Abstract

The Isabelle homepage describes Isabelle as “a generic proof assistant. It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus.” While this, without doubts, what most users of Isabelle are using Isabelle for, there is much more to discover: Isabelle is also a framework for building formal methods tools.

In this talk, I will report on our experience in using Isabelle for building formal tools for high-level specifications languages (e.g., OCL, Z) as well as using Isabelle’s core engine for new applications domains such as generating test cases from high-level specifications.
Motivation

This is only the tip of the iceberg

Outline

1 Motivation

2 Isabelle tools on top of Isabelle (Add-on)
   - HOL-OCL 1.x
   - HOL-OCL 2.x
   - HOL-TestGen

3 Conclusion
UML/OCL in a nutshell

- **UML**
  - Visual modeling language
  - Object-oriented development
  - Industrial tool support
  - OMG standard
  - Many diagram types, e.g.,
    - activity diagrams
    - class diagrams
    - ...

- **OCL**
  - Textual extension of the UML
  - Allows for annotating UML diagrams
  - In the context of class-diagrams:
    - invariants
    - preconditions
    - postconditions

<table>
<thead>
<tr>
<th>Challenges (for a shallow embedding)</th>
</tr>
</thead>
</table>

**Challenge 1:**

Can we find a injective, type preserving mapping of an object-oriented language (and datatypes) into HOL

\[ e : T \rightarrow e :: T \]

(including subtyping)?

**Challenge 2:**

Can we support verification in a modular way (i.e., no replay of proof scripts after extending specifications)?

**Challenge 3:**

Can we ensure consistency of our representation?

Developing formal tools for UML/OCL?

Turning UML/OCL into a formal method

1. A formal semantics of **object-oriented data models** (UML)
   - typed path expressions
   - inheritance
   - ...

2. A formal semantics of **object-oriented constraints** (OCL)
   - a logic reasoning over path expressions
   - large libraries
   - three-valued logic
   - ...

3. And of course, we want a tool (**HOL-OCL**)
   - a formal, machine-checked semantics for OO specifications,
   - an interactive proof environment for OO specifications.

Representing class types

- The "extensible records" approach
  - We assume a common superclass (\(O\)).
  - A `tag` type guarantees uniquenessby \(O::\{\alpha\} := \text{class}(O)\).
  - Construct class type as tuple along inheritance hierarchy:

- Advantages:
  - it allows for extending class types (inheritance),
  - subclasses are type instances of superclasses

- It allows for modular proofs, i.e.,
  - a statement \(\phi(x : (\alpha B))\) proven for class \(B\) is still valid after extending class \(B\).

- However, it has a major disadvantage:
  - modular proofs are only supported for **one** extension per class
The “extensible records” approach
- We assume a common superclass (0).
- A tag type guarantees uniqueness by ($O_{tag} := \text{classO}$).
- Construct class type as tuple along inheritance hierarchy:

$$B := (O_{tag} \times \text{oid}) \times (A_{tag} \times \text{String})$$

Advantages:
- it allows for extending class types (inheritance),
- subclasses are type instances of superclasses
  $\Rightarrow$ **it allows for modular proofs**, i.e.,
  a statement $\phi(x : (\alpha B))$ proven for class B is still valid after extending class B.

However, it has a major disadvantage:
- modular proofs are only supported for **one** extension per class

Advantages:
- it allows for extending class types (inheritance),
- subclasses are type instances of superclasses
  $\Rightarrow$ **it allows for modular proofs**, i.e.,
  a statement $\phi(x : (\alpha B))$ proven for class B is still valid after extending class B.

However, it has a major disadvantage:
- modular proofs are only supported for **one** extension per class
Representing class types

- The “extensible records” approach
  - We assume a common superclass (O).
  - A tag type guarantees uniqueness (O<sub>α</sub> := classO).
  - Construct class type as tuple along inheritance hierarchy:

\[
\alpha \ B := (O_{\alpha} \times \text{oid}) \times \left( (A_{\alpha} \times \text{String}) \times ((B_{\alpha} \times \text{Integer}) \times \alpha) \right)
\]

- Advantages:
  - it allows for extending class types (inheritance),
  - subclasses are type instances of superclasses

- It allows for modular proofs, i.e.,
  a statement \(\phi(x : : (\alpha \ B))\) proven for class B is still valid after extending class B.

- However, it has a major disadvantage:
  - modular proofs are only supported for one extension per class

An extensible object store

\[U_1^{\alpha_1} = O \times a_1^{\alpha_1}\]

\[U_n^{\alpha_n} = O \times a_n^{\alpha_n}\]

Idea: a general universe type

A universe type representing all classes of a class model

- supports modular proofs with arbitrary extensions
- provides a formalization of an extensible typed object store
An extensible object store

\[ U^O_{(\alpha O)} = O \times \alpha O \]

\[ U^A_{(\alpha A, \beta O)} = O \times (A \times \alpha A \sqcup \beta O) \sqcup \]

\[ U^B_{(\alpha B, \beta O, \beta A)} = O \times (A \times (B \times \alpha B \sqcup \beta A) \sqcup \beta O) \sqcup \]

\[ U^{A,B,C}_{(\alpha A, \alpha B, \alpha C, \beta O, \beta A)} = O \times (A \times (B \times \alpha B \sqcup (C \times \alpha C \sqcup \beta A)) \sqcup \beta O) \sqcup \]
An extensible object store

Merging universes

Operations accessing the object store

- injections: \( \text{mk}_\alpha o = \text{Inl} o \) with type \( \alpha^0 0 \rightarrow U^0_\alpha \)
- projections: \( \text{get}_\alpha u = u \) with type \( U^0_\alpha \rightarrow \alpha^0 0 \)
- type casts: 
  - \( \text{A}_{\text{OJ}} = \text{get}_\alpha \circ \text{mk}_\alpha \) with type \( \alpha^A A \rightarrow (\alpha^A + \beta^A) 0 \)
  - \( \text{O}_{\text{AJ}} = \text{get}_\alpha \circ \text{mk}_\beta \) with type \( (\alpha^A + \beta^A) 0 \rightarrow \alpha^A A \)
  - ...
“Checking” subtyping

For each UML model, we have to show several properties:

- subclasses are of the superclasses kind:
  
  \[
  \text{isType}_B \self \quad \text{isKind}_A \self
  \]

- "re-casting":
  
  \[
  \text{isType}_B \self \quad \text{self}_{A\cup B} \neq \bot \land \text{isType}_B (\text{self}_{A\cup B})
  \]

- monotonicity of invariants, ...

All rules are derived automatically

HOL-OCL

- HOL-OCL provides:
  
  - a formal, machine-checked semantics for OO specifications,
  - an interactive proof environment for OO specifications.

- HOL-OCL is integrated into a toolchain providing:
  
  - extended well-formedness checking,
  - proof-obligation generation,
  - methodology support for UML/OCL,
  - a transformation framework (including PO generation),
  - code generators,
  - support for SecureUML.

- HOL-OCL is publicly available:
  

The HOL-OCL architecture

HOL-OCL User Interface (extended Proof General)

su4sml

Encoder

Repository

Code-Gen.

UML/OCL

PO-Manager

Model-Trans.

Isabelle/HOL

Standard ML (PolyML, sml/NJ)

The HOL-OCL user interface
The HOL-OCL high-level language

The HOL-OCL proof language is an extension of Isabelle’s Isar language:

- importing UML/OCL:
  ```isar```
  ```import_model "SimpleChair.zargo" "AbstractSimpleChair.ocl"
  include_only "AbstractSimpleChair"
  ```

- check well-formedness and generate proof obligations for refinement:
  ```isar```
  ```analyze_consistency [data_refinement] "AbstractSimpleChair"
  ```

- starting a proof for a generated proof obligation:
  ```isar```
  ```po "AbstractSimpleChair.findRole_enabled"
  ```

- generating code:
  ```isar```
  ```generate_code "java"
  ```

The encoder

The model encoder is the main interface between su4sml and the Isabelle based part of HOL-OCL. The encoder

- declares HOL types for the classifiers of the model,
- encodes
  - type-casts,
  - attribute accessors, and
  - dynamic type and kind tests implicitly declared in the imported data model,
- encodes the OCL specification, i.e.,
  - class invariants
  - operation specifications
  and combines it with the core data model, and
- proves (automatically) methodology and analysis independent properties of the model.

Tactics (proof procedures)

- OCL, as logic, is quite different from HOL (e.g., three-valuedness)
- Major Isabelle proof procedures, like simp and auto, cannot handle OCL efficiently.
- HOL-OCL provides several UML/OCL specific proof procedures:
  - embedding specific tactics (e.g., unfolding a certain level)
  - a OCL specific context-rewriter
  - a OCL specific tableaux-prover
  - ...
  These language specific variants increase the degree of proof for OCL.

Proof obligation generator

A framework for proof obligation generation:

- Generates proof obligation in OCL plus minimal meta-language.
- Only minimal meta-language necessary:
  - Validity: \( \models \bullet \models \bullet \)
  - Meta level quantifiers: \( \exists \bullet \forall \bullet \)
  - Meta level logical connectives: \( \bullet \lor \bullet \land \bullet \neg \bullet \)
- Examples for proof obligations are:
  - (semantical) model consistency
  - Liskov’s substitution principle
  - refinement conditions
  - ...
- Can be easily extended (at runtime).
- Builds, together with well-formedness checking, the basis for tool-supported methodologies.
# Outline

1. Motivation
2. Isabelle tools on top of Isabelle (Add-on)
   - HOL-OCL 1.x
   - HOL-OCL 2.x
   - HOL-TestGen
3. Conclusion

---

## HOL-OCL 2.0 (Featherweight OCL)

This section covers Featherweight OCL (HOL-OCL 2.0), a tool developed by Achim D. Brucker. It discusses how to ensure system correctness, security, and safety using both (inductive) verification and testing.

### How to ensure system correctness, security, and safety?

<table>
<thead>
<tr>
<th><strong>Verification</strong></th>
<th><strong>Testing</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- Formal (mathematical) proof</td>
<td>- Execution of test cases</td>
</tr>
<tr>
<td>- Can show absence of all failures relative to specification</td>
<td>- Can show failures on real system</td>
</tr>
<tr>
<td>- Specification of based on abstractions</td>
<td>- Only shows failures for the parts of the system</td>
</tr>
<tr>
<td>- Requires expertise in Formal Methods</td>
<td>- Requires less skills in Formal Methods</td>
</tr>
<tr>
<td><strong>In industry:</strong> only for highly critical systems (regulations, certification)</td>
<td><strong>In industry:</strong> widely used (often &gt; 40% of dev. effort)</td>
</tr>
</tbody>
</table>

---

Achim D. Brucker

Isabelle: Not Only a Proof Assistant

September 24, 2015
Is testing a “poor man’s verification?”
Or: Why should I test if I did a verification and vice versa?

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

- Assume you can choose between two aircraft for your next travel:
  - Aircraft A:
    - Fully formally verified
    - Total number of flights: 0
  - Aircraft B:
    - Fully tested
    - Total number of flights: 1,000

- Which aircraft would you take for your next trip?
- Which aircraft would Dijkstra take?

Observe:
Both methods have their unique advantages
Recommendation:
Use a combination of verification and testing
Our Vision:
An integrated approach for test and verification
Implementing our vision in Isabelle: HOL-TestGen

An interactive model-based test tool
- built upon the theorem prover Isabelle/HOL
- specification language: HOL
- unique combination of test and proof
  - verification environment
  - user controllable test-hypotheses
  - verified transformations
- supports the complete MBT workflow
- basis for domain-specific extensions
- successfully used in large case-studies

freely available at: http://www.brucker.ch/projects/hol-testgen/

The HOL-TestGen architecture

Seamless combination of testing and verification

Black-box vs. white-box:
- Specification-based black-box test as default
- White-box and Grey-box also possible

Unit vs. sequence testing
- Unit testing straight forwards
- Sequence testing via monadic construction

Coverage:
- Path Coverage (on the specification) as default

Scalability:
- Verified test transformations can increase testability by several orders of magnitude

Excursus: test hypothesis – the difference between test and proof

Idea: We introduce formal test hypothesis “on the fly”
- Technically, test hypothesis are marked using the following predicate:
  \[ \text{THYP}(x) \equiv x \]
- Two test hypotheses are common:
  - Regularity hypothesis: captures infinite data structures (splits), e.g., for lists
    \[
    \begin{align*}
    \forall x &. (\exists a \in \mathbb{N}. (x = [a]) \lor (x = [a, b])) \rightarrow \\
    \bigwedge &. P \land \bigwedge &. a \rightarrow \bigwedge &. b \rightarrow \bigwedge &. h \rightarrow \text{THYP}(\forall x. k < \text{size} x \rightarrow P x) \\
    \end{align*}
    \]
  - Uniformity hypothesis: captures test data selection
    “Once a system under test behaves correct for one test case, it behaves correct for all test cases”
    \[ n = 1 \rightarrow \text{THYP}(\exists x. C x \rightarrow TS x) \rightarrow \text{THYP}(\forall x. C x \rightarrow TS x) \rightarrow \text{THYP}(\forall x. C x \rightarrow TS x) \]

Test case generation: an example

theory TestPrimRec
imports Main
begin
primrec
x mem [] = False
x mem (y#S) = if y = x then True else x mem S

Result:
1. prog ?x1 [?x1]
2. prog ?x2 [?x2, ?x2]
3. ?a3 # ?x3 == prog ?x3 [?a3, ?x3]
4. \[ \text{THYP}(\exists x. \text{prog} x [x] \rightarrow \text{prog} x [x]) \]
7. \[ \text{THYP}(\forall S. 3 \leq \text{size} S \rightarrow \text{x mem} S \rightarrow \text{prog} x [S]) \]

test_spec:
"x mem S == prog x S"  
apply(gen_testcase)
Use case: testing firewall policies

<table>
<thead>
<tr>
<th>source</th>
<th>destination</th>
<th>protocol</th>
<th>port</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet</td>
<td>dmz</td>
<td>udp</td>
<td>25</td>
<td>allow</td>
</tr>
<tr>
<td>Internet</td>
<td>dmz</td>
<td>tcp</td>
<td>80</td>
<td>allow</td>
</tr>
<tr>
<td>dmz</td>
<td>intranet</td>
<td>tcp</td>
<td>25</td>
<td>allow</td>
</tr>
<tr>
<td>intranet</td>
<td>Internet</td>
<td>udp</td>
<td>80</td>
<td>allow</td>
</tr>
<tr>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>deny</td>
</tr>
</tbody>
</table>

Our goal: Show correctness of the configuration and implementation of active network components.

Today: firewalls are stateless packet filters.

Our approach also supports (not considered in this talk):
- network address translation (NAT)
- port translation, port forwarding
- stateful firewalls

The policy

<table>
<thead>
<tr>
<th>source</th>
<th>destination</th>
<th>protocol</th>
<th>port</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet</td>
<td>dmz</td>
<td>udp</td>
<td>25</td>
<td>allow</td>
</tr>
<tr>
<td>Internet</td>
<td>dmz</td>
<td>tcp</td>
<td>80</td>
<td>allow</td>
</tr>
<tr>
<td>dmz</td>
<td>intranet</td>
<td>tcp</td>
<td>25</td>
<td>allow</td>
</tr>
<tr>
<td>intranet</td>
<td>Internet</td>
<td>udp</td>
<td>80</td>
<td>allow</td>
</tr>
<tr>
<td>any</td>
<td>any</td>
<td>any</td>
<td>any</td>
<td>deny</td>
</tr>
</tbody>
</table>

Definition TestPolicy where

TestPolicy = allow_port udp 25 internet dmz ⊙
allow_port tcp 80 internet dmz ⊙
allow_port tcp 25 dmz intranet ⊙
allow_port tcp 993 intranet dmz ⊙
allow_port udp 80 intranet internet ⊙

D_U

where D_U is the policy that denies all traffic.

HOL model of a firewall policy

A firewall makes a decision based on single packets.
- \textbf{types} \((\alpha, \beta)\) packet
  \[ = \text{id} \times (\alpha::adr) \text{ src} \times (\alpha::adr) \text{ dest} \times \beta \text{ content} \]

Different address and content representations are possible.

- A policy is a mapping from packets to decisions (allow, deny, . . .):
  \textbf{types} \((\alpha, \beta)\) Policy = \((\alpha, \beta)\) packet \rightarrow \text{decision}

- Policy combinators allow for defining policies:
  \textbf{definition}
  allow_all_from :: (\alpha::adr) net \Rightarrow (\alpha, \beta) Policy
  where
  allow_all_from src_net = \{pa. src pa \sqsubseteq src_net\} \sqsubset AU

Testing stateless firewalls

The test specification:
- \textbf{test_spec} test: "P x \Rightarrow \text{FUT} x = \text{Policy} x"

- \textbf{FUT}: Placeholder for Firewall Under Test
- Predicate P restricts packets we are interested in, e.g., wellformed packets which cross some network boundary

- Core test case generation algorithm:
  - compute conjunctive-normal form
  - find satisfying assignments for each clause (partition)

- Generates test data like (simplified):
  \text{FUT}(1, ((8,13,12,10),6,tcp), ((172,168,2,1),80,tcp), data) = [(\text{deny}())]
Problems with the direct approach

The direct approach does not scale:

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Networks</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Rules</td>
<td>12</td>
<td>9</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>TC Generation Time (sec)</td>
<td>26382</td>
<td>187</td>
<td>59364</td>
<td>1388</td>
</tr>
<tr>
<td>Test Cases</td>
<td>1368</td>
<td>264</td>
<td>1544</td>
<td>470</td>
</tr>
</tbody>
</table>

**Reason:**
- Large cascades of case distinctions over input and output
- Many combinations due to subnets
- Pre-partitioning of test space according to subnets

Model transformations for TCG

Idea is fundamental to model-based test case generation. E.g.:
- if \( x < -10 \) then if \( x < 0 \) then \( P \) else \( Q \) else \( Q \)
- if \( x < -10 \) then \( P \) else \( Q \)
lead to different test cases

The following two policies produce a different set of test cases:
- AllowAll dmz internet ⊕ DenyPort dmz internet 21 ⊕ DU
- AllowAll dmz internet ⊕ DU

A typical transformation

Remove all rules
- allowing a port between two networks,
- if a former rule already denies all the rules between these two networks

```haskell
fun removeShadowRules2 ::
where
removeShadowRules2 ((AllowPortFromTo x y p)#z) =
  if (DenyAllFromTo x y) ∈ (set z)
  then removeShadowRules2 z
  else (AllowPortFromTo x y p)#(removeShadowRules2 z)
| removeShadowRules2 (x#y) = x#(removeShadowRules2 y)
| removeShadowRules2 [] = []
```

Correctness of the normalisation

Correctness of the normalization must hold for arbitrary input policies, satisfying certain preconditions. As HOL-TestGen is built upon the theorem prover Isabelle/HOL, we can prove formally the correctness of such normalisations:

```isabelle
theorem C_eq_normalize:
  assumes member DenyAll p
  assumes allNetsDistinct p
  shows C (list2policy (normalize p)) = C p
```
Empirical results

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Normalized</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Networks</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Rules</td>
<td>12</td>
<td>9</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>TC Generation Time (sec)</td>
<td>26382</td>
<td>187</td>
<td>59364</td>
<td>1388</td>
</tr>
<tr>
<td>Test Cases</td>
<td>1368</td>
<td>264</td>
<td>1544</td>
<td>470</td>
</tr>
<tr>
<td>Normalized</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rules</td>
<td>14</td>
<td>14</td>
<td>24</td>
<td>26</td>
</tr>
<tr>
<td>Normalization (sec)</td>
<td>0.6</td>
<td>0.4</td>
<td>1.1</td>
<td>0.8</td>
</tr>
<tr>
<td>TC Generation Time (sec)</td>
<td>0.9</td>
<td>0.6</td>
<td>1.2</td>
<td>0.7</td>
</tr>
<tr>
<td>Test Cases</td>
<td>20</td>
<td>20</td>
<td>34</td>
<td>22</td>
</tr>
</tbody>
</table>

The normalization of policies decreases the number of test cases and the required test case generation time by several orders of magnitude.

Conclusion

Modern interactive theorem provers can be used as frameworks for building formal methods tools.

If you “prototype” formal methods tools, consider
- to reuse the infrastructure of your theorem prover of choice

Isabelle provides a lot of features:
- defining nice syntax for DSLs
- defining new top-level commands
- developing own tactics
- generate code
- ...

There is another nice example: attend the next talk by Sebastian!

Thank you for your attention!

Any questions or remarks?
Related Publications I

Achim D. Brucker, Lukas Brügger, Paul Kearney, and Burkhart Wolff.
Verified firewall policy transformations for test-case generation.

Achim D. Brucker, Lukas Brügger, and Burkhart Wolff.
HOL-TestGen/FW: An environment for specification-based firewall conformance testing.

Achim D. Brucker, Lukas Brügger, and Burkhart Wolff.
Formal firewall conformance testing: An application of test and proof techniques.

Achim D. Brucker, Delphine Longuet, Frédéric Tugot, and Burkhart Wolff.
On the semantics of object-oriented data structures and path expressions.

Achim D. Brucker and Burkhart Wolff.
hol-ocl – A Formal Proof Environment for UML/OCL.
in José Fiadeiro and Paola Inverardi, editors, Fundamental Approaches to Software Engineering (FASE), number 4961 in Lecture Notes in Computer Science, pages 97–100. Springer-Verlag, 2008.

Achim D. Brucker and Burkhart Wolff.
Extensible universes for object-oriented data models.

Achim D. Brucker and Burkhart Wolff.
Semantics, calculi, and analysis for object-oriented specifications.

Achim D. Brucker and Burkhart Wolff.
On theorem prover-based testing.

Related Publications II

Achim D. Brucker and Burkhart Wolff.
hol-z 2.0: A proof environment for Z specifications.

Achim D. Brucker and Burkhart Wolff.
Ehrhard-Bruijn notation for the pi-calculus.
in Interactive Computer Science and Mathematics (ICS M), 2006.