# From Program Execution to Automatic Reasoning

## Integrating Ontologies into Programming Languages

Alexander Paar

# State of the Art

- Widely used object-oriented programming languages include a built-in static type system
- Conceptual framework that makes it particularly easy to
    - design,
    - understand, and
    - maintain object-oriented systems
- In recent years *schema languages* and *ontology languages* emerged
    - Programming language agnostic data types and content models
    - Automatic reasoning about domain model
- Powerful modeling features readily available
- Impedance mismatch makes programming difficult and error-prone
- Facilitate the development of intelligent applications

# The Extensible Markup Language (XML)

- Separate formatting of a document from its content
- *General Markup Language* (GML) proposed by Charles Goldfarb, Edward Mosher, and Raymond Lorie in 1967
- Standard General Markup Language (SGML) became an ISO standard in 1986
- In 1999, the Extensible Markup Language (XML) became a W3C standard
- XML is not a language

# XML Schema Definition (XSD)

- Published as a W3C recommendation in 2001
- XSD specification comprises two parts
    - XML Schema Part 1: Structures
    - XML Schema Part 2: Datatypes
- Atomic XSD data types are triples
    - Value space
    - Lexical space
    - Fundamental facets

# Built-in XSD Data Types

- 19 primitive built-in data types
- Several isomorphic mappings to programming language data types

| .NET value type | XSD type | .NET value type | XSD type |
|---|---|---|---|
| *System.String* | *xsd#string* | *System.Int16* | *xsd#short* |
| *System.Boolean* | *xsd#boolean* | *System.UInt16* | *xsd#unsignedShort* |
| *System.Single* | *xsd#float* | *System.Int32* | *xsd#int* |
| *System.Double* | *xsd#double* | *System.UInt32* | *xsd#unsignedInt* |
| *System.SByte* | *xsd#byte* | *System.Int64* | *xsd#long* |
| *System.Byte* | *xsd#unsignedByte* | *System.UInt64* | *xsd#unsignedLong* |

- Why are there no isomorphic mappings in general?

# User-defined Type „age“

- A valid age is any integer number greater than or equal to zero and less than 110.

```
<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema>
  <xsd:simpleType name="age">
    <xsd:restriction base="xsd:int">
      <xsd:minInclusive value="0"/>
      <xsd:maxExclusive value="110"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

# XSD Data Types in C#

- Mapping option 1

```
class Person {
  public uint HasAge;
}
```

- Mapping option 2

```
using System.Diagnostics;
internal class Person {
 private uint _HasAge;
 public uint HasAge {
  get { return _HasAge; }
  set { Trace.Assert(value >= 0 && value < 110);
        _HasAge = value; }
 }
}
```

# XSD Data Types in C# cont'd

- Mapping option 3

```
using System.Diagnostics;
internal struct uint110 {
 private uint value;
 public static implicit operator uint110(uint value) {
  Trace.Assert(value < 100);
  return new uint110 {value = value};
 }
 public static implicit operator uint(uint110 value) {
  return value.value;
 }
}
```

# XSD Data Types in C# cont'd

- Mapping option 4

```
abstract class XsdAnySimpleType {
 protected object value;
}
class XsdUnsignedInt : XsdAnySimpleType {
 public static implicit operator
  XsdUnsignedInt(uint value) {
   return new XsdUnsignedInt {value = value};
 }
 public static implicit operator
  uint (XsdUnsignedInt value) {
   return (uint) value.value;
 }
}
```

# XSD Constraining Facets

- 12 constraining facets

| Constraint | Description |
|---|---|
| *length* | defines the number of *units of length* |
| *minLength* | defines the minimum number of *units of length* |
| *maxLength* | defines the maximum number of *units of length* |
| *pattern* | constrains the lexical space to literals that match a specific pattern |
| *enumeration* | constrains the value space to a specified set of values |
| *minInclusive* | defines the inclusive lower bound of the value space |
| *minExclusive* | defines the exclusive lower bound of the value space |
| *maxExclusive* | defines the exclusive upper bound of the value space |
| *maxInclusive* | defines the inclusive upper bound of the value space |
| *totalDigits* | defines the maximum number of values in the value space |
| *fractionDigits* | defines the minimum difference between values in the value space |
| *whiteSpace* | controls the normalization of string data types (modifying) |

- A *constraining facet* is an optional property that can be applied to an atomic data type to constrain its value space
- A value space $v(T)$ is the set of values for a given atomic data type $T$

# Constraint Application

- The value space function $\upsilon(P)$ is defined semantically for each primitive base type $P$ under consideration
- The value space of a base type $T$ can be restricted by the application of one or more constraints $c_i = \phi(TV)b_i \ ^{i \in 1..n}$
- A constraint body $b = \{x | x \in \upsilon(TV)\} \bigcap_{k \in 1..m} \{x | x \prec literal_k\}$ defines the intersection of the value space of $TV$ and those values that satisfy the properties $x \prec literal_k \ ^{k \in 1..m}$
- $T_n = T.c_1. \ldots .c_n \equiv \bigcap_{i \in 1..n}^{T} c_i$

# Constraint Subsumption

- S-CstrVSpace (non-algorithmic)

$$\frac{\upsilon(c_1\{\{TV \leftarrow T\}\}) \subseteq \upsilon(c_2\{\{TV \leftarrow T\}\})}{c_1 <:: c_2}$$

- S-CstrWidth

$$\phi(TV)\{x|x \in \upsilon(TV)\} \bigcap_{i \in 1..n+k} \{x|x \prec y_i\} <:: \phi(TV)\{x|x \in \upsilon(TV)\} \bigcap_{i \in 1..n} \{x|x \prec y_i\}$$

- S-CstrDepth

$$\frac{\text{for each } i \ \{x|x \prec y_i\} \subseteq \{x|x \prec z_i\}}{\phi(TV)\{x|x \in \upsilon(TV)\} \bigcap_{i \in 1..n} \{x|x \prec y_i\} <:: \phi(TV)\{x|x \in \upsilon(TV)\} \bigcap_{i \in 1..n} \{x|x \prec z_i\}}$$

# Subtyping of XSD Data Types

- S-VSpace (non-algorithmic)

$$\frac{v(S) \subseteq v(T)}{S <: T}$$

- S-Width

$$\frac{S = \bigcap_{i \in 1..n+k}^{T} c_i \qquad U = \bigcap_{i \in 1..n}^{T} c_i}{S <: U}$$

- S-Depth

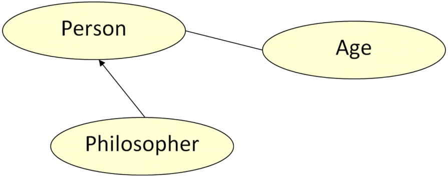
$$\frac{\text{for each } i \quad c_i <:: d_i \qquad S = \bigcap_{i \in 1..n}^{T} c_i \qquad U = \bigcap_{i \in 1..n}^{T} d_i}{S <: U}$$

# An XSD Type System

- Interpretation of XSD data types as computational structures
- Foundation for the integration into programming language type system
- Will it be *sound* (will there be *progress* and *preservation*)?
- Yes, it will because. . .
  - XSD type construction does not pertain to programming language typing and evaluation rules.
  - Subsumption property (if $S' <: S$ and $S <: T$ then $S' <: T$) can be easily proved.
- Practical implementation in the Zhi# programming language

# Initial Knowledge Representation Schemes

- In the 1960s, network-based notations occured
- Hierarchical structures in monotonic inheritance networks (1970s)




- Translation of semantic networks into fragments of first-order predicate calculus (1980s)

$$\frac{\text{Philosophers are men.}}{\text{Therefore philosophers' thoughts are thoughts of men.}}$$

# Description Logics (DL)

- Decision procedures of DL systems always terminate
- Efficient query answering
- Define the concepts of a domain of discourse
- Use concepts to specify properties of objects
- General knowledge contained in the *TBox*
- Contingent knowledge contained in the *ABox*
- Fundamental inference is *subsumption*: $C \sqsubseteq D$

# The $\mathcal{AL}$ Description Logic

- Basic Description Logic introduced 1991

| **Constructor Name** | **Syntax** | **Semantics** |
|---|---|---|
| atomic concept | $A$ | $A^{\mathcal{I}} \subseteq \triangle^{\mathcal{I}}$ |
| top level concept | $\top$ | $\top^{\mathcal{I}} = \triangle^{\mathcal{I}}$ |
| bottom concept | $\bot$ | $\bot^{\mathcal{I}} = \{\}$ |
| atomic negation | $\neg A$ | $(\neg A)^{\mathcal{I}} = \triangle^{\mathcal{I}} \backslash A^{\mathcal{I}}$ |
| intersection | $C_1 \sqcap C_2$ | $(C_1 \sqcap C_2)^{\mathcal{I}} = C_1{}^{\mathcal{I}} \cap C_2{}^{\mathcal{I}}$ |
| value restriction | $\forall R.C$ | $(\forall R.C)^{\mathcal{I}} = \{x \mid \forall y.\langle x, y\rangle \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$ |
| ltd. ex. quantification | $\exists R.\top$ | $(\exists R.\top)^{\mathcal{I}} = \{x \mid \exists y.\langle x, y\rangle \in R^{\mathcal{I}}\}$ |

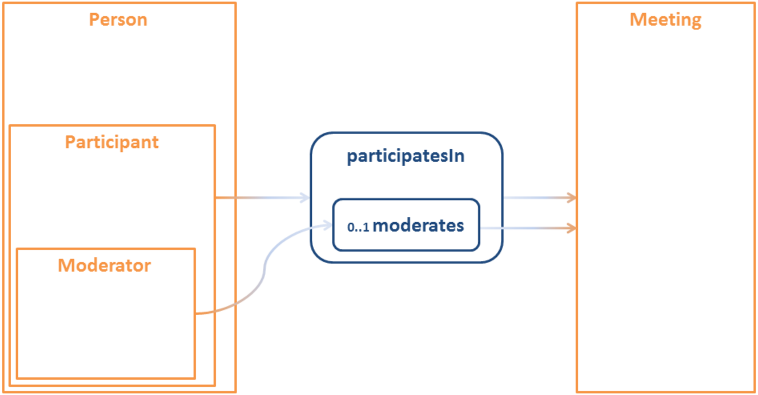
- Set theoretic interpretation $\mathcal{I} = (\triangle^{\mathcal{I}}, \cdot^{\mathcal{I}})$

# A Business Meeting Scenario

- Atomic concepts: *Person*, *Meeting*
- General concept inclusion: $Employee \sqsubseteq Person$
- Ltd. ex. quantification:
  $MeetingParticipant \equiv Person \sqcap \exists attendsMeeting.\top$
- Value restriction:
  $MarketingMeeting \equiv Meeting \sqcap \forall hasTopic.MarketingTopic$
- Negation: $Guest \equiv Person \sqcap \neg Employee$

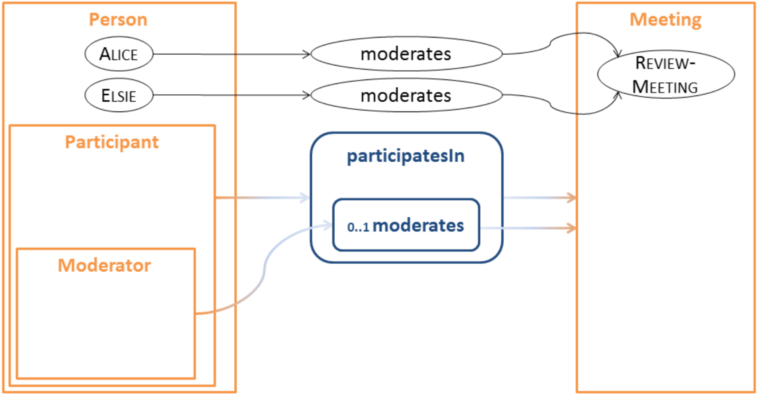
# Reasoning With The $\mathcal{SHOIN}$ DL

- $\mathcal{SHOIN}$ extends $\mathcal{AL}$ with additional modelling features: negation ($\mathcal{C}$), qualified number restrictions ($\mathcal{Q}$), role hierarchies ($\mathcal{H}$), and inverse ($\mathcal{I}$) and transitive roles ($\mathcal{R}+$)
- $\mathcal{SHOIN}$(**D**) includes a concrete domain: data types



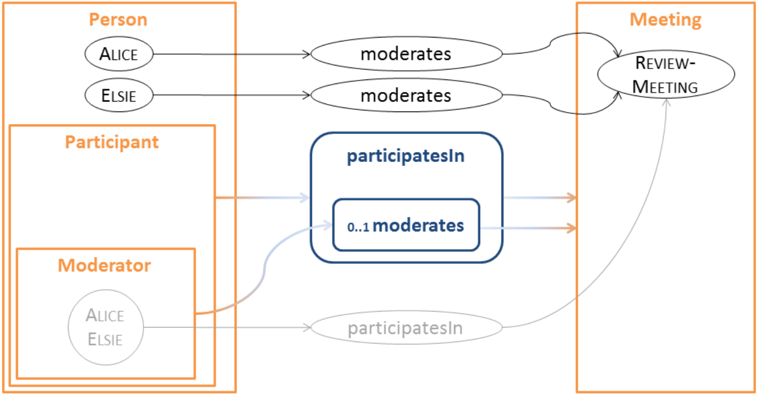
# Reasoning With The $\mathcal{SHOIN}$ DL

- $\mathcal{SHOIN}$ extends $\mathcal{AL}$ with additional modelling features: negation ($\mathcal{C}$), qualified number restrictions ($\mathcal{Q}$), role hierarchies ($\mathcal{H}$), and inverse ($\mathcal{I}$) and transitive roles ($\mathcal{R}+$)
- $\mathcal{SHOIN}$(**D**) includes a concrete domain: data types



# Reasoning With The $\mathcal{SHOIN}$ DL

- $\mathcal{SHOIN}$ extends $\mathcal{AL}$ with additional modelling features: negation ($\mathcal{C}$), qualified number restrictions ($\mathcal{Q}$), role hierarchies ($\mathcal{H}$), and inverse ($\mathcal{I}$) and transitive roles ($\mathcal{R}+$)
- $\mathcal{SHOIN}(\mathbf{D})$ includes a concrete domain: data types



# API-based Ontology Management

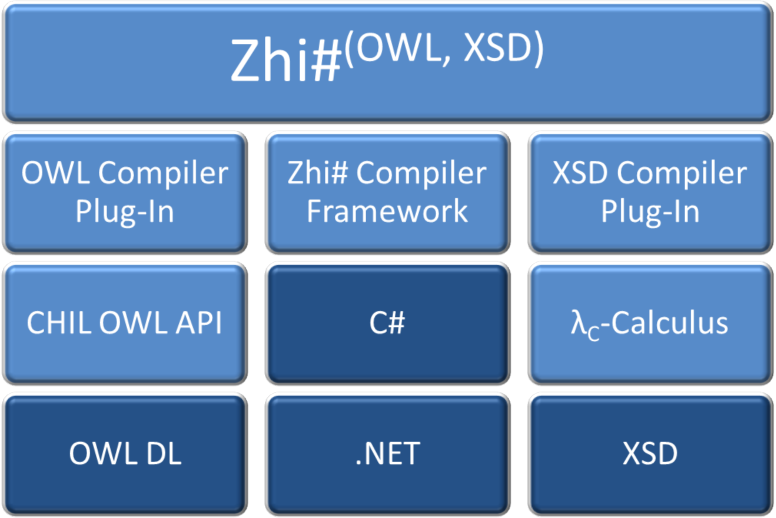
- No programming language integration
- Insufficient compiler/IDE support
- Jena Semantic Web Framework widely used

```
public class Program {
 public static void main(String[] args) {
  String eval = "http://www.zhimantic.com/eval#";

  OntModel m = ModelFactory.createOntologyModel(PelletReasonerFactory.THE_SPEC);
  m.read(new FileInputStream("Evaluation.owl"), "");

  Individual a = m.getOntClass("http://www.w3.org/2002/07/owl#Thing").
   createIndividual(eval + "a");

  Individual o = m.getOntClass("http://www.w3.org/2002/07/owl#Thing").
   createIndividual(eval + "o");

  a.addProperty(m.getObjectProperty(eval + "R"), o);

  for (Iterator it = m.getOntClass(eval + "B").listInstances(); it.hasNext();) {
   System.out.println(((Individual) it.next()).getURI());
  }

  o.addProperty(m.getDatatypeProperty(eval + "T"),
   m.createTypedLiteral("23", "http://www.w3.org/2001/XMLSchema#positiveInteger"));
 }
}
```

# Ontologies vs. Object-Orientation

- In Java/C# properties are class members
- In DL ontologies. . .
    - . . . properties form a separate hierarchy (i.e. property centric modeling)
    - . . . property domain and range restrictions are used for ontological reasoning
- Description Logics make the *open world assumption*
- Description Logics do not make the *unique name assumption*

## The Zhi# („Semantic C#“) Approach

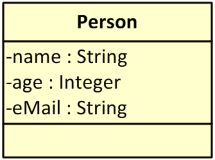


# XSD Data Types in Zhi# Programs

- Precise static typing and type inference for XSD data types

```
using System;
import XML xsd = http://www.w3.org/2001/XMLSchema;
import XML app = http://www.example.com/datatypes;
class C {
 public void f() {
  int i = new Random().Next(); //i:int
  if (i < 110)
   #app#age a = i; //a:int{>= 0}{< 110}, i:int{< 110} Error!
 }
}
```

- Implement OCL expressions with XSD-like data types




**context** *Person*
 **inv** *age >= 0 && age < 110*
**context** *Person*
 **inv** *name.size() < 40*

# Ontologies in Zhi# Programs

- Combination of static typing and dynamic checking
- Full support for datatype properties

```
using System;
import XML xsd = http://www.w3.org/2001/XMLSchema;
import OWL ont = http://www.zhimantic.com/ont;
class C {
 public DateTime f(#ont#Meeting m, #xsd#dateTime xdt) {
  m.#ont#beginsAt = xdt;
  #xsd#duration xd = "P0Y0M0DT1H30M0S";
  m.#ont#endsAt = xdt + xd;

  #ont#Person alice = new #ont#Person("Alice");
  #ont#Person bob = new #ont#Person("Bob");
  #ont#Person charlie = new #ont#Person("Charlie");

  m.#ont#hasParticipant = bob;
  charlie.#ont#participatesIn = m;
  alice.#ont#moderates = m;

  foreach (#ont#Participant p in m.#ont#hasParticipant)
   Console.WriteLine(p + " participates in " + m);

  if (m is #ont#ImportantMeeting)
   Console.WriteLine(m + " is an important meeting");

  return (DateTime) m.#ont#endsAt;
 }
}
```

# Conclusion

- Schema and ontology languages can be integrated in object-oriented programming language
- Object-oriented notation sufficient
- Separation of concerns without impedance mismatch
- Applications become intelligent
- Pay as you go
- Programming languages need to become aware of information spaces
- First Workshop on Programming the Semantic Web
  `www.inf.puc-rio.br/~psw12/`

# Contact


## Alexander Paar

alexpaar@acm.org

www.alexpaar.de


## The Zhi# Approach

zhisharp.sourceforge.net


## Acknowledgements

I kindly acknowledge the support of my employer TWT GmbH Science and Innovation in making my participation possible.

www.twt-gmbh.de


C#
XSD
OWL