

Theory Panel Report

Egon Boerger, Xavier Leroy, Markus Mueller-Olm, Peter O'Hearn, John Reynolds, Peter Sestoft, Steve Zdancewic

Dave Naumann Stevens Institute of Tech., USA

Goals of panel

- ◆ create a road map challenges/milestones in theory of program construction and analysis that are critical for the overall goals of VS
- ◆ interact with the other VS panels to refine these challenges as increasingly specific goals and activities emerge
- ◆ coordinate activities of researchers who seek to meet those challenges
- ◆ publicize the effort to broaden participation and support

Role of theory

- ◆ semantics underlies modeling languages, design methods, code verification methods incl. proofs systems
- ◆ semantics is basis for specifications of tools
- ◆ integral to development of analysis algorithms, e.g., CF-reachability, predicate abstraction, linear types, type inference, combining decision procedures...
- ◆ specification: composable properties; notions of simu.; decidable fragments; enforceable by ref. monitor...
- ◆ unification and linking of theories: improved methods, integrated tools

Theory in the Roadmap

Theory pervades tool development and experiment challenges.

Panel focus on what's of interest to theorists (and ourselves).

Panel formulating challenges and milestones to be integrated in roadmap; organized in accord with its research directions —the rest of this talk.

Modeling of requirements: areas

- ◆ decomposition and modularization of ground models
- ◆ specifying intensional properties (e.g., time and space complexity of lazy functional; resource parameters for embedded components; covert channels)
- ◆ specifying security properties, e.g., info flow policy with declassification, policy revision, distributed negotiation
- ◆ different models of computation in hybrid systems (discrete event, continuous time, sync dataflow, etc)

Modeling of requirements: challenges

- ◆ (near term) specs for a range of interesting security properties
- ◆ (near) instrument high level model execution tools (e.g., ASM or event-B interpreter, TLA+ model checker) to monitor properties, for exploration of ground models
- ◆ (long term) compositional modeling tools and notations with good formal semantics, tailored for various sectors, used in common practice

Design methods

- ◆ refinement of abstract models down to designs and code (cf. CxC panel) (Boerger)
- ◆ program generation and staging: improving ad hoc generators (e.g., for J2EE deployment descriptors and interfaces, web pages, Javascripts, SQL, glue code) (Sestoft)
- ◆ explicit manipulation of dynamic resources, design of separation and encapsulation patterns (O'Hearn, Naumann)

Design methods: challenges

- ◆ (near) refinement generator: practical model refinement schemes that capture established development and programming knowledge together with justifications
- ◆ (near) refinement verifier for these schemes
- ◆ (near) refinement validator: link ground-model simulators with design simulators and unit testers
- ◆ (near) type theories for checking multistage notations
- ◆ (long) design and reasoning principles for components (large, w/complex interactions) and composition (e.g, web services) using only interface specs

Milestones for multistage

Check gen'd code (a) syntactic well-formed; (b) scope well-formedness (binders, cross-references); (c) type correctness; (d) dynamic semantics correctness.

- (near) Theory of types for multistage languages that covers correctness levels (a), (b), (c) and (d). Support type notations and mechanized consistency checking.

Dependent types (Martin-Lof) and nominal logic (Pitts).

- (near) Theory of types to support safe runtime reflection.
- (near) Theory to support typesafe database access from a mainstream programming language.

Verification methods

- ◆ proof techniques for refinements, incl. last step to code
- ◆ model checking and abstract interpretation
- ◆ type and effect systems (resources, calling protocols,...)
- ◆ logics and semantics (code pointers; frame conditions; inheritance; dynamic thread forking)
- ◆ concurrency; weak memory models, wait-free algos and transactional memory; grainless semantics
- ◆ intensional properties (performance of SAT solver, cache-aware data rep'n.)

Verification methods: challenges

- ◆ (near) modular reasoning about frame conditions (separation logic, JML, Boogie)
- ◆ (near) verification of wait-free objects
- ◆ (near) add property checking to simulators (ASM, event-B, TVLA+)
- ◆ (long) composable concurrency abstractions
- ◆ (long) complete verification of a realistic runtime system, e.g., a realtime runtime for Java, including intensional properties (code won't overflow stack, code is fast enough and will meet its deadlines)

Integrated verification environments

- ◆ interoperable tools
- ◆ linking theories as foundation (Hoare, Woodcock)
- ◆ problem reductions (e.g., simulation reduced to safety)
- ◆ verification of tools: translation validation (Pnueli);
compiler verification; machine-checked meta-theory;
justify integration of disparate tools, to build new tools
and in ad hoc scripting for particular verifications

Integrated verification: challenges

- (near) linking theory for embedding ACL2 in HOL
- (near) POPLmark challenge (Zdancewic); binders and alpha-conversion; logical relations
- (near) certifying compiler for information flow types (Mobius)
- (near) verify an analysis tool (e.g., SAT solver, Java type checker), from high level requirements in terms of program logic down to the code of a production implementation
- (near) common semantics for (std subsets of) JML, in tools (ESC/Java, KeY, Krakatoa,...)
- (long) unifying theory of theorem proving serving as a

framework for linking theories connecting different logics and lang/logic subsets (logosphere.org)

- (long) theory of design and reasoning principles to design and verify (large) system components and their compositions (say, a web service model, design, or implementation) using only the interface specifications.

Compiler verification

- (near) List Machine challenge
- (near) verify a simple compiler for a widely used language
- Proving existing compilers to be correct (runtime verification; JBook milestones: machine-checked version of Boerger/Staerk)
- (long) verify an optimizing compiler for a widely used language, or for critical embedded software
- (long) finer properties: from memory safety to observational equivalence to full abstraction (for security)
- Theory of compiler verification that is generic for different

target processors and (less important as changes occur less often) source languages.

- Theory of compiler verification that interfaces directly with specification from which target processor is designed, and with the prog lang model used in verifying source code w.r.t. specs