CONTENT

- Software Engineering - The Problem
- Software Modeling - The Domain
- How to formalize OO software models
- Why Description Logic?
- Why CNL instead of OCL?
- My approach – what is done so far
- The Toolchain
- Possible future work
- Discussion
THE PROBLEM

- Software systems are more and more complex
  - Without a support of formal-methods it is almost impossible to trace and understand consequences of even small change of design in a complex software system. Strategic decisions that are made by the authorities lack of information about the real state of the software product

- Software engineers are in general not familiar with formal methods:
  - They use natural language to describe constrains (in comments)
SOFTWARE MODELING

- **UML+OCL**
  - UML is a standard software modeling language nowadays
  - Users of the UML can use OCL to specify constraints and other expressions attached to their models

- **LePUS3**
  - Capture and convey the building-blocks of object-oriented design
  - Automatic verification of Design Patterns

- **SEMAT Initiative**
  - *Ivar Jacobson*, Bertrand Meyer, and Richard Sole
  - Identifies “Methods&Tools” as one of macro-trends in modern Software Engineering
  - Back to the Clean-Room methodologies
  - Searching for a Kernel Language
**SEMAT – Kernel Language Requirements**

- The language can cover all relevant practices and patterns, and their composition, in today’s methods.
- It supports composing them in different ways to describe new method elements.
- It is extendible, allowing the description of yet-to-be invented method elements and their elements (such as individual practices).
- The descriptions are easy to understand; the language should be designed for the developer community (not just process engineers and academics).
- The language support simulating the application of method elements.
- The language provide validation mechanisms.
THE DOMAIN

What is a static structure (or a code structure) of a computer program?
- Opposite to runtime-structure
- Well defined in terms of compilation-time to machine code
- Not well defined from theoretical perspective

Why is it important?
- Separates the computer program to design and implementation
- The software design can be treated as an ontology
- Shifts imperative parts to declarative ones
DESCRIPTION LOGIC

- Mostly used in Semantic-Web
  - The math behind OWL

- DL is decidable fragment of FOL (with additional features)
  - It is a core property for static structures

- Complexity vs. Responsiveness
  - Effective reasoning implemented

- It is possible to Verbalize it in CNL
  - ACE ↔ OWL
  - $\text{CE}_{DL}$—prototypical implementation in JavaCC
THE TOOLCHAIN

- Model Description Language Generator
- Model Description Language Parser
- Ontology
- Model subsumptions
- Description Logic Reasoner
- Source code
- Source code generator
- Reverse engineering knowledge
# DL Constructors

<table>
<thead>
<tr>
<th>Construction</th>
<th>Meaning</th>
<th>Transcription</th>
</tr>
</thead>
<tbody>
<tr>
<td>C(\land) A</td>
<td>atomic concept</td>
<td>&lt;noun&gt;</td>
</tr>
<tr>
<td></td>
<td>universal concept</td>
<td>something</td>
</tr>
<tr>
<td></td>
<td>bottom concept</td>
<td>nothing</td>
</tr>
<tr>
<td>□ A</td>
<td>atomic negation</td>
<td>not &lt;noun&gt;</td>
</tr>
<tr>
<td>C(\cap) D</td>
<td>intersection</td>
<td>... and ...</td>
</tr>
<tr>
<td>C(\cup) C</td>
<td>value restriction</td>
<td>&lt;verb&gt; only &lt;noun&gt;</td>
</tr>
<tr>
<td></td>
<td>weak existential restriction</td>
<td>&lt;verb&gt; something</td>
</tr>
<tr>
<td>□ D</td>
<td>union</td>
<td>... or ...</td>
</tr>
<tr>
<td>□ R(\cap) C</td>
<td>existencial restriction</td>
<td>&lt;verb&gt; &lt;noun&gt;</td>
</tr>
<tr>
<td>□ C</td>
<td>negation</td>
<td>not ...</td>
</tr>
<tr>
<td>□ nR</td>
<td>at most</td>
<td>&lt;verb&gt; at most &lt;num&gt; &lt;noun&gt; things</td>
</tr>
<tr>
<td>□ nR</td>
<td>at least</td>
<td>&lt;verb&gt; at least &lt;num&gt; &lt;noun&gt; things</td>
</tr>
<tr>
<td>□ a1, a2, ..., an</td>
<td>one of</td>
<td>either a1 or a2, ... an</td>
</tr>
<tr>
<td>P, Q, R(\circ) R(\hat{\circ})</td>
<td>inverse relation</td>
<td>&lt;verb&gt; by</td>
</tr>
<tr>
<td>□ nR(\cap) C</td>
<td>quantified</td>
<td>&lt;verb&gt; at most &lt;num&gt; &lt;noun&gt;</td>
</tr>
<tr>
<td>□ nR(\cap) C</td>
<td>number restriction</td>
<td>&lt;verb&gt; at least &lt;num&gt; &lt;noun&gt;</td>
</tr>
<tr>
<td>P(\cap) Q(\hat{\circ}) R</td>
<td>complex role inclusion</td>
<td>if &lt;role expr&gt;</td>
</tr>
</tbody>
</table>
## DL Expressions

<table>
<thead>
<tr>
<th>Construction</th>
<th>Meaning</th>
<th>Transcription</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C \subseteq D$</td>
<td>concept</td>
<td>Every $&lt;C&gt;$ is a $&lt;D&gt;$</td>
</tr>
<tr>
<td>$C \sqsubseteq \exists R \cdot D$</td>
<td>specification</td>
<td>Every $&lt;C&gt; \ &lt;R&gt; \ a \ &lt;D&gt;$</td>
</tr>
<tr>
<td>$C(I)$</td>
<td>assertions</td>
<td>$&lt;I&gt;$ is $&lt;C&gt;$</td>
</tr>
<tr>
<td>$R(I,J)$</td>
<td></td>
<td>$&lt;I&gt; \ &lt;R&gt; \ &lt;J&gt;$</td>
</tr>
<tr>
<td>$C(x), \ {x} \subseteq D$</td>
<td>unnamed</td>
<td>The $&lt;C&gt;$ is a $&lt;D&gt;$</td>
</tr>
<tr>
<td>$D(x), C(x)$</td>
<td>instances</td>
<td>The $&lt;C&gt;$ is the $&lt;D&gt;$</td>
</tr>
</tbody>
</table>
The semantic of OWL and UML are different

- How to map UML to OWL
  - UML classes are like templates that define a runtime object and its storage.
  - OWL classes are like labels for concepts,
  - There is no 1:1 full mapping (problems include aggregation and polymorphism)
  - There is no way to describe high-level structures

- Diego Calvanese (2005): Mapping UML to ALUNI.
  - Reasoning on UML class diagrams

- OMG
  - Metamodels of UML and OWL
THE ALTERNATIVE MAPPING

- **Class as an instance**
- It can be threaded as an instance in fact – in some OO languages class is an runtime object - reflexion
- We can build complex expressions in DL for classes
### Basic OO Model Domains and Ranges

<table>
<thead>
<tr>
<th></th>
<th>class</th>
<th>object</th>
<th>function</th>
<th>method</th>
<th>attribute</th>
<th>signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>extend</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>materialize</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>have</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fill</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>implement</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>identify</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>call</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>create</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CLASS AND OBJECT

C is a class.

{C} is a class

O is an object.

{O} is an object

{O} materializes {C}

O materializes C.
CLASS INHERITANCE

B extends A.  

C extends A.
Properties of Roles

materialize, realize 
materialize, extend, realize 
materialize(extend)* mrealize
**Simple Constrains**

- **Abstract class** $\rightarrow$ \$realize.$^\wedge$
- **Final class** $\rightarrow$ \$extend.$^\wedge$
- **The singleton**

```
Everything that realizes Singleton is one object
```

```
\(\text{realize.\{Singleton\}\{\{object\}\}}\)
```
Hierarchies

- Hierarchy of classes is a set that contain all classes related with “extend” relationship
- If $C$ is a class
  - $\{C\}$
  - $\{C\}$
  - $\{C\}$
  - $H_C$ 

$\subseteq \{C\}$
CLASS HIERARCHY

There is something that includes C.

D is extended by nothing. There is something that includes D.

A is realized by nothing and extends nothing. There is something that includes A.

Every H_c is equivalent to something that is included by something that includes C.

D <<final>>

H_c <<include{C} (include{C})

{D} <<extend{D} (include{D})

{A} <<realize{A} (include{A})

C <<include{C}
THE MODEL OF PART-WHOLE

- Modeled by a “have” role
  - extendéhavemhave
  - Implies that have-éextend-мhave-
  - if between two classes C and D exists chain of “extend” roles: \{C\}м$extend$....$extend$.\{D\}
  - then it can be inferred that: $have^{-}.\{C\}м$have^{-}.\{D\}
  - That means that everything that is had by C is had also by D
  - It can be interpreted as inheritance
Every object that materializes class C has at most two objects that realize class D.
**Parts of class and parts of object**

- C is a class.
- \( +C_M() \) is a method that is had by C.
- \( +C_M() = F_{CM} \)
- \( -C_P \) is an attribute that is had by C.
- \( O \) materializes C.
- \( O \) is an object.
- \( X \) is an object that is had by O and that fills \( C_P \).
- \( X \) fills \( C_M \).
- \( O \) fills \( C_M \).
- \( O \) fills \( C_P \).
EXTENDING FUNCTION NAMES TO SUPERIMPOSITIONS OF SIGNATURES AND CLASSES
CALL AND CREATE

Every function that implements a method (that is identified by M and had by C) that is had by an object that materializes C creates an object that materializes D.
THE PRACTICAL PROBLEMS

- Inconsistencies between requirements, system design, test cases and the source code that appear in the process of software development.
- Violations of design constraints and the existing architectural style found in the source code.
- The effects of changes on the reliability (traceability of the system) of the system and analyze the risks of a change.
**DESIGN PATTERNS**

- Why we want to formalize Design Pattern?
  - We can detect them in existing code
  - We can check if there are no violations of DP
  - We can reuse them – they become language independent
  - With CNL we can use the natural language as a design-pattern language
Adapter design pattern (often referred to as the wrapper pattern or simply a wrapper) is a design pattern that translates one interface for a class into a compatible interface. An adapter allows classes to work together that normally could not because of incompatible interfaces, by providing its interface to clients while using the original interface. The adapter translates calls to its interface into calls to the original interface, and the amount of code necessary to do this is typically small (...)

ADAPTER
Everything that implements something that is identified by Request and is had by Adapter) that had something that materializes Adapter is call by something that implements something that is identified by SpecificRequest and is had by Adaptee.
FUTURE WORK

- The Tool – Ontology Aided Software Engineering (OASE)

- Other Ontologies in SW:
  - Requirement specification
  - Run time testing - contracts
DISCUSSION

...