PVS, SAL, and the ToolBus

John Rushby

Computer Science Laboratory
SRI International
Menlo Park, CA
Overview

• Backends (PVS) and frontends (SAL)

• What’s wrong with that?

• A Tool Bus

• Trust and evidence

• An Evidential Tool Bus
PVS and its Backends

PVS included powerful automation since its first release (1993)

- Shostak's combination of integer and real linear arithmetic with uninterpreted functions
  - Previously in STP (1980) and Ehdm (1988)
  - Conjunctions only

- Propositional calculus at an outer level
  - Using BDDs (1995)

- Rewriting and heuristic quantifier instantiation

- Symbolic model checking for CTL (1995)

- Predicate abstraction
  - Invented and first realized as a PVS extension (1996)

- WS1S, via Mona library (2000)

PVS and its Backends (ctd.)

- Nonlinear arithmetic
  - RAHD (Real Algebra in High Dimensions)
    - By Grant Passmore (2008)
    - Not tightly integrated with other DPs, but loses little

- SMT solvers (Satisfiability Modulo Theories)
  - ICS (2002)
    - The “lazy” integration of decision procedures and a SAT solver (supplanted the earlier “eager” integration)
  - Yices 0.1 and Simplics (2005)
  - Decides the combination of uninterpreted function symbols with equality, linear real and integer arithmetic, scalar types, recursive datatypes, tuples, records, extensional arrays, fixed-size bit-vectors, quantifiers, and lambda expressions
Progress of 2007 and 2008 competitors on real difference logic

John Rushby, SRI
How Best to Exploit the Power of SMT Solvers?

- SMT solvers routinely solve problems with tens of thousands of variables and constraints
  - A disruptive innovation

- You don’t interact with formulas this big

- So an uncomfortable match for interactive provers like PVS
  - Only useful for endgames; elsewhere grind is better

- But SMT solvers also provide satisfying assignments

- Hence a natural application is bounded model checking over “infinite” domains ("infinite BMC," SAL 2002)
  - Extends to verification via $k$-induction (SAL 2003)
The Awesome Power of SMT Solvers

Example: biphase mark

- An asynchronous communications protocol with realtime constraints

**ACL2**: Moore (1994), one of his “10 best ideas”

**PVS**: three different versions
  - One by Groote and Vaandrager used PVS + UPPAAL
  - Required 37 invariants, 4,000 proof steps, hours of prover time to check

**SAL**: Brown and Pike (2006)
  - Compact, readable specification
  - Verification by infinite BMC with $k$-induction
  - Three trivial lemmas
  - And one large systematic one (disjunctive invariant)
  - Under 5 seconds of prover time
SAL and its Frontends

- SAL is a model checking environment
  - **Finite state:** symbolic (BDD), and bounded (SAT)
    - Also used for test case generation
  - **Infinite state:** infinite bounded (SMT)
  - **Hybrid:** hybrid abstraction (RAHD-like abstractor)
- Has its own language; type system is similar to PVS
- Intended as a target for translation from widely used languages—e.g., Simulink/Stateflow
  - Formal semantics for Stateflow and prototype translator, and cool method of test case generation (Hamon 2004)
  - Simulink Design Verifier is a standard Mathworks product: test generation, checking, verification (Hamon 2007)
    - Uses an SMT-like solver (Prover) directly
Rockwell-Collins Simulink/Stateflow Toolchain

Verification - Rockwell Collins Translation Framework

- Simulink → SCADE
- StateFlow → Safe State Machines
- Reactis
- Lustre → NuSMV
- Lustre → Prover
- Lustre → ACL2
- Lustre → PVS
- Lustre → Design Verifier

- SAL Symbolic Model Checker
- SAL Bounded Model Checker
- SAL Infinite Model Checker

© Copyright 2008 Rockwell Collins, Inc.  All rights reserved.
What’s Wrong With All That?

- **Deduction is no longer the hard part of verification**

- **Invariant generation is now the hard problem**
  - Abstract interpretation (over logical lattices), templates (solve $\exists \forall$ problems), predicate/data abstraction, CEGAR and interpolants, van Eijk methods, dynamic analysis

- **Verification is not the only task of interest**
  - Debugging, test case generation, static analysis, abstraction, scheduling, plan generation, controller synthesis, approximate and maxSAT-like problems

- **Users want direct access to components**
So We Need to Link Tools

Desiderata:

- Make it worthwhile for people to open up their tools
  - Extract and package components
  - Make internal computations available
- Low cost of entry, network effect
- Allow tools to work together in ad-hoc combinations

Observation:

- Components return values other than (sub)proof outcomes
- E.g., Unsat cores, sets of predicates, counterexamples, invariants, abstractions
- i.e., heterogeneous functionality
Backends or Bus?

- Heterogeneity argues against theorem prover backends
- Bus is a federation of equals; theorem prover is just another component
Interactions on A Tool Bus

• We need ways for one tool to find another

• We need ways for one tool to invoke another

• We need ways for one tool to provide the right kind of input for another, and to understand its outputs

• This starts to look like Service-Oriented Architecture (SOA)
  ◦ SOAP, WSDL, RDF, etc.

• Ugh! Want a higher-level, more declarative notation
Tool Bus Judgements

Propose that Tool Bus interactions take the form of judgments

- $T \vdash A$

- Tool instance $T$ verifies assertion $A$

- Both the tool and the assertion can include variables

Query: $\vdash $\text{predicate-abstraction}(?, B, \phi)$

Response:

- $SAL\text{-abstractor}(\ldots) \vdash $\text{predicate-abstraction?}(A, B, \phi)$

- Tools operate by implicit invocation

- The responding tool constructs the witness, and returns the result or its handle, along with its own invocation
Arguments in Tool Bus Assertions

- These are formulas, counterexamples, sets of predicates, state machines, etc.

- Do we need a universal Tool Bus Language to specify all of these?

- Or are they opaque to the bus, interpreted only by the relevant tools?

- Propose that semantics are opaque, labeled by a Tool Bus Ontology
  - Two dimensions
    - Logic (e.g., Yices)
    - Representation (e.g., state machine)

- But syntax is registered (as XML, e.g., using RELAX NG)
Tool Bus Ontology

Logics

etc

COQ

PVS

higher-order

first-order + theories

first-order

SMT

propositional

formulas counterexamples state-machines unsat-cores etc.

state-machines in SMT

Representations
Tool Bus Operation

- The tool bus operates like a distributed Datalog framework, chaining forward and backward on queries and responses.

- Similar to SRI AIC’s Open Agent Architecture (OAA)
  - And maybe similar to MyGrid, Linda, TIB, ...?

- Can have hints, preferences etc.

- The bus needs to integrate with version management.

- Tools can be local or remote.

- Tools can run in parallel, in competition.

- Some tools may be simple scripts.
Tool Bus Scripts

- Example
  - If A is a *finite* state machine and P a safety property, then a *model checker* can verify P for A
  - If B is a *conservative abstraction* of B, then verification of B verifies A
  - If A is a state machine, and B is a *predicate abstraction* for A, then B is conservative for A

- How do we know this is *sound*?

- And that we can *trust the computations* performed by the components?
Trustworthy Software

- The World is not interested in software
- It’s interested in (socio-technic) systems
- Formal analysis of software contributes only part of the evidence required for evaluation or certification of trustworthy systems
- Also need hazard analysis and its kin (FTA, FMEA, HAZOP), consideration of the environment, human factors etc.
- Modern treatment uses idea of a Safety or Assurance Case
  - Explicit claims, evidence, argument
- In critical systems, will often run multiple software channels: primary/backup or operational/monitor
- How much confidence do we need in a verified backup or monitor?
Confidence in Verification

- By consideration of aleatory uncertainty, Littlewood (2008) shows that failures of a reliable (tested) channel $A$ and a possibly perfect (verified) channel $B$ are conditionally independent.

- Hence $P(\text{failure}) = pf_A \times pn_B$.

- We have epistemic uncertainty about these parameters, but can estimate:

  $$P(\text{failure}) = C + pf_A \times pn_B$$

  - Where $C$ estimates common mode failures across testing and verification (e.g., misunderstood requirements)
  - $pf_A$ is estimated failure rate of tested channel
  - $pn_B$ is estimated probability of unsound verification

- Modest confidence in verification (e.g., $1 - 10^{-4}$) is adequate.
Sound Deduction

- Most failures in verification are due to incorrect formalization, bugs in language processing (e.g., typechecking errors), translation errors
- Very few (if any) can be traced to unsound deduction
- But a verification will certainly fail if your tools and deductive components lack the power to complete it
- We need ways to guarantee soundness that do not compromise deductive power
- Many options: trusted core, proof generation and verified checker, computational reflection, diverse verifiers
- Our preference is for verified checkers that are rather powerful, driven by compact hints, or certificates
Shankar and Marc Vaucher have verified a modern SAT solver that is executable (modulo lacunae in the PVS evaluator)
An Evidential Tool Bus

- Each tool should deliver evidence for its judgments
  - Could be hints and certificates for a reference kernel
  - Could be reputation ("Proved by PVS")
  - Could be diversity ("using both Yices and Z3")
  - Could be declaration by user
    - "By testing," or "Because I say so"

- A full judgment is $T \vdash E : A$, which is the claim that tool instance $T$ provides evidence $E$ for assertion $A$

- And the tool bus assembles these (on demand)

- Can chain on the evidence component

- To construct evidence for overall analysis for use in an assurance case—hence evidential tool bus

- In fact, an evidential tool bus could be (part of) ideal support environment for assurance cases
Summary

• We’ve built and used powerful tools
• And linked them with backends and frontends
• But what The World wants are components
• Individual components must be tightly integrated (e.g., SMT solvers may do $10^{12}$ internal interactions)
• But separate components can be loosely integrated
• And this should be done as peers on a bus
• Proposed a fairly specific outline for an Evidential Tool Bus
• We have a built prototype
  • Too heavyweight: used OAA, toolbus metalogic
• Now starting the second iteration
• Please join in