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

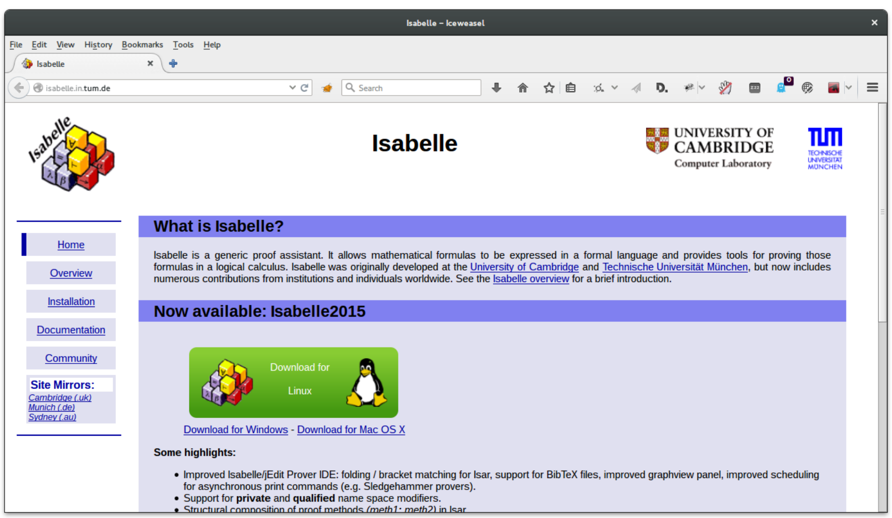
Proof Assistants and Related Tools - The PART Project
Technical University of Denmark, Kgs. Lyngby, Denmark
September 24, 2015

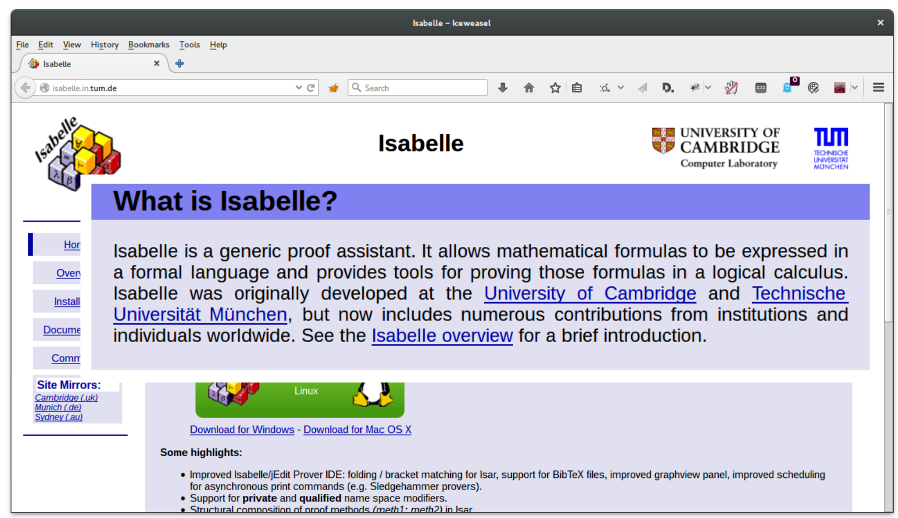
Isabelle: Not Only a Proof Assistant

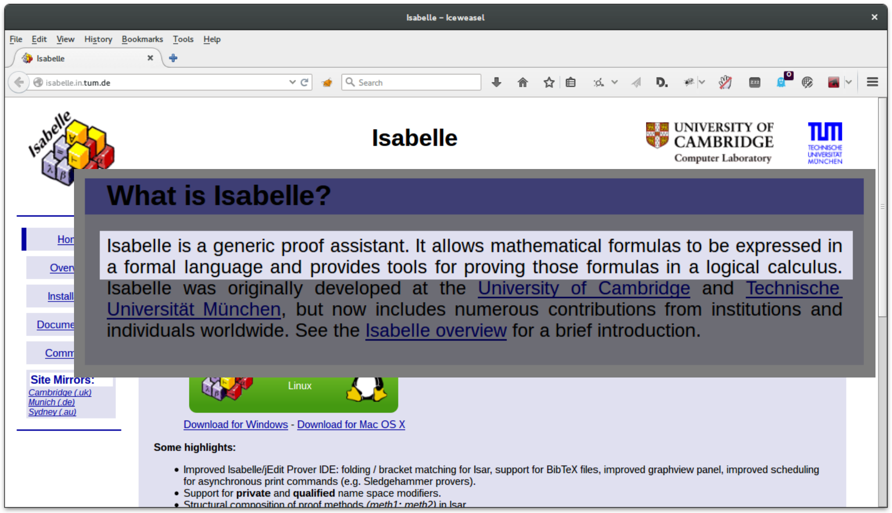
Abstract

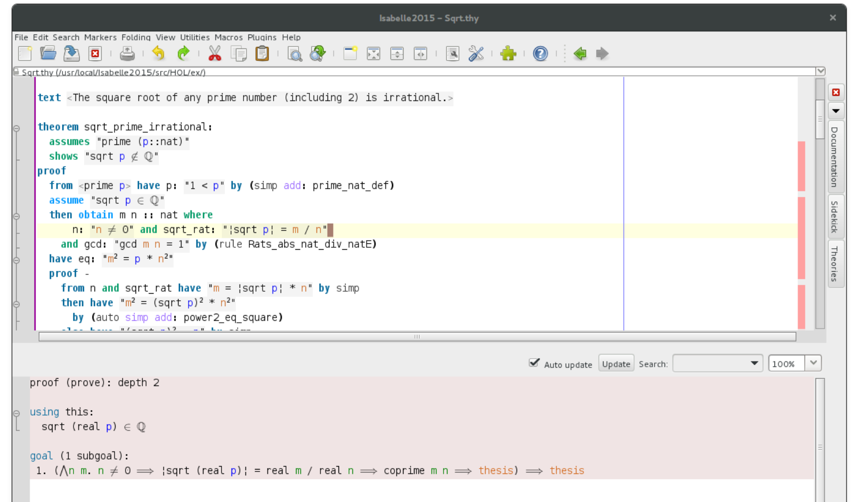
The Isabelle homepage describes Isabelle as "a generic proof assistant. It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus." While this, without doubts, what most users of Isabelle are using Isabelle for, there is much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

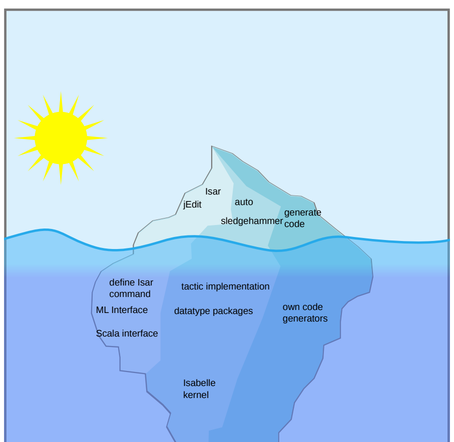








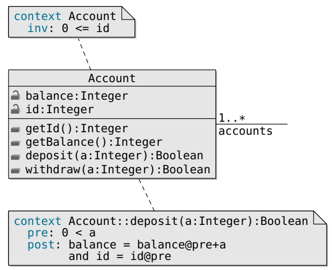
This is only the tip of the iceberg



Outline

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



Developing formals 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

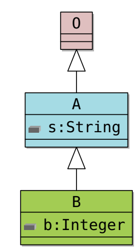
  $$e{:}T \quad \longrightarrow \quad e :: T$$

  (including subtyping)?
- **Challenge 2:**

  Can we support verification in a modular way (i.e., no replay of proof scripts after extending specifications)?
- **Challenge 3:**

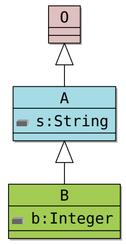
  Can we ensure consistency of our representation?

Representing class types



- The "extensible records" approach
  - We assume a common superclass (O).
  - A *tag type* guarantees uniquenessby (O_{tag} := classO).
  - 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 $\phi(x :: (\alpha\ \mathrm{B}))$ 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

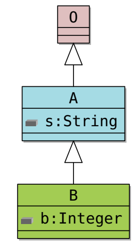
Representing class types



- The “extensible records” approach
  - We assume a common superclass (0).
  - A *tag type* guarantees uniquenessby ($\mathrm{O}_{\mathrm{tag}}$:= classO).
  - Construct class type as tuple along inheritance hierarchy:

    $\mathrm{B} :=$
- Advantages:
  - it allows for extending class types (inheritance),
  - subclasses are type instances of superclasses
  - ⇒ **it allows for modular proofs**, i.e.,
    a statement $\phi(x :: (\alpha\ \mathrm{B}))$ 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

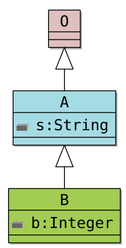
Representing class types



- The “extensible records” approach
  - We assume a common superclass (0).
  - A *tag type* guarantees uniquenessby ($\mathrm{O}_{\mathrm{tag}}$:= classO).
  - Construct class type as tuple along inheritance hierarchy:

    $\mathrm{B} := (\mathrm{O}_{\mathrm{tag}} \times \mathrm{oid})$
- Advantages:
  - it allows for extending class types (inheritance),
  - subclasses are type instances of superclasses
  - ⇒ **it allows for modular proofs**, i.e.,
    a statement $\phi(x :: (\alpha\ \mathrm{B}))$ 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

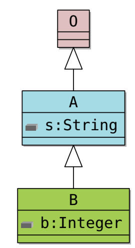
Representing class types



- The “extensible records” approach
  - We assume a common superclass (0).
  - A *tag type* guarantees uniquenessby ($\mathrm{O}_{\mathrm{tag}}$:= classO).
  - Construct class type as tuple along inheritance hierarchy:

    $\mathrm{B} := (\mathrm{O}_{\mathrm{tag}} \times \mathrm{oid}) \times \Big((\mathrm{A}_{\mathrm{tag}} \times \mathtt{String}) \qquad \Big)$
- Advantages:
  - it allows for extending class types (inheritance),
  - subclasses are type instances of superclasses
  - ⇒ **it allows for modular proofs**, i.e.,
    a statement $\phi(x :: (\alpha\ \mathrm{B}))$ 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

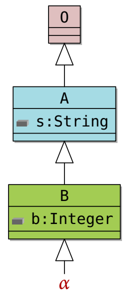
Representing class types



- The “extensible records” approach
  - We assume a common superclass (0).
  - A *tag type* guarantees uniquenessby ($\mathrm{O}_{\mathrm{tag}}$:= classO).
  - Construct class type as tuple along inheritance hierarchy:

    $\mathrm{B} := (\mathrm{O}_{\mathrm{tag}} \times \mathrm{oid}) \times \Big((\mathrm{A}_{\mathrm{tag}} \times \mathtt{String}) \times \big((\mathrm{B}_{\mathrm{tag}} \times \mathtt{Integer}) \quad \big)\Big)$
- Advantages:
  - it allows for extending class types (inheritance),
  - subclasses are type instances of superclasses
  - ⇒ **it allows for modular proofs**, i.e.,
    a statement $\phi(x :: (\alpha\ \mathrm{B}))$ 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 uniquenessby ($\mathrm{O}_{\mathrm{tag}} := \mathrm{classO}$).
  - Construct class type as tuple along inheritance hierarchy:

$$\alpha\ \mathrm{B} := (\mathrm{O}_{\mathrm{tag}} \times \mathrm{oid}) \times \Big((\mathrm{A}_{\mathrm{tag}} \times \mathtt{String}) \times \big((\mathrm{B}_{\mathrm{tag}} \times \mathtt{Integer}) \times \alpha\big)\Big)$$

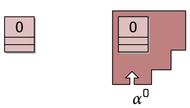
- Advantages:
  - it allows for extending class types (inheritance),
  - subclasses are type instances of superclasses
  - ⇒ **it allows for modular proofs**, i.e., a statement $\phi(x :: (\alpha\ \mathrm{B}))$ proven for class B is still valid after extending class B.
- However, it has a major disadvantage: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

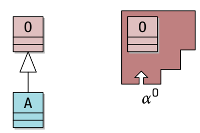
- supports modular proofs with arbitrary extensions
- provides a formalization of a extensible typed object store

An extensible object store



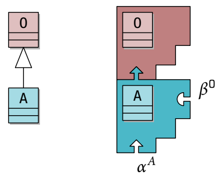
$$\mathcal{U}^0_{(\alpha^0)} = O \times \alpha^0_\perp$$

An extensible object store



$$\mathcal{U}^0_{(\alpha^0)} = O \times \alpha^0_\perp$$

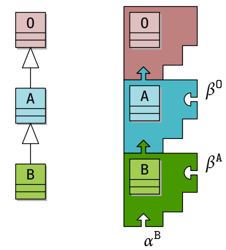
An extensible object store



$$\mathcal{U}^0_{(\alpha^0)} = O \times \alpha^0_\perp$$

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

An extensible object store

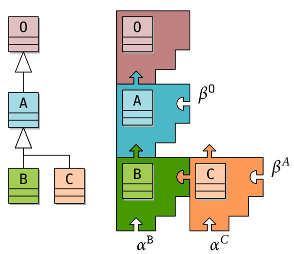


$$\mathcal{U}^0_{(\alpha^0)} = O \times \alpha^0_\perp$$

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

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

An extensible object store

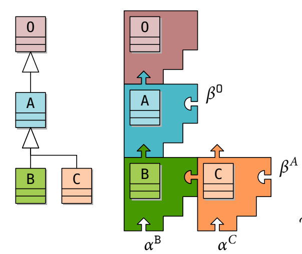


$$\mathcal{U}^0_{(\alpha^0)} = O \times \alpha^0_\perp$$

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

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

An extensible object store



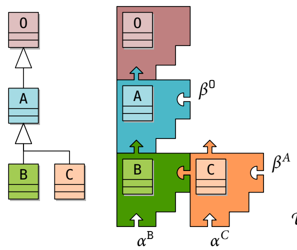
$$\mathcal{U}^0_{(\alpha^0)} = O \times \alpha^0_\perp$$

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

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

$$\mathcal{U}^3_{(\alpha^B,\alpha^C,\beta^0,\beta^A)} = O \times (A \times (B \times \alpha^B_\perp + (C \times \alpha^C_\perp + \beta^A))_\perp + \beta^0)_\perp$$

An extensible object store



$$\mathcal{U}^0_{(\alpha^0)} = O \times \alpha^0_\perp$$

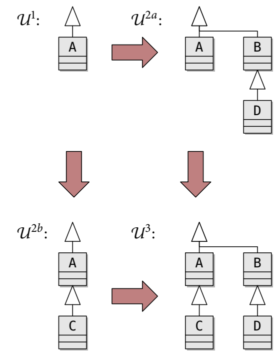
$$\mathcal{U}^1_{(\alpha^A,\beta^0)} = O \times (A \times \alpha^A_\perp + \beta^0)_\perp$$

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

$$\mathcal{U}^3_{(\alpha^B,\alpha^C,\beta^0,\beta^A)} = O \times (A \times (B \times \alpha^B_\perp + (C \times \alpha^C_\perp + \beta^A))_\perp + \beta^0)_\perp$$

$\mathrm{U}^3_{(\alpha^B,\alpha^C,\beta^0,\beta^A)} \prec \mathrm{U}^2_{(\alpha^B,\beta^0,\beta^A)} \prec \mathrm{U}^1_{(\alpha^A,\beta^0)} \prec \mathrm{U}^0_{(\alpha^0)}$

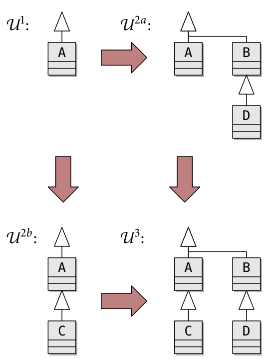
Merging universes



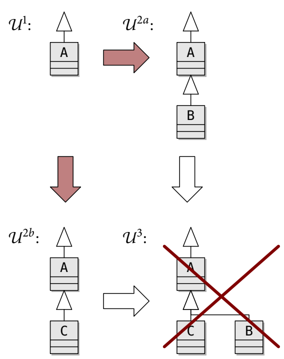
Non-conflicting Merges

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

Merging universes



Non-conflicting Merges



Conflicting Merges

Achim D. Brucker Isabelle: Not *Only* a Proof Assistant September 24, 2015 13

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

Operations accessing the object store

- injections

  $\mathrm{mk}_\mathrm{O}\, o = \mathrm{Inl}\, o$ with type $\alpha^O\ 0 \to \mathrm{U}^0_{\alpha^O}$

- projections

  $\mathrm{get}_\mathrm{O}\, u = u$ with type $\mathrm{U}^0_{\alpha^O} \to \alpha^O\ 0$

- type casts

  $\mathrm{A}_{[\mathrm{O}]} = \mathrm{get}_\mathrm{O} \circ \mathrm{mk}_\mathrm{A}$ with type $\alpha^A\ \mathrm{A} \to (A \times \alpha^A_\perp + \beta^O)\ 0$

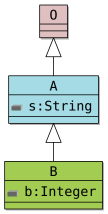
  $\mathrm{O}_{[\mathrm{A}]} = \mathrm{get}_\mathrm{A} \circ \mathrm{mk}_\mathrm{O}$ with type $(A \times \alpha^A_\perp + \beta^O)\ 0 \to \alpha^A\ \mathrm{A}$

- ...

All definitions are generated automatically

“Checking” subtyping

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



- subclasses are of the superclasses kind:

$$\frac{\text{isType}_{\text{B}}\ \mathit{self}}{\text{isKind}_{\text{A}}\ \mathit{self}}$$

- “re-casting”:

$$\frac{\text{isType}_{\text{B}}\ \mathit{self}}{\mathit{self}_{[\text{A}][\text{B}]} \neq \bot \land \text{isType}_{\text{B}}\ (\mathit{self}_{[\text{A}][\text{B}][\text{A}]})}$$

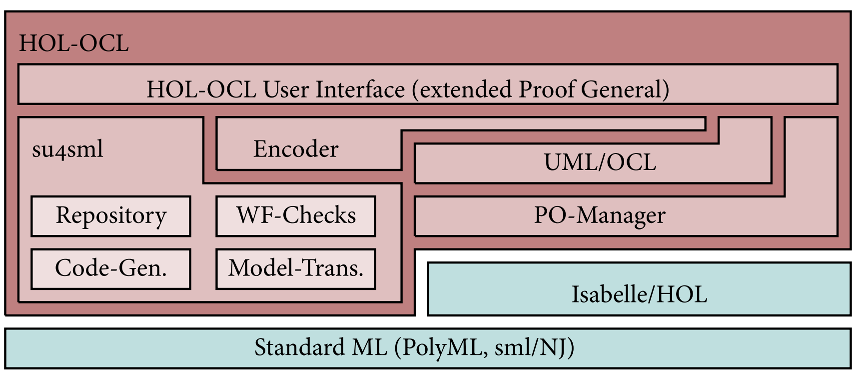
- monotonicity of invariants, . . .

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:
  `http://www.brucker.ch/projects/hol-ocl/`.

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:

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

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

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

- starting a proof for a generated proof obligation:

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

- generating code:

```
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.

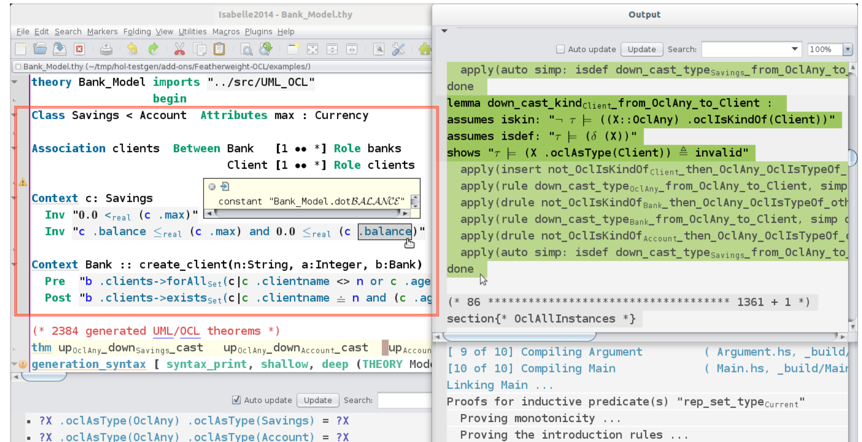
Proof obligation generator

A framework for proof obligation generation:

- Generates proof obligation in OCL plus minimal meta-language.
- Only minimal meta-language necessary:
  - Validity: $\models \_$, $\_ \models \_$
  - Meta level quantifiers: $\exists \_ .\ \_$, $\exists \_ .\ \_$
  - Meta level logical connectives: $\_ \vee \_$, $\_ \wedge \_$, $\neg \_$
- 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

HOL-OCL 2.0 (Featherweight OCL)



Outline

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

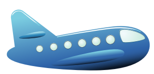
- Execution of test cases
- Can show failures on real system
- Only shows failures for the parts of the system
- Requires less skills in Formal Methods
- **In industry:** widely used (often $>$ 40% of dev. effort)

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 you next travel:
  - Aircraft A:



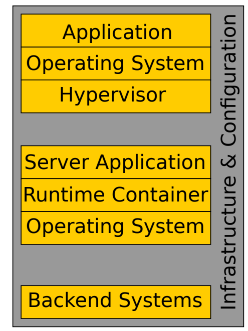
  - Fully formally verified
  - Total number of flights: 0

  - 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



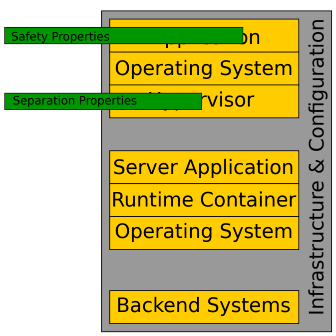
Observation:
- Both methods have their unique advantages

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



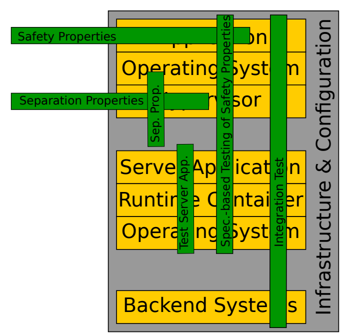
Observation:
- Both methods have their unique advantages

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

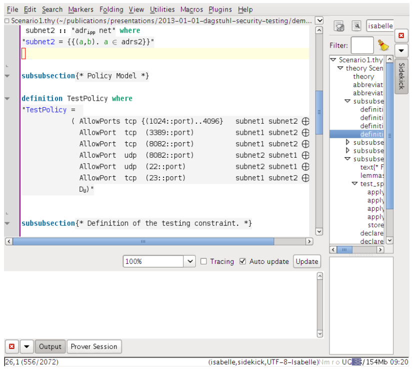


Observation:
- Both methods have their unique advantages

Recommendation:
- Use a combination of verification and testing

Our Vision:
- An integrated approach for test and verification

Implementing our vision in Isabelle: HOL-TestGen



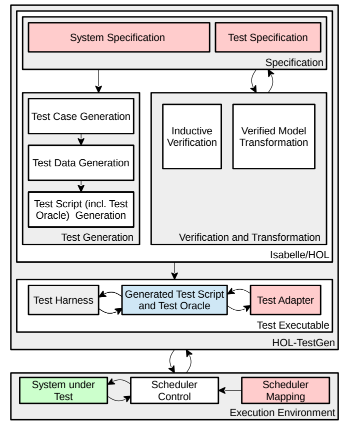
An **interactive** model-based test tool

- 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:

`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:

$$\mathrm{THYP} : \mathrm{bool} \Rightarrow \mathrm{bool}$$
$$\mathrm{THYP}(x) \equiv x$$

- Two test hypotheses are common:
  - **Regularity hypothesis:** captures infinite data structures (splits), e.g., for lists

$$\frac{\begin{array}{c}[x=[]]\\ \vdots \\ P\end{array} \quad \bigwedge a \quad \begin{array}{c}[x=[a]]\\ \vdots \\ P\end{array} \quad \bigwedge a\ b\ h \quad \begin{array}{c}[x=[a,b]]\\ \vdots \\ P\end{array} \quad \mathrm{THYP}(\forall x. k < \mathrm{size}\, x \longrightarrow P\, x)}{P}$$

  - **Uniformity hypothesis:** captures test data selection
    "Once a system under test behaves correct for one test case, it behaves correct for all test cases"

n) $[\![\mathrm{C1}\ ?x; \ldots; \mathrm{Cm}\ ?x]\!] \Longrightarrow \mathrm{TS}\ ?x$

n+1) $\mathrm{THYP}((\exists x.\ \mathrm{C1}\ x \ldots \mathrm{Cm}\ x \longrightarrow \mathrm{TS}\ x) \longrightarrow (\forall x.\ \mathrm{C1}\ x \ldots \mathrm{Cm}\ x \longrightarrow \mathrm{TS}\ x))$

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

test_spec:
  "x mem S ⟹ prog x S"
apply(gen_testcase)
```

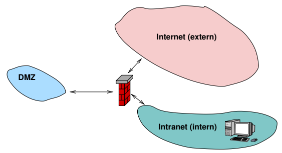
Result:

1. prog ?x1 [?x1]
2. prog ?x2 [?x2,?b2]
3. ?a3≠?x3 ⟹ prog ?x3 [?a3,?x3]
4. THYP(∃x.prog x [x] ⟶ prog x [x]

   ...

7. THYP(∀ S. 3 ≤ size S ⟶ x mem S ⟶ prog x S)

Use case: testing firewall policies



source	destination	protocol	port	action
Internet	dmz	udp	25	allow
Internet	dmz	tcp	80	allow
dmz	intranet	tcp	25	allow
intranet	dmz	tcp	993	allow
intranet	Internet	udp	80	allow
any	any	any	any	deny

- **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

HOL model of a firewall policy

- A firewall makes a decision based on single packets.

  types (α,β) packet
  $= \text{id} \times (\alpha\text{::adr})\ \text{src} \times (\alpha\text{::adr})\ \text{dest} \times \beta \text{content}$

  Different address and content representations are possible.
- A policy is a mapping from packets to decisions (allow, deny, . . .):

  types $\alpha \mapsto \beta = \alpha \rightharpoonup \beta\text{decision}$
  types (α,β) Policy $= (\alpha,\beta)$ packet $\mapsto$ unit
- Remark: for policies with network address translation:

  types (α,β) NAT_Policy $= (\alpha,\beta)$ packet $\mapsto (\alpha,\beta)$ packet set
- Policy combinators allow for defining policies:

  definition
  allow_all_from :: $(\alpha\text{::adr})$ net $\Rightarrow (\alpha,\beta)$ Policy **where**
  allow_all_from src_net = {pa. src pa $\sqsubset$ src_net} $\lhd A_U$

The policy

source	destination	protocol	port	action
Internet	dmz	udp	25	allow
Internet	dmz	tcp	80	allow
dmz	intranet	tcp	25	allow
intranet	dmz	tcp	993	allow
intranet	Internet	udp	80	allow
any	any	any	any	deny

definition TestPolicy **where**
TestPolicy = allow_port udp 25 internet dmz $\oplus$
allow_port tcp 80 internet dmz $\oplus$
allow_port tcp 25 dmz intranet $\oplus$
allow_port tcp 993 intranet dmz $\oplus$
allow_port udp 80 intranet internet $\oplus$
D_U

where D_U is the policy that denies all traffic

Testing stateless firewalls

- The test specification:

  test_spec test: “P x $\Longrightarrow$ FUT x = Policy x”
  - 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):
  FUT(1,((8,13,12,10),6,tcp),((172,168,2,1),80,tcp),data)= $\lfloor$(deny()$\rfloor$

Problems with the direct approach

- The direct approach **does not scale**:

	R1	R2	R3	R4
Networks	3	3	4	3
Rules	12	9	13	13
TC Generation Time (sec)	26382	187	59364	1388
Test Cases	1368	264	1544	470

- **Reason**:
  - Large cascades of case distinctions over input and output
    $\Longrightarrow$ However, many of these case splits are redundant
  - Many combinations due to subnets
    $\Longrightarrow$ 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

```
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

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

The normalization of policies **decreases**
- the number of test cases and
- the required test case generation time

by several orders of magnitude.

Outline

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.
In *Third International Conference on Software Testing, Verification, and Validation (ICST)*, pages 345–354. IEEE Computer Society, 2010.
http://www.brucker.ch/bibliography/abstract/brucker.ea-firewall-2010.

Achim D. Brucker, Lukas Brügger, and Burkhart Wolff.
HOL-TestGen/FW: An environment for specification-based firewall conformance testing.
In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *International Colloquium on Theoretical Aspects of Computing (ICTAC)*, number 8049 in Lecture Notes in Computer Science, pages 112–121. Springer-Verlag, 2013.
http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-testgen-fw-2013.

Achim D. Brucker, Lukas Brügger, and Burkhart Wolff.
Formal firewall conformance testing: An application of test and proof techniques.
Software Testing, Verification & Reliability (STVR), 25(1):34–71, 2015.
http://www.brucker.ch/bibliography/abstract/brucker.ea-formal-fw-testing-2014.

Achim D. Brucker, Delphine Longuet, Frédéric Tuong, and Burkhart Wolff.
On the semantics of object-oriented data structures and path expressions.
In Jordi Cabot, Martin Gogolla, István Ráth, and Edward D. Willink, editors, *Proceedings of the MoDELs 2013 OCL Workshop (OCL 2013)*, volume 1092 of *CEUR Workshop Proceedings*, pages 23–32. ceur-ws.org, 2013.
http://www.brucker.ch/bibliography/abstract/brucker.ea-path-expressions-2013.

Achim D. Brucker, Frank Rittinger, and Burkhart Wolff.
hol-z 2.0: A proof environment for Z-specifications.
Journal of Universal Computer Science, 9(2):152–172, February 2003.
http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-z-2003.

Related Publications II

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.
http://www.brucker.ch/bibliography/abstract/brucker.ea-hol-ocl-2008.

Achim D. Brucker and Burkhart Wolff.
Extensible universes for object-oriented data models.
In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming*, number 5142 in Lecture Notes in Computer Science, pages 438–462. Springer-Verlag, 2008.
http://www.brucker.ch/bibliography/abstract/brucker.ea-extensible-2008.

Achim D. Brucker and Burkhart Wolff.
Semantics, calculi, and analysis for object-oriented specifications.
Acta Informatica, 46(4):255–284, July 2009.
ISSN 0001-5903.
http://www.brucker.ch/bibliography/abstract/brucker.ea-semantics-2009.

Achim D. Brucker and Burkhart Wolff.
On theorem prover-based testing.
Formal Aspects of Computing, 25(5):683–721, 2013.
ISSN 0934-5043.
http://www.brucker.ch/bibliography/abstract/brucker.ea-theorem-prover-2012.