Using Theorem Provers for Testing: Foundations, Challenges and Future Directions

Burkhart Wolff

Université Paris-Sud, LRI, CNRS
Abstract

While Formal Testing and Theorem-Proving are still perceived as antagonisms by many, there is a growing research field using the combination of both to increase the applicability of Formal Methods in industry, in particular in the area of Safety-and Security critical systems requiring formal certifications.

In this talk, I will present research (partially funded by the Digiteo Foundation) around the HOL-TestGen System, which strives for a synthesis of interactive and automated theorem proving as well of different formal testing techniques. I will present results which are of mutual interest for both research areas as well as an outlook for future directions.
Overview

• Test vs. Proof: An old controversy
  • Can proofs guarantee the “Absence of Errors”
  • Are deductive verifiers “better” than testers?
  • Can we avoid Tests? Or Reality?

• HOL-TestGen: A verification and validation approach by Model-based Testing (MBT)

• HOL-TestGen: Achievements FOR Proofs

• The Future of (Model-based) Testing
Test vs. Proof:
An old controversy

- "Dijkstra's Verdict":
  
  Program testing can be used to show the presence of bugs, but never to show their absence!
Test vs. Proof:
An old controversy

● “Dijkstra's Verdict”:
  
  Program testing can be used to show the presence of bugs, but never to show their absence!

● Well, Dijkstra was party; so can he be trusted?
Test vs. Proof: An old controversy

- "Dijkstra's Verdict":

  Program testing can be used to show the presence of bugs, but never to show their absence!

- So: can proof-based verifications guarantee the "absence of bugs"?
Test vs. Proof: An old controversy

- An Architecture of a Program Verifier (VCC) HOL-Boogie [Böhme, Wolff]

![Diagram showing the flow of C compilation through Boogie and Z3](image-url)
Test vs. Proof: An old controversy

- The ugly reality:
  deductive verification methods make a lot of assumptions besides being costly in brain-power!
  - operational semantics should be faithfully executed
  - complex memory-machine model consistent (VCC: 800 axioms)
  - correctness of the vc generation (for concurrent C with “ownership”, “locks”, ... !):
    - correctness of the vc generator and prover
    - absence of an environment (= Operating System) that manipulates the underlying state.
Test vs. Proof: An old controversy

• Back to “Dijkstra's Verdict”:
  
  > Program testing can be used to show the presence of bugs, but never to show their absence!

• Deductive Verification infers Properties on infinite sets of inputs; aren't they then “always better than tests”? 
Test vs. Proof: An old controversy

Well, this depends on these assumptions ...
See the (very nice) example of Maria Christakis, where for a simple program:

```java
public class Cell {
    public int v;

    public static int M(Cell c, Cell d)
        requires c != null && d != null;
        requires c.v != 0 && d.v != 0;
        ensures result < 0;
    {
        if (sign(c.v) == sign(d.v))
            c.v = (-1) * c.v;

        return c.v * d.v;
    }
}
```
Test vs. Proof: An old controversy

- Well, this depends on these assumptions ...

... two different tools
- Clousot (deductive based verification)
- Pex (white-box tester)

provide altogether differently false results, since their underlying assumptions on arithmetics and memory model are simply different.

Accidently, the Pex-Verdict is actually more correct than Clousots ...
Test vs. Proof:
An old controversy

- "Dijkstra's Verdict":
  
  Program testing can be used to show the presence of bugs, but never to show their absence!

Can we actually always avoid testing?
Test vs. Proof: An old controversy

- “Dijkstra's Verdict”:
  
  Program testing can be used to show the presence of bugs, but never to show their absence!

- “Einstein's scepticism”:

  As far as the laws of mathematics refer to reality, they are not certain, as far as they are certain, they do not refer to reality.
Test vs. Proof: An old controversy

Model (behaviour, and data !)

System (hard + software)

a posteriori learning by experimenting
Test vs. Proof: An old controversy

Model (behaviour, and data !)

System (hard + software)

a priori
test-case generation

a posteriori
learning by experimenting
Test vs. Proof: An old controversy

Model (behaviour, and data !)

System (hard + software)

a priori

test-case generation

a posteriori

learning by experimenting

Validation Problem: What you can't do with Verification
Verification by
Model-based Testing ...

- ... can be done post-hoc; significant projects “reverse engineer” the model of a legacy system

- ... attempts to find bugs in specifications EARLY (and can thus complement proof-based verification ...)

- ... can help system integration processes in a partly unknown environment (“embedded systems”)

Nothing of this can be done by deductive verification methods!
Test vs. Proof: Is it actually still a controversy?

- Dijkstra - Test:
  - Would Dijkstra fly with an aeroplane which is verified by deduct. methods alone?

- Well, that's illegal.
  Certification bodies (CC, DO183) require tests, (and are very reluctant at proofs)
Test vs. Proof:
Is it actually still a controversy?

- Microsoft: Five major verification tools: Pex (Structural Test), SAGE (Fuzz Test) and Dafny, Spec#, VCC (VCG) use SMT solver Z3!

- Test and Proofs, are they actually adversaries? (Tony Hoare, POPL2012, “says meanwhile no”).
HOL-TestGen: A model-based approach to Verification

- Vision of HOL-Testgen
  - HOL-TestGen provides:
    - A formal testcase-generation method based on the solution of logical constraints
HOL-TestGen: A model-based approach to Verification

• HOL-TestGen provides:
  • A formal testcase-generation method based on the solution of logical constraints
  • Built-on top of an interactive theorem proving environment, it allows to combine automated provers with user intelligence
HOL-TestGen as Plugin in the Isabelle Architecture

Tools
- HOL-Z, HOL-TestGen,
- Simpl, HOL-Boogie, HOL-OCL

PIDE / jEdit

Scala System Interface
- integrators
- proof procedures
(simp, fast, auto, etc...)
- components:
  - datatype
  - record, ...
- code gen.

nano-kernel
- + kernel

ML running on multi-core arch
- C1
- C2
- C3
- C4

ATP

Argo/UML

Boogie/VCC
Why Reusing Isabelle

Isabelle has:

... a lot of Infrastructure not worth to re-invent.

We use it as:

Formal Methods Tool Framework

“The ECLIPSE of FM – Tools”
HOL-TestGen: “The Standard Workflow”

- Writing a test-theory (the “model”)
HOL-TestGen: “The Standard Workflow”

• Writing a **test-theory** (the “model”)

Example: Sorting in HOL

```haskell
fun ins :: "'(a::linorder) ⇒ 'a list ⇒ 'a list"
where    "ins x [] = [x]"
        "ins x (y#ys) = (if (x < y) then x#y#ys
                        else (y#(ins x ys)))"

fun sort :: "'(a::linorder) list ⇒ 'a list"
where    "sort [] = []"
        "sort (x#xs) = ins x (sort xs)"
```
HOL-TestGen: “The Standard Workflow”

- Writing a test-theory
- Writing a test-specification TS
HOL-TestGen: “The Standard Workflow”

• Writing a test-theory
• Writing a test-specification TS

pattern:

\[
\text{test_spec } \text{“pre } x \rightarrow \text{ post } x \ (\text{prog } x)"
\]
HOL-TestGen: “The Standard Workflow”

• Writing a **test-theory**
• Writing a **test-specification TS**

example:

```plaintext
test_spec "sort(l) = prog(l)"
```
HOL-TestGen: “The Standard Workflow”

• Writing a test-theory
• Writing a test-specification TS
• Conversion into test-theorem

(“Testcase Generation”)
HOL-TestGen: “The Standard Workflow”

• Writing a test-theory
• Writing a test-specification TS
• Conversion into test-theorem

(“Testcase Generation”)

apply(gen_test_cases 3 1 “prog”)
HOL-TestGen:
“The Standard Workflow”

• Writing a test-theory

• Writing a test-specification TS

• Conversion into test-theorem
  ("Testcase Generation")

\[
TC_1 \Rightarrow \ldots \Rightarrow TC_n \Rightarrow \text{THYP}(H_1) \Rightarrow \ldots \Rightarrow \text{THYP}(H_m) \Rightarrow TS
\]

• where testcases \( TC_i \) have the form

\[
\text{Constraint}_1(x) \Rightarrow \ldots \Rightarrow \text{Constraint}_k(x) \Rightarrow P(\text{prog} \ x)
\]

• and where \( \text{THYP}(H_i) \) are test-hypothesis
HOL-TestGen:
“The Standard Workflow”

• Writing a test-theory
• Writing a test-specification TS
• Conversion into test-theorem

Example:
[] = prog([])
[?X1] = prog([?X1])
[?X1 ≤ ?X2] ⇒ [?X1, ?X2] = prog([?X1, ?X2])
[?X1 > ?X2] ⇒ [?X2, ?X1] = prog([?X1, ?X2])


HOL-TestGen: “The Standard Workflow”

• Writing a test-theory
• Writing a test-specification TS
• Conversion into test-theorem

... 

5: THYP(∃ x y. is_sorted(PUT[x,y]) → 
               ∀ x y. is_sorted(PUT[x,y]))
6: is_sorted(PUT [?X, ?Y, ?X])
7: THYP(∃ x y z. is_sorted(PUT [x,y,z]) → 
               ∀ x y z. is_sorted(PUT [x,y,z]))
8: THYP(3 < |l| → is_sorted(PUT l))
HOL-TestGen:
“The Standard Workflow”

• Writing a test-theory
• Writing a test-specification TS
• Conversion into test-theorem
• Generation of test-data
HOL-TestGen: “The Standard Workflow”

• Writing a test-theory
• Writing a test-specification TS
• Conversion into test-theorem
• Generation of test-data

gen_test_data “...”
HOL-TestGen: "The Standard Workflow"

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem
- Generation of test-data

\[
\begin{align*}
[] &= \text{prog} [\ ] \\
[6,8] &= \text{prog} [6, 8] \\
[0,19] &= \text{prog} [19, 0]
\end{align*}
\]
HOL-TestGen: “The Standard Workflow”

• Writing a test-theory
• Writing a test-specification TS
• Conversion into test-theorem
• Generation of test-data
• Generating a test-harness
HOL-TestGen: “The Standard Workflow”

• Writing a test-theory
• Writing a test-specification TS
• Conversion into test-theorem
• Generation of test-data
• Generating a test-harness
• Run of test-harness and generation of test-document (a “test plan”)
HOL-TestGen:
A Larger Example: Red Black Trees

Red-Black-Trees: Test Specification

testspec :
(redinv t ∧
blackinv t)

(redinv (delete x t) ∧
blackinv (delete x t))

where delete is the program under test.
Theory: Explicit Test-Hypothesises

- What to do with infinite data-structures?
- What is the connection between test-cases and test statements and the test theorems?

⇒ Two problems, one answer: Introducing Testhypothesis “on the fly” ...

THYP :: “bool ⇒ bool”
THYP (x) ≡ x
Theory of HOL-TestGen

- One type of test hypothesis:

  Uniformity-Hypothesis (for TestCase C)

  \[ \text{THYP} \left( \exists a \in C. \ P a \rightarrow \forall a \in C. \ P a \right) \]
Theory of HOL-TestGen

● Another: Regularity Hypothesis $(\tau, k)$

● Consider the case $\tau = \text{list}(\alpha)$, $k = 2, 3, 4$:

\[
\begin{align*}
\text{size}(x::\tau) < 2 & = (x = []) \lor (\exists \ a. x = [a]) \\
\text{size}(x::\tau) < 3 & = (x = []) \lor (\exists \ a. x = [a]) \lor (\exists \ a \ b. x = [a,b]) \\
\text{size}(x::\tau) < 4 & = (x = []) \lor (\exists \ a. x = [a]) \lor (\exists \ a \ b. x = [a,b]) \lor (\exists \ a \ b \ c. x = [a,b,c])
\end{align*}
\]
Theory of HOL-TestGen

• ... derive the rule \((\tau \tau = \text{list}(\alpha \alpha), \ k = 2)\):

\[
\begin{array}{cccc}
[x=\text{[]}] & [x=\text{[a]}] & [x=\text{[a,b]}] \\
\ldots & \ldots & \ldots \\
P & \Lambda a. \ P & \Lambda a \ b. \ P & \text{THYP M}
\end{array}
\]

\[
P
\]

where \(M = (\text{size } x \geq k \rightarrow P)\)

• data separation lemma vs. “regularity hypothesis”
Theory Test-Hypothesis: Verifying Uniformity

• Reconsider the Uniformity Hypothesis:
  Case A: We test the hypothesis:

\[5: \text{THYP}(\exists x \ y. \ y < x \rightarrow [y,x] = \text{sort}([x,y])) \rightarrow \forall x \ y. \ y < x \rightarrow [y,x] = \text{sort}([x,y]))\]

i.e. we state the hypothesis as test-spec!
HOL-TestGen:
Achievements FOR TP Community

• Larger Case-Studies in Test and Proof
  • NPfIT, Firewalls
  • Recently: Test-case generation for Hardware
  • EURO-MILS Projects in Certification CC
• Isabelle Distributions 2009-1, 2011, 2012
  including pervasive parallelisation of Kernel
• P-IDE Interface
HOL-TestGen: Achievements FOR TP Community

- National Program for IT (NPfIT): Large Case-Study together with British Telecom
- Test-Goal: NHS patient record access control mechanism
- Large Distributed, Heterogeneous System
- Legally required Access Control Policy (practically mostly enforced on the application level)
HOL-TestGen: Achievements FOR TP Community

- National Program for IT (NPfIT):
  Large Case-Study together with British Telecom
- Led to development of the

  Unified Policy Framework (Isabelle theory)
  encompassing RBAC, ARBAC and Firewall Policies

  Work on Verified Policy-Transformations
  led to interesting application in Network-Security testing
Models of Systems for Tests

UPF Model instantiated with NPfIT AC Model

British Telecom Spine

a priori
test-case generation

a posteriori
learning by experimenting
HOL-TestGen:
Achievements FOR TP Community

- ANR Project Paral ITP: Testing fueled the Parallelization in the Kernel of Isabelle
  (thanks Digiteo!)

PIDE jEdit

PolysML multi-core
C1 C2 C3 C4

Isa Scala System Interface

decision procedures

nano-kernel + kernel

PIDE jEdit

Coq Scala System Interface

parallel Coq decision procedures

Parallel Coq kernel

Parallal OCaml Engine

PolySML multi-core
C1 C2 C3 C4

C1 C2 ... ... ... ... C15 C16

PolySML multi-core

C1 C2 C3 C4

C1 C2 ... ... ... ... C15 C16
HOL-TestGen: Achievements FOR TP Community

- a technology which is now attempted to be transferred to Coq ...
HOL-TestGen: Achievements FOR TP Community

- Isabelle: PIDE / jedit is meanwhile robust and stable and part of the Isabelle Distribution. Since Version 2013 the default interface.
- Support for advanced (nested) tool-tipping and hypertexting in the entire session.
- experiments with JAVA-Browsers.
- Coq: First Proof-of-Technologies to replace CoqIde available.
HOL-TestGen: Achievements FOR TP Community

Application: AFP

Isabelle/AFP:

- $\approx$ 122 sessions with diversity of single-core run-time (3s $\ldots$ 1h)
- parameters of fully pervasive parallelism:
  
<table>
<thead>
<tr>
<th>Hardware</th>
<th>CPU Cores</th>
<th>ML Worker</th>
<th>GC Threads</th>
<th>Theory and Proof Checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon with hyperthreading</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>(Isabelle/Isar)</td>
</tr>
</tbody>
</table>

- timing results:

  Finished LatticeProperties (0:00:15 elapsed time, 0:00:22 cpu time, factor 1.46)
  ...

  Finished JinjaThreads (0:32:59 elapsed time, 1:56:55 cpu time, factor 3.54)
  0:36:01 elapsed time, 5:17:18 cpu time, factor 8.80
Parallel fine-grained validation of structured proofs in jEdit - PIDE

theory Example
imports Main
begin

inductive path for rel :: "'a ⇒ 'a ⇒ bool" where
  base: "path rel x x"
| step: "rel x y ⇒ path rel y z ⇒ path rel x z"

theorem example:
  fixes x z :: 'a assumes "path rel x z" shows "P x z"
  using assms
proof induct
  case (base x)
  show "P x x" by auto
next
  case (step x y z)
  note "rel x y" and "path rel y z"
  moreover note "P y z"
  ultimately show "P x z" by auto
qed

end

(Isabelle2012-D)
The Future of (Model-based) Testing

• More “Formal Methods under the Hood”
  • HOL-TestGenFW, SAGE, SAL,
    Excel-Expert, Crash-Analyser, ...

• Cloudification
  • Excel-Expert ...

• Gamification
  • Pex4Fun, Edutainment...

• Parallelization
  • In Test and Proof ... but “no free lunch”...

• New Domain-Specializations
  • GUI-Testing, Model-Search, MKM ...
Conclusion: Test & Proof

• ... can never ever establish the absense of “Bugs” in a system! Never ever. Both of them.
• ... can, when combined, further increase confidence in verification results by using mutually independent assumptions.
• ... can, when combined, offer new ways to tackle abstraction and state space explosion. (Normalization Theorems, Massage of Constraint Systems, …)
Conclusion: HOL-Testgen

A formal testcase-generation method based on the solution of logical constraints

- Built-on top of an interactive theorem proving environment, it allows to combine automated provers with user intelligence
- has been applied in substantial case-studies (Firewall Case-Study, see TestCom/Fates 08)
- produces explicit test-hypothesis to establish a logical link between test and proof
- profits a lot from massive paralellization of symbolic computation ...
Sources Available

Version HOL-TestGen 1.7 (Isabelle 2011-1)

http://www.brucker.ch/projects/hol-testgen

Including Example Suite . . .
Case-Study: NPfIT

• Challenges:
  
  • access control rules for patient-identifiable information are complex and reflect the trade-off between patient confidentiality, usability, functional, and legislative constraints.
  
  • Traditional discretionary and mandatory access control and RBAC are insufficiently expressive to capture complex policies such as Legitimate Relationships, Sealed Envelopes or Patient Consent Management.
  
  • access rules of such a large system comprise not only elementary rules of data-access, but also access to security policies themselves enabling policy management. The latter is conventionally modeled in ABAC [6–8] and administrative RBAC [9, 10] models; A uniform modelling framework must be able to accommodate this.
  
  • The requirements are mandated by laws, official guidelines and ethical positions (e. g. [11, 12]) that are prone to change.
Case-Study: NPfIT

- Different “Information Gouvernance Principles” (= Policies):
  - Role-Based Access Control (RBAC): NPfIT uses administrative RBAC [9] to control who can access what system functionality. Each user is assigned one or more User Role Profile (URP). Each URP permits the user to perform several Activities.
  - Legitimate Relationship (LR): A user is only allowed to access the data of patients in whose care he is actually involved. Users are assigned to hierarchically ordered workgroups that reflect the organisational structure of a workplace.
  - Patient Consent (PC): Patients can opt out in having a Summary Care Record (SCR) at all, or to control uploads of data into the SCR. This requires additional mechanisms to manage consent.
  - Sealed Envelope (SE): The sealing concept is used to hide parts of an SCR from users. Kinds of seals: seal, seal and lock, clinician seal.
Modeling Framework: Unified Policy Framework (UPF)

- UPF (A Theory in HOL / for HOL-TestGen)
  - A Policy: A Decision Function
    (Modeling a “Policy Enforcement Point” in a System)

```plaintext
datatype α decision = allow α | deny α

types (α, β) policy = α ↦ β decision  (* = α ⇒ β option *)

notation α ↦ β = (α, β) policy
```
Modeling Framework: Unified Policy Framework (UPF)

- UPF (A Theory in HOL / for HOL-TestGen)

  - Policy Constructors

  ```
definition ∅ ≡ λ y. None

(* ∅ :: α ↦ β *)
```

```
definition p(x+↦t) ≡ p(x ↦ Some(allow t))
  p(x−↦t) ≡ p(x ↦ Some(deny t))

(* p :: α ↦ β *)

(* where  p(x ↦ t) ≡ λ y. if y = x then A else p y *)
```

```definition (*AllowAll :: "(α → β) ⇒ (α ↦ β)" *)

∀ A x. pf(x) ⇒ (λ x. case pf x of  Some y ⇒ Some(allow(y))
  I None ⇒ None)

(*DenyAll :: "(α → β) ⇒ (α ↦ β)"*)

∀ D x. pf(x) ⇒ (λ x. case pf x of  Some y ⇒ Some(allow(y))
  I None ⇒ None)```
Modeling Framework: Unified Policy Framework (UPF)

- UPF (A Theory in HOL / for HOL-TestGen)
  - Domain, Range and Restrictions on Policies (Z-like)

\[
\text{definition } A \equiv \{x. \exists y. x = \text{allow } y\}, \ D \equiv \{x. \exists y. x = \text{deny } y\}
\]

\[
\text{definition } \text{dom} :: \alpha \rightarrow \beta \Rightarrow \alpha \text{ set}
\]
\[
\text{where } \quad \text{dom } f \equiv \{x. f x \neq \text{None}\}
\]

\[
\text{definition } \text{ran} :: \alpha \rightarrow \beta \Rightarrow \beta \text{ set} \quad ...
\]

\[
\text{definition } _ \_ :: \alpha \text{ set} \Rightarrow \alpha \rightarrow \beta \Rightarrow \alpha \rightarrow \beta
\]
\[
\text{where } \quad S \_ p \equiv (\lambda x. \text{ if } x \in S \text{ then } p x \text{ else none}) \quad (* \text{domain restriction} *)
\]

\[
\text{definition } _ \_ :: \alpha \rightarrow \beta \Rightarrow \alpha \text{ set} \Rightarrow \alpha \rightarrow \beta \quad (* \text{range restriction} *)
\]

\[
\text{definition } _ \oplus _ :: \alpha \rightarrow \beta \Rightarrow \alpha \rightarrow \beta \Rightarrow \alpha \rightarrow \beta \quad (* \text{first fit override} *)
\]
Example: Firewalls

- Firewall Policies in UPF
  - Data:
    
    \[
    \text{ip-address} = \text{int} \times \text{int} \times \text{int} \times \text{int} \\
    \text{ip-packet} = \text{ip-address} \times \text{protocol} \times \text{content} \times \text{ip-address}
    \]

  - Firewall - Policies:
    
    \[
    \text{policy} : \text{ip-packet} \mapsto \text{ip-packet}
    \]

    ... this covers also Network Adress Translations (NAT's)
Example: Firewalls

- Firewall Policies in UPF
  
  Elementary Policies

  \[
  \text{definition}\quad \text{me-ftp} :: \text{ip-packet} \mapsto \text{ip-packet} \\
  \text{where}\quad \text{me-ftp} \equiv \emptyset ((192,22,14,76),\text{ftp},d,(192,22,14,76)) \\
  +
  (192,22,14,76),\text{ftp},d,(192,22,14,76))
  \]
Example: Firewalls

- **Firewall Policies in UPF**
  
  - **Elementary Policies**

    ```
    definition me-ftp :: ip-packet ⇔ ip-packet
    where       me-ftp ≡  ∅ ((192,22,14,76),ftp,d,(192,22,14,76)
                 +⇒(192,22,14,76),ftp,d,(192,22,14,76))
    ```

  - **Combined Policies:**

    ```
    definition me-none-else:: ip-packet ⇔ ip-packet
    where       me-none-else ≡ me-ftp ⊕ ∀D x. x
    ```
Example: Firewalls

- Firewall Policies in UPF
  
  - Elementary Policies

    definition me-ftp :: ip-packet ⇒ ip-packet
    where me-ftp ≡ ∅ ((192,22,14,76),ftp,d,(192,22,14,76) +⇒(192,22,14,76),ftp,d,(192,22,14,76))

  - Combined Policies:

    definition me-none-else:: ip-packet ⇒ ip-packet
    where me-none-else = me-ftp ⊕ ∀D x. x