Centrum Wiskunde & Informatica

Hogeschool van Amsterdam
Amsterdam University of Applied Sciences

Model Evolution

Towards Live Domain-Specific Languages

IPA Fall Days – Garderen, Friday November 2nd 2018

Riemer van Rozen[1,2,3] **@rvrozen**

joint work with Tijs van der Storm[2,4] **@tvdstorm**

[1] Amsterdam University of Applied Sciences (HvA)

[2] Centrum Wiskunde & Informatica (CWI)

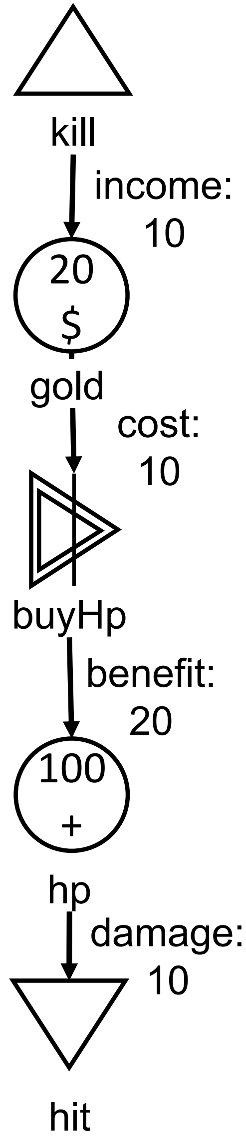
[3] University of Amsterdam – Master SE (UvA)

[4] University of Groningen (RUG)

Textual Models

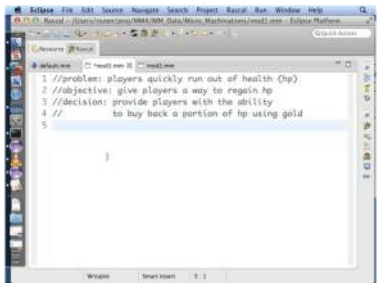
- Models encoded as text
  - Textual DSLs
  - Programming Languages
- **DSL for the Game Domain:** Micro-Machinations is a language and library that enables game designers to modify a game's rules at run-time.
- **Example:** Johnny Jetstream

```
source kill
income: kill -10-> gold
pool gold is "$" at 20
cost: gold -10-> buyHp
user converter buyHp
benefit: buyHp -20-> hp
pool hp is "+" at 100
damage: hp -10-> hit
drain hit
```



Step 1: Play Test v1

Step2: Re-design



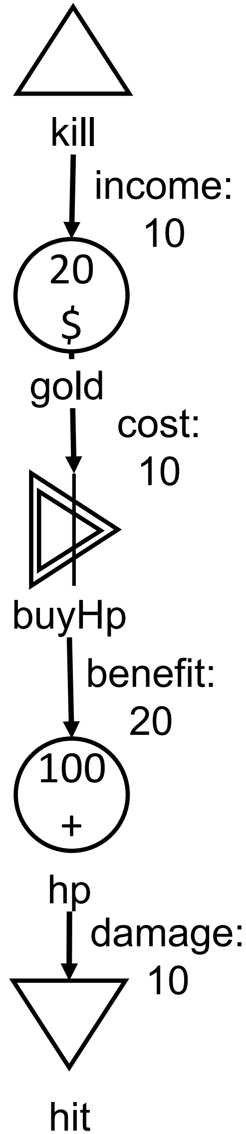
Step 3: Play Test v2

Textual Models

- Evolution perspective
  - Changes between different versions of a program
  - Live DSLs modify running programs

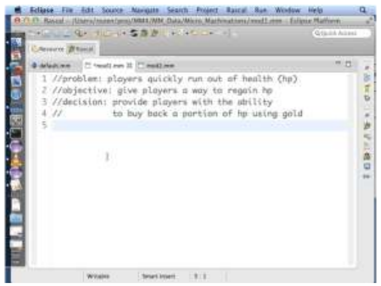
- How to (1) determine the difference between two textual models and (2) evolve running programs?

```
source kill
income: kill -10-> gold
pool gold is "$" at 20
cost: gold -10-> buyHp
user converter buyHp
benefit: buyHp -20-> hp
pool hp is "+" at 100
damage: hp -10-> hit
drain hit
```



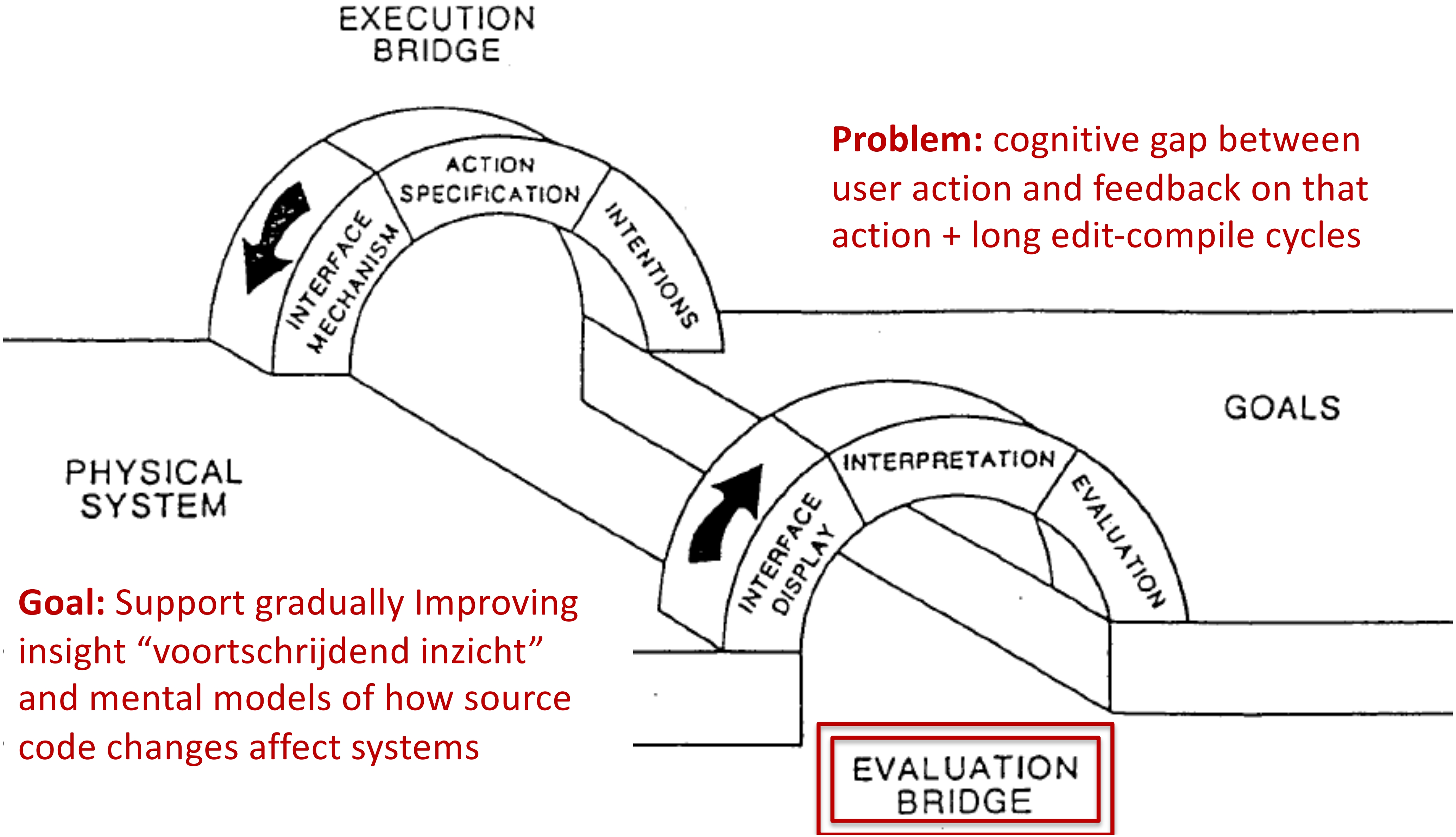
Step 1: Play Test v1

Step2: Re-design



Step 3: Play Test v2

Live Modeling aims to bridge the *"gulf of evaluation"* (D. Norman 1988)

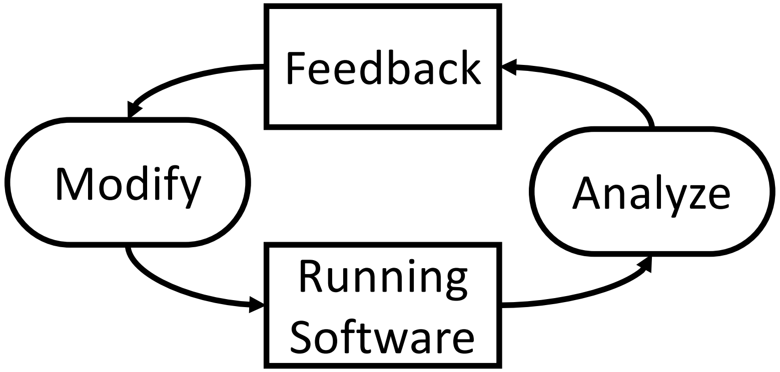


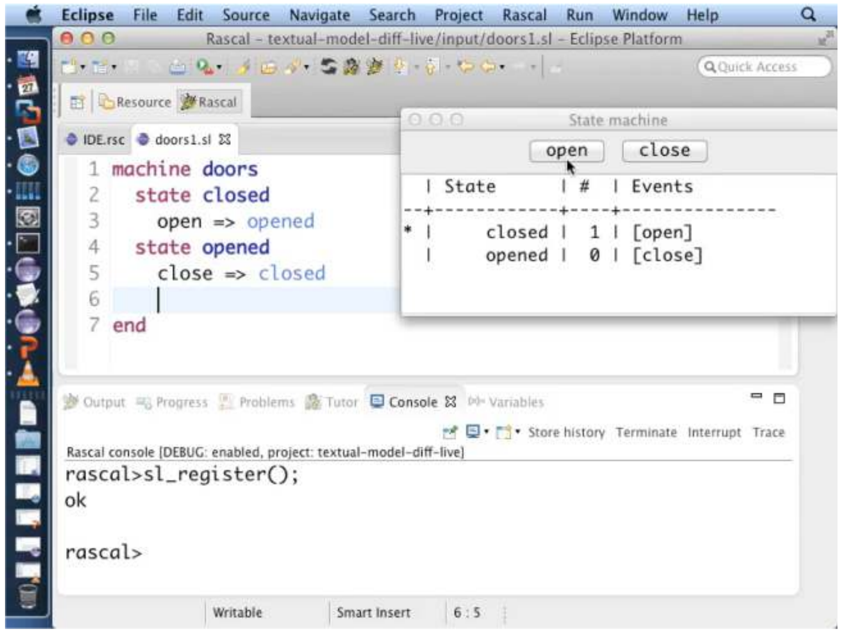
Problem: cognitive gap between user action and feedback on that action + long edit-compile cycles

Goal: Support gradually Improving insight "voortschrijdend inzicht" and mental models of how source code changes affect systems

Live Programming

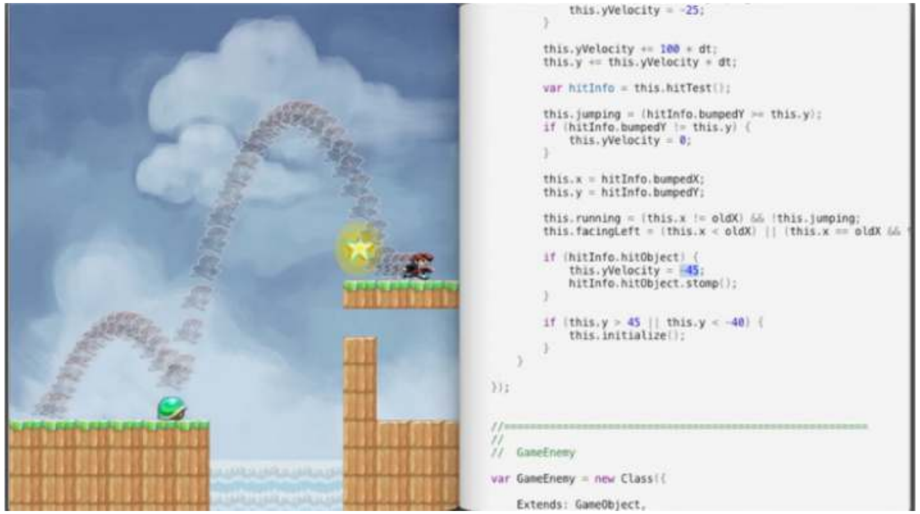
- Live programming aims to bridge the gulf of evaluation by shortening the feed-back loop between editing a program's textual source code and observing its behavior.
- In a live programming environment, the running program is updated instantly after every change in the code.
  - see the behavioral effects of actions immediately
  - learn predicting how the program adapts to targeted improvements to the code
- **Question**: how to bridge the gap between running programs and textual DSLs?





Suggestion: Not State Machines

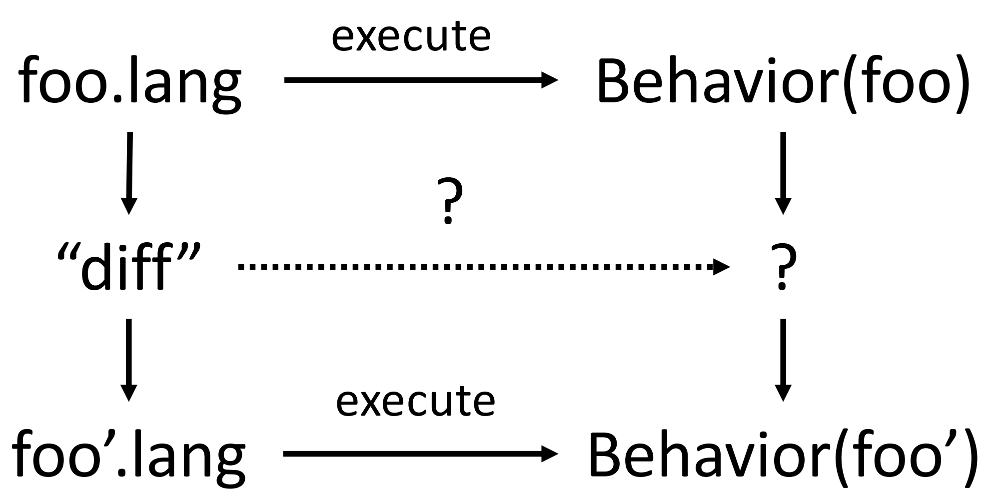
- Games Research
  - *``Applications to games other than Super* ***Mario*** *Bros are especially welcome" – Call for papers of the Procedural Content Generation in Games Workshop*.
- Language Research
  - *``Applications to languages other than* ***State Machines*** *are especially welcome"* – future call for papers
- Suggested Alternatives
  - Behavior Trees http://aigamedev.com/open/article/behavior-trees-part1/
  - PuzzleScript https://www.puzzlescript.net
  - Machinations



Pros: state machines are simple, explainable, research can be compared
Cons: state machines may not be representative, tedious repetition

Problem Statement and Objectives

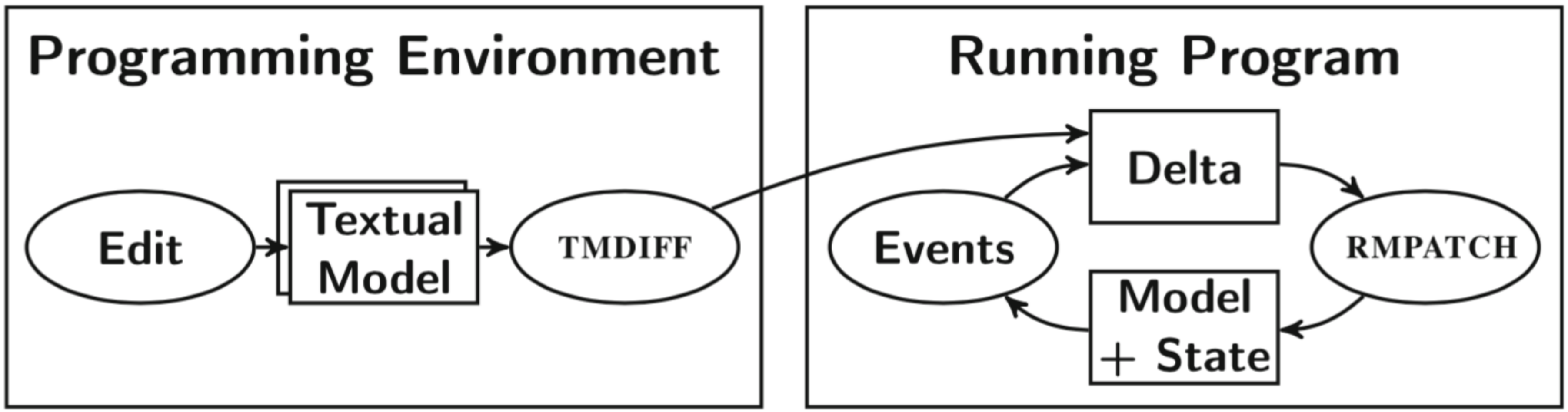
- **Challenge:** How to build DSLs for live programming?
- **Objective:** provide generic language technology for constructing DSLs for live programming
- **Question:** How can a textual difference between successive source code versions and origin tracking be leveraged for obtaining a run-time difference in behavior?



Approach

- **Approach:** Apply Textual Model Differencing (TMDiff) to obtain model-based deltas and Run-time Model Patching (RMPatch) to migrate models at run time.
  - Program migrations as part of the language semantics
  - One correct result of a state migration is assumed

$$
\begin{tikzcd}
\text{foo.lang} \arrow[r, "\text{parse/resolve}"] \arrow[d] & \text{MM} \arrow[r, "\text{execute}"] \arrow[d] & \text{MM+} \arrow[d] \\
\text{“diff”} \arrow[r, "\text{TMDiff}"] \arrow[d] & \Delta(\text{MM}) \arrow[r, "\boxed{\text{RMPatch}}"] & [\![\delta]\!] \arrow[d] \\
\text{foo'.lang} & & \text{MM+}
\end{tikzcd}
$$



Background:
Difference and Union of Models

UML, 2003

Marcus Alanen and Ivan Porres

Model Differencing

- Difference and Union of Models
  - **Context**: version control
  - **Motivation:** Two designers make separate changes to a model. How to merge the two models?

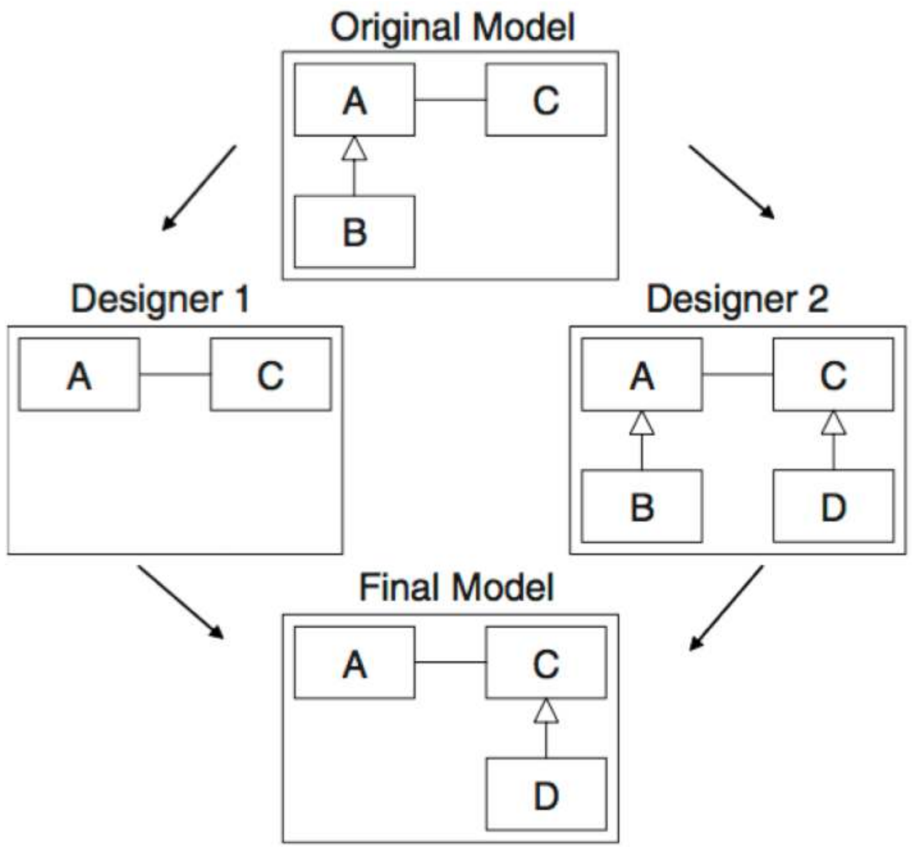


Fig. 1. Example of the Union of Two Versions of a Model

Source: Marcus Alanen and Ivan Porres. Difference and union of models. UML 2003.

Source: Ivan Porres, Difference and Union of Models, 10 years later (invited presentation). MODELS 2013

Model Differencing

- Difference and Union of Models
  - **Difference.** calculate the difference between two models. $M_2 - M_1 = \Delta$
  - **Union.** merging two models by applying the difference. $M_1 + \Delta = M_2$

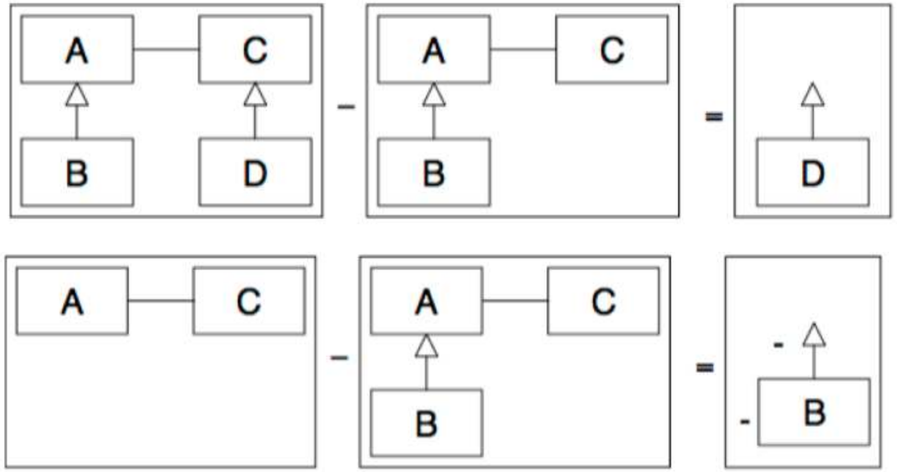


Figure 2: Example of the Difference of Models

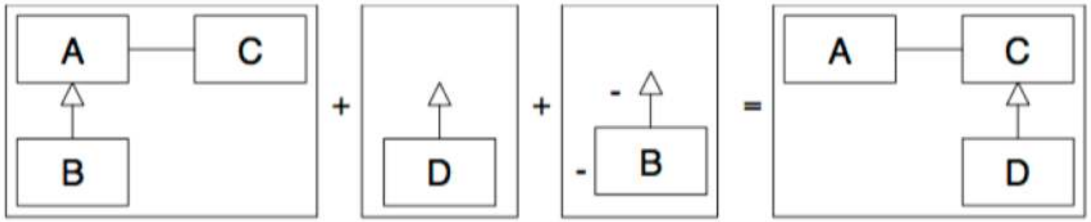


Figure 3: Example of the Union Based on Differences

Source: Marcus Alanen and Ivan Porres. Difference and union of models. UML 2003.
Source: Ivan Porres, Difference and Union of Models, 10 years later (invited presentation). MODELS 2013

Edit Script Operations

- Edit script operations
  - Differences or deltas are expressed as a sequence of operations, the definition of Δ.
- Element creation and deletion
  - **new(*e, t*)** : Create a new element of type *t* with UUID *e*. By default, a new element has all its features set to their default values.
  - **del(*e, t*)** : Delete an element of type *t* with UUID *e*. An element may only be deleted if all its features are set to their default values.

Operation O	Dual operation $\tilde{O}$
$\mathrm{new}(e,t)$	$\mathrm{del}(e,t)$
$\mathrm{del}(e,t)$	$\mathrm{new}(e,t)$
$\mathrm{set}(e,f,v_o,v_n)$	$\mathrm{set}(e,f,v_n,v_o)$
$\mathrm{insert}(e,f,e_t)$	$\mathrm{remove}(e,f,e_t)$
$\mathrm{remove}(e,f,e_t)$	$\mathrm{insert}(e,f,e_t)$
$\mathrm{insertAt}(e,f,e_t,i)$	$\mathrm{removeAt}(e,f,e_t,i)$
$\mathrm{removeAt}(e,f,e_t,i)$	$\mathrm{insertAt}(e,f,e_t,i)$

Table 1: The Map Between Operations and Dual Operations.

Source: Marcus Alanen and Ivan Porres. Difference and union of models. UML 2003.

Edit Script Operations

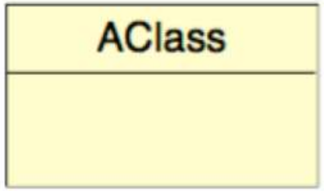
- Modification of a feature of type *f* of an element with UUID *e*. Where necessary, e_t refers to another element.
  - **set(e, f, v_o, v_n):** Set the value of *e*.*f* from v_o to v_n for an attribute of primitive type.
  - **insert(e, f, e_t):** Add a link from *e*.*f* to e_t, for an unordered feature.
  - ***insertAt(e, f, e_t, i):*** Add a link from *e*.*f* to e_t, at index *i*, for an ordered feature.
  - **removeAt(e, f, e_t, i):** Remove a link from *e*.*f* to e_t, which is at index *i*, for an ordered feature.

Operation O	Dual operation $\tilde{O}$
$\text{new}(e,t)$	$\text{del}(e,t)$
$\text{del}(e,t)$	$\text{new}(e,t)$
$\text{set}(e,f,v_o,v_n)$	$\text{set}(e,f,v_n,v_o)$
$\text{insert}(e,f,e_t)$	$\text{remove}(e,f,e_t)$
$\text{remove}(e,f,e_t)$	$\text{insert}(e,f,e_t)$
$\text{insertAt}(e,f,e_t,i)$	$\text{removeAt}(e,f,e_t,i)$
$\text{removeAt}(e,f,e_t,i)$	$\text{insertAt}(e,f,e_t,i)$

Table 1: The Map Between Operations and Dual Operations.

Source: Marcus Alanen and Ivan Porres. Difference and union of models. UML 2003.

Edit Script Example



$$
\begin{aligned}
\Delta = [[\quad & \mathrm{new}(\mathrm{Class}, u_2), \\
& \mathrm{new}(\mathrm{Generalization}, u_3)], \\
[\quad & \mathrm{insert}(u_3, \mathrm{namespace}, u_0), \\
& \mathrm{insert}(u_3, \mathrm{parent}, u_1), \\
& \mathrm{insert}(u_3, \mathrm{child}, u_2), \\
& \mathrm{insert}(u_1, \mathrm{specialization}, u_3), \\
& \mathrm{insert}(u_0, \mathrm{ownedElement}, u_2), \\
& \mathrm{insert}(u_0, \mathrm{ownedElement}, u_3), \\
& \mathrm{insert}(u_2, \mathrm{namespace}, u_0), \\
& \mathrm{insert}(u_2, \mathrm{generalization}, u_3), \\
& \mathrm{set}(u_2, \mathrm{name}, \text{“”}, \text{“Sub”})], \\
[\quad &] \\
] &
\end{aligned}
$$

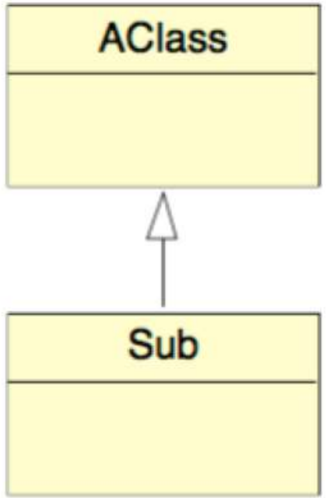


Figure 4: Difference Between Two Simple Models.

Source: Marcus Alanen and Ivan Porres. Difference and union of models. UML 2003.

Implications, Benefits and Limitations

- Differences can be
  - Programmed manually
  - Leveraged for algorithms and modeling tools
  - Generated from DSLs
  - Recorded, played back
  - Applied on systems and rolled back
  - Analyzed formally for predicting results
  - Used for understanding the evolution of models

- Main limitations of A&P approach.
  - Requires unique, stable, universal model element identifiers across model revisions.
  - Metamodel is assumed to be static.

- In addition: Encode history, NOT scripts! (operations go stale)

Source: Ivan Porres, Difference and Union of Models, 10 years later (invited presentation). MODELS 2013

Origin Tracking + Text Differencing = Textual Model Differencing

Theory and Practice of Model Transformations, 2015

Riemer van Rozen[1,2,3] and Tijs van der Storm[2,4]

@rvrozen **@tvdstorm**

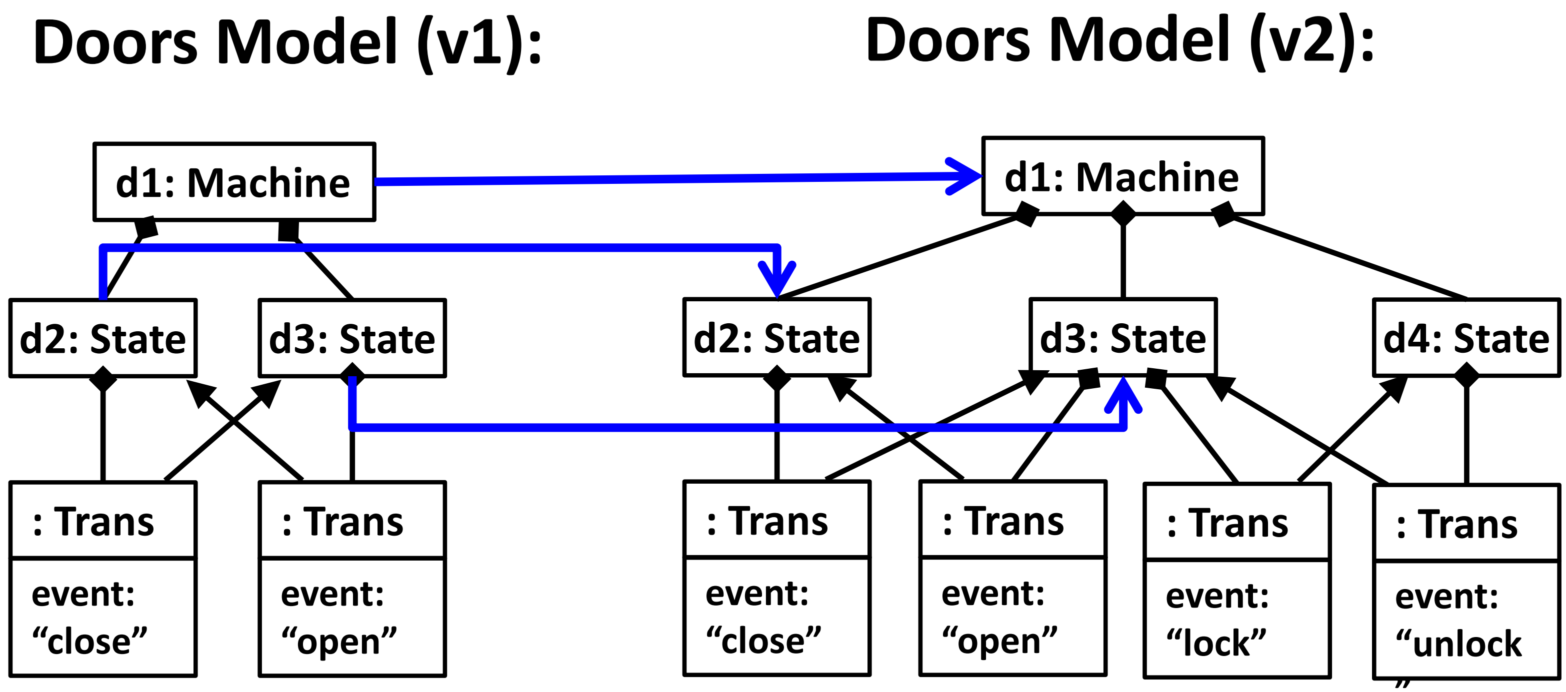
[1] Amsterdam University of Applied Sciences (HvA)

[2] Centrum Wiskunde & Informatica (CWI)

[3] University of Amsterdam – Master SE (UvA)

[4] Rijksuniversiteit Groningen (RUG)

Problem: Differencing with identity



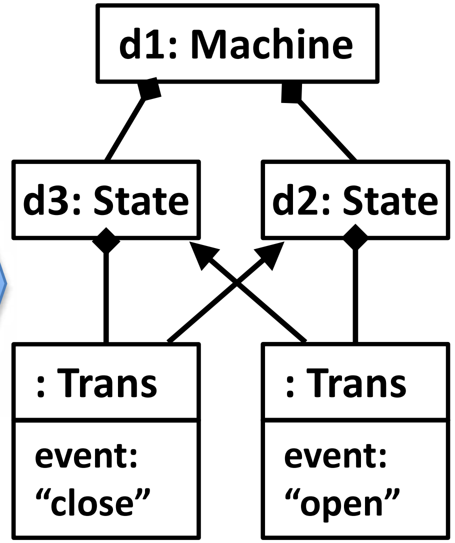
- Problem
  - We cannot simply apply model differencing to models encoded as text.

Problem: Textual Model Differencing

- What are the entities?
  - First parse to obtain a tree
  - Referential structure is determined by scoping rules
    - Definitions: machine, state
    - Uses: transition

Doors.sml (v1):

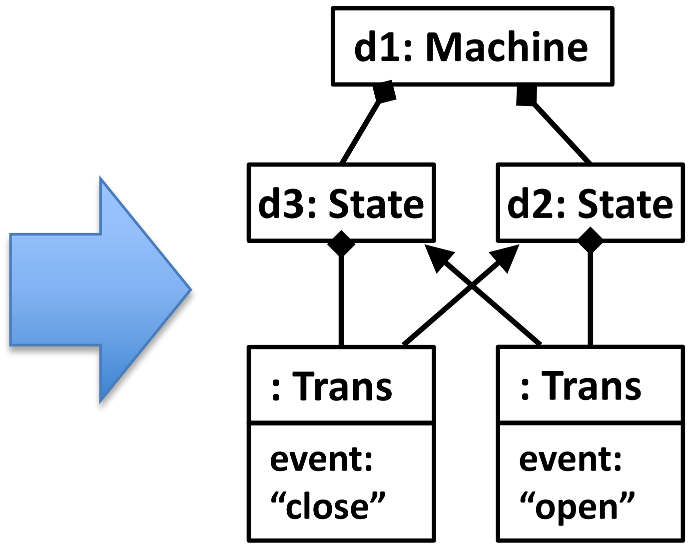
```
machine doors  [d1]
 state closed  [d2]
  open => opened  [u1]

 state opened  [d3]
  close => closed  [u2]
end
```



- Problem
  - Textual model elements have no stable identity across source versions.

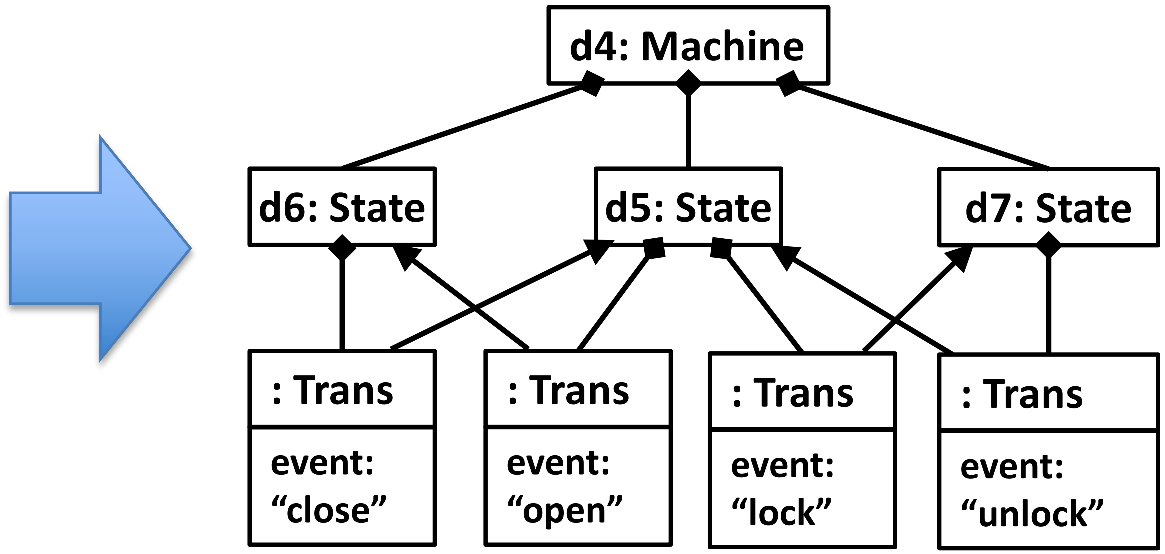
Doors.sml (v1):

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

 state opened      d3
   close => closed       u2
end
```



Doors.sml (v2):

```
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
```



Textual model elements have no stable identity across source versions

Objectives: Computing Deltas

- Question
  - How to apply model differencing to models encoded as text?
- What are the differences?
  - Imperative edit scripts encode deltas
  - Multiple deltas can express the difference between two models → ambiguity
  - Deltas can capture user intent

```
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
```

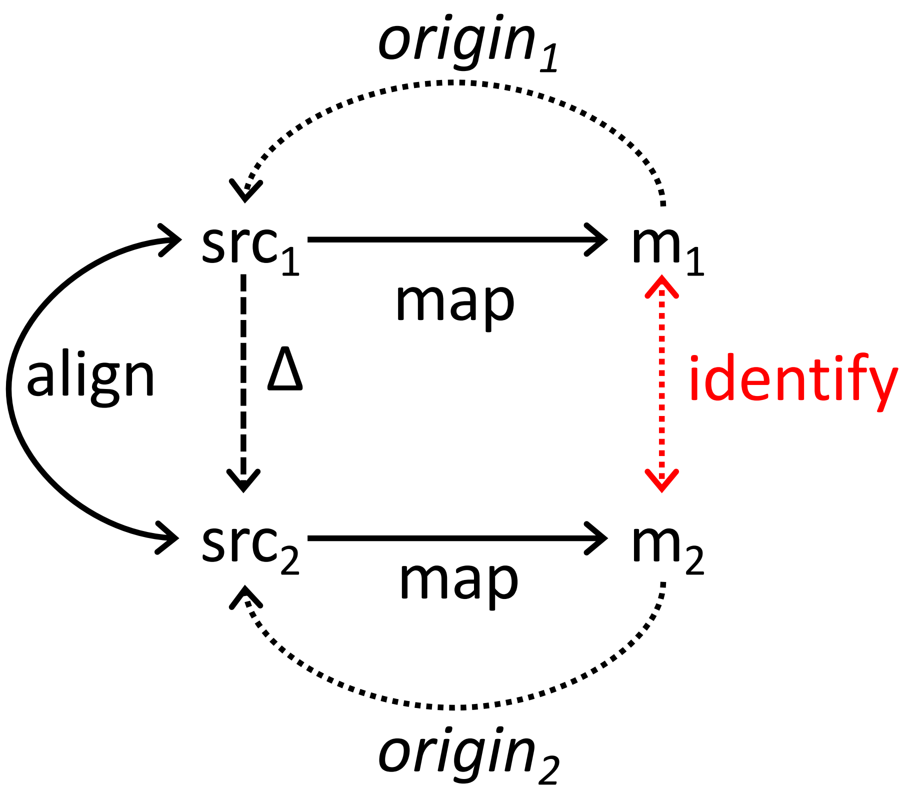
```
//create a State def with label d7
create State d7
//initialize the new State "locked"
d7 = State("locked",[Trans("unlock",d2)])
//store 2nd Trans in state "closed"
d2.out[1] = Trans("lock", d7)
//store new State
d1.states[2] = d7
```

Contributions

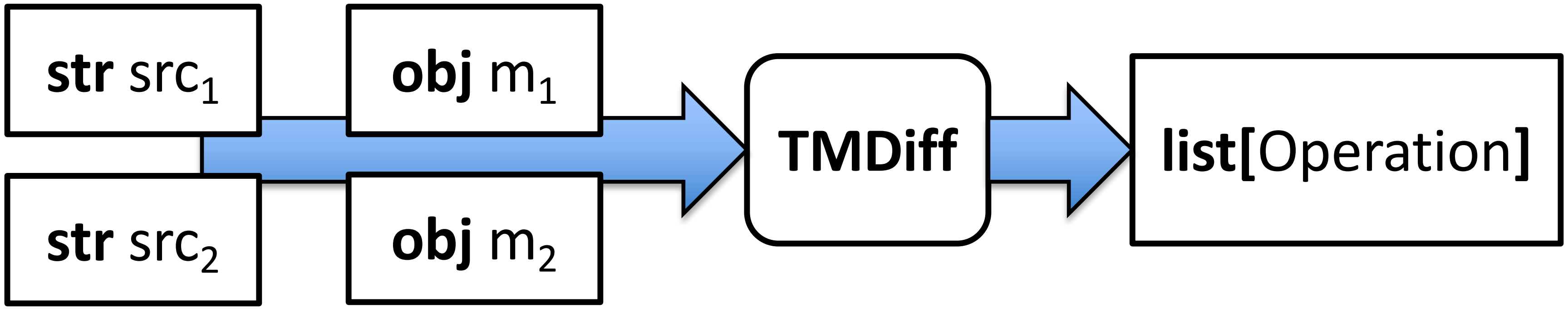
- Question
  - How can **textual differencing** be used to match model elements based on **origin tracking**?
- Contributions
  - TMDiff
  - Apply TMDiff to DSL programs

Objectives: Computing Deltas

- Origin
  - src_n has an origin relation with m_n
- Align
  - Use the text diff Δ between src_1 and src_2 to align tokens of entities.
- Objective: Identify
  - Given textual models src_1 and src_2 determine which entities in m_1 are still in m_2



Approach: TMDiff



- TMDiff steps
  - **Matching**: generate a tuple of added, removed and identified entities
  - Added: generate *Create* and *SetTree* operations
  - Identified: **difference nodes** definitions
  - Removed: generate *Delete* operations

Matching Entities: Text diff

```
1  machine doors
2   state closed
3     open => opened
4
5   state opened
6     close => closed
7  end
```

```
1   machine doors
2    state closed
3     open => opened
4     lock => locked
5
6    state opened
7      close => closed
8
9    state locked
10     unlock => closed
11  end
```

```
--- a/doors1.sl
+++ b/doors2.sl
@@ -3,0 +4
+ lock => locked
@@ -6,0 +8,3
+
+ state locked
+ unlock => closed
```

Matching Entities: Project, Identify

```
1  machine doors  [d1]
2   state closed  [d2]
3     open => opened
4
5   state opened  [d3]
6     close => closed
7  end
```

```
1   machine doors  [d4]
2    state closed  [d5]
3      open => opened
4      lock => locked
5
6    state opened  [d6]
7      close => closed
8
9    state locked  [d7]
10     unlock => closed
11  end
```

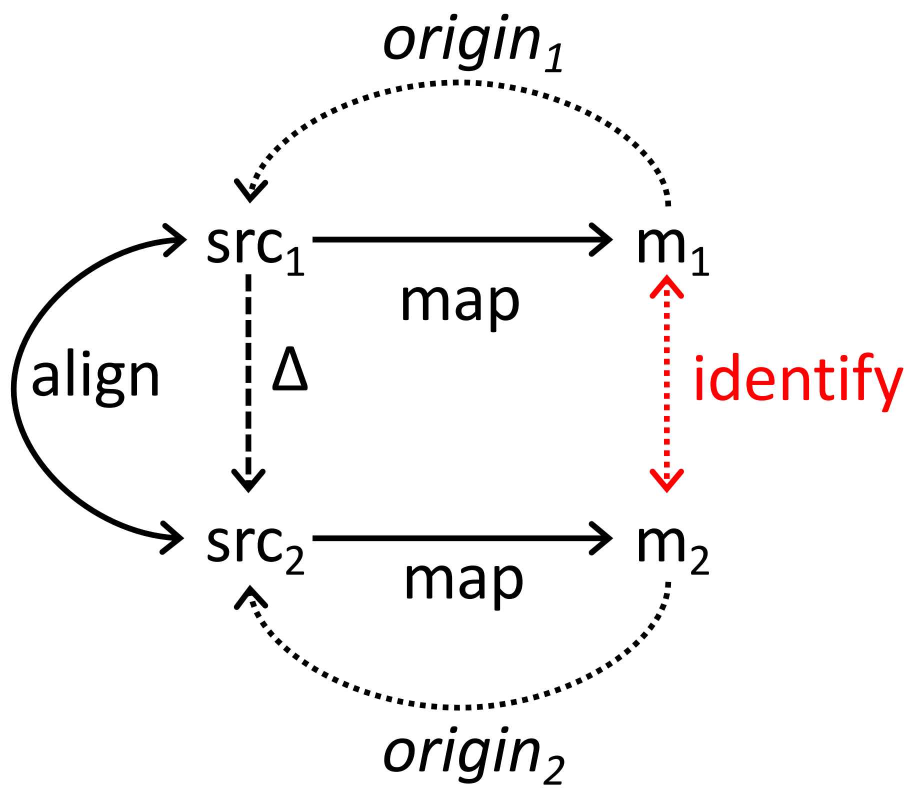
```
P1 =
 [⟨doors,   Machine, 1, d1⟩
  ⟨closed,  State,   2, d2⟩,
  ⟨opened,  State,   5, d3⟩]

P2 =
 [⟨doors,   Machine, 1, d4⟩
  ⟨closed,  State,   2, d5⟩,
  ⟨opened,  State,   6, d6⟩
  ⟨locked,  State,   9, d7⟩]  add
```

- Calculate Matching
  - added, removed, identified entities
  - $M_{1,2} = \langle \{d7\}, \{\}, \{\langle d1, d4\rangle, \langle d2, d5\rangle, \langle d3, d6\rangle\}\rangle$

Differencing

- We now have
  - Textual sources
  - Models
  - Origin relations
  - Matching

- We now can
  - Apply well-known model differencing algorithms.



Implementation & Evaluation

- Rascal
  - Meta-programming language and language work bench http://www.rascal-mpl.org
  - TMDiff https://github.com/cwi-swat/textual-model-diff
- Evaluated on Derric
  - A DSL for digital forensics
  - Describes file formats for analyzing large amounts of unstructured data.
  - File format evolution is available on GitHub. https://github.com/jvdb/derric

```
format gif
extension gif

strings ascii
sign false
unit byte
size 1
type integer
endian big

Sequence
  (Header87a Header89a)
        LogicalScreenDesc
        (
          [GraphicControlExtension? TableBasedImage
CompressedDataBlock*]
          [GraphicControlExtension?
PlainTextExtension DataBlock*]
          [ApplicationExtension DataBlock*]
          [CommentExtension DataBlock*]
        )
```

Towards Live Domain-Specific Languages

From Text Differencing to **Adapting Models at Runtime**

Journal of Software & Systems Modeling, 2017

Riemer van Rozen[1,2,3] and Tijs van der Storm[2,4]

@rvrozen **@tvdstorm**

[1] Amsterdam University of Applied Sciences (HvA)

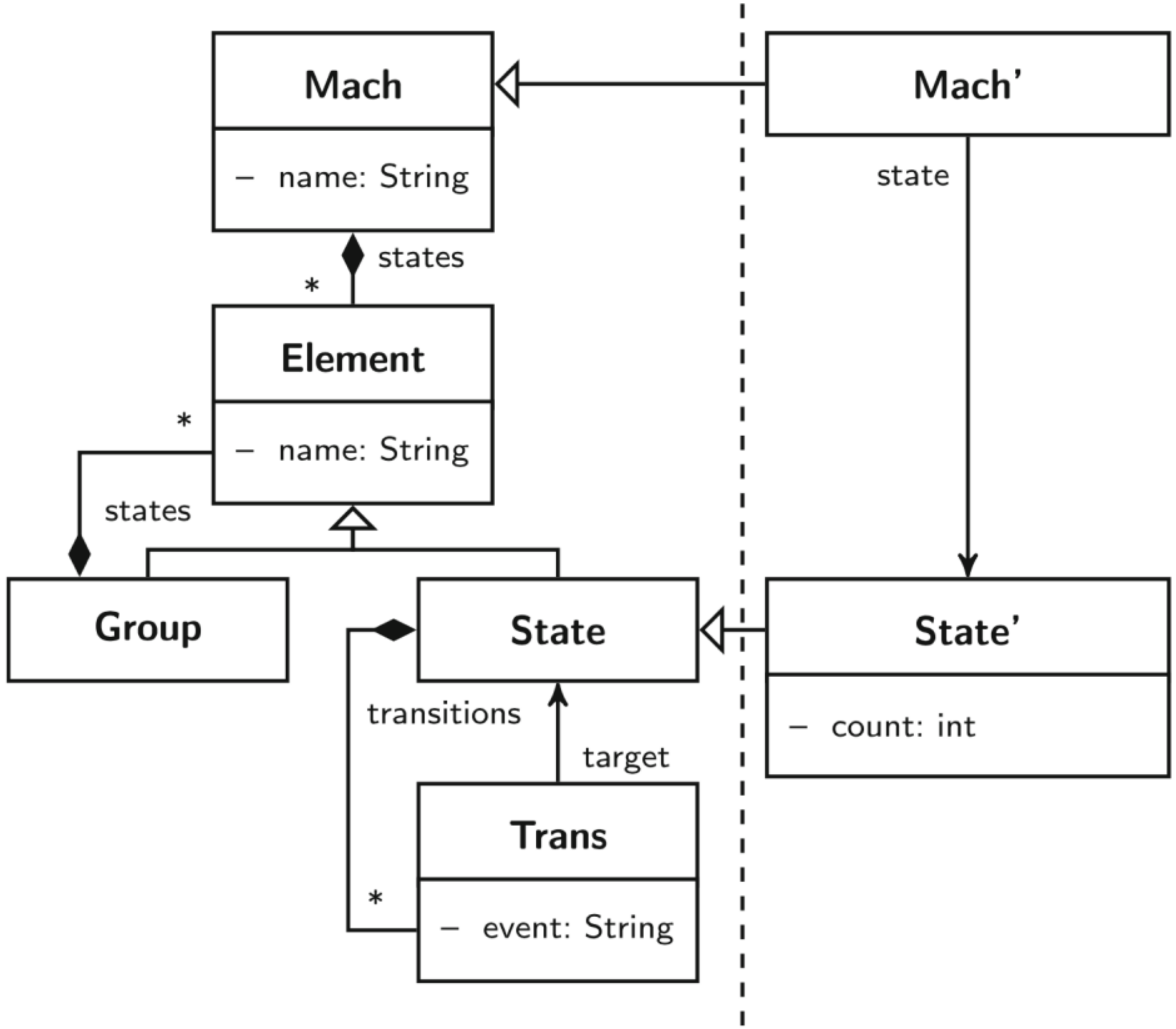
[2] Centrum Wiskunde & Informatica (CWI)

[3] University of Amsterdam – Master SE (UvA)

[4] University of Groningen (RUG)

Case Study: Live SML

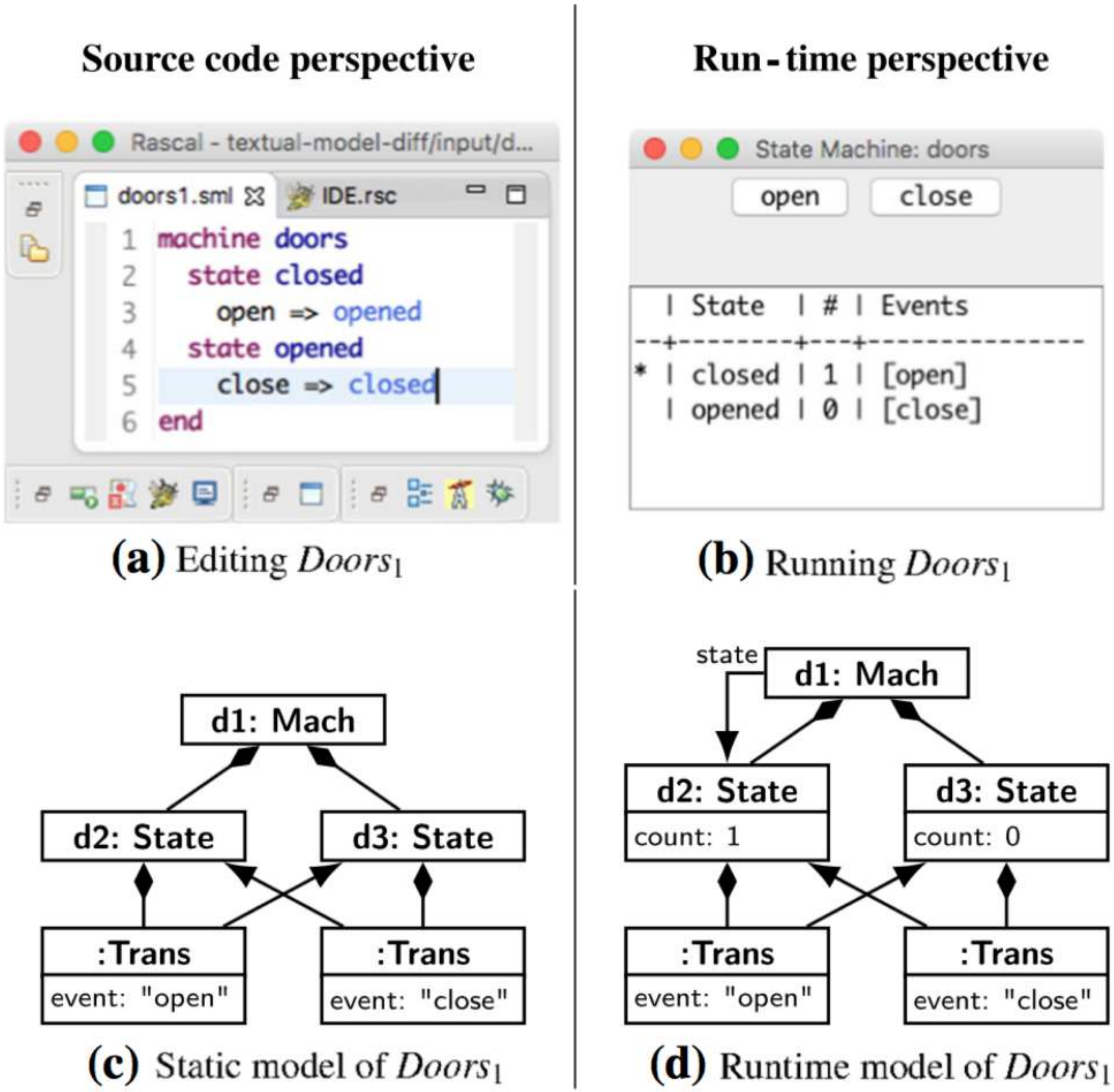
- LiveSML Metamodels
  a) Static metamodel
  b) Dynamic metamodel extension:
     - Machine current state
     - State count
- **Note:** The run-time meta-model of LiveSML “extends” its static meta-model, which is not true in general



(a) Meta model **(b)** Runtime extension

Live SML: Components & Models

- Live SML components
  - a) programming environment
  - b) program execution as an interactive GUI
- Live SML Models
  - c) static SML model representing the textual source code
  - d) dynamic SML model that is executing at run time



(a) Editing *Doors*$_1$

(b) Running *Doors*$_1$

(c) Static model of *Doors*$_1$

(d) Runtime model of *Doors*$_1$

Live SML: State Migration

- *Creation of a new machine*
  - Initially there is no machine because we start with an empty object space.
  - We store a reference to the machine when it is first created (lines 9 and 10).
- *Creation of a new state*
  - The *count* attribute is initialized to 0 (lines 12–15).

```
1  class MigrateSML extends ApplyDelta {
2    private Mach machine; //run-time model to migrate
3
4    @Override
5    public void visit(Create create) {
6      super.visit(create);
7
8      Object x = create.getCreated(this);
9      if (x instanceof Mach) { //new machine
10       this.machine = (Mach) x;
11     }
12     else if (x instanceof State) { //new state
13       Edit e = new SetPrim(reverseLookup(x),
14                    new Path(new Field("count")), 0);
15       e.accept(this);
16     }
17   }
18
```

Live SML – State Migration

- *Insertion of an element in an uninitialized machine.*
  - When a state or group is inserted into a machine that has no current state (lines 24–29), it is initialized to the *initial state* (lines 43–54).
  - The initial state is the first state in the textual model.

- *Deletion of the current state*
  - When a machine's current state is deleted (lines 36–37), it is reinitialized to the initial state (lines 43–54).

```
19 @Override
20 public void visit(Insert insert) {
21   super.visit(insert);
22
23   Object owner = insert.getOwner(this);
24   if (machine != null && machine.state == null
25         && owner == machine) {
26     // Added a group or state to a machine
27     // without a current state.
28     goToInitialState();
29   }
30 }
31
32 @Override
33 public void visit(Delete delete) {
34   super.visit(delete);
35
36   Object x = delete.getDeleted(this);
37   if (machine != null && x == machine.state) {
38     // Deleted the current state.
39     goToInitialState();
40   }
41 }
```

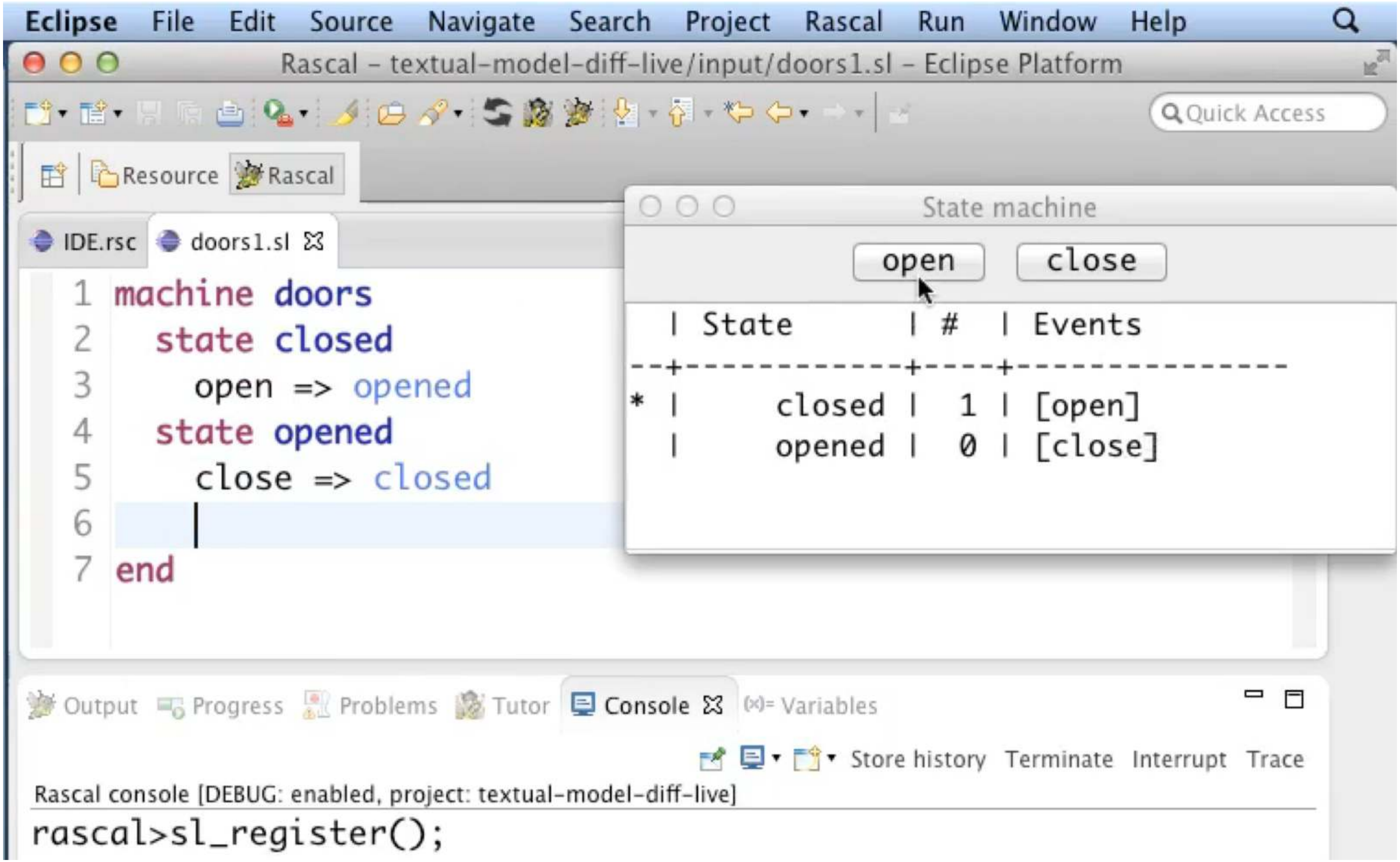
Live SML – State Migration

- *Insertion of an element in an uninitialized machine.*
  - When a state or group is inserted into a machine that has no current state (lines 24–29), it is initialized to the *initial state* (lines 43–54).
  - The initial state is the first state in the textual model.

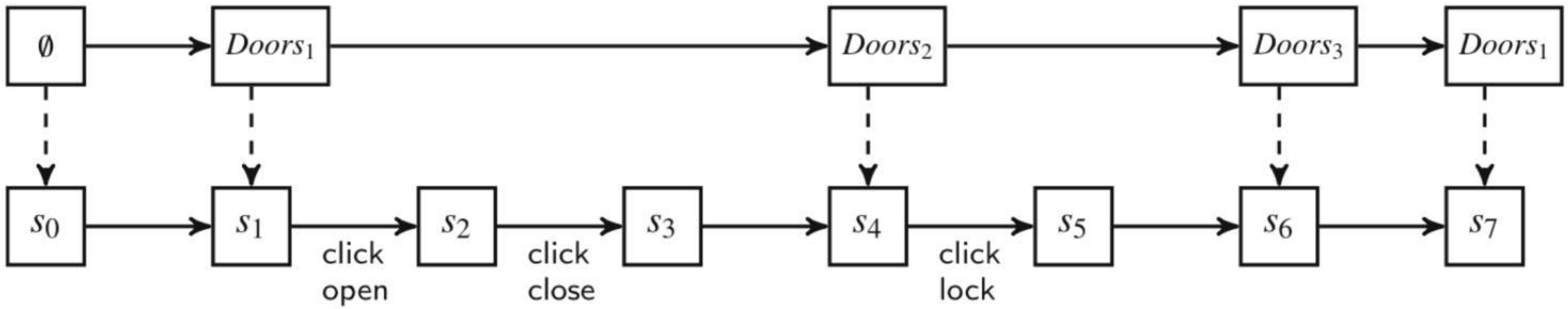
```
43      private void goToInitialState(){
44        State s = machine.findInitial();
45        Edit e1 = new Set(reverseLookup(machine),
46            new Path(new Field("state")), s);
47        e1.accept(this); //Set the current state.
48
49        if (s != null){
50          Edit e2 = new Set(reverseLookup(s),
51              new Path(new Field("count")), s.count+1);
52          e2.accept(this); //Increment current state count.
53        }
54      }
55    }
```

- *Deletion of the current state*
  - When a machine's current state is deleted (lines 36–37), it is reinitialized to the initial state (lines 43–54).

Live State Machine Language in Rascal



Live SML: Modeling Scenario



- Interleaved coevolution of models *Doors*$_n$ and application run-time states *S*$_n$ over time

- Next: TMDiff deltas + migration deltas

Live SML: Modeling Scenario

Model	State	Event	Edit operation		Origin
∅	$s0$	Save *Doors*1	δ1	**create** State d2	TMDiff ∅ *Doors*1
			δ2	d2.count = 0	side effect
			δ3	**create** State d3	
			δ4	d3.count = 0	side effect
			δ5	**create** Mach d1	
			δ6	d2 = State(name("closed"), [Trans("open",d3)])	
			δ7	d3 = State(name("opened"), [Trans("close",d2)])	
			δ8	d1 = Mach(name("doors"), [d2,d3])	
			δ9	d1.state = d2	side effect
			δ10	d2.count = 1	side effect

At the end of this sequence we are in Model *Doors*1 and State *s*1.

Model	State	Event	Edit operation		Origin
*Doors*1	s1	Click *open*	δ11	d1.state = d3	user action
			δ12	d3.count = 1	
*Doors*1	s2	Click *close*	δ13	d1.state = d2	user action
			δ14	d2.count = 2	
*Doors*1	s3	Save *Doors*2	δ15	**create** State d7	TMDiff *Doors*1 *Doors*2
			δ16	d7.count = 0	side effect
			δ17	d7 = State(name("locked"), [Trans("unlock",d2)])	
			δ18	**insert** d2.transitions[1] = Trans("lock",d7)	
			δ19	**insert** d1.states[2] = d7	
			δ20	**rekey** d1 → d4	
			δ21	**rekey** d2 → d5	
			δ22	**rekey** d3 → d6	
*Doors*2	s4	Click *lock*	δ23	d4.state = d7	user action
			δ24	d7.count = 1	

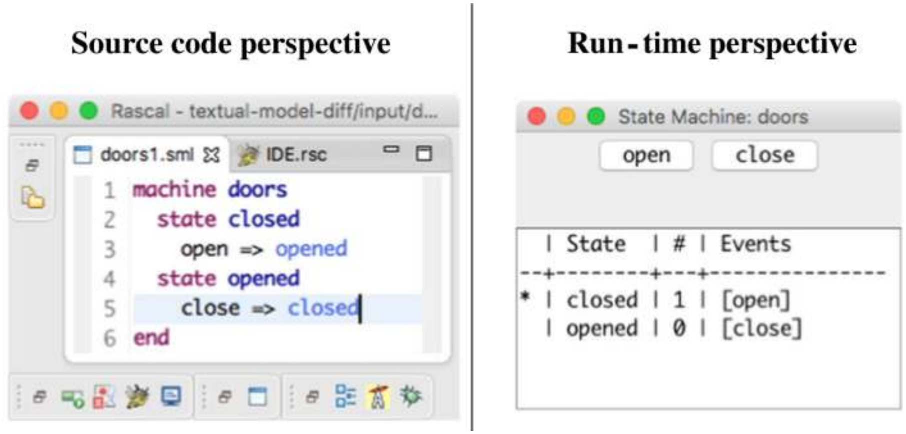
Model	State	Event	Edit operation		Origin
Doors2	s5	Save *Doors3*	δ25	**create** Group d11	TMDiff *Doors2* *Doors3*
			δ26	d11 = Group("locking",[d6])	
			δ27	**remove** d4.states[2]	
			δ28	**insert** d4.states[2] = d0	
			δ29	**rekey** d4 → d8	
			δ30	**rekey** d5 → d9	
			δ31	**rekey** d6 → d10	
			δ32	**rekey** d7 → d12	
Doors3	s6	Save *Doors1*	δ33	**remove** d8.states[2]	TMDiff *Doors3* *Doors1*
			δ34	**remove** d9.transitions[1]	
			δ35	**delete** d11	
			δ36	**delete** d12	
			δ37	d13.state = d9	Side effect
			δ38	d9.count = 3	Side effect
			δ39	**rekey** d8 → d13	
			δ40	**rekey** d9 → d14	
			δ41	**rekey** d10 → d15	

Discussion, Benefits and Limitations

Feature / benefit	Trade-off / limitation	Mitigating argument
Edit operations: record history as edit scripts for do, undo, replay	Large memory foot print, a potential memory leak	Recording differences can be turned off or limited
TMDiff is language-parametric (needs name resolution) and calculates model-based deltas “for free”	The results of the differencing algorithm bleed into the language semantics (which entities live and die)	Facilitates rapid Live prototyping of DSLs for live and textual modeling. The default is usually OK due to small incremental changes
RMPatch helps construct DSL interpreters for live programming	High implementation effort. The granularity of edit scripts operations is too fine (does not scale).	Some languages require exact state migrations and precise steering

Conclusions and Future Work

- Questions
  1. How can **textual differencing** be used to match model elements based on **origin tracking**?
  2. How can **"Live DSL"** construction be supported with generic reusable frameworks?
- Contributions
  – TMDiff and RMPatch
  – Apply TMDiff to DSL programs
  – LiveSML illustrative example
- Current work
  – Modeling extensible state migrations that scale to larger DSLs
  – Live Machinations



References

- Riemer van Rozen and Tijs van der Storm. "Origin Tracking + Text Differencing = Textual Model Differencing." *International Conference on Theory and Practice of Model Transformations*. Springer, 2015.
- Riemer van Rozen and Tijs van der Storm. "Toward Live Domain-Specific Languages: From Text Differencing to Adapting Models at Runtime" *Software & Systems Modeling*(2017): 1-18.

Modeling with Side-Effects

current work

in the context of the Live Game Design RAAK-MKB project

Riemer van Rozen[1,2,3]

@rvrozen

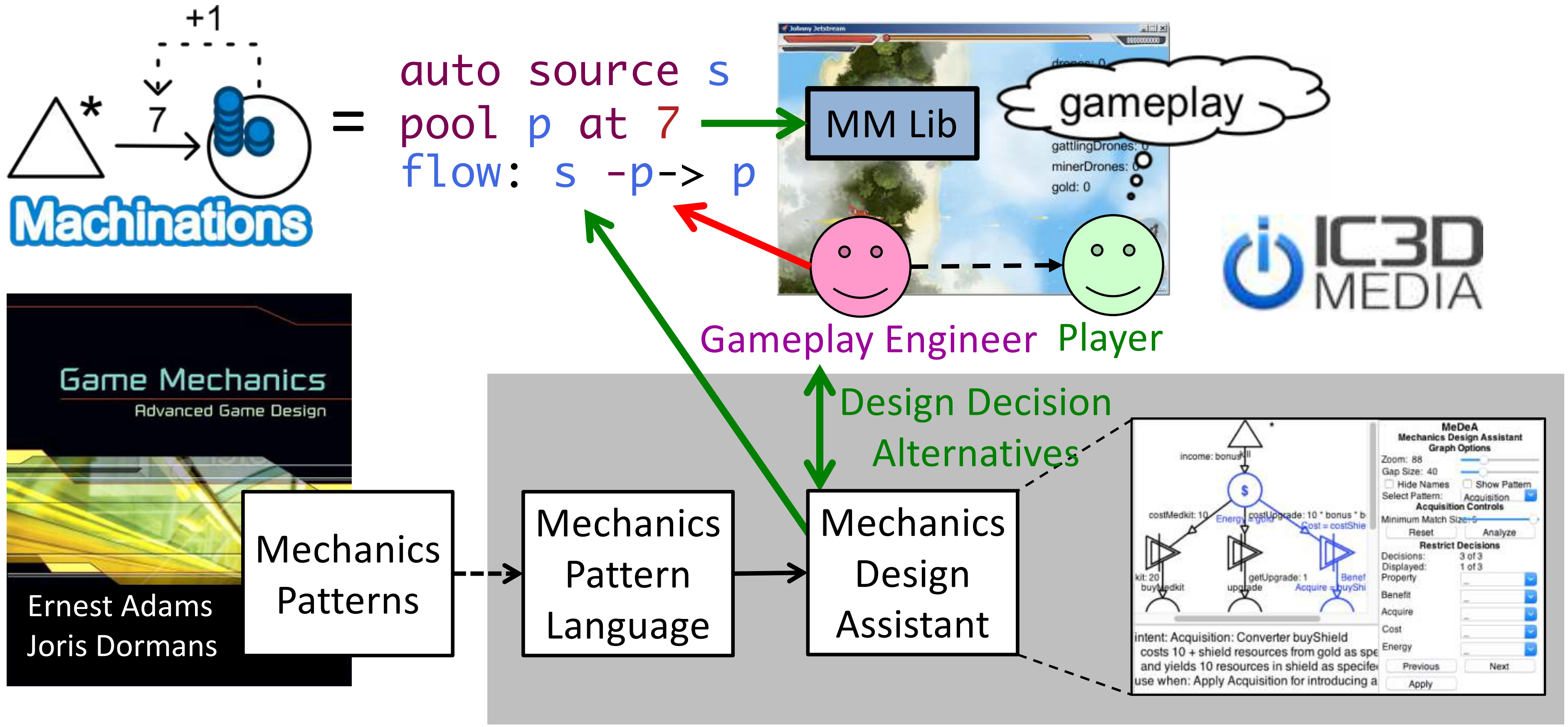
[1] Amsterdam University of Applied Sciences (HvA)

[2] Centrum Wiskunde & Informatica (CWI)

[3] University of Amsterdam – Master SE (UvA)

Machinations Evolution & Approach

2009 Conceptual Game Design Aid → **2013** Formal Analysis + text, modules → **2014** Live Adaptations v1.0 C++ → **2015** A Pattern Based Game Design Assistant → **2018** Live Adaptations v2.0 C# Unity

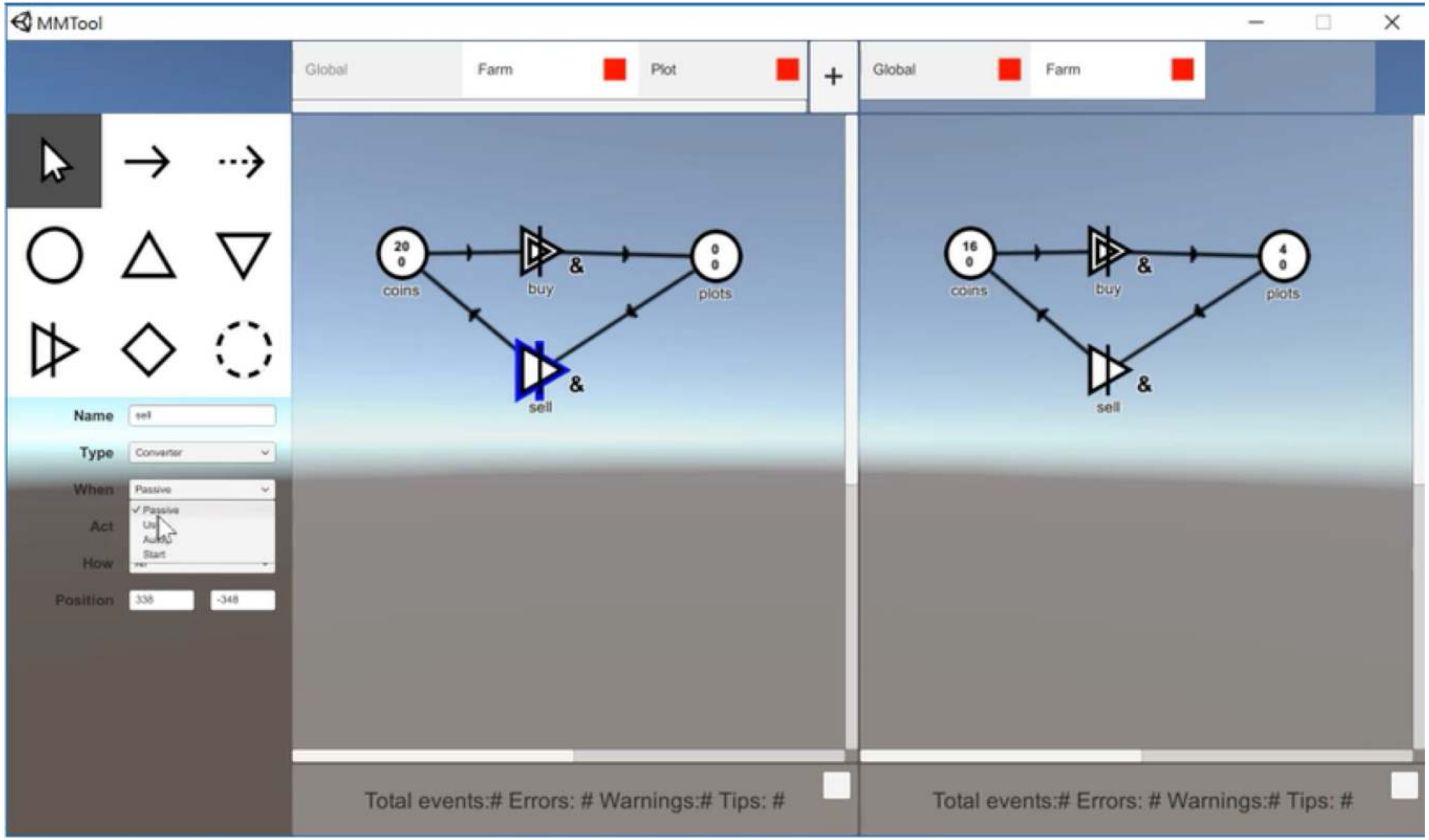


Machinations Evolution & Approach

2009 Conceptual Game Design Aid → **2013** Formal Analysis + text, modules → **2014** Live Adaptations v1.0 C++ → **2015** A Pattern Based Game Design Assistant → **2018** Live Adaptations v2.0 C# Unity

- Joris Dormans. Machinations: Elemental Feedback Patterns for Game Design. In GAMEON-NA, 2009.
- Ernest Adams and Joris Dormans. Game Mechanics: Advanced Game Design. New Riders Publishing, 2012.
- Paul Klint and Riemer van Rozen. Micro-Machinations: A DSL for Game Economies. In Software Language Engineering, 2013
- Riemer van Rozen and Joris Dormans. Adapting Game Mechanics with Micro-Machinations. In Foundations of Digital Games, 2014.
- Riemer van Rozen. A Pattern-Based Game Mechanics Design Assistant. In Foundations of Digital Games, 2015.
- Riemer van Rozen and Tijs van der Storm. Toward Live Domain-Specific Languages. *Software & Systems Modeling*, 2017.

Live Machinations: Model + State



Live Machinations: Model + State

rozen — Xamarin Studio External Console — mono32 --debug --debugger-agent=t

```
Micro-Machinations Design Space Navigator v0.01

MMMMMAMMAMMMMMMAMMMMMAMMMMMAMMAMMAAAMMMMMMMAMMAMMAMMMMMAMMAAAMMMMMMMAMMAMMAMMAMM
[Model]
pool fork at 1
Philosopher
{
  ref spoon
  user pool righthand at 0
  s0: spoon-->spoon
  ref fork
  push source eatSource
  all drain eatDrain
  eatDrain.*.>eatDrain
  user all converter eat
  s1: righthand-->righthand
  s2: eat-->eat
  user pool lefthand at 0
  f1: lefthand-->lefthand
  f2: eat-->eat
  f0: fork-->fork
}
pool spoon at 1
pool Jan of Philosopher at 1
fork.=.>fork
spoon.=.>spoon
pool Hans of Philosopher at 1
spoon.=.>spoon
fork.=.>fork

[State]
pool fork = 1
pool spoon = 1
pool Jan = 1
[0]: Philosopher
  pool righthand = 0
  source eatSource
  drain eatDrain
  converter eat
  pool lefthand = 0
pool Hans = 1
[0]: Philosopher
  pool righthand = 0
  source eatSource
  drain eatDrain
  converter eat
  pool lefthand = 0
```