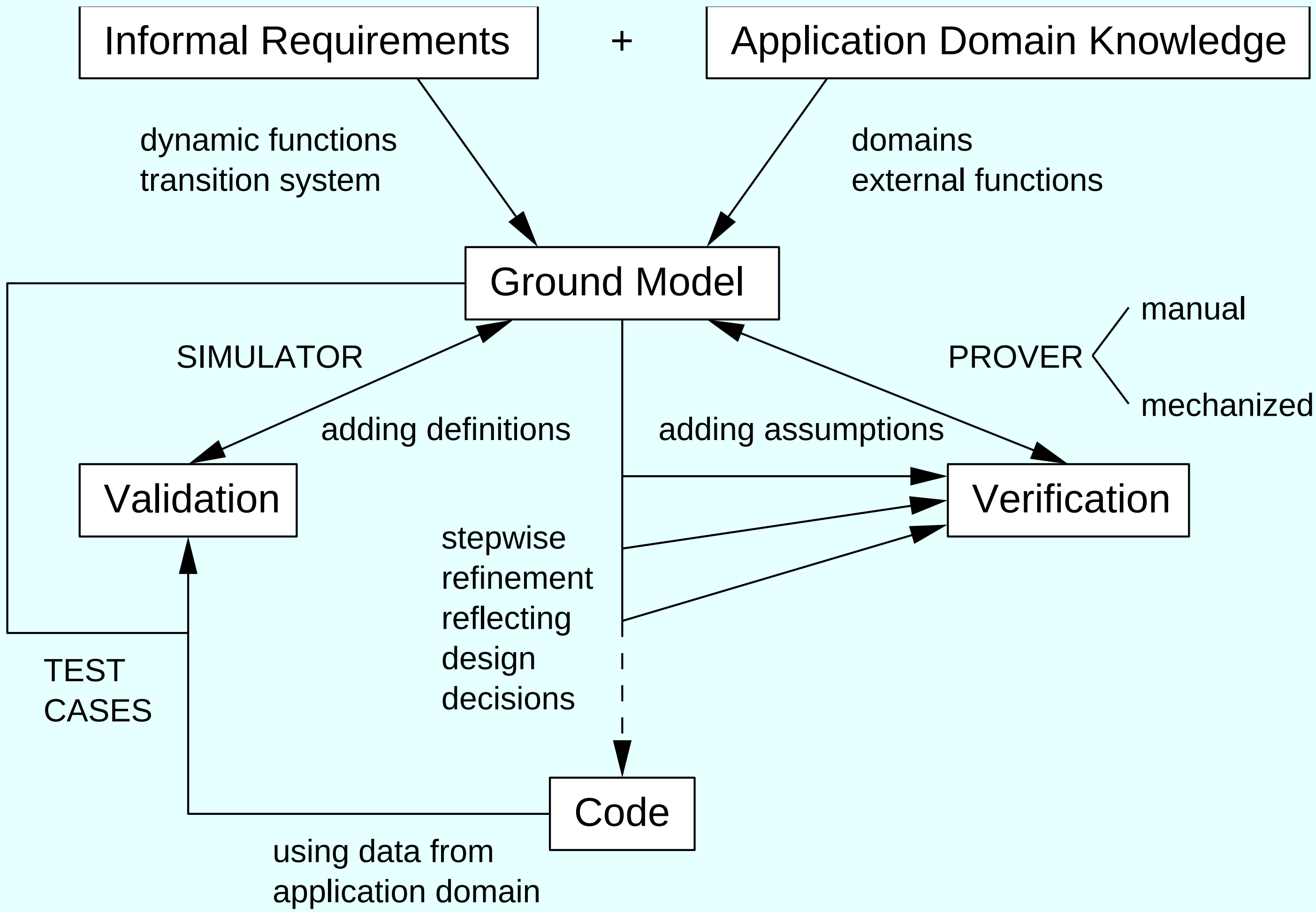
Why ASMs support practical stepwise system development

ASM refinement concept offers freedom to choose notions of:

- *abstract/refined state*
- correspondence bw pairs (S, S^*) of abstract/refined states of interest
- abstract/refined computation segments of m/n single abstract/refined steps τ_i/σ_j leading from/to corresponding states of interest
- *locations of interest* and *corresponding* abstract/refined locs of interest
- equivalence of values in corresponding locations of interest



Models and methods in an ASM-based development process



1

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

Reference

E. Börger:

Why Programming Must be Supported by Modeling and How.

in: T. Margaria and B. Steffen (Eds): ISoLA 2018, pg. 122, Springer LNCS 11244.
https://doi.org/10.1007/978-3-030-03418-4_6

Further 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*. ibid. 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