The derivation of object behavior from source code

Dominik Rauner−Reithmayer

UNIVERSITÄT KLAGENFURT

Outline

- Motivation
- States, events and source−code
- Class behavior
- Object behavior
- The derivation of object behavior from source code
- Conclusion and Outlook

Motivation

Reverse engineering and object−orientation

- Three different aspects within object−orientation:
  1) **structural** aspect
  2) **dynamic** aspects
  3) **functional** aspect

- Reverse engineering covers only two aspects:
  1) **structural** aspect with object−identification
  2) **functional** aspect with program−understanding

Motivation
Dynamics on code level

- How can we identify the dynamics of a class or object on the code level?
  1) How can we identify a state on the code level?
  2) How can we identify an event on the code level?
  3) How are states and events connected on the class (object) level?
  4) How can we interpret the so identified dynamics on a higher level of abstraction?

States, events and source−code

States (1)

- Two typical methods to implement states:

1. explicit state
   - ✗ The state of an object depends directly on the values of some or all attributes.

2. implicit state
   - ✗ The state of an object depends on the relation to other objects.

The most natural implementation →

explicit state

States, events and source−code

States (2)

Definition 1 (State−indicator): A state−indicator for a class C is an attribute of C for which both off the following conditions hold:

1) The value of the attribute is used or defined in at least two methods of C.
2) The attribute appears in at least one condition, which is controlling state−indicator defining statements.

Definition 2 (State): A state is the set of tuples containing the state−indicator values, for which the same state indicator defining statements can be performed.

States, Events and Source−code
Events

What is an event in procedural source−code?

Method Call ⇔ Event

1. (+) A method call depends not on the state of the object.
2. (+) Within the method the state of an object can be changed.
3. (+/−) Compared with the lifetime of an object, the time a method needs to perform its work is not important.
4. (−) A method call is a two way communication (if the procedure has a return value)

Class behavior
State−event diagrams (1)

- Formally, a state−event diagram can be seen as a state event automaton:

  A state−event automaton is a quintupel (Z,E,δ,z_0,Z_E)

  Z : set of states
  E : set of events
  δ : transition−function
  z_0 : starting state
  Z_E : set of end−states

Class behavior
State−event diagrams (2)

- To generate a state−event automaton from source code we need:
  - The set of identified states S
  - The set of identified events E_K
  - The transition−function f_e for every event e in E_K

    The derivation of the transition function is the most complex part.

Class behavior

Generation of a state−event automaton

- The state event automaton $ZEA(S,E_K,F_{trans},z_{init},\{z_{final}\})$ can be generated in four steps:

1) $S=Z \cup \{z_{init},z_{final}\}$

2) $F_{trans}=\{((s,e,b),ss)|((s,b),ss) \in f_e \wedge s \in Z \wedge ss \in Z\}$

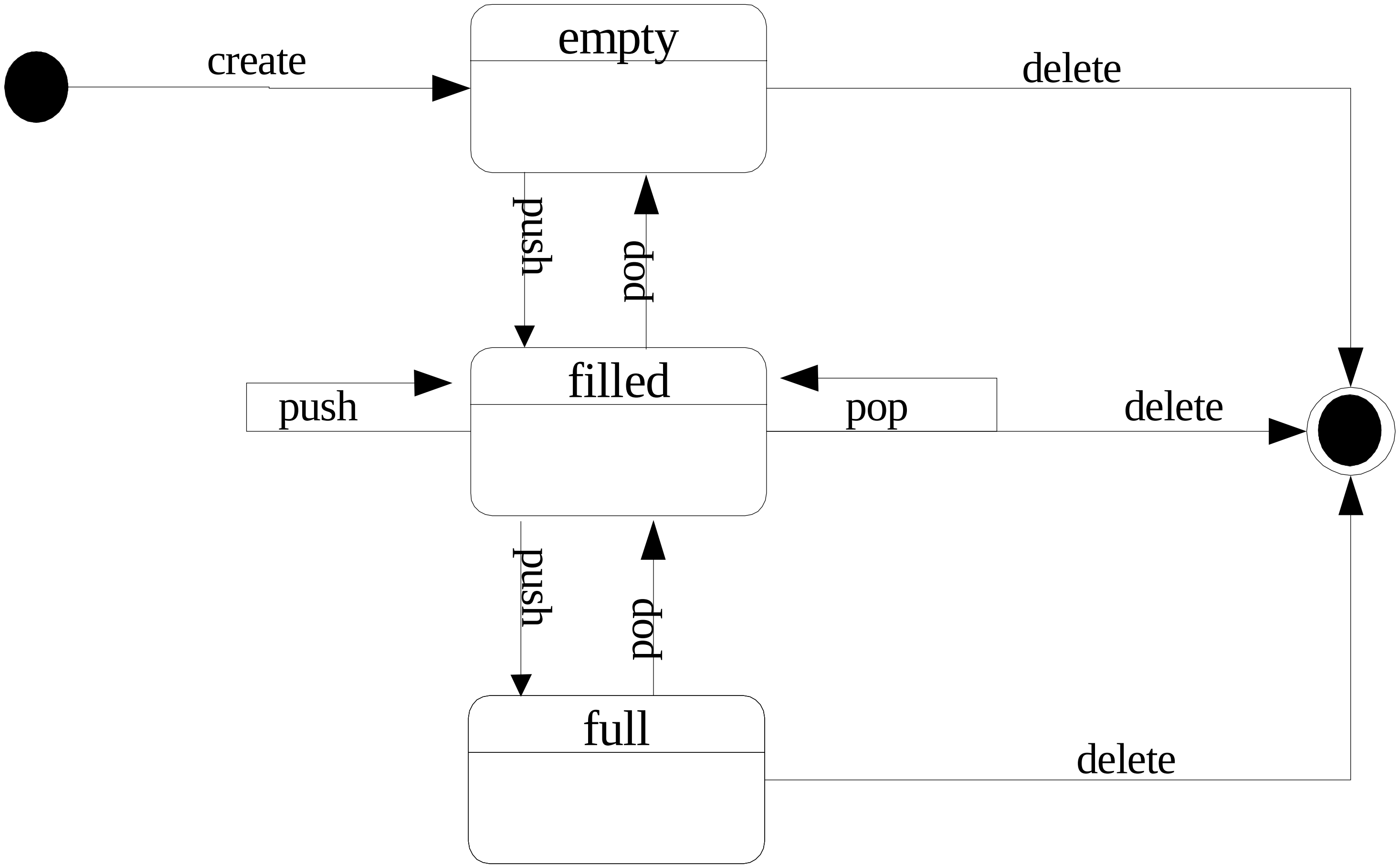
3) $F_{trans}=F_{trans} \cup \{((s,e,b),ss)|((\epsilon,b),ss) \in f_e \wedge s = z_{init} \wedge ss \in Z\}$

4) $F_{trans}=F_{trans} \cup \{((s,e,b),ss)|((s,b),\emptyset) \in f_e \wedge s \in Z \wedge ss=z_{final}\}$

Class behavior
Example(1)

- States S={empty, full, filled}
- Events E={create, delete, push, pop}
- Transition functions:
  - » f_{create}={(ϵ, empty)}
  - » f_{push}={(empty, filled),(filled, filled), (filled, full)}
  - » f_{pop}={(filled, empty),(filled, filled), (full, filled)}
  - » f_{delete}={(empty,∅),(filled, ∅),(full,∅)}

Class behavior
Example(2)



Object behavior

- Problem:

  To describe the potential behavior of one object in its context.

  – What is the context of an object?:

    - event trace (dynamic trace)
    - control flow graph (static trace)

Object behavior
event trace vs. control flow graph

- event trace
  - – exact description
  - – hard to get
  - – not complete
- control flow graph
  - – description is not so exact
  - – easy to get
  - – complete

Object behavior and source code
Idea

- Reduce the class state−event automaton to a state event automaton that describes only the behavior of one concrete object.

  – The idea is to associate every node in the CFG with states in class state−event diagram.

Object behavior and source code
Algorithm

function *restrict* $((S_K, E_K, \delta_K, q_{K_0}, q_{K_{final}}), (V_{CFG}, A_{VFG}, En, Ex))$:

$(S_o, E_o, \delta_o, q_{o_0}, q_{o_{final}})$

$\delta_o = \emptyset; q_{o_0} = q_{K_0}; V = \emptyset; toV = \{(En, q_{K_0})\}$

repeat

$V = toV \cup V$

$toV' = \{(n_t, q_t) \mid$

$\exists (n_f, q_f) \in toV \bullet (n_f, n_t) \in A_{CFG} \wedge (Event(n_t) \wedge (\exists ((q, e, b), ss) \in \delta_K \mid q_f = q \wedge e = n_t.name \bullet q_t \in ss))$

$\vee (Condition(n_t) \wedge q_f = q_t)\}$

$\delta_o = \delta_o \cup \{((q_o, e_o, b_o), ss_o) \in \delta_K \mid \exists (n_f, q_f) \in toV \mid q_o = q_f \bullet$

$\forall (n_t, q_t) \in toV' \mid (n_f, n_t) \in A_{CFG} \wedge Event(n_t) \wedge e_o = n_t.name \bullet q_t \in ss_o\}$

$toV = toV'$

until $(toV \subseteq V)$

$S_o = \{q \mid \exists ((q_f, e, b), ss) \in \delta_o \bullet q = q_f \vee q \in ss\}$

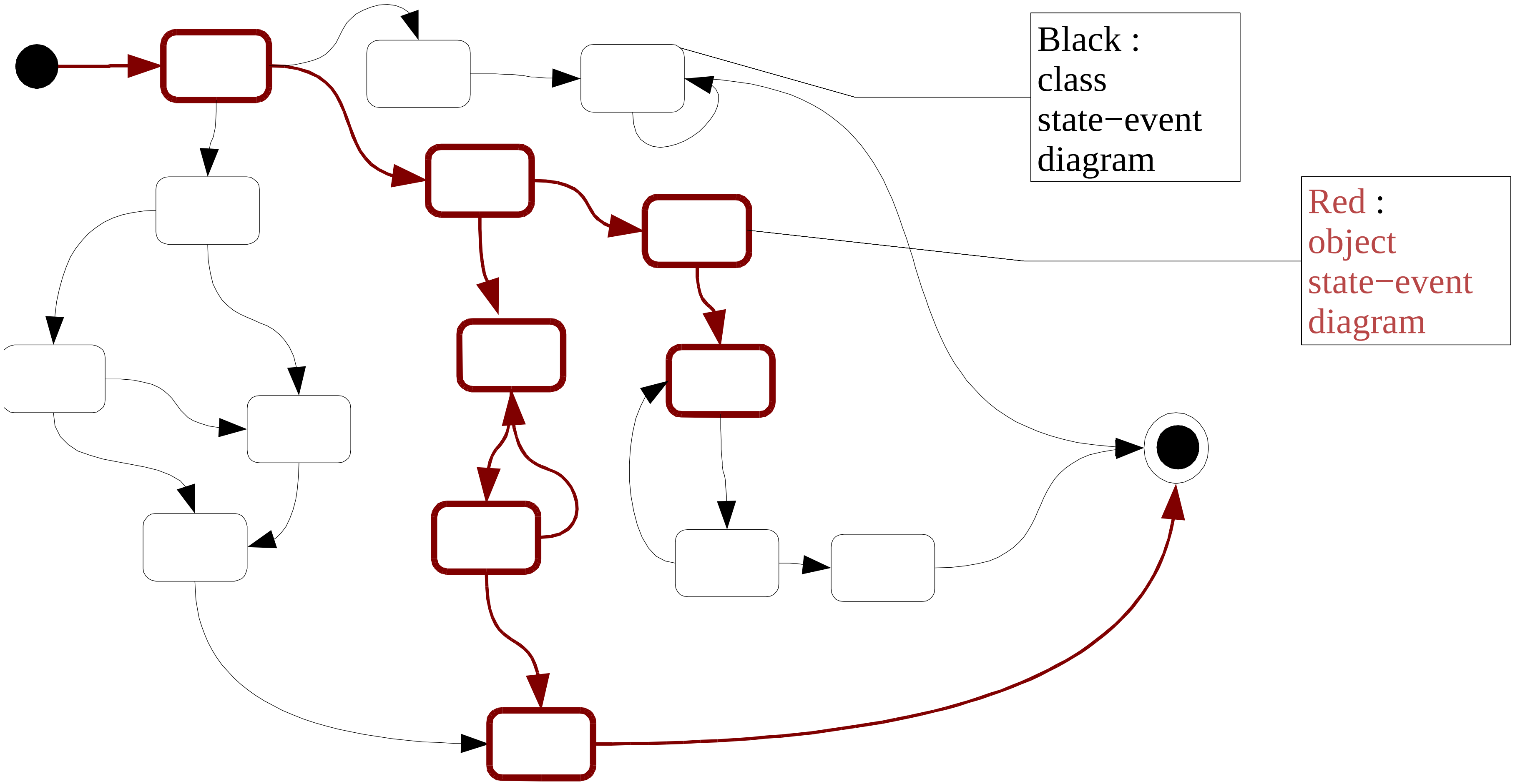
$E_o = \{event \mid \exists ((q_f, e, b), ss) \in \delta_o \bullet e = event$

$q_{o_{final}} = S_o \cap q_{K_{final}}$

end function

Object behavior and source code

Example



Object behavior
Special cases

I. There are states in the state−event diagram with no path to a final state.

II. There are no final states in the state−event diagram.

Reasons:

- No corresponding event in the CFG.
- Not all state changes occur within methods.

Conclusion and Outlook

- Behavior abstraction is necessary for oo reverse engineering.
- It is possible to derive class state−events diagrams with a simple abstraction mechanism.
- It is possible to describe concrete object behavior on the same abstraction level.

Future Work:

- INTEPRETATION