univaq lecture series:
Software Language Engineering

meets
Model-driven Engineering

Ralf Lämmel
Software Languages Team
University of Koblenz-Landau
http://www.softlang.org/
univaq lecture series

• Tue May 23, 9:30am–11:30am
  
  Internal DSLs

• Wed May 24, 11:30am–1:30pm
  
  External DSLs

• Tue Jun 6, 9:30am–11:30am
  
  Metaprogramming

• Wed Jun 7, 11:30am–1:30pm
  
  Megamodeling
A quick introduction to domain-specific languages (DSLs)
A DSL for finite state machines (FSMs)

A FSM for a ‘turnstile’ in a metro system

Imagine the FSM language (FSML) to have started on the black/whiteboard a long time ago.
Concepts of FSML illustrated for the turnstile FSM

States
locked  The turnstile is locked. No passenger is allowed to pass.
unlocked The turnstile is unlocked. A passenger may pass.
exception A problem has occurred and metro personnel needs to intervene.

Events
ticket  A passenger inserts a ticket into the card reader.
pass A passenger passes the turnstile as noticed by a sensor.
mute Metro personnel turns off alarm after exception.
release Metro personnel turns on normal operation again.

Actions
collect The ticket is collected by the card reader.
eject The ticket is ejected by the card reader.
alarm An alarm is turned on and metro personnel is requested.

Transitions
Semantics (I/O behavior) of FSML illustrated for the turnstile FSM

Input (= sequence of events)

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ticket</td>
<td>A ticket is inserted. (The turnstile is unlocked, thus.)</td>
</tr>
<tr>
<td>ticket</td>
<td>Another ticket is inserted. (The superfluous ticket is ejected.)</td>
</tr>
<tr>
<td>pass</td>
<td>Someone passes the turnstile. (This is Ok.)</td>
</tr>
<tr>
<td>pass</td>
<td>Someone passes the turnstile. (This triggers alarm.)</td>
</tr>
</tbody>
</table>

Output (= sequence of actions)

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect</td>
<td>The inserted ticket is collected.</td>
</tr>
<tr>
<td>eject</td>
<td>A ticket inserted in unlocked state is ejected.</td>
</tr>
<tr>
<td>alarm</td>
<td>An attempt to pass in locked state triggers alarm.</td>
</tr>
</tbody>
</table>
DSL implementation in different ‘styles’

• **External DSL:**
  Designated parser, checker, interpreter, compiler

• **Internal DSL:**
  Implementation as library using host language features

Our initial focus

N.B.: This is a gross oversimplification. There are options or hybrids using extensible languages, extensible compilers, metaprogramming systems, and language workbenches.
Online resources

YAS’ GitHub repository contains all code.
YAS (Yet Another SLR (Software Language Repository))
http://www.softlang.org/yas
See here specifically:
https://github.com/softlang/yas/tree/master/languages/FSML
Subdirectories Java and Python

The Software Languages Book
http://www.softlang.org/book
See Chapter 2 in particular.
Internal DSL style
We are going to do here ...

**Internal DSL style**

with ![Java logo](https://www.oracle.com/java/technologies/) and ![Python logo](https://www.python.org/) libraries

N.B.: If we were using C++, Scheme, Haskell, or others for internal DSL implementation, additional or different techniques could or should be leveraged, e.g., operator overloading, macros, or templates.
turnstile = new Fsm();
State s = new State();
s.setStateid("locked");
s.setInitial(true);
turnstile.getStates().add(s);
s = new State();
s.setStateid("unlocked");
turnstile.getStates().add(s);
s = new State();
s.setStateid("exception");
turnstile.getStates().add(s);
Transition t = new Transition();
t.setSource("locked");
t.setEvent("ticket");
t.setAction("collect");
t.setTarget("unlocked");
turnstile.getTransitions().add(t);
t = new Transition();
... add more transitions ...

‘Imperative’
Java API

N.B.: Arguably this was common style until perhaps 200?.. Even today, this sort of code may still be written.
Java API with functional constructors

```java
turnstile = new Fsm();
turnstile.getStates().add(new State("locked", true));
turnstile.getStates().add(new State("unlocked"));
turnstile.getStates().add(new State("exception"));
turnstile.getTransitions().add(new Transition("locked", "ticket", "collect", "unlocked"));
turnstile.getTransitions().add(new Transition("locked", "pass", "alarm", "exception"));
... add more transitions ...
```

N.B.: Functional constructors have been used by C++ and Java et al. programmers for a long time, but they are insufficient to avoid repetitive code and to hide the internal representation.
public class Fsm {
    private List<State> states = new LinkedList<>();
    private List<Transition> transitions = new LinkedList<>();
    public List<State> getStates() { return states; }
    public List<Transition> getTransitions() { return transitions; }
}

public class State {
    private String id;
    private boolean initial;
    public String getStateid() { return id; }
    public void setStateid(String state) { this.id = state; }
    public boolean isInitialState() { return initial; }
    public void setInitialState(boolean initial) { this.initial = initial; }
    public State() {
    }
    public State(String id) { this.id = id; }
    public State(String id, boolean initial) { this.id = id; this.initial = initial; }
}

public class Transition {
    private String source;
    private String event;
    private String action;
    }
private String id;
private boolean initial;
public String getStateid() { return id; }
public void setStateid(String state) { this.id = state; }
public boolean isInitial() { return initial; }
public void setInitial(boolean initial) { this.initial = initial; }

public State() {}
public State(String id) { this.id = id; }
public State(String id, boolean initial) { this.id = id; this.initial = initial; }

public class Transition {
    private String source;
    private String event;
    private String action;
    private String target;
    ... getters and setters ...
    public Transition() {}
    public Transition(String source, String event, String action, String target) {
        this.source = source;
        this.event = event;
        this.action = action;
        this.target = target;
    }
}

© 2017 Ralf Lämmel, Software Languages Team http://www.softlang.org/, University of Koblenz-Landau. All rights reserved.
Use of a fluent API in Java

Fsm turnstile =
    fsm()
    .addState("locked")
    .addTransition("ticket", "collect", "unlocked")
    .addTransition("pass", "alarm", "exception")
    .addState("unlocked")
    .addTransition("ticket", "eject", "unlocked")
    .addTransition("pass", null, "locked")
    .addState("exception")
    .addTransition("ticket", "eject", "exception")
    .addTransition("pass", null, "exception")
    .addTransition("mute", null, "exception")
    .addTransition("release", null, "locked");

Leveraged techniques:
• Factory methods
• Method chaining
• Implicit parameters
• Conventions (defaults)

N.B.: The current state is maintained along the way.
The state declared first is assumed to be the initial one.
The representation is not revealed—no constructors are used.
Use of a **fluent API** in Python

```python
turnstile = Fsm() \
    .addState("locked") \
    .addTransition("ticket", "collect", "unlocked") \
    .addTransition("pass", "alarm", "exception") \
    .addState("unlocked") \
    .addTransition("ticket", "eject", "unlocked") \
    .addTransition("pass", None, "locked") \
    .addState("exception") \
    .addTransition("ticket", "eject", "exception") \
    .addTransition("pass", None, "exception") \
    .addTransition("mute", None, "exception") \
    .addTransition("release", None, "locked")
```

**Leveraged techniques:**
- Factory methods
- Method chaining
- Implicit parameters
- Conventions (defaults)

N.B.: If we were using C++, Scheme, Haskell, or others for internal DSL implementation, additional or different techniques could or should be leveraged, e.g., operator overloading, macros, or templates.
Definition of **fluent API** in Java

```java
public interface Fsm {
    public Fsm addState(String state);
    public Fsm addTransition(String event, String action, String target);
    public String getInitial();
    public ActionStatePair makeTransition(String state, String event);
}
```

```java
public class ActionStatePair {
    public String action;
    public String state;
}
```

N.B.: This interface does not expose the internal representation. The interface does not just cover fluent construction; it also covers ‘observation’ of the opaque representation.
public class FsmlImpl implements Fsm {
    private String initial; // the initial state
    private String current; // the "current" state
    // A cascaded map for maintaining states and transitions
    private HashMap<String, HashMap<String, ActionStatePair>> fsm =
        new HashMap<>();
    private FsmlImpl() { }
    // Construct FSM object
    public static Fsm fsm() { return new FsmlImpl(); }
    // Add state and set it as current state
    public Fsm addState(String id) {
        // First state is initial state
        if (initial == null) initial = id;
        // Remember state for subsequent transitions
        this.current = id;
        if (fsm.containsKey(id)) throw new FsmlDistinctIdsException();
        fsm.put(id, new HashMap<String, ActionStatePair>());
        return this;
    }
    // Add transition for current state
    public Fsm addTransition(String event, String action, String target) { ...
if (fsm.containsKey(id)) throw new FsmlDistinctIdsException();
fsm.put(id, new HashMap<String, ActionStatePair>());
return this;
}

// Add transition for current state
public Fsm addTransition(String event, String action, String target) {
    if (fsm.get(current).containsKey(event)) throw new FsmlDeterministismException();
    ActionStatePair pair = new ActionStatePair();
pair.action = action;
pair.state = target;
fsm.get(current).put(event, pair);
    return this;
}

// Getter for initial state
public String getInitial() {
    return initial;
}

// Make transition
public ActionStatePair makeTransition(String state, String event) {
    if (!fsm.containsKey(state)) throw new FsmlResolutionException();
    if (!fsm.get(state).containsKey(event)) throw new FsmlInfeasibleEventException();
    return fsm.get(state).get(event);
}
public class FluentTest {

    private static final String[] input = {
        "ticket", "ticket", "pass", "pass", "ticket", "mute", "release"};

    private static final String[] output = {
        "collect", "eject", "alarm", "eject"};

    @Test
    public void runSample() {
        assertArrayEquals(output, run(Sample.turnstile, input));
    }
}

N.B.: This is how a Java programmer (a DSL user) would document a use case of a specific FSM (and validate intuitions).
An interpreter (a ‘simulator’) in Java

```java
public class FsmlInterpreter {
    public static String[] run(Fsm fsm, String[] input) {
        ArrayList<String> output = new ArrayList<>();
        String state = fsm.getInitial();
        for (String event : input) {
            ActionStatePair pair = fsm.makeTransition(state, event);
            if (pair.action != null) output.add(pair.action);
            state = pair.state;
        }
        return output.toArray(new String[output.size()]);
    }
}
```

N.B.: The interpreter essentially models the dynamic semantics of FSML. This is a non-interactive interpreter. In practice, an interactive DSL implementation may be required.
Implementation of **fluent API** in Python

```python
class Fsm():
    def __init__(self):
        self.fsm = defaultdict(list)
        self.current = None
    def addState(self, id):
        return self.addStateNoDefault(self.current is None, id)
    def addStateNoDefault(self, initial, id):
        if id in self.fsm[id]: raise FsmlDistinctIdsException;
        self.stateObject = dict()
        self.stateObject['transitions'] = defaultdict(list)
        self.stateObject['initial'] = initial
        self.fsm[id] += [self.stateObject]
        self.current = id
        return self
    def addTransition(self, event, action, target):
        if event in self.stateObject['transitions']: raise FsmlDeterminismException;
        self.stateObject['transitions'][event] += [
            [(action, self.current if target is None else target)]
        return self
```

N.B.: no high-level API is provided for ‘observation’; one would access the dictionary directly.
def run(fsm, input):
    # Determine initial state
    for id, [decl] in fsm.iteritems():
        if decl["initial"]:
            current = decl
            break
    # Consume input; produce output
    output = []
    while input:
        event = input.pop(0)
        if event not in current["transitions"]: raise FsmlInfeasibleEventException
        else:
            [(action, target)] = current["transitions"][event]
            if action is not None: output.append(action)
            if target not in fsm: raise FsmlResolutionException
            [current] = fsm[target]
    return output

N.B.: When compared to the Java-based interpreter, we access directly the presentation.
‘Minimum’ DSL implementation

✓ Syntax (fluent API for internal DSL)

✓ (Dynamic) semantics (e.g., by means on an interpreter)
  - Well-formedness / -typedness (aka static semantics)

N.B.: Just like the interpreter, we implement a ‘well-formedness checker’ as functionality on top of (the API for) the internal DSL representation. (We could use a constraint language such as OCL.)
Well-formedness of FSMs

distinctStateIds   The state ids of the state declarations must be distinct.
singleInitialState   An FSM must have exactly one initial state.
deterministicTransitions   The events must be distinct per state.
resolvableTargetStates   The target state of each transition must be declared.
reachableStates   All states must be reachable from the initial state.

resolutionNotOk = \[ \begin{array}{l}
Fsm() \\
  .addState("stateA") \\
  .addTransition("event1", "action1", "stateB") \\
  .addTransition("event2", "action2", "stateC") \\
  .addState("stateB")
\end{array} \]

N.B.: This sample violates resolvableTargetStates.

N.B.: a violated resolvableTargetStates can (should) be detected even before running an FSM on a specific input.

© 2017 Ralf Lämmel, Software Languages Team http://www.softlang.org/, University of Koblenz-Landau. All rights reserved.
Python-based well-formedness checker

```python
def ok(fsm):
    for fun in [
        distinctStateIds,
        singleInitialState,
        deterministicTransitions,
        resolvableTargetStates,
        reachableStates ]:
        fun(fsm)

    N.B.: Violations of distinctStateIds and deterministicTransitions can be detected during construction, but we may need explicit checks if we also accommodate ‘serialization’.

def distinctStateIds(fsm):
    for state, decls in fsm.iteritems():
        if not len(decls) == 1: raise FsmlDistinctIdsException()

def singleInitialState(fsm):
    initials = [initial for initial, [decl] in fsm.iteritems() if decl["initial"]]
    if not len(initials) == 1: raise FsmlSingleInitialStateException()

def deterministicTransitions(fsm):
    for state, [decl] in fsm.iteritems():
        for event, transitions in decl["transitions"].iteritems():
            if not len(transitions) == 1: raise FsmlDeterminismException()

def resolvableTargetStates(fsm): ...
```

26
def singleInitialState(fsm):
    initials = [initial for initial, [decl] in fsm.iteritems() if decl["initial"]]
    if not len(initials) == 1: raise FsmlSingleInitialStateException()

def deterministicTransitions(fsm):
    for state, [decl] in fsm.iteritems():
        for event, transitions in decl["transitions"].iteritems():
            if not len(transitions) == 1: raise FsmlDeterminismException()

def resolvableTargetStates(fsm):
    for _, [decl] in fsm.iteritems():
        for _, transitions in decl["transitions"].iteritems():
            for (_, target) in transitions:
                if not target in fsm: raise FsmlResolutionException()

def reachableStates(fsm):
    for initial, [decl] in fsm.iteritems():
        if decl["initial"]: 
            reachables = set([initial])
            chaseStates(initial, fsm, reachables)
        if not reachables == set(fsm.keys()): raise FsmlReachabilityException()

# Helper for recursive closure of reachable states
def chaseStates(source, fsm, states): ...
Code generation for DSLs
Problem statement

Model of finite state machine → Program generator → C code for finite state machine

N.B.: The C code should run on some ‘target hardware’.
An FSM for a turnstile in a metro

- **States** (nodes): locked, unlocked, exception
- **Events**: ticket, pass, release, mute
- **Actions**: collect, eject, alarm
- **Transitions** (edges)
enum State { LOCKED, UNLOCKED, EXCEPTION, UNDEFINED };
enum State initial = LOCKED;
enum Event { TICKET, RELEASE, MUTE, PASS };
void alarm() {}  
void eject() {}  
void collect() {}  
enum State next(enum State s, enum Event e) {
    switch(s) {
    case LOCKED:
        switch(e) {
            case TICKET: collect(); return UNLOCKED;
            case PASS: alarm(); return EXCEPTION;
            default: return UNDEFINED;
        }
    case UNLOCKED:
        switch(e) {
            case TICKET: eject(); return UNLOCKED;
            case PASS: return LOCKED;
            default: return UNDEFINED;
        }
    case EXCEPTION:
        switch(e) {
            case TICKET: eject(); return EXCEPTION;
            case PASS: return EXCEPTION;
            case MUTE: return EXCEPTION;
            case RELEASE: return LOCKED;
            default: return UNDEFINED;
        }
    default: return UNDEFINED;
    }
}
How to design and implement the program generator?

- How to emit/produce/... the output—the actual C code?
- We are going to use `template processing`!
- What data structure (model) to use for code generation?
- Do we map FSML `objects’ directly to C code?
  - No — the objects are not aligned with the wanted C code.
- Do we build an `intermediate representation`?
  - Yes — we use a model close to the wanted C code.
‘Hello, World’ of template processing

Source: https://github.com/antlr/stringtemplate4/blob/master/doc/introduction.md

**Templates** contain text and references to names:

Hello, `<name>`

**Actual parameters can be passed to a rendering process**

```java
import org.stringtemplate.v4.*;
...
ST hello = new ST("Hello, <name>");
hello.add("name", "World");
System.out.println(hello.render());
```

**The rendering result is text.**

Hello, World

N.B.: There exist many template processors. We use StringTemplate [http://www.stringtemplate.org/](http://www.stringtemplate.org/) which is a mainstream option for the Java platform. StringTemplate can also be used with other languages.
Design and implement the code generator

Model of finite state machine → Program generator → C code for finite state machine

Define templates for the output → Map model to intermediate representation → Instantiation (rendering)
// Main template for complete source code

main(states, initial, events, actions, tgroups) ::= << ... >>

// Template for functions that implement actions

action(a) ::= "..."

// switch-case for transitions grouped by source state

tgroup(g) ::= <<...>>

// case for a specific transition

transition(t) ::= <%. ...%>

N.B.: Because the templates have names, they can refer to each other. This is called a template group in StringTemplate.
main(states, initial, events, actions, tgroups) ::= <<
enum State { <states; format="upper", separator="", "> };
enum State initial = <initial; format="upper">;
enum Event { <events; format="upper", separator="", "> };
<actions:action(); format="lower", separator="\n">enum State next(enum State s, enum Event e) {
    switch(s) {
    <tgroups:tgroup(); separator="\n">                default: return UNDEFINED;
    }
}>>

action(a) ::= "void <a>() { }"

tgroup(g) ::= <<
    case <g.stateid; format="upper">:
        switch(e) {
            <g.ts:transition(); separator="\n">          default: return UNDEFINED;
        }>>

transition(t) ::= <<%
    case <t.event; format="upper">:
    <if(t.action)<<t.action; format="lower">(); <endif>
    return <t.target; format="upper">;>%
A model for finite state machines

```java
public class Fsm {
    public List<State> states = new LinkedList<>();
    public List<Transition> transitions = new LinkedList<>();
}

public class State {
    public String stateid;
    public boolean initial;
}

public class Transition {
    public String source;
    public String event;
    public String action;
    public String target;
}

N.B.: In reality, we use private fields, public getters, and possibly functional constructors instead of public fields. In fact, we may also assume a fluent API.
import org.stringtemplate.v4.ST;
import org.stringtemplate.v4.STGroup;
import org.stringtemplate.v4.STGroupFile;
import org.stringtemplate.v4.StringRenderer;
import java.io.File;
import java.util.*;

public class FsmICGenerator {
    private static class TGroup {
        public String stateid;
        public List<Transition> ts;
    }

    public static String generate(Fsm fsm) {
        // Build list of states with extra "UNDEFINED"
        List<String> states = ...
        // Build set of events
        Set<String> events = ...
        // Build set of actions
        Set<String> actions = ...
        // Group transitions by state
        List<TGroup> tgroups = ...

        // Load template group and retrieve top-level template
        STGroup group = new STGroupFile(... + "Fsm.stg");
        group.registerRenderer(String.class, new StringRenderer());
        ST main = group.getInstanceOf("main");
        // Set template parameters and render
        main.add("states", states);
        main.add("initial", fsm.getInitial());
        main.add("events", events);
        main.add("actions", actions);
        main.add("tgroups", tgroups);
        return main.render();
    }
}

© 2017 Halil Lämmel, Software Languages Team http://www.sottling.org/. University of Koblenz-Landau. All rights reserved.
// Build list of states with extra "UNDEFINED"
List<String> states = new LinkedList<>();
for (State s : fsm.getStates()) states.add(s.getStateid());
states.add("UNDEFINED");
// Build set of events
Set<String> events = new HashSet<>();
for (Transition t : fsm.getTransitions()) events.add(t.getEvent());
// Build set of actions
Set<String> actions = new HashSet<>();
for (Transition t : fsm.getTransitions())
    if (t.getAction() != null) actions.add(t.getAction());
// Group transitions by state
List<TGroup> tgroups = new LinkedList<>();
for (State s : fsm.getStates()) {
    TGroup tg = new TGroup();
    tg.stateid = s.getStateid();
    tg.ts = new LinkedList<>();
    for (Transition t : fsm.getTransitions())
        if (tg.stateid == t.getSource()) tg.ts.add(t);
    tgroups.add(tg);
}

N.B.: We aim at a strict separation of model and view and thus, we do not try to ‘compute’ anything in the template.
## StringTemplate’s metanotation

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tmpl(p) ::= &quot;...&quot;</code></td>
<td>Define a template <code>tmpl</code> with parameter <code>p</code></td>
</tr>
<tr>
<td><code>tmpl(p) ::= &lt;&lt;...&gt;&gt;</code></td>
<td>Multiline template with indentation and linebreaks</td>
</tr>
<tr>
<td><code>tmpl(p) ::= &lt;%/...%&gt;</code></td>
<td>Ignore indentation and linebreaks</td>
</tr>
<tr>
<td><code>&lt;p&gt;</code></td>
<td>Refer to parameter <code>p</code></td>
</tr>
<tr>
<td><code>&lt;p; format=&quot;upper&quot;&gt;</code></td>
<td>Format in uppercase</td>
</tr>
<tr>
<td><code>&lt;p; separator=&quot;&quot;, &quot;&quot;&gt;</code></td>
<td>Separate elements of collection by comma</td>
</tr>
<tr>
<td><code>&lt;p:tmpl()&gt;</code></td>
<td>Apply <code>tmpl</code> on (each element of) <code>p</code></td>
</tr>
<tr>
<td><code>&lt;p.x&gt;</code></td>
<td>Refer to property <code>x</code> of <code>p</code></td>
</tr>
<tr>
<td><code>&lt;if(p.x)&gt;...&lt;endif&gt;</code></td>
<td>Include text only if <code>p.x</code> is true or not NULL</td>
</tr>
</tbody>
</table>
Template processing issues

- Escaping template metanotation
- Spacing, indentation, and linebreaks
- Underlying use of reflection (typically)
- Formatting (e.g., upper- versus lowercase)
- Mapping the model to the template parameters
- Idiosyncrasies of particular template metanotation
A Jinja2 template for use with Python

```python
enum State { {{states|join(’, ’)|upper()}} };
enum State initial = {{initial|upper}};
enum Event { {{events|join(’, ’)|upper()}} };

{% for a in actions %}void {{a}}() { }{% endfor %}

enum State next(enum State s, enum Event e) {
    switch(s) { {% for (s, ts) in transitions %}
        case {{s|upper()}}:
            switch(e) { {% for (e, a, t) in ts %}
                case {{e|upper()}}: {% if a %}{{a}}() ; {%endif%}return {{t|upper()}};{% endif %}{% endfor %}
                default: return UNDEFINED;
            }{% endfor %}
        default: return UNDEFINED;
    }{% endfor %}
}
```

N.B.: This approach slightly differs from the one shown before. In particular, we use a for-loop. Also, we use a single template as opposed to a group.
Grammarware speak

Input text → Parsing → AST or CST → Translation → IR → Formatting → Output text

versus

Modelware speak

Input text → Text to model trans. → Model → Model to model trans. → Model’ → Model to text trans. → Output text

N.B.:
- IR = Intermediate Representation (an AST closer to output).
- We massage FSML ‘models’ to appeal to code generation patterns.
- We could also translate FSML ‘models’ directly into C ‘models’.
Comparison with data flow in a compiler

Grammar

Rules for type system etc.

Rules for code generation

Source code

Parser

Parse tree

Semantic analysis

Enriched parse tree

Code generator

Machine code
External DSL style
Let’s have a **textual syntax** for FSML: the finite state machine (FSM) language

*An FSM for a turnstile in a metro system*

- **States** (nodes): locked, unlocked, exception
- **Events**: ticket, pass, release, mute
- **Actions**: collect, eject, alarm
- **Transitions** (edges)
Proposal for a textual syntax for FSML

```
initial state locked {
  ticket/collect → unlocked;
  pass/alarm → exception;
}

state unlocked {
  ticket/eject;
  pass → locked;
}

state exception {
  ticket/eject;
  pass;
  mute;
  release → locked;
}
```

N.B.: DSL design
(concrete or abstract syntax design)
commences in a sample-driven manner.
DSL implementation in different ‘styles’

- **External DSL:**
  Designated parser, checker, interpreter, compiler

- **Internal DSL:**
  Implementation as library using host language features

N.B.: This is a gross oversimplification. There are options or hybrids using extensible languages, extensible compilers, metaprogramming systems, and language workbenches.
We are going to do here …

External DSL style

with ANTLR and Java

N.B.: We are committing to a particular parser generator (ANTLR). We could also be using hand-written parsers, parser combinators, and model-to-text technologies. ANTLR, by itself, also serves other ‘target’ languages, e.g., Python.
Grammar-based concrete syntax definition

fsm : state+ EOF ;
state : 'initial'? 'state' stateid '{' transition* '}' ;
transition : event ('/' action)? ('->' target=stateid)? ';' ;
stateid : NAME ;
event : NAME ;
action : NAME ;
NAME : ('a'..'z'|'A'..'Z')+ ;

N.B.: Action and target state are optional.

N.B.: This is essentially Extended Backus Naur Form. ‘?’ for options, ‘*’/‘+’ for lists, etc. Well, we use the specific grammar notation of ANTLR.
ANTLR+Java-based syntax checker

grammar Fsml;
@header {package org.softlang.fsml;}

fsm : state+ EOF;
state : 'initial'? 'state' stateid '{' transition* '}' ;
transition : event ('/' action)? ('-->' target=stateid)? ';;'
stateid : NAME ;
event : NAME ;
action : NAME ;
NAME : ('a'..'z'|'A'..'Z')+ ;
WS : [ \t\n\r]+ -> skip ;

N.B.: A grammar is almost an effective definition of a syntax checker. We need ‘pragmas’ and driver code (see next slide).
Data flow in (scannerfull) parsing

Character sequence → Scanner (Lexer) → Token sequence → Parser → CST/AST

Lexical Grammar defines
Context-free Grammar defines

We don’t care too much about it.

How would we operate on a CST? How do we make ANTLR give us an AST?
Sample FSM

initial state x {
    ping / pong -> x
}

Token stream

terminal("initial"),
terminal("state"),
NAME("x"),
terminal("{"),
NAME("ping"),
terminal("/"),
NAME("pong"),
terminal("->"),
NAME("x"),
terminal("{"})
CST for sample FSM

```
fsm : {state}*;
```

```
state : {'initial'}? 'state' id '{' {transition} '*' '}'
```

```
{''initial''}? 'state' 'x' '{' {transition} '*' '}'
```

```
transition : input {'/' action}? {'->' id}? ';
```

```
'ping' {'/' action}? {'->' id}? ';
```

```
'pong' {'/' action}? {'->' id}? ';
```

```
'x'
```

N.B.:
- The leaf infos represent the input string.
- The inner nodes correspond to rule applications.

© 2017 Ralf Lämmel, Software Languages Team http://www.softlang.org/, University of Koblenz-Landau. All rights reserved.
Parser generation with ANTLR

MyLang.g4 → org.antlr.v4.Tool → MyLangParser.java
MyLangLexer.java
MyLangListener.java

MySample → MyLangDriver.java → Parse tree

Code generation-time data flow
Code-level dependencies
Runtime data flow

Parser generator (ANTLR)
Programmer-provided artifacts
Generated code

© 2017 Ralf Lämmel, Software Languages Team http://www.softlang.org/, University of Koblenz-Landau. All rights reserved.
Java code **driving** the syntax checker based on generated classes *Parser* and *Lexer*

```java
public class FsmlSyntaxChecker {
    public static void main(String[] args) throws IOException {
        FsmlParser parser =
            new FsmlParser(
                new CommonTokenStream(
                    new FsmlLexer(
                        new ANTLRFileStream(args[0])))
            );
        parser.fsm();
        System.exit(parser.numberOfSyntaxErrors()
            - Integer.parseInt(args[1]));
    }
}
```

We assume a command-line interface for running positive and negative test cases.

**N.B.: This is boilerplate code.**
Makefile

for **building** and **testing** the syntax checker

```bash
cp = -cp :./.././lib/Java/antlr-4.5.3-complete.jar
antlr = java ${cp} org.antlr.v4.Tool -o org/softlang/fsml
fsmlSyntaxChecker = java ${cp} org.softlang.fsml.FsmlSyntaxChecker

all:
    make generate
    make compile
    make test

generate:
    ${antlr} Fsml.g4

compile:
    javac ${cp} org/softlang/fsml/*.*.java

test:
    ${fsmlSyntaxChecker} ./sample.fsml 0
    ${fsmlSyntaxChecker} ./tests/syntaxError.fsml 1
```

N.B.: Automation (build management and testing) is crucial in DSL implementation to deal with experimentation and evolution.
From checking to parsing

N.B.:
- AST/CST = abstract/concrete syntax tree
- CSTs are served by a parsing technology like ANTLR.
- ASTs are modeled by language-specific object models.
Abstraction based on walking the CST, in fact, based on ‘listening’ to a walk

N.B.: These are the first few steps of a depth-first walk.
An ANTLR listener for abstraction

```java
public class FsmlToObjects extends FsmlBaseListener {
    private Fsm fs;
    private State current;
    public Fsm getFsm() { return fs; }
    @Override public void enterFsm(FsmlParser.FsmContext ctx) {
        fs = new Fsm();
    }
    @Override public void enterState(FsmlParser.StateContext ctx) {
        current = new State();
        current.setStateId(ctx.stateid().getText());
        fs.getStates().add(current);
    }
    @Override public void enterTransition(FsmlParser.TransitionContext ctx) {
        Transition t = new Transition();
    }
    ...
}
```

N.B.:
- There are ‘events’ for entering (and leaving) nonterminals.
- The listener applies to the grammar for syntax checking.
public class FsmlToObjects extends FsmlBaseListener {
    private Fsm fsm;
    private State current;
    public Fsm getFsm() { return fsm; }
    @Override public void enterFsm(FsmlParser.FsmContext ctx) {
        fsm = new Fsm();
    }
    @Override public void enterState(FsmlParser.StateContext ctx) {
        current = new State();
        current.setStateid(ctx.stateid().getText());
        fsm.getStates().add(current);
    }
    @Override public void enterTransition(FsmlParser.TransitionContext ctx) {
        Transition t = new Transition();
        fsm.getTransitions().add(t);
        t.setSource(current.getStateid());
        t.setEvent(ctx.event().getText());
        if (ctx.action() != null) t.setAction(ctx.action().getText());
        t.setTarget(ctx.target != null ? ctx.target.getText() : current.getStateid());
    }
}
Java code driving the parser

```java
public Fsm textToObjects(String filename) throws IOException {
    FsmlParser parser = new FsmlParser(
        new CommonTokenStream(
            new FsmlLexer(
                new ANTLRFileStream(filename))));
    ParseTree tree = parser.fsm();
    assertEquals(0, parser.getNumberOfSyntaxErrors());
    FsmlToObjects listener = new FsmlToObjects();
    ParseTreeWalker walker = new ParseTreeWalker();
    walker.walk(listener, tree);
    return listener.getFsm();
}
```

N.B.: This is essentially the same code as for the earlier checker—except that we now walk the parse tree.
‘Minimum’ DSL implementation

• Syntax:
  • Object model for abstract syntax
  • **Parser based on grammar for concrete (textual) syntax**

• (Dynamic) semantics:
  • Interpreter operating on abstract syntax (object model)

• Well-formedness/typedness (aka static semantics):
  • Checker operating on abstract syntax (object model)

N.B.: Everything **not in bold face** can be implemented in the same way as in a DSL implementation in internal style. (Clearly, we only consider here a particular approach to DSL implementation.)
Metaprogramming

(A comparison of approaches)

Acknowledgement: This is joint work with Simon Schauß, Johannes Härtel, Kevin Klein, Wojciech Kwasnik (all SoftLang), and Thorsten Berger (Chalmers).
(Comparison of) metaprogramming approaches

- A **metaprogram** consumes or produces (or analyzes or manipulates) programs.

- A **DSL implementation** (both in internal and external style) is a metaprogram.

- A **metaprogramming approach** is an approach to developing metaprograms, subject to designated language, library, and other tool support.

- **MetaLib** is a project for the comparison for metaprogramming approaches. In fact, it is a **chrestomathy**.
The Finite State Machine Language (FSML)
Visual FSML notation

```
exception -> locked: release
locked  -> unlocked: ticket/collect
unlocked  -> locked
```

Actions:
- mute
- pass
- ticket/eject
- release
- pass/alarm
- ticket/collect
- pass
initial state locked {
    ticket/collect $\rightarrow$ unlocked;
    pass/alarm $\rightarrow$ exception;
}

state unlocked {
    ticket/eject;
    pass $\rightarrow$ locked;
}

state exception {
    ticket/eject;
    pass;
    mute;
    release $\rightarrow$ locked;
}
initial state locked {
    ticket/collect → unlocked;
    pass/alarm → exception;
}

state unlocked {
    ticket/eject;
    pass → locked;
}

state exception {
    ticket/eject;
    pass;
    mute;
    release → locked;
}
Grammar of textual notation

fsm : {state}* ;
state : {'initial'}? 'state' stateid '{' {transition}* '}';
transition : event {'/' action}? {'->' stateid}? ';;'
stateid : name ;
event : name ;
action : name ;
Signature of abstract syntax

type fsm = state* ;
type state = initial × stateid × transition* ;
type initial = boolean ;
type transition = event × action? × stateid ;
type stateid = string ;
type event = string ;
type action = string ;
Metamodel of abstract syntax

class fsm {
    part states : state* ;
}
class state {
    value initial : boolean ;
    value stateid : string ;
    part transitions : transition* ;
}
class transition {
    value event : string ;
    value action : string? ;
    reference target : state ;
}
Small-step operational semantics

\[ \langle \ldots, \langle b, x, \langle \ldots, \langle e, \langle a \rangle, x' \rangle, \ldots \rangle, \ldots \rangle, \ldots \rangle \vdash \langle x, e \rangle \rightarrow \langle x', \langle a \rangle \rangle \quad \text{[action]} \]

\[ \langle \ldots, \langle b, x, \langle \ldots, \langle e, \langle \rangle \rangle, x' \rangle, \ldots \rangle, \ldots \rangle \vdash \langle x, e \rangle \rightarrow \langle x', \langle \rangle \rangle \quad \text{[no-action]} \]
Well-formedness
(a violation thereof)

initial state stateA { event1/action1 → stateB; }
state stateB { }
state stateC { }
Generated C code

```
enum State {LOCKED,UNLOCKED,EXCEPTION,UNDEFINED};
enum State initial = LOCKED;
enum Event {TICKET,RELEASE,MUTE,PASS};
void alarm() {} 
void eject() {} 
void collect() {} 
enum State next(enum State s, enum Event e) {
    switch(s) {
        case LOCKED: 
            switch(e) {
                case TICKET: collect(); return UNLOCKED;
                case PASS: alarm(); return EXCEPTION;
                default: return UNDEFINED;
            }
        case UNLOCKED: ...
        case EXCEPTION: ...
        default: return UNDEFINED;
    }
}
```
Feature model for the implementation of a DSL
Language implementation

- Syntax
- Semantics
- Ecosystem

Legend
- ⚫ Mandatory
- ○ Optional
- ▲ Or
- ▲ Alternative
- ◼ Abstract
- □ Concrete
Syntax

Abstract syntax

Concrete syntax
Scanningfull parsing
Scannerless parsing

- Character sequence
- Parser
- CST/AST

Context-free Grammar delegates to Lexical Grammar
defines

© 2017 Ralf Lämmel, Software Languages Team http://www.softlang.org/, University of Koblenz-Landau. All rights reserved.
Rename refactoring

**Input**

```plaintext
initial state locked {
    ticket / collect -> unlocked;
    pass / alarm -> exception;
}

state unlocked {
    ticket / eject;
    pass -> locked;
}
...```

**Output**

```plaintext
initial state closed {
    ticket / collect -> open;
    pass / alarm -> exception;
}

state open {
    ticket / eject;
    pass -> closed;
}
...```
Merge composition

```
initial state locked {
    ticket / collect -> unlocked;
}
state unlocked {
    ticket / eject;
    pass -> locked;
}
```

```
state locked {
    pass / alarm -> exception;
}
state exception {
    ticket / eject;
    pass;
    mute;
    release -> locked;
}
```

```
initial state locked {
    ticket / collect -> unlocked;
    pass / alarm -> exception;
}
state unlocked {
    ticket / eject;
    pass -> locked;
}
state exception {
    ticket / eject;
    pass;
    mute;
    release -> locked;
}
```
So let's approach metaprogramming (the comparison of approaches) in a chrestomathic manner.

- Community effort (for aggregation and evaluation)
- Requirement specification
- Multiplicity of languages

chrestomathy

/kreˈstəməθi/  

noun  formal

a selection of passages from an author or authors, designed to help in learning a language.

- Rich metadata
- Process management[Google]
An example of a software chrestomathy

http://rosettacode.org/wiki/Rosetta_Code

Rosetta Code

Rosetta Code is a programming chrestomathy site. The idea is to present solutions to the same task in as many different languages as possible, to demonstrate how languages are similar and different, and to aid a person with a grounding in one approach to a problem in learning another. Rosetta Code currently has 850 tasks, 198 draft tasks, and is aware of 651 languages, though we do not (and cannot) have solutions to every task in every language.
Another example of a software chrestomathy

https://101wiki.softlang.org/

The project ‘101’ is an open knowledge resource covering software technologies, languages, and concepts. 101 targets programmers, software engineers, teachers, learners, and technologists; they can leverage 101 and they are encouraged to contribute to 101.
Characteristics of a software chrestomathy

- Community effort (for aggregation and evaluation)
- Requirement specification
- Multiplicity of languages
- Infrastructural support
- Revision and access control
- Quality assurance
- Rich metadata
- Process management
Chrestomathies related to metaprogramming

We will look at two software chrestomathies that are specifically focused on being useful for learning about software languages at the level of metaprogramming (or language implementation).
MetaLib at https://softlang.github.io/metalib/

External DSL style with ANTLR and Java

<table>
<thead>
<tr>
<th>Perspectives</th>
<th>Features</th>
<th>Languages</th>
<th>Technologies</th>
<th>Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>meta</td>
<td>AST+</td>
<td>ANTLR4</td>
<td>ANTLR4</td>
<td>Boilerplate code</td>
</tr>
<tr>
<td>object</td>
<td>Abstraction</td>
<td>Java</td>
<td>JUnit</td>
<td>External DSL</td>
</tr>
<tr>
<td>test</td>
<td>Concrete syntax</td>
<td></td>
<td></td>
<td>Functional constructor</td>
</tr>
<tr>
<td></td>
<td>Scanning</td>
<td></td>
<td></td>
<td>Getter</td>
</tr>
<tr>
<td></td>
<td>Text-to-AST</td>
<td></td>
<td></td>
<td>Listener</td>
</tr>
<tr>
<td></td>
<td>Text-to-CST</td>
<td></td>
<td></td>
<td>Parser generation</td>
</tr>
</tbody>
</table>

Initial state locked {
  tickets/collect -> unlocked;
  pass/alarm -> exception;
}

State unlocked {
  tickets/eject;
  pass -> locked;
}
Model-based documentation

```json
{
  "name": "pythonExternal",
  "headline": "Python external",
  "sections": [
    {
      "features": [
        "Concrete syntax",
      ],
      "perspective": "object",
      "languages": [],
      "type": "component",
      "concepts": [
        "External DSL"
      ]
    }
  ]
}
```
Metamodel of documentation

// Documentation of contributions
class document {
  value name : string; // The name of the contribution
  value headline : string; // A one-liner explanation
  value baseuri : string; // Base URI for links
  part sections : section*; // Sections of the documentation
}

// Sections in a documentation
class section {
  value headline : string?; // Optional one-liner explanation
  part perspectives : perspective*; // Perspective of section
  value features : string*; // Features addressed by section
  value languages : string*; // Languages used
  value technologies : string*; // Technologies used
  value concepts : string*; // Concepts used
  part artifacts : artifact*; // Artifacts to be shown
}

// Perspectives of documentation
dataset perspectives {
}
abstract class perspective {
}

// Metaprogram, e.g., grammar or interpreter
class meta extends perspective {
}

// Object program in different representations
class object extends perspective {
}

// Data other than object programs, e.g., input
class data extends perspective {
}

// Build—step for functionality
class build extends perspective {
}

// Test for functionality
class test extends perspective {
}

// Capture of behavior or appearance
class capture extends perspective {
}

// Artifacts for projection
abstract class artifact {
    value link : string; // A relative URI
    value format : string; // MIME—like format type
}

// Nothing to show
class none extends artifact {

}

// All to show
// Build step for functionality
class build extends perspective { }

// Test for functionality
class test extends perspective { }

// Capture of behavior or appearance
class capture extends perspective { }

// Artifacts for projection
abstract class artifact {
    value link : string; // A relative URI
    value format : string; // MIME-like format type
}

// Nothing to show
class none extends artifact { }

// All to show
class all extends artifact { }

// A specific line range to show
class some extends artifact {
    value from : integer;
    value to : integer;
}
Megamodeling

Acknowledgement: This is joint work with Andrei Varanovich, Marcel Heinz, Johannes HärTEL (all SoftLang), and Jean-Marie Favre (University of Grenoble).
What’s a megamodel?

A model whose *model elements are models.*

The notion of model is to be interpreted broadly:

- model, metamodel, model transformation,
- program, grammar, metaprogram,
- document, schema, transformation,
- ...
An example of a megamodel:
A tombstone diagram for bootstrapping a compiler
Another example of a megamodel:
Mechanics of an ATL-based transformation
Technology modeling
(as a form of megamodeling)
Basic terminology

- Technology = Software technology, e.g.:
  - Web-application framework
  - O/R mapper

- Technology modeling:
  - an **ontological documentation** of technology
  - a form of **megamodeling**
  - also referred to as (a model of) **linguistic architecture**
Linguistic architecture of xsd.exe
1. <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
3.    name="Company">
4.    <xs:complexType>
5.      <xs:sequence>
6.        <xs:element name="Name" type="xs:string" />
7.        <xs:element name="TopLevelDepartment" type="Company" />
8.      </xs:sequence>
9.    </xs:complexType>
10.  </xs:element>
11.  </xs:complexType>
12.  <xs:complexType name="Department">
13.      <xs:sequence>
14.        <xs:element name="Manager" type="xs:string" />
15.        <xs:element name="SubDepartment" type="xs:string" />
16.        <xs:element name="Employee" type="xs:string" />
17.      </xs:sequence>
18.  </xs:complexType>
19.  <xs:element name="Company">
20.    <xs:complexType>
21.      <xs:sequence>
22.        <xs:element name="Name" type="xs:string" />
23.        <xs:element name="TopLevelDepartment" type="Company" />
24.      </xs:sequence>
25.      public partial class Company {
26.      
27.      private string nameField;
28.      
29.      private Department[] topLevelDepartment;
30.      
31.      /// <remarks/>
32.      public string Name {
33.        get {
34.          return this.nameField;
35.        }
36.        set {
37.          this.nameField = value;
38.        }
39.      }
40.      
41.      /// <remarks/>
42.      }
43.    </xs:complexType>
44.  </xs:element>
45.  </xs:complexType>
46.  </xs:schema>
Overarching research questions

- What are problems with *classic documentation*?
- How to ontologically structure and enrich documentation?
- What is the underlying *vocabulary* and *ontology*?
- How to create *value for developers* (like Stackoverflow)?
- What sort of *tool support* is necessary or helpful?
- How to actually *renarrate* such documentation?
A technology model for using EMF+Xtext+ATL for the purpose of DSML implementation
module XML import (Prelude) // Import basic vocabulary
XML : Language // Declare XML as a language entity
XSD : Language // and XSD (XML Schema), too
XSD subsetOf XML // Subset relationship on XSD and XML
xmlFile : Artifact // Declare artifact
xsdFiles : Artifact+ // Declare collection of artifacts
xmlFile elementOf XML // Assign language to artifact
gxsdFiles elementOf XSD // Assign language to artifact
xmlFile conformsTo xsdFiles // XSD–based validation
Megamodel of the EMF story

module EMF import (Prelude)
Ecore : Language // As defined by metaMetaModel
Custom : Language // As defined by metaModel
metaModel : Artifact // A metamodel artifact
metaMetaModel : Artifact // The metametamodel
metaModel _elementOf_ Ecore
metaMetaModel _elementOf_ Ecore
metaModel _conformsTo_ metaMetaModel
metaMetaModel _conformsTo_ metaMetaModel
metaModel _defines_ Custom
metaMetaModel _defines_ Ecore
Megamodel of the ATL story

module ATL import (EMF) // ATL depends on EMF
transformation : Custom → Custom // Function on DSL
input : Artifact // Input artifact
output : Artifact // Output artifact
input _elementOf Custom // input (source) of transformation
output _elementOf Custom // output (target) of transformation
transformation(input) ↦ output // Function application
ATL : Language // The ATL language
atlmodule : Artifact // An ATL transformation module
atlmodule _elementOf ATL
atlmodule _defines transformation // Semantics of ATL module
Megamodel of the Xtext story

module Xtext import (EMF) // Xtext integrates with EMF
Xtext : Language // Xtext language
grammar : Artifact // An artifact for the grammar
grammar _elementOf Xtext // An Xtext grammar
EcoreWithoutOps : Language // Relevant subset of Ecore
EcoreWithoutOps _subsetOf Ecore
metaModel _elementOf EcoreWithoutOps // Restriction of import
metaModel _correspondsTo grammar // Correspondence
generator : Xtext → EcoreWithoutOps // Generator function
generator(grammar) ↦ metaModel // Generator application
MWE2 : Language // Language for generator configuration
workflow : Artifact // Workflow artifact
workflow _elementOf MWE2 // Workflow is written in MWE2
workflow _defines generator // Workflow defines generator function
module EMFModelAPI import (EMF, // The EMF module is enhancedXML) // The XML module is needed for serializationJava : Language // Java is a LanguageEcoreJava : Language // A Java subset for EMF Model APIsEcoreJava subsetOf Java // An EMF Model API is valid JavaEMFGenModel : Language // Language for the generator modelgenModel : Artifact // Parameters of the generationgenModel elementOf EMFGenModelgenModel references metaModel // ReferencingEMFGenerator : EMFGenModel → EcoreJavaEMFGenerator(genModel) → javaFiles // Application of generatorCustomObjects : Language // Object graphs for CustomSerialization : CustomObjects → CustomDeserialization : Custom → CustomObjectsXMI : Language // Format for default persistence for EMFXML subsetOf XML // XML is a subset of XML
XMI: Language // Format for default persistence for EMF
XMI subsetOf XML // XMI is a subset of XML
Custom subsetOf XMI // Custom uses default persistence
javaFiles: Artifact+ // The modeled/defined API
javaFiles elementOf EcoreJava
metaModel correspondsTo javaFiles // Close resemblance
javaFiles defines CustomObjects
javaFiles defines CustomSerialize
javaFiles defines CustomDeserialize
model: Artifact // A serialized artifact
model elementOf Custom // ... of Custom language
model conformsTo metaModel // Conformance to metamodel
objectGraph: Transient // A runtime artifact
objectGraph elementOf CustomObjects
objectGraph conformsTo javaFiles // Conformance to Java classes
CustomSerialize(objectGraph) ↦ model
CustomDeserialize(model) ↦ objectGraph
## Discovery of entities and relationships

<table>
<thead>
<tr>
<th>Id</th>
<th>Question</th>
<th>Relevant MegaL constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>Which languages can be identified?</td>
<td>Type <em>Language</em></td>
</tr>
<tr>
<td></td>
<td>Is one language contained in another?</td>
<td>Relationship <em>subsetOf</em></td>
</tr>
<tr>
<td>A1</td>
<td>What artifacts participate in the scenario?</td>
<td>Type <em>Artifact</em></td>
</tr>
<tr>
<td></td>
<td>What is the language of each artifact?</td>
<td>Relationship <em>elementOf</em></td>
</tr>
<tr>
<td>A3</td>
<td>Does an artifact conform to another artifact?</td>
<td>Relationship <em>conformsTo</em></td>
</tr>
<tr>
<td>A4</td>
<td>Does an artifact define a language?</td>
<td>Relationship <em>defines</em></td>
</tr>
<tr>
<td>F1</td>
<td>Is one artifact derived from another artifact?</td>
<td>Type <em>Function</em></td>
</tr>
<tr>
<td></td>
<td>What is domain and range of a function?</td>
<td>Function with domain &amp; range</td>
</tr>
<tr>
<td></td>
<td>How is a function applied?</td>
<td>Function application ‘f (x) → y’</td>
</tr>
<tr>
<td></td>
<td>How is a function defined?</td>
<td>Relationship <em>defines</em></td>
</tr>
<tr>
<td>R1</td>
<td>Are artifacts closely similar to each other?</td>
<td>Relationship <em>correspondsTo</em></td>
</tr>
<tr>
<td></td>
<td>Can a correspondence be structured?</td>
<td>Relationship <em>partOf</em></td>
</tr>
<tr>
<td></td>
<td>What causes a correspondence?</td>
<td>Function application ‘f (x) → y’</td>
</tr>
<tr>
<td>C1</td>
<td>Can the entity be described conceptually?</td>
<td>Type <em>Concept</em></td>
</tr>
<tr>
<td></td>
<td>Does the entity use the concept?</td>
<td>Relationship <em>uses</em></td>
</tr>
<tr>
<td>C3</td>
<td>Does the entity help to use the concept</td>
<td>Relationship <em>facilitates</em></td>
</tr>
</tbody>
</table>
Linguistic architecture in forward and reverse engineering
Interconnected linguistic architecture
(as an advanced form of modeling
that emphasizes connection between model
and system and other elements)
URI-based resolution

github://user/project/files/data.jar/content.xml/root/models/model#1

- GitHub repo
- File
- Archive content
- File again
- XML-based (XPath-like) selection

// module EMF continued
metaMetaModel = 'eclipse://org.eclipse.emf.ecore/model/Ecore.ecore'
Semantic annotations

‘Identity’ links to Wikipedia etc.

XML | http://dbpedia.org/page/XML
EMF | https://eclipse.org/modeling/emf/

// module XML continued
XML = 'http://dbpedia.org/page/XML'

// module EMFModelAPI continued
Persistence : Concept
Persistence =
    'http://dbpedia.org/page/Persistence_(computer_science)'
CustomSerialize facilitates Persistence
CustomDeserialize facilitates Persistence
Pluggable analyses

```
xmlFile elementOf XML
xmlFile

File not element of language:
The element type "name" must be terminated by the matching end-tag "</name>".
```
class XMLConformsToXSD extends MegaLEvaluator {

    // Returns an evaluation report on the model element
    protected Report<Void> evaluate(Relationship element) {
        // Use SAX for validation; translate exceptions to report

        ...
    }
}

Plugging in

```plaintext
conformsTo < Artifact * Artifact // Relationship type per prelude
ConformsToEvaluator : Plugin // Root plugin for conformance
ConformsToEvaluator = "classpath:ConformsToEvaluator"
conformsTo evaluatedBy ConformsToEvaluator
XMLConformsToXSD : Plugin // XML/XSD conformance
XMLConformsToXSD = "classpath:XMLConformsToXSD"
XMLConformsToXSD partOf ConformsToEvaluator
SAX : Technology // Semantic annotation of plugin
SAX = 'http://dbpedia.org/page/Simple_API_for_XML'
XMLConformsToXSD uses SAX
```
Modularized models

```xml
module xml

xmlFile = library.xml
xsdFiles = libraries.xsd

import Substitute xmlFile by input

import Substitute xmlFile by output

module xmlTrafo

input = library-in.xml
output = library-out.xml
xsdFiles = libraries.xsd
```

VersionOfReasoner
XML
XMLAcceptor
xmlFile
XSD

XMLAcceptor : Plugin

Bound to `classpath:plugins.jaxb.AcceptXML`

Relationships in defined document:

- XMLAcceptor partOf StringAcceptor

© 2017 Ralf Lämmel, Software Languages Team http://www.softlang.org/, University of Koblenz-Landau. All rights reserved.
Transient artifacts

- A and B: web request and response
- C: piped program output
- D: transient data in memory or database
Model inference

- Decomposition in parts
- Correspondence between parts
- Artifact bindings

```java
class EMFPartInferrer extends MegaLInferrer {
    // Returns an evaluation report and a model extension
    protected Report<Megamodel> infer(Entity element) {
        ... }
}
```
Explorable connections
## Traceability links

![Diagram showing traceability links between Java files, metadata, and XML files.](image)

<table>
<thead>
<tr>
<th>xsdFiles</th>
<th>javaFiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>/xs:schema/xs:complexType</td>
<td>org/softlang/company/xjc/Employee.java</td>
</tr>
<tr>
<td>/xs:schema/xs:element#0</td>
<td>org/softlang/company/xjc/Company.java</td>
</tr>
<tr>
<td>/xs:schema/xs:element#1</td>
<td>org/softlang/company/xjc/Department.java</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>xmlFile</th>
<th>objectGraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>~ company/department#0</td>
<td>org.softlang.company.xjc.Department@5fd1a6aa</td>
</tr>
<tr>
<td>~ employee#0</td>
<td>org.softlang.company.xjc.Employee@1a56a6c6</td>
</tr>
<tr>
<td>address:Utrecht</td>
<td>Utrecht</td>
</tr>
<tr>
<td>name:Erik</td>
<td>Erik</td>
</tr>
<tr>
<td>salary:12345</td>
<td>12345.0</td>
</tr>
<tr>
<td>~ employee#1</td>
<td>org.softlang.company.xjc.Employee@748e432b</td>
</tr>
</tbody>
</table>
## Literature survey

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
</tr>
<tr>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
</tr>
<tr>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
</tr>
<tr>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
</tr>
<tr>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
</tr>
<tr>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
</tr>
<tr>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
<td>⋯</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$L_3$</th>
<th>$L_2$</th>
<th>$L_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>○</td>
<td>⋆</td>
<td>⋼</td>
</tr>
<tr>
<td>⋼</td>
<td>⋼</td>
<td>⮔</td>
</tr>
<tr>
<td>⮔</td>
<td>⮔</td>
<td>⮔</td>
</tr>
<tr>
<td>⮔</td>
<td>⮔</td>
<td>⮔</td>
</tr>
<tr>
<td>⮔</td>
<td>⮔</td>
<td>⮔</td>
</tr>
<tr>
<td>⮔</td>
<td>⮔</td>
<td>⮔</td>
</tr>
</tbody>
</table>

- $L_1$-$L_3$ maturity levels
- ○ implementation
- ⮔ demonstration

<table>
<thead>
<tr>
<th>3.8</th>
<th>3.1</th>
<th>3.6</th>
<th>3.3</th>
<th>3.7</th>
<th>3.4</th>
<th>3.2</th>
<th>3.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traceability links</td>
<td>Artifact binding</td>
<td>Model inference</td>
<td>Pluggable analyses</td>
<td>Explorable connections</td>
<td>Modularized models</td>
<td>Semantic annotations</td>
<td>Transient artifacts</td>
</tr>
</tbody>
</table>
Axioms of linguistic architecture
## Entity types in megamodeling survey

<table>
<thead>
<tr>
<th>Paper</th>
<th>Artifact</th>
<th>Function</th>
<th>Record</th>
<th>System</th>
<th>Technology</th>
<th>Language</th>
<th>Inf. resource</th>
<th>Fragment</th>
<th>Collection</th>
<th>Trace</th>
<th>Concept</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[2]</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[3]</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[4]</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[5]</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[6]</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[7]</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[8]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[9]</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[10]</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[11]</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[12]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[13]</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[14]</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Relationship types in megamodelling survey

<table>
<thead>
<tr>
<th>Paper</th>
<th>Conformance</th>
<th>Definition</th>
<th>Correspondence</th>
<th>Implementation</th>
<th>Usage</th>
<th>Membership</th>
<th>Typing</th>
<th>Dependency</th>
<th>Abstract rel.</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>[2]</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[3]</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>[4]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[5]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[6]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>[7]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>[8]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>[9]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>[10]</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>[11]</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>[12]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>[13]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>[14]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
Understanding Membership

company.ecore defines XMLCompany

conformsTo

company.xmi

elementOf
Understanding Membership

- **s**: Artifact
- **l**: Language
- **a**: Artifact

**Diagram:***
- `s` defines `l`.
- `a` conformsTo `s`.
- `a` is an elementOf `l`.

**Table:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s</code></td>
<td><code>a</code></td>
</tr>
<tr>
<td>Grammar</td>
<td>Code</td>
</tr>
<tr>
<td>Schema</td>
<td>Instance</td>
</tr>
<tr>
<td>Metamodell</td>
<td>Model</td>
</tr>
</tbody>
</table>
Understanding Membership

\[ \text{elementOf}(a, l) \Rightarrow \text{Artifact}(a) \land \text{Language}(l) \ldots \]

\[ \text{elementOf}(a, l) \iff \exists s. \text{defines}(s, l) \land \text{conformsTo}(a, s). \]
Understanding Membership

- Specification(a) ⇒ Artifact(a).
- Language(l) ⇒ ∃s.Specification(s) ∧ defines(s, l) ...
- defines(a, e) ⇒ Artifact(a) ∧ Entity(e).
- conformsTo(a, s) ⇒ Artifact(a) ∧ Artifact(s).
- conformsTo(a, s) ⇐ (∀p_a.partOf(p_a, a) ∧ ∃p_s.partOf(p_s, s) ∧ conformsTo(p_a, p_s)) ∧ ∃t.defines(s, t) ∧ elementOf(a, t).
Ontology engineering

SoLaSoTe ontology

Linguistic architecture
Social coding

Software chrestomathy 101
Code  Doc  Wiki

Megamodeling with MegaL
EMF
Java
Django
Megamodeling
for repository management
(proxying for build management and regression testing)
YAS (Yet Another SLR
(Software Language Repository))

- 107 languages. (This includes different representation types.)
- 558 language-typed artifacts.
- 121 language-typed functions.
- 391 function applications.
- 252 Prolog modules.
- 171 Haskell modules.
- 111 Java classes.
- 19 Python scripts.

How to build and test such a repository?
How to understand all the dependencies?
Illustrative YAS artifacts
A binary number

101.01

BTW, everything is a file in a/this SLR.
2 The YAS software language repository

2.1 Examples of languages

Figure 2 shows basic representation types in YAS and a few more specific software languages related to different aspects of a simple language BNL—Binary Number Language. The nodes in the figure denote languages including ‘formats’ (e.g., XML-based ones) or general ‘representation types’ (e.g., text). The directed edges (arrows) denote subset relationship for languages in a set-theoretical sense. For instance, language $bnl(text)$ corresponds to the concrete textual syntax of BNL. Thus, language $text$ can be viewed as the universe for text-based languages. We explain the various languages in the sequel.

Here is an example of a binary number represented as text, i.e., an element of $bnl(text)$:

$$
\text{Text resource } \text{languages/BNL/samples/5comma25.bnl}
$$

101.01

Language $bnl(json)$ corresponds to the abstract, tree-based syntax of BNL using JSON for representation; here is the JSON representation of ‘101.01’:

© 2017 Ralf Lämmel, Software Languages Team http://www.softlang.org/, University of Koblenz-Landau. All rights reserved.
Figure 2 shows basic representation types in YAS and a few more languages related to different aspects of a simple language BNL-Language. The nodes in the figure denote languages including ‘for-based ones) or general ‘representation types’ (e.g., text). The directe denote subset relationship for languages in a set-theoretical sen language $bnl(text)$ corresponds to the concrete textual syntax of BNL text can be viewed as the universe for text-based languages. We ex languages in the sequel.

Here is an example of a binary number represented as text, i.e. $bnl(text)$:

Text resource languages/BNL/samples/5comma25.bnl

101.01

Language $bnl(json)$ corresponds to the abstract, tree-based synt JSON for representation; here is the JSON representation of ‘101.01’
Initial commit due to repo reorg

1 contributor

2 lines (1 sloc) 7 Bytes

1 101.01
Branch: programming17

yas / languages / BNL / samples / 5comma25.bnl

rallemmel Initial commit due to repo reorg

1 contributor

2 lines (1 sloc)  |  7 Bytes

1  101.01
commit due to repo reorg

7 Bytes
Decimal representation of $101.01$

$5.25.$
A JSON-based AST of 101.01

```json
{
    "bits": ["one", "zero", "one"],
    "rest": ["zero", "one"]
}
```
Symbolic conversion from 101.01 to 5.25

\[ 2^{(1+1+1-1)} + (0 + 2^{(1+1+1-1-1)}) + (0 + 2^{(-1-1)}). \]
An ANTLR grammar for binary numbers

```antlr
grammar BnlEbnf;
@header { package org.softlang; }
number : bit+ (\'.\' bit+)? WS? EOF;
bit : '0' | '1';
WS : [ \t\n\r]+;
```
A BNF for binary numbers

[number] number : bits rest ;
[single] bits : bit ;
[many] bits : bit bits ;
[zero] bit : '0' ;
[one] bit : '1' ;
[integer] rest : ;
[rational] rest : '.' bits ;
An algebraic signature for binary numbers

symbol number: bits × rest → number;
symbol single: bit → bits;
symbol many: bit × bits → bits;
symbol zero: → bit;
symbol one: → bit;
symbol integer: → rest;
symbol rational: bits → rest;
A DCG for
binary-to-decimal number of conversion

\[2^{(1+1+1-1)} + (0+2^{(1+1+1-1-1)}) + (0+2^{(-1-1)})\]

\[
\begin{align*}
\text{number}(\text{Val1}+\text{Val2}) & \rightarrow \text{bits}(\text{Len1}-1, \text{Len1}, \text{Val1}), \text{rest}(\text{Val2}). \\
\text{bits}(\text{Pos}, 1, \text{Val}) & \rightarrow \text{bit}(\text{Pos}, \text{Val}). \\
\text{bits}(\text{Poso}, \text{Len1}+1, \text{Val1}+\text{Val2}) & \rightarrow \text{bit}(\text{Poso}, \text{Val1}), \text{bits}(\text{Poso}-1, \text{Len1}, \text{Val2}). \\
\text{bit}(\_\text{Pos}, 0) & \rightarrow ['0']. \\
\text{bit}(\text{Pos}, 2^{\_\text{Pos}}) & \rightarrow ['1']. \\
\text{rest}(0) & \rightarrow []). \\
\text{rest}(\text{Val}) & \rightarrow ['.'], \text{bits}(-1, \_\text{Len}, \text{Val}).
\end{align*}
\]
An evaluator

\[ 2^{(1+1+1-1)} + (0 + 2^{(1+1+1-1-1)}) + (0 + 2^{(-1-1)}) \]

\text{evaluate}(F, V) \iff V \text{ is } F.

5.25.
Languages as types in an SLR
Relations and functions on languages
Relations and functions on languages

- `bnl(tokens(term))`
  - `convert # bnlTextConverter`
  - `parse # bglParser`

- `bnl(term)`
  - `convert # bnlTermConverter`
  - `explode # bglExploder`
  - `evaluate # bnlTermEvaluator`
  - `unparse # bglTreeToTokens`

- `bnl(formula(term))`
  - `solve # bnlSolver`

- `bnl(value(term))`

- `parse # bglParser(bnlScann)`
Relations and functions on languages

bnl(tokens(term))

parse # bglParser

bnl(term)

termConverter

explosion # bglExploder

bnl(tree(term))

unparse # bglTreeToTokens
unparse # bglTreeToText
implode # bglImploder

bnl(text)

parse # bglParser(bnlScanner)
ueber's language concepts

**Ueber concept**

- **Language** represented as
- **Membership** defined by
  - `elementType`
- **Relation** models
  - `conformance`
  - `compatibility`
  - `traceability`
  - `correspondence`
- **Function** models
  - `parsing`
  - `resolution`
  - `transformation`
  - `analysis`
  - `formatting`
A small megamodel

language(text).
language(term).
language(bnl(text)).
language(bnl(term)).
elementOf(‘…/samples/5comma25.bnl’, bnl(text)).
elementOf(‘…/samples/5comma25.term’, bnl(term)).
membership(bnl(text), bglAcceptor(bnlScanner), ['languages/BNL/cs.term']).
membership(bnl(term), bslAcceptor, ['languages/BNL/as.term']).
relation(conformsTo, [term, bsl(term)], bslConformance, []).
relatesTo(conformsTo, ['…/samples/5comma25.term', 'languages/BNL/as.term']).
function(parse, [bnl(text)], [bnl(term)], bglParser(bnlScanner), ['languages/BNL/cs.term']).
mapsTo(parse, ['…/samples/5comma25.bnl'], ['…/samples/5comma25.term']).
Abstract syntax of ueber

type model = decl*;
symbol language : lang → decl;
symbol elementOf : file × lang → decl;
symbol notElementOf : file × lang → decl;
symbol membership : lang × goal × file* → decl;
symbol relation : rela × lang* × goal × file* → decl;
symbol relatesTo : rela × file* → decl;
symbol function : func × lang* × lang* × goal × file* → decl;
symbol mapsTo : func × file* × file* → decl;
symbol equivalence : lang × goal × file* → decl;
symbol normalization : lang × goal × file* → decl;
symbol macro : goal → decl;
type file = string; // filenames

type rela = string; // names of relations

type func = string; // names of functions

type lang = term; // names of languages

type goal = term; // Prolog literals
ueber's semantics

SLR
- ueber
  - languages
    - bnl
      - .ueber
      - .ueber
      - 5comma25.bnl
      - 5comma25.term
      - ...
    - ...
  - ...

Collection

Well-formedness checking

Ueber declarations

Repository verification

Problems

Artifact override and creation
‘Breaking changes to a language processor’

○ Baseline languages/PPL/tests/hseplist.txt: disagreeing.
○ mapsTo(pp,[languages/PPL/tests/hseplist.ppl],[....txt]): UNVERIFIED.

‘Development of a new test case’

○ Baseline languages/PPL/tests/indent.txt: missing.
○ elementOf(languages/PPL/tests/indent.txt,text): UNVERIFIED.
○ mapsTo(pp,[languages/PPL/tests/indent.ppl],[....txt]): UNVERIFIED.

‘Modeling a new relationship’

○ Overload evaluate: ([languages/BNL/samples/5comma25.bnl]) -&gt; ([....value]): missing.
○ mapsTo(evaluate,[languages/BNL/samples/5comma25.bnl],[....value]): NOT OK.
○ mapsTo(evaluate,[languages/BNL/samples/5comma25.bnl],[....value]): UNVERIFIED.
# Integrated compile-and run-time

<table>
<thead>
<tr>
<th>Implement relations &amp; functions</th>
<th>Prolog</th>
<th>Haskell</th>
<th>Java</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicates</td>
<td>Main functions</td>
<td>Main methods</td>
<td>Scripts</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code location</th>
<th>Module auto loading</th>
<th>Automated module search path</th>
<th>Automated CLASSPATH</th>
<th>Automated PYTHONPATH</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Compilation</th>
<th>N/A</th>
<th>On the fly</th>
<th>On the fly</th>
<th>N/A</th>
</tr>
</thead>
</table>

# Representation across different implementation languages

<table>
<thead>
<tr>
<th>Parse text</th>
<th>Prolog</th>
<th>Haskell</th>
<th>Java</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCG, ...</td>
<td>Parsec, ...</td>
<td>ANTLR, ...</td>
<td>ANTLR, ...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Represent trees</th>
<th>Prolog terms</th>
<th>read/show conversion, JSON, XML</th>
<th>JSON, XML</th>
<th>str/repr conversion, JSON, XML</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Represent graphs</th>
<th>Prolog terms</th>
<th>N/A</th>
<th>Serializable objects, XMI, ...</th>
<th>Serializable objects, ...</th>
</tr>
</thead>
</table>
Advanced aspect of the YAS SLR

• Semantic annotations and LOD
• Incremental building
• Test-data generation
• Migration of a project to ueber
• Integration with version control
• Infer ueber declarations based on conventions
• A useful model of language variants
Thank you!