Live Little Languages

Tijs van der Storm
storm@cwi.nl / @tvdstorm

Colloquium at university of groningen 10-6-2015

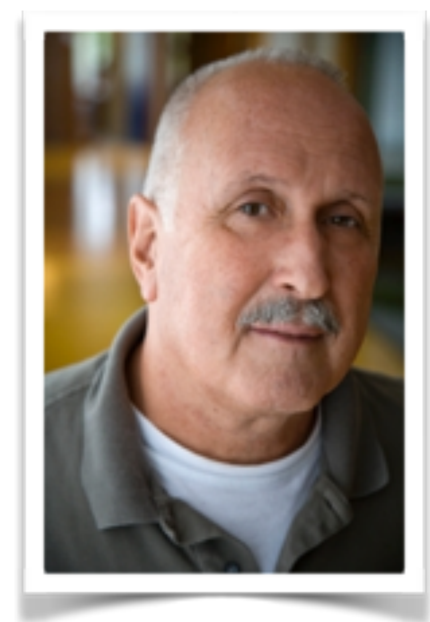
CWI
Centrum Wiskunde & Informatica

About me

- Senior researcher CWI
- Software Analysis and Transformation (SWAT)
- Currently supervising 2 PhD students
- Teach at Master Software Engineering at UvA
- Supervise around 10 MSc students per year

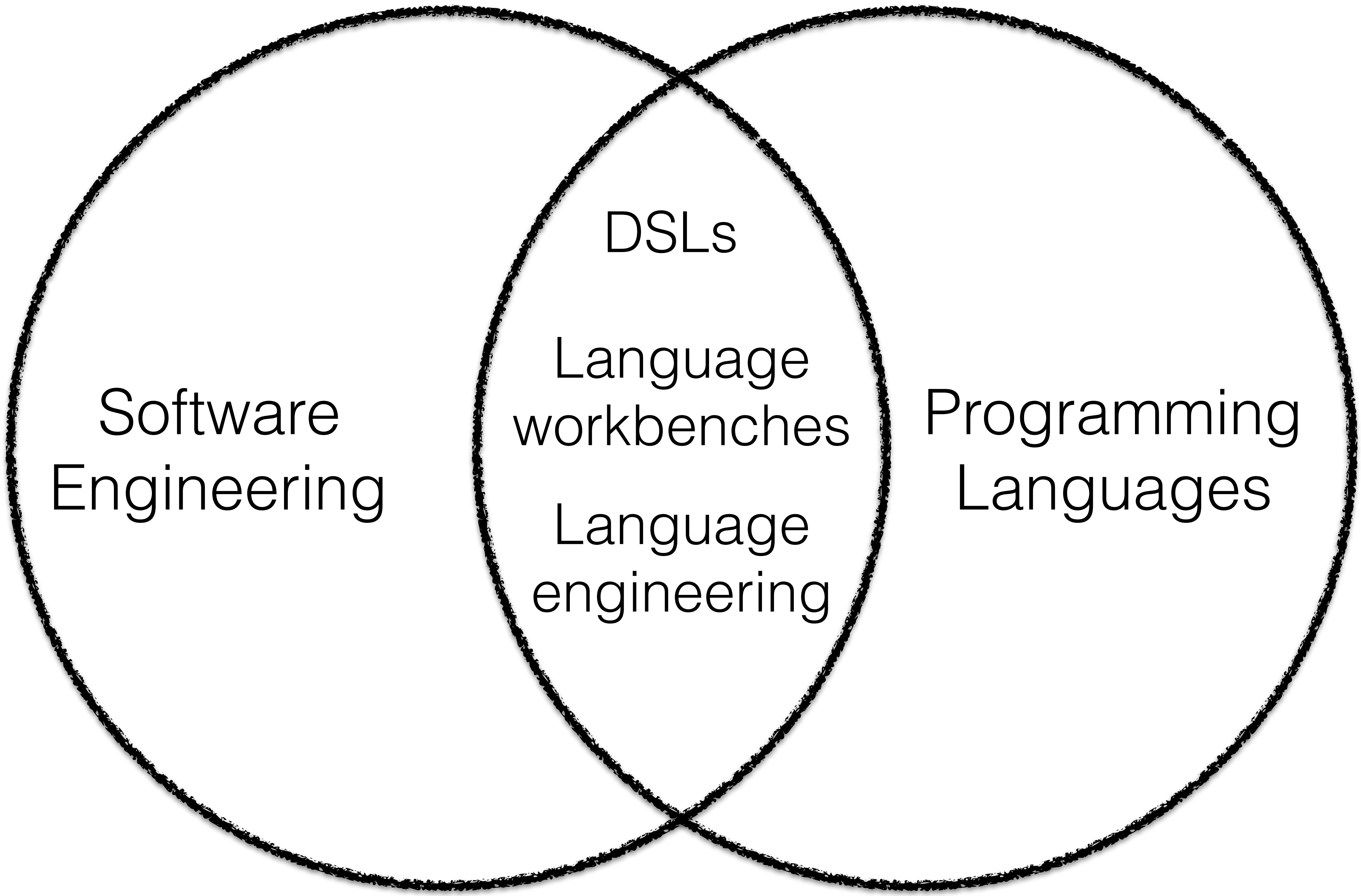
Rascal

- Meta programming language
- Language workbench
- w/ Paul Klint and Jurgen Vinju
- … and many others!
- http://www.rascal-mpl.org




Ensō

- Model-driven programming framework
- Composition of executable specification languages
- “App = Models + Interpreters”
- with William Cook, UT Austin
- http://www.enso-lang.org


Software
Engineering
DSLs
Language
workbenches
Language
engineering
Programming
Languages

Some recent topics

- Object Algebras
  - OOPSLA'15 (hopefully), GPCE'14, ECOOP'13
- Language workbenches
  - ICMT'15, ECOOP'14, ICMT'14, SLE'13
- Domain-specific language for digital forensics
  - ECMFA'13, ICMT'12, ICSE'11 SEIP

Reaching out

- Co-organized SDA'13, SDA'14
- Talks/workshops at
  - Code generation
  - Joy of Coding
  - Bits&Chips
- Sioux, Belastingdienst, NSpyre, NFI, Optiver…

Live little languages

Live little languages

- Live: “editing a program while it runs”
    - Continuous feedback
    - Textbook example: spreadsheet
- Little languages
    - Domain-specific languages (DSLs)
    - Notations close to problem domain

Traditional programming:
aim, shoot, miss/hit, repeat

Live programming:
continuous aiming,
with continuous feedback

General purpose languages

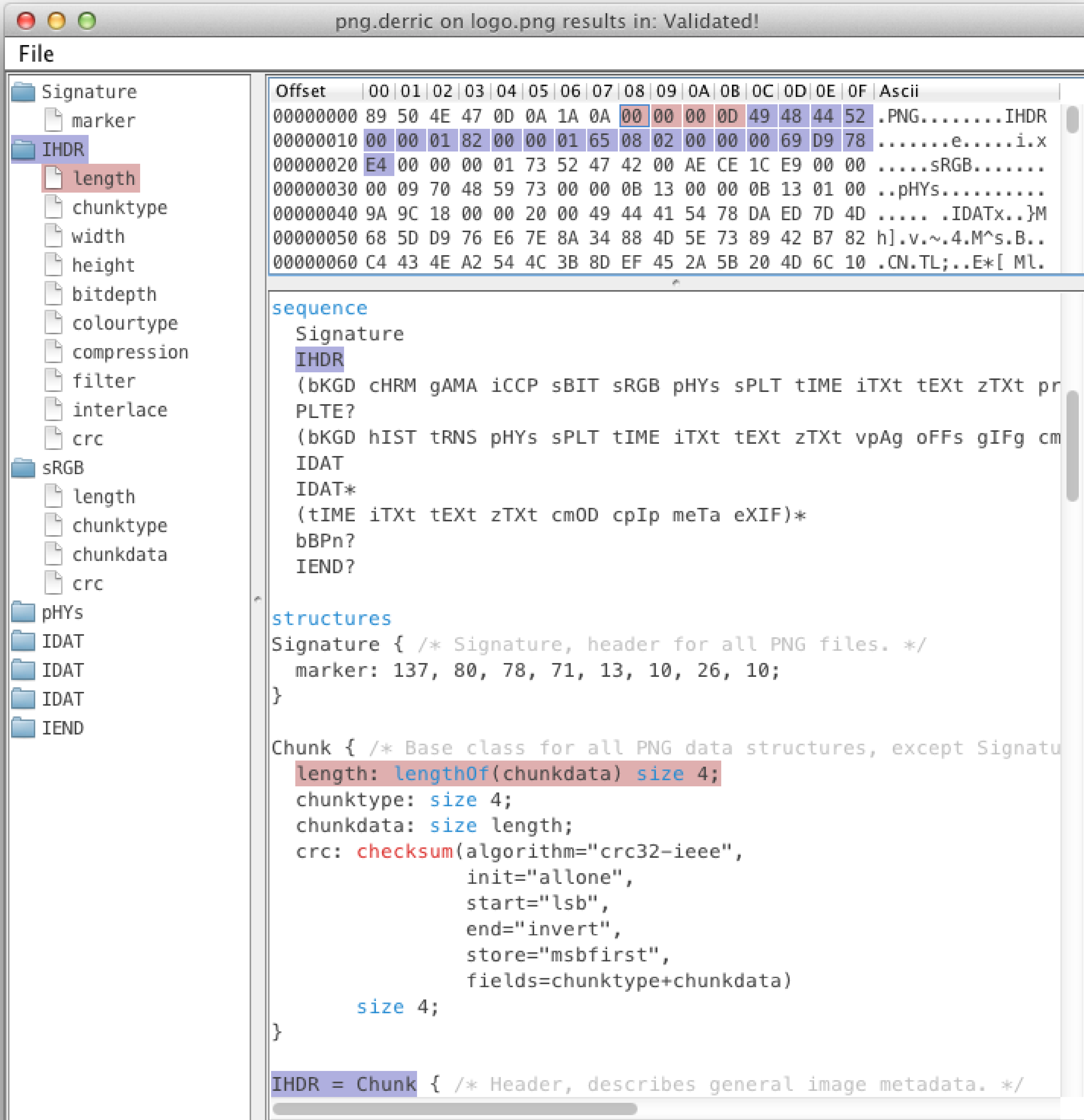
Domain-specific languages

Why live little languages?

EuSpRiG
European Spreadsheet Risks Interest Group

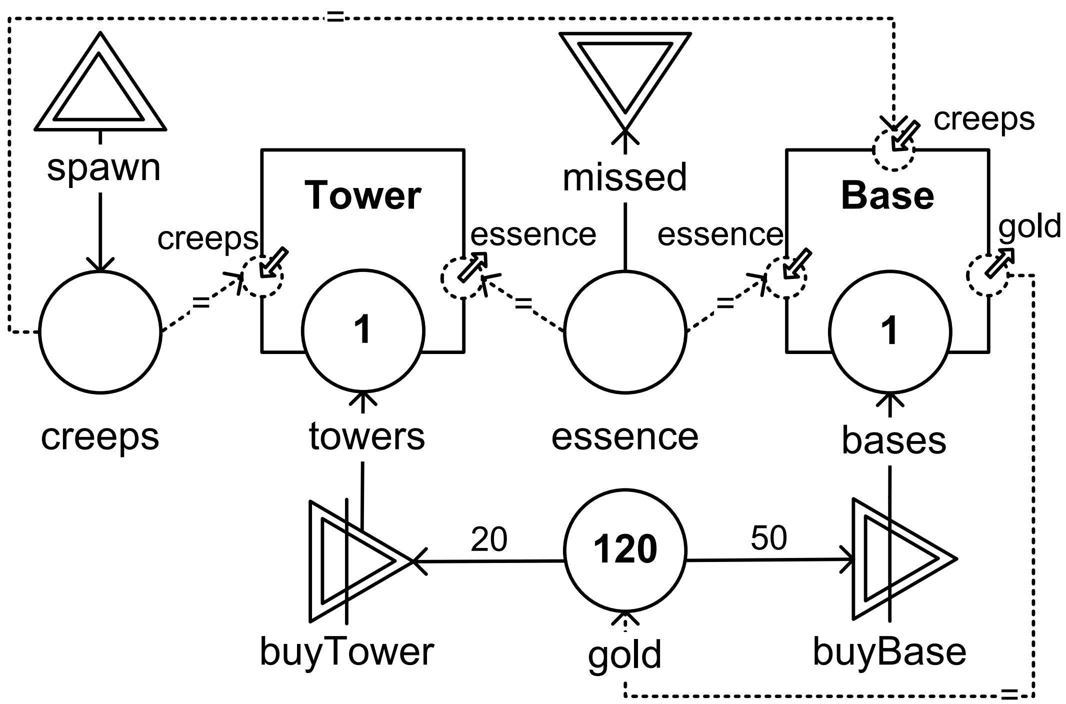
http://www.eusprig.org/horror-stories.htm

Derric: a DSL for digital forensics



Trinity: an IDE for the Matrix

Micro Machinations

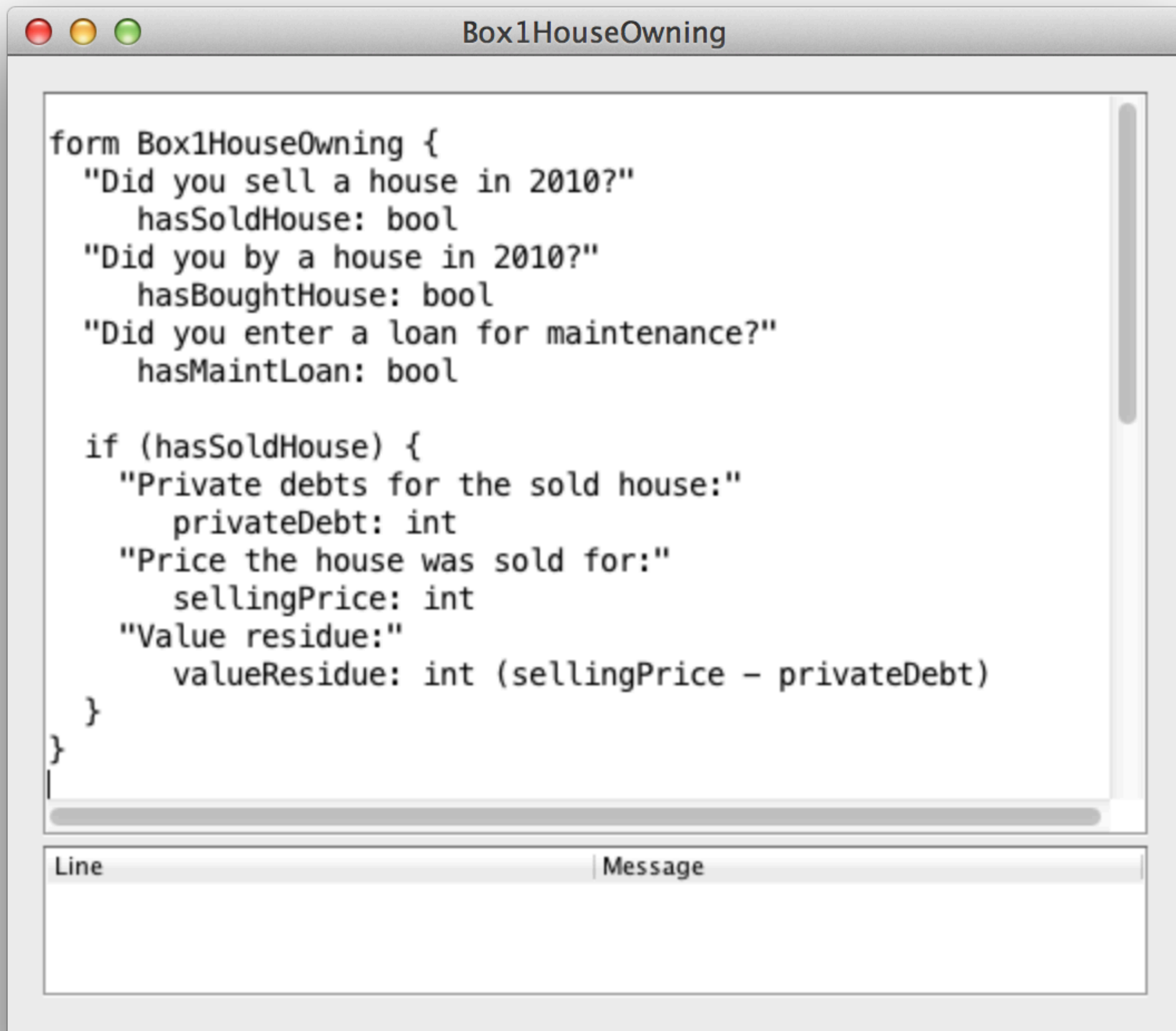


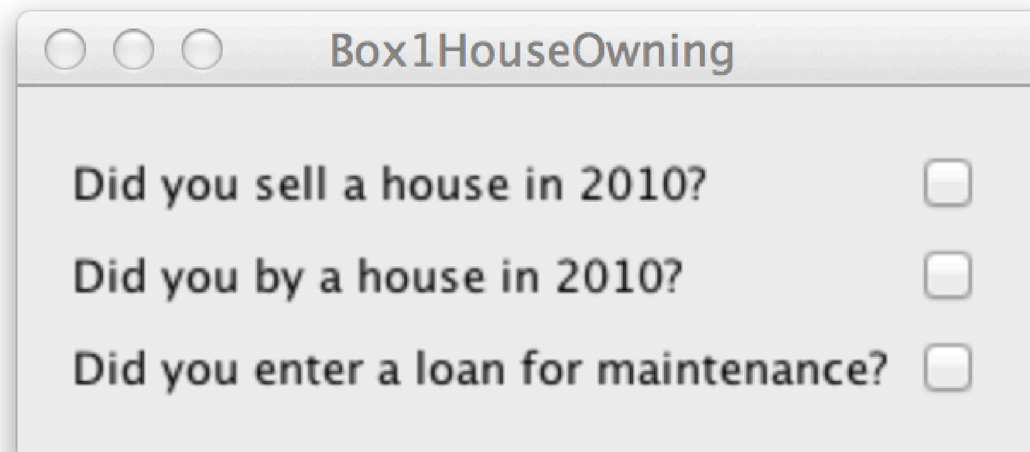
Van Rozen, Dormans, Adapting game mechanics with Micro-Machinations, FDG'14

Celldown: demo

```
table grades = #  A / B   / C    / D                 / E
               1:   | Lab | Exam | Avg               | Grade
               2:   | 7   | 7    | = (B2 + B3) / 2 | = round(D2)
               3:   | 3   | 7    | = (B3 + C3) / 2 | = round(D3)
               4:   | 9   | 10   | = (B4 + C4) / 2 | = round(D4).


view grades = #  A / B   / C    / D   / E
              1:   | Lab | Exam | Avg | Grade
              2:   | 7   | 7    | 5.  | 5.
              3:   | 3   | 7    | 5.  | 5.
              4:   | 9   | 10   | 9.5 | 10..

test grades E2 * 2 == B2 + C2 expected 14., got 10.

repl for grades
> A2 + B2.
=> 7.0
>B2 + B2.
=> 7.0
>
```

Live QL





van der Storm, *Semantic Deltas for Live DSL Environments*, LIVE'13

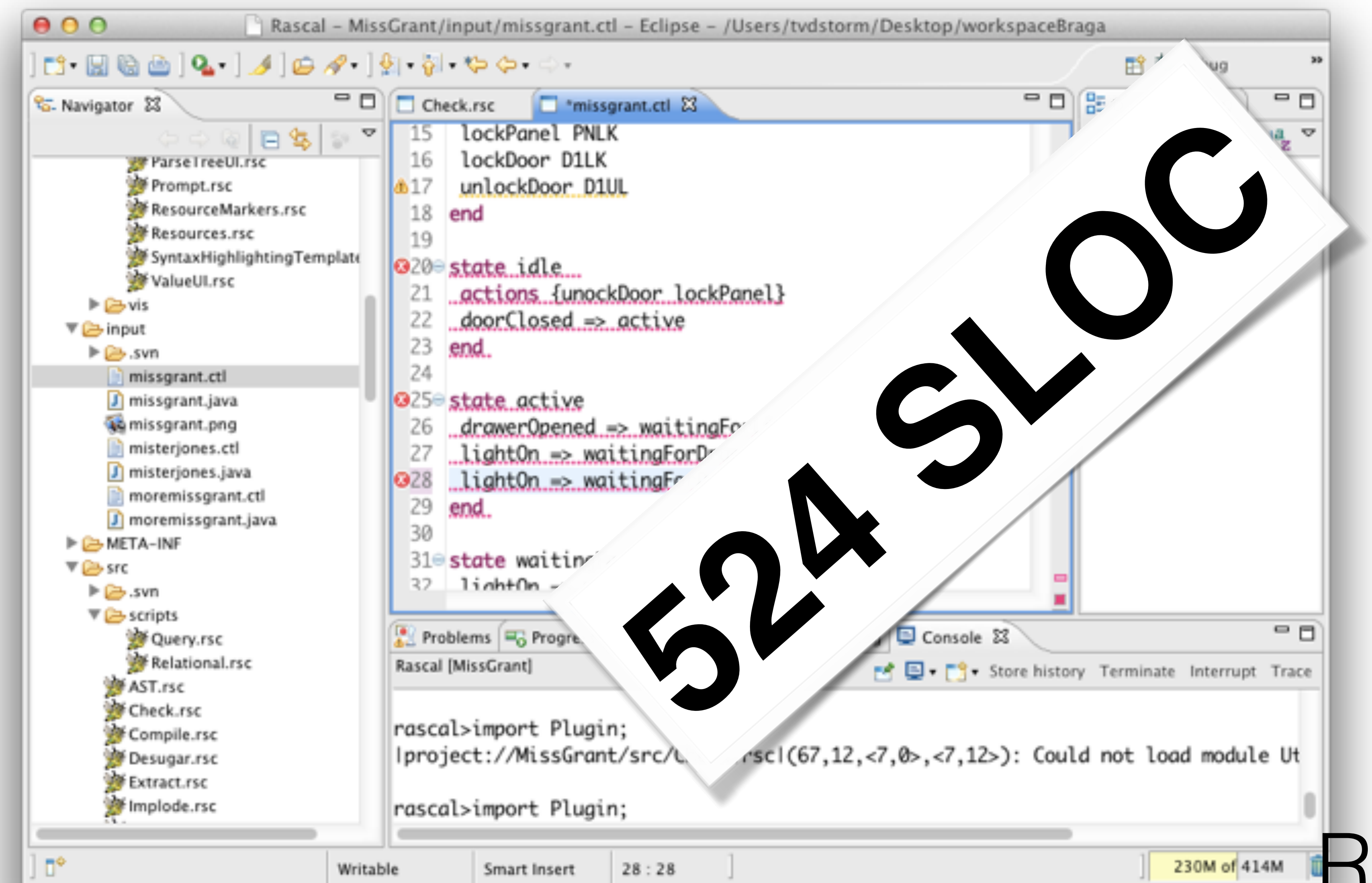
Live little languages

- Trinity: runtime data and source program are inter linked
- Machinations: game adapts as game mechanics is changed
- Celldown: data, computation, test, repl etc. all in a single, integrated interface (in this case: text)
- LiveQL: source changes have immediate effect on the questionnaire

Language workbenches

- IDE + meta-language(s) to build languages + IDEs
- Power tools for building DSLs
- Our workbench of choice: Rascal
- Productivity game changer

State machine DSL in Rascal



- Concrete syntax
- Abstract syntax
- Unparse
- Desugaring
- Checking
- Outline
- Hyperlinking
- Compilation
- Visual simulation
- Rename refactoring
- Parallel merge

No liveness :-(

- Compiler typically is a **batch** transformation system
- No notion of **interacting** with the system as **whole**
- Edit/compile/restart is **slow** and loses **runtime state**
- Disconnect between **generated code** and input

Research questions

- What are generic concepts and techniques for linking and integrating program and runtime?
  - => origin tracking, bidirectional transformation
- What are generic concepts and techniques for continuous feedback?
  - => incremental updates, coupled transformations,…
- How to support building live languages in language workbenches?

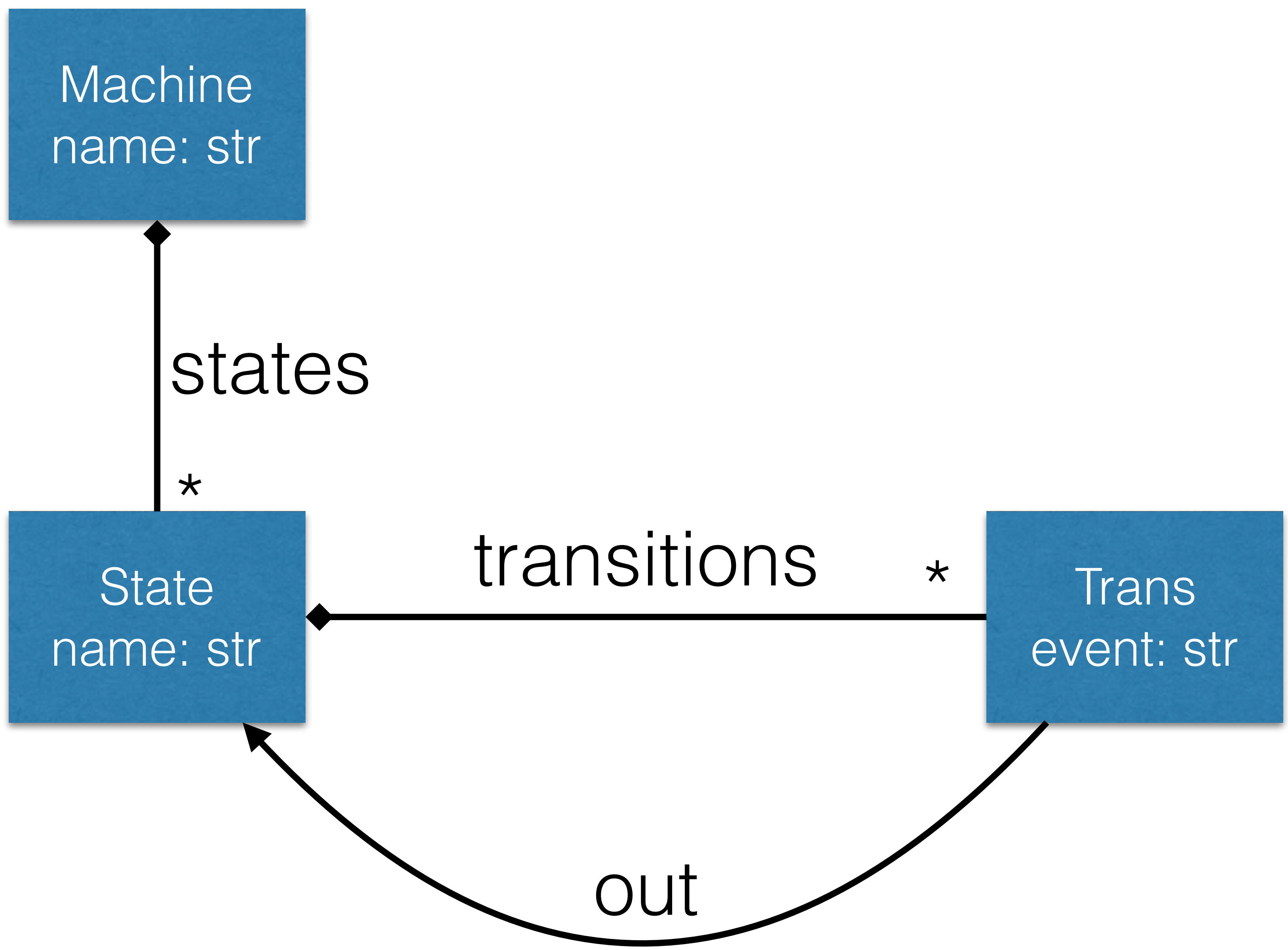
Semantic Deltas

- Represent programs as models
- Execution = interpreting model + state
- Editing program => *semantic delta*
- Interpret the delta at runtime
- Migrate runtime state where needed

State machines

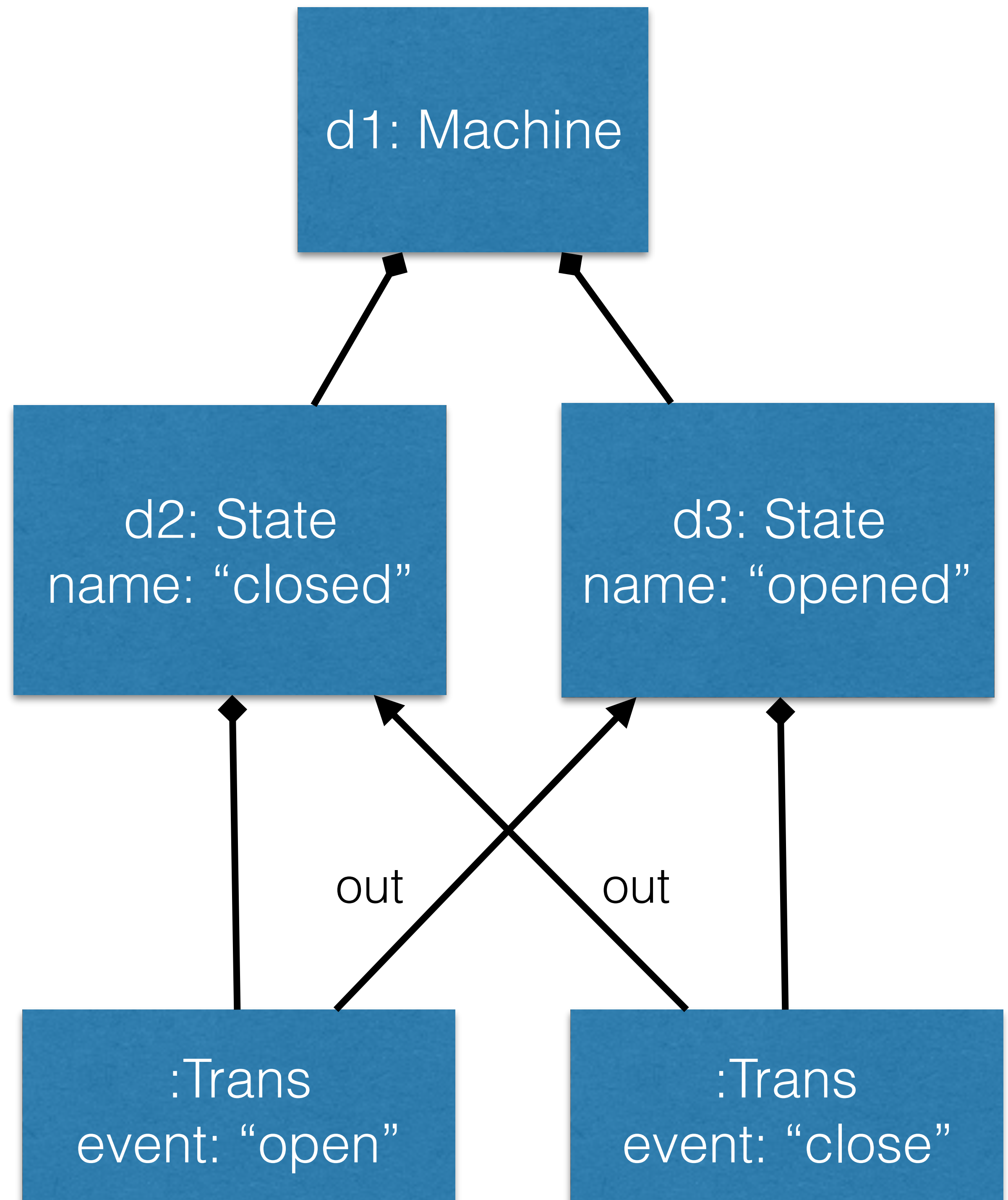
```
machine doors d1
  state closed d2
    open => opened u1

  state opened d3
    close => closed u2


end
```

Static meta model

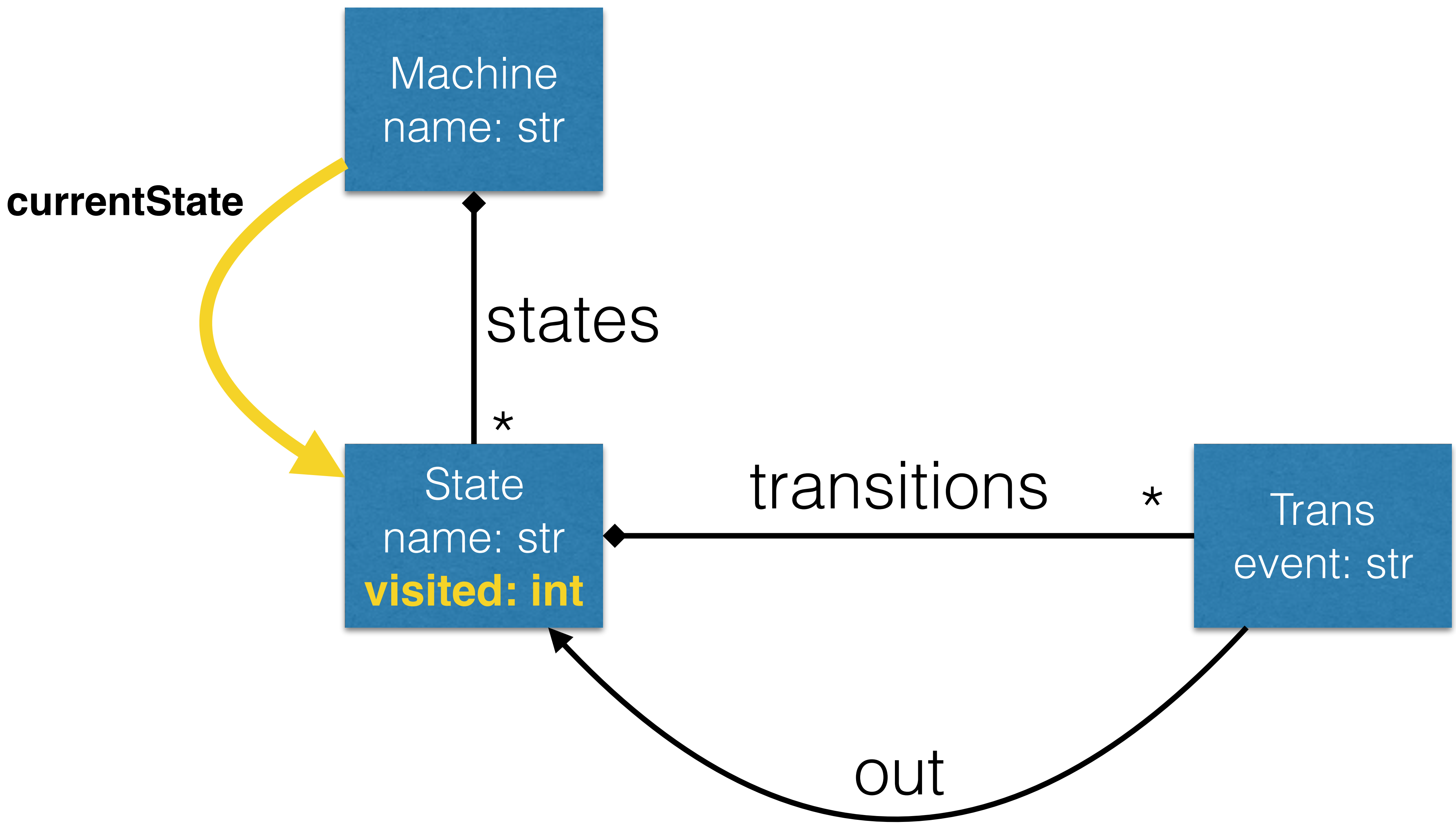


Static model

```
machine doors d1
  state closed d2
    open => opened u1

  state opened d3
    close => closed u2

end
```

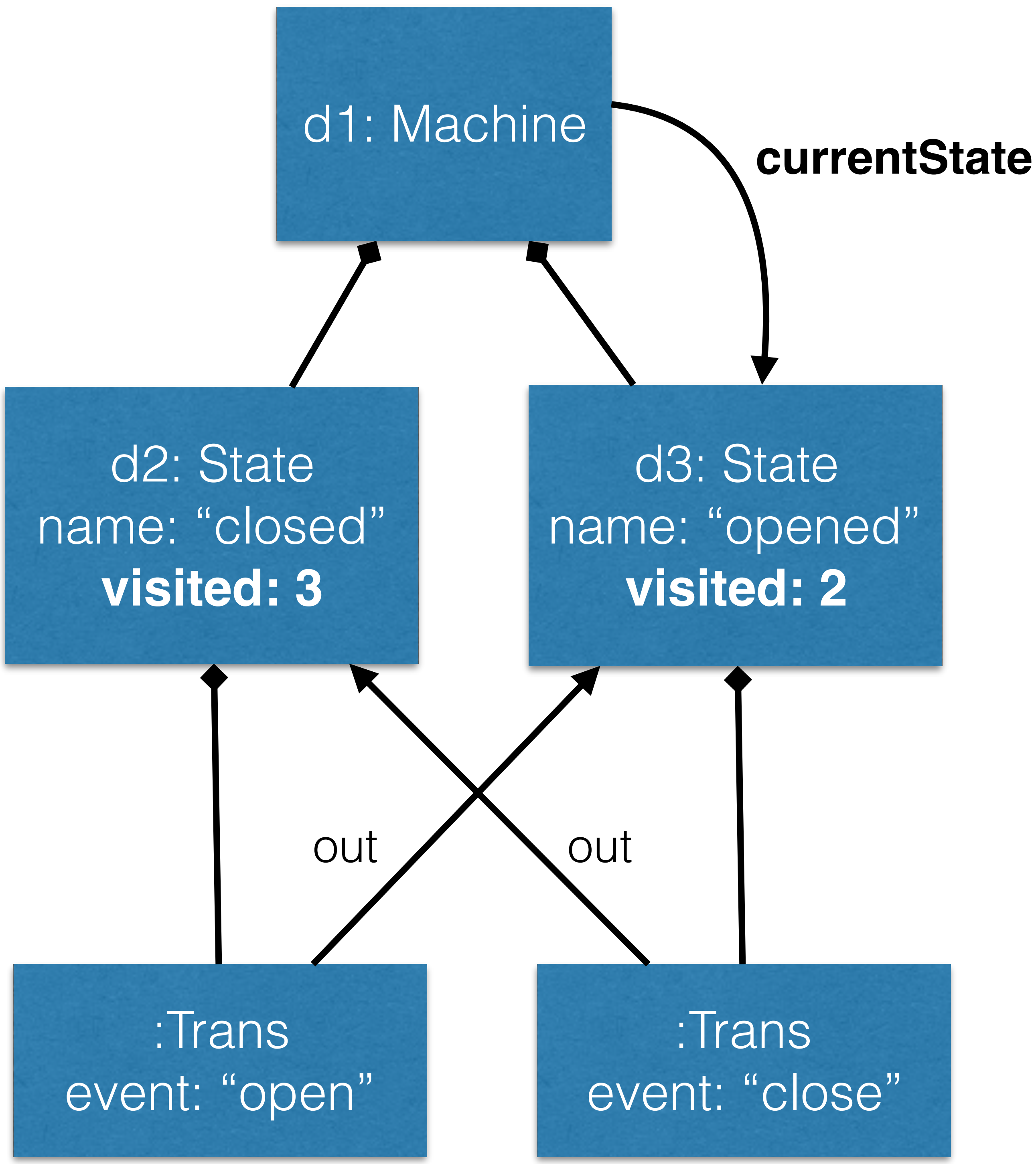


Runtime meta model



Runtime model

```
machine doors d1
  state closed d2
    open => opened u1

  state opened d3
    close => closed u2

end
```



```
machine doors d1
  state closed d2
    open => opened u1

  state opened d3
    close => closed u2

end
```

```
machine doors d4
  state closed d5
    open => opened u3
    lock => locked u4

  state opened d6
    close => closed u5

  state locked d7
    unlock => closed u6

end
```

diff (

```
machine doors d1
  state closed d2
    open => opened u1

  state opened d3
    close => closed u2

end
```

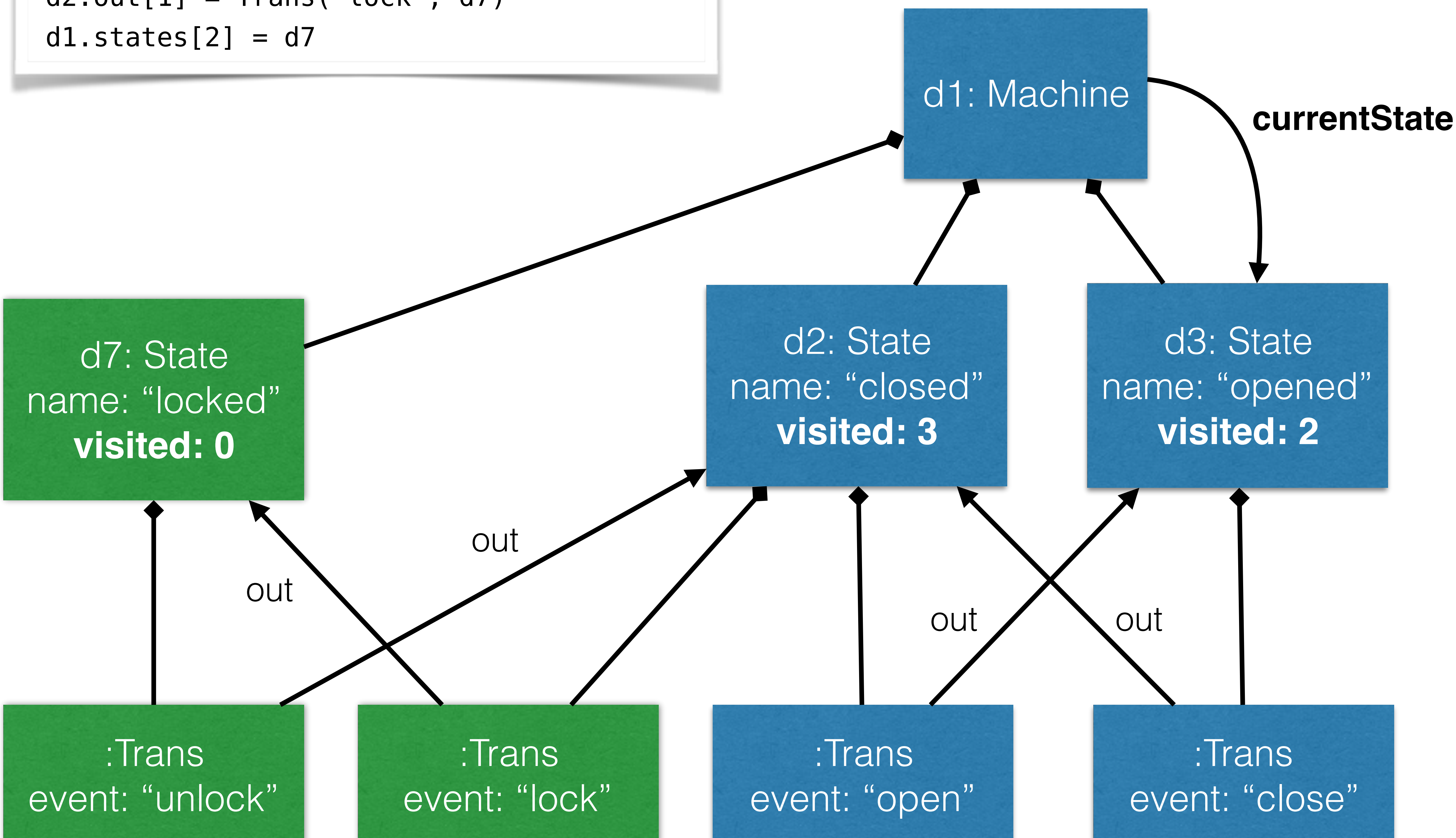
,

```
machine doors d4
  state closed d5
    open => opened u3
    lock => locked u4

  state opened d6
    close => closed u5

  state locked d7
    unlock => closed u6

end
```

) =

```
create State d7
d7 = State("locked",[Trans("unlock",d2)])
d2.out[1] = Trans("lock", d7)
d1.states[2] = d7
```

```
create State d7
d7 = State("locked",[Trans("unlock",d2)])
d2.out[1] = Trans("lock", d7)
d1.states[2] = d7
```



Demo: State machines

```
machine doors d1
  state closed d2
    open => opened u1

  state opened d3
    close => closed u2

end
```

Screenshot

Future directions

- Time travel (undo, replay)
- Time branching (what-if scenarios)
- Versioning (operation-based)
- Persistence (EventStores!)
- Collaboration (operational transformation)

Live little languages

- DSLs have been shown to be effective for SE
- Live = continuous feedback during programming
- Want: generic techniques for live DSLs
- Need: foundations and engineering principles
- Semantic deltas promising first step