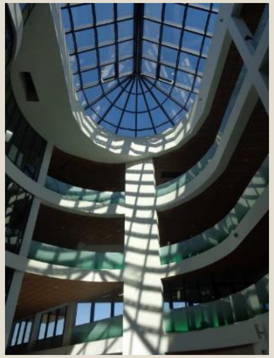
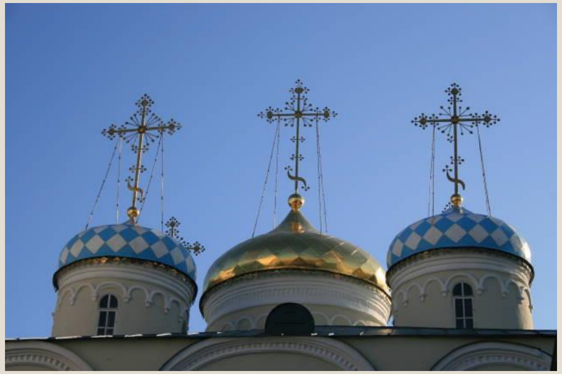
HOW PRINCESS TEACHES YOU TO THINK

Thomas Baar
KeY-Workshop Summer 2016, Giersch-Chalet, France

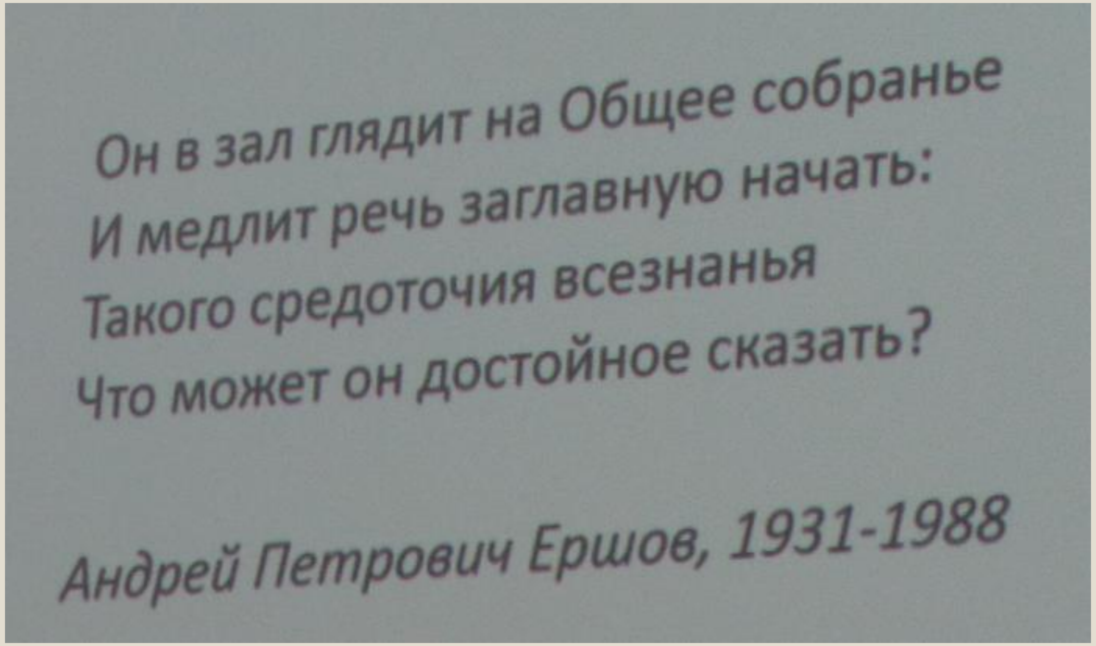
Results of my Sabbatical in Russia

(including outcome of discussions at PSI 2015 in **Kazan**)





In Memoriam

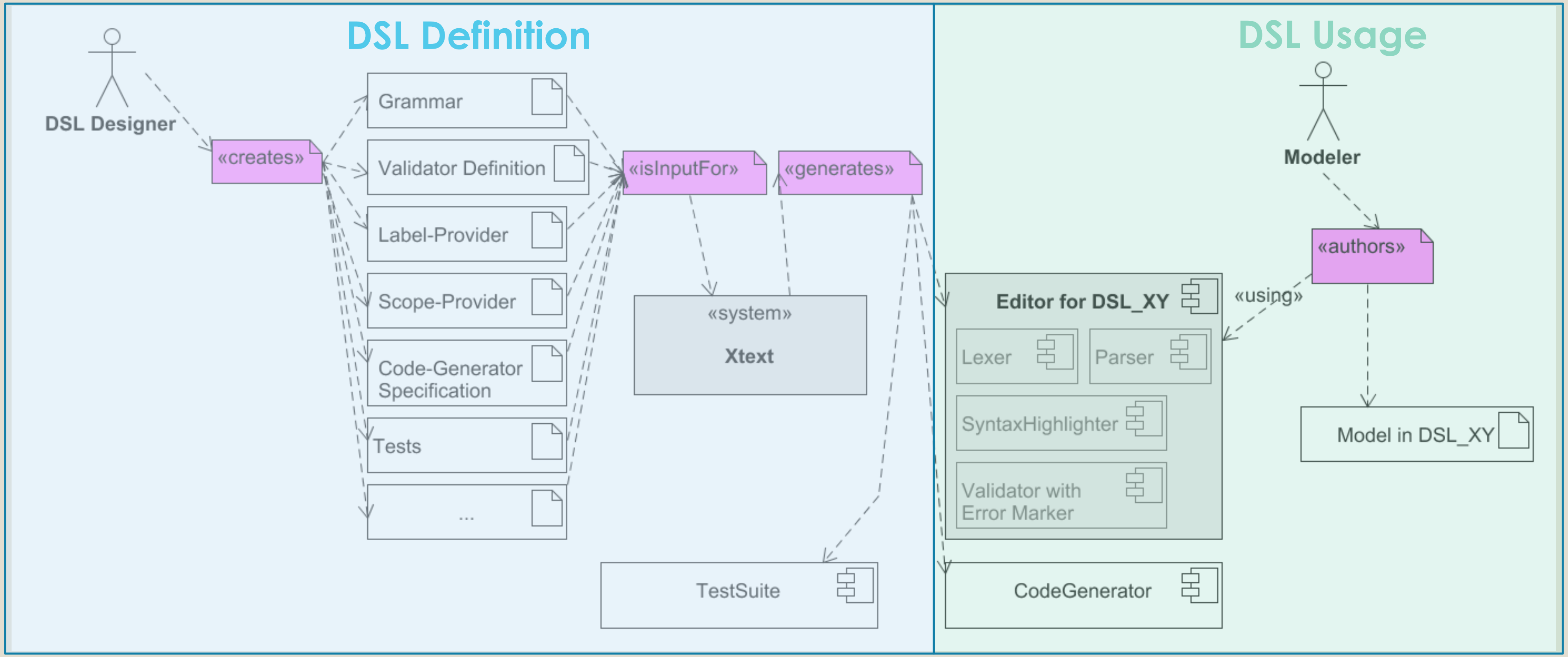


Helmut Veith (February 5, 1971 -- March 12, 2016)

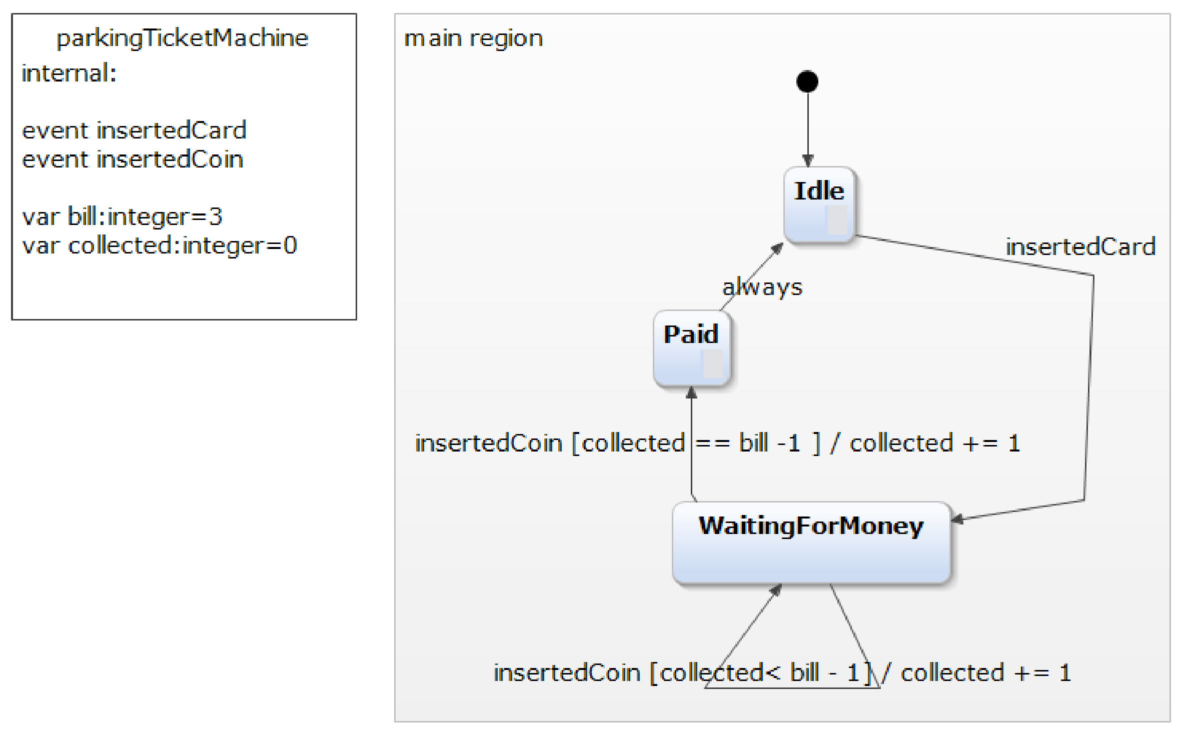
Talk‘s Topic: The Value of PRINCESS-Integration into a DSL - Toolset

- Definition of DSLs with **Xtext**
- A concrete DSL: SMINV
  - **Grammar**
  - Checking Syntactic Well-Formedness Rules
  - Checking **Semantic Well-Formedness Rules** using **PRINCESS**
- **Application** of SMINV **for Student Quizes**
  - Analyzing **Control-Flow-graphs**
  - Analyzing **Petri-Nets**
    - Developing a Front-end language for SMINV
- Future Work

Defining and Using DSLs with Xtext



Yakindu - A valuable Tool to Teach State Machines

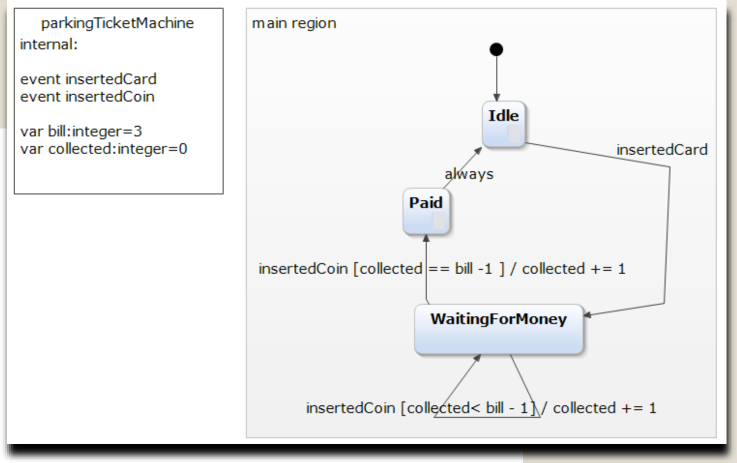


- **Yakindu** (by Itemis)
  - **Graphical editor** for State Machines
  - **Simulator** to execute modeled State Machine
    - debugging (only !) concrete traces
  - **Code generator** for Java, C++, ...
    - Basically enables **Graphical Programming** !!!!

- **However:** No support for
  - **adding invariants** on certain states
  - checking **consistency of invariants**

SMINV – A textual DSL for **S**tate **M**achines With **Inv**ariants

Textual Encoding of Yakindu's State Machine



Declarations

```
vars : collected bill

states: start idle waitingForMoney paid

events: cardInserted coinInserted
```

Transition

```
transitions :

start => idle / collected = 0 bill = 3

idle => waitingForMoney cardInserted

waitingForMoney => waitingForMoney  coinInserted [collected < bill - 1 ] / collected += 1

waitingForMoney => paid coinInserted [collected == bill - 1] / collected += 1

paid => idle
```

Pre-State Post-State Action (Var-Update)

Event Guard

SMINV – Grammar is straight-forward

```
StateDiag:
    vd=VarDecl
    sd=StateDecl
    ed=EventDecl
    td=TransDecl

VarDecl:
    'vars' ':' vars+=Var*;

Var:
    name=ID;

Transition:
    pre=[State] '=>' post=[State]
    (ev=[Event])? ('[' g=Fml ']')? ('/' act+=Update+)?;

Update:
    variable=[Var] op=('=' | '+=' | '-=') value=Term;
```

Semantics of Update as in KeY:

- when executing the transition, **change the value of the variable** (LHS) to the value of the given term (RHS) **and does not change anything else** !

SMINV – Integrating **Invariants** into the language

```
StateDiag:
    vd=VarDecl
    sd=StateDecl
    ed=EventDecl
    td=TransDecl
    id=InvDecl;
```

New language-construct
„invariant of a state"

```
InvDecl:
    {InvDecl}
    'invariants' invs+=Inv*;

Inv:
    state=[State] ':' inv=Term ';';
```

Term

- represents arithmetic **expression language** over variables
- is **imported** and adapted from different project

Validator – Check Conditions on AST

Grammar

```
Transition:
    pre=[State] '=>' post=[State]
    (ev=[Event])? ('[' g=Fml ']')? ('/' act+=Update+)?;
```

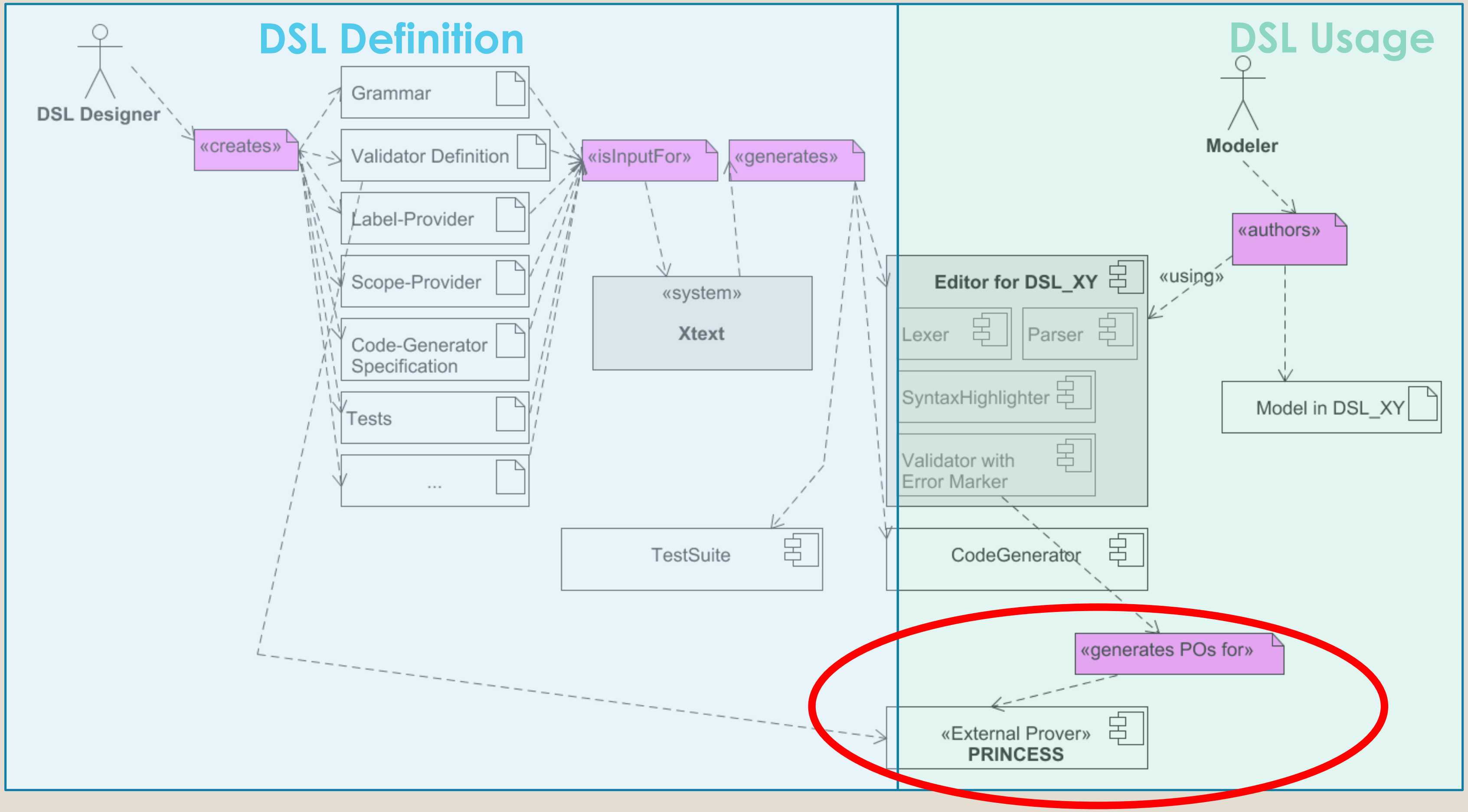
- **Validator**
  - Check condition on the parsed AST
  - implemented in Java-dialect Xtend

Transparent walking through AST strictly adhering to the grammar

Validator

```
@Check
def checkTransitionsTargetStateIsNeverStartState(Transition t) {
    if (t.post.isStartState)
        error("target of transition cannot be a start state", SminvDslPac
            STARTSTATE_IS_NO_TRANSITIONTARGET)
}
```

Integration of PRINCESS for „**semantic validation**“



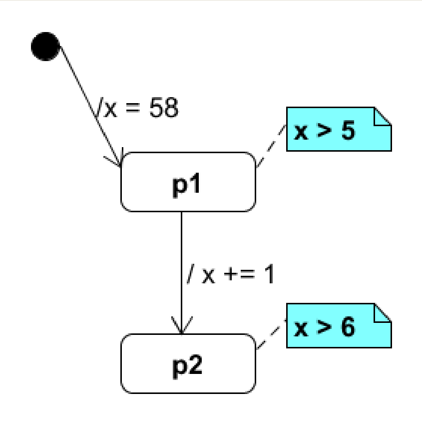
Semantic Validator „*Transition Preserves Post-State Invariants*“

$$(I_{pre(t)} \wedge guard(t)) \longrightarrow I_{post(t)}[v \leftarrow update(t)]$$

Implemented As

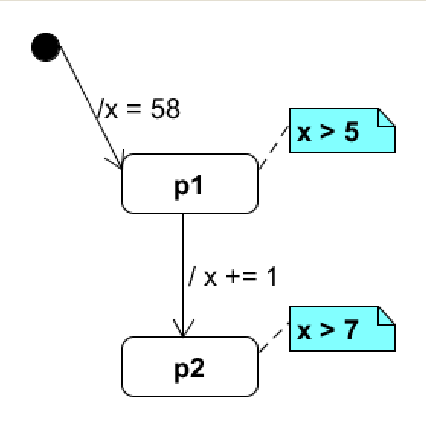
```
def Fml generatePO(Transition t) {

    val invPre = t.pre.invariantConjunction.fclone
    val guard = t.guard.fclone
    val invPost = t.post.invariantConjunction.fclone

    val map = createSubMap(t.act)

    val premise = factory.createAnd => [left = invPre right = guard]

    val result = factory.createImplies => [left = premise right = invPost.fsub(map)]

    return result
}
```

Example: Simple Update




```
transitions
start => p1 / x = 58;

p1 => p2 / x += 1;

invariants
p1 : x > 5;
p2 : x > 6;
```

No Error – every transition obeys invariants

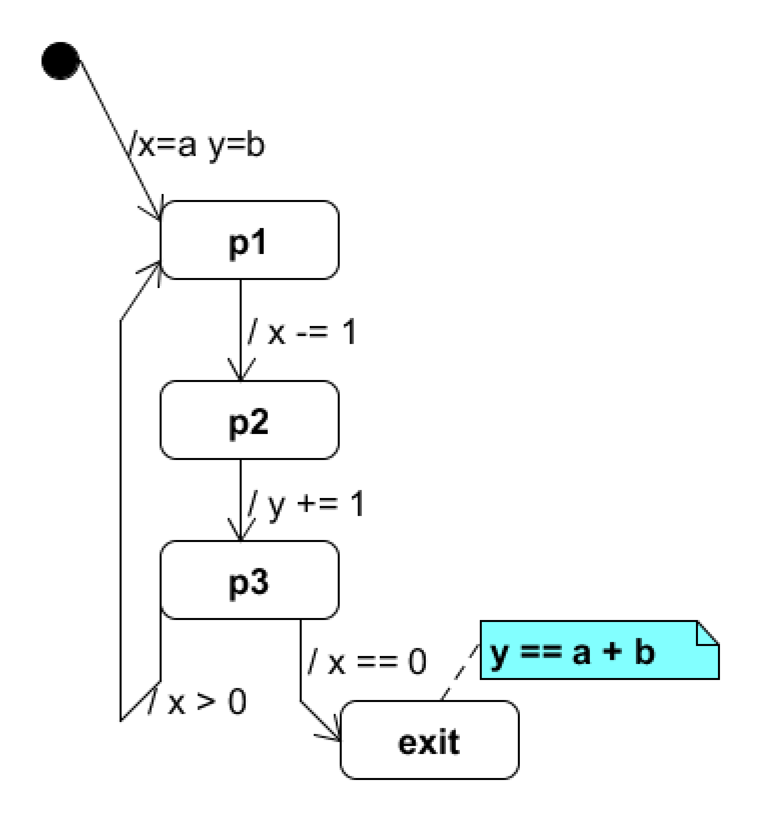



```
transitions
start => p1 / x = 58;   //TODO: allow al
                         //TODO: as an a
p1 => p2 / x += 1;

invariants
p1 : x > 5;
p2 : x > 7;
```

NOT INVARIANT-PRESERVING: the transition p1 => p2 / x += 1
in the following pre-state: x = 6

Error – feedback in which situation invariant is broken

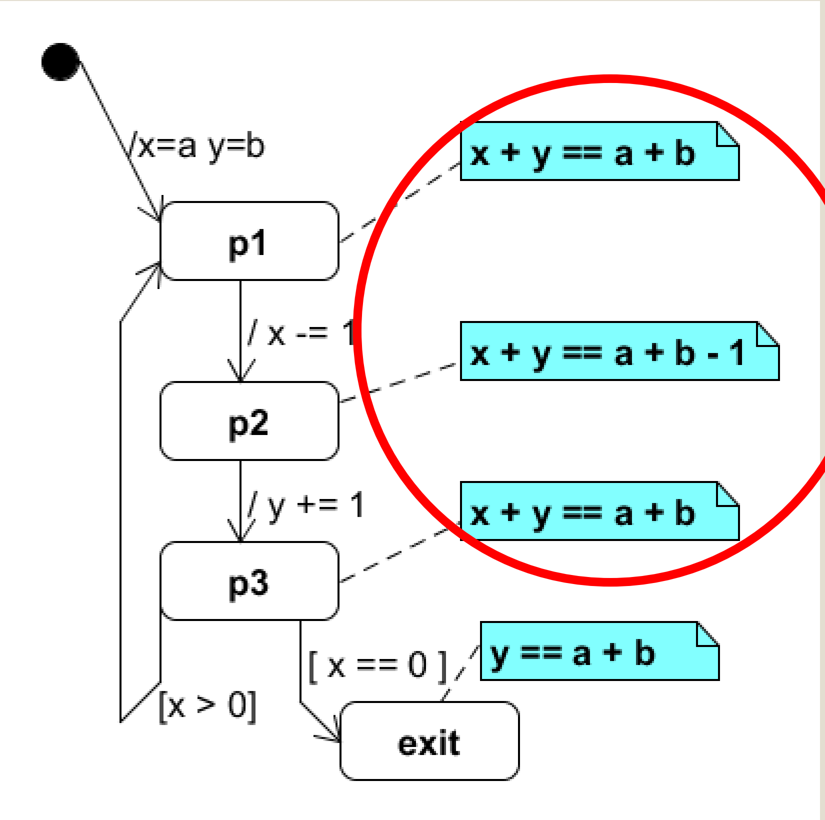
Example: Simple Loop




```
transitions
start => p1 / x = a y = b;
p1 => p2 / x -= 1;
p2 => p3 / y += 1;
p3 => p1 [x > 0];
p3 => exit [x == 0];

invariants
// the claim to prove
exit : y == a + b;
```

NOT INVARIANT-PRESERVING: the t

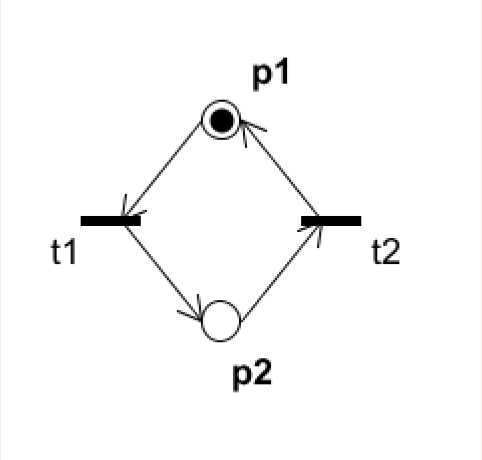
Example: Simple Loop (Solution)

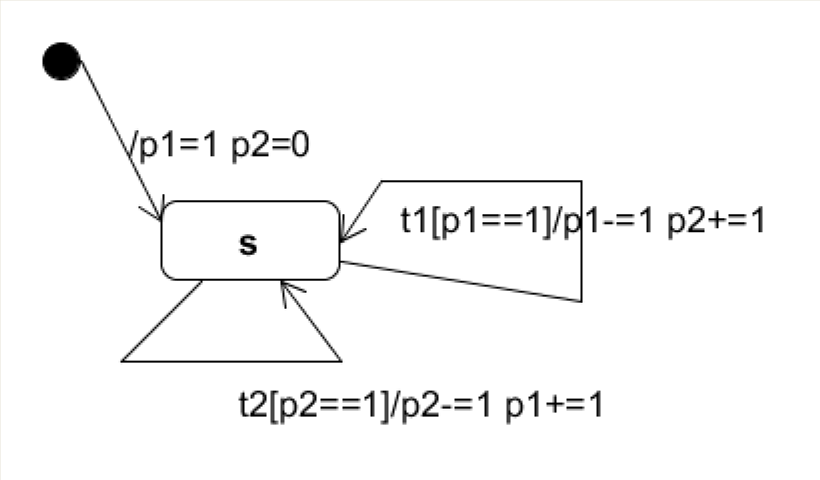


Additional invariants are semantic arguments for original claim

```
transitions
start => p1 / x = a y = b;
p1 => p2 / x -= 1;
p2 => p3 / y += 1;
p3 => p1 [x > 0];
p3 => exit [x == 0];

invariants
// the claim to prove
exit : y == a + b;

// helper invariants
p1 : x+y == a+b;
p2 : x+y == a+b-1;
p3 : x+y == a+b;
```

Encoding of **Petri-Nets** within SMINV





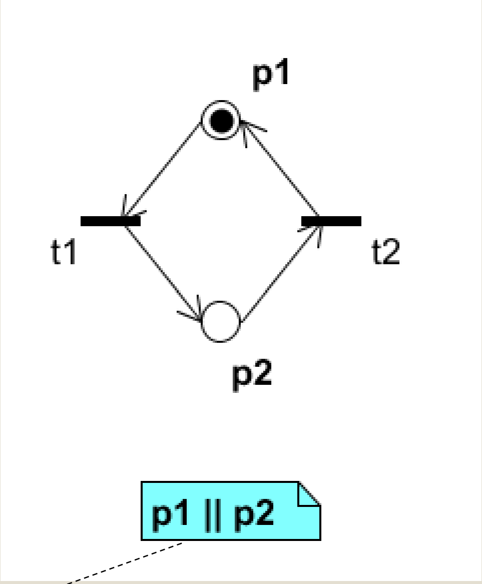
Encoding:

- place -> variable
- transition -> event
    - the semantics of PN-transitions is encoded by guard/action
- -> one global state ‚s‘
- initialization -> updates ‚start‘ – ‚s‘

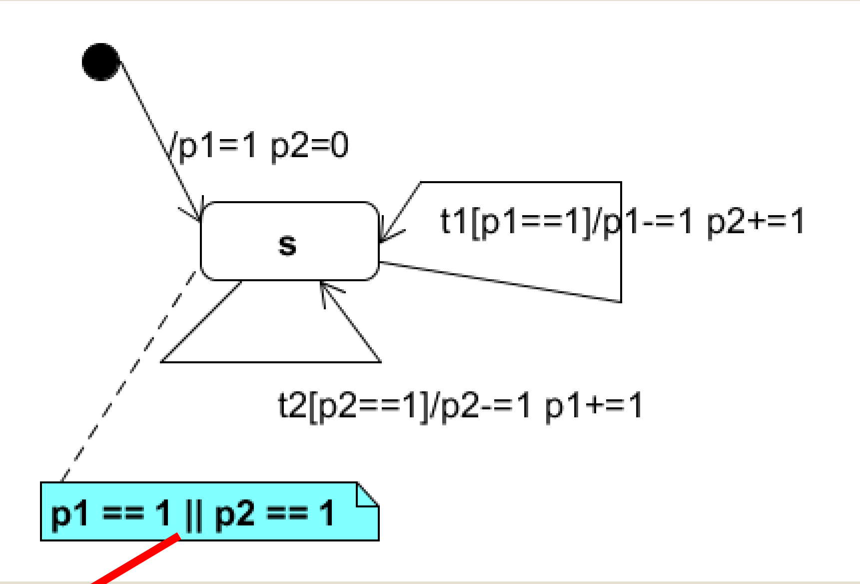
DSL_PN → **DSL_SMINV**

Encoding by Code-Generator

Proving Safety-Props for Petri-Nets



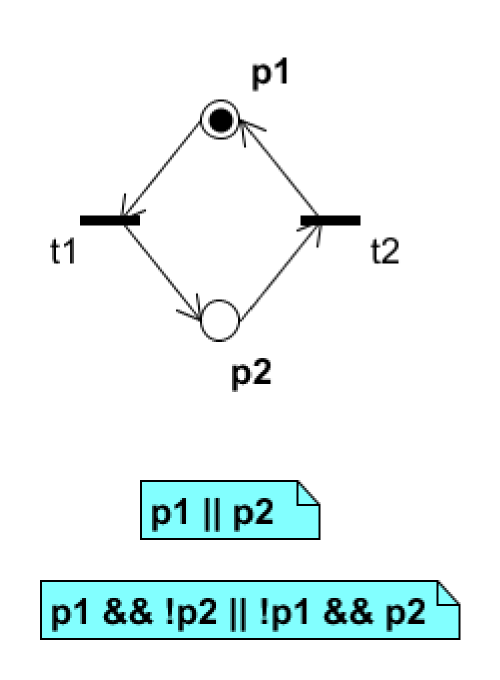
To be read as:
Always (in all reachable states), there is a token on p1 or p2

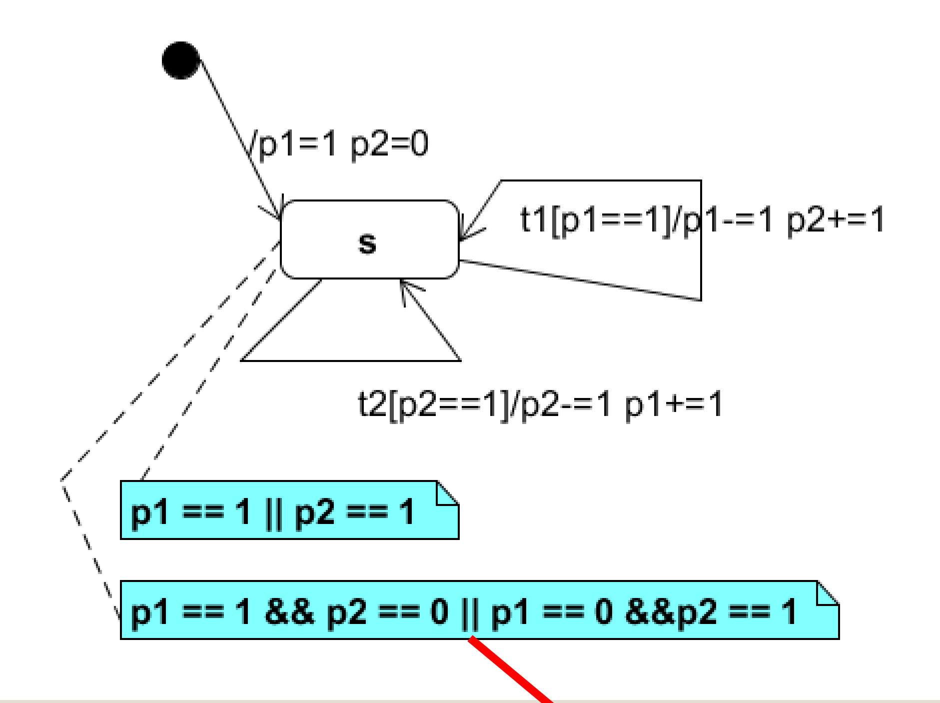


Not Provable !!!

Reason: Encoding 'p1' -> 'p1 == 1' is rather strict and only justified for nets with at most one token per place

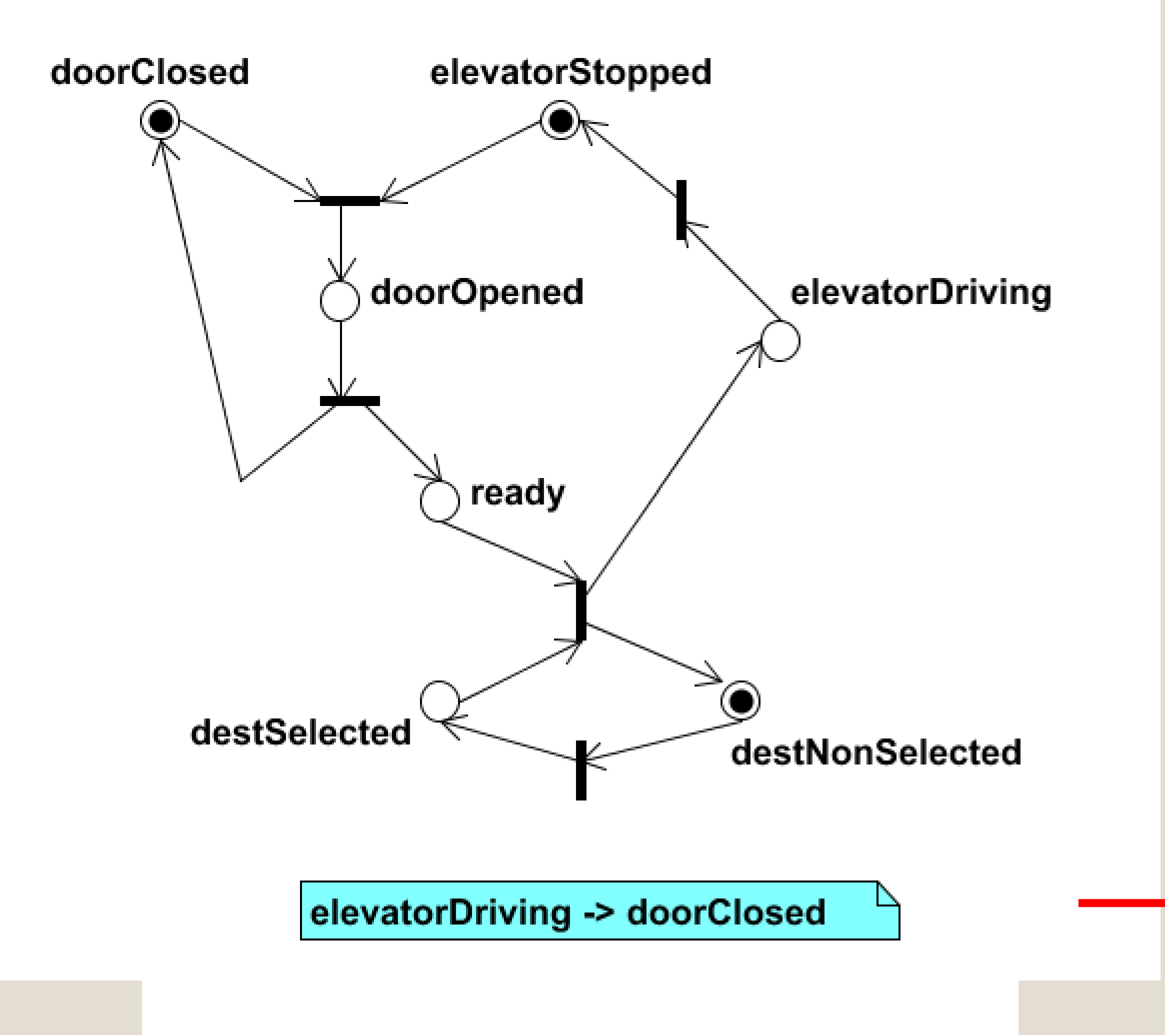
Proving Safety-Props for Petri-Nets





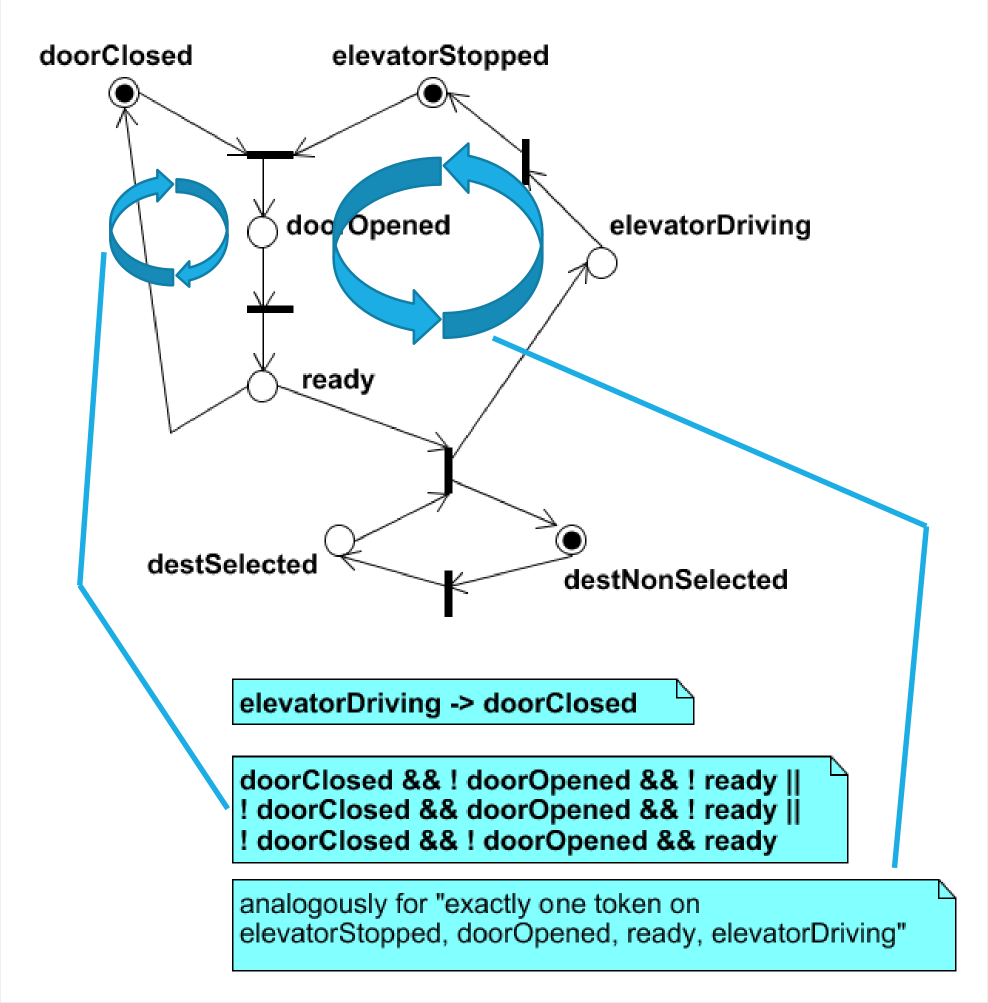
Provable (explicit statement that number of tokens is always 0 or 1)

Example: Elevator specified by as Petri-Net



Not Provable !!!

Example: Elevator as Petri-Net



Provable !!!

Summary

- Starting Point: **Yakindu**
  - Xtext-Grammar for State-Machines is folklore
- **Adding invariants** to language
  - easy to realize but **increases** dramatically **expressive power**
  - **PRINCESS** has been integrated to discard proof obligations
    - **very fast** -> **instant feedback** to the user !!!
- SMINV can **simulate Petri-nets**
  - Lightweight analysis of Petri-nets now possible

- **Target audience** of tool: **students** doing state modelling

Everything is available on GitHub ☺

`https://github.com/thomasbaar/simplesma.git`

Future Work

- **Graphical editor** for Xtext languages
  - currently, a Bachelor-thesis works on this
- Better **support for „front-end“ languages**
  - errors should be shown directly in Petri-Net editor (not only in encoded SMINV-file)