(Deeply) Integrating Proving and Programming

Joseph Kiniry
KindSoftware Research Group,
a part of the Systems Research Group
University College Dublin
Dublin, Ireland

Funding under EU FP6 FET project “Mobius”

joint work with Cees-Bart Breunesse, Julien Charles, Radu Grigore, and Sam Owre
Provers and Programmers

- these two communities rarely interact
- most programmers do not prove, and most provers do not program
- but formal specifications are of comparable size and complexity to program (components) these days
- why not exchange the best ideas from each community and build a platform for such?
Overall Goal

- always use the appropriate tool for the job
- tie together appropriate “small” systems
- build platform for specification and composition of formal-centric components
- i.e., this is an experimental platform realizing the vision of the verified software community & Rusby’s “evidential toolbus”
Original Context of Work

- Security of Systems (SoS) group (née LOOP, Logic of Object-Oriented Programs) in Nijmegen’s Loop compiler
  - (full, sound, coalgebraic) semantics of Java and JML in ~50K of PVS
  - generate 10Ks of specifications and a handful of lemmas for a given verification
  - proof states grew to multiple GB in size

ESC/Java2
Current Context

- **Mobius** EU FP6 FET project
  - 16 partners focusing on logic- and type-based verification of (concurrent and single-threaded) JML-annotated Java code at the source and bytecode level using proof-carrying/evidence-carrying code
  
- UCD in charge of all tool development

- Hoare’s Verified Software Grand Challenge
  
- contribute code, tools, proofs, etc.
Our Work on PVS and Motivations

- while at Radboud University Nijmegen (Bart Jacob’s LOOP/SoS group)
- initial work “porting” PVS to an Open Source Common Lisp system
- aforementioned extensions to foundation and interface
- the Loop tool: verification system for JML-annotated Java
PVS, SAL, and Yices

What does this have to do with SRI’s tools?

- We use and extend PVS heavily.
- We next will incorporate SAL and Yices into ESC/Java2 and/or the Mobius PVE.
- We find PVS is an excellent platform for experimentation because natural syntax for programmers and its “frankenstein” nature.
Emacs-centric experimentation

take best components for software development in Emacs & apply them to PVS

new features: theory browsing, non-ASCII fonts, customized theory and sequent pretty-printing, context awareness, type-aware completion, smart cut-and-paste, implicit hyperlinking, etc.
% booleans declares the type $\mathbb{B}$ and its abbreviation $\mathbb{B}$, along
% with the $\mathbb{B}$ constants true and false and the $\mathbb{B}$ connectives
% The properties of the connectives are given later, but the connectives
% are built in to the typechecker so must be provided early on.
% Note: the $\mathbb{B}$ type could be defined as the enumeration type {false,
% true}, but booleans are primitive; the correct handling of enumeration
% types requires the $\mathbb{B}$ type.

booleans: THEORY
BEGIN

$\mathbb{B}$: NONEMPTY_TYPE
$\mathbb{B}$: NONEMPTY_TYPE = $\mathbb{B}$
⊥, T: $\mathbb{B}$
¬: [$\mathbb{B} \rightarrow $ $\mathbb{B}$]
⊗, &, ⊕, ⇒, ⇒, ⇔, ⇔: [$\mathbb{B}, \mathbb{B} \rightarrow $ $\mathbb{B}$]

END booleans

% equalities contains the declaration for =. It has a single type
% parameter. Properties of equality are given in equality_props.

equalities [T: TYPE]: THEORY
BEGIN

=: [T, T → $\mathbb{B}$]

END equalities

notequal[T: TYPE]: THEORY
BEGIN
κ, y: VAR T

≠(κ, y): $\mathbb{B}$ = ¬ (κ = y)

END notequal
% The PVS prelude.

% The prelude consists of theories that are built in to the PVS system.
% It is typechecked the same as any other PVS theory, but there are hooks
% in the typechecker that require most of these theories to be available,
% hence the order of the theories is important. For example, no formulas
% may be declared before the bools are available, as the formula is
% expected to have type bool. Since definitions implicitly involve both
% formulas and equality, the bools theory may not include any
% definitions. Formulas are given below as AXIOMs, POSTULATES, and LEMMAS.
% POSTULATES are formulas that can be proved using the decision procedures,
% but would have to be given as axioms in a pure development of the theory.
% AXIOMs are formulas that cannot be proved, and LEMMAS are formulas that
% have been proved.

booleans: THEORY
BEGIN

boolean: NONEMPTY_TYPE
bool: NONEMPTY_TYPE = boolean
FALSE, TRUE: bool
NOT: [bool -> bool]
AND, OR, IMPLIES, =>, WHEN, IFF, <=>: [bool, bool -> bool]

END bools

% equalities contains the declaration for =, It has a single type
% parameter. Properties of equality are given in equality_props.

equalities [T: TYPE]: THEORY
BEGIN

=: [T, T -> boolean]

END equalities

notequal[T: TYPE] THEORY
BEGIN

x, y: VAR T

END notequal
Problems with Emacs-based Version

- not “componentized”
- not a good platform for integration experimentation
- fragile and slow due to implementation in elisp (used wisent package = bison in elisp)
- Emacs as an interface/foundation inspires both devotion and revulsion
A Small Automatic Prover Interface

- formally specified interface for arbitrary provers performing fully automatic actions
- small interface to configure and control automated prover
- provided with demonstration implementations in Java, C, and OCaml
Tackle the Platform Problems First

- develop a generic component platform for formal tools
- derive platform from concrete experiences
- first case study is in generic interfaces to automated provers for ESC/Java2
- deal with interface and “sales” issues later
Automated Provers Interface

- only eight methods
  - `start_prover` and `stop_prover`
  - `set_prover_resource_flags`
  - `signature`
  - `declare_axiom`
  - `make_assumption` and `retract_assumption`
  - `is_valid`
Interface Design

Constraints

- resource specifications
  - general-purpose time, space, memory, and prover behavior hinting
- signature and theory independence
  - formally contextualize prover use
- assumption manipulation
  - retractions essential for primary use cases
Realizations of Interface

- Implementations now available/ongoing for Simplify, CVC3, PVS, and Coq
- Experimentation with multiple object logics and derivation of logics for different provers from “canonical” PVS theory
- All being implemented as Eclipse plugins
Interactive Provers Interface

- **challenge**: provide the same (lightweight) interface for an interactive theorem prover

- **observation**: user interactions are not (significantly) different than programmatic interactions

- **solution**: model the software interface according to using user behavior
**Prover/Editor**

- lightweight interactive prover framework
- implemented as an Eclipse plugin
- provides framework for common “perspectives” (i.e., theory browsing, syntax highlighting, preferences, contextual documentation, etc.) and prover interaction
- experimentation has shown that incorporating a new prover requires only couple hundred lines of code
(* <Insert License Here> *)

$Id: Domain.v 69 2006-03-06 20:16:11Z davidpichardie $ *


@author David Pichardie, ... *

Require Import Program.
Require Export Zarith.
Require Export List.
Open Scope Z_scope.

(** All semantic domains and basic operation are encapsulated in a module si **) Module Type SEMANTIC_DOMAIN.

(** We depend on the choices done for program data structures *)
Declare Module Prog : PROGRAM. Import Prog.

(** Common interface for numerics domains *)
Module Type NUMERIC.

Parameter t : Set.
Parameter toZ : t -> Z.
Parameter fromZ : Z -> t.
End NUMERIC.

Declare Module Byte : NUMERIC.
Declare Module Short : NUMERIC.
Declare Module Int : NUMERIC.

(** range of numerics interpretations *)
Parameter byte_range : forall b, \(-2^7 \leq b < 2^7\).
Parameter short_range : forall s, \(-2^{15} \leq s < 2^{15}\).
Parameter int_range : forall i, \(-2^{31} \leq i < 2^{31}\).

(* Remark: we don't need surjection, it will be a consequence of further declarations *)

(** conversion *)
Parameter b2Z : Byte -> Z.

Eclipse-independence

- Eclipse, Eclipse, Eclipse... enough about Eclipse!

- I agree!

- each subsystem is implemented as
  - a formally specified component
  - a command-line tool (e.g., for composition with pipes)
  - standard GNU interface (e.g., for Emacs)
Other Components Under Development

- automated provers: Simplify, SMT-LIB
- solvers like CVC3, Yices
- PVS, Coq, and Isabelle
- BML and Umbra
- VC generators and wp/sp calculi
- JML tool suite and ESC/Java2
- proof-transforming compiler
Challenge to the Community

• beg, borrow, or steal the best ideas from the software communities for your tools

• focus more on usable and useful interfaces (both GUI and API)

• eat your own dogfood

• get involved in the various “evidential toolbus” efforts (i.e., SRI’s stuff and the Mobius platform)