Encoding Object-oriented Datatypes in HOL: Extensible Records Revisited

The HOL-OCL Expierence

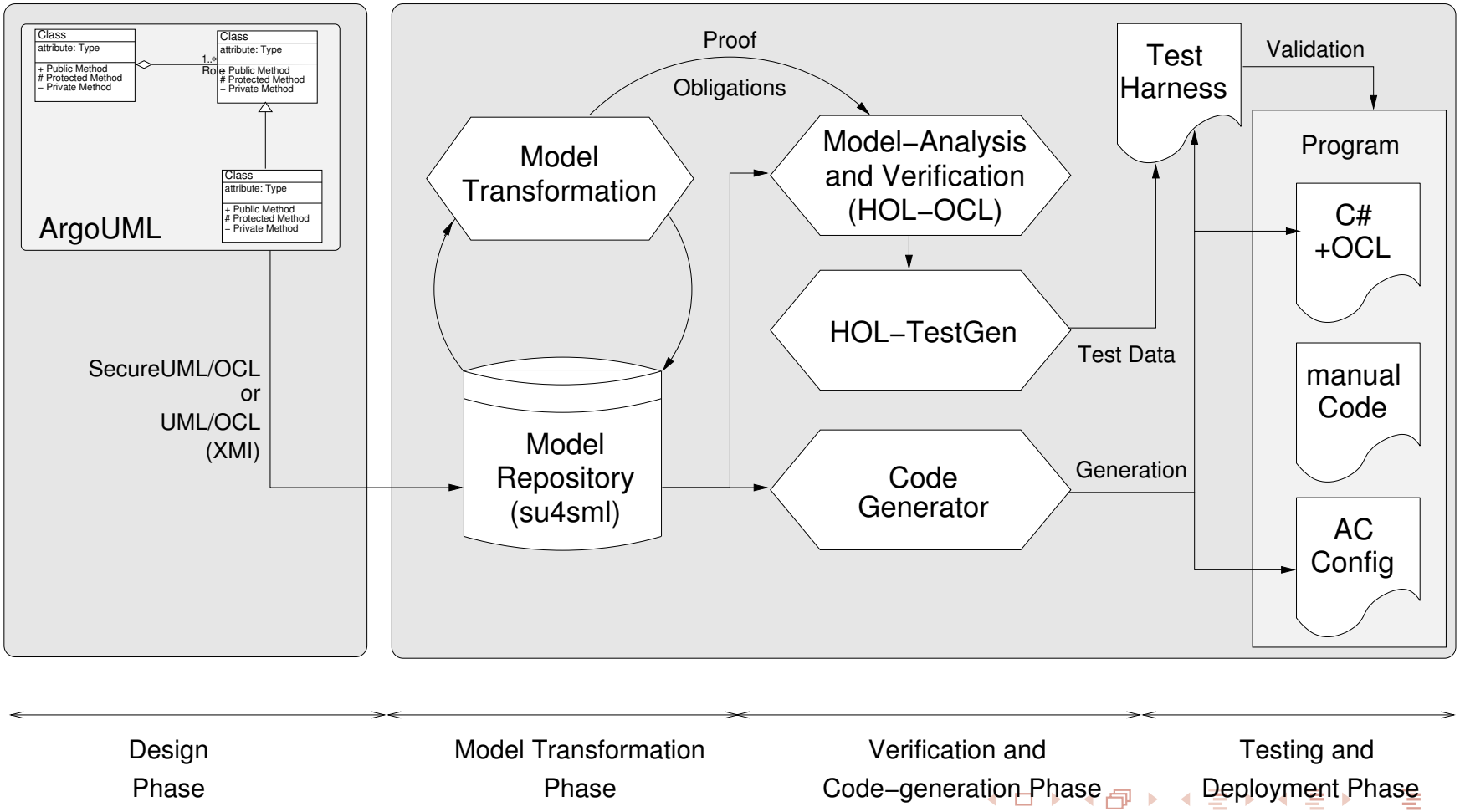
Achim D. Brucker
achim@brucker.ch http://www.brucker.ch/

Isabelle Developers Workshop (IDW 2010)
Cambridge, UK, 17th June 2010

Outline

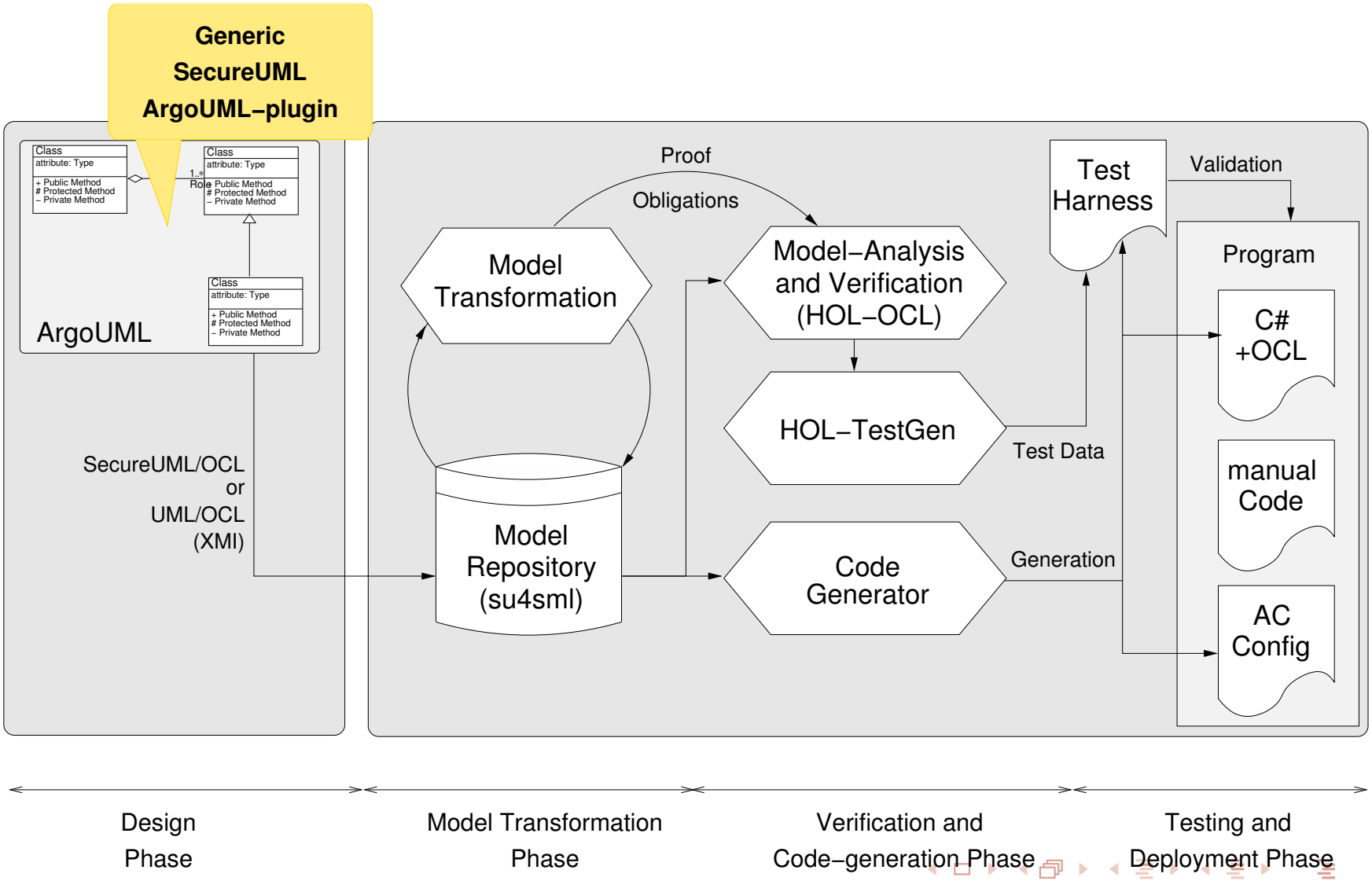
Our Vision:

Tool Supported Formal Methods for (Model-driven) Software Development



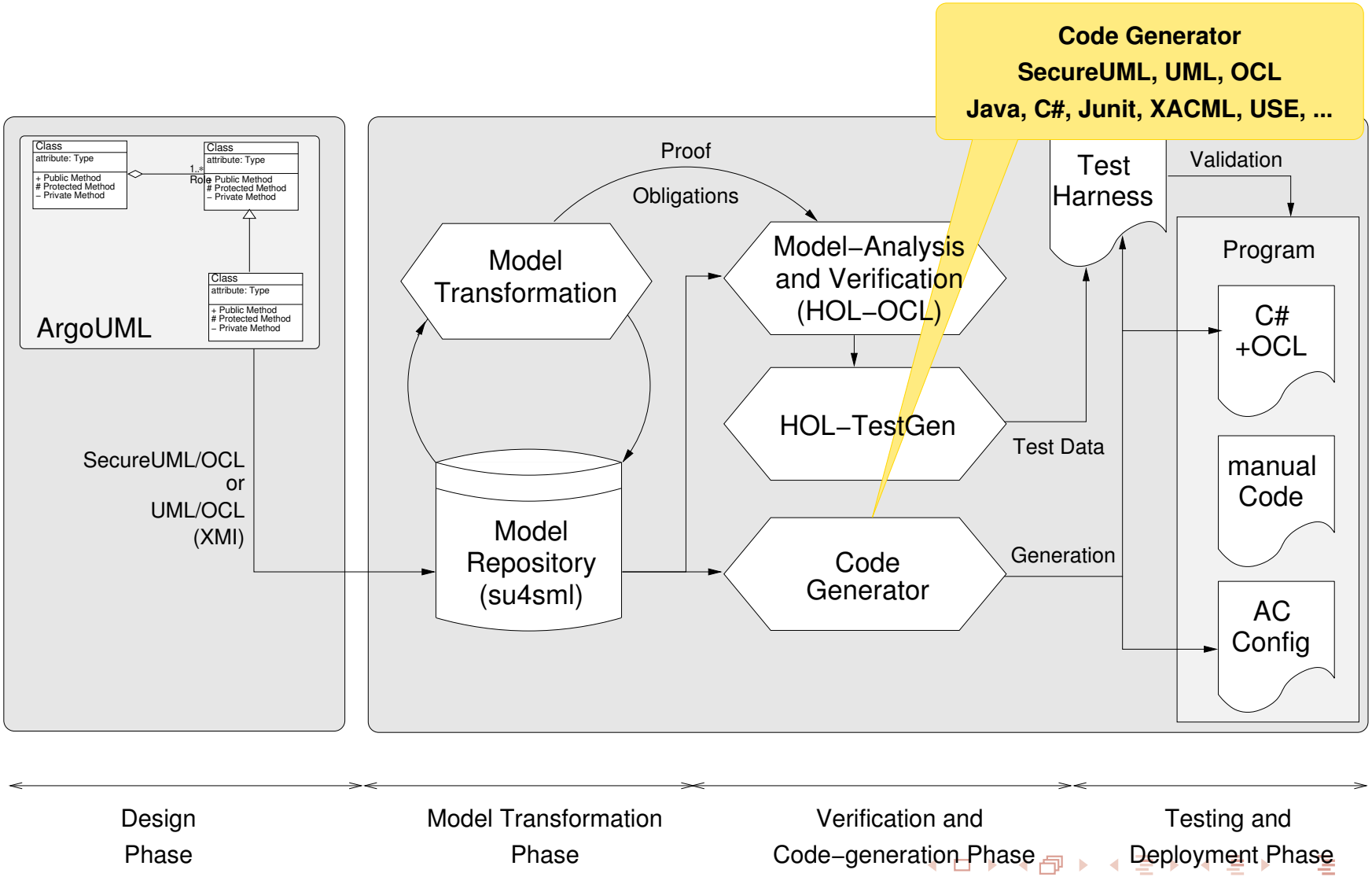
Our Vision:

Tool Supported Formal Methods for (Model-driven) Software Development



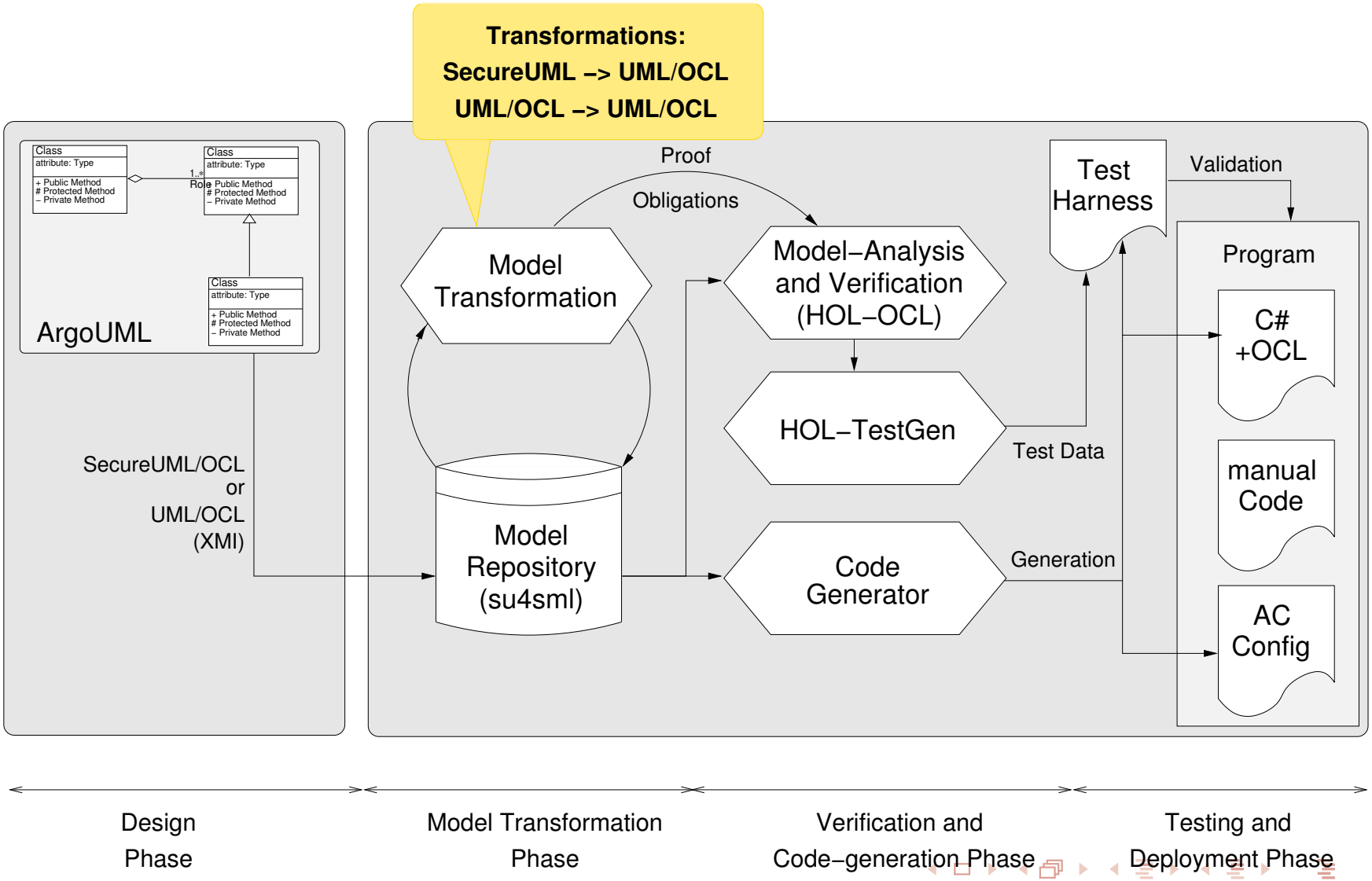
Our Vision:

Tool Supported Formal Methods for (Model-driven) Software Development



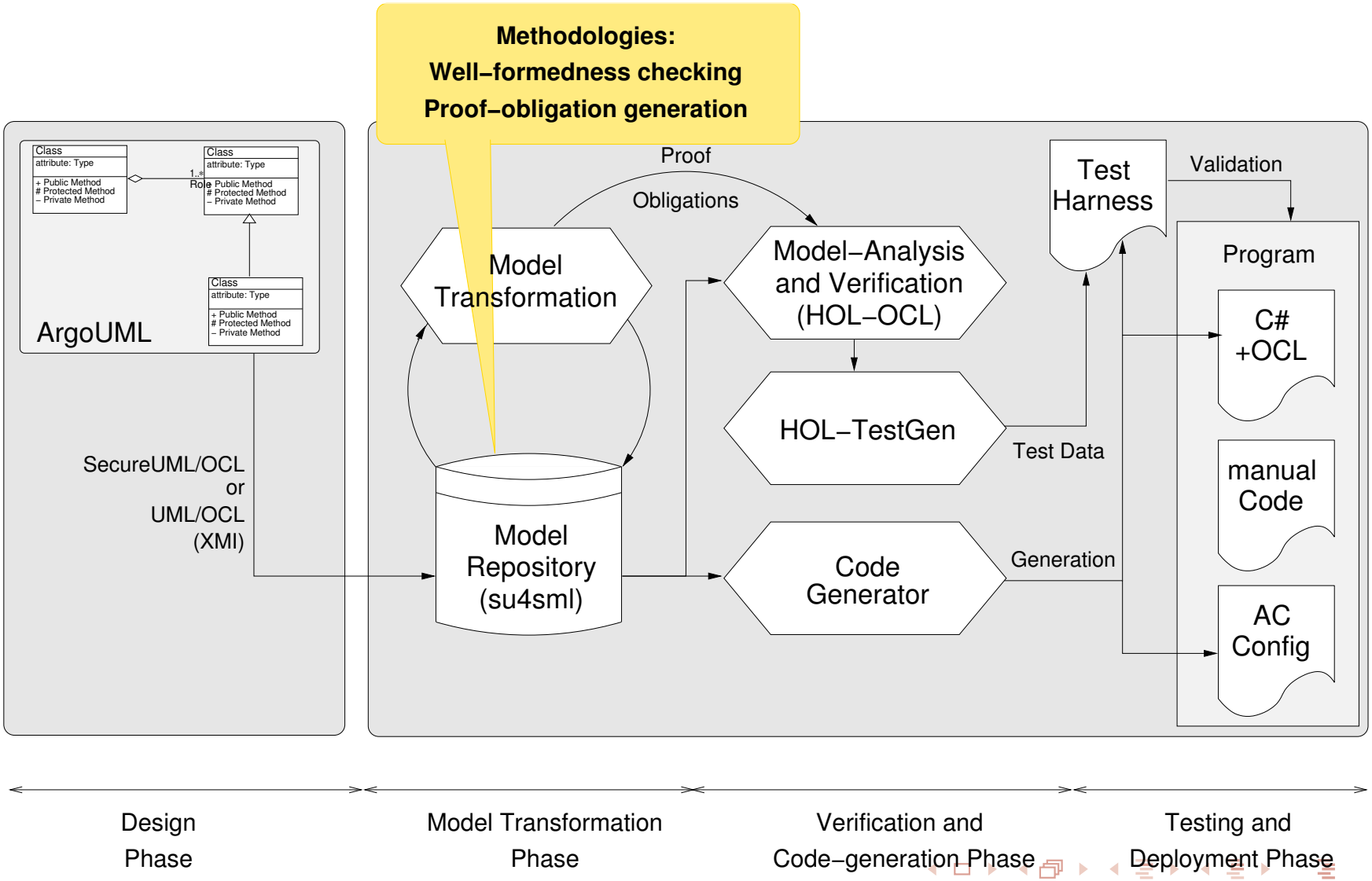
Our Vision:

Tool Supported Formal Methods for (Model-driven) Software Development



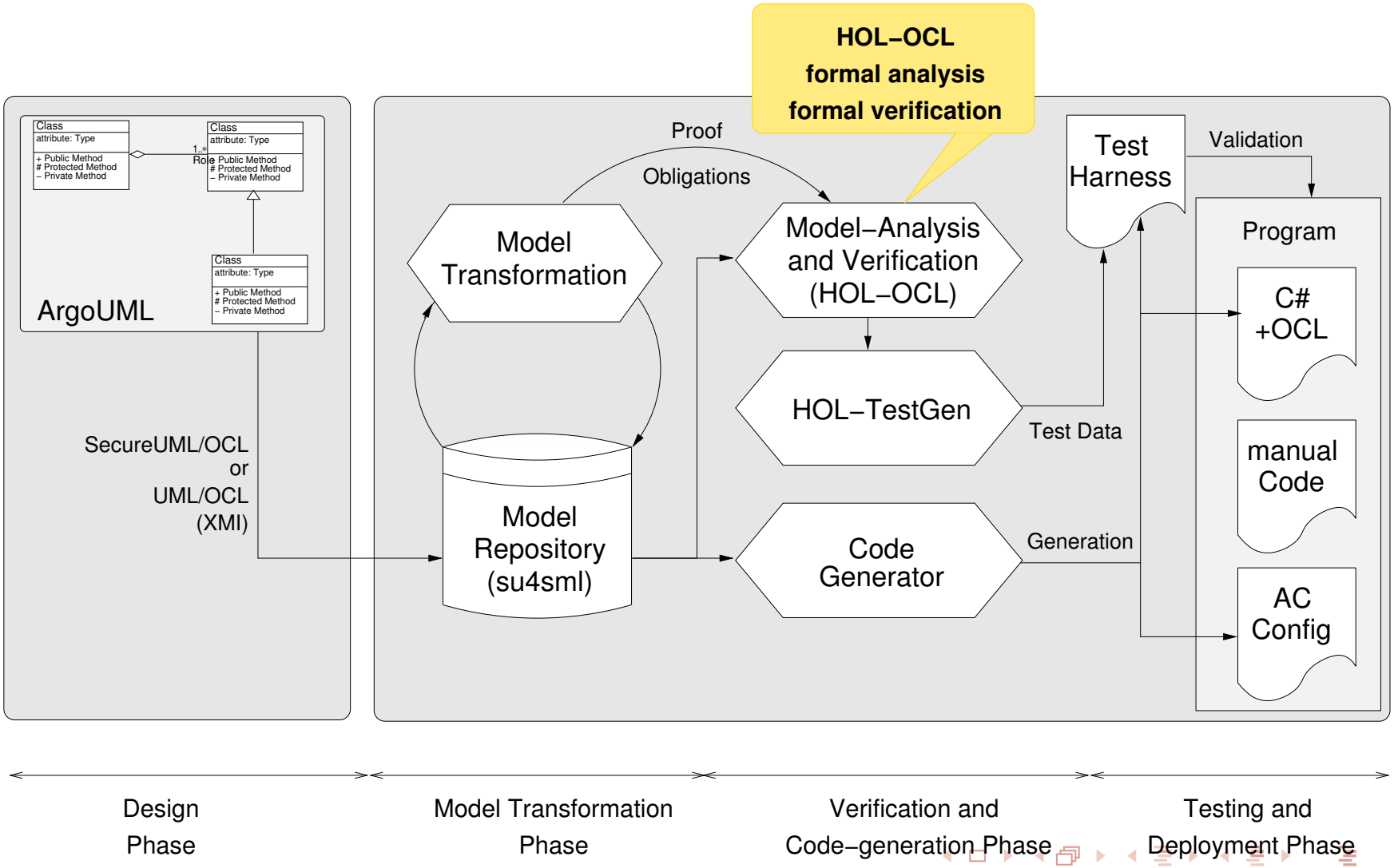
Our Vision:

Tool Supported Formal Methods for (Model-driven) Software Development



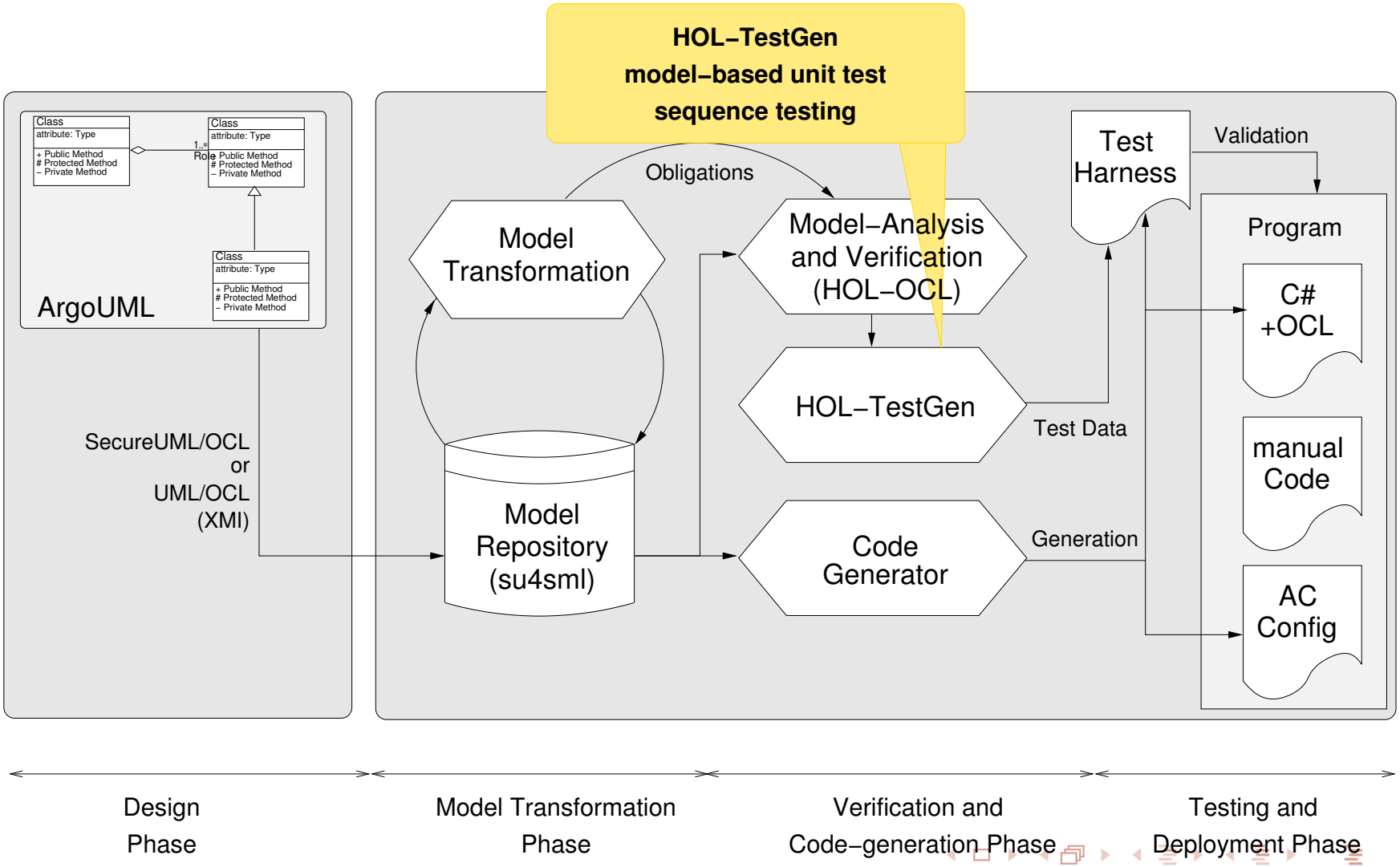
Our Vision:

Tool Supported Formal Methods for (Model-driven) Software Development



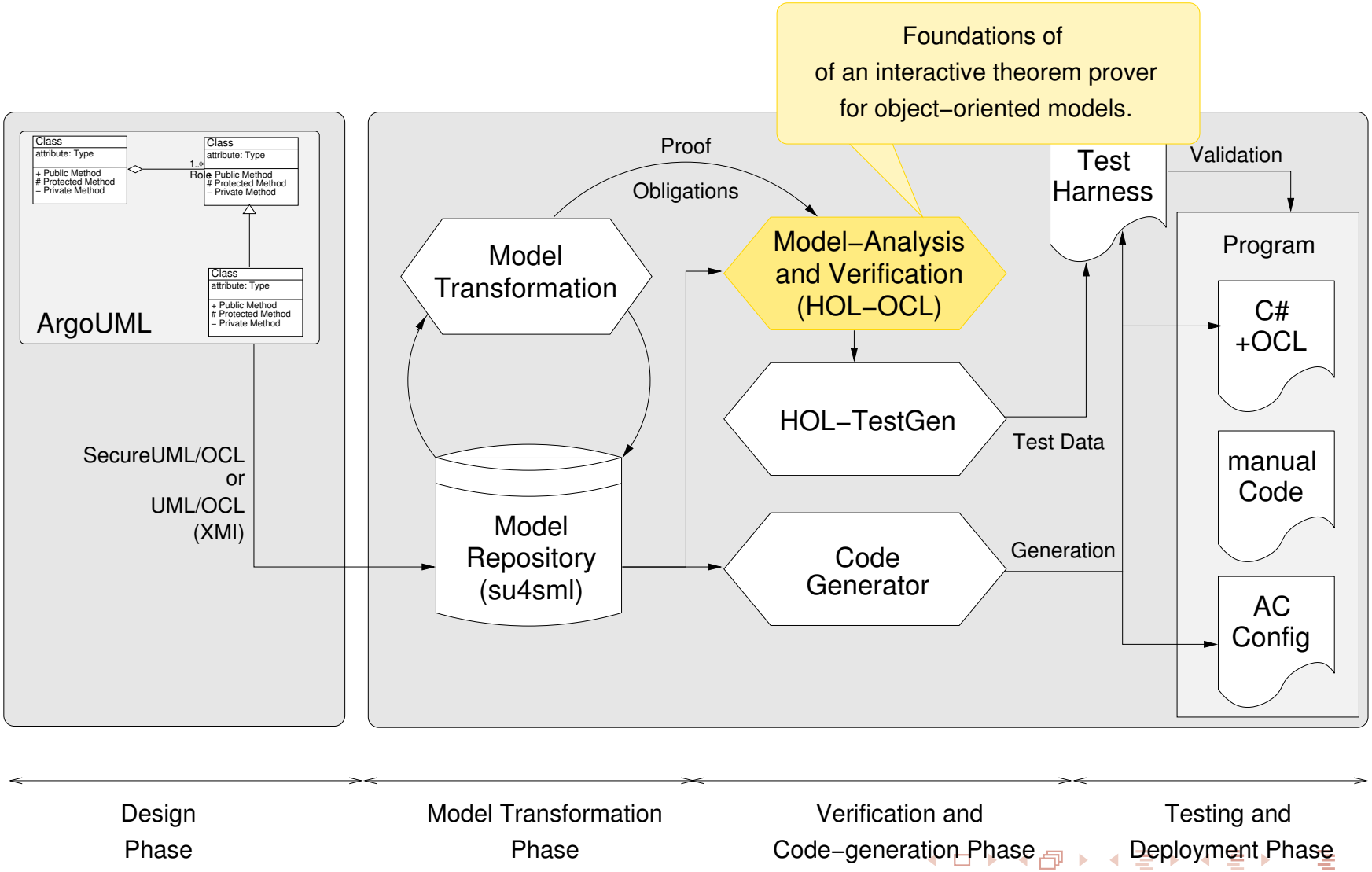
Our Vision:

Tool Supported Formal Methods for (Model-driven) Software Development



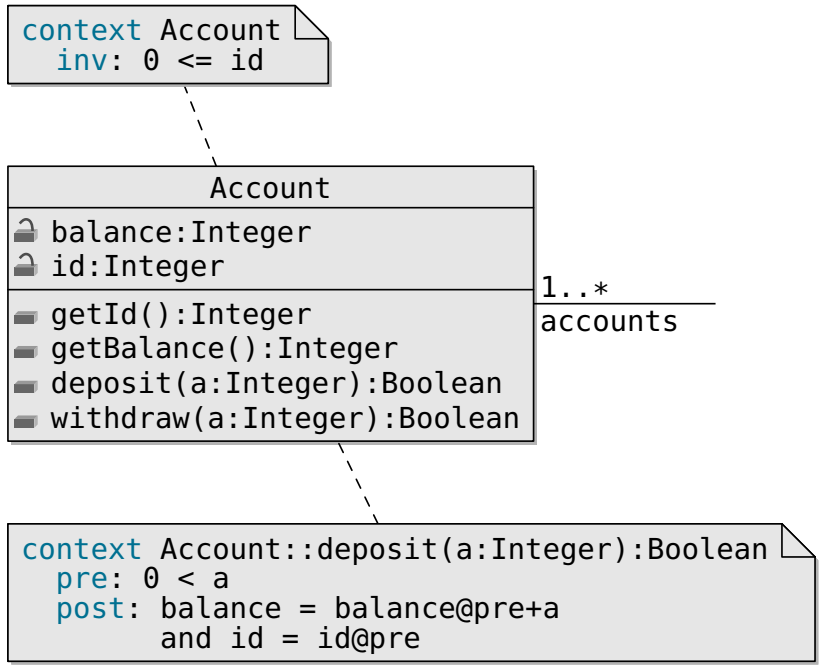
Our Vision:

Tool Supported Formal Methods for (Model-driven) Software Development



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:**

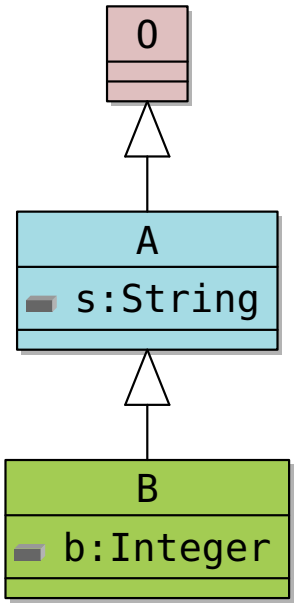
  Can we support verification in a modular way 1(i. e., no replay of proof scripts after extending specifications)?

- **Challenge 3:**

  Can we ensure consistency of our representation?

Representing Class Types

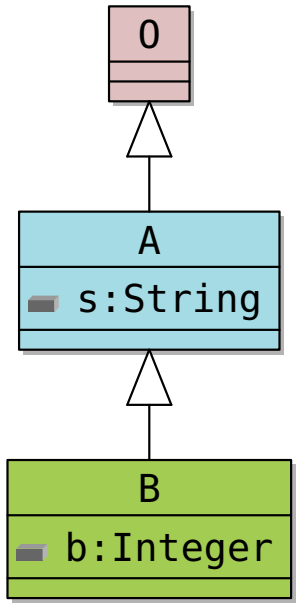
- The "extensible records" approach
  - We assume a common superclass (O).
  - A *tag type* guarantees uniquenessby (O_{tag} := classO).
  - Construct class type as tuple along inheritance hierarchy:



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

  $\Rightarrow$ **it allows for modular proofs**, i. e.,
  a statement $\phi(x :: (\alpha\ \mathtt{B}))$ 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



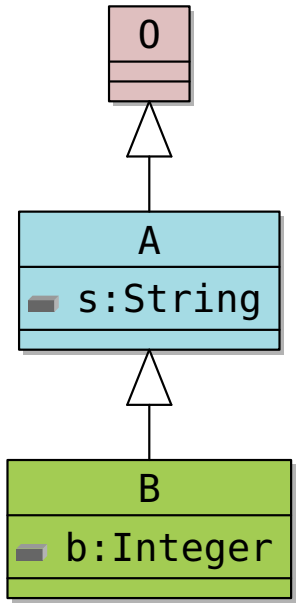
- The "extensible records" approach
  - We assume a common superclass (O).
  - A *tag type* guarantees uniquenessby (O_{tag} := classO).
  - Construct class type as tuple along inheritance hierarchy:

B :=

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

  $\Rightarrow$ **it allows for modular proofs**, i. e.,
  a statement $\phi(x :: (\alpha\ \mathtt{B}))$ 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 uniquenessby (O_{tag} := classO).
  - Construct class type as tuple along inheritance hierarchy:

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

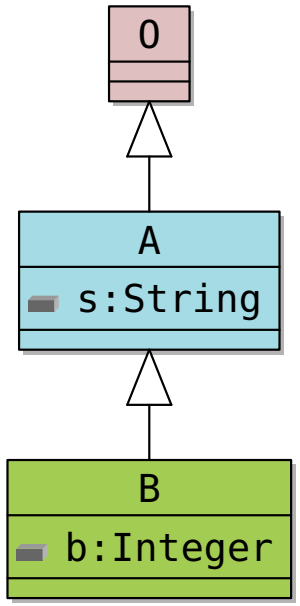
- Advantages:
  - it allows for extending class types (inheritance),
  - subclasses are type instances of superclasses

  $\Rightarrow$ **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 uniquenessby (O_{tag} := classO).
  - Construct class type as tuple along inheritance hierarchy:

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



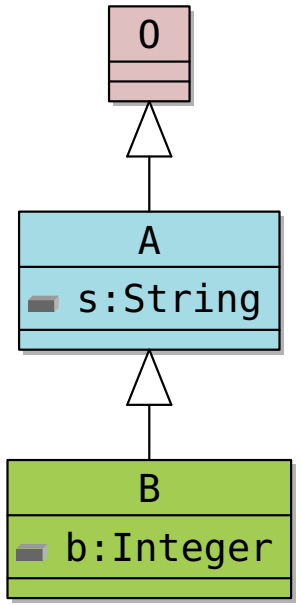
- Advantages:
  - it allows for extending class types (inheritance),
  - subclasses are type instances of superclasses

  $\Rightarrow$ **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 uniquenessby (O_{tag} := classO).
  - Construct class type as tuple along inheritance hierarchy:

$$B := (O_{tag} \times \text{oid}) \times \Big((A_{tag} \times \texttt{String}) \times \big((B_{tag} \times \texttt{Integer}) \quad \big)\Big)$$



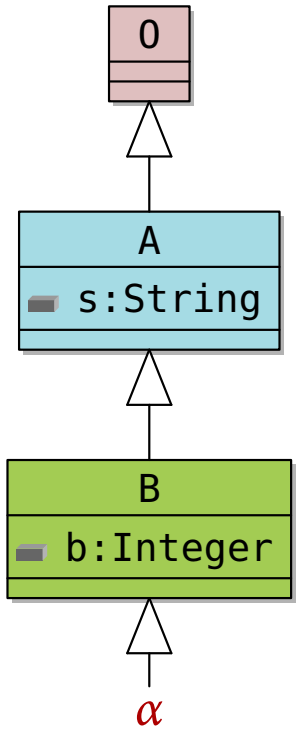
- Advantages:
  - it allows for extending class types (inheritance),
  - subclasses are type instances of superclasses

  $\Rightarrow$ **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 uniquenessby (O_{tag} := classO).
  - Construct class type as tuple along inheritance hierarchy:

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



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

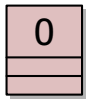
  $\Rightarrow$ **it allows for modular proofs**, i. e.,
  a statement $\phi(x :: (\alpha\ \mathrm{B}))$ proven for class B is still valid after extending class B.
- However, it has a major disadvantage:
  - modular proofs are only supported for **one** extension per class

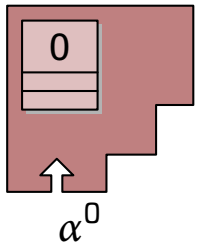
Idea: A General Universe Type

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

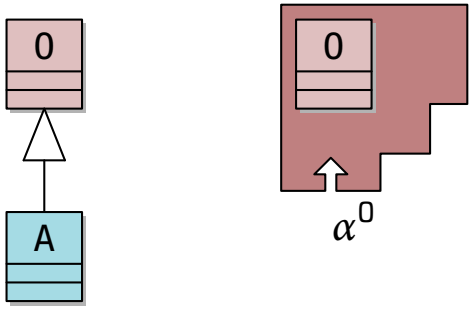
An Extensible Object Store





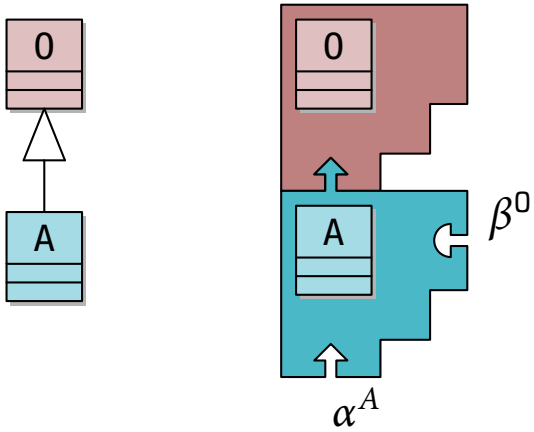
$$\mathcal{U}^0_{(\alpha^0)} = O \times \alpha^0_\perp$$

An Extensible Object Store



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

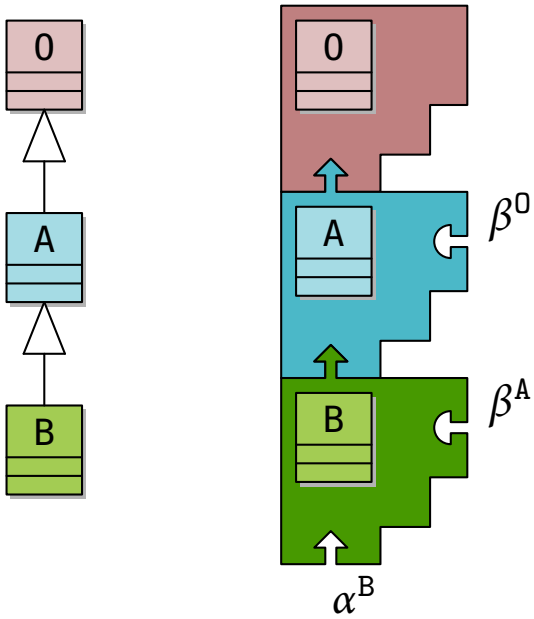
An Extensible Object Store



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

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

An Extensible Object Store

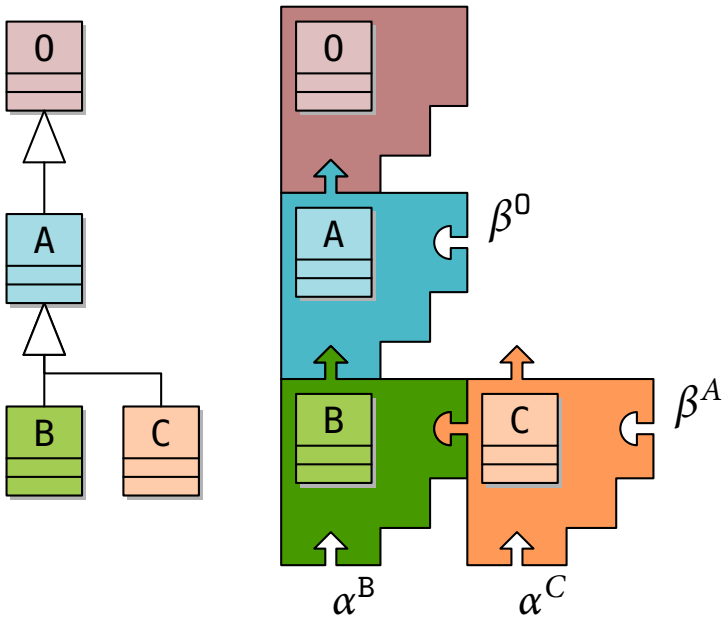


$$\mathcal{U}^0_{(\alpha^0)} = O \times \alpha^0_\perp$$

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

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

An Extensible Object Store

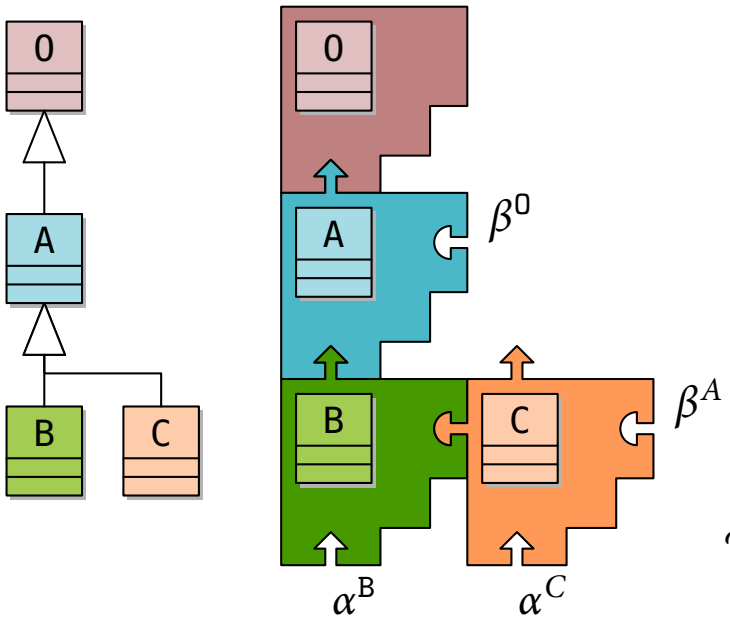


$$\mathcal{U}^0_{(\alpha^0)} = O \times \alpha^0_\perp$$

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

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

An Extensible Object Store



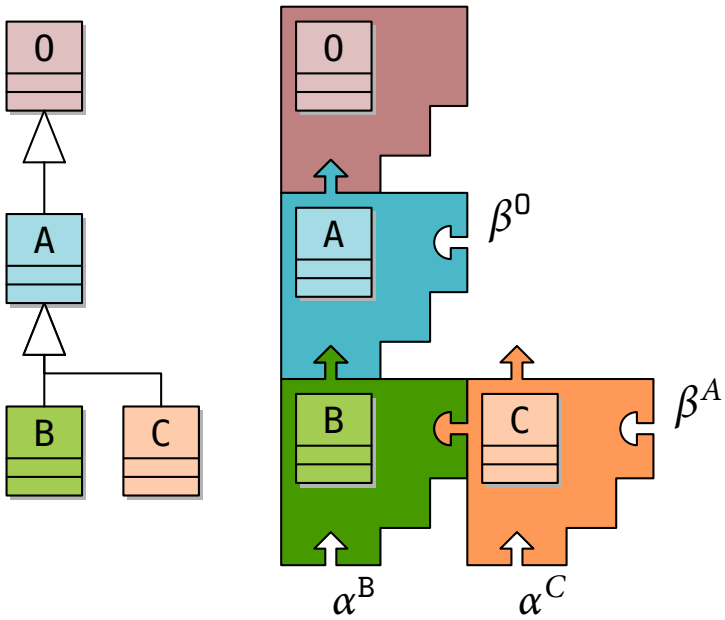
$$\mathcal{U}^0_{(\alpha^0)} = O \times \alpha^0_\perp$$

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

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

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

An Extensible Object Store



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

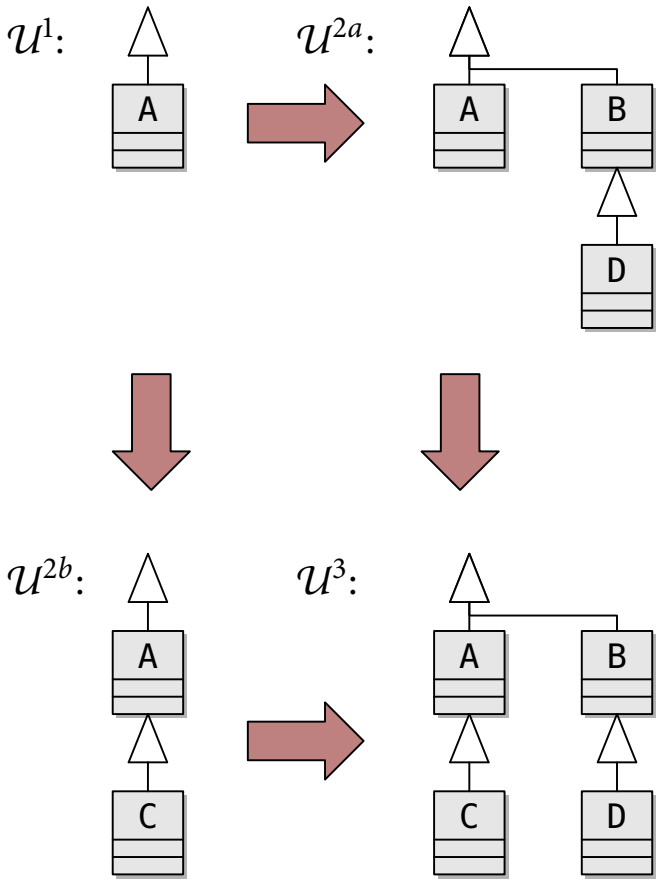
$$\mathcal{U}^1_{(\alpha^A,\beta^O)} = O \times (A \times \alpha^A_\perp + \beta^O)_\perp$$

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

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

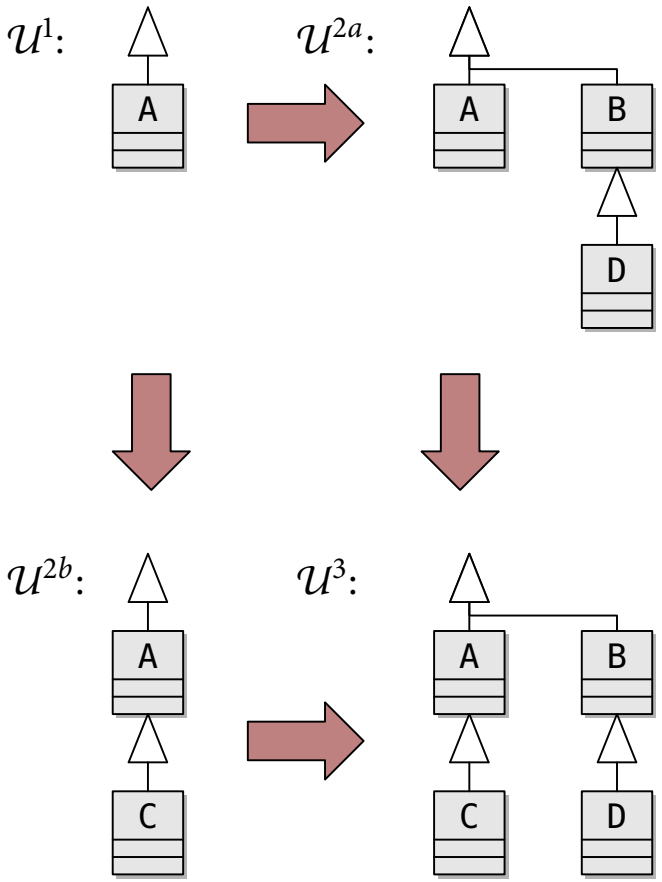
$$\mathcal{U}^3_{(\alpha^B,\alpha^C,\beta^O,\beta^A)} < \mathcal{U}^2_{(\alpha^B,\beta^O,\beta^A)} < \mathcal{U}^1_{(\alpha^A,\beta^O)} < \mathcal{U}^0_{(\alpha^O)}$$

Merging Universes

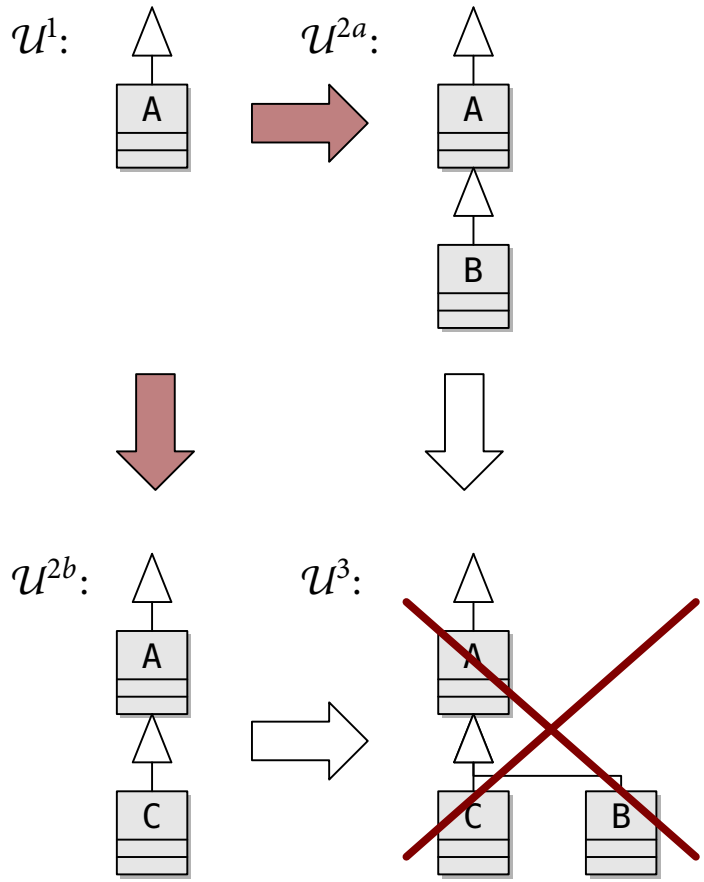


Non-conflicting Merges

Merging Universes



Non-conflicting Merges



Conflicting Merges

Operations Accessing the Object Store

- injections

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

- projections

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

- type casts

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

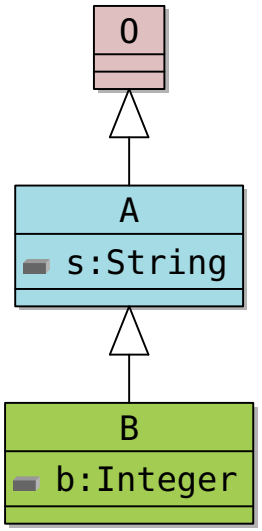
$$\mathrm{O}_{[\mathrm{A}]} = \mathrm{get}_{\mathrm{A}} \circ \mathrm{mk}_{\mathrm{O}} \qquad \text{with type } (A \times \alpha^{A}_{\perp} + \beta^{O})\ \mathsf{0} \to \alpha^{A}\ \mathrm{A}$$

- …

All definitions are generated automatically

Does This Really Model Object-orientation?

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



- subclasses are of the superclasses kind:

$$\frac{\mathrm{isType}_B\ self}{\mathrm{isKind}_A\ self}$$

- "re-casting":

$$\frac{\mathrm{isType}_B\ self}{self_{[A][B]} \neq \bot \wedge \mathrm{isType}_B\ (self_{[A][B][A]})}$$

- monotonicity of invariants, …

All rules are derived automatically

This is only the beginning …

- **Type-safety** of "object-type accessors" needs further processing.
- **Encoding invariants** requires (co-)-inductive definitions.
- **Solution:** encoding based on three levels:
  1. weakly typed data types
  2. strongly typed data types (and support for operations)
  3. constrained data modes

Encoding Attribute Accessors

Assume a class `Node` with an attribute `next`:

1. **Unsafe access** (reference, value, or $\bot$):

$$self.\,\mathrm{next}^{(0)} \equiv (\mathrm{fst} \circ \mathrm{snd} \circ \mathrm{snd} \circ \mathrm{fst})\ \lceil \mathrm{base}\ self \rceil$$

2. **Type-safe access** (typed object, value, or $\bot$):

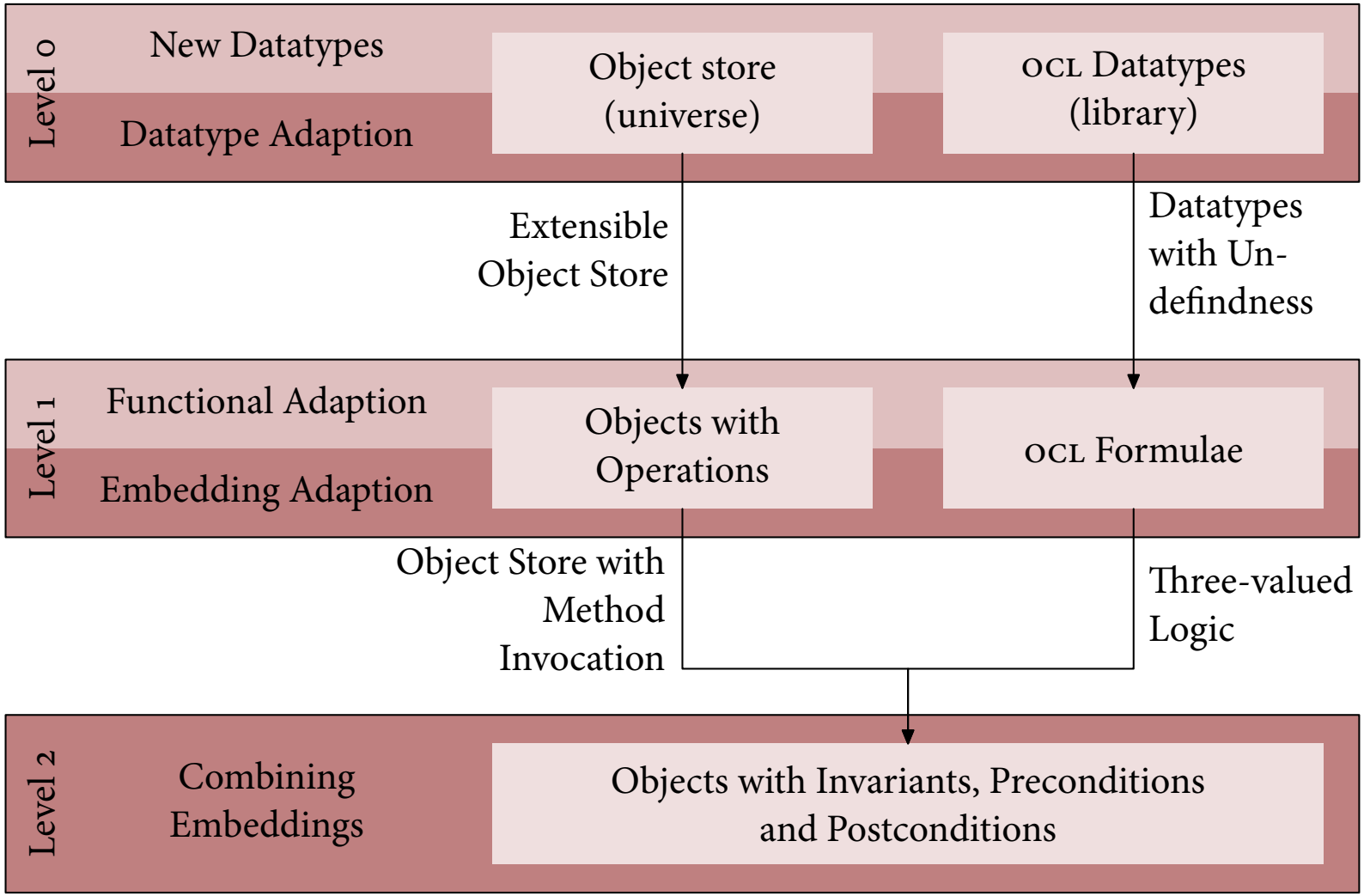
$$self.\,\mathrm{next}^{(1)} \equiv \lambda\,\sigma.\ \begin{cases} \mathrm{get}^{(0)}_{\mathrm{Node}}\ u & \text{if } \sigma(self.\,\mathrm{next}^{(0)}) = \lfloor u \rfloor, \\ \bot & \text{otherwise.} \end{cases}$$

3. **Semantically-safe access** (object satisfying invariant, value, or $\bot$):

$$self.\,\mathrm{next}^{(2)} \equiv \lambda\,\sigma.\ \begin{cases} self.\,\mathrm{next}^{(1)} & \text{if } \sigma \vDash self.\,\mathrm{next}^{(1)} \in \mathfrak{K}_{Node} \\ \bot & \text{otherwise.} \end{cases}$$

where $\mathfrak{K}_{Node}$ is the (co-) inductively defined *characteristic kind* set of class `Node`.

An multi-level object-oriented datatype-package



Case Studies (Datatype Package)

- Importing object-oriented models:

	Invoice	eBank	Company	R&L
classes	3	8	7	13
specification (lines)	149	114	210	520
generated theorems	647	1444	1312	2516
time (in seconds)	12	42	49	136

- The core library takes about 20 minutes (1200 secconds) to built
- Extensionality saves about 20 minutes on each import

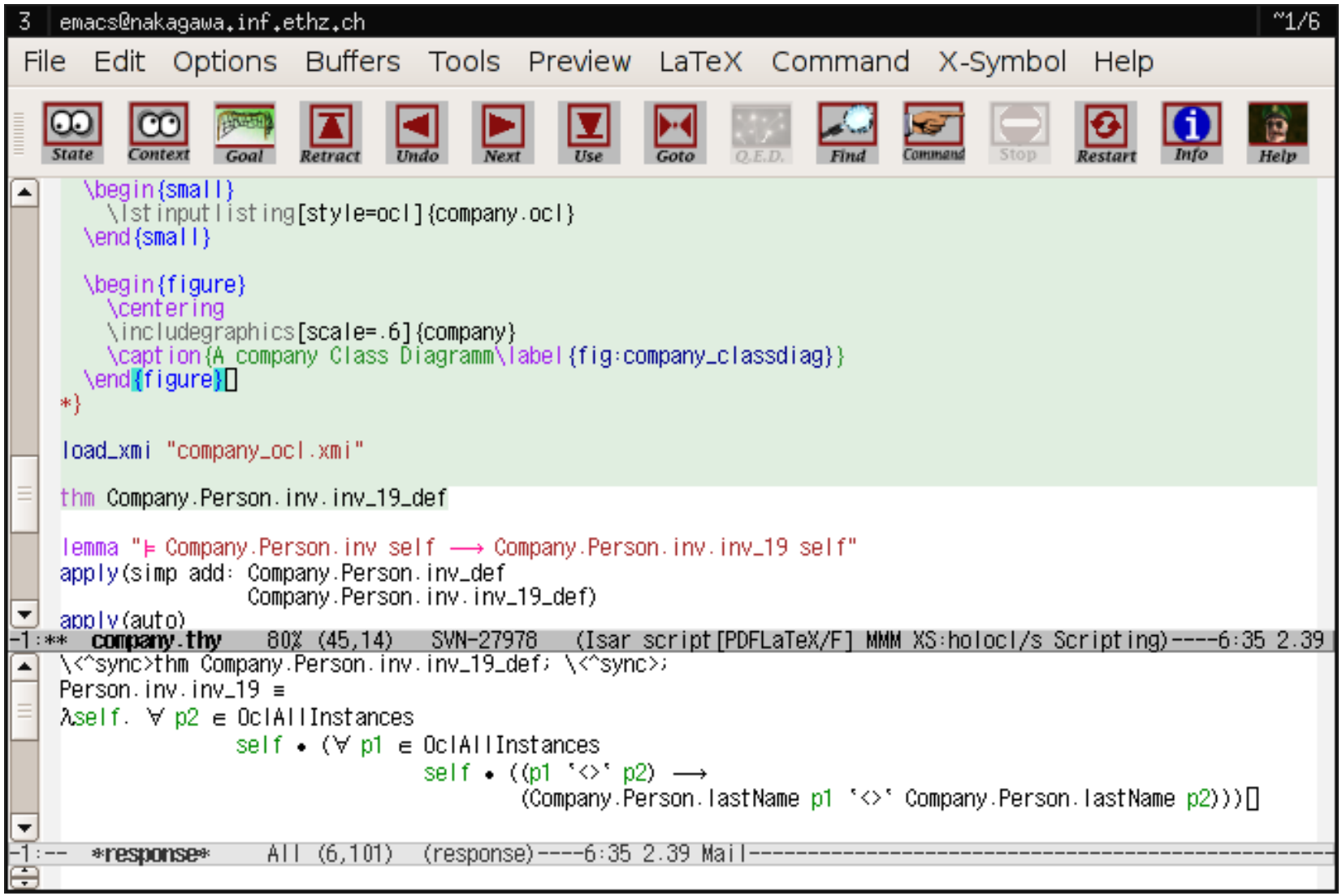
Challenges (Revisited)

- **Challenge 1:** *Can we find a injective, type preserving mapping of an object-oriented language (and datatypes) into HOL?*
  Yes, our encoding is even bijective.
- **Challenge 2:** *Can we support verification in a modular way (i. e., no replay of proof scripts after extending specifications)?*
  Yes, a specific form of extensionality can be supported.
- **Challenge 3:** *Can we ensure consistency of our representation?*
  Yes, by using a conservative embedding (deriving all rules).

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 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"
```

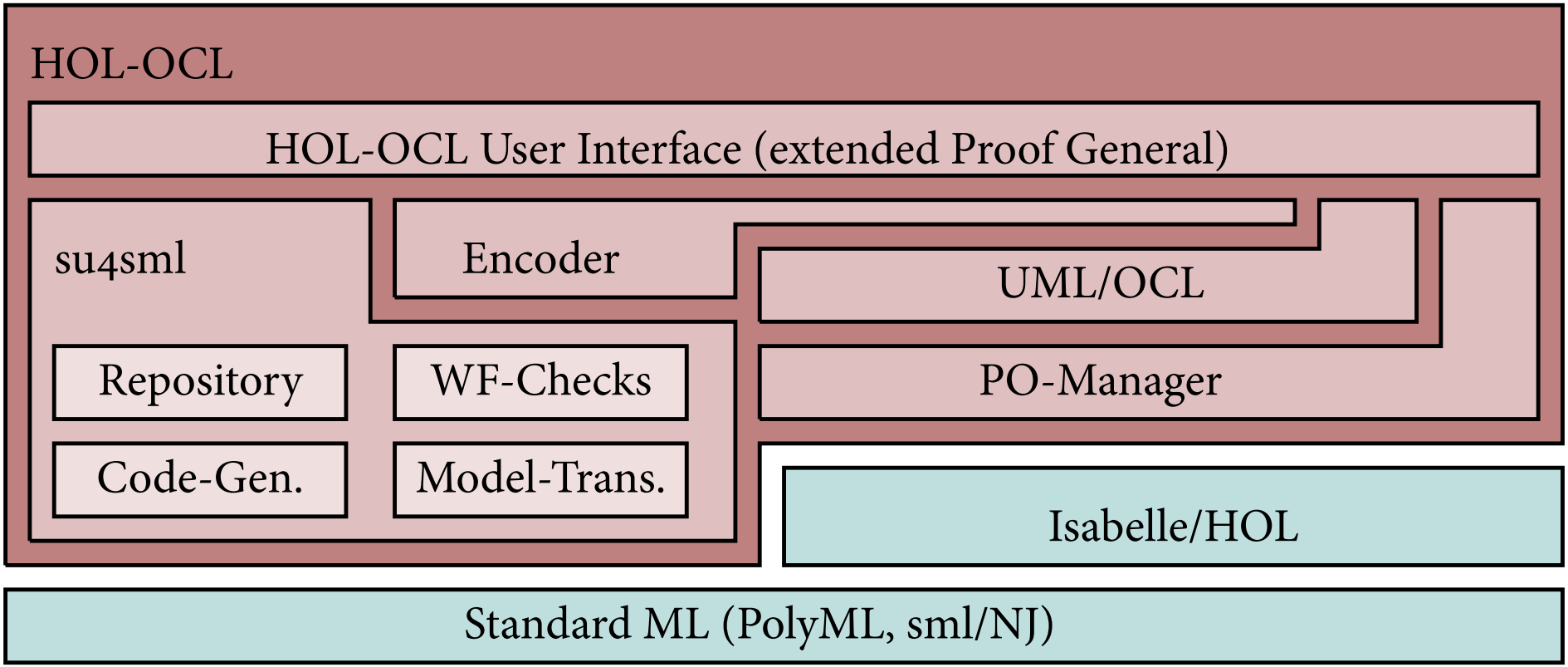
- starting a proof for a generated proof obligation:

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

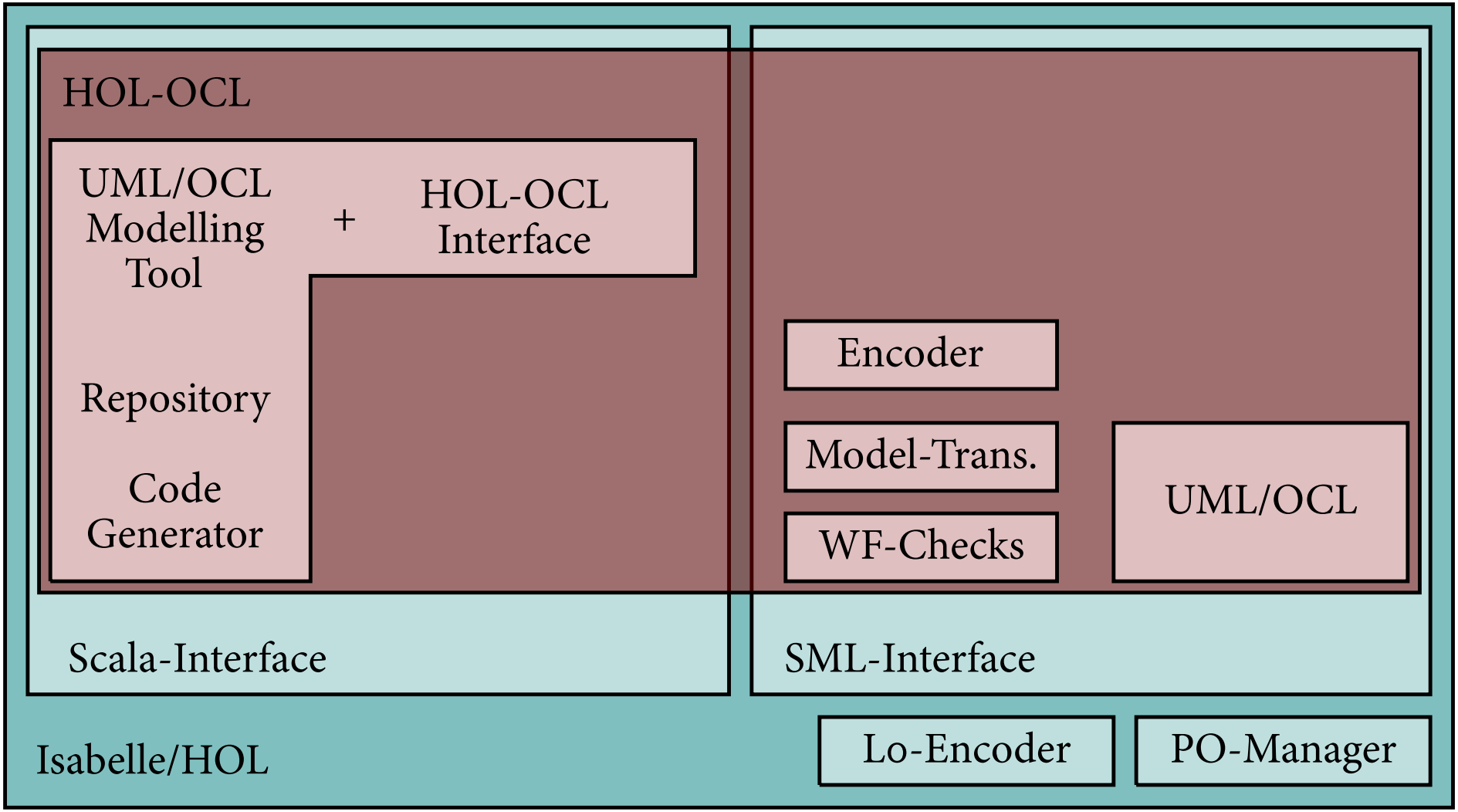
- generating code:

```
generate_code "java"
```

The HOL-OCL Architecture



The HOL-OCL Architecture (Next Generation)



Conclusion

- Technical challenges:
  - parsing and typing (!) concrete syntax can be slow
  - debugging simplifier setups is painful
  - defining new X-Symbol syntax is quite limited (compared to LaTeX)
  - Best practice for communicating with external tools is missing
  - …
- Conclusion
  - Isabelle is a framework for developing formal tools
    (even for tools where Isabelle not seen by the end-user)
  - The Scala-Layer enables many new features, e.g.,
    - Integration of new interaction paradigms
  - Isabelle can be (smoothly) integrated with external tools and libraries
  - …

Thank you for your attention!

Any questions or remaks?

Bibliography I

Achim D. Brucker, Jürgen Doser, and Burkhart Wolff.

An MDA framework supporting OCL.

Electronic Communications of the EASST, 5, 2006.

Achim D. Brucker.

An Interactive Proof Environment for Object-oriented Specifications.

PhD thesis, ETH Zurich, March 2007.

ETH Dissertation No. 17097.

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 (FASE08)*, number 4961 in Lecture Notes in Computer Science, pages 97–100. Springer-Verlag, 2008.

Bibliography II

Achim D. Brucker and Burkhart Wolff.

An extensible encoding of object-oriented data models in HOL.

Journal of Automated Reasoning, 41:219–249, 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.

Part I

Appendix

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.

The Library

The HOL-OCL library

- formalizes the built-in operations of UML/OCL,
- comprises over 10 000 definitions and theorems,
- build the basis for new, OCL specific, proof procedures,
- provides proof support for (formal) development methodologies.

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.

su4sml – Overview

su4sml is a UML/OCL (and SecureUML) model repository providing

- a database for syntactic elements of UML core, namely class models and state machines as well as OCL expressions.
- support for SecureUML.
- import of UML/OCL models in different formats:
  - XMI and ArgoUML (class models and state machines)
  - OCL (plain text files)
  - USE (plain text files describing class models with OCL annotations)
- a template-based code generator (export) mechanism.
- an integrated framework for model transformations.
- a framework for checking well-formedness conditions.
- a framework for generating proof obligations.
- an interface to HOL-OCL (encoder, po manager).

su4sml – Code Generators

su4sml provides a template-based code generator for

- Java, supporting
  - class models and state machines
  - OCL runtime enforcement
  - SecureUML
- C#, supporting
  - class models and state machines
  - SecureUML
- USE
- …

su4sml – Model Transformations

su4sml provides a framework for model transformation that

- supports the generation of proof obligations
- can be programmed in SML.

Currently, the following transformations are provided:

- a family of semantic preserving transformations for converting associations (e. g., n-ary into binary ones)
- a transformation from SecureUML/ComponentUML to UML/OCL.

su4sml – Well-formedness Checks

su4sml provides an framework for extended well-formedness checking:

- Checks if a given model satisfies certain syntactic constraints,
- Allows for defining dependencies between different checks
- Examples for well-formedness checks are:
  - restricting the inheritance depth
  - restringing the use of private class members
  - checking class visibilities with respect to member visibilities
  - …
- Can be easily extended (at runtime).
- Is integrated with the generation of proof obligations.

su4sml – Proof Obligation Generator

su4sml provides an 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

Motivation

Observation:

- UML/OCL is a *generic* modeling language:
  - usually, only a sub-set of UML is used and
  - there is no standard UML-based development process.
- Successful usage of UML usually comprises
  - a well-defined development process and
  - tools that integrate into the development process.

Conclusion:

- Formal methods for UML-based development should
  - support the local UML development methodologies and
  - integrate smoothly into the local toolchain.

A toolchain for formal methods should provide tool-support for **methodologies**.

Well-formedness of Models

Well-formedness Checking

- Enforce **syntactical** restriction on (valid) UML/OCL models.
- Ensure a minimal quality of models.
- Can be easily supported by automated tools.

Example

- There should be at maximum five inheritance levels.
- The Specification of public operations may only refer to public class members.
- …

Proof Obligations for Models

Proof Obligation Generation

- Enforce **semantical** restriction on (valid) UML/OCL models.
- Build the basis for formal development methodologies.
- Require formal tools (theorem prover, model checker, etc).

Example

- Liskov's substitution principle.
- Model consistency
- Refinement.
- …

Proof Obligations: Liskov's Substitution Principle

Liskov substitution principle

Let $q(x)$ be a property provable about objects x of type T. Then $q(y)$ should be true for objects y of type S where S is a subtype of T.

For constraint languages, like OCL, this boils down to:

- *pre-conditions* of overridden methods must be *weaker*.
- *post-conditions* of overridden methods must be *stronger*.

Which can formally expressed as implication:

- Weakening the pre-condition:

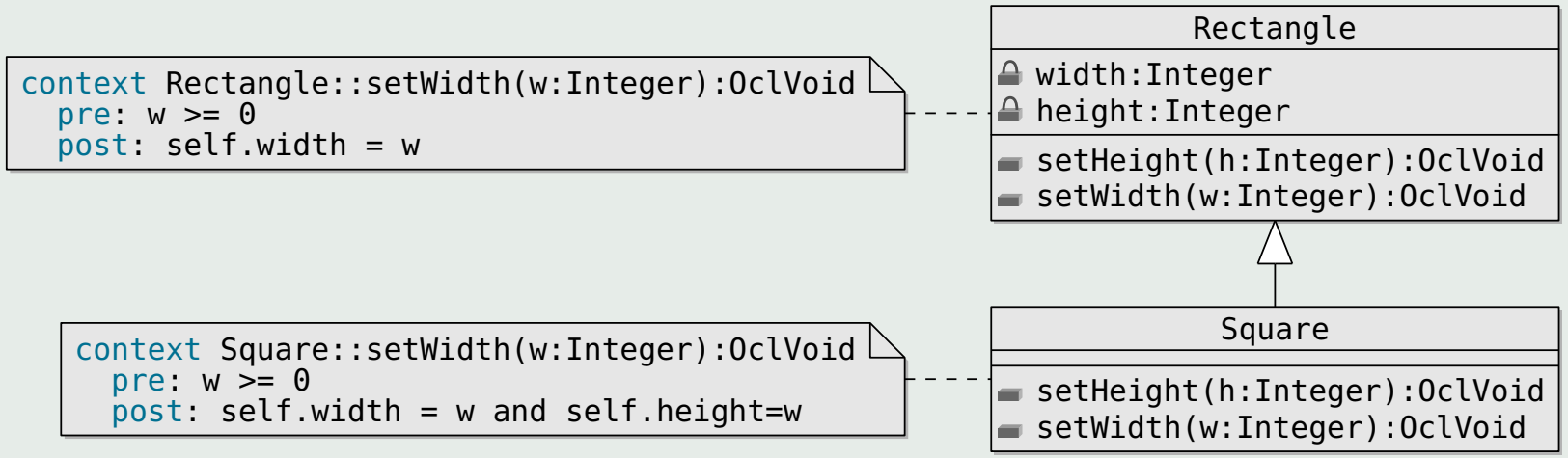
$$op_{\text{pre}} \longrightarrow op^{\text{sub}}_{\text{pre}}$$

- Strengthening the post-condition:

$$op^{\text{sub}}_{\text{post}} \longrightarrow op_{\text{post}}$$

Proof Obligations: Liskov's Substitution Principle

Example



- Weakening the pre-condition:

$$(w >= 0) \longrightarrow (w >= 0)$$

- Strengthening the post-condition:

$$(self.width = w \text{ and } self.height = w) \longrightarrow (self.width = w)$$

Methodology

A tool-supported methodology should

- integrate into existing toolchains and processes,
- provide a unified approach, integrating ,
  - syntactic requirements (well-formedness checks),
  - generation of proof obligations,
  - means for **verification** (proving) or **validation**, and of course
- all phases should be supported by tools.

Example

A package-based object-oriented refinement methodology.

Refinement – Motivation

Support top-down development from an abstract model to a more concrete one.

- We start with an abstract transition system

$$sys_{\text{abs}} = (\sigma_{\text{abs}}, init_{\text{abs}}, op_{\text{abs}})$$

- We refine each abstract operation op_{abs} to a more concrete one: op_{conc}.
- Resulting in a more concrete transition system

$$sys_{\text{conc}} = (\sigma_{\text{conc}}, init_{\text{conc}}, op_{\text{conc}})$$

- Such refinements can be chained:

$$sys_1 \rightsquigarrow sys_2 \rightsquigarrow \cdots \rightsquigarrow sys_n$$

E.g., from an abstract model to one that supports code generation.

Refinement: Well-formedness

If package B refines a package A, then
one should be able to
substitute every usage of package A with package B.

1. For each **public class** c of A, B must provide a corresponding public class c'.
2. Types of **public attributes** and **public operations** (arguments and return type) must be either basic datatypes or public classes.
3. For **each public** class c of A, we require that the corresponding class c' of B provides at least
   1. public attributes with the same name and
   2. public operations with the same name.
4. The types of corresponding and attributes and operations are compatible.

Refinement: Proof Obligations – Consistency

A transition system is consistent if:

- The set of initial states is non-empty, i. e.,

$$\exists \sigma.\ \sigma \in init$$

- The state invariant is satisfiable, i. e.,
  the conjunction of all invariants is invariant-consistent:

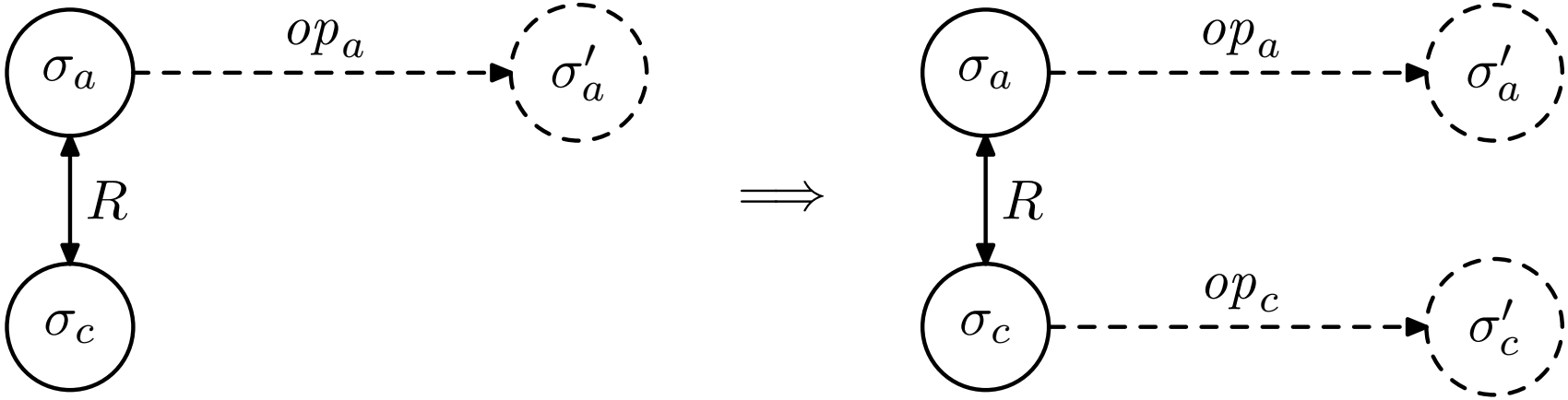
$$\exists \sigma.\ \sigma \vDash inv_1 \wedge \exists \sigma.\ \sigma \vDash inv_2 \wedge \cdots \wedge \exists \sigma.\ \sigma \vDash inv_n$$

- All operations op are implementable, i. e.,
  for each satisfying pre-state there exists a satisfying post-state:

$$\forall\ \sigma_{\text{pre}} \in \Sigma, self, i_1, \ldots, i_n.\ \sigma_{\text{pre}} \vDash \text{pre}_{op} \longrightarrow \exists\ \sigma_{\text{post}} \in \Sigma, result.\ (\sigma_{\text{pre}}, \sigma_{\text{post}}) \vDash \text{post}_{op}$$

Refinement: Proof Obligations – Implements

- Given an abstraction relation $R : \mathbb{P}(\sigma_{\text{abs}} \times \sigma_{\text{conc}})$ relating a concrete state S and an abstract states T.
- A forward refinement $S \sqsubseteq^{R}_{FS} T \equiv po_1(S, R, T) \wedge po_2(S, R, T)$ requires two proof obligations po_1 and po_2.
- **Preserve Implementability (po_1):**

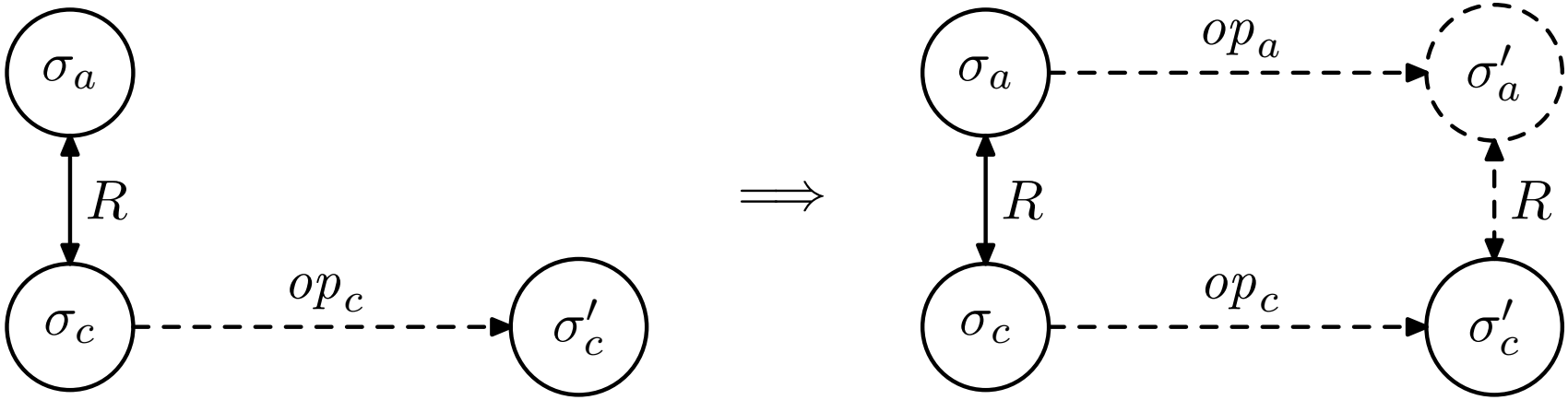


$$po_1(S, R, T) \equiv \forall \sigma_a \in \text{pre}(S), \sigma_c \in V.$$

$$(\sigma_a, \sigma_c) \in R \longrightarrow \sigma_c \in \text{pre}(T)$$

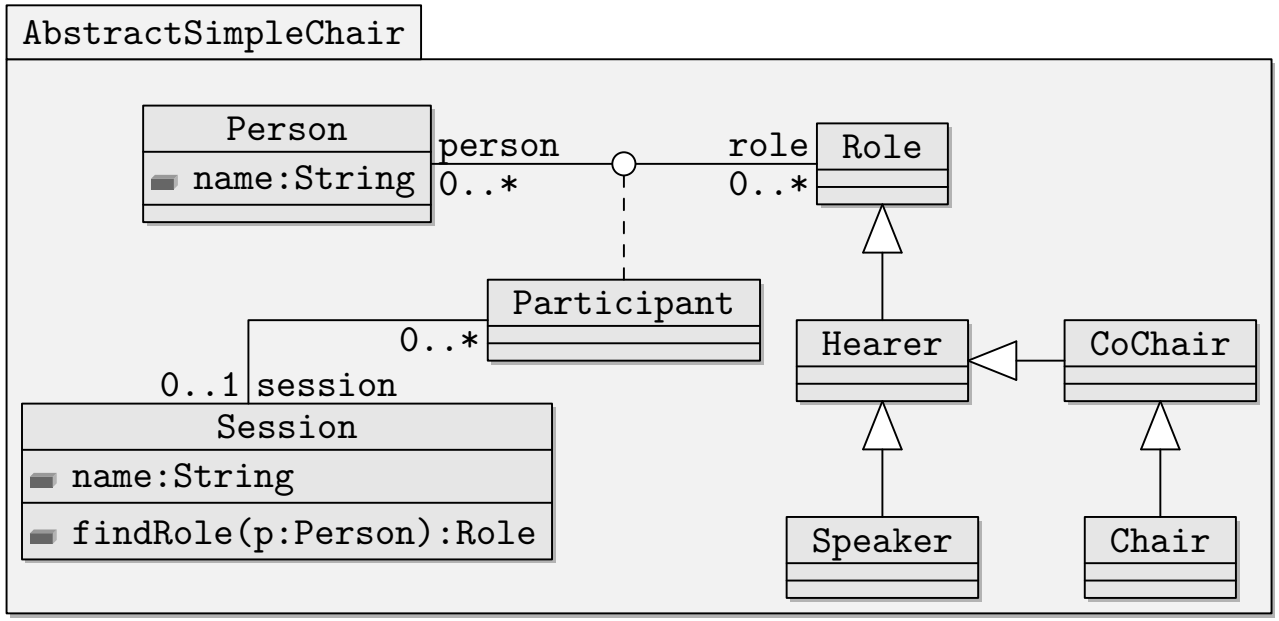
Refinement: Proof Obligations – Refines

- Given an abstraction relation $R : \mathbb{P}(\sigma_{\text{abs}} \times \sigma_{\text{conc}})$ relating a concrete state S and an abstract states T.
- A forward refinement $S \sqsubseteq^{R}_{FS} T \equiv po_1(S, R, T) \wedge po_2(S, R, T)$ requires two proof obligations po_1 and po_2.
- **Refinement (po_2):**



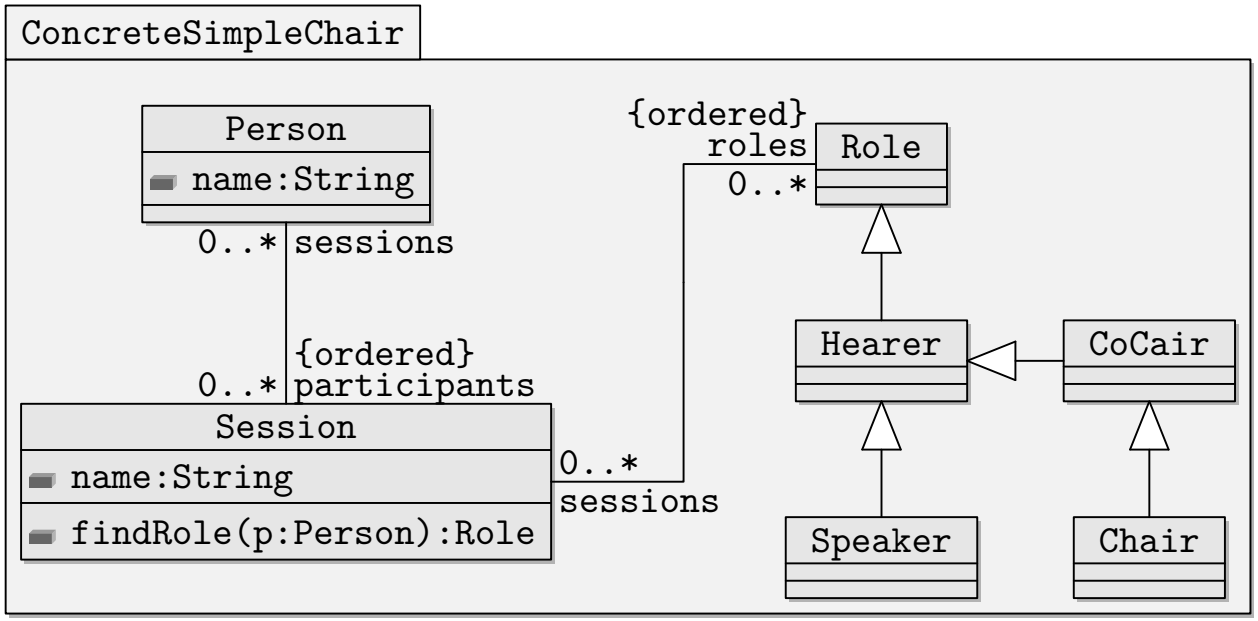
$$po_2(S, R, T) \equiv \forall \sigma_a \in \text{pre}(S), \sigma_c \in V.\ \sigma_{c'}.\ (\sigma_a, \sigma_c) \in R$$
$$\wedge\ (\sigma_c, \sigma'_c) \vDash_M T \longrightarrow \exists \sigma'_a \in V.\ (\sigma_a, \sigma'_a) \vDash_M S \wedge (\sigma_{a'}, \sigma_{c'}) \in R$$

Refinement Example: Abstract Model




```
context Session::findRole(person:Person):Role
  pre:  self.participates.person->includes(person)
  post: result=self.participants->one(p:Participant |
                                 p.person ≐ person ).role
        and self.participants ≐ self.participants@pre
        and self.name ≐ self.name@pre
```

Refinement Example: Concrete Model




```
context Session::findRole(person:Person):Role
  pre:  self.participants->includes(peson)
  post: result ≐ roles.at(participants.indexOf(p))
```

Refinement Example: Theory Sketch

```
theory SimpleChairRefinement imports OCL_methodology begin
import_model "SimpleChair.zargo" "SimpleChair.ocl"
refine "AbstractSimpleChair" "ConcreteSimpleChair"
po Refinement.findRole
```

$$\frac{\begin{array}{c}\forall \sigma \in \text{pre } S, \sigma' \in \text{pre } T.\ R_{Session}\ \sigma\ \sigma'\ self\ self' \\ \forall \sigma \in \text{pre } S, \sigma \in \text{pre } T.\ R_{Person}\ \sigma\ \sigma'\ p\ p' \\ \forall \sigma \in \text{pre } S, \sigma \in \text{pre } T.\ R_{Role}\ \sigma\ \sigma'\ result\ result'\end{array}}{\begin{array}{r}\text{AbstractSimpleChair.Session.findRole } self\ p\ result \\ \sqsubseteq^{R}_{FS} \text{ ConcreteSimpleChair.Session.findRole } self'\ p'\ result'\end{array}}$$

```
    apply( ... )
discharged
```

Outline

Simple Consistency Analysis I

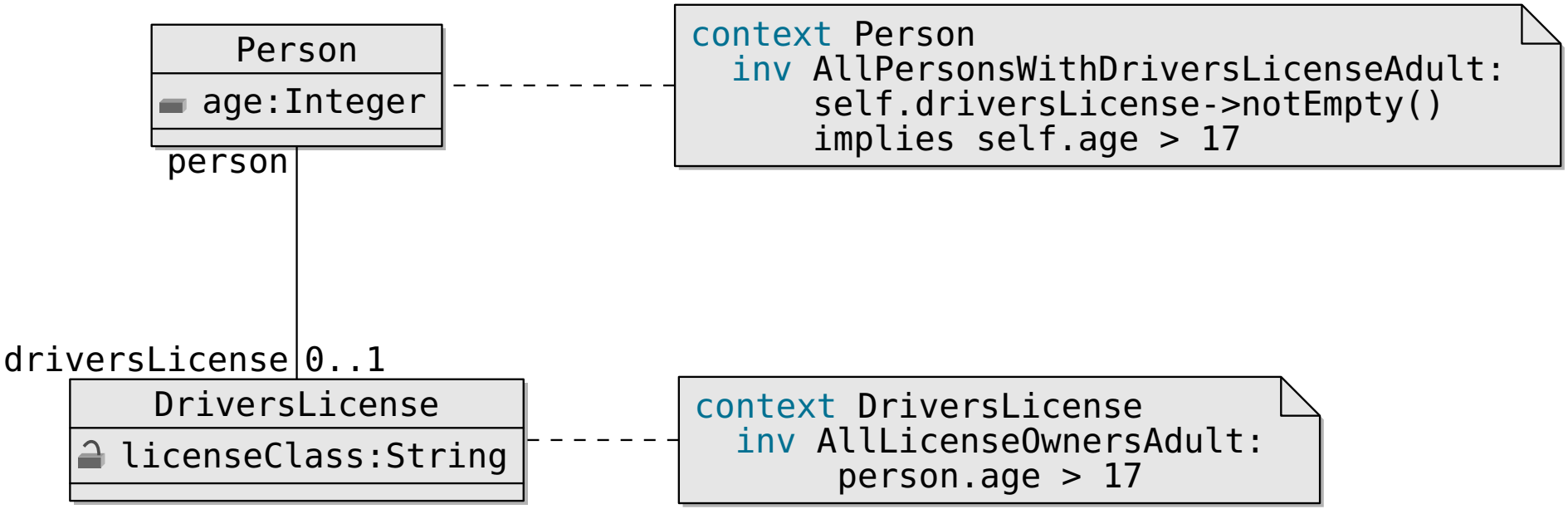


Figure: A simple model of vehicles and licenses

Simple Consistency Analysis II

```
lemma
assumes "τ ⊨ (Vehicles.Person.driversLicense(
                    Vehicles.DriversLicense.person self)).IsDefined()"
                    and "τ ⊨ (Vehicles.Person.age
                          (Vehicles.DriversLicense.person self)).IsDefined() "
  shows "τ ⊨ Person.inv.AllPersonsWithDriversLicenseAdult (
                      Vehicles.DriversLicense.person self)
             ⟶ τ ⊨ DriversLicense.inv.AllLicenseOwnersAdult self"
  apply(auto elim!: OclImpliesE)
  apply(cut_tac prems)
  apply(auto simp: inv.AllPersonsWithDriversLicenseAdult_def
                       inv.AllLicenseOwnersAdult_def
              elim!: OclImpliesE SingletonSetDefined)
done
```

Liskov's Substitution Principle I

```
context A::m(p:Integer):Integer
  pre: p > 0
  post: result > 0

context A::m(p:Integer):Integer
  pre: p >= 0
  post: result = p*p + 5

-- The following constraints overrides the specification for
-- m(p:Integer):Integer that was originally defined in
-- class A, i.e., C is a subclass of A.
context C::m(p:Integer):Integer
  pre:  p >= 0
  post: result > 1 and result = p*p+5
```

Liskov's Substitution Principle II

```
import_model "overriding.zargo" "overriding.ocl"

generate_po_liskov "pre"
generate_po_liskov "post"

po "overriding.OCL_liskov-po_lsk_pre-1"
   apply(simp add: A.m_Integer_Integer.pre1_def
                   A.m_Integer_Integer.pre1.pre_0_def
                   C.m_Integer_Integer.pre1_def
                   C.m_Integer_Integer.pre1.pre_0_def
                   A.m_Integer_Integer.pre1.pre_1_def)
   apply(ocl_auto)
discharged
```