Isabelle: Not Only a Proof Assistant

Achim D. Brucker
achim@brucker.ch http://www.brucker.ch/

joint work with Lukas Brügger, Delphine Longuet, Yakoub Nemouchi, Frédéric Tuong, Burkhart Wolff

Proof Assistants and Related Tools - The PART Project
Technical University of Denmark, Kgs. Lyngby, Denmark
September 24, 2015
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.
Isabelle

What is Isabelle?

Isabelle is 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. Isabelle was originally developed at the University of Cambridge and Technische Universität München, but now includes numerous contributions from institutions and individuals worldwide. See the Isabelle overview for a brief introduction.

Now available: Isabelle2015

Download for Linux

Download for Windows - Download for Mac OS X

Some highlights:

- Improved Isabelle/Isar Prover IDE: folding / bracket matching for Isar, support for BibTeX files, improved graphview panel, improved scheduling for asynchronous print commands (e.g. Sledgehammer provers).
- Support for private and qualified name space modifiers.
- Structural composition of proof methods (meth1; meth2) in Isar.
What is Isabelle?

Isabelle is 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. Isabelle was originally developed at the University of Cambridge and Technische Universität München, but now includes numerous contributions from institutions and individuals worldwide. See the Isabelle overview for a brief introduction.
What is Isabelle?

Isabelle is 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. Isabelle was originally developed at the University of Cambridge and Technische Universität München, but now includes numerous contributions from institutions and individuals worldwide. See the Isabelle overview for a brief introduction.
Motivation

Achim D. Brucker

Isabelle: Not Only a Proof Assistant

September 24, 2015

text

The square root of any prime number (including 2) is irrational.

theorem sqrt_prime_irrational:
  assumes "prime (p :: nat)"
  shows "\sqrt p \notin \mathbb{Q}"
proof
  from <prime p> have p: "1 < p" by (simp add: prime_nat_def)
  assume \sqrt p \in \mathbb{Q}
  then obtain m :: nat where
    n: "n \neq 0" and sqrt_rat: "\sqrt p = m / n"'
    and gcd: "gcd m n = 1" by (rule Rats_abs_nat_div_natE)
  have eq: "m^2 = p * n^2"
  proof -
    from n and sqrt_rat have 'm = \sqrt p * n" by simp
    then have "m^2 = (\sqrt p)^2 * n^2"
    by (auto simp add: power2_eq_square)
  qed
proof (prove): depth 2

using this:
  sqrt (real p) \in \mathbb{Q}

goal (1 subgoal):
  1. (\forall m. n \neq 0 \implies \sqrt (real p) * real m / real n \implies coprime m n \implies thesis) \implies thesis
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

```
context Account
inv: 0 <= id

context Account::deposit(a:Integer):Boolean
pre: 0 < a
post: balance = balance@pre+a
and id = id@pre
```

```
Account
balance:Integer
id:Integer
getId():Integer
getBalance():Integer
deposit(a:Integer):Boolean
withdraw(a:Integer):Boolean

accounts
1..*
```

```
context Account::deposit(a:Integer):Boolean
pre: 0 < a
post: balance = balance@pre+a
and id = id@pre
```

Achim D. Brucker
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.
Challenges (for a shallow embedding)

■ Challenge 1:

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

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

(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?
Representing class types

- The “extensible records” approach
  - We assume a common superclass (0).
  - A tag type guarantees uniqueness by \( O_{\text{tag}} := \text{classO} \).
  - Construct class type as tuple along inheritance hierarchy:

\[
\alpha_B := (O_{\text{tag}} \times \text{oid}) \times ((A_{\text{tag}} \times \text{String}) \times ((B_{\text{tag}} \times \text{Integer}) \times \alpha))
\]

- 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.
Representing class types

- The “extensible records” approach
  - We assume a common superclass (0).
  - A tag type guarantees uniqueness by \( O_{\text{tag}} := \text{classO} \).
  - Construct class type as tuple along inheritance hierarchy:
    \[
    B := (O_{\text{tag}} \times \text{oid}) \times ((A_{\text{tag}} \times \text{String}) \times (B_{\text{tag}} \times \text{Integer}) \times \alpha)
    \]

- 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
Representing class types

- The “extensible records” approach
  - We assume a common superclass (O).
  - A tag type guarantees uniqueness by (O\text{tag} := \text{classO}).
  - Construct class type as tuple along inheritance hierarchy:

\[
B := (O\text{tag} \times \text{oid})
\]

- Advantages:
  - it allows for extending class types (inheritance),
  - subclasses are type instances of superclasses
  \[\Rightarrow \text{it allows for modular proofs}, \ i.e.,\]
  - 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 (0).
  - A tag type guarantees uniqueness by (O\text{tag} := \text{classO}).
  - Construct class type as tuple along inheritance hierarchy:
    \[ B := (O_{\text{tag}} \times \text{oid}) \times \left( (A_{\text{tag}} \times \text{String}) \times (B_{\text{tag}} \times \text{Integer}) \times \alpha \right) \]

- Advantages:
  - It allows for extending class types (inheritance),
  - Subclasses are type instances of superclasses
  \[ \Rightarrow \text{it allows for modular proofs}, \text{ i.e.,} \]
  - 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 by \( O_{\text{tag}} := \text{classO} \).
  - Construct class type as tuple along inheritance hierarchy:
    \[
    B := (O_{\text{tag}} \times \text{oid}) \times ((A_{\text{tag}} \times \text{String}) \times ((B_{\text{tag}} \times \text{Integer}) \times \alpha))
    \]

- 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
Representing class types

- The “extensible records” approach
  - We assume a common superclass (0).
  - A tag type guarantees uniqueness by \((O_{\text{tag}} := \text{class0})\).
  - Construct class type as tuple along inheritance hierarchy:
    \[
    \alpha \ B := (O_{\text{tag}} \times \text{oid}) \times \left( (A_{\text{tag}} \times \text{String}) \times \left( (B_{\text{tag}} \times \text{Integer}) \times \alpha \right) \right)
    \]

- 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
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 a extensible typed object store
An extensible object store

\[ \mathcal{U}(\alpha^0) = O \times \alpha_\bot^0 \]
An extensible object store

\[ U^{(a^0)} = O \times a^0 \]
An extensible object store

\[ U^0_{(\alpha^0)} = O \times \alpha^0_\perp \]

\[ U^1_{(\alpha^A, \beta^0)} = O \times (A \times \alpha^A_\perp + \beta^0_\perp) \]
An extensible object store

\[ \mathcal{U}^0_{(\alpha^0)} = O \times \alpha^0 \]

\[ \mathcal{U}^1_{(\alpha^A, \beta^O)} = O \times (A \times \alpha^A + \beta^O) \]

\[ \mathcal{U}^2_{(\alpha^B, \beta^O, \beta^A)} = O \times (B \times \alpha^B + \beta^A + \beta^O) \]
An extensible object store

\[ U_{(\alpha^0)}^0 = O \times \alpha^0 \]

\[ U_{(\alpha^A, \beta^0)}^1 = O \times (A \times \alpha^A_\perp + \beta^0)_\perp \]

\[ U_{(\alpha^B, \beta^0, \beta^A)}^2 = O \times (A \times (B \times \alpha^B_\perp + \beta^A) \perp + \beta^0) \perp \]
An extensible object store

\[ \mathcal{U}_{(\alpha^0)} = O \times \alpha^0 \]
\[ \mathcal{U}_{(\alpha^A, \beta^0)} = O \times (A \times (\alpha^A \perp + \beta^0) \perp) \]
\[ \mathcal{U}_{(\alpha^A, \alpha^C, \beta^0, \beta^A)} = O \times (A \times (B \times (\alpha^B \perp + (C \times \alpha^C \perp + \beta^A) \perp) \perp + \beta^0) \perp) \]

Achim D. Brucker
An extensible object store

\[ \mathcal{U}^0_{(\alpha^0)} = O \times \alpha^0 \]

\[ \mathcal{U}^1_{(\alpha^A, \beta^0)} = O \times (A \times \alpha^A \perp + \beta^0) \]

\[ \mathcal{U}^2_{(\alpha^B, \alpha^C, \beta^0, \beta^A)} = O \times (A \times (B \times \alpha^B \perp + (C \times \alpha^C \perp + \beta^A)) \perp + \beta^0) \]

\[ \mathcal{U}^3_{(\alpha^B, \alpha^C, \beta^0, \beta^A)} \prec \mathcal{U}^2_{(\alpha^B, \beta^0, \beta^A)} \prec \mathcal{U}^1_{(\alpha^A, \beta^0)} \prec \mathcal{U}^0_{(\alpha^0)} \]
Merging universes

\[ u^1: \quad u^2a: \]

\[ u^2b: \quad u^3: \]

Non-conflicting Merges
Merging universes

Non-conflicting Merges

Conflicting Merges
Operations accessing the object store

- **injections**
  \[ \text{mk}_O \circ o = \text{Inl} \circ o \]
  with type \( \alpha^O \ 0 \rightarrow U^0_{\alpha^O} \)

- **projections**
  \[ \text{get}_O \circ u = u \]
  with type \( U^0_{\alpha^O} \rightarrow \alpha^O \ 0 \)

- **type casts**
  \[ A^{[O]} = \text{get}_O \circ \text{mk}_A \]
  with type \( \alpha^A \ A \rightarrow (A \times \alpha^A_\bot + \beta^O) \ 0 \)
  \[ O^{[A]} = \text{get}_A \circ \text{mk}_O \]
  with type \( (A \times \alpha^A_\bot + \beta^O) \ 0 \rightarrow \alpha^A \ A \)

- ...
“Checking” subtyping

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

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

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

- monotonicity of invariants, ...
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
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:

  ```isabelle
  import_model "SimpleChair.zargo" "AbstractSimpleChair.ocl"
  include_only "AbstractSimpleChair"
  ```

- check well-formedness and generate proof obligations for refinement:

  ```isabelle
  analyze_consistency [data_refinement] "AbstractSimpleChair"
  ```

- starting a proof for a generated proof obligation:

  ```isabelle
  po "AbstractSimpleChair.findRole_enabled"
  ```

- generating code:

  ```isabelle
  generate_code "java"
  ```
The encoder

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

- declarers 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.
A framework for proof obligation generation:

- Generates proof obligation in OCL plus minimal meta-language.
- Only minimal meta-language necessary:
  - Validity: \( \models _\_ , _\_ \models _\_ \)
  - Meta level quantifiers: \( \exists _\_ , _\_ \exists _\_ , _\_ \)
  - Meta level logical connectives: \( _\_ \lor _\_ , _\_ \land _\_ , _\_ \neg _\_ \)
- 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)

theory Bank_Model imports "./src/UML_OCL"
begin

Class Savings < Account Attributes max : Currency

Association clients Between Bank [1 .. * ] Role banks
Client [1 .. * ] Role clients

Context c: Savings
  Inv "0.0 <\text{real} (c\text{.max})"
  Inv "c\text{.balance} \leq\text{real} (c\text{.max}) \text{ and } 0.0 \leq\text{real} (c\text{.balance})"

Context Bank ::= create_client(n:String, a:Integer, b:Bank)
  Pre "b\text{.clients} ->\text{forall}\text{Set}(c |c\text{.clientname} = n \text{ and } c\text{.age} > a)"
  Post "b\text{.clients} ->\text{exists}\text{Set}(c |c\text{.clientname} = n \text{ and } c\text{.age} > a)"

(* 2384 generated UML/OCL theorems *)

thm upOclAny_downSavings_Cast upOclAny_downAccount_cast upAccount

generation_syntax [ syntax_print, shallow, deep (THEORY Mode)

apply(auto simp: isdef down_cast_type savings_from_OclAny_to_OclAny)
done

lemma down_cast_kindClient_from_OclAny_to_Client :
  assumes iskin: "\tau \vdash ((X\cdot OclAny) . oclIsKindOf(Client))"
  assumes isdef: "\tau \vdash (\delta (X))"
  shows "\tau \vdash (X . oclAsType(Client)) \not\in\text{valid}"

apply(insert not_OclIsKindOfClient_then_OclAny_OclIsTypeOf)
apply(rule down_cast_type_oclany_from_OclAny_to_Client, simp)
apply(drule not_OclIsKindOfBank_then_OclAny_OclIsTypeOf, simp)
apply(rule down_cast_type_bankFromOclAny_to_Client, simp)
apply(drule not_OclIsKindOfAccount_then_OclAny_OclIsTypeOf, simp)
apply(auto simp: isdef down_cast_type savings_from_OclAny_to_OclAny)
done

[ 9 of 10] Compiling Argument
(Argument.hs, _build)
[10 of 10] Compiling Main
(Main.hs, _build/Main)

Linking Main ...

Proofs for inductive predicate(s) "rep_set_typeCurrent"
  Proving monotonicity ...
  Proving the introduction rules ...

Achim D. Brucker

Isabelle: Not Only a Proof Assistant

September 24, 2015
Outline

1. Motivation

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

3. Conclusion
How to ensure system correctness, security, and safety?

(Inductive) Verification

- Formal (mathematical) proof
- Can show absence of all failures relative to specification
- Specification of based on abstractions
- Requires expertise in Formal Methods

**In industry:** only for highly critical systems (regulations, certification)

Testing

- Execution of test cases
- Can show failures on real system
- Only shows failures for the parts of the system
- Requires less skills in Formal Methods

**In industry:** widely used (often > 40% of dev. effort)
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?
What should we do?

Vision: Use the Optimal Combination of Verification and Testing in an Integrated Approach

- Observation: Both methods have their unique advantages

- Recommendation: Use a combination of verification and testing

- Our Vision: An integrated approach for test and verification
What should we do?

Vision: Use the Optimal Combination of Verification and Testing in an Integrated Approach

Observation:
- Both methods have their unique advantages

Recommendation:
- Use a combination of verification and testing

Our Vision:
- An integrated approach for test and verification
What should we do?

Vision: Use the Optimal Combination of Verification and Testing in an Integrated Approach

Observation:
- 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:  
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:
  
  \[
  \text{THYP} : \text{bool} \Rightarrow \text{bool} \\
  \text{THYP}(x) \equiv x
  \]

- Two test hypotheses are common:
  - **Regularity hypothesis:** captures infinite data structures (splits), e.g., for lists
    
    \[
    \begin{align*}
    [x & = []] \\
    \ldots & \\
    \bigwedge a & \\
    P & \\
    \end{align*}
    \]
    
    \[
    P \implies \bigwedge a \implies P
    \]
    
    \[
    \text{THYP}(\forall x. k < \text{size} x \implies P)
    \]
  
  - **Uniformity hypothesis:** captures test data selection
    
    “Once a system under test behaves correct for one test case, it behaves correct for all test cases”
    
    \[
    \begin{align*}
    n) & \quad \{ C1 ?x; \ldots; Cm ?x \} \implies \text{TS} ?x \\\n    n+1) & \quad \text{THYP}((\exists x. C1 x \ldots Cm x \implies \text{TS} x) \implies (\forall x. C1 x \ldots Cm x \implies \text{TS} x))
    \end{align*}
    \]
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

test_spec:
  "x mem S \Rightarrow prog x S"
apply(gen_testcase)

Result:
1. prog ?x1 [?x1]
2. prog ?x2 [?x2,?b2]
3. ?a3\neq ?x3 \Rightarrow prog ?x3 [?a3,?x3]
4. THYP(\exists x. prog x [x] \Rightarrow prog x [x]

...
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

<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>dmz</td>
<td>tcp</td>
<td>993</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>
HOL model of a firewall policy

- A firewall makes a decision based on single packets.
  
  \textbf{types} \ (\alpha, \beta) \ \text{packet}  
  \quad = \text{id} \times (\alpha::\text{adr}) \ \text{src} \times (\alpha::\text{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, \ldots):
  
  \textbf{types} \ \alpha \mapsto \beta = \alpha \to \beta \ \text{decision}  
  
  \textbf{types} \ \alpha, \beta \ \text{Policy} = (\alpha, \beta) \ \text{packet} \to \text{unit}  

- Remark: for policies with network address translation:
  
  \textbf{types} \ \alpha, \beta \ \text{NAT Policy} = (\alpha, \beta) \ \text{packet} \to (\alpha, \beta) \ \text{packet set}  

- Policy combinators allow for defining policies:
  
  \textbf{definition}  
  
  \text{allow all from} :: (\alpha::\text{adr}) \ \text{net} \to (\alpha, \beta) \ \text{Policy} \ \text{where}  
  
  \text{allow all from src net} = \{ \text{pa. src pa} \ \sqsubseteq \text{src net} \} \ \downarrow_{\mathcal{A}_U}
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>dmz</td>
<td>tcp</td>
<td>993</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 ⊕
ablew_port udp 80 intranet internet ⊕

\( D_U \)

where \( D_U \) is the policy that denies all traffic
Testing stateless firewalls

The test specification:

\(\text{test_spec} \text{ test: } \text{"P x } \implies \text{FUT x = Policy x"} \)

- FUT: Placeholder for \textit{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):

\[
\text{FUT(1,((8,13,12,10),6,tcp),((172,168,2,1),80,tcp),data)= (deny())}
\]
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
  \[\Rightarrow\] However, many of these case splits are redundant
- Many combinations due to subnets
  \[\Rightarrow\] 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 $\oplus$ DenyPort dmz internet 21 $\oplus D_U$
  - AllowAll dmz internet $\oplus D_U$
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:

  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.
Outline

1 Motivation

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

3 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 Tuong, and Burkhart Wolff.
On the semantics of object-oriented data structures and path expressions.

Achim D. Brucker, Frank Rittinger, and Burkhart Wolff.
hol-z 2.0: A proof environment for ZSpecifications.
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.
ISSN 0001-5903.

Achim D. Brucker and Burkhart Wolff.
On theorem prover-based testing.
ISSN 0934-5043.