Towards Measuring the Abstractness of Statemachines based on Mutation-Testing

Thomas Baar

thomas.baar@htw-berlin.de

Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

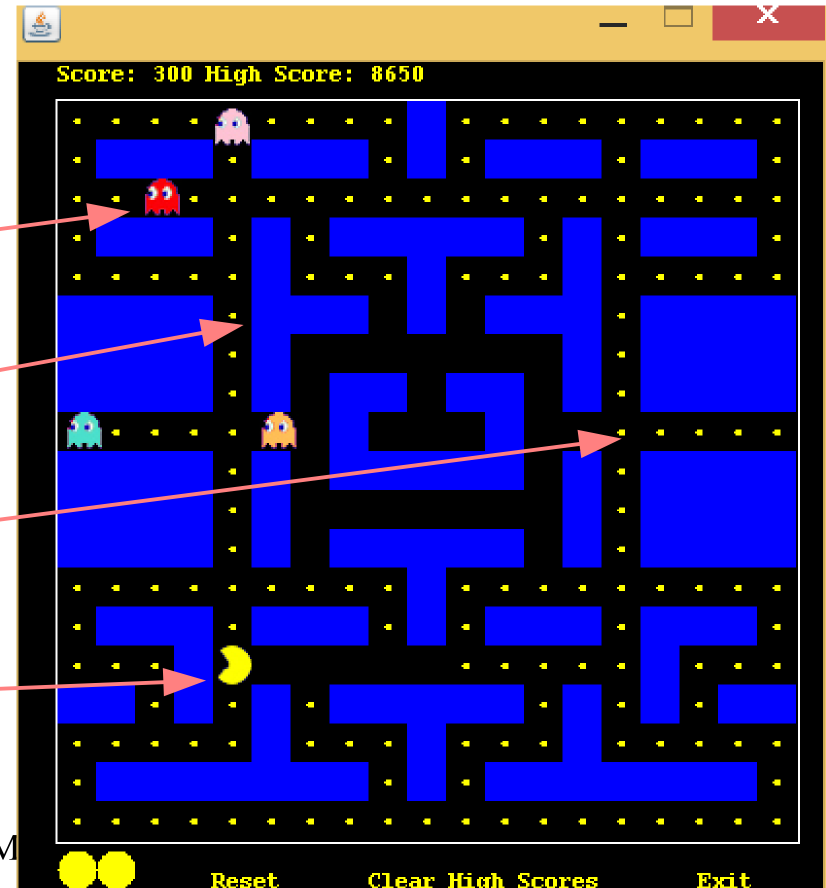
Workshop PSSV (Program Semantics, Specification and Verification: Theory and Applications), Moscow, June 26th, 2017

A Typical Student Assignment

Develop a state model (e.g. expressed using UML State Machines) for a given application

Example:

(PacMan)



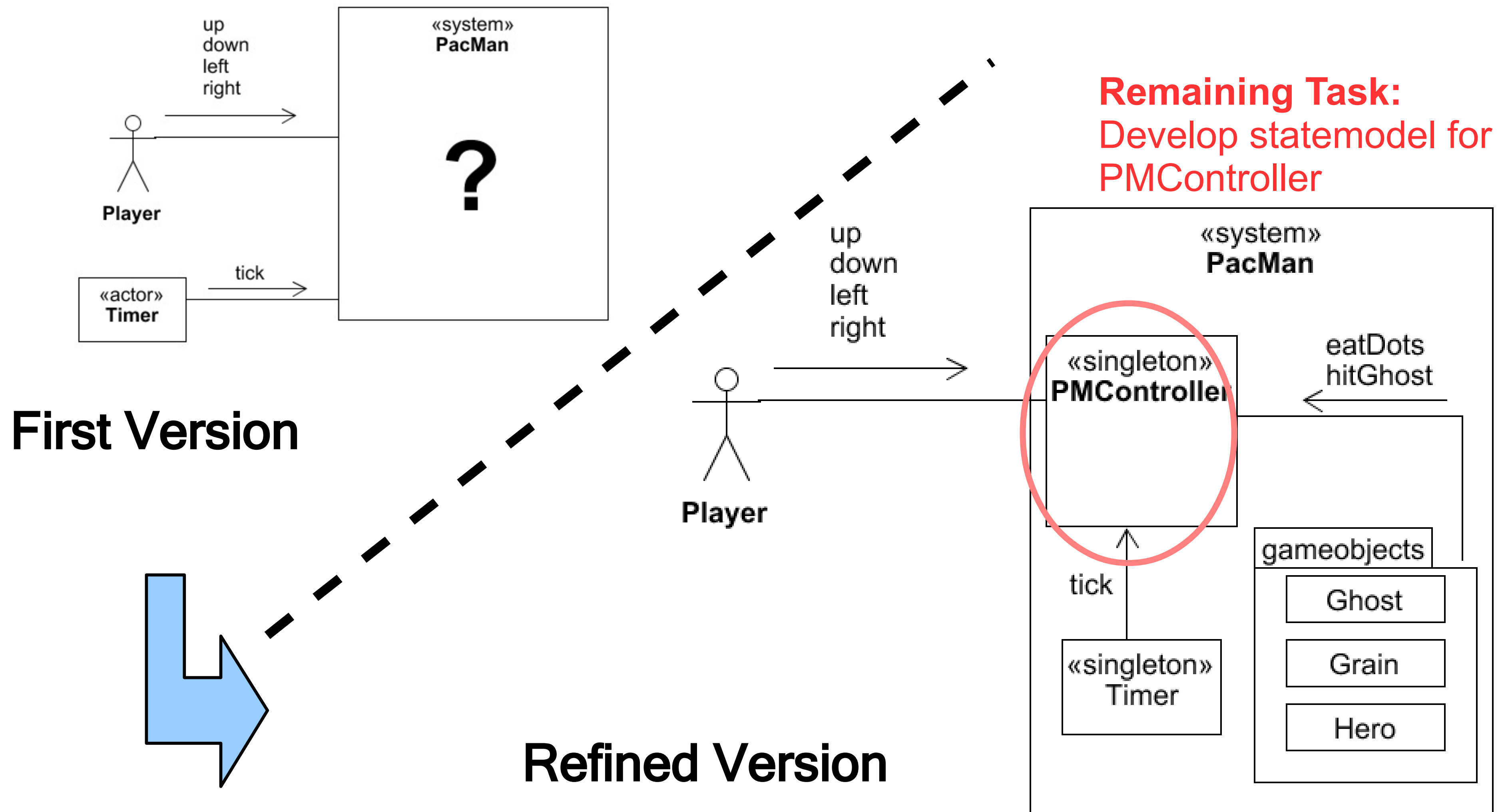
Ghosts
(collisions to be avoided)

Walls

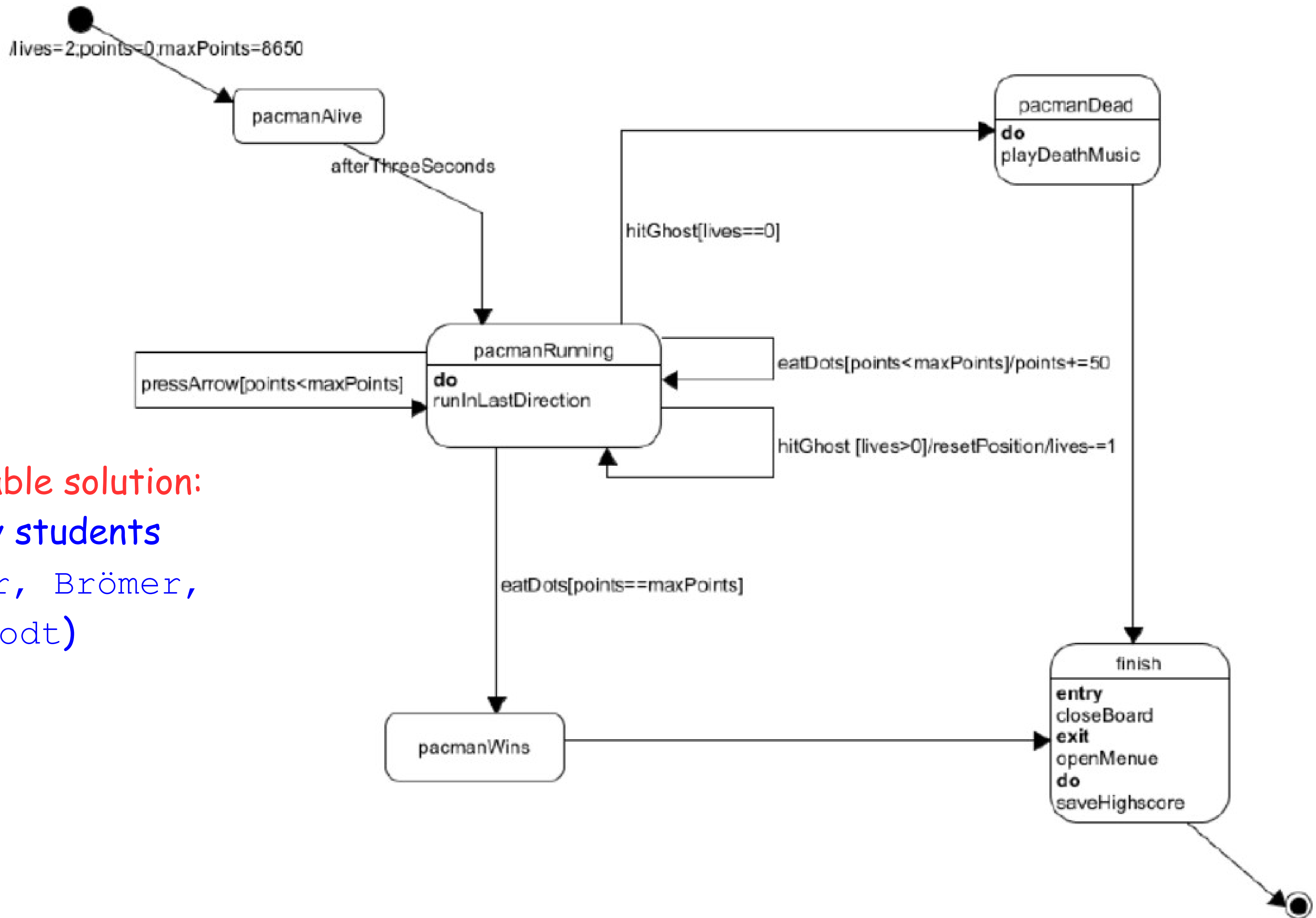
Grains
(to be eaten by the Hero)

Hero (Puck)
(controlled by user via keys)

Environment Model
(Define the Incoming Events)



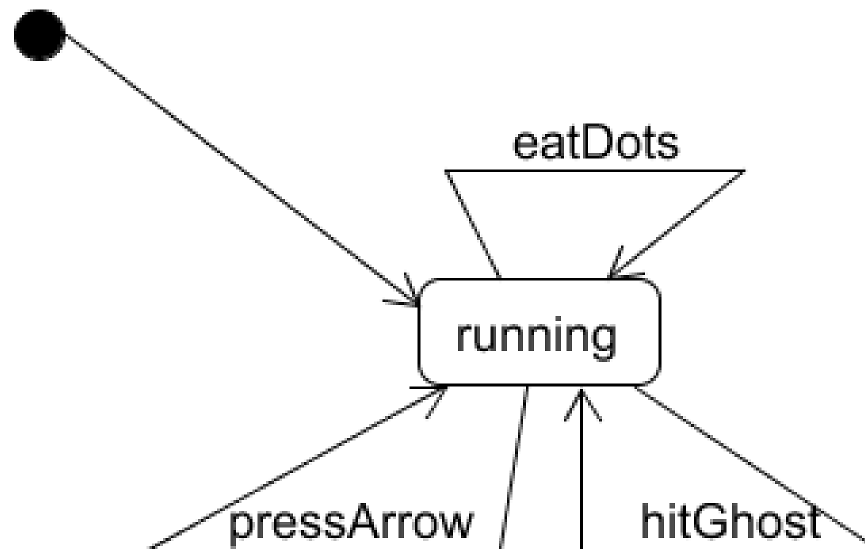
Statemodel for PacMan-Controller



Reasonable solution:
given by students
(Keller, Brömer, Vaterrodt)

Statemodel for PacMan-Controller

(useless) solution - not submitted yet, but imaginable



(My) Problems when Judging Students' Submissions

Q1: Is the statemachine correct?

- Does the implementation really behaves as described?

Q2: Is the statemachine too trivial/too abstract?

- Have all important states of the implementation been captured by the statemachine?

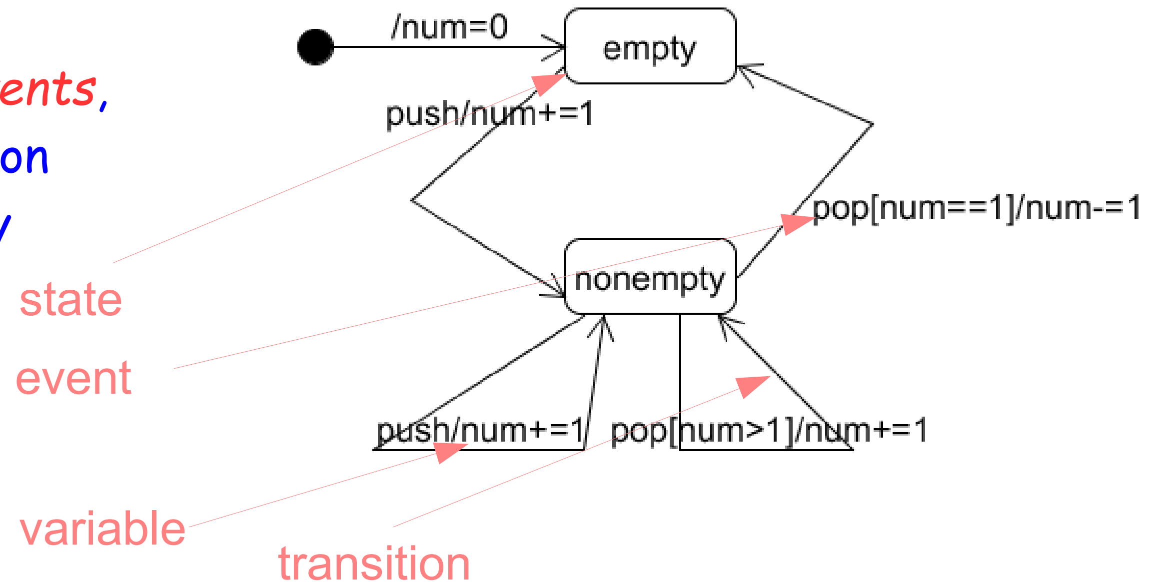
Though these are **central questions** when assessing the quality of modeling artefacts wrt. an implemented system,
I could not find any tool helping to answer them!

Goal of the Paper

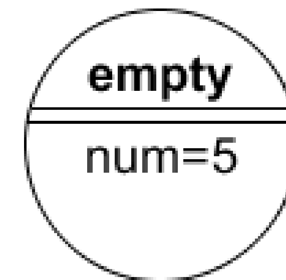
Define a machinery for
measuring correctness and abstractness
of statemachines wrt. a given implementation

Statemachines - Syntax/Semantics

A **statemachine** is a tuple *states*, *events*, *variables*, and *transitions*. A transition connects two states and is optionally annotated with a *guard* and list of *variable updates*.



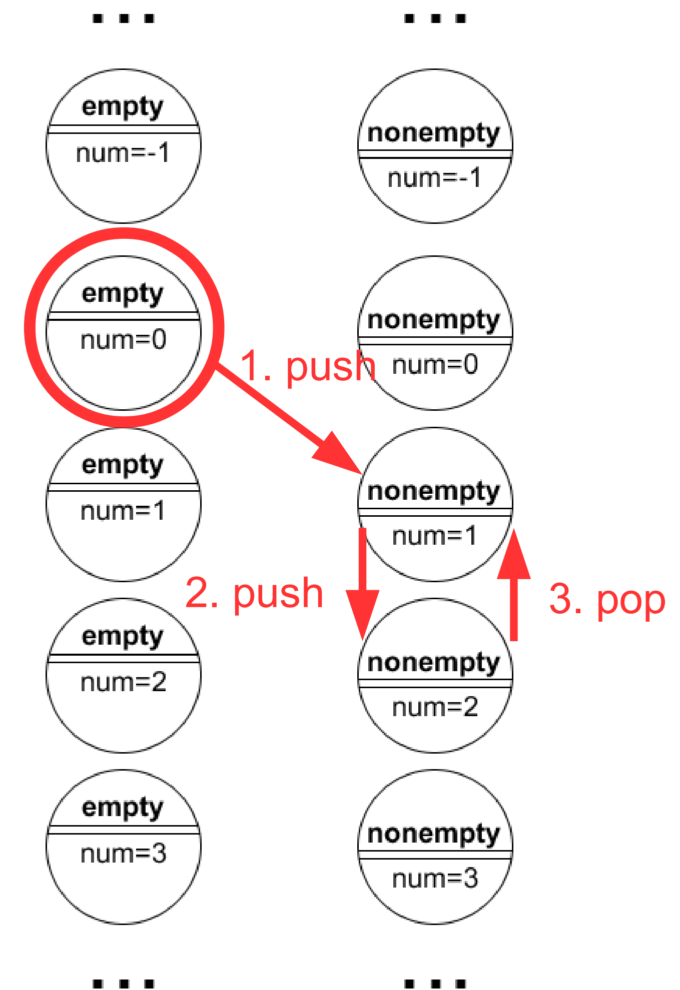
An **execution state** of the statemachine a state combined with a binding of all variables to concrete values.



Statemachines - Syntax/Semantics

The **statespace of a statemachine** is the set of all possible execution states

A **trace** is defined for a given sequence of events as a sequence of execution states, in which each state is connected with its successor by a fired transition. Each trace starts with an execution state satisfying the start condition.



InputEvents: [push, push, pop]

Implementation

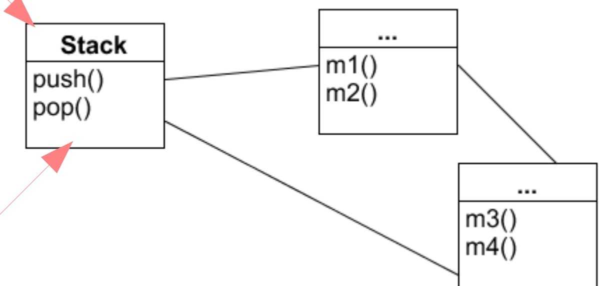
The **implementation** is written in an OO implementation language. We assume a *Facade*-class offering methods with same names as the events of state machine.

We assume the *Facade*-class to control the execution flow: Whenever a method on the Facade-class is invoked, the system executes the method and waits for the next method call.

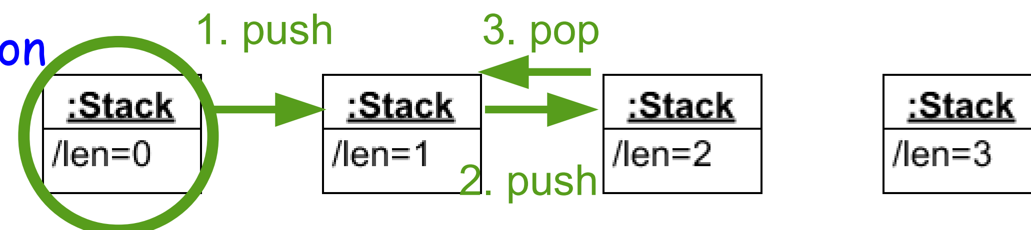
A **trace** is defined for the sequence of method calls on the *Facade*-class. The trace consists of those implementation states when the system is waiting for the next method call. Often, all relevant information about the implementation state can be captured by additional derived attributes on the Facade-class.

```
 7 public class Stack1 {
 8
 9     private List<Item> items = new ArrayList<Item>();
10
11     public void push(Item i){
12         items.add(i);
13     }
14
15     public Item pop(){
16         if (items.isEmpty())
17             return null;
18         int lastIndex = items.size()-1;
19
20         return items.remove(lastIndex);
21     }
22
23     // allow the adapter to get necessary information
24     public int getLength(){
25         return items.size();
26     }
27 }
28
```

Facade-class



Methods have same name as events



InputEvents: [push, push, pop]

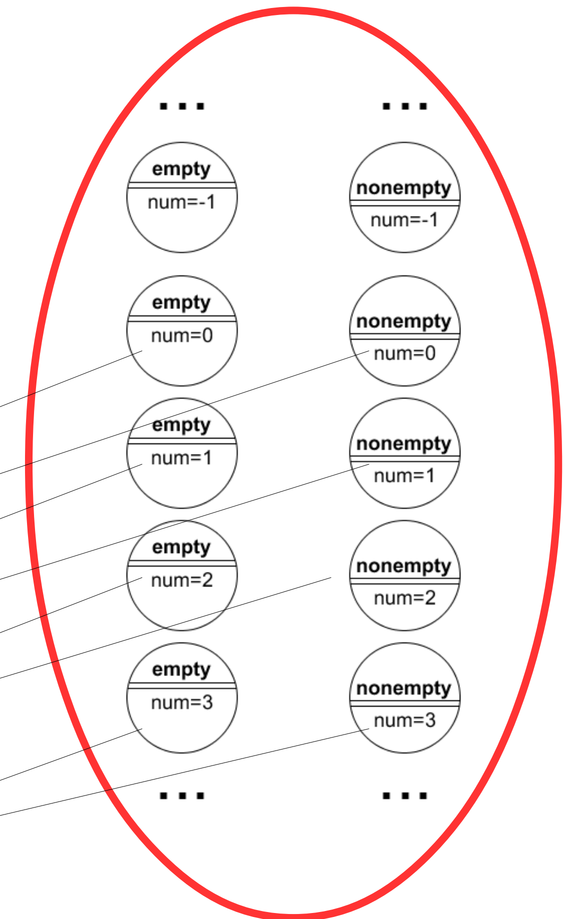
Bridging Statemachine and Impl.

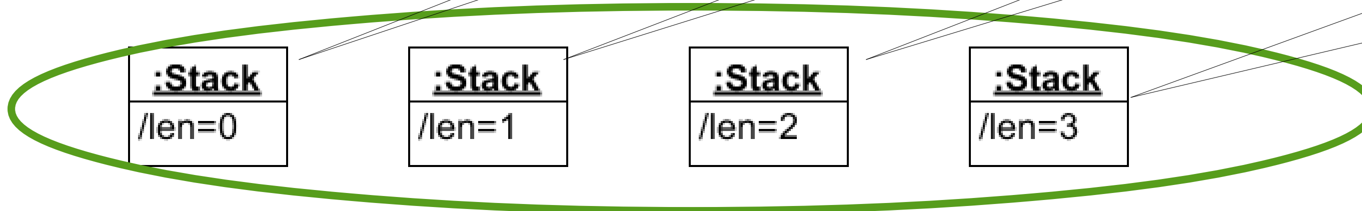
A **bridge** is a relation over the both statespaces (statemachine and implementation).

We define the bridge by attaching a predicate on each state, for example:

```
inState(empty) -> num=len
inState(nonempty) -> num=len
```

Statespace of statemachine



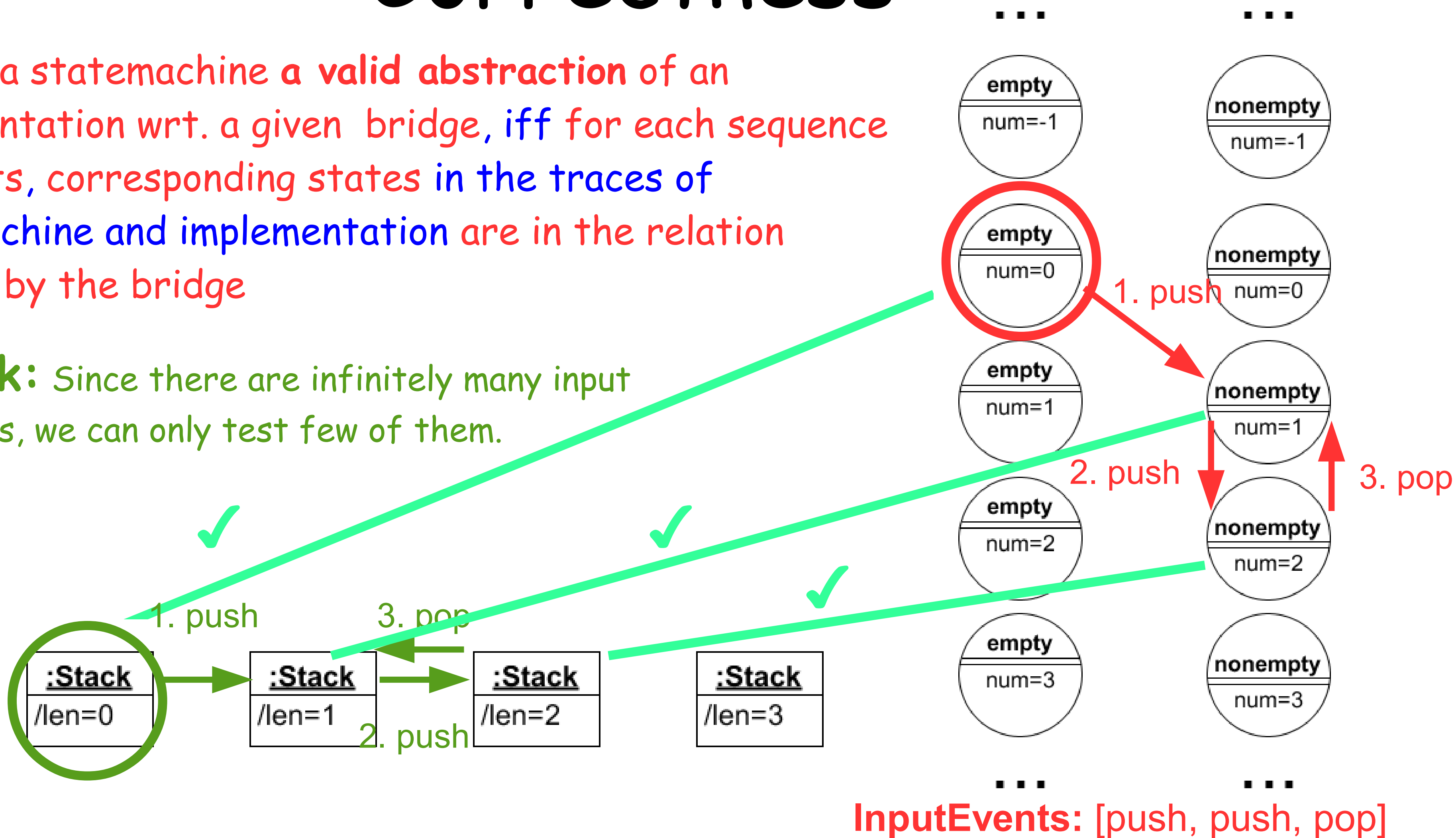


Statespace of implementation

Correctness

We call a statemachine **a valid abstraction** of an implementation wrt. a given bridge, iff for each sequence of events, corresponding states in the traces of statemachine and implementation are in the relation defined by the bridge

Remark: Since there are infinitely many input sequences, we can only test few of them.

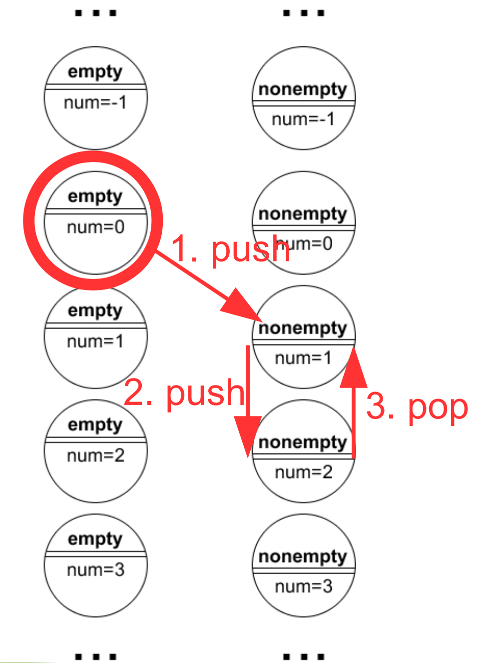


Q1 is answered (but requires in practice the definition of a bridge)

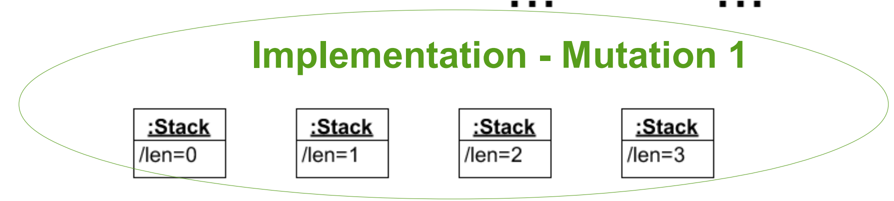
Idea for Measuring Abstractness

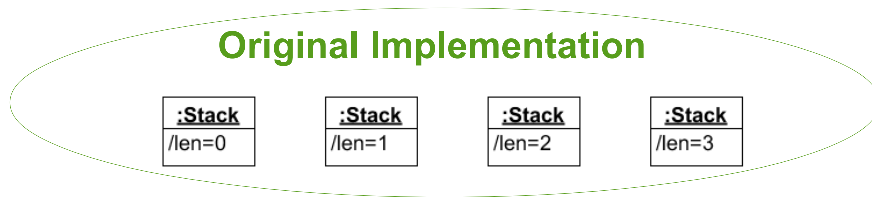
Repeating Runs on Mutated Implementations

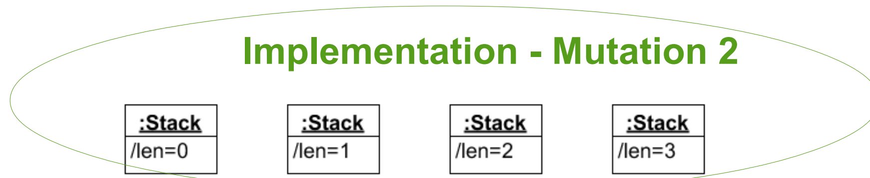
Idea: The runs certifying the correctness of a statemachine are repeated on changed (**mutated**) implementations.



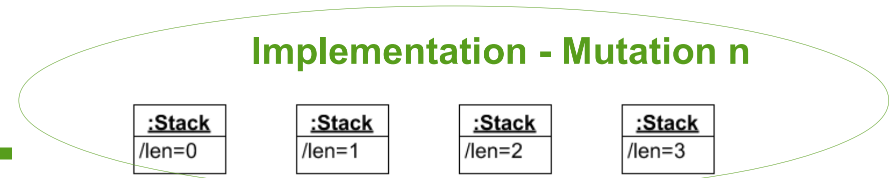
InputEvents: [push, push, pop]







...



Computing the Abstractness

A statemachine is considered more abstract wrt. an implementation and a bridge, the more traces are still correct. So, we define the abstractness as follows:

$$\textbf{abstractness}(\text{sm}) = \frac{\text{number of correct traces on mutated implementation}}{\textbf{number of all traces}\text{ on mutated implementation}}$$

Examples:

- abstracness(sm) = 0
  - All traces on all mutations fail.
- abstractness(sm) = 1
  - All traces are still correct (statemachine works fine for all mutated versions of the implementation!)

Problems/Things to discuss

Correctness: There are infinitely many sequences of events!

- How to become confident that statemachine is correct for ALL possible input(event) sequences?

Abstractness: For the computation of abstractness, not all traces can be taken into account!

- How to select the representative cases?
  - If a mutated implementation fails for [e1,e2,...,ek], it will also fail for [e1,e2,...,ek, ..., en]

Summary

- We addressed a problem of quality assessment
  - If models are too abstract, they don't tell any interesting story
- Abstraction measurement of statemachines is in literature done using structural criteria (counting states, transitions, etc.)
  - Our approach needs working implementation and formally defined bridge.