The AutoProof Verifier: Usability by Non-Experts and on Standard Code
F-IDE at FM 2015, Oslo

Carlo A. Furia, Chris Poskitt, and Julian Tschannen

Chair of Software Engineering
ETH Zürich
An observation

verification tools are often built by experts for experts

- usability by programmers with little formal methods experience may be very limited

- what can tool developers learn from non-expert usage?
“Non-expert” ...?
“Non-expert” ... ?
Two specific contexts of non-expert usage

- use by “serious non-experts”
- use on “standard code”

=> explored with reference to the AutoProof verifier
AutoProof: a brief introduction

• **AutoProof** is an auto-active verifier for the object-oriented language Eiffel
  => users interact indirectly through annotations in the source code,
  e.g. pre-/postconditions, class invariants, intermediate assertions

• annotated Eiffel code → Boogie code → verification conditions

• powerful and flexible *(when you know how!)*
  => *model-based contracts, semantic collaboration, major benchmarks*

• accessible within a **full-fledged formal IDE** or through a web-based interface
ACCOUNT

--- Deposit `amount` in this account.

**require**

amount_non_negative: 0 <= amount
do
  balance := balance + amount
**ensure**
  balance_set: balance = old balance + amount
end

withdraw (amount: INTEGER)

--- Withdraw `amount` from this account.

**require**

amount_non_negative: 0 <= amount
amount_available: amount <= balance
do
  balance := balance - amount
**ensure**
  balance_set: balance = old balance - amount
end

transfer (amount: INTEGER; other: ACCOUNT)

--- Transfer `amount` from this account to `other`.

**note** explicit: wrapping

**require**

amount_non_negative: 0 <= amount
amount_available: amount <= balance
not_alias: Current /= other
**modify** (Current, other)
do
  withdraw (amount)
  other.deposit (amount)
**ensure**
  deposit_made: other.balance = old other.balance + amount
end
Our web-based interface to AutoProof

```plaintext
-- Simple bank account class.
-- Try to fix it and make the verification go through.

class ACCOUNT

feature -- Access
    balance: INTEGER
        -- Balance of account.

feature -- Element change
    deposit (amount: INTEGER)
        -- Deposit `amount` on account.
        require
            amount_not_negative: amount >= 0
        do
            balance := balance + amount
        ensure

Run...

<table>
<thead>
<tr>
<th>Feature</th>
<th>Line</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCOUNT (invariant admissibility)</td>
<td></td>
<td>Successfully verified.</td>
</tr>
<tr>
<td>ANY.default_create (creator, inherited by ACCOUNT)</td>
<td></td>
<td>Successfully verified.</td>
</tr>
<tr>
<td>ACCOUNT.deposit</td>
<td></td>
<td>Successfully verified.</td>
</tr>
<tr>
<td>ACCOUNT.withdraw</td>
<td>31</td>
<td>Postcondition balance_decreased may be violated.</td>
</tr>
<tr>
<td>ACCOUNT.transfer</td>
<td>49</td>
<td>Postcondition balance_decreased may be violated.</td>
</tr>
</tbody>
</table>
```
Two specific contexts of non-expert usage

use by “serious non-experts”

use on “standard code”

=> explored with reference to the AutoProof verifier
Usability by “serious non-experts”

- students taking our graduate-level course on “Software Verification” used AutoProof in their projects

- can be considered serious users
  - \( \Rightarrow \) significant time spent learning the tool
  - \( \Rightarrow \) received proper training in deductive verification

- not thrown in at the deep end – some knowledge about what goes on “behind the magic”

- but still non-experts
  - \( \Rightarrow \) little-to-no prior experience with auto-active verifiers
  - \( \Rightarrow \) generally unaware of research challenges / state-of-the-art
Our “Software Verification” course

- overarching goal: convey the diversity of verification techniques – as well as their interplay
  => program logics, abstract interpretation, model checking, testing
  => fundamental topics weaved with advanced research

- strong basis in theory, but also advocates practice through a take-home project
  => chance to experience verification from start to finish
  – “warts and all”

- part of the ETH software engineering Master’s program; also open to undergraduates / PhD students
  => 10-20 students; 75% Master’s; 20% Bachelor’s; 5% PhD
Where AutoProof comes in
– the take-home project

• students had to specify, implement, and verify a **basic list data structure** and **sorting algorithms** that operate on it

• we provided a **skeleton Eiffel class**
  => signatures of routines to specify, implement, and verify
  => **partial specs / implementations of simpler routines**
  => **annotations for object structures (ownership / collaboration)**

```eiffel
sort
do
  if count > Max_count // 2 and has_small_elements (array) then
    array := bucket_sort (array)
  else
    array := quick_sort (array)
  end
ensure
  is_sorted (array)
  is_permutation (array.sequence, old array.sequence)
end
```
Where AutoProof comes in
– the take-home project

• students had to specify, implement, and verify a basic list data structure and sorting algorithms that operate on it

• we provided a skeleton Eiffel class
  => signatures of routines to specify, implement, and verify
  => partial specs / implementations of simpler routines
  => annotations for object structures (ownership / collaboration)

```eiffel
sort
do
  if count >= Max_count // 2 and has_small_elements (array) then
    array := bucket_sort (array)
  else
    array := quick_sort (array)
  end
ensure
  is_sorted (array)
  is_permutation (array.sequence, old array.sequence)
end
```

model-based annotations
Where AutoProof comes in
– the take-home project

Abstract arrays to mathematical sequences

is_permutation (a, b: SEQUENCE [INTEGER]): BOOLEAN

note
    status: functional, ghost

do
    Result := a.to_bag ~ b.to_bag
end
...and what happened?

– *results at a glance*

<table>
<thead>
<tr>
<th>ROUTINE</th>
<th>IMPLEMENTED</th>
<th>SPECIFIED</th>
<th>FULL PROOF</th>
<th>is_sorted ONLY</th>
</tr>
</thead>
<tbody>
<tr>
<td>add, extend, remove, ...</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>–</td>
</tr>
<tr>
<td>quick_sort</td>
<td>9</td>
<td>8</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>bucket_sort</td>
<td>9</td>
<td>6</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

(out of a total of 9 groups)
...and what happened?

– *model-based contracts*

- fully specifying functionality was not a problem

- but **struggled** to get verification to fully “go through”

- fixable through **lemmas** or **additional assertions**
  
  => *i.e. to trigger “some” missing Boogie axiom*
  
  => *but often non-obvious without knowing the Boogie encodings of models – somehow against the spirit?*

- led to **bloated annotation overheads**; almost “randomly” asserting general facts about models in the hope of triggering something to push the proof through
...and what happened?
– inconsistent specifications

• many groups altered the provided object structure annotations without understanding them

led to inconsistent specifications

• the vacuous proofs caught some groups out
  => false sense of success!
  => whereas experts, who know the capabilities of tools, might have been suspicious

• need to improve education and tool support
  => e.g. Boogie-style “smoke tests” for AutoProof
...and what happened?
– interface to formal IDE

• web-based interface was preferred almost universally

• its ease of use was seen as more valuable than having the additional tools of a full-fledged formal IDE

• (but we remark that this was a single-class project)
  => the full-fledged formal IDE might be better suited to more complex endeavours
Two specific contexts of non-expert usage

use by “serious non-experts”

use on “standard code”

=> explored with reference to the AutoProof verifier
Usability on “standard code”

• deductive verification often hinges on programs being written, specified, and annotated in a way that is amenable to automated reasoning
  => hardly scales to poorly modularised code
  => certain language features may require more cumbersome annotations than other, semantically equivalent features

• as “experts” we know the dark corners of tools, their limitations, and the software patterns/structures for which they don’t scale

• how does AutoProof fare when these standard assumptions are not met?
Our “standard code”

• assignments from our “Introduction to Programming” course

• three applications simulating increasingly complex board games; one library modelling an urban transport system
  => all rely on common data structures from the EiffelBase2 library
  (NB: recently fully verified!)
  => relied on master solutions by previous course instructors

• goal: verify given specifications against given implementations, whilst changing as little as possible
...and what happened?
– changes were unavoidable for verification

• given interface specification often too weak for modular reasoning – solved with stronger specifications

• frame specifications were absent (no native syntax)

• ownership / collaboration annotations needed to reason about dependent objects; complex invariants cluttered the proof space by default

• once (à la “static”) routines not natively supported

• several loop invariants and intermediate assertions needed
...and what happened?
– the hardest stuff

• hardest and most time-consuming task was determining the **missing loop invariants** – necessary for proving the (given) postconditions
  => sometimes required re-ordering instructions
  => or factoring out “initialisation” of variables

• some inter-dependent object structures could not be verified (without significant refactoring) – especially when inheritance was involved

• **default constructors** not compatible with AutoProof’s ownership / collaboration methodology

• **centralised, non-modular design** of the transport library
Final remarks

- **power/flexibility of rich specifications**
  => encouraged the ambitious students
  => *but led to some “pot shot” proofs towards the end*

- **inconsistent specs / vacuous verification**
  => *false sense of success for non-experts*
  => *education and better tool-level support needed*

- **complex program structures and object dependencies**
  => *not easily verified as an afterthought*

- **tool & documentation:** [http://se.inf.ethz.ch/research/autoproof/](http://se.inf.ethz.ch/research/autoproof/)
Upcoming talk:
A Fully Verified Container Library

Session 6A
12 noon, Thursday

Nadia Polikarpova