

Live Little Languages

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

Colloquium at  / **university of
 groningen** 10-6-2015

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>

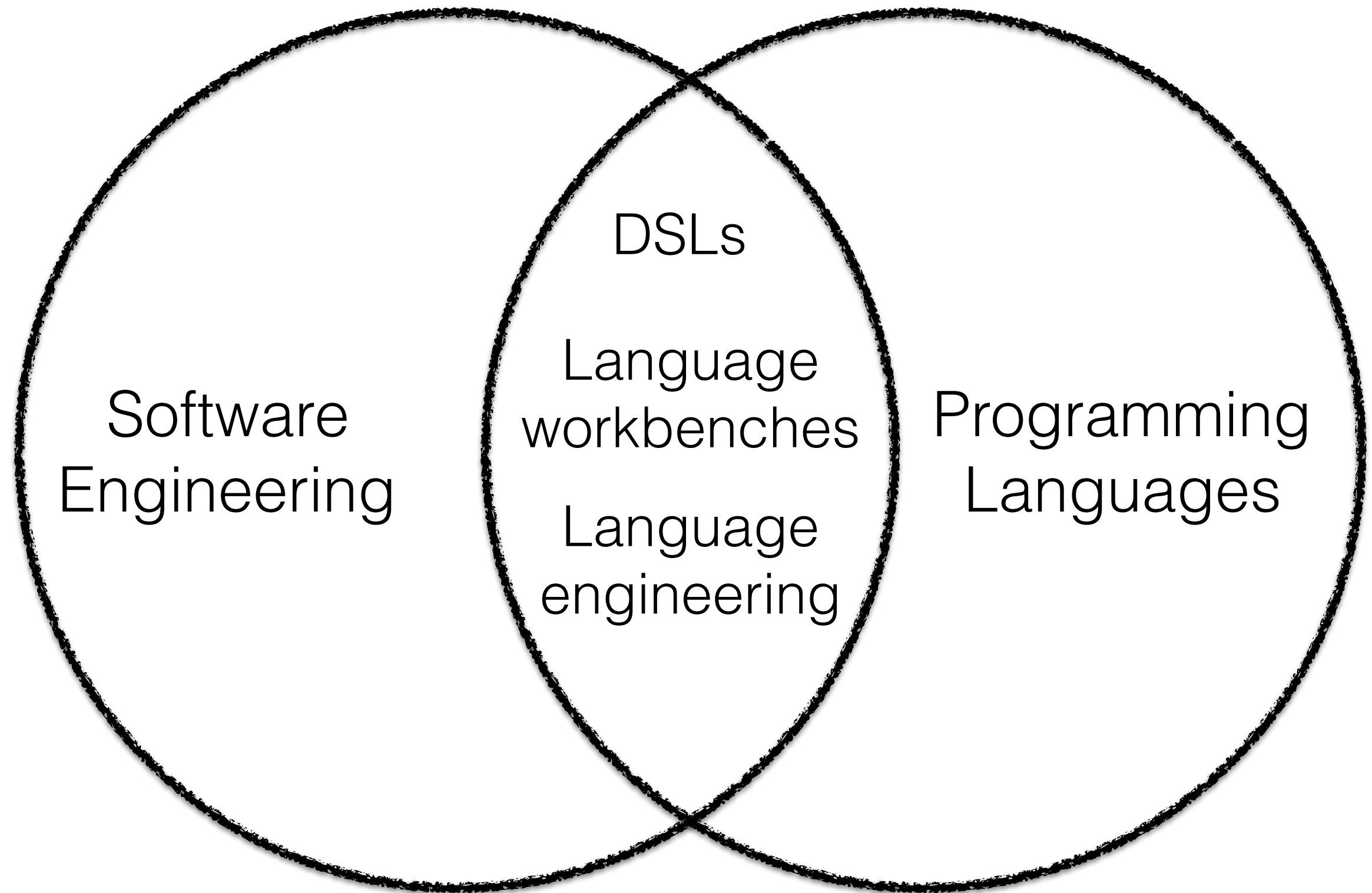


Ensō



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





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?



Richard Paige

@richpaige

300+ million Excel users, 9M Java users. Ouch. #sems15

18/5/15, 15:13

11

RETWEETS

6

FAVORITES



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

png.derric on logo.png results in: Validated!

File

- Signature
 - marker
- IHDR
 - length
 - chunktype
 - width
 - height
 - bitdepth
 - colourtype
 - compression
 - filter
 - interlace
 - crc
- sRGB
 - length
 - chunktype
 - chunkdata
 - crc
- pHYs
- IDAT
- IDAT
- IDAT
- IEND

Offset	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Ascii
00000000	89	50	4E	47	0D	0A	1A	0A	00	00	00	0D	49	48	44	52	.PNG.....IHDR
00000010	00	00	01	82	00	00	01	65	08	02	00	00	00	69	D9	78e....i.x
00000020	E4	00	00	00	01	73	52	47	42	00	AE	CE	1C	E9	00	00sRGB.....
00000030	00	09	70	48	59	73	00	00	0B	13	00	00	0B	13	01	00	..pHYs.....
00000040	9A	9C	18	00	00	20	00	49	44	41	54	78	DA	ED	7D	4DIDATx..}M
00000050	68	5D	D9	76	E6	7E	8A	34	88	4D	5E	73	89	42	B7	82	h].v.~.4.M^s.B..
00000060	C4	43	4E	A2	54	4C	3B	8D	EF	45	2A	5B	20	4D	6C	10	.CN.TL;..E*[Ml.

sequence

```

Signature
IHDR
(bKGD cHRM gAMA iCCP sBIT sRGB pHYs sPLT tIME iTXt tEXt zTXt pr
PLTE?
(bKGD hIST tRNS pHYs sPLT tIME iTXt tEXt zTXt vpAg oFFs gIFg cm
IDAT
IDAT*
(tIME iTXt tEXt zTXt cmOD cpIp meTa eXIF)*
bBpN?
IEND?
  
```

structures

```

Signature { /* Signature, header for all PNG files. */
  marker: 137, 80, 78, 71, 13, 10, 26, 10;
}

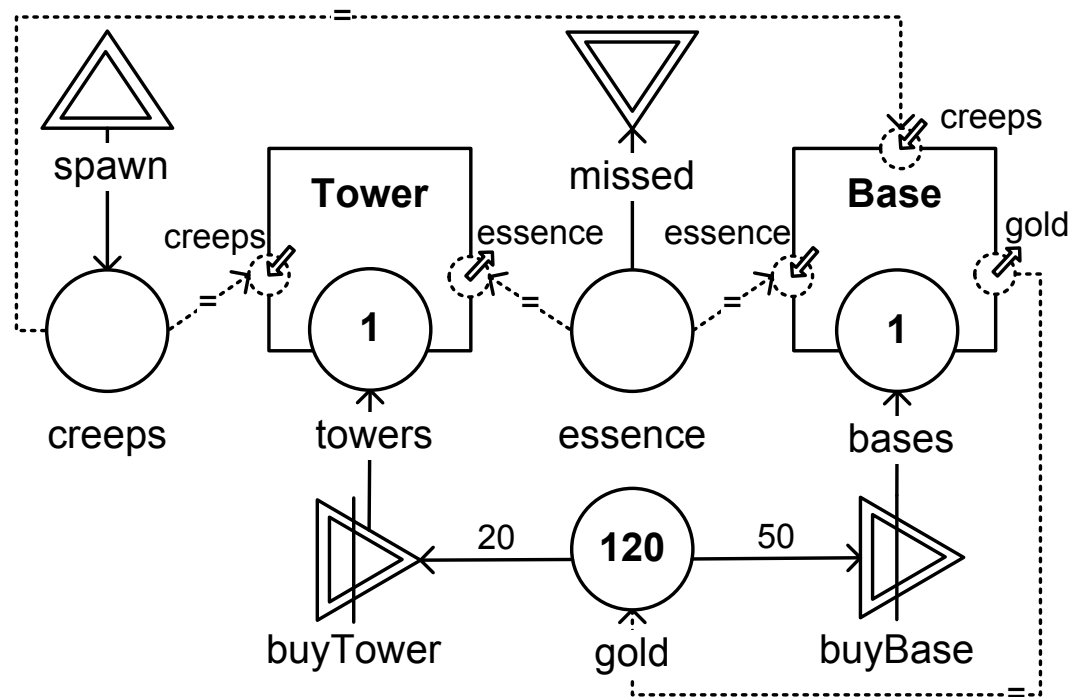
Chunk { /* Base class for all PNG data structures, except Signatu
  length: lengthOf(chunkdata) size 4;
  chunktype: size 4;
  chunkdata: size length;
  crc: checksum(algorithm="crc32-ieee",
    init="allone",
    start="lsb",
    end="invert",
    store="msbfirst",
    fields=chunktype+chunkdata)
    size 4;
}

IHDR = Chunk { /* Header, describes general image metadata. */
  
```

Derric: a DSL for digital forensics

Trinity: an IDE for the Matrix

Micro Machinations



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.
```

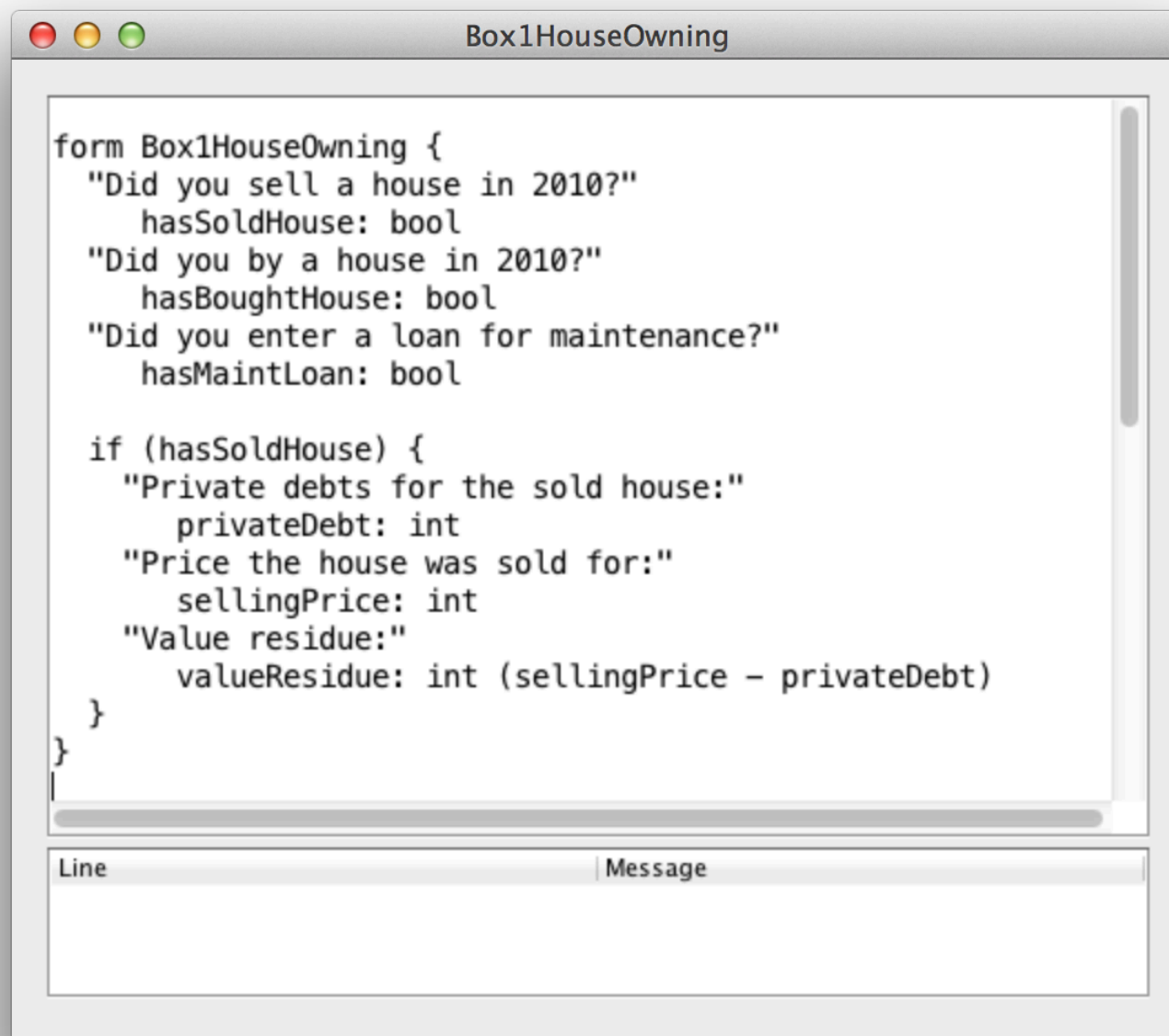
```
=> 7.0
```

```
>B2 + B2.
```

```
=> 7.0
```

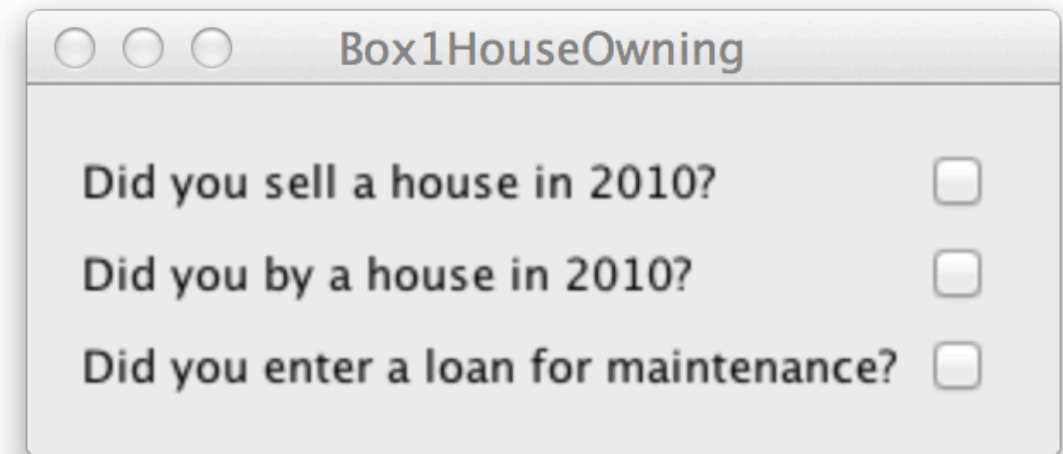
```
>
```

Live QL



```
form Box1HouseOwning {  
  "Did you sell a house in 2010?"  
    hasSoldHouse: bool  
  "Did you buy a house in 2010?"  
    hasBoughtHouse: bool  
  "Did you enter a loan for maintenance?"  
    hasMaintLoan: bool  
  
  if (hasSoldHouse) {  
    "Private debts for the sold house:"  
      privateDebt: int  
    "Price the house was sold for:"  
      sellingPrice: int  
    "Value residue:"  
      valueResidue: int (sellingPrice - privateDebt)  
  }  
}
```

Line	Message
------	---------



Box1HouseOwning

Did you sell a house in 2010? ☐

Did you buy a house in 2010? ☐

Did you enter a loan for maintenance? ☐

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

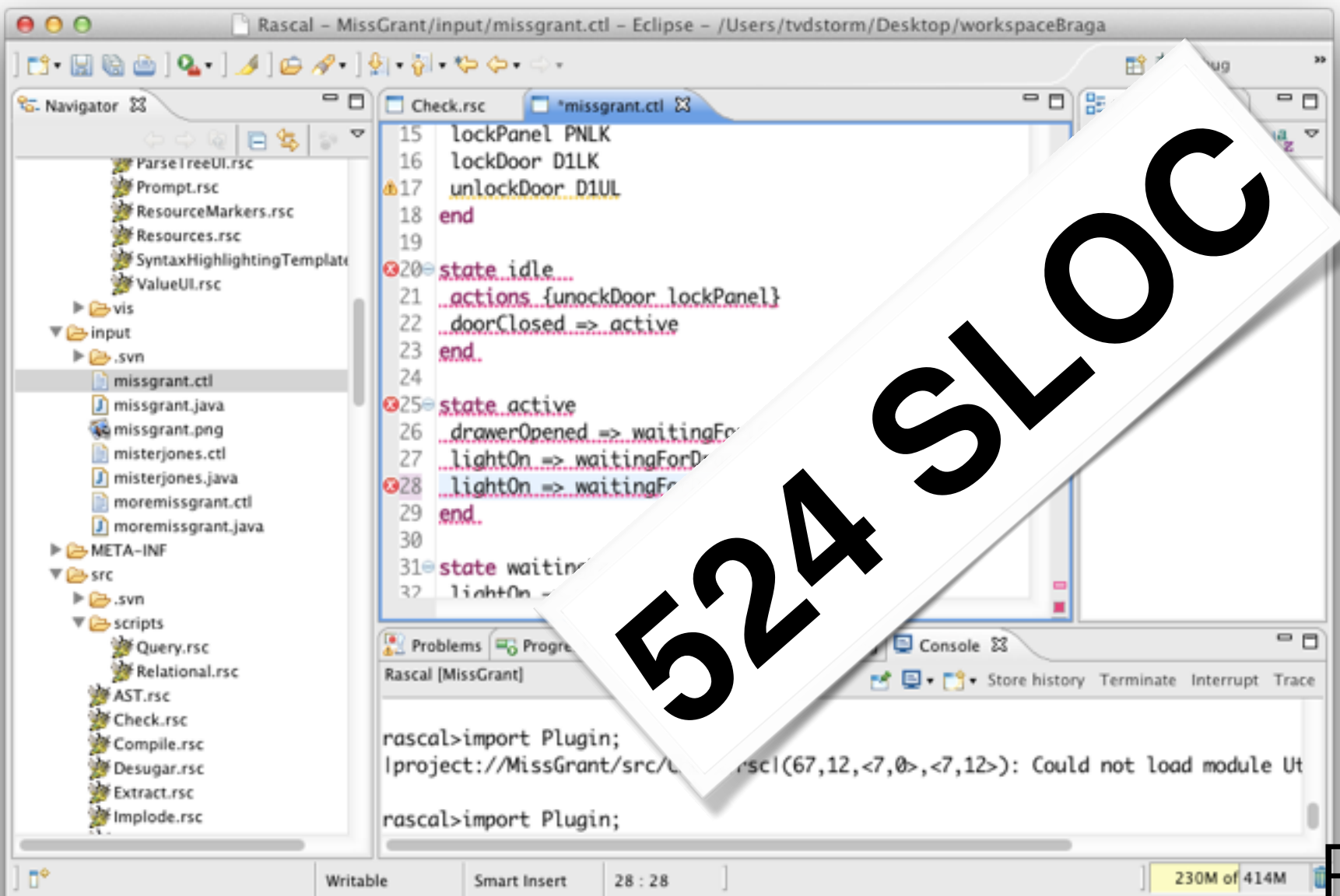
Hyperlinking

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 \Rightarrow *semantic delta*
- Interpret the delta at runtime
- Migrate runtime state where needed

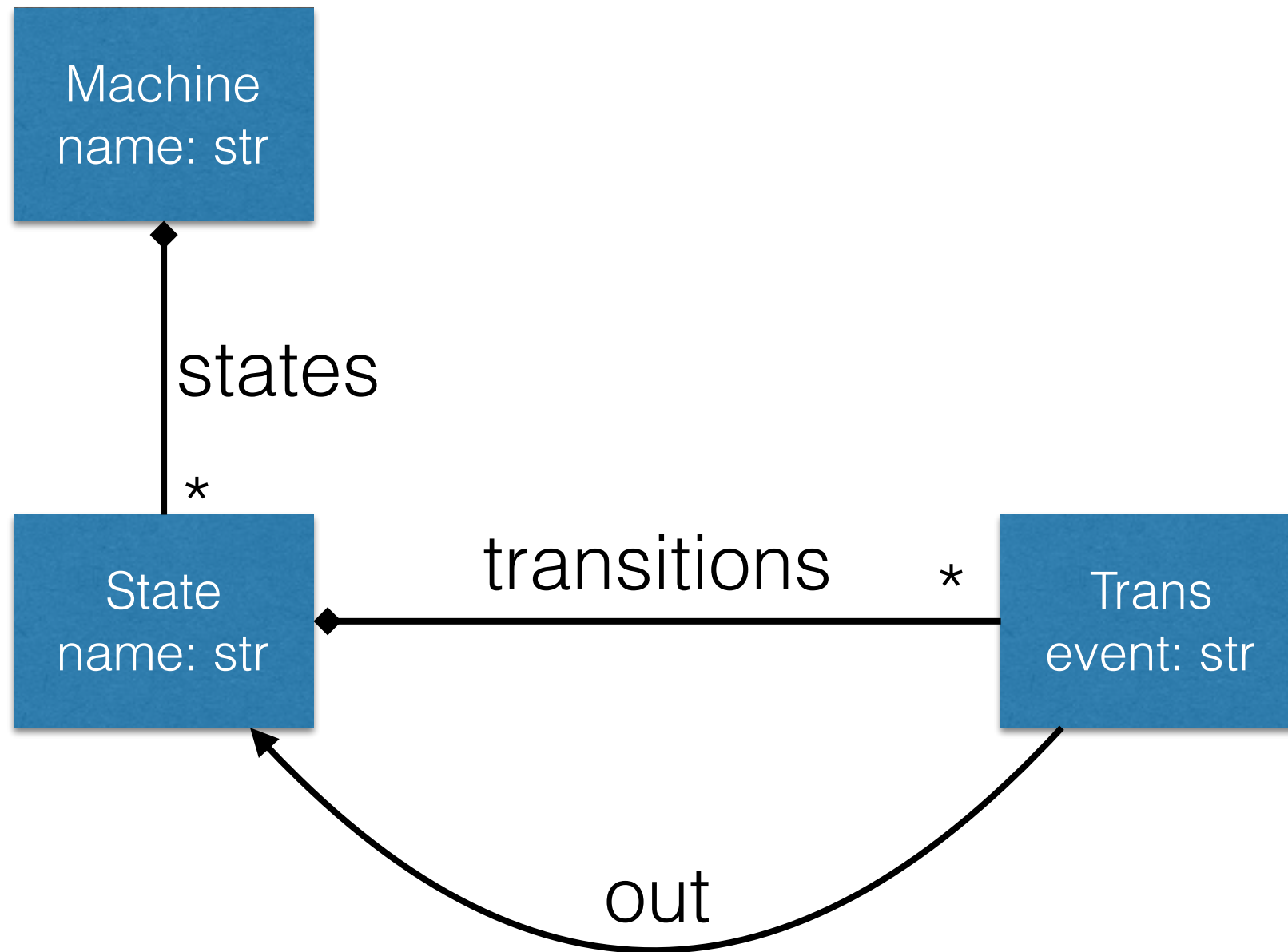
State machines

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

  state opened d3
    close => closed u2

end
```

Static meta model

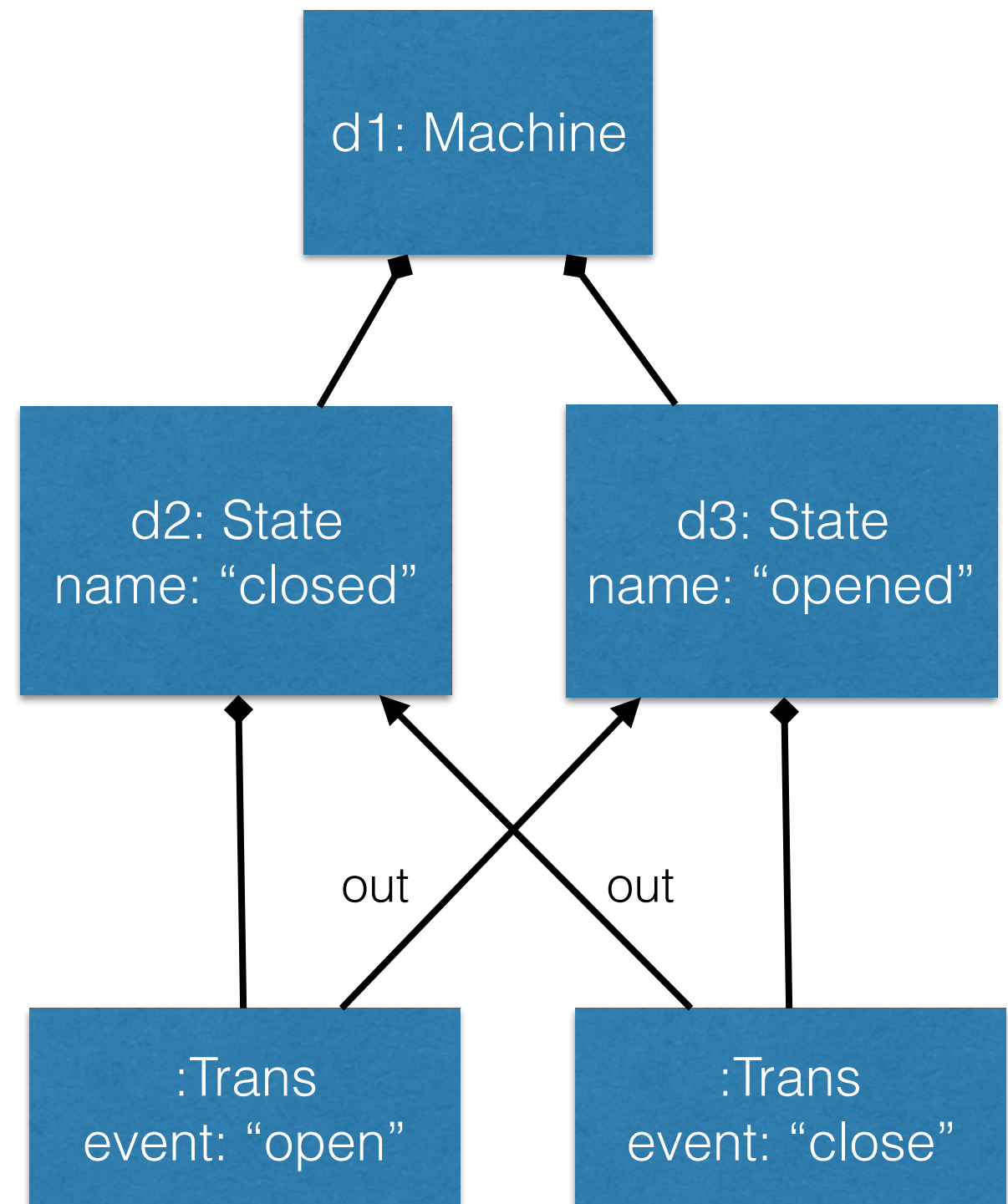


Static model

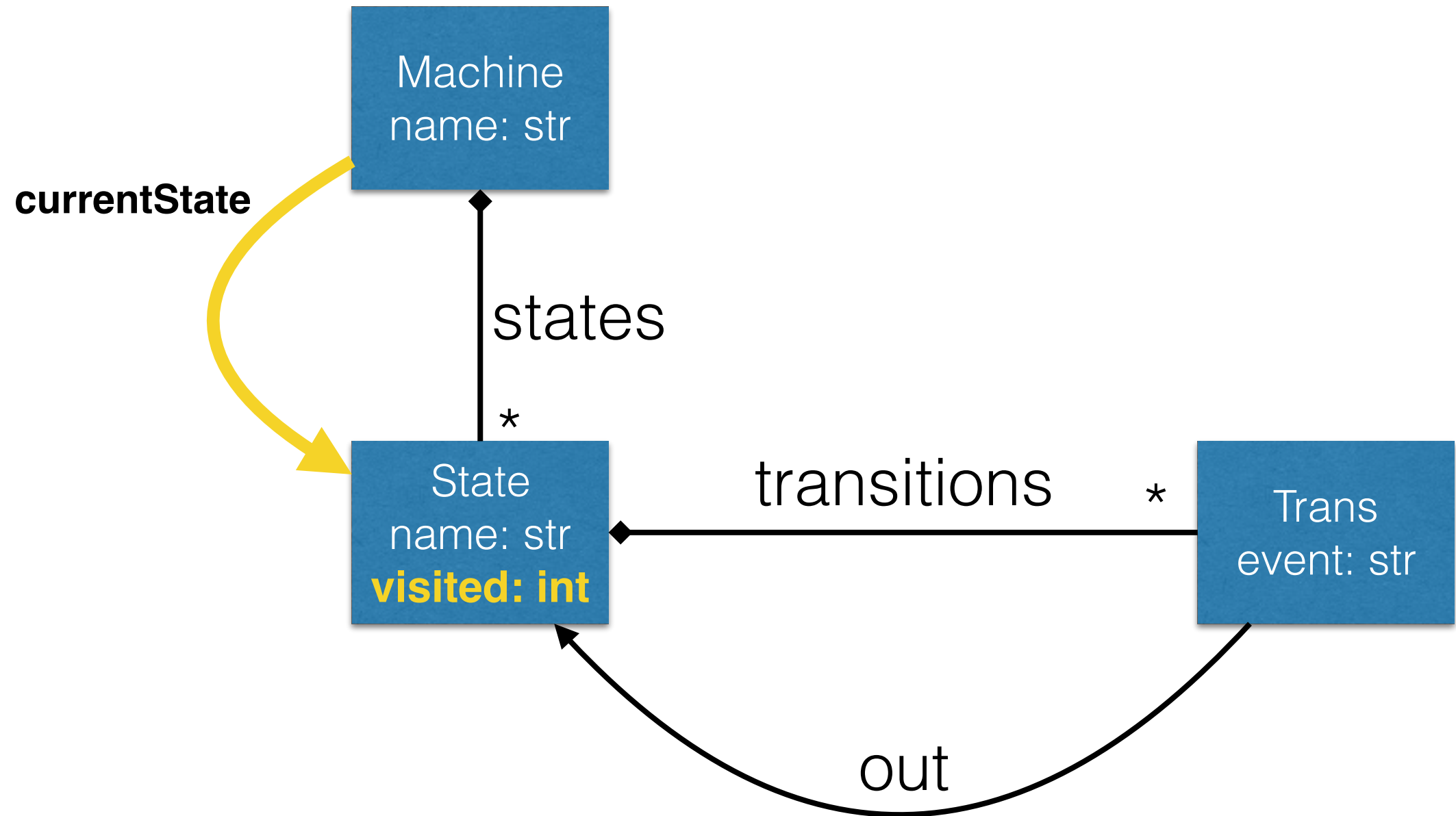
```
machine doors d1
  state closed d2
    open => opened u1

  state opened d3
    close => closed u2

end
```



Runtime meta model

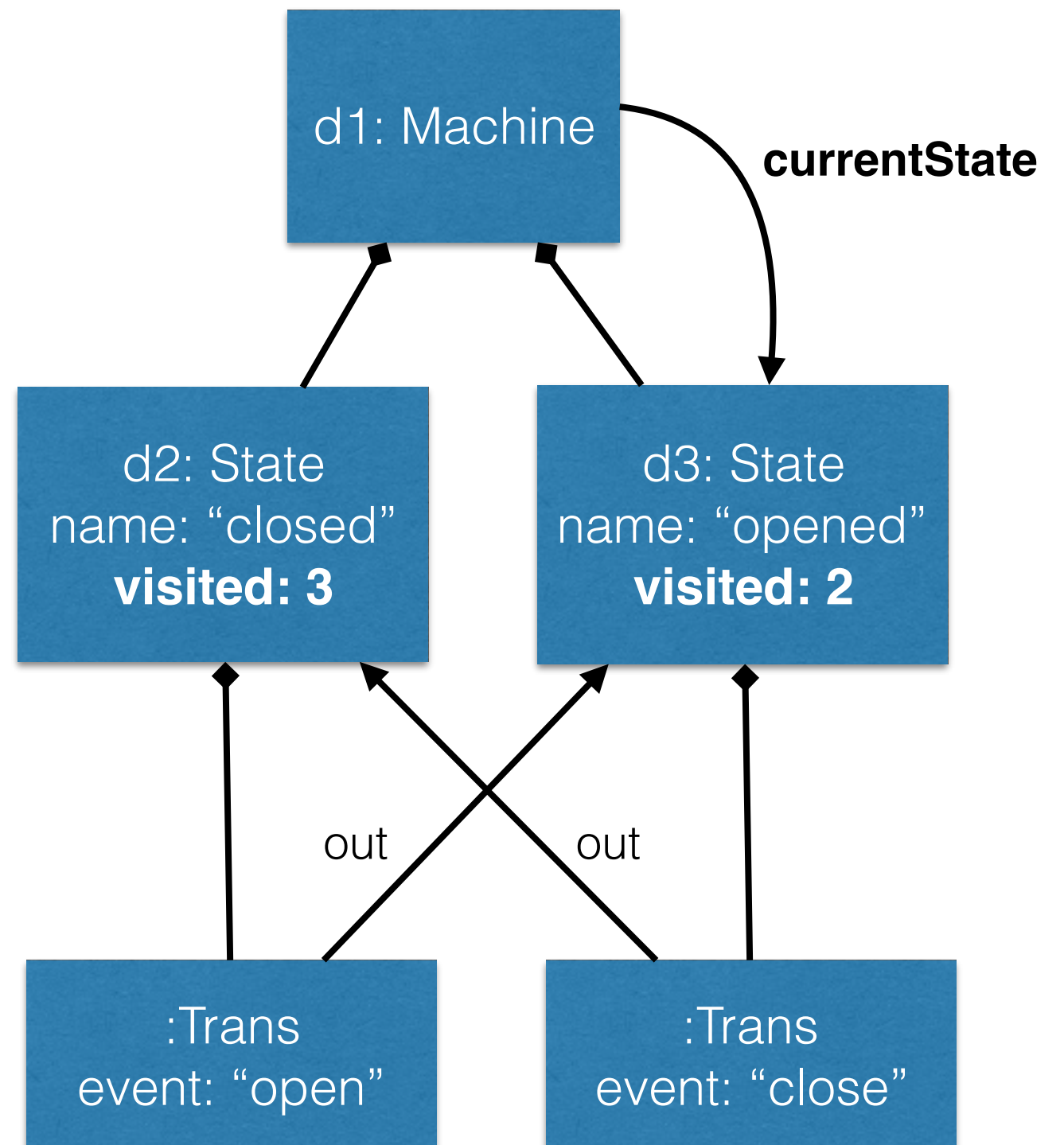


Runtime model

```
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

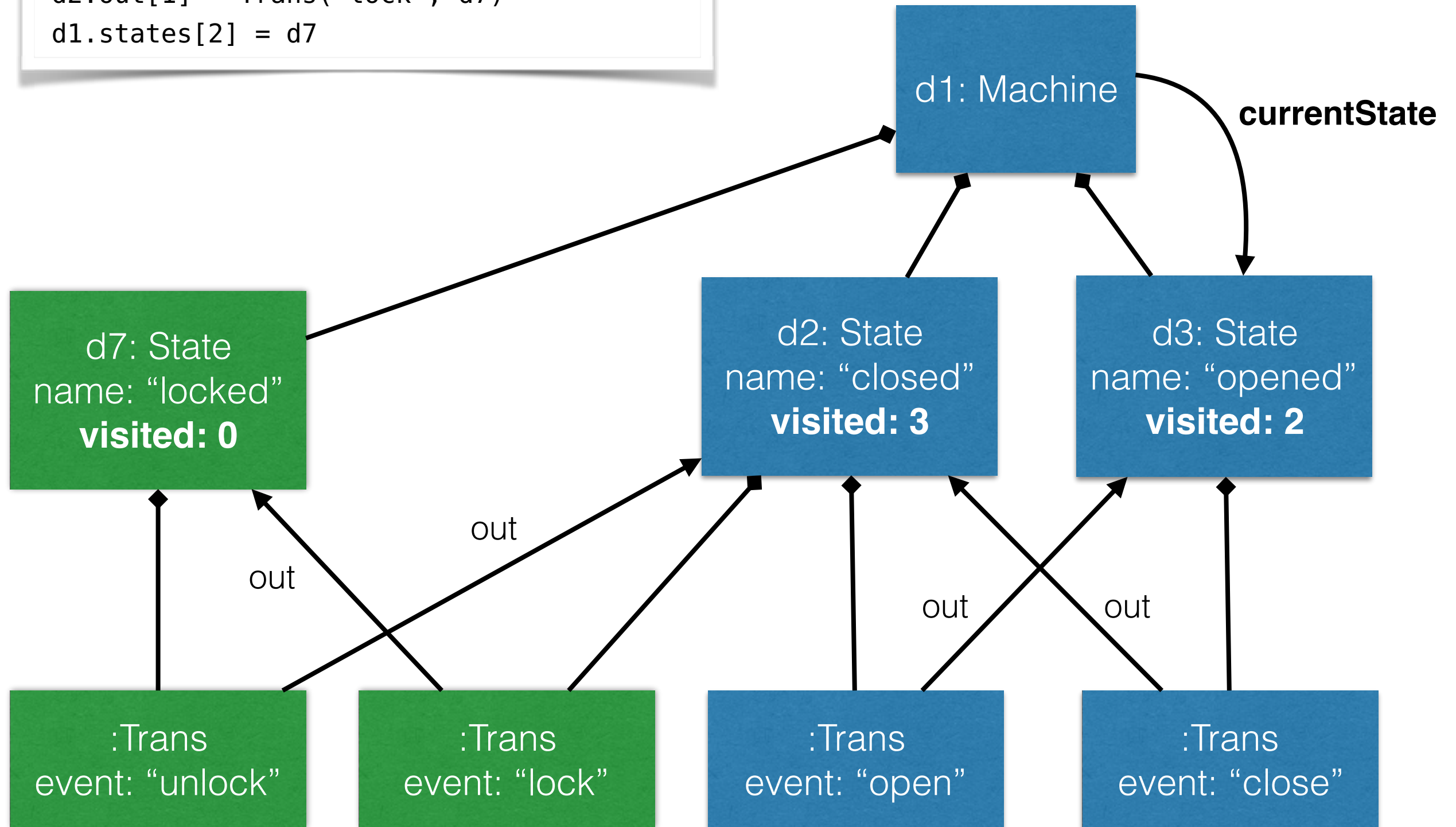
end
```

)

=

```
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

