Program Testing and Analysis:

Symbolic and Concolic Testing (Part 2)

Dr. Michael Pradel

Software Lab, TU Darmstadt

Warm-up Quiz

What does the following code print?

```
var sum = 0;
var array = [11, 22, 33];
for (x in array) {
  sum += x;
}
console.log(sum);
```

112233 **0012** **66** **Something else**

Warm-up Quiz

What does the following code print?

```
var sum = 0;
var array = [11, 22, 33];
for (x in array) {
  sum += x;
}
console.log(sum);
```

112233 **0012** **66** Some JS engines **Something else**

Warm-up Quiz

What does the following code print?

```
var sum = 0;
var array = [11, 22, 33];
for (x in array) {
  sum += x;
}
console.log(sum);
```

Arrays are objects

For-in iterates over object property names (not property values)

112233 | **0012** | **66** | Some JS engines **Something else**

Warm-up Quiz

What does the following code print?

```
var sum = 0;
var array = [11, 22, 33];
for (x in array) {
  sum += x;
}
console.log(sum);
```

For arrays, use traditional for loop:

```
for (var i=0;
     i < array.length;
     i++) ...
```

112233 | **0012** | 66 | Some JS engines: **Something else**

Outline

1. Classical Symbolic Execution
2. Challenges of Symbolic Execution
3. Concolic Testing 
4. Large-Scale Application in Practice

Mostly based on these papers:

- *DART: directed automated random testing*, Godefroid et al., PLDI'05
- *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*, Cadar et al., OSDI'08
- *Automated Whitebox Fuzz Testing*, Godefroid et al., NDSS'08

Problems of Symbolic Execution

- Loops and recursion: Infinite execution trees
- Path explosion: Number of paths is exponential in the number of conditionals
- Environment modeling: Dealing with native/system/library calls
- Solver limitations: Dealing with complex path conditions
- Heap modeling: Symbolic representation of data structures and pointers

Problems of Symbolic Execution

- Loops and recursion: Infinite execution trees
- Path explosion: Number of paths is exponential in the number of conditionals
- Environment modeling: Dealing with native/system/library calls
- Solver limitations: Dealing with complex path conditions
- Heap modeling: Symbolic representation of data structures and pointers

One approach: Mix symbolic with concrete execution

Concolic Testing

Mix concrete and symbolic execution = ”concolic”

- Perform concrete and symbolic execution side-by-side
- Gather path constraints while program executes
- After one execution, negate one decision, and re-execute with new input that triggers another path

Example

hand-written notes

Exploring the Execution Tree

(hand-written notes)

Algorithm

Repeat until all paths are covered

- Execute program with concrete input i and collect symbolic constraints at branch points: C
- Negate one constraint to force taking an alternative branch b': Constraints C'
- Call constraint solver to find solution for C': New concrete input i'
- Execute with i' to take branch b'
- Check at runtime that b' is indeed taken
  Otherwise: ”divergent execution”

Divergent execution: Example

```
function f(a) {
  if (Math.random() < 0.5) {
    if (a > 1) {
      console.log("took it");
    }
  }
}
```

First exec.

$a = 0$

taken
not taken

Second exec

$a = 2$

not taken

→ divergent

Path constraint:

$a_0 \leq 1$

Solver:

$a = 2$

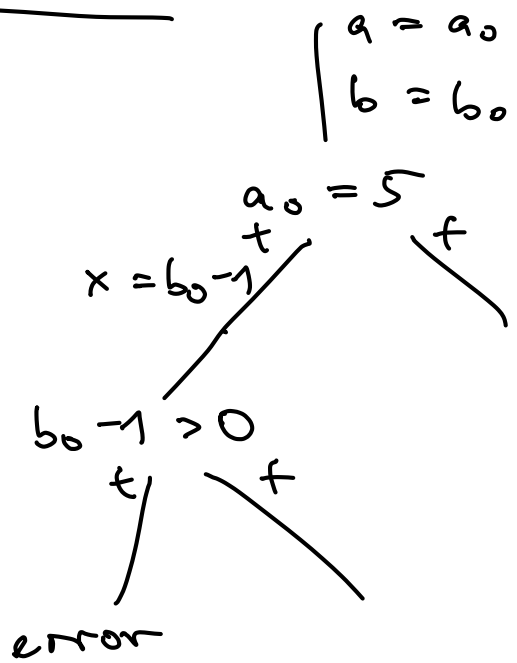
Quiz

After how many executions and how many queries to the solver does concolic testing find the error?

Initial input: `a=0, b=0`

```
function concolicQuiz(a, b) {
  if (a === 5) {
    var x = b - 1;
    if (x > 0) {
      console.log("Error");
    }
  }
}
```

Quiz:



Exec 1

$a = 0, b = 0$

Solve $a_0 = 5$ → $a_0 = 5$

Exec 2

$a = 5, b = 0$

Solve: $b_0 - 1 > 0 \wedge a_0 = 5$

→ $a_0 = 5, b_0 = 2$

Exec 3

→ reach error

⇒ 3 executions, 2 queries

Benefits of Concolic Approach

When symbolic reasoning is impossible or impractical, fall back to concrete values

- Native/system/API functions
- Operations not handled by solver (e.g., floating point operations)

Outline

1. Classical Symbolic Execution
2. Challenges of Symbolic Execution
3. Concolic Testing
4. Large-Scale Application in Practice ←

Mostly based on these papers:

- *DART: directed automated random testing*, Godefroid et al., PLDI'05
- *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*, Cadar et al., OSDI'08
- *Automated Whitebox Fuzz Testing*, Godefroid et al., NDSS'08

Large-Scale Concolic Testing

- SAGE: Concolic testing tool developed at Microsoft Research
- Test robustness against unexpected inputs read from files, e.g.,
  - Audio files read by media player
  - Office documents read by MS Office
- Start with known input files and handle bytes read from files as symbolic input
- Use concolic execution to compute variants of these files

Large-Scale Concolic Testing (2)

- Applied to hundreds of applications
- Over 400 machine years of computation from 2007 to 2012
- Found hundreds of bugs, including many security vulnerabilties
  - One third of all the bugs discovered by file fuzzing during the development of Microsoft's Windows 7

Details: Bounimova et al., ICSE 2013

Summary: Symbolic & Concolic Testing

Solver-supported, whitebox testing

- Reason symbolically about (parts of) inputs
- Create new inputs that cover not yet explored paths
- More systematic but also more expensive than random and fuzz testing
- Open challenges
  - Effective exploration of huge search space
  - Other applications of constraint-based program analysis, e.g., debugging and automated program repair

Program Testing and Analysis:

GUI Testing

Dr. Michael Pradel

Software Lab, TU Darmstadt

GUI Testing

- **Test application via its graphical user interface (GUI)**
- **Possible approaches**
  - Manual testing
  - Semi-automated, e.g., Selenium
- **Here: Automated GUI testing**
  - Purely random testing, e.g., Monkey for Android
  - Today: Sophisticated approaches beyond purely random testing

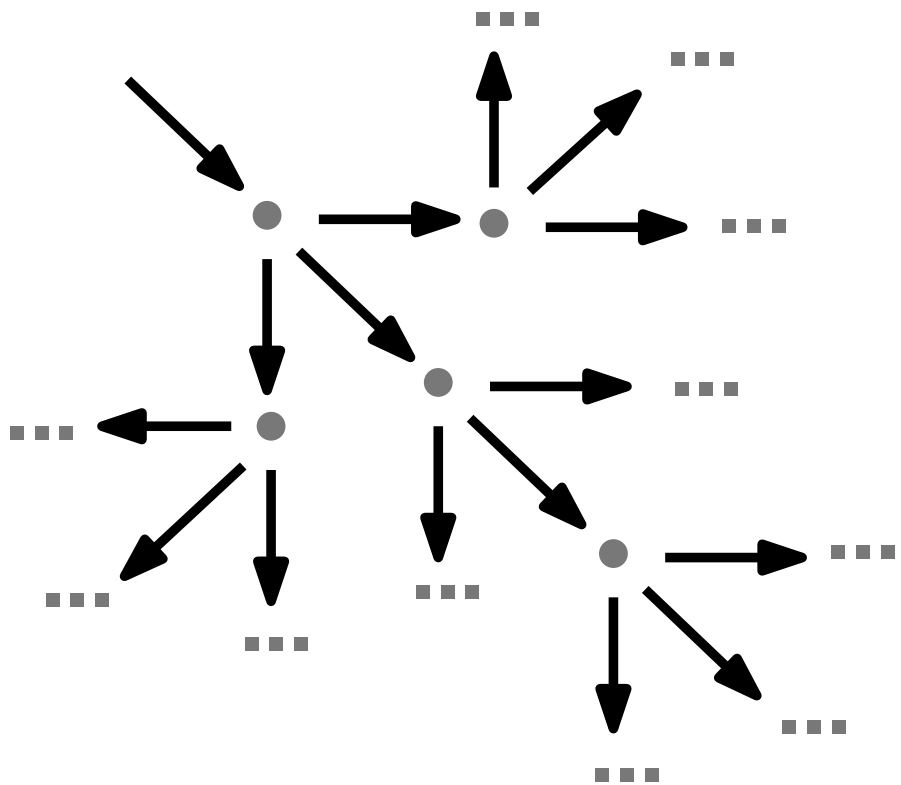
Challenges

- **Two input dimensions**
  - Sequences of events, e.g., clicks, mouse movements (focus of today's lecture)
  - Input values, e.g., strings entered into form
- **Not all events are known ahead of time**
  - Web apps load content dynamically
- **Need a test oracle**
  - When does an execution expose a bug?
- **Huge search space**

Huge Search Space

Challenge: Huge search space

- Too large to explore exhaustively

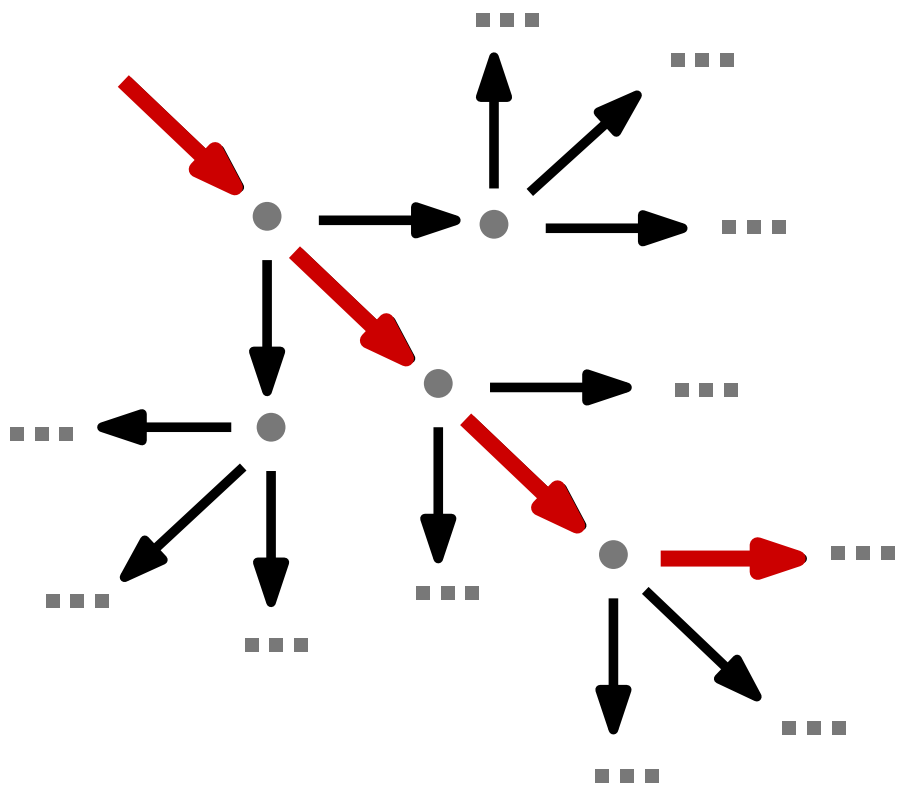


Huge Search Space

Approach:

Steer search toward potential bugs

or not yet explored behavior



Outline

- **Feedback-directed GUI testing**

  Based on *A Framework for Automated Testing of JavaScript Web Applications*, Artzi et al., ICSE 2011

- **Model inference-based testing**

  Based on *Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning* , Choi et al., OOPSLA 2013

- **Responsiveness testing**

  Based on *EventBreak: Analyzing the Responsiveness of User Interfaces through Performance-Guided Test Generation*, Pradel et al., OOPSLA 2014

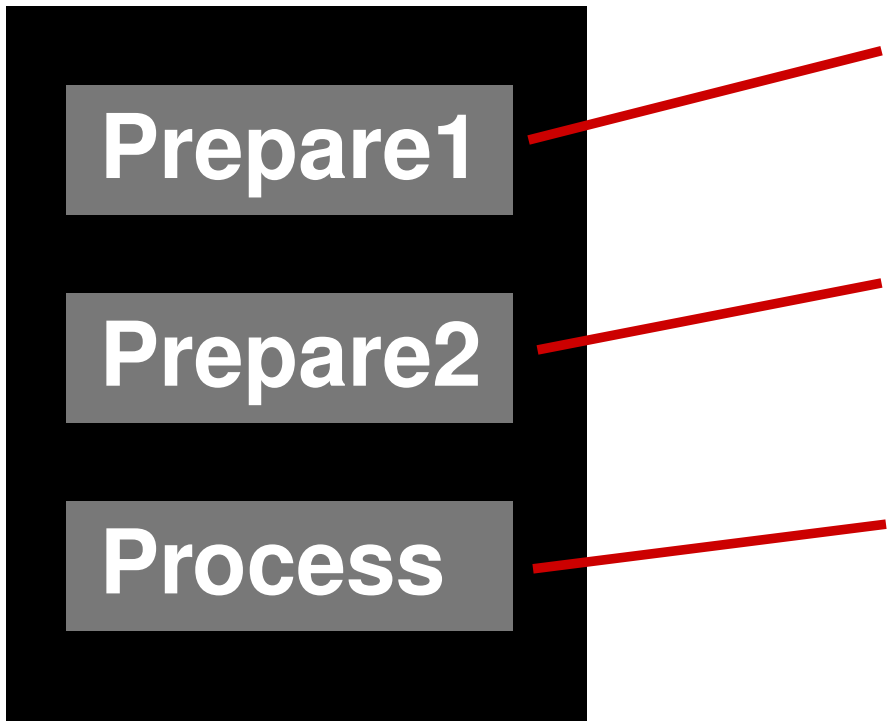
Artemis

Feedback-directed generation of GUI tests

- Start with randomly selected events
- Gather feedback from execution
- Steer toward particularly interesting behavior
- Implemented for web applications
- Test oracle: JavaScript exceptions and invalid HTML

Example

Application with 3 buttons:




```
prepared1 = true;
```

```
prepared2 = true;
```

```
if (prepared1)
  if (prepared2)
    window.alert("hi");
```

Initially, `prepared1=prepared2=false`

Gathering Feedback

Feedback gathered while executing generated sequences of events:

- Available events
- Source code of handlers attached to events
- Memory locations read & written
- Branch coverage

Artemis: Algorithm

- Input: URL u
- Add sequence $[load\ u]$ to worklist
- While worklist not empty
  - Execute next sequence and gather feedback
  - Add new sequences to worklist
    - Modify inputs of existing sequence
    - Extend sequence with additional event
    - Create new sequence with new start URL
  - Prioritize worklist

Prioritization 1

Coverage-guided prioritization

- Keep track of branch points in each handler
  - Branch point = entry of handler or control flow branch
- Prioritize sequences that trigger handlers with low coverage

$$P(e_1, ..., e_k) = 1 - cov(e_1) \cdot ... \cdot cov(e_k)$$

where

$$cov(e) = \frac{\text{covered branch points of } e\text{'s handler}}{\text{all discovered branch points of } e\text{'s handler}}$$

Coverage-guided prioritization: Example

Suppose to have executed:

Prepare 2 , Process

$$cov(Prep\,1) = \frac{0}{1} = 0$$

$$cov(Prep\,2) = \frac{1}{1} = 1$$

$$cov(Process) = \frac{2}{5}$$

Possible next sequences:

- Prepare 2, Process, Prepare 1 $\rightarrow$ $P = 1 - 1 \cdot \frac{2}{5} \cdot 0 = 1$
- — " — , Prepare 2 $\rightarrow$ $P = 1 - 1 \cdot \frac{2}{5} \cdot 1 = \frac{3}{5}$
- — " — , Process $\rightarrow$ $P = 1 - 1 \cdot \frac{2}{5} \cdot \frac{2}{5} = \frac{21}{25}$

highest priority $\Rightarrow$ execute next

Priorization 2

Prioritize based on read/write sets

- Keep track of memory locations read/written by each handler
- Prioritize sequences where some handlers write values read by a subsequence handler

$$P(e_1, ..., e_k) = \frac{|(w(e_1) \cup ... \cup w(e_{k-1})) \cap r(e_k)| + 1}{|r(e_k)| + 1}$$

- Intuition: Can cover interesting behavior only after some handlers have set the right pre-conditions

R/W set-based prioritization: Example

Suppose has execution:

Process$_1$ | Prepare2$_1$ | Prepare1
r: prep1 | w: prep2 | w: prep1

Possible next sequence:

. ____ u ____ , Prepare1
. ___ -- ____ , Prepare2
. ____ - _____ , Process

} to be completed...

Outline

- **Feedback-directed GUI testing**

  Based on *A Framework for Automated Testing of JavaScript Web Applications*, Artzi et al., ICSE 2011

- **Model inference-based testing** 

  Based on *Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning* , Choi et al., OOPSLA 2013

- **Responsiveness testing**

  Based on *EventBreak: Analyzing the Responsiveness of User Interfaces through Performance-Guided Test Generation*, Pradel et al., OOPSLA 2014

SwiftHand

- **Challenge: Restarting the application is expensive**
- **Learn finite-state model of application while exploring it**
  - Explore states with unknown outgoing transitions
  - Continuously refine model by splitting states
- **Explores application with small number of restarts**

Outline

- **Feedback-directed GUI testing**

  Based on *A Framework for Automated Testing of JavaScript Web Applications*, Artzi et al., ICSE 2011

- **Model inference-based testing**

  Based on *Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning* , Choi et al., OOPSLA 2013

- **Responsiveness testing** 

  Based on *EventBreak: Analyzing the Responsiveness of User Interfaces through Performance-Guided Test Generation*, Pradel et al., OOPSLA 2014

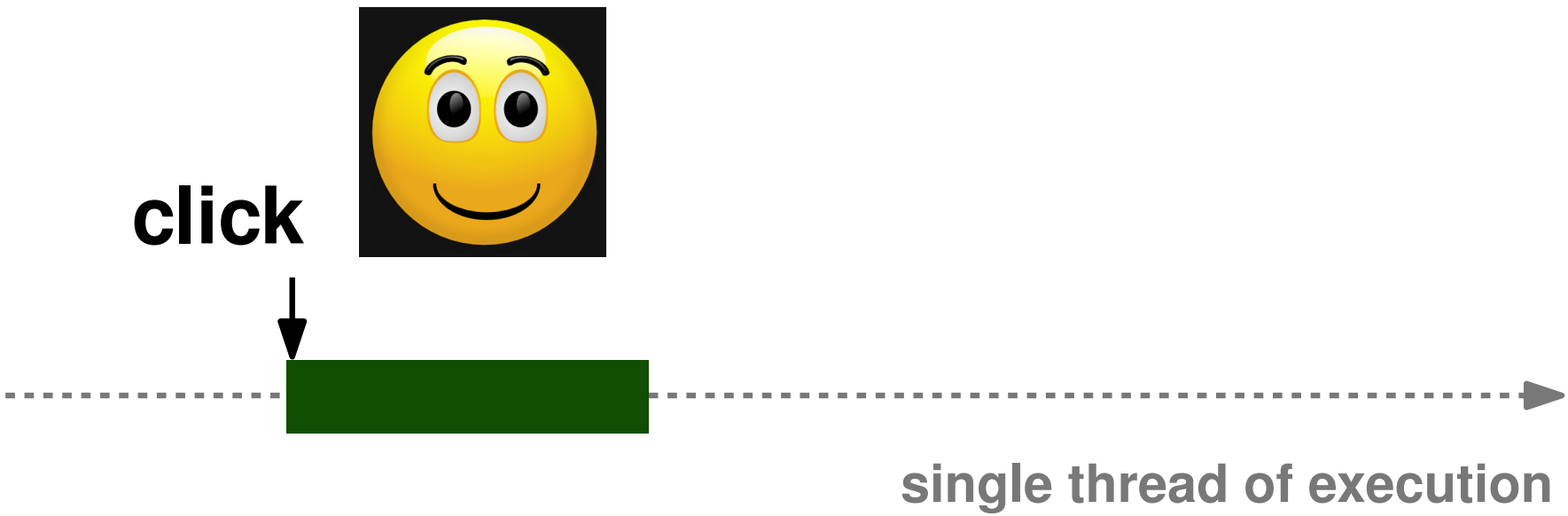
Motivation

Event-based UI applications should be responsive

single thread of execution

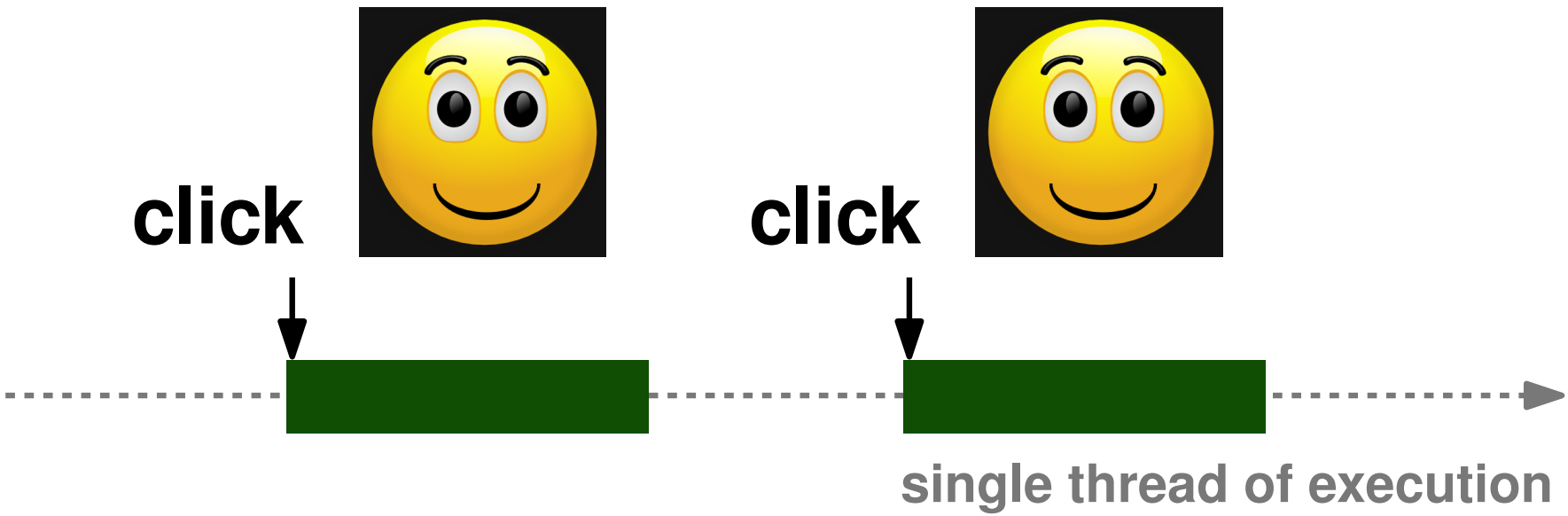
Motivation

Event-based UI applications should be responsive



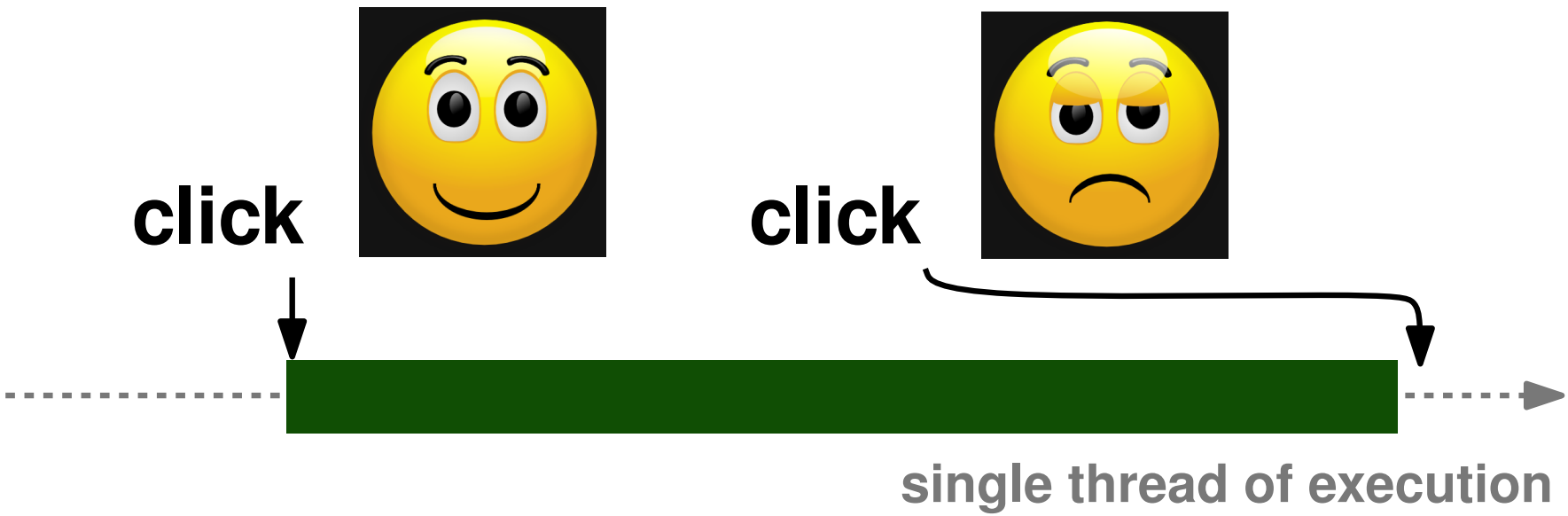
Motivation

Event-based UI applications should be responsive



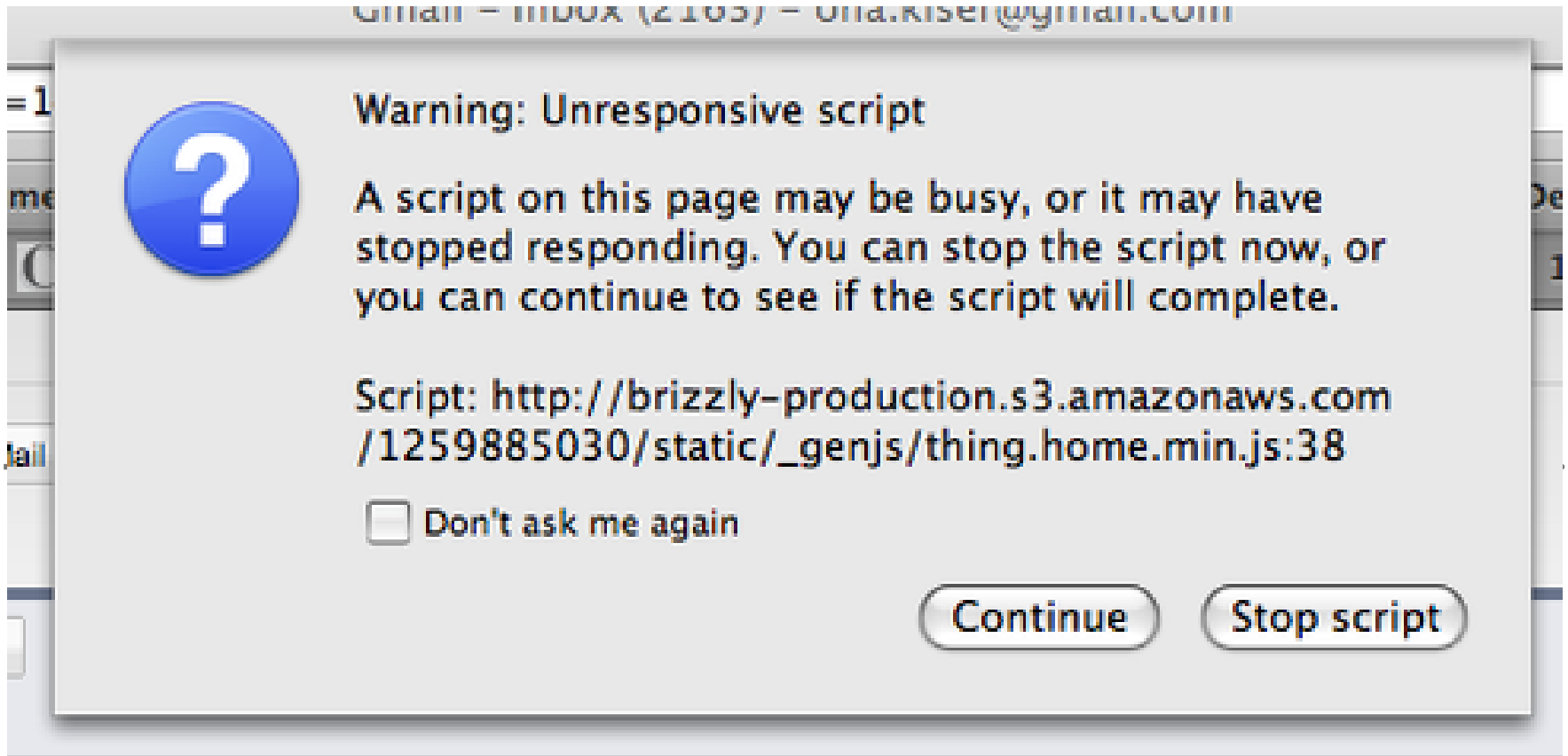
Motivation

Event-based UI applications should be responsive

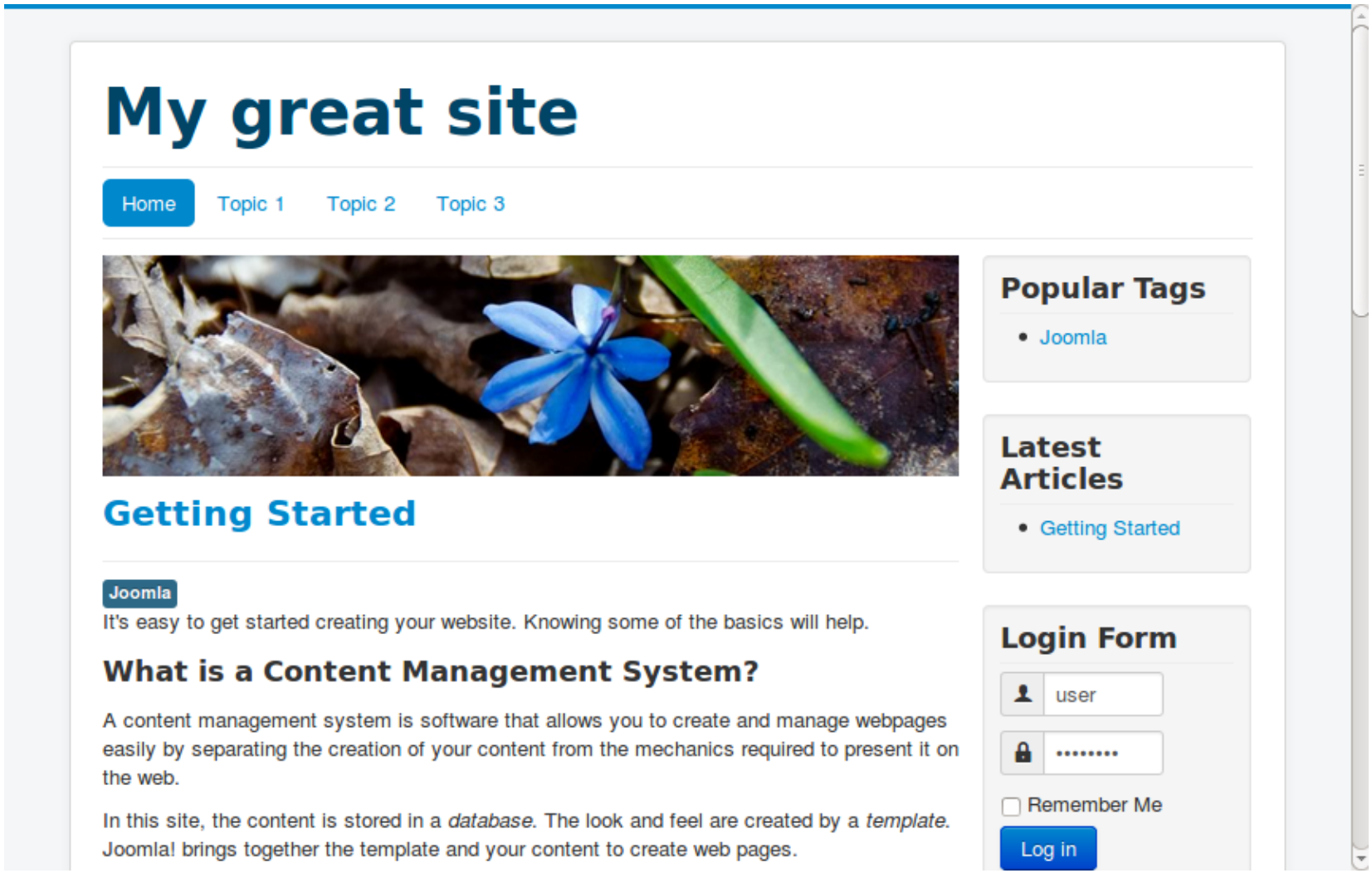


Motivation

Event-based UI applications should be responsive

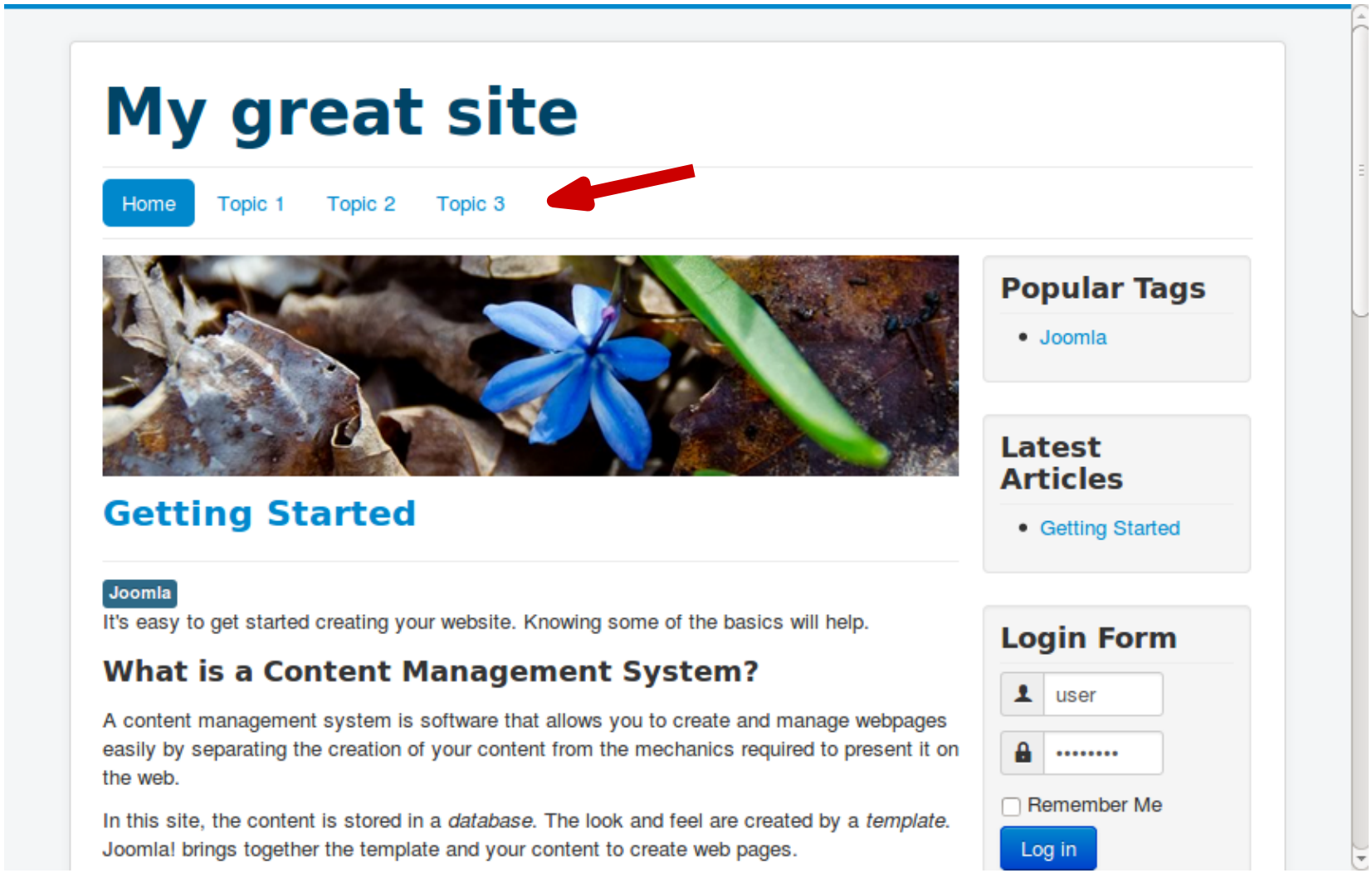


Real-World Example



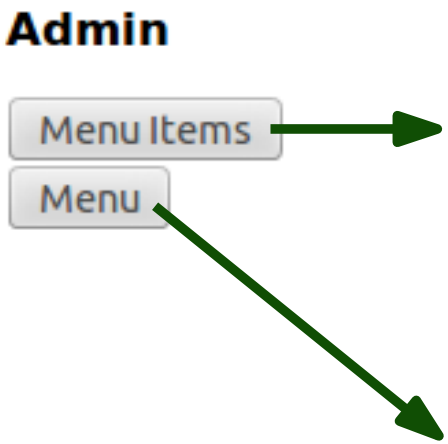
Joomla bug 30274

Real-World Example



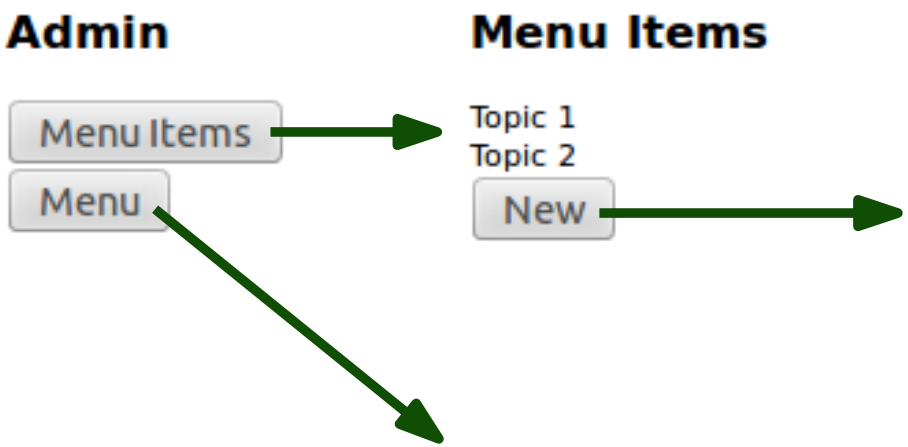
Joomla bug 30274

Real-World Example



Joomla bug 30274

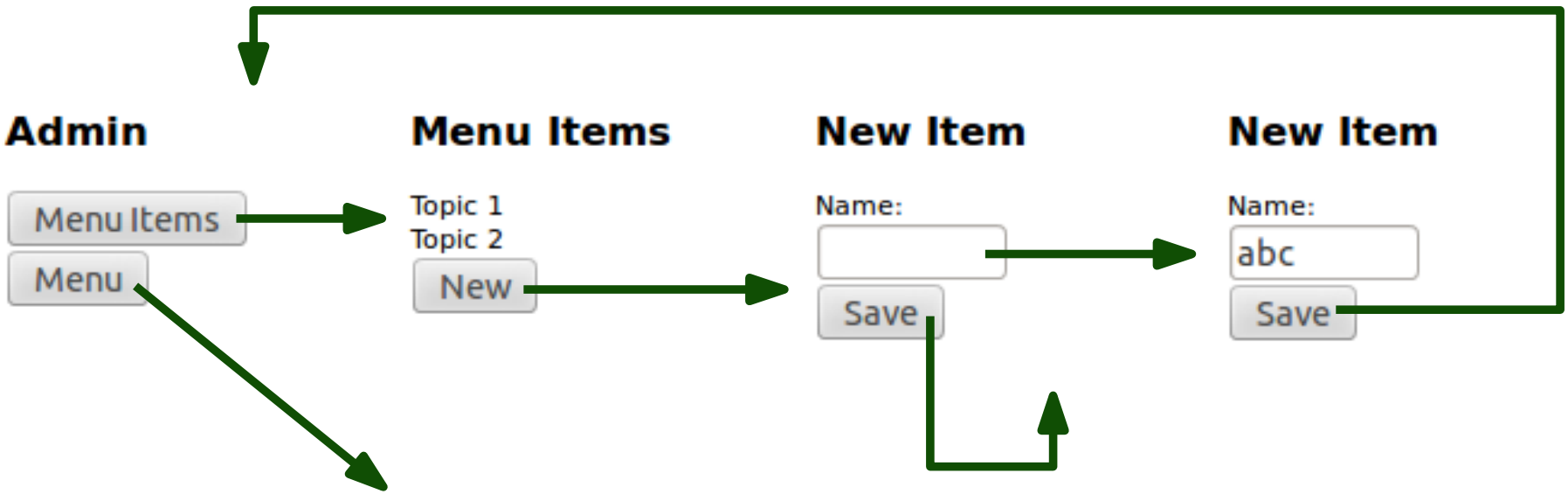
Real-World Example



Real-World Example

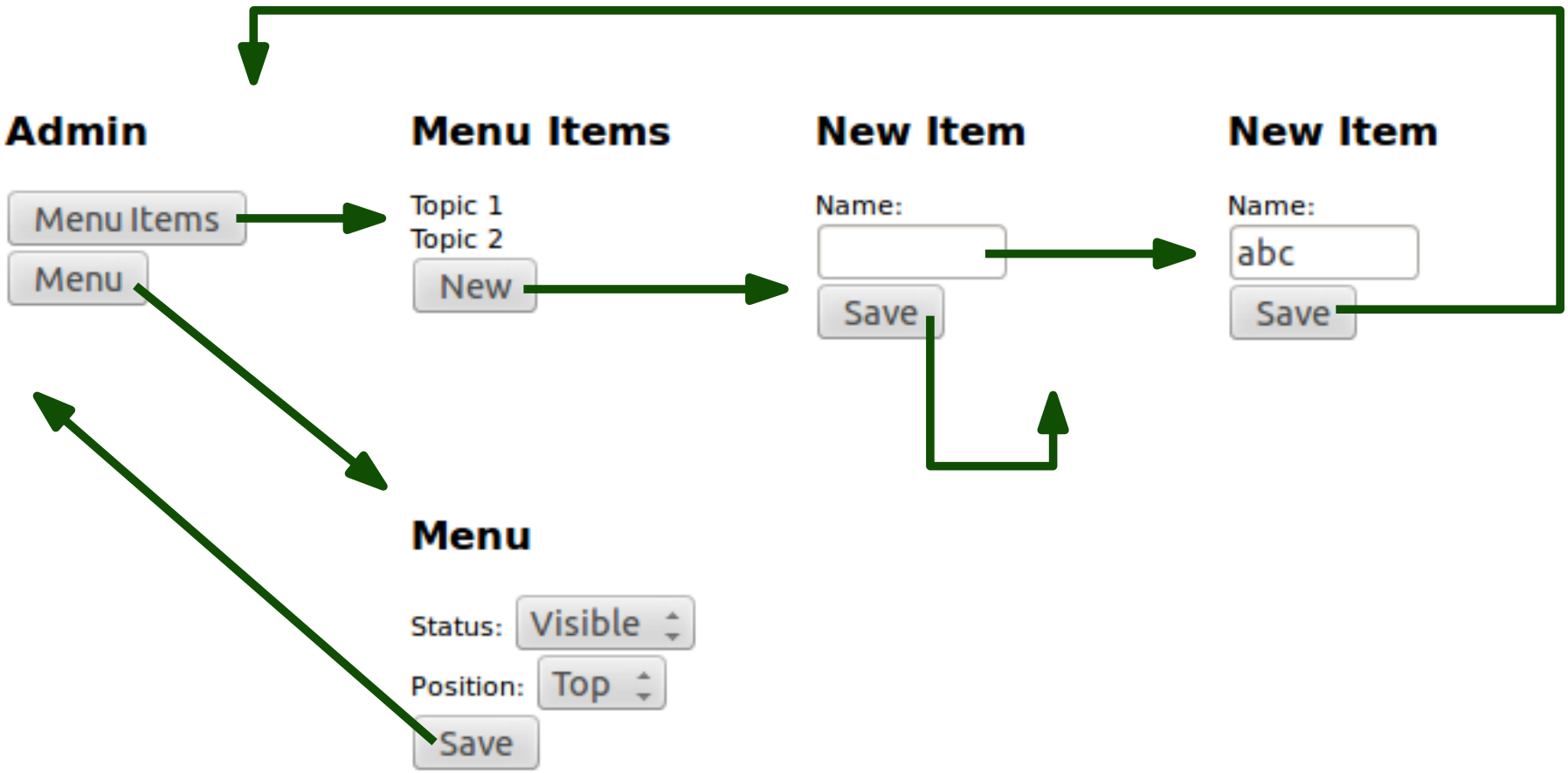
Joomla bug 30274

Real-World Example



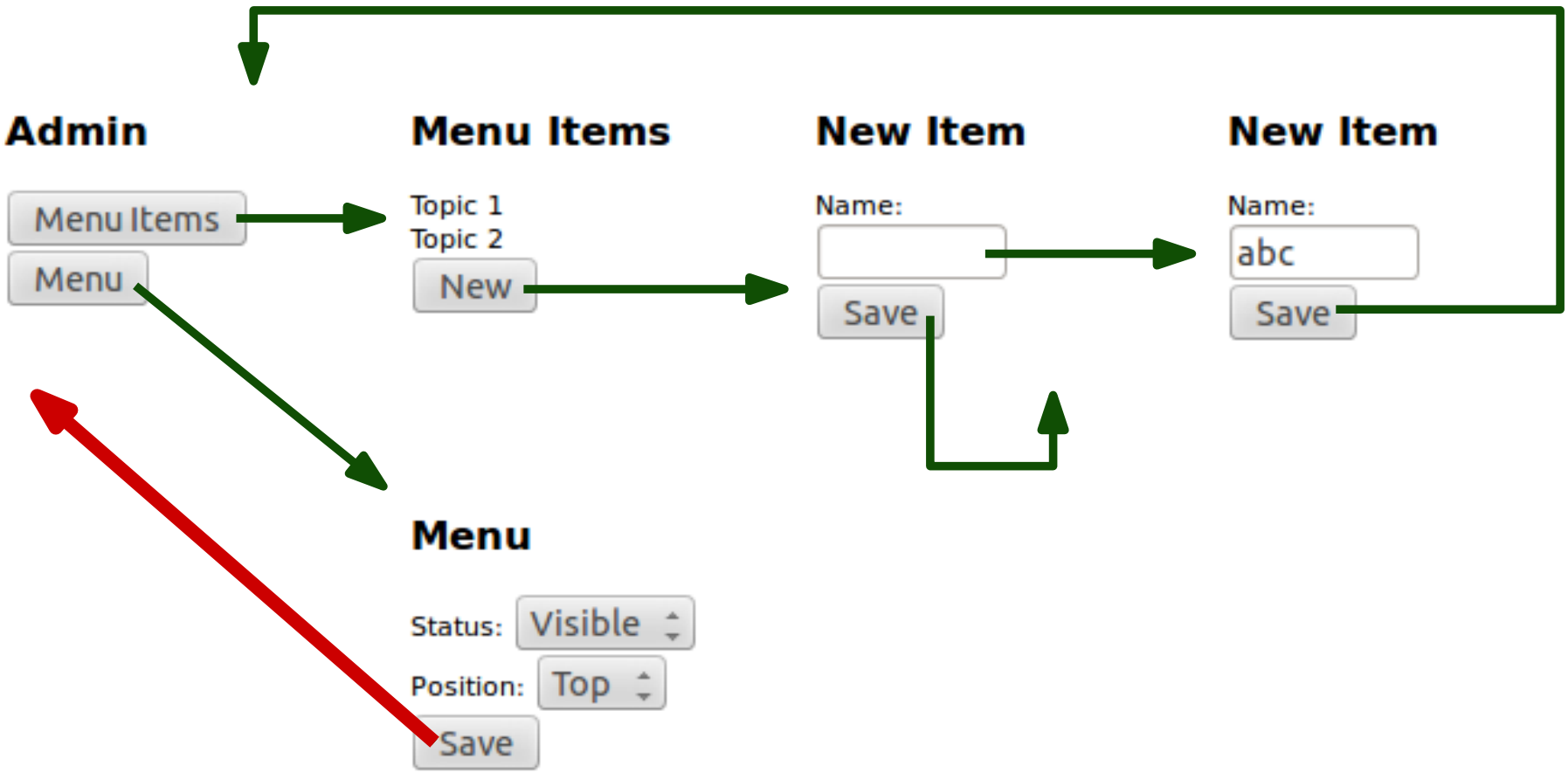
Joomla bug 30274

Real-World Example



Joomla bug 30274

Real-World Example

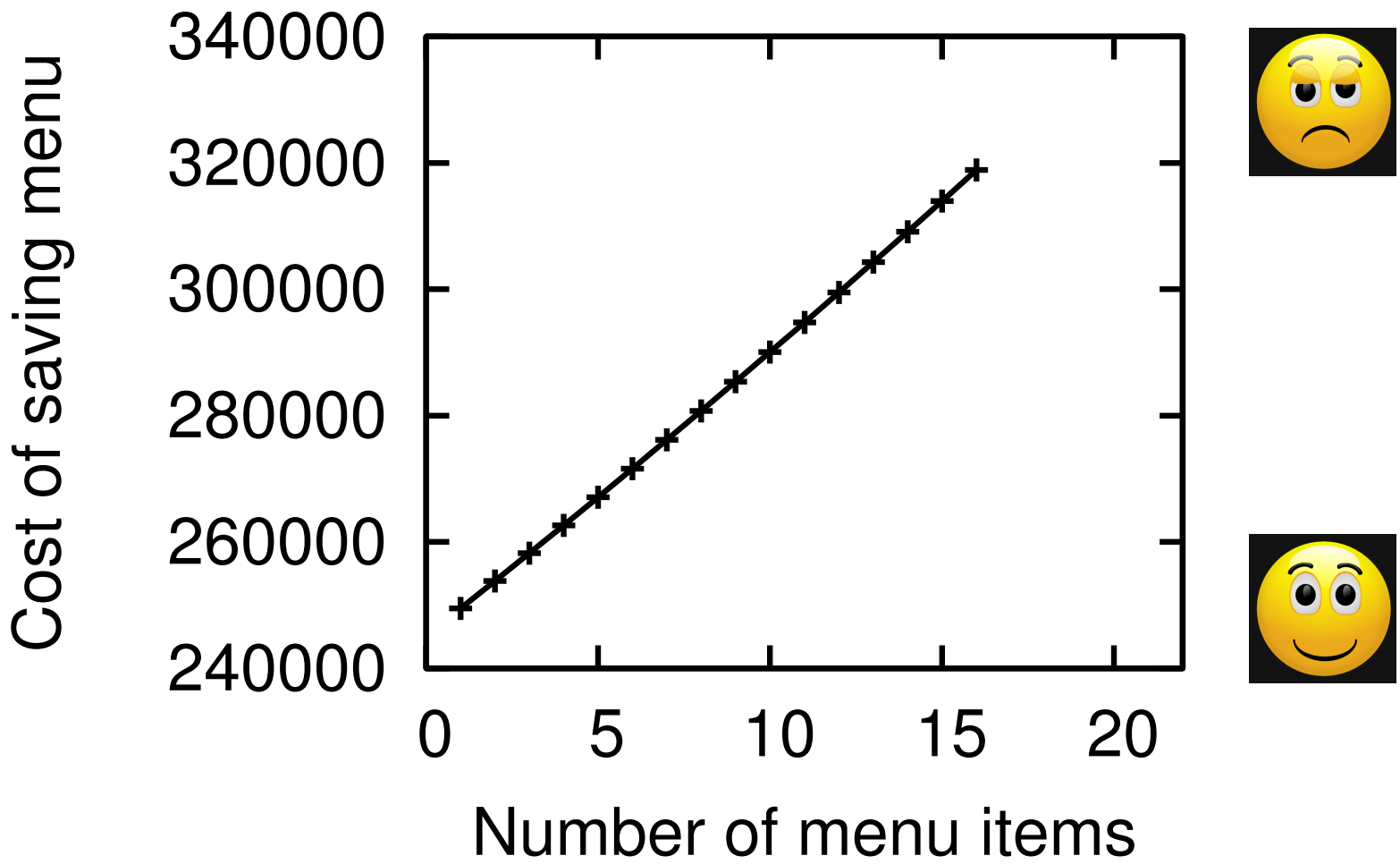


Unresponsive

Joomla bug 30274

Real-World Example

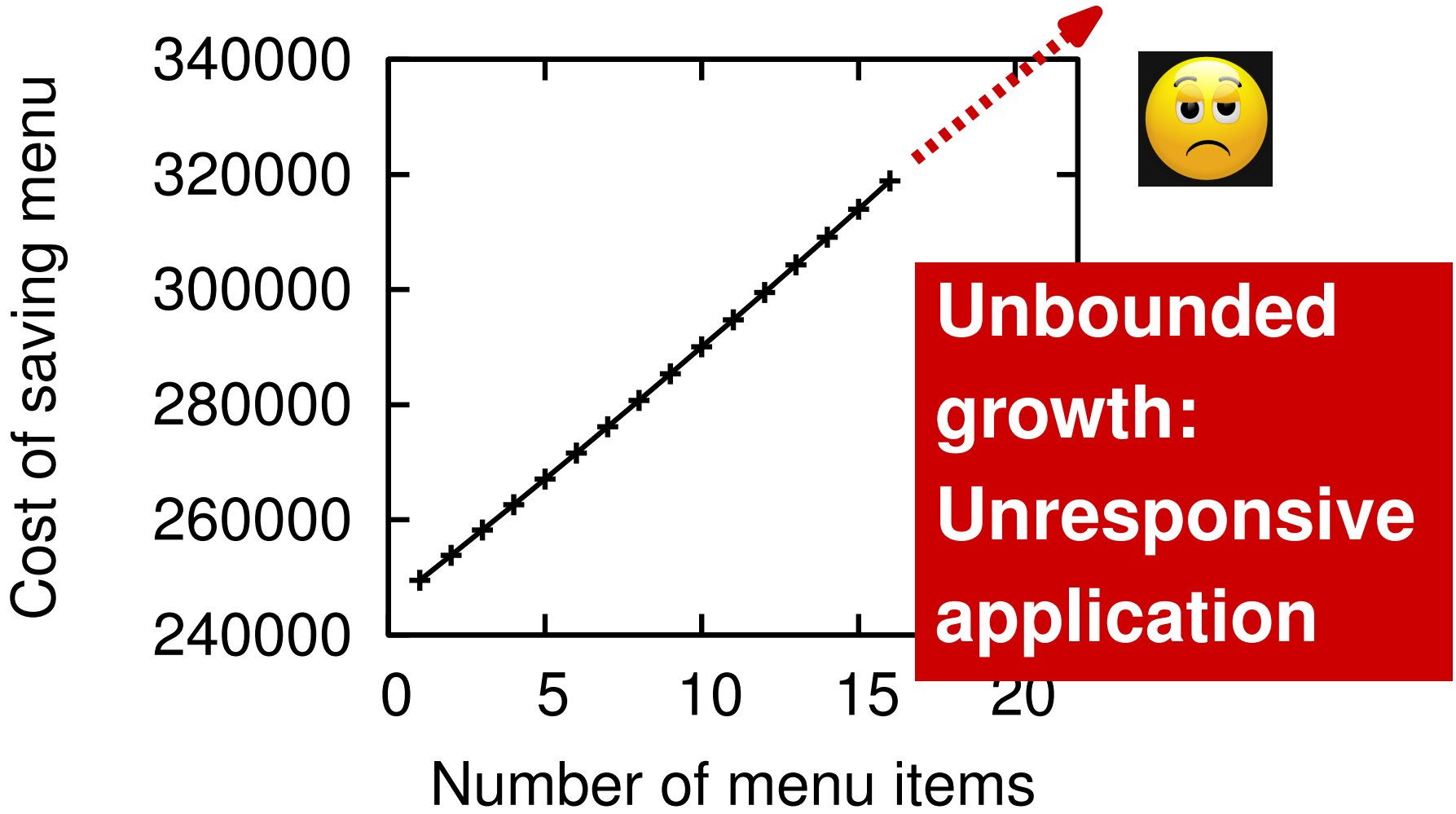
Cost plot for responsiveness problem



Joomla bug 30274

Real-World Example

Cost plot for responsiveness problem



Joomla bug 30274

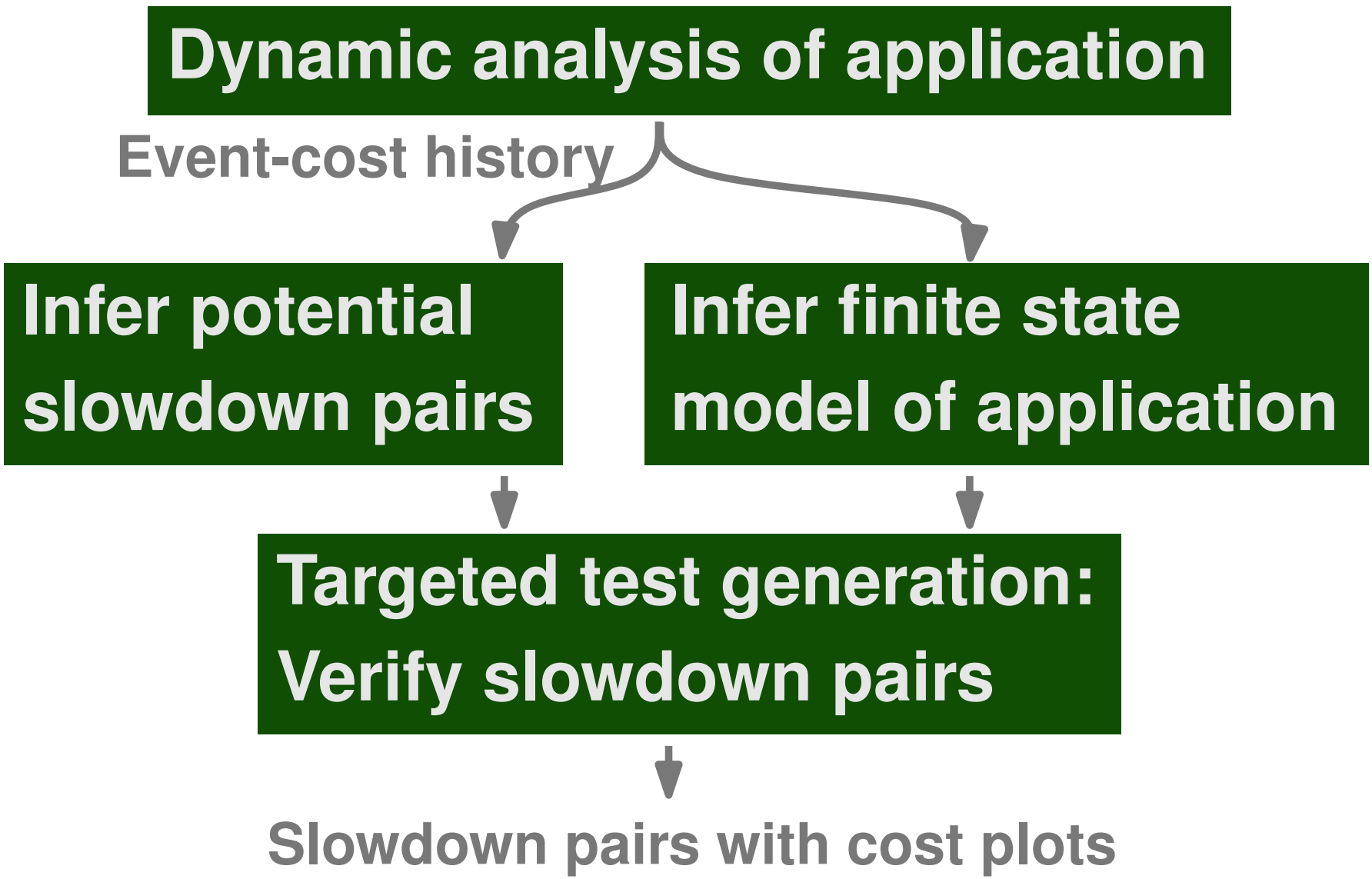
EventBreak: Idea

Analyze responsiveness of web applications through automated testing

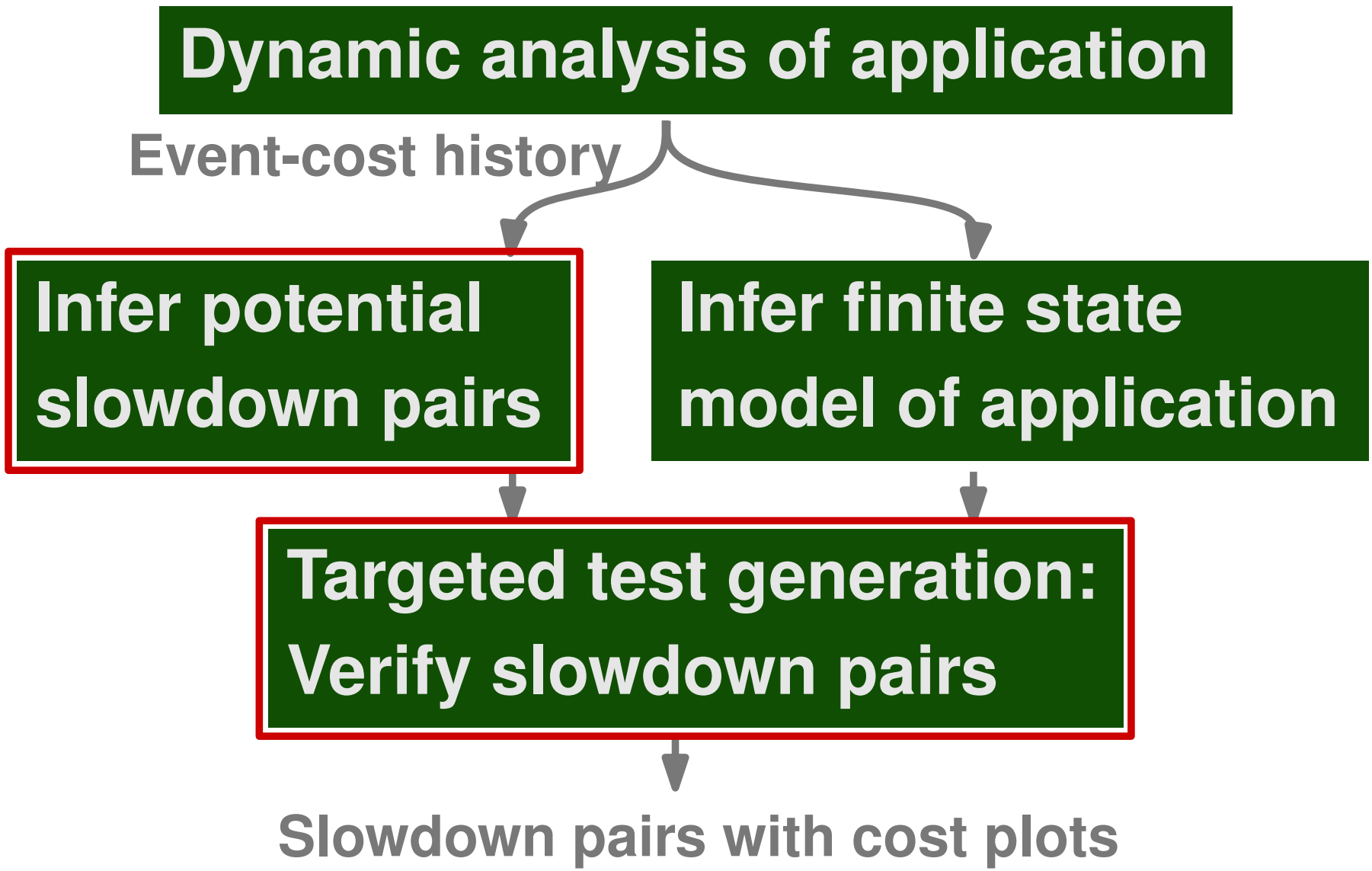
Focus: Slowdown pairs

Event E_{cause} increases cost of event E_{effect}

Overview


Dynamic analysis of application
Event-cost history
Infer potential slowdown pairs
Infer finite state model of application
Targeted test generation: Verify slowdown pairs
Slowdown pairs with cost plots

Overview


Dynamic analysis of application
Event-cost history
Infer potential slowdown pairs
Infer finite state model of application
Targeted test generation: Verify slowdown pairs
Slowdown pairs with cost plots

Potential Slowdown Pairs

Does A increase cost of B?

Event	Cost
B	5
A	3
B	10
B	12
A	3
B	12

Supporting evidence S

Refuting evidence R

$$Supp = |S| = 1 \qquad Conf = \frac{|S|}{|S| + |R|} = 33\%$$

Targeted Test Generation

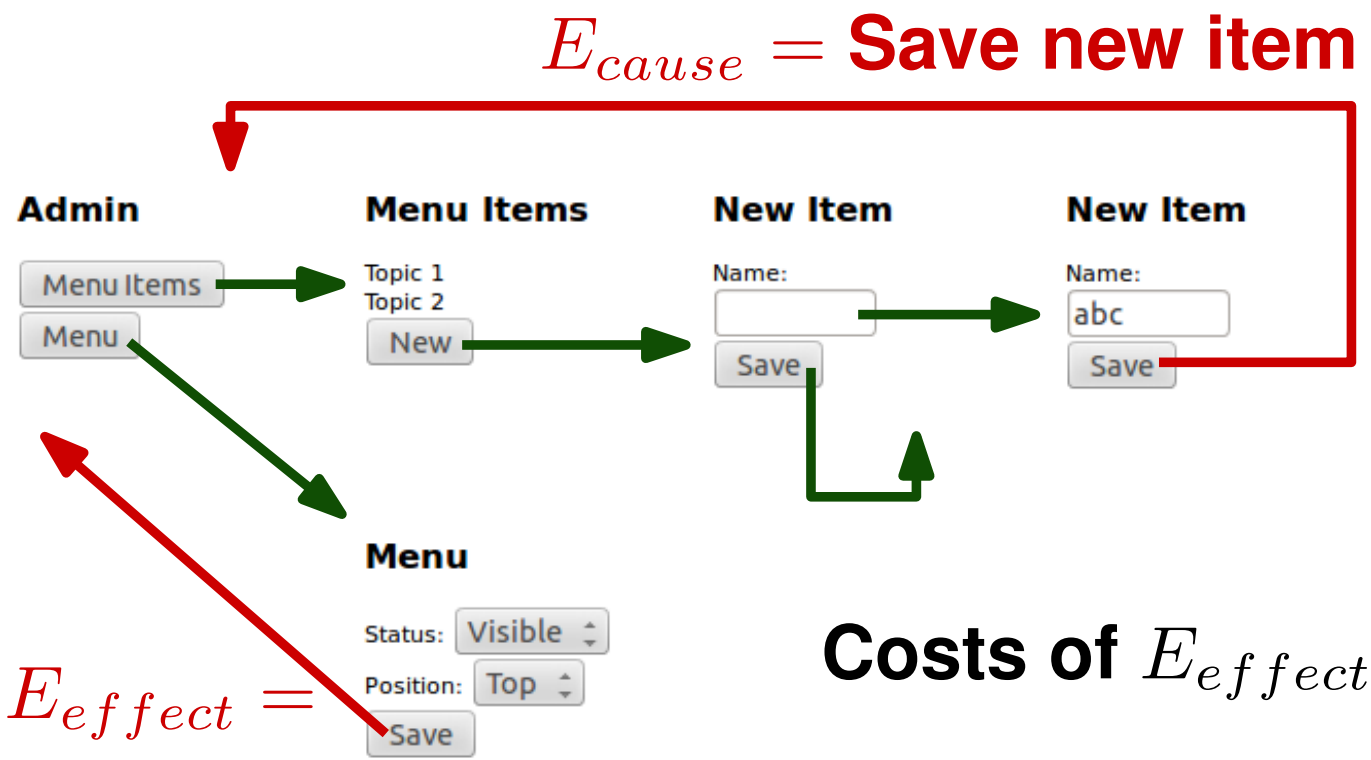
Confirm or reject slowdown pair:

Alternate between E_{effect} and E_{cause}

Targeted Test Generation

Confirm or reject slowdown pair:

Alternate between E_{effect} and E_{cause}



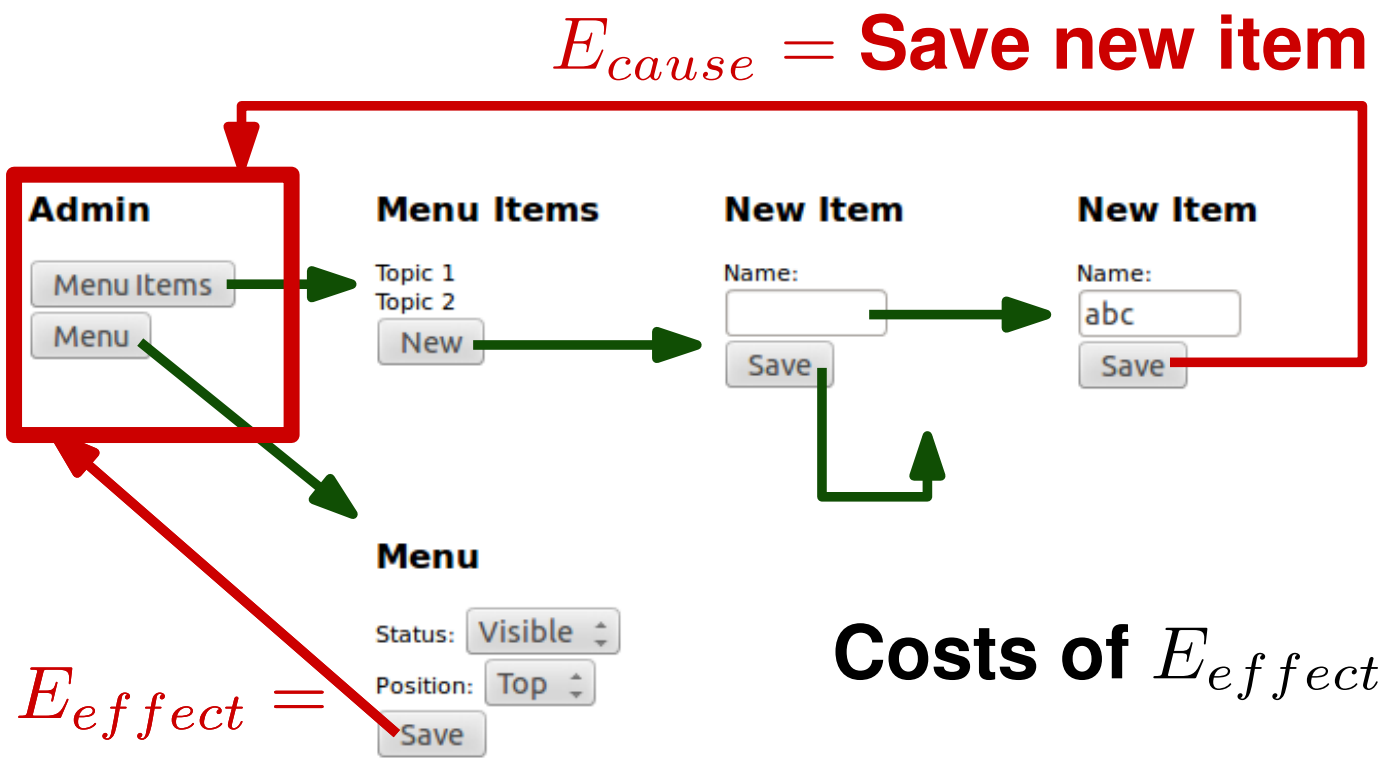
Costs of E_{effect}:

current state → target event

Targeted Test Generation

Confirm or reject slowdown pair:

Alternate between E_{effect} and E_{cause}



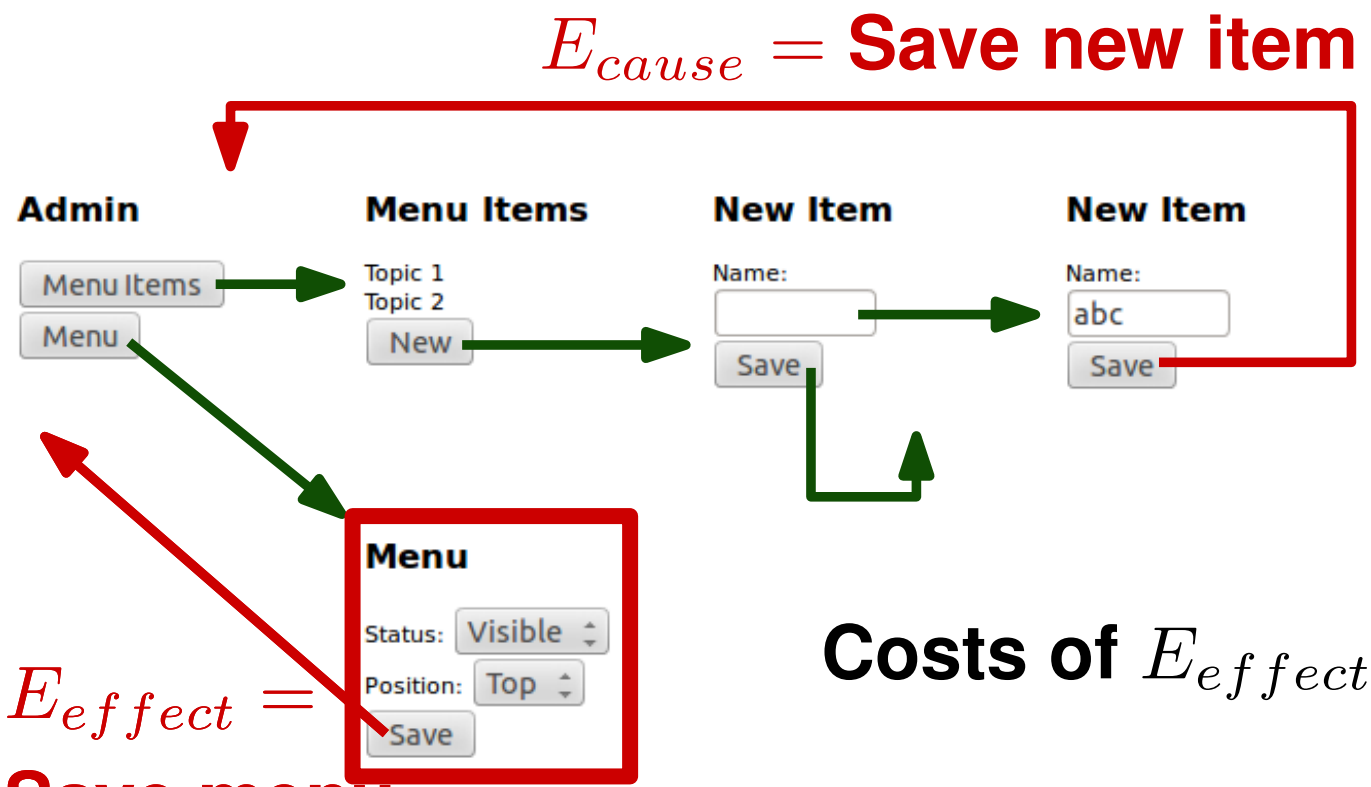
Costs of E_{effect}:

current state → target event

Targeted Test Generation

Confirm or reject slowdown pair:

Alternate between E_{effect} and E_{cause}



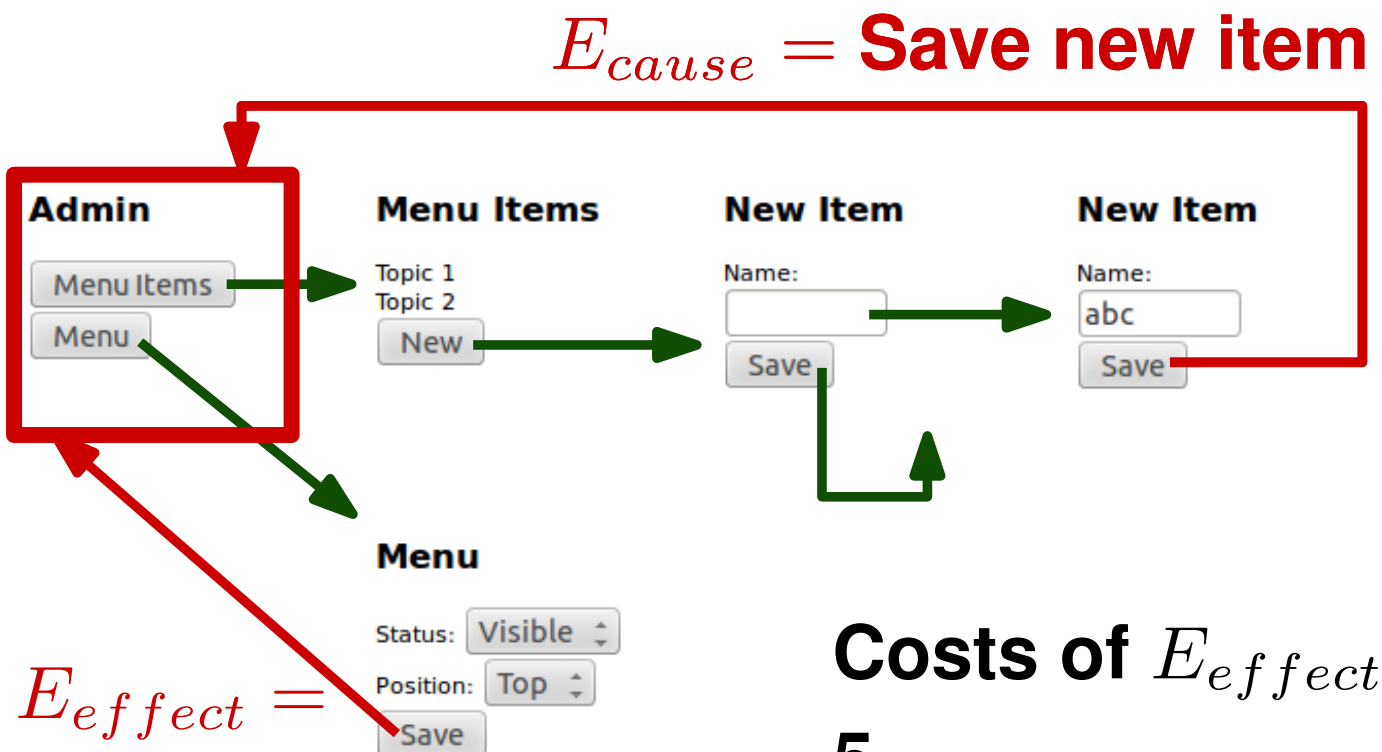
Costs of E_{effect}:

current state → target event

Targeted Test Generation

Confirm or reject slowdown pair:

Alternate between E_{effect} and E_{cause}



Costs of E_{effect}:

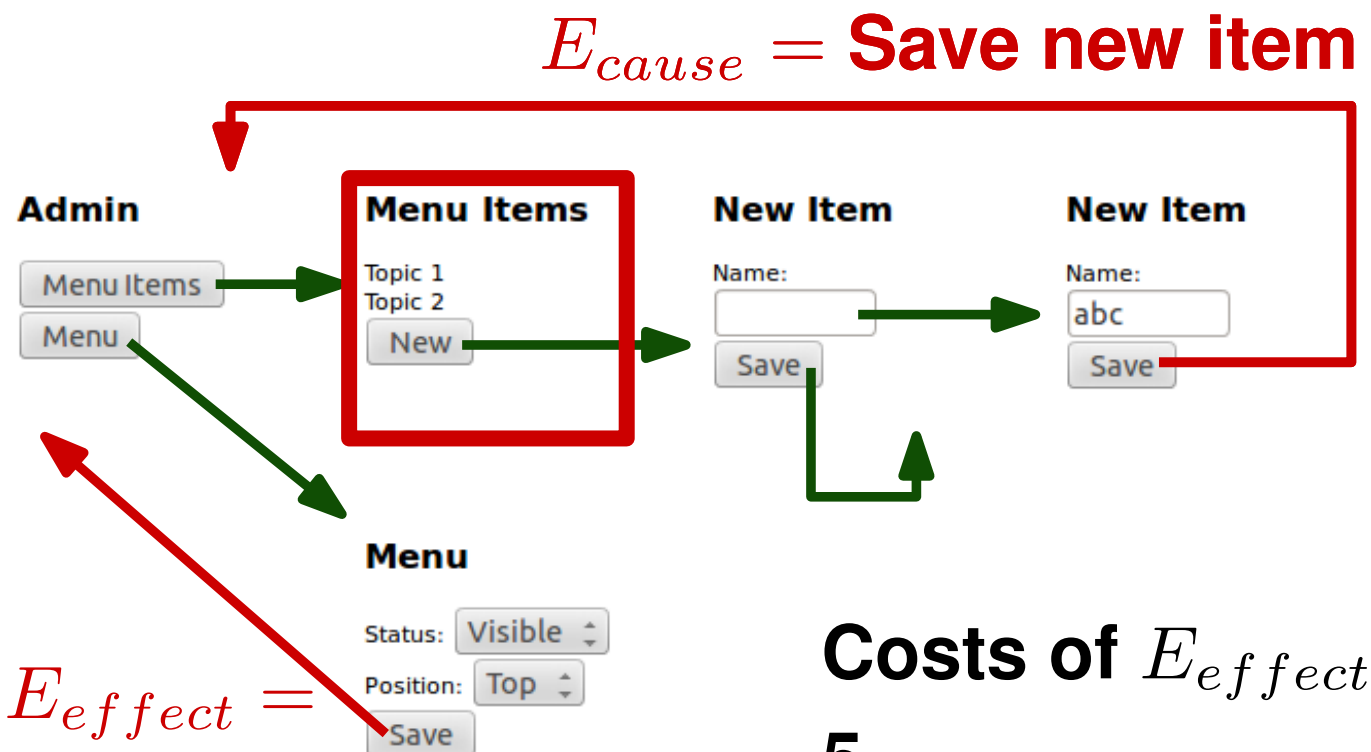
5

current state → target event

Targeted Test Generation

Confirm or reject slowdown pair:

Alternate between E_{effect} and E_{cause}



Costs of E_{effect}:

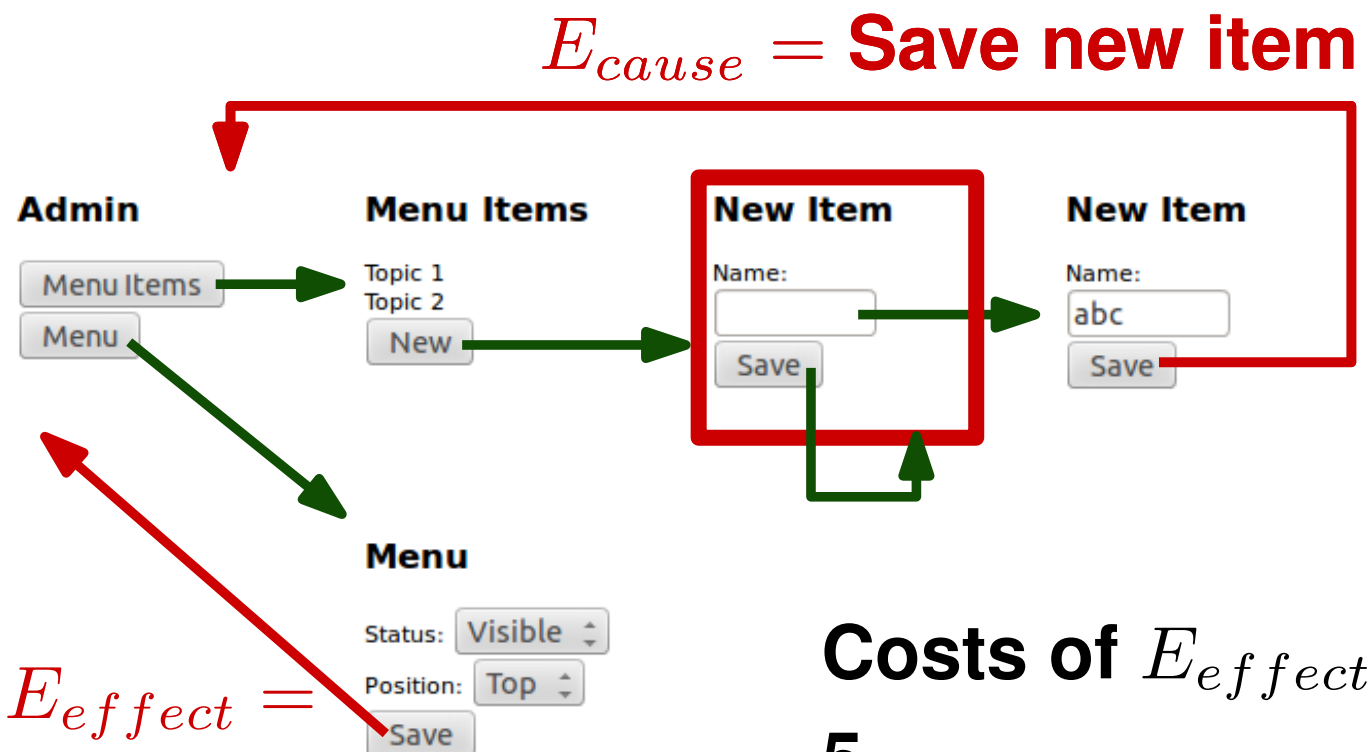
5

current state → target event

Targeted Test Generation

Confirm or reject slowdown pair:

Alternate between E_{effect} and E_{cause}



Costs of E_{effect}:

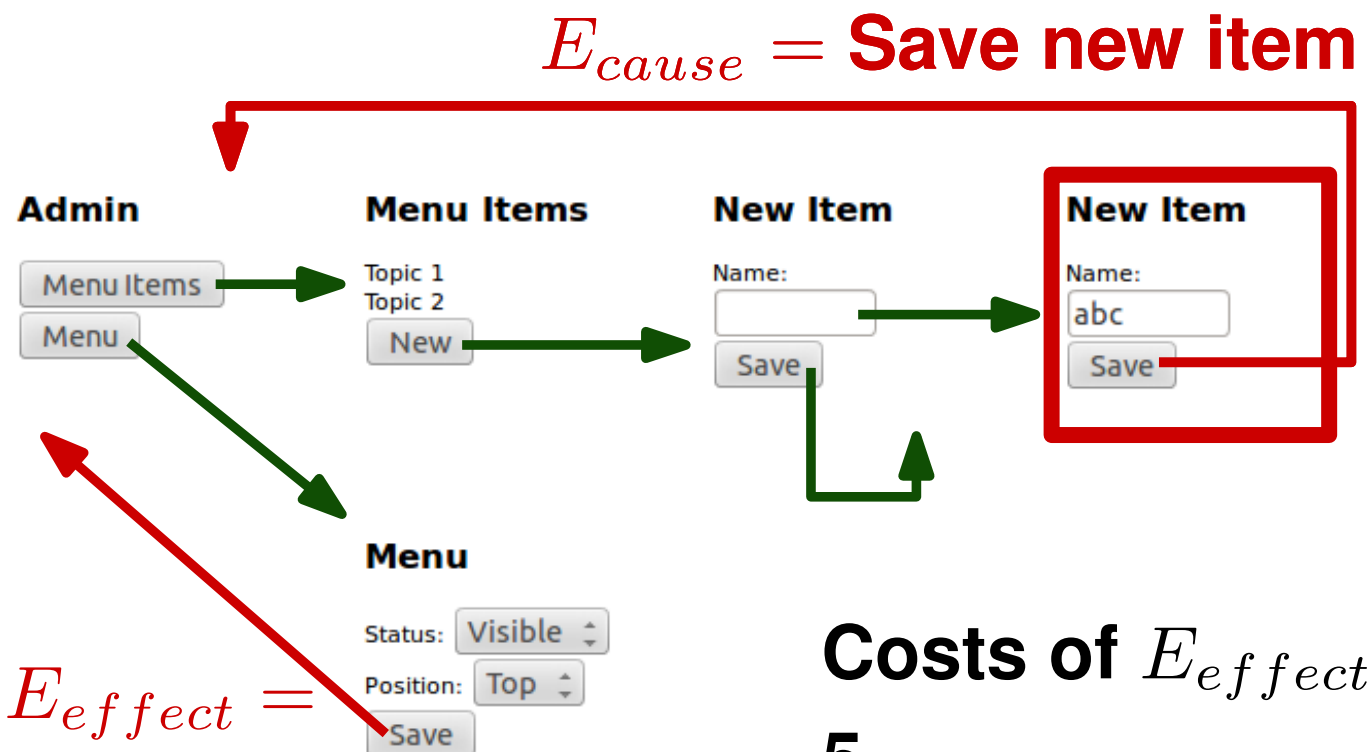
5

current state → target event

Targeted Test Generation

Confirm or reject slowdown pair:

Alternate between E_{effect} and E_{cause}



Costs of E_{effect}:

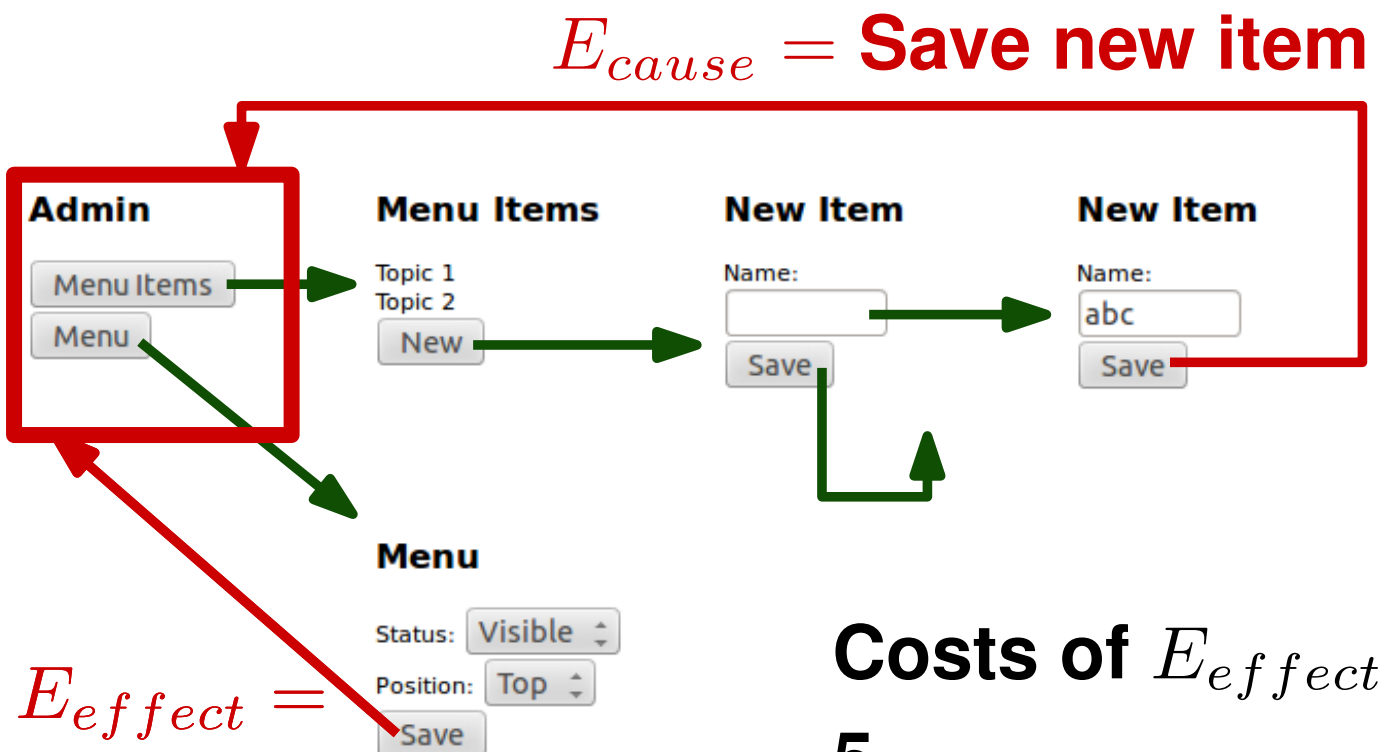
5

current state → target event

Targeted Test Generation

Confirm or reject slowdown pair:

Alternate between E_{effect} and E_{cause}



Costs of E_{effect}:

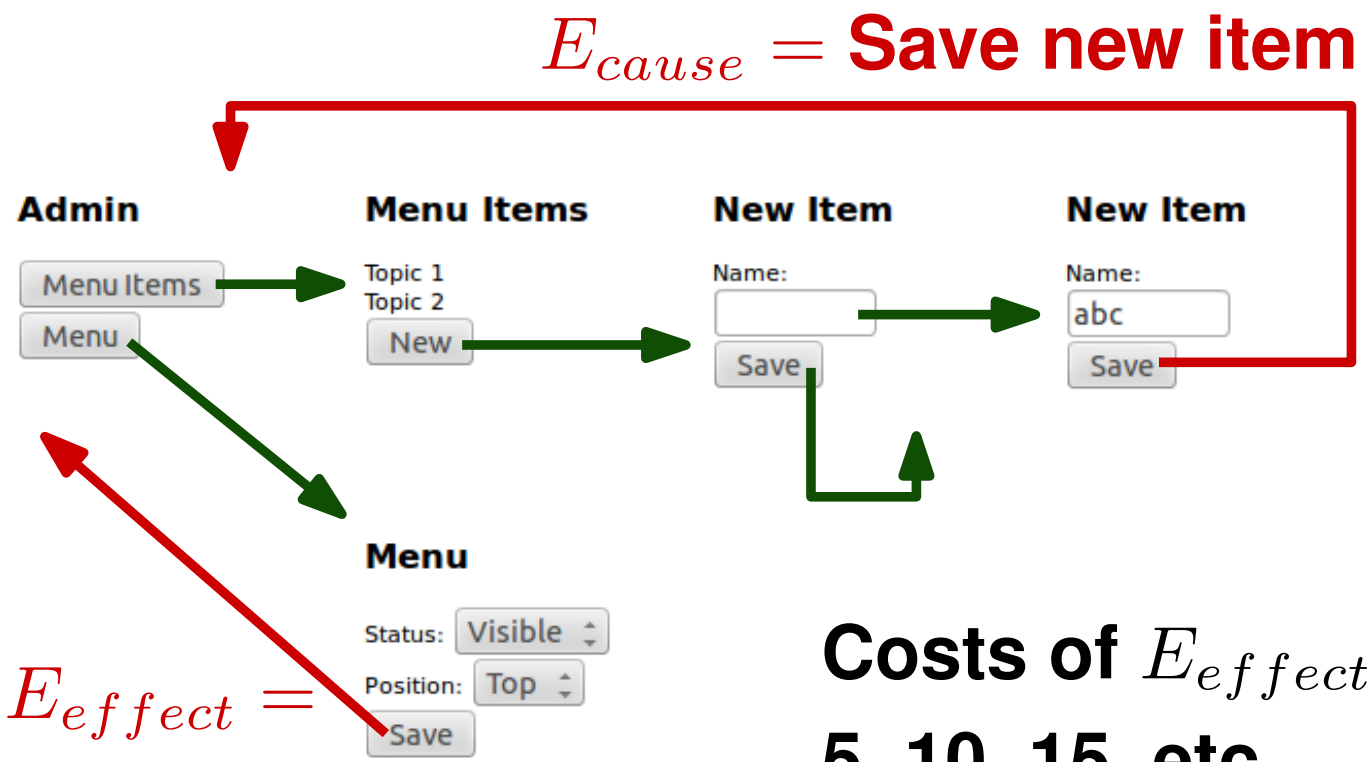
5

current state → target event

Targeted Test Generation

Confirm or reject slowdown pair:

Alternate between E_{effect} and E_{cause}



Costs of E_{effect}:

5, 10, 15, etc.

current state → target event

Summary: GUI Testing

- Automated system-level testing
- Black-box and white-box approaches to explore huge search space
  - Artemis: Whitebox
  - SwiftHand and EventBreak: Mostly blackbox
- Different test oracles possible
  - Application crashes (robustness testing)
  - Consistency criterion, e.g., HTML validation
  - Responsiveness (performance testing)