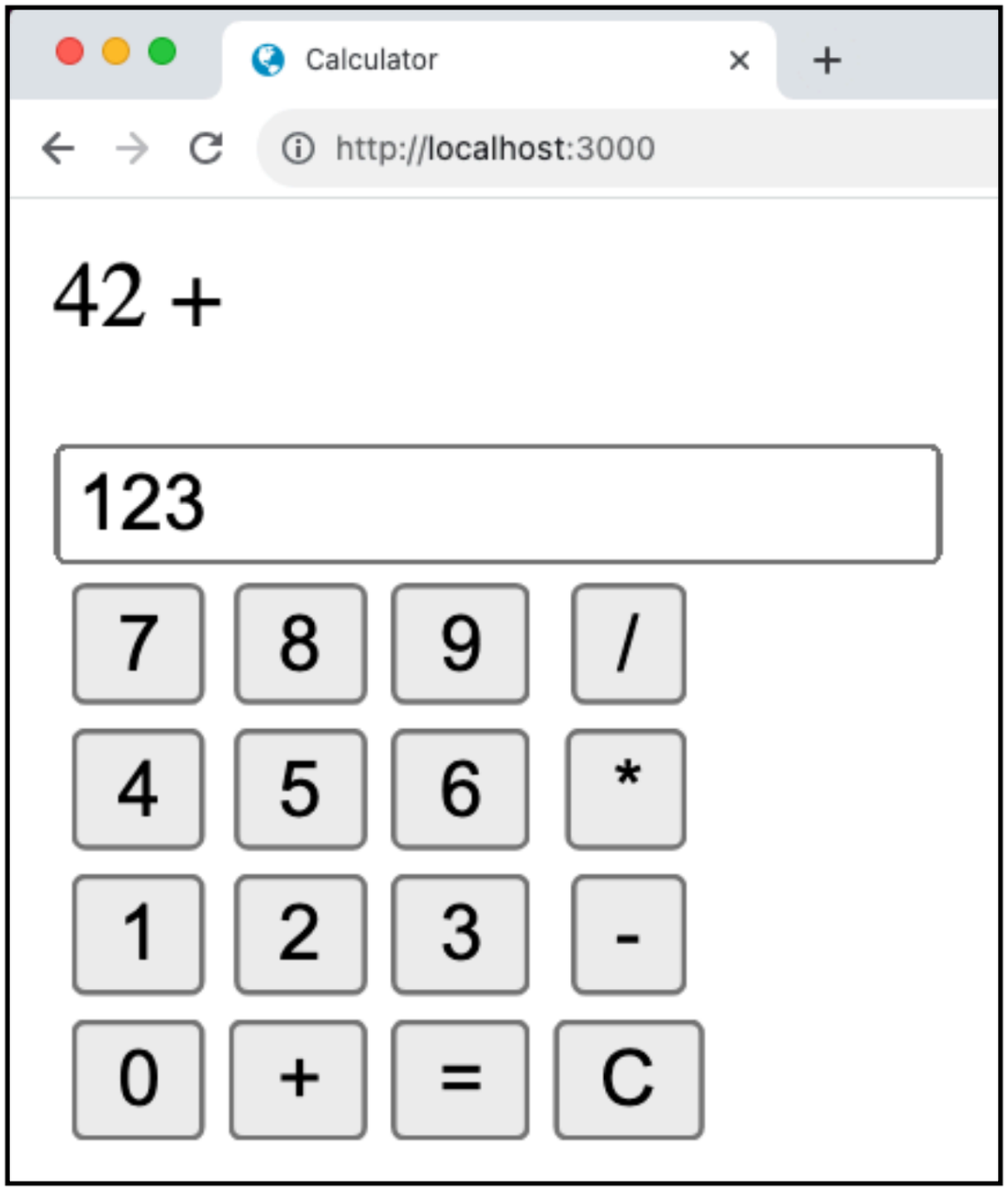
# State Merging for Concolic Testing of Event-driven Applications

Maarten Vandercammen, Coen De Roover

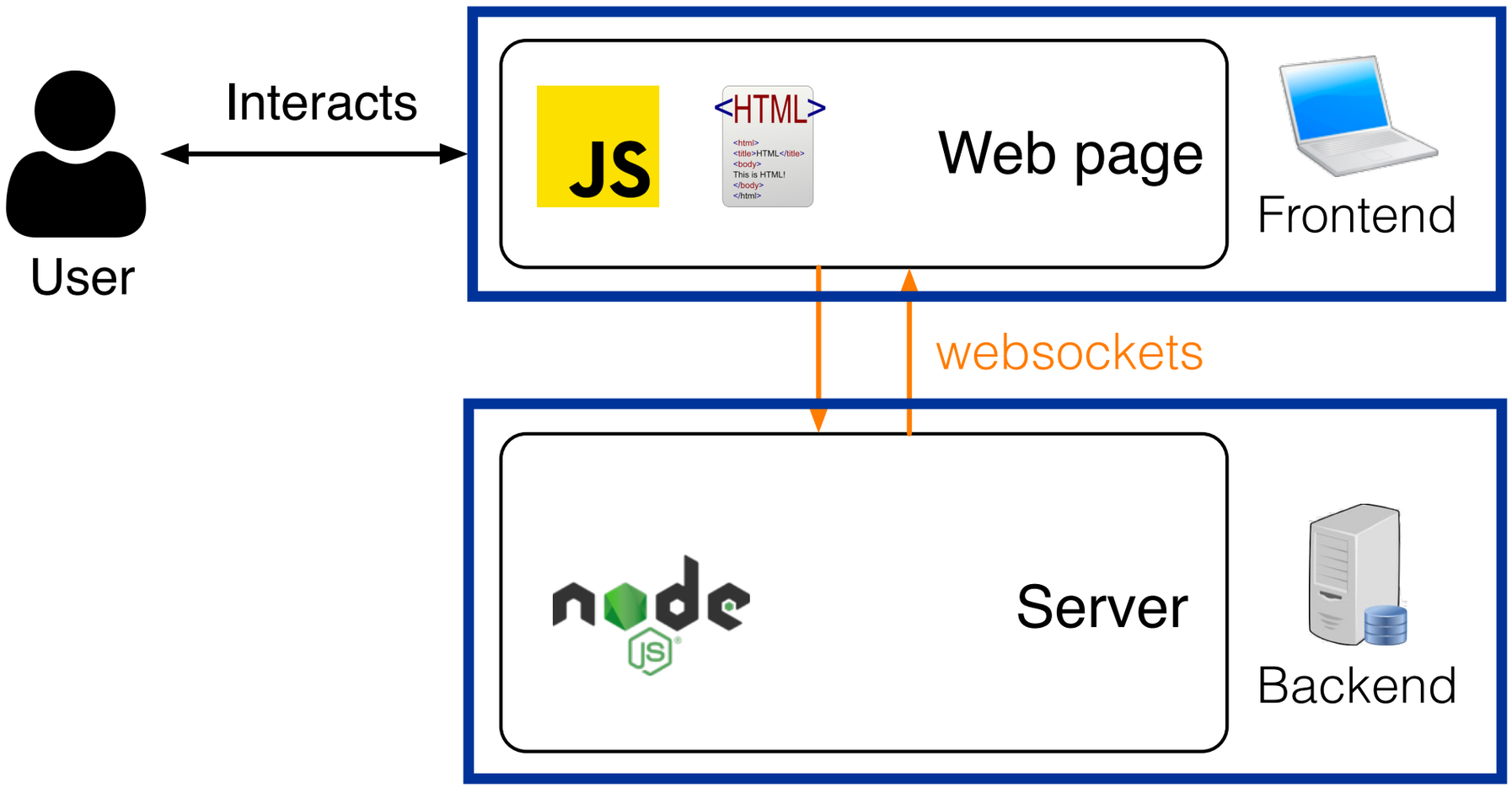
VUB SOFTWARE LANGUAGES LAB

# Full-stack JS Web Applications

Concolic tester **StackFul** for full-stack JS web applications



# Full-stack JS Web Applications




Full-stack JavaScript web applications are web applications consisting of:

- **components implemented in JavaScript**
- **technologies accessed via JavaScript**

# Challenges in Testing Full-stack Web Applications

1. Dynamic nature of JavaScript
2. Handling different process compositions
3. Communication between processes
4. **Event-driven code**

https://github.com/softwarelanguageslab/StackFul


Inter-process Concolic Testing of Full-stack JavaScript Web Applications
https://soft.vub.ac.be/Publications/2023/vub-soft-phd-13-10.pdf

# Testing Event-driven Code

1. Allow for dynamic (de)registration of event handlers

```
io.on("connection", function(socket) {
   socket.on("compute", function (input) {
```

```
if (someCondition) {
  button.addEventListener("click", (evt) => doSomething());
}
```

**2. Reduce event space**

```
Click "button 1"
Click "button /"
Click "button 0"
Click "button ="
```

```
Click "button 1"
Click "button 2"
Click "button 3"
Click "button 4"
Click "button 5"
Click "button 6"
Click "button 7"
...
```

# State Merging Overview

- Reduce event space by **merging** "similar" states

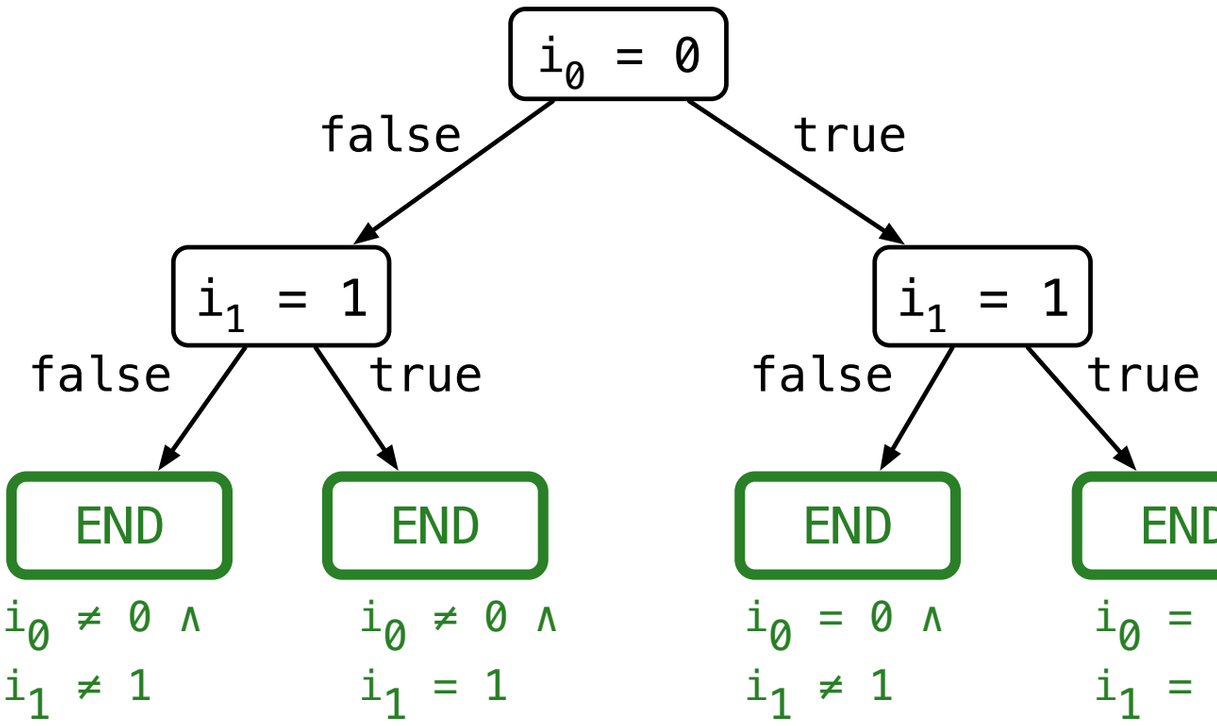
- Described for **sequential** applications by:
  - Kuznetsov *et al.* (2012)
  - Copeland (2014)
  - Sen *et al.* (2015)
  - Avgerinos *et al.* (2016)
  - Zhang *et al.* (2018)
  - and others

- Transpose state merging to **concolic testing** of **event-driven** applications

# State Merging for Sequential Programs

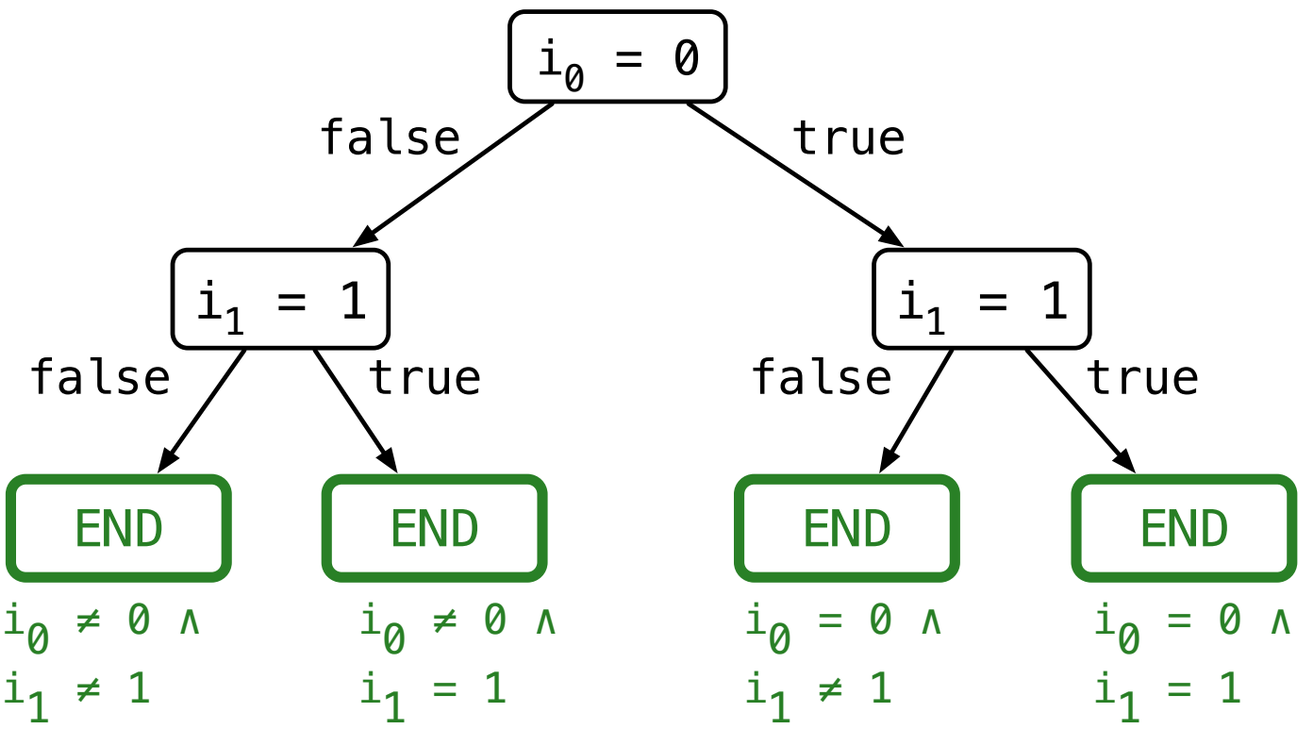
# State Explosion

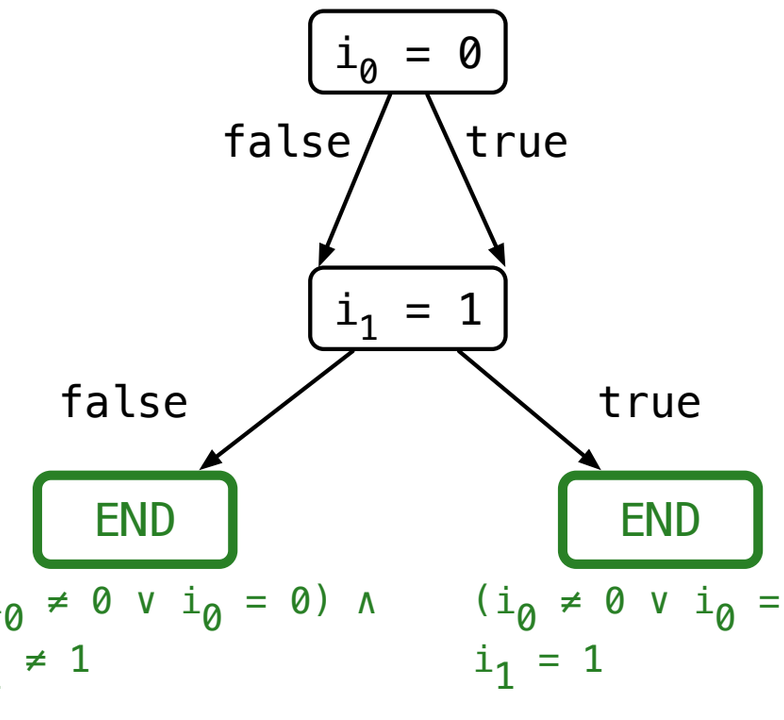
```
let a, b;
if (randomInt() === 0) {
  a = 1;
} else {
  a = 0;
}
if (randomInt() === 1) {
  b = 1;
} else {
  b = 0;
}
```




Symbolic execution tree with 4 **end states**

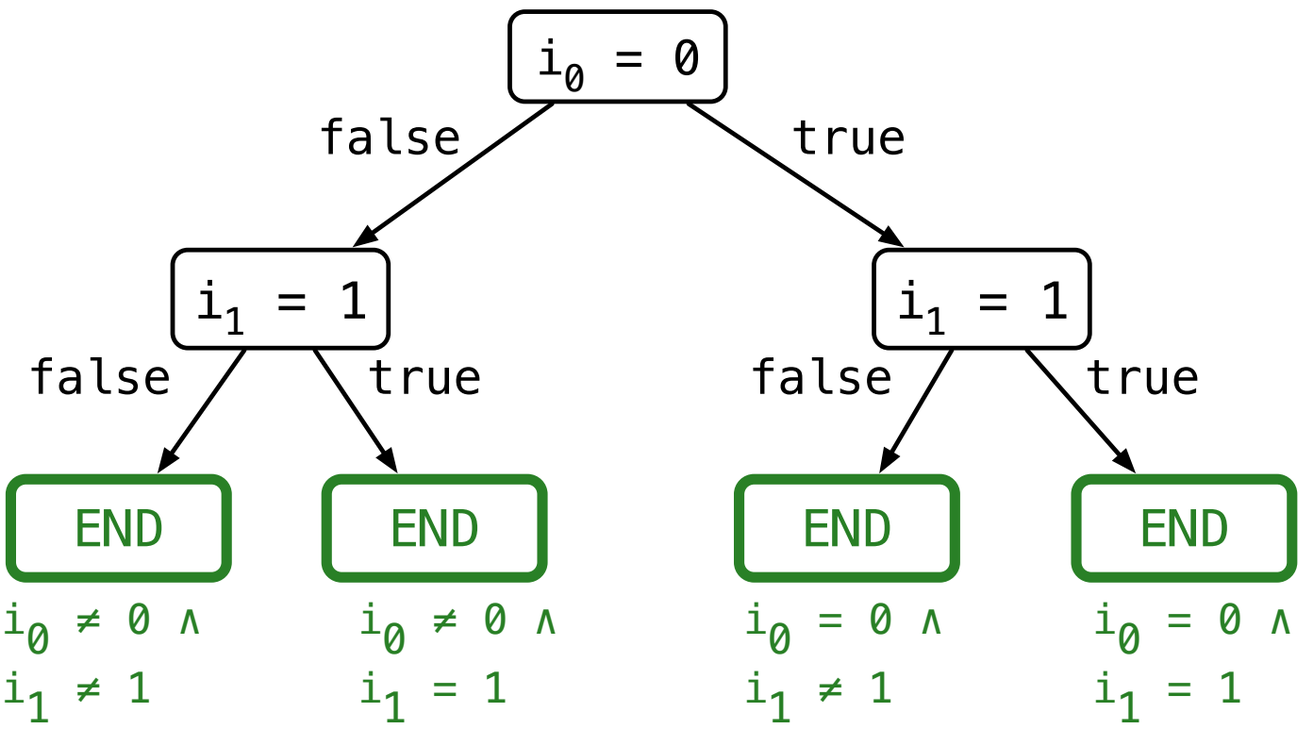
# State Merging

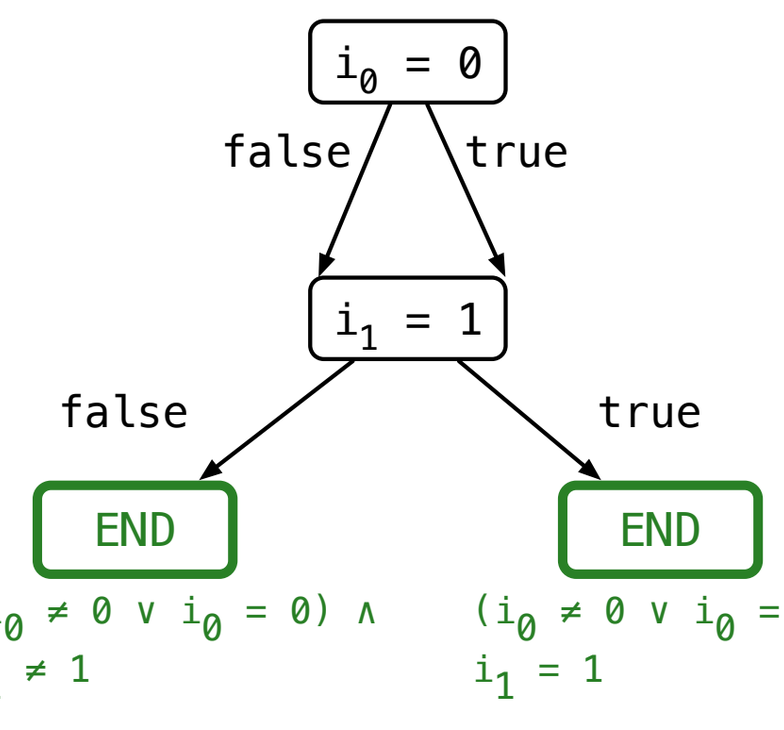



Symbolic execution **Directed Acyclic Graph (DAG)** with 2 **end states**
**Requires fewer test iterations**

# State Merging




Symbolic execution **Directed Acyclic Graph (DAG)** with 2 **end states**
Use `If-Then-Else` symbolic expressions to maintain precision

```
if (randomInt() === 0) {
  a = 1;
} else {
  a = 0;
}
```

$a \rightarrow ITE(i_0 = 0, 1, 0)$

# State Merging Formally

Define tree nodes as state triples:
⟨**Program point**, **Path constraint**, **Store**⟩

**Program point**
Defines current point in execution of program
(more generic definition increases # states merged)

**Path constraint**
Specifies path to node

**Store**
Maps program variables to symbolic expressions

# State Merging Formally

State merging operation ~ defined on two states

$\langle P, PC_1, \sigma_1 \rangle \sim \langle P, PC_2, \sigma_2 \rangle = \langle P, PC_1 \vee PC_2, \sigma_m \rangle$

# State Merging Formally

State merging operation ~ defined on two states

$\langle P, PC_1, \sigma_1 \rangle \sim \langle P, PC_2, \sigma_2 \rangle = \langle P, PC_1 \vee PC_2, \sigma_m \rangle$

**Program point P**

Only states at the same point in execution are merged

# State Merging Formally

State merging operation ~ defined on two states

$\langle P, PC_1, \sigma_1 \rangle \sim \langle P, PC_2, \sigma_2 \rangle = \langle P, PC_1 \vee PC_2, \sigma_m \rangle$

**Program point** $P$

Only states at the same point in execution are merged

**Merged path constraint** $PC_1 \vee PC_2$

Merged state reachable via either $PC_1$ or $PC_2$

# State Merging Formally

State merging operation ~ defined on two states

$\langle P, PC_1, \sigma_1 \rangle \sim \langle P, PC_2, \sigma_2 \rangle = \langle P, PC_1 \vee PC_2, \sigma_m \rangle$

**Program point** $P$

Only states at the same point in execution are merged

**Merged path constraint** $PC_1 \vee PC_2$

Merged state reachable via either $PC_1$ or $PC_2$

**Merged store** $\sigma_m$

Defined as $\forall v \in \sigma_1 : \sigma_m[v] = ITE(PC_1, \sigma_1[v], \sigma_2[v])$

# Costs and Benefits

**Advantages**

+ Reduces number of states

# Costs and Benefits

**Advantages**

+ Reduces number of states

**Disadvantages**

- Stresses SMT solver
- Introduces more symbolic variables

Value for program variable

$v \rightarrow 42$

Transformed to

$v \rightarrow ITE(PC3, \sigma3[v], ITE(PC1, \sigma1[v], \sigma2[v]))$

# Costs and Benefits

**Advantages**

+ Reduces number of states

**Disadvantages**

- Stresses SMT solver
- Introduces more symbolic variables

Value for program variable

$v \rightarrow 42$

Transformed to

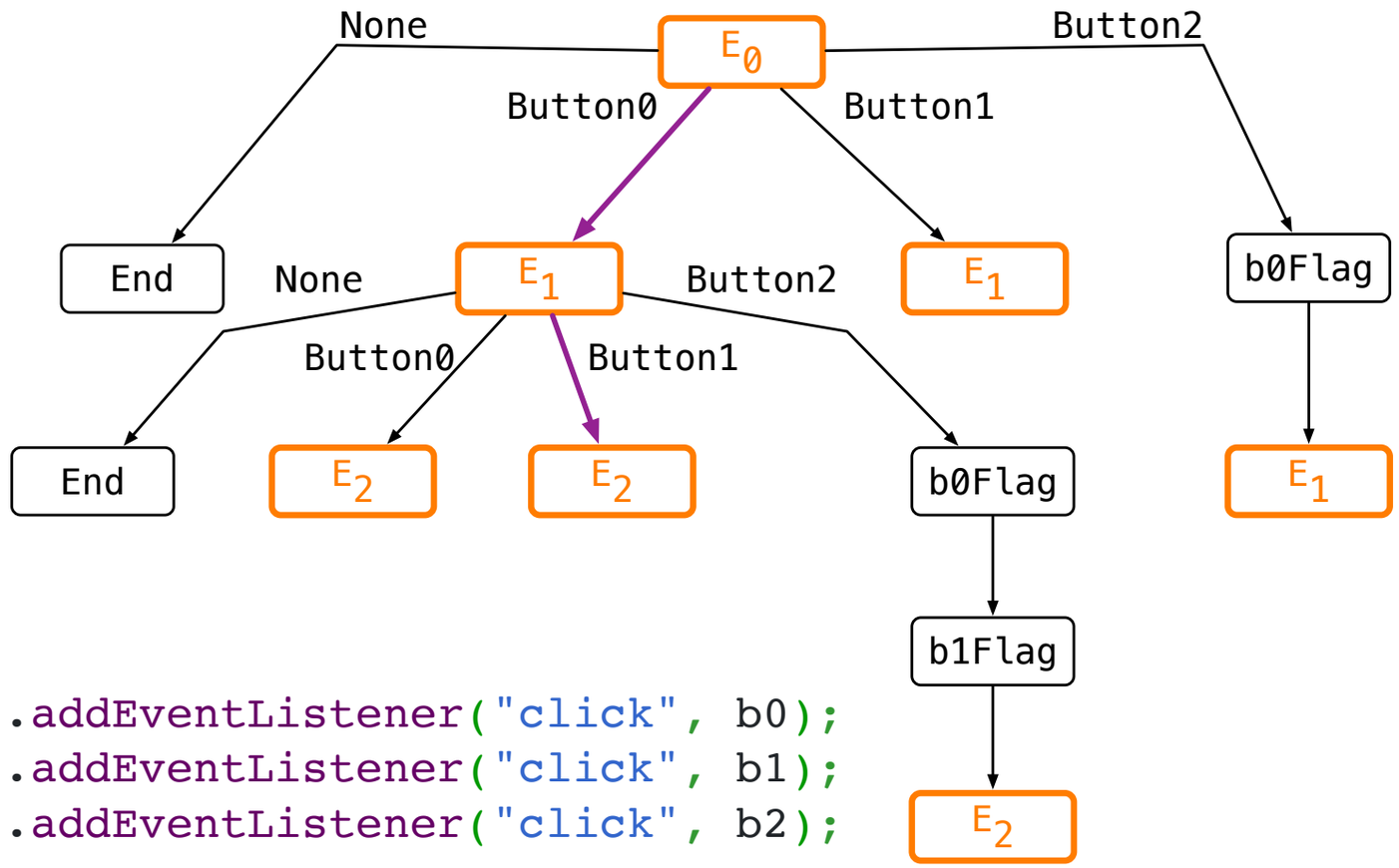
$v \rightarrow ITE(PC3, \sigma3[v], ITE(PC1, \sigma1[v], \sigma2[v]))$

Computational burden **moved** from exploring program paths to SMT solving of constraints

# State Merging for Event-driven Programs

# Event-driven State Explosion

```
let b0Flag = false, b1Flag = false;
function b0() {
  b0Flag = true;
}
function b1() {
  b1Flag = true;
}
function b2() {
  if (b0Flag) {
    if (b1Flag) { ... }
    else { ... }
  } else { ... }
  ...
}
document.getElementById("Button0").addEventListener("click", b0);
document.getElementById("Button1").addEventListener("click", b1);
document.getElementById("Button2").addEventListener("click", b2);
```



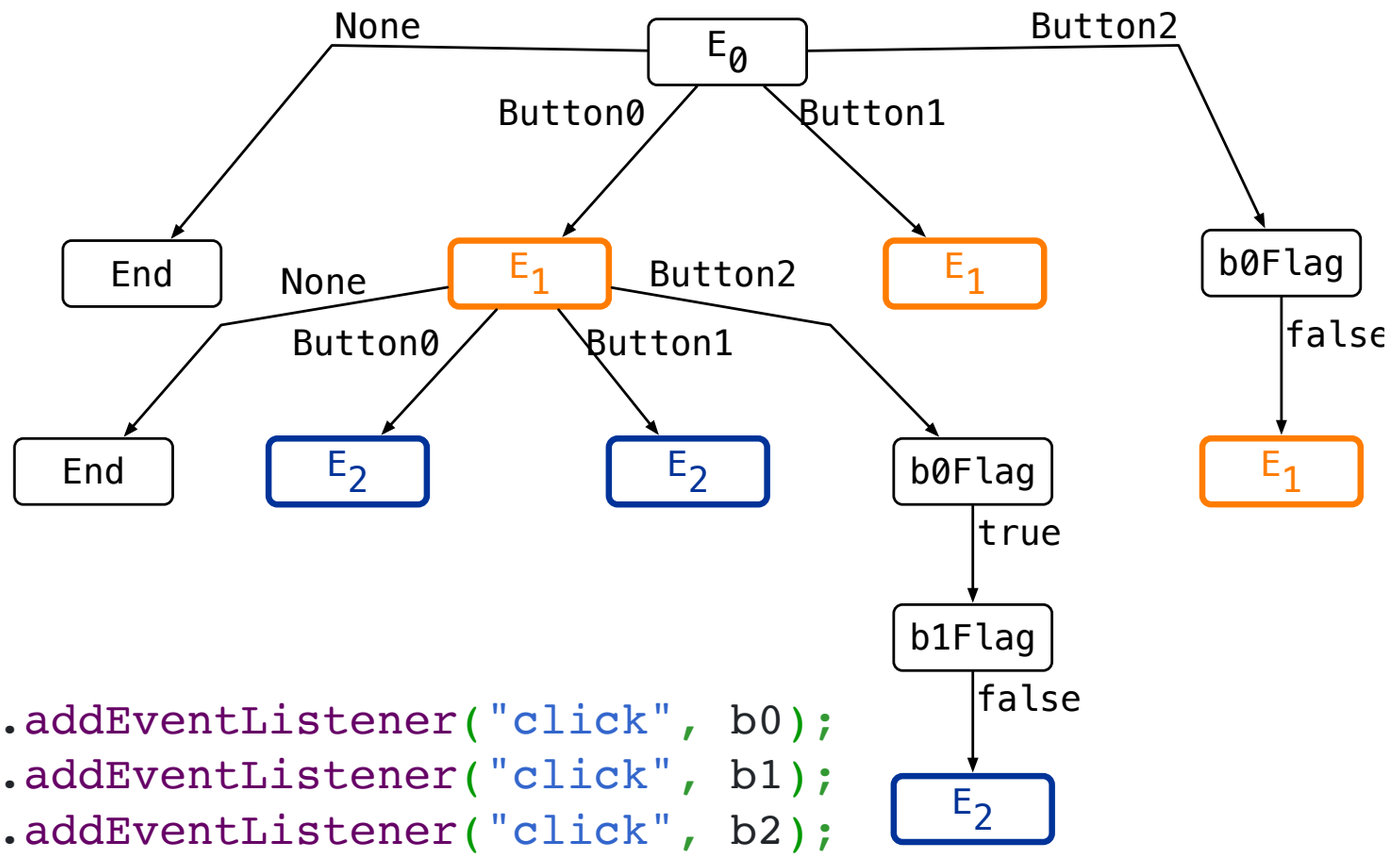

**Event branching nodes** represent choice of event at a point in the event sequence

To explore the **selected path**, StackFul generates:

- `Click` event on `Button0`
- `Click` event on `Button1`

# Event-driven State Explosion

```
let b0Flag = false, b1Flag = false;
function b0() {
  b0Flag = true;
}
function b1() {
  b1Flag = true;
}
function b2() {
  if (b0Flag) {
    if (b1Flag) { ... }
    else { ... }
  } else { ... }
  ...
}
document.getElementById("Button0").addEventListener("click", b0);
document.getElementById("Button1").addEventListener("click", b1);
document.getElementById("Button2").addEventListener("click", b2);
```

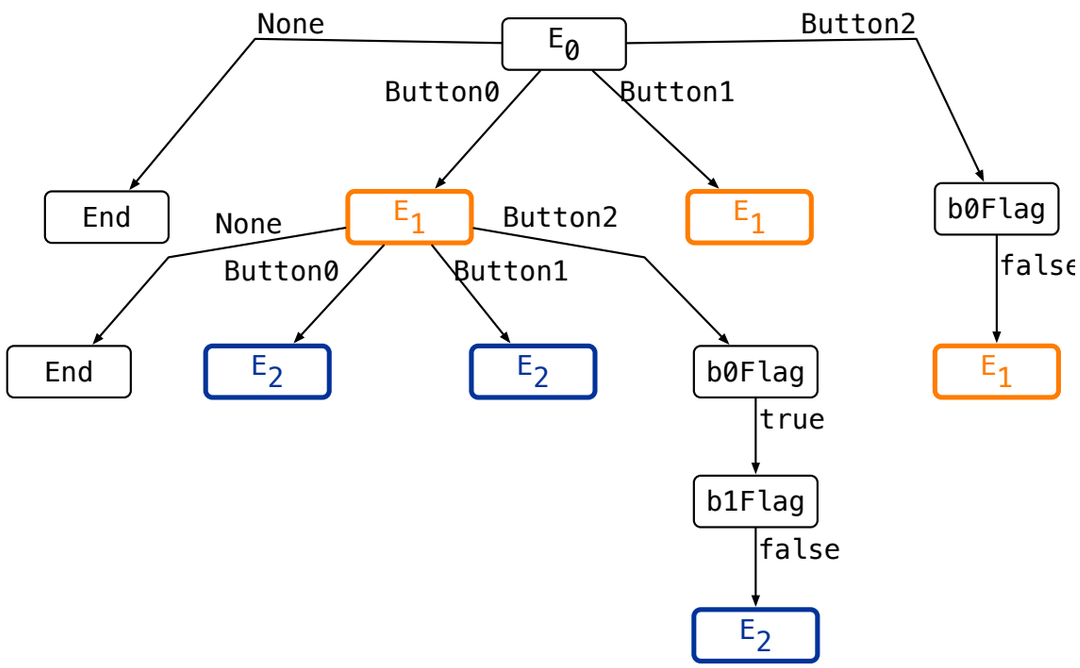



| | | |
|---|---|---|
| Click "Button0"<br>Click "Button0" | Click "Button0"<br>Click "Button1" | Click "Button0"<br>Click "Button2" |
| Click "Button1"<br>Click "Button0" | Click "Button1"<br>Click "Button1" | Click "Button1"<br>Click "Button2" |
| Click "Button2"<br>Click "Button0" | Click "Button2"<br>Click "Button1" | Click "Button2"<br>Click "Button2" |

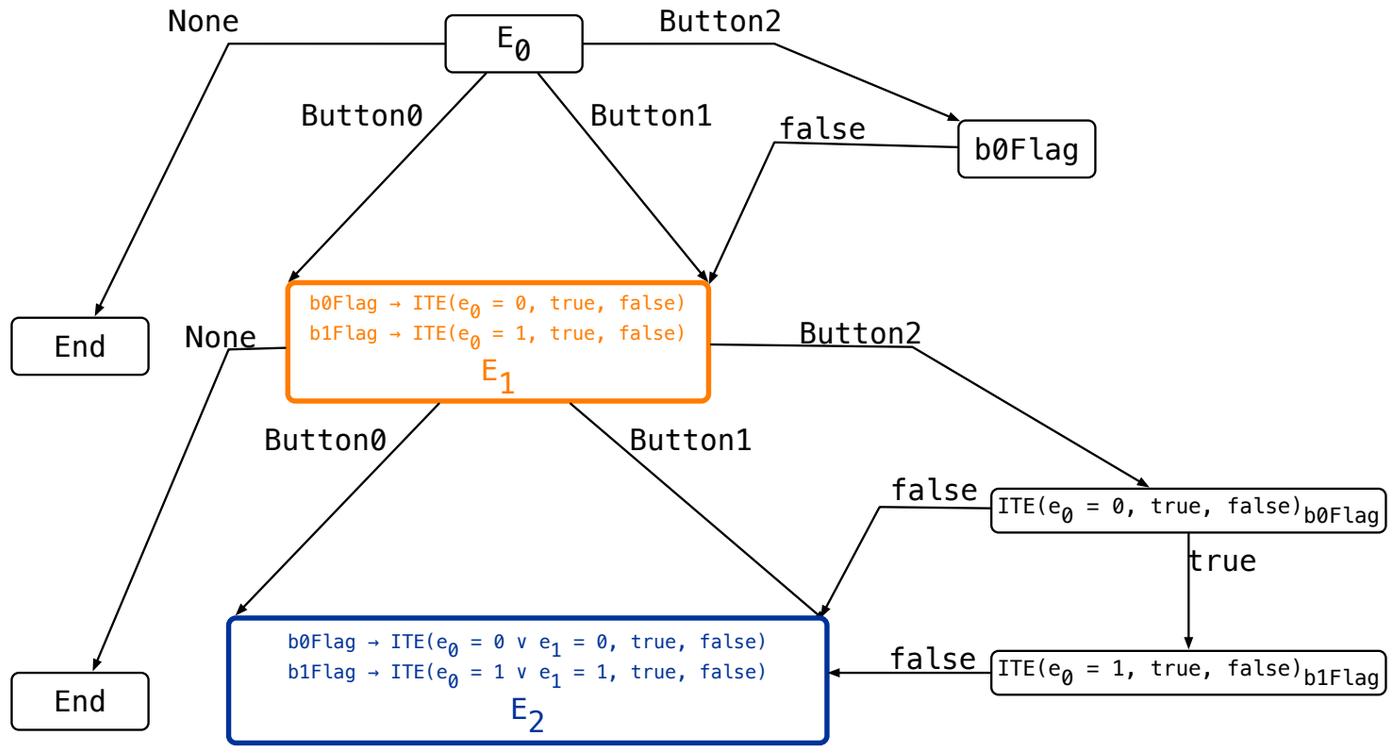
Event branching nodes
**$E_1$** and **$E_2$** duplicated

# Merging Event-driven Programs

## Without state merging




## With state merging




Merge similar event branching nodes (instances of $E_1$ and $E_2$)

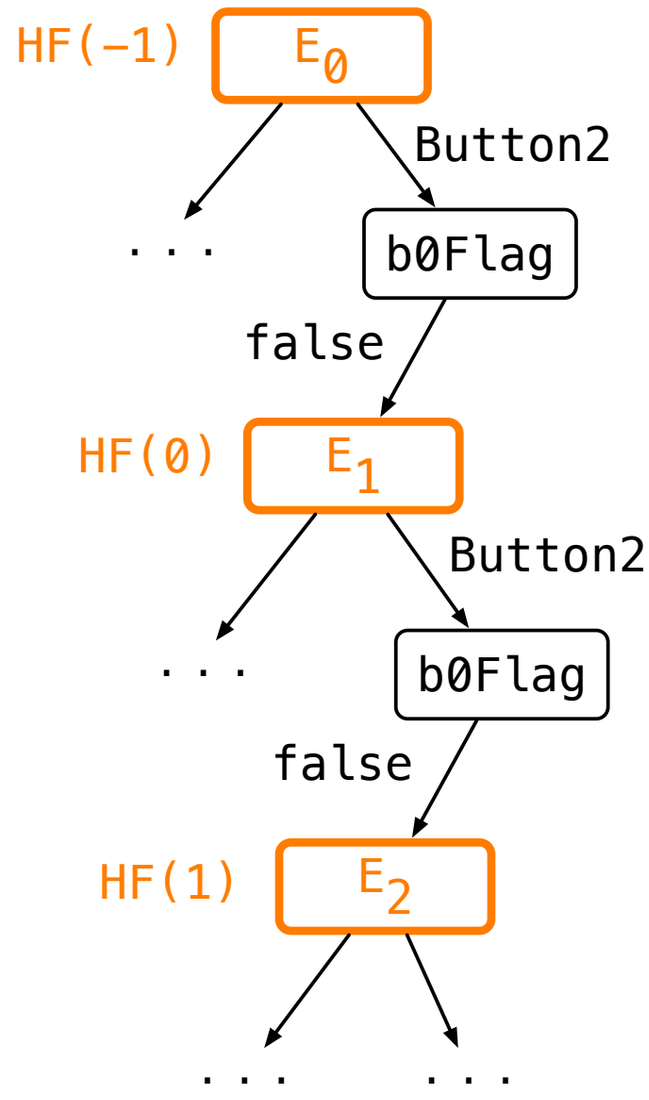
Merging event branching nodes enabled by:

- Redefining program point
- Translating event branching nodes to symbolic expressions

# Challenge 1: Program Points

Program points should identify **event branching nodes**

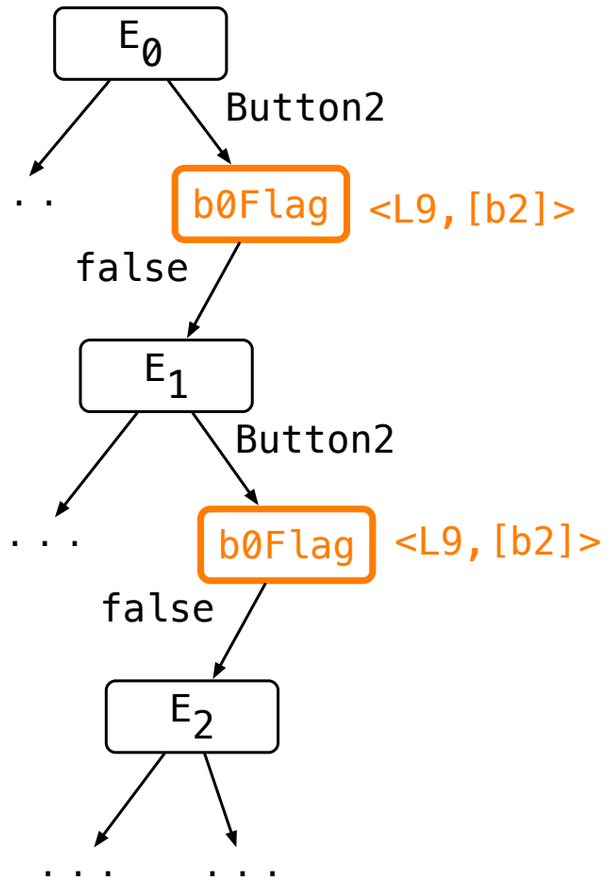
**`HandlerFinished(i)`**
program points for
event branching nodes

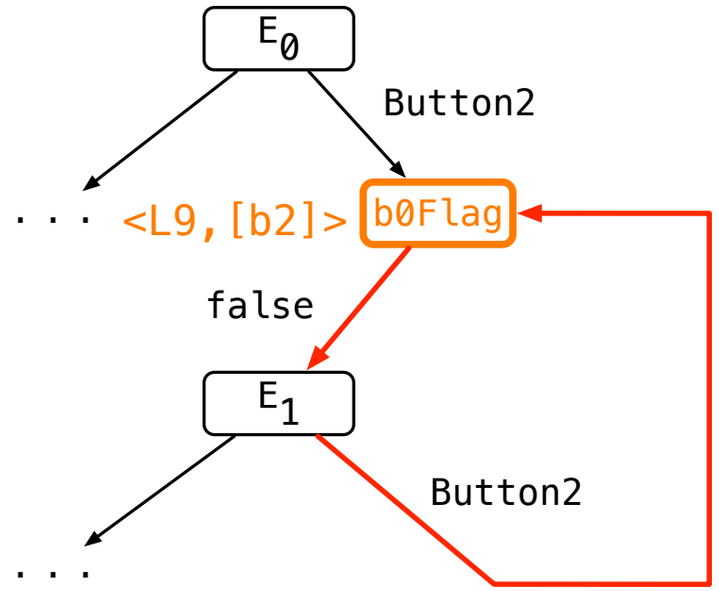


# Challenge 1: Program Points

Program points should prevent **loops** forming in the DAG

```
function b2() {
  if (b0Flag) { ... }
}
```

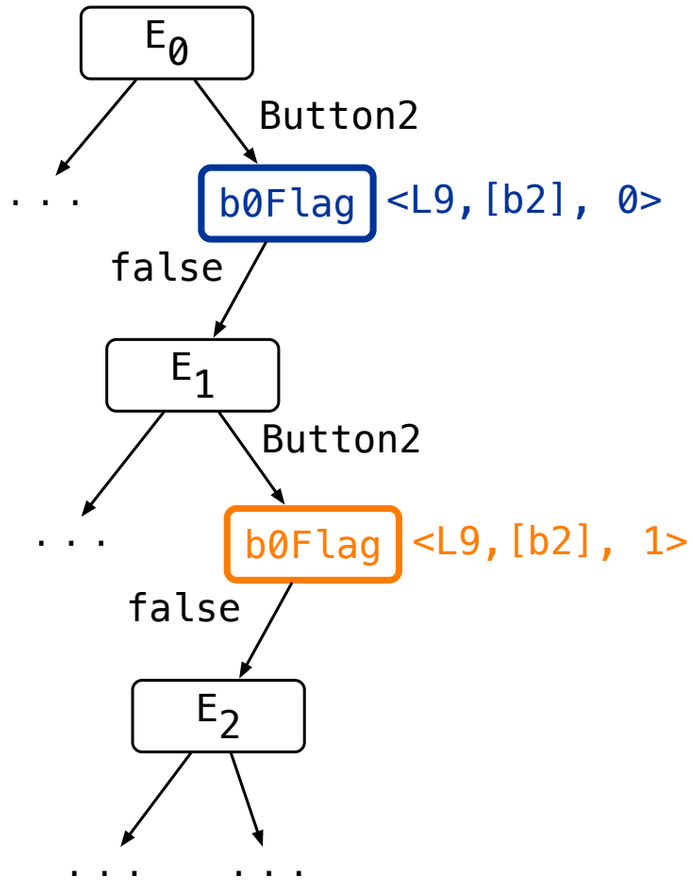



**Path constraint of $E_1$ state indeterminate**

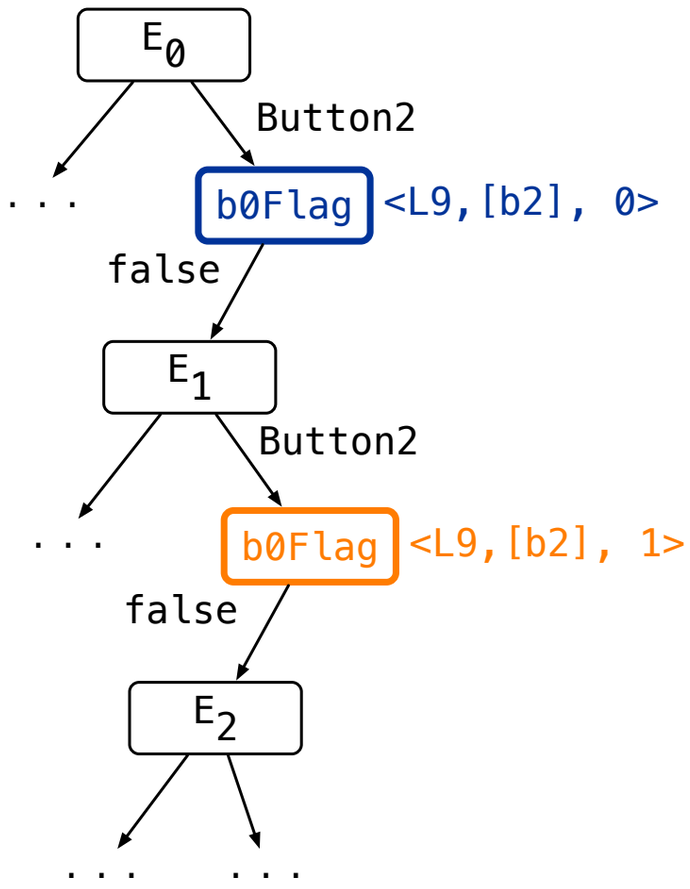
# Challenge 1: Program Points

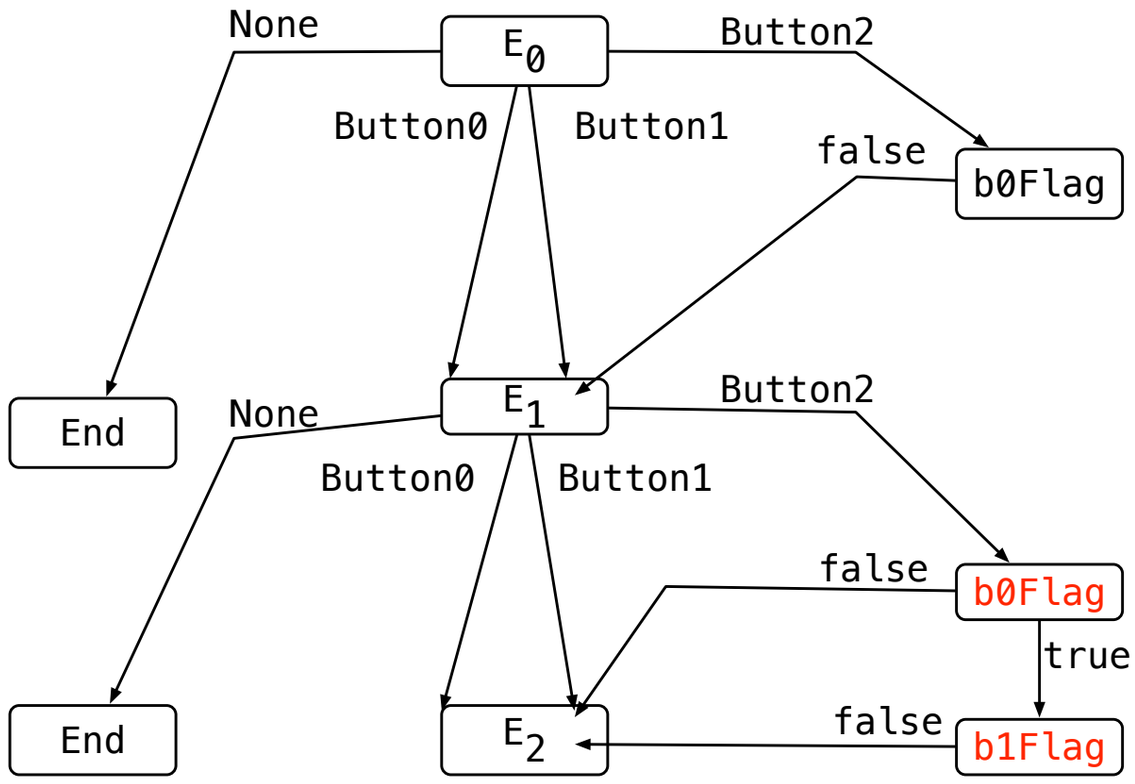
Prevented by including event handler id in program point



# Challenge 1: Program Points

Prevented by including event handler id in program point



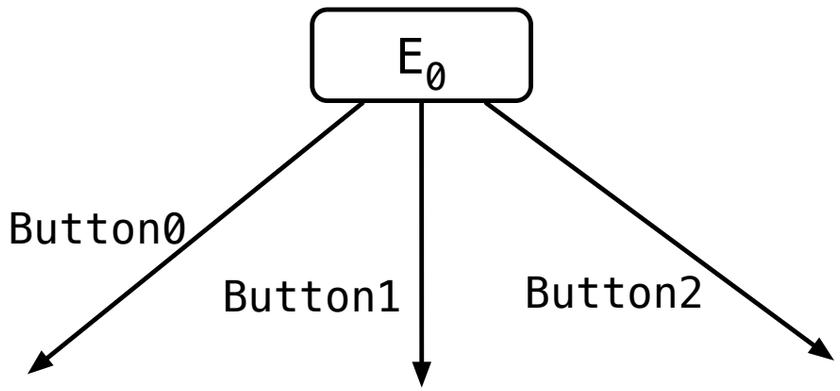




```
let b0Flag = false, b1Flag = false;
function b0() {
  b0Flag = true;
}
function b1() {
  b1Flag = true;
}
```

**Values of `b0Flag` and `b1Flag` should depend on event handler invoked**

# Challenge 2: Events as Expressions

Translate edges of event branching nodes to symbolic expressions

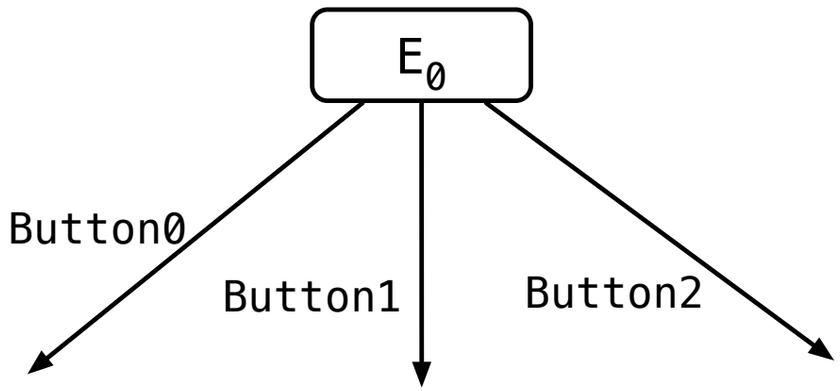



$E_i = target_{id}$

$E_0 =$ "Button1"

# Challenge 2: Events as Expressions

Translate edges of event branching nodes to symbolic expressions




$E_i$ = $target_{id}$

$E_0$ = "Button1"

```
let b0Flag = false, b1Flag = false;
function b0() {
  b0Flag = true;
}
function b1() {
  b1Flag = true;
}
```

b0Flag → ITE($E_0$ = "Button0", true, false)
b1Flag → ITE($E_0$ = "Button1", true, false)

# Evaluation

Research questions:

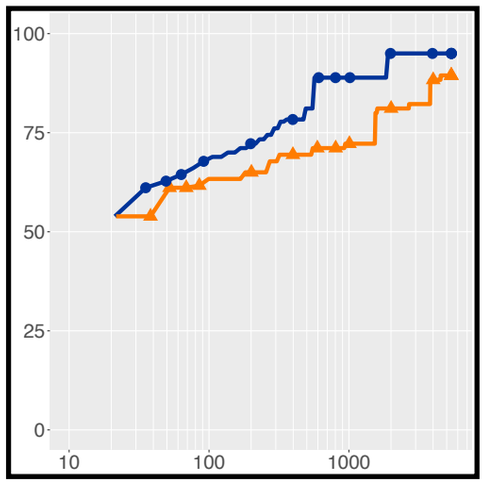
1. Does state merging improve the code coverage of the tester per test run?

2. Does state merging increase the computational overhead per test run?

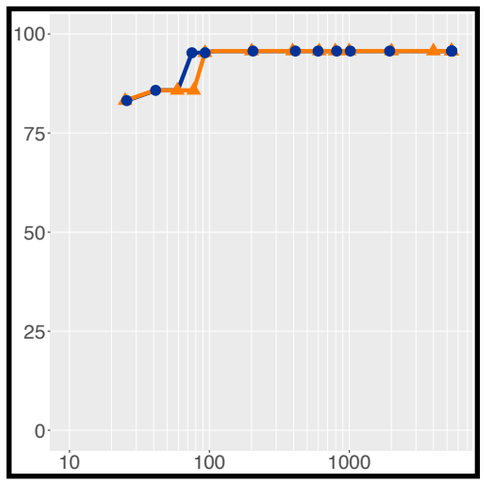
3. **Does state merging make it possible to cover a larger part of the application in less time?**

Evaluation based on eight full-stack JavaScript web applications

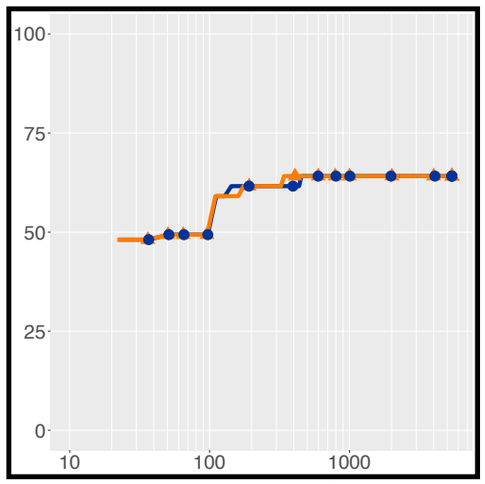
# RQ3: Code Coverage over Time



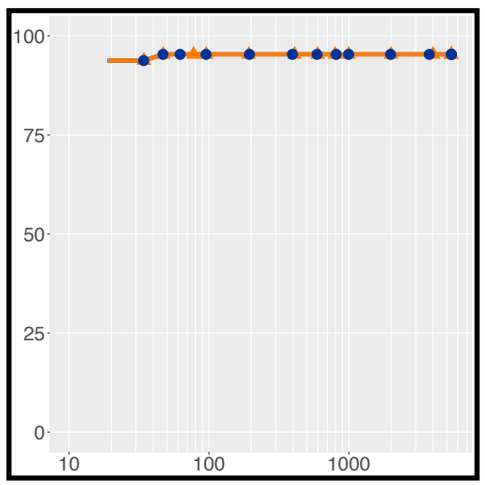
Calculator



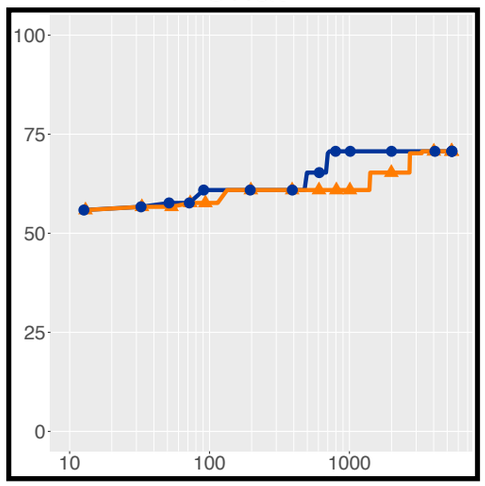
Chat



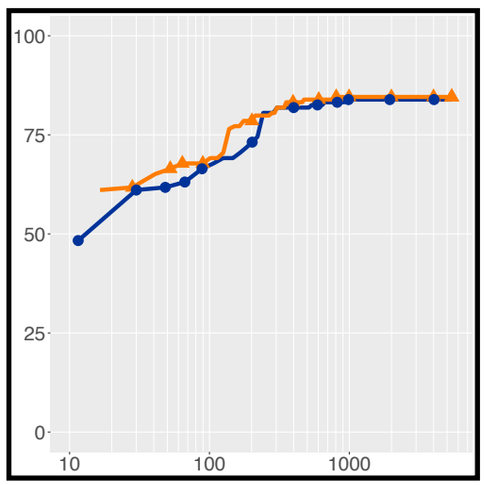
Game of Life



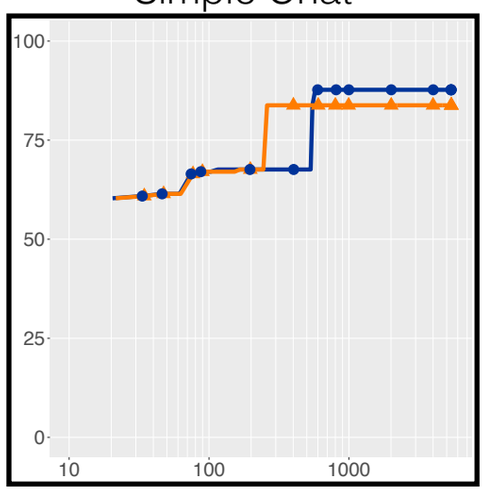
Simple Chat



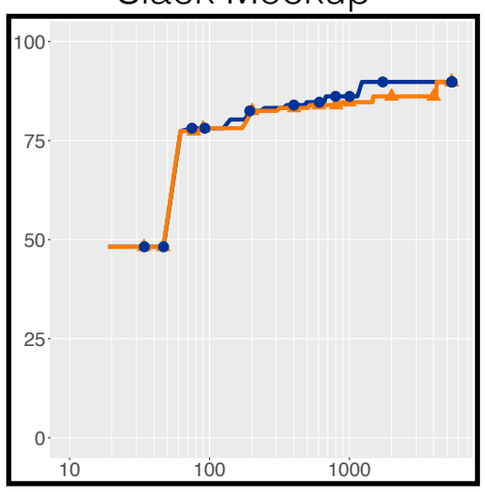
Slack Mockup



TOHacks



Totems

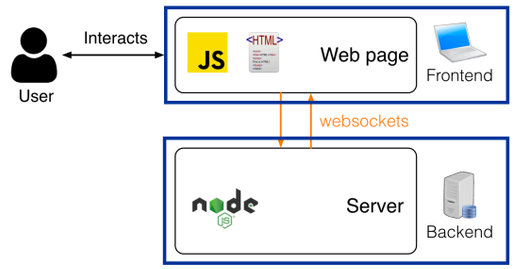


Whiteboard


Improved code coverage:
2/8 cases
Reduced code coverage:
1/8 cases
Equal code coverage:
5/8 cases

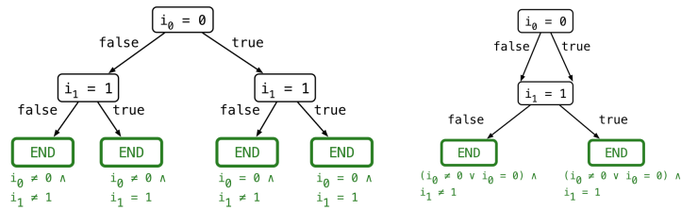
# Conclusion

## Full-stack JS Web Applications




Full-stack JavaScript web applications are web applications consisting of:
- **components implemented in JavaScript**
- **technologies accessed via JavaScript**

## State Merging

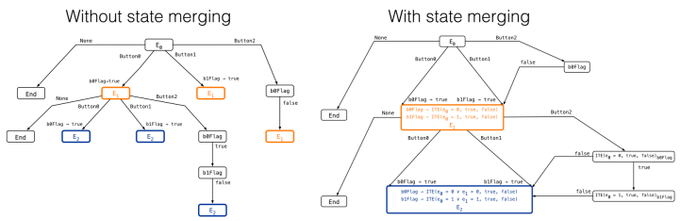



Symbolic execution **Directed Acyclic Graph (DAG)** with 2 **end states**
Use `If-Then-Else` symbolic expressions to maintain precision

```
if (randomInt() === 0) {
  a = 1;
} else {
  a = 0;
}
```

$a \rightarrow ITE(i_0 = 0, 1, 0)$

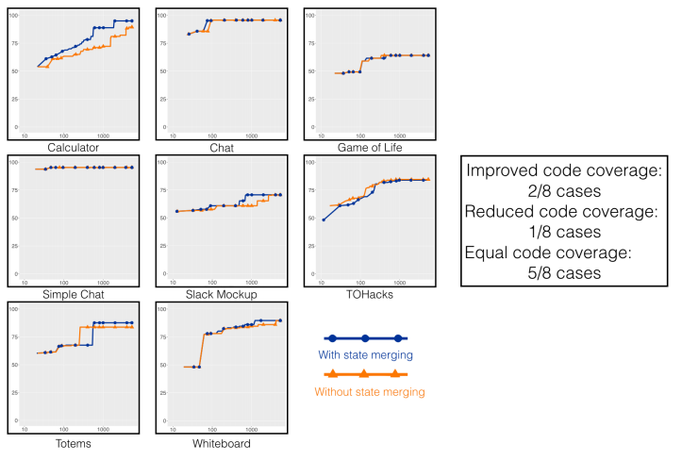
## Merging Event-driven Programs




Merge similar event branching nodes (instances of $E_1$ and $E_2$)

Merging event branching nodes enabled by:
- Redefining program point
- Translating event branching nodes to symbolic expressions

## RQ3: Code Coverage over Time




Improved code coverage: 2/8 cases
Reduced code coverage: 1/8 cases
Equal code coverage: 5/8 cases

https://github.com/softwarelanguageslab/StackFul

Inter-process Concolic Testing of Full-stack JavaScript Web Applications
https://soft.vub.ac.be/Publications/2023/vub-soft-phd-13-10.pdf