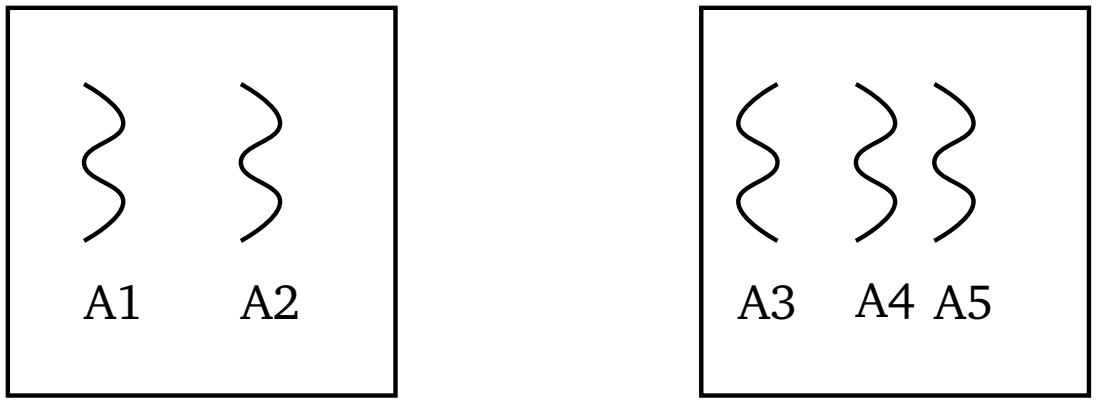
Compile-Time Analysis and Specialization of Clocks in Concurrent Programs

Nalini Vasudevan (Columbia University)
Olivier Tardieu (IBM Research)
Julian Dolby (IBM Research)
Stephen A. Edwards (Columbia University)

Background

The X10 Programming Language

- Concurrent programming model
- Activities are light weight threads
- Places are distributed memory locations



The X10 Programming Language

- Activities created using *async*

```
async {
/* Body of async
executed locally */
}
```

```
async (p2) {
/* Body of async
executed at p2 */
}
```

- Synchronization between activities through
  - *finish*
  - *atomic*
  - clocks

Clocks in X10

- Barriers in X10

```
final clock c = clock.factory.clock();
async clocked (c) {
    ..
    c.next();
    ..
    c.next();
}
async clocked (c) {
    ..
    c.next();
    ..
}
c.next();
..
c.next();
```

Clocks in X10

- Barriers in X10

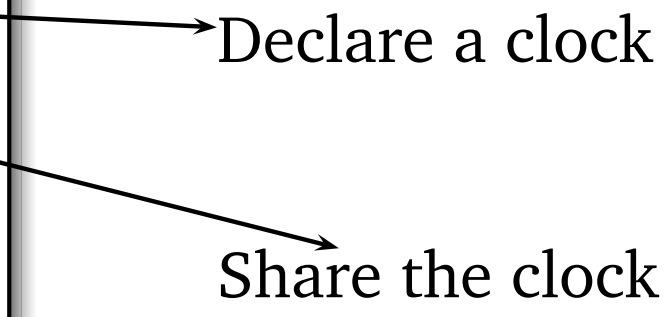
```
final clock c = clock.factory.clock();
async clocked (c) {
    ..
    c.next();
    ..
    c.next();
}
async clocked (c) {
    ..
    c.next();
    ..
}
c.next();
..
c.next();
```

Declare a clock

Clocks in X10

- Barriers in X10

```
final clock c = clock.factory.clock();
async clocked (c) {
    ..
    c.next();
    ..
    c.next();
}
async clocked (c) {
    ..
    c.next();
    ..
}
c.next();
..
c.next();
```



Clocks in X10

- Barriers in X10

```
final clock c = clock.factory.clock();
async clocked (c) {
    ..
    c.next();
    ..
    c.next();
}
async clocked (c) {
    ..
    c.next();
    ..
}
c.next();
..
c.next();
```

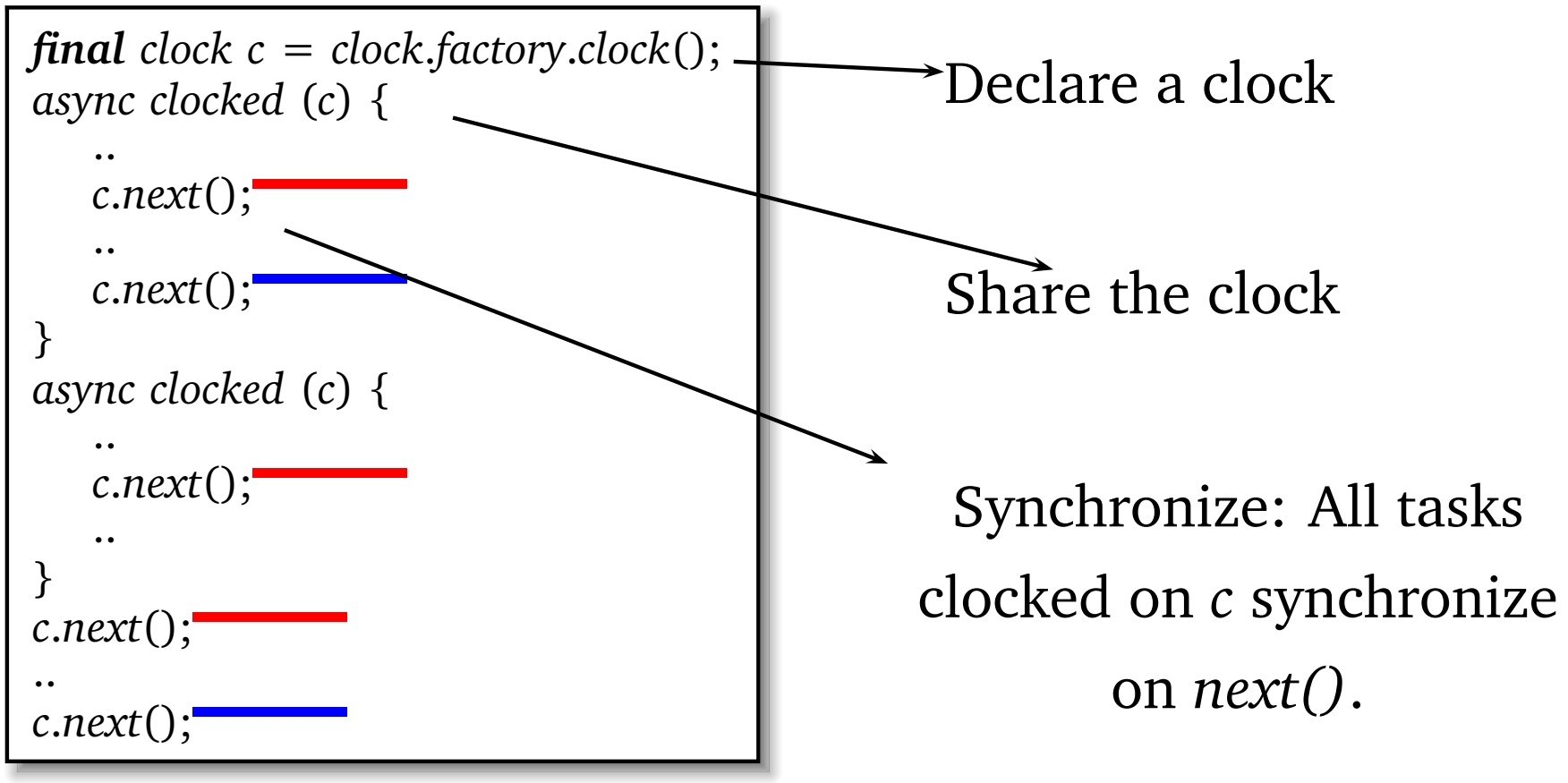
Declare a clock

Share the clock

Synchronize: All tasks clocked on *c* synchronize on *next()*.

Clocks in X10

- Barriers in X10



More about Clocks

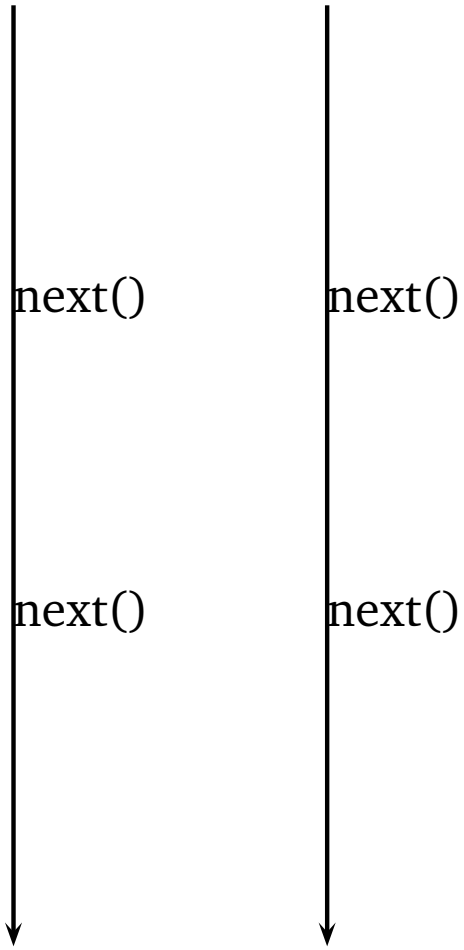
- *next()*

```
final clock c = clock.factory.clock();
async clocked (c) {
   ..
   c.next();
   ..
   ..
   ..
   c.next();
   ..
}
   ..
   c.next();
   ..
   c.next();
```

More about Clocks

- *next()*

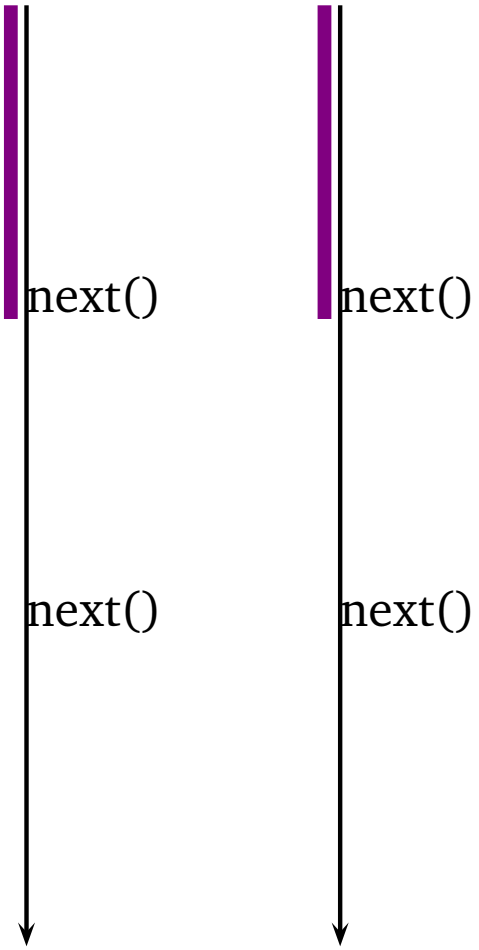
```
final clock c = clock.factory.clock();
async clocked (c) {
   ..
   c.next();
   ..
   ..
   ..
   c.next();
   ..
}
   ..
   c.next();
   ..
   c.next();
```



More about Clocks

- *next()*

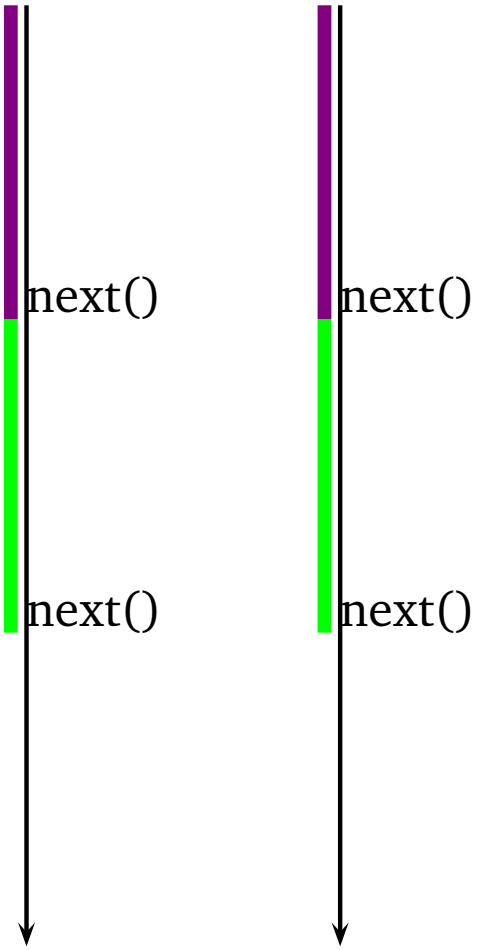
```
final clock c = clock.factory.clock();
async clocked (c) {
   ..
   c.next();
   ..
   ..
   ..
   c.next();
   ..
}
   ..
   c.next();
   ..
   c.next();
```



More about Clocks

- *next()*

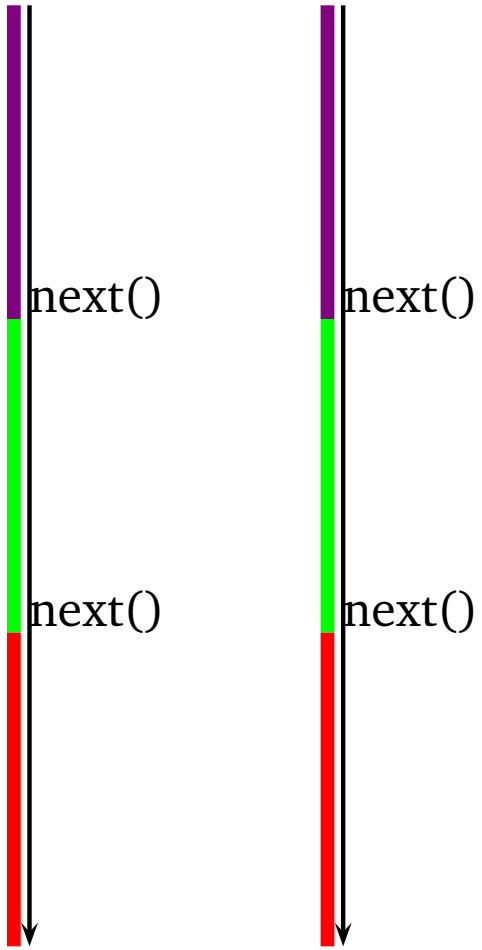
```
final clock c = clock.factory.clock();
async clocked (c) {
   ..
   c.next();
   ..
   ..
   ..
   c.next();
   ..
}
   ..
   c.next();
   ..
   c.next();
```



More about Clocks

- *next()*

```
final clock c = clock.factory.clock();
async clocked (c) {
   ..
   c.next();
   ..
   ..
   ..
   c.next();
   ..
}
   ..
   c.next();
   ..
   c.next();
```



More about Clocks

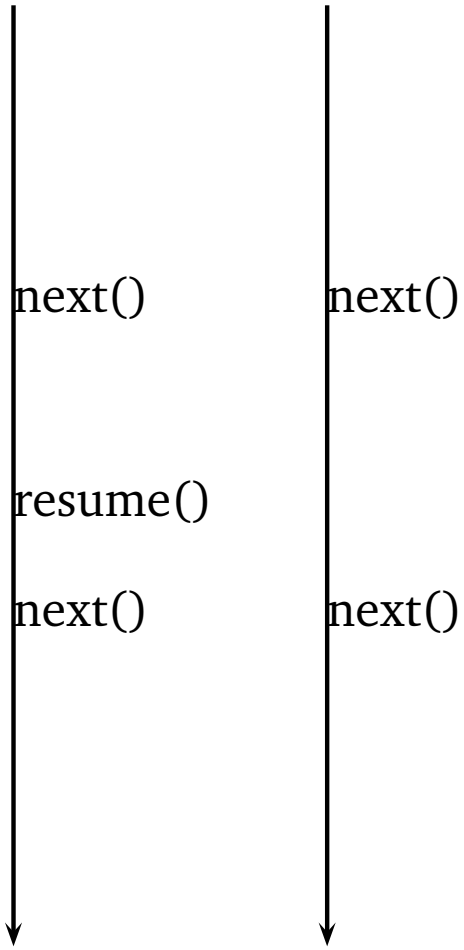
- *resume()*

```
final clock c = clock.factory.clock();
async clocked (c) {
    ..
    c.next();
    ..
    c.resume();
    ..
    c.next();
    ..
}
    ..
    c.next();
    ..
    c.next();
```

More about Clocks

- *resume()*

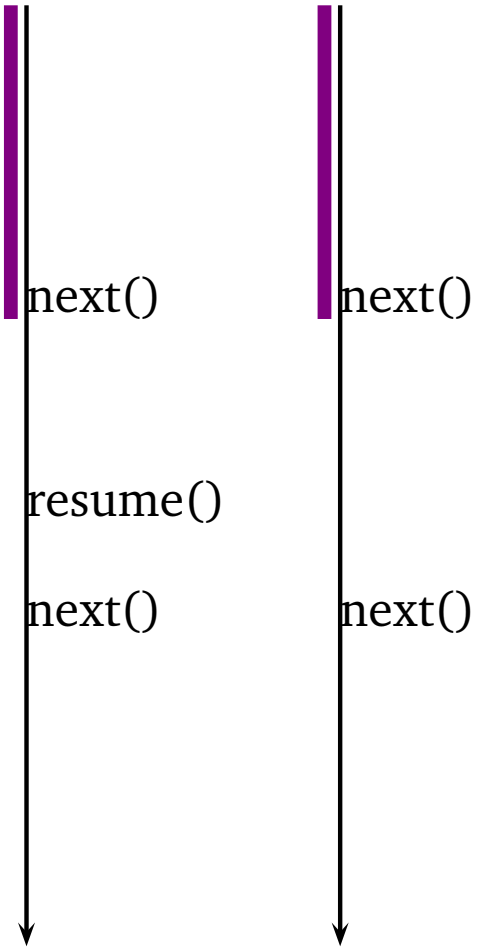
```
final clock c = clock.factory.clock();
async clocked (c) {
   ..
   c.next();
   ..
   c.resume();
   ..
   c.next();
   ..
}
   ..
   c.next();
   ..
   c.next();
```



More about Clocks

- *resume()*

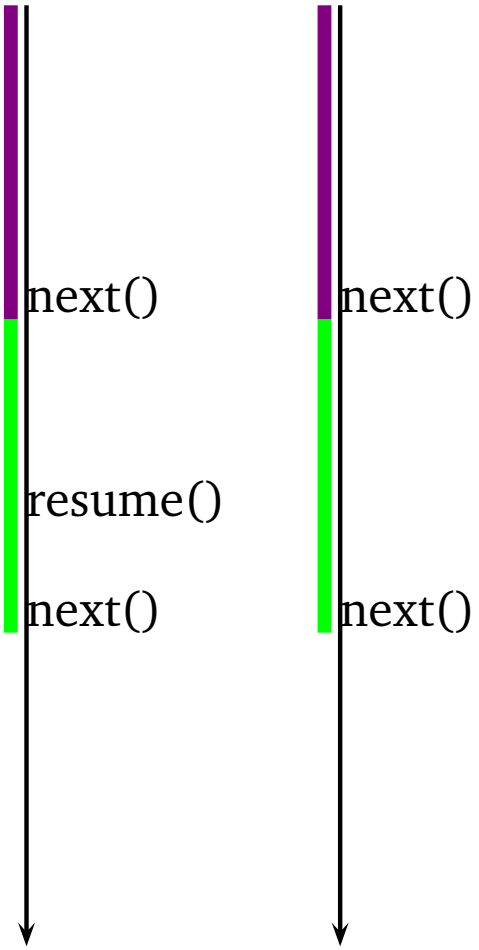
```
final clock c = clock.factory.clock();
async clocked (c) {
   ..
   c.next();
   ..
   c.resume();
   ..
   c.next();
   ..
}
   ..
   c.next();
   ..
   c.next();
```



More about Clocks

- *resume()*

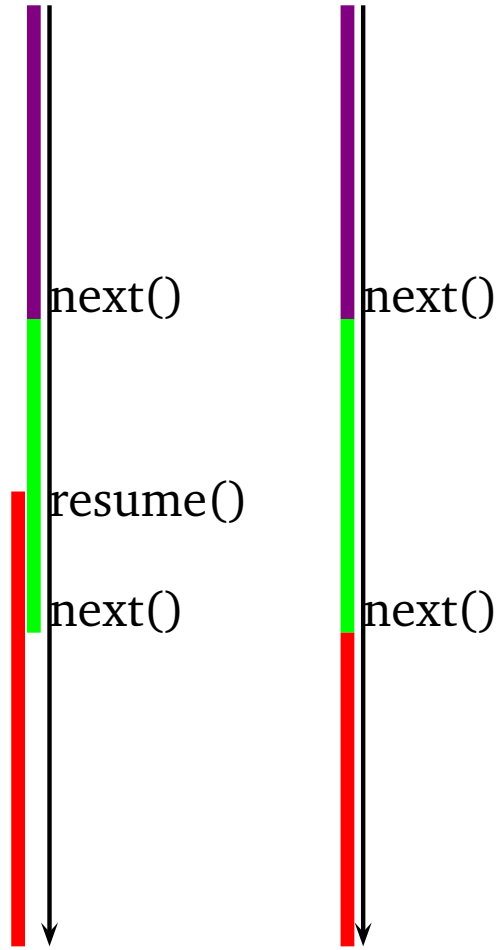
```
final clock c = clock.factory.clock();
async clocked (c) {
   ..
   c.next();
   ..
   c.resume();
   ..
   c.next();
   ..
}
   ..
   c.next();
   ..
   c.next();
```



More about Clocks

- *resume()*

```
final clock c = clock.factory.clock();
async clocked (c) {
   ..
   c.next();
   ..
   c.resume();
   ..
   c.next();
   ..
}
   ..
   c.next();
   ..
   c.next();
```



More about Clocks

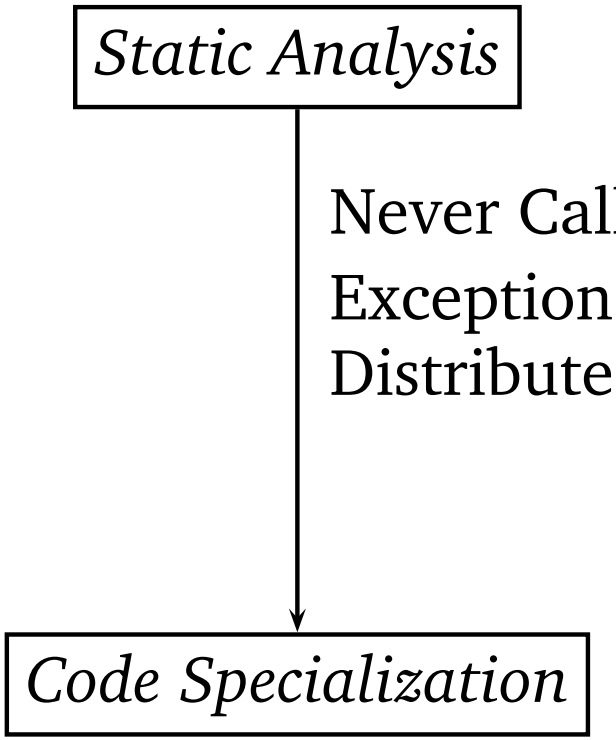
- *drop()*
  - Explicitly drop the clock
  - Task does not have to synchronize anymore

```
final clock c = clock.factory.clock();
async clocked (c) {
   ..
   c.next();
   ..
   c.drop();
   ..
}
   ..
   c.next();
   ..
   c.next();
```

Motivation

- Default implementation handles all cases
- Protocol violation is handled by exceptions
  - Example: call *next()* after *drop()*
- We can generate more efficient code if
  - Activity never violates the protocol
  - Activity never calls *resume()* on clock c

Design Overview



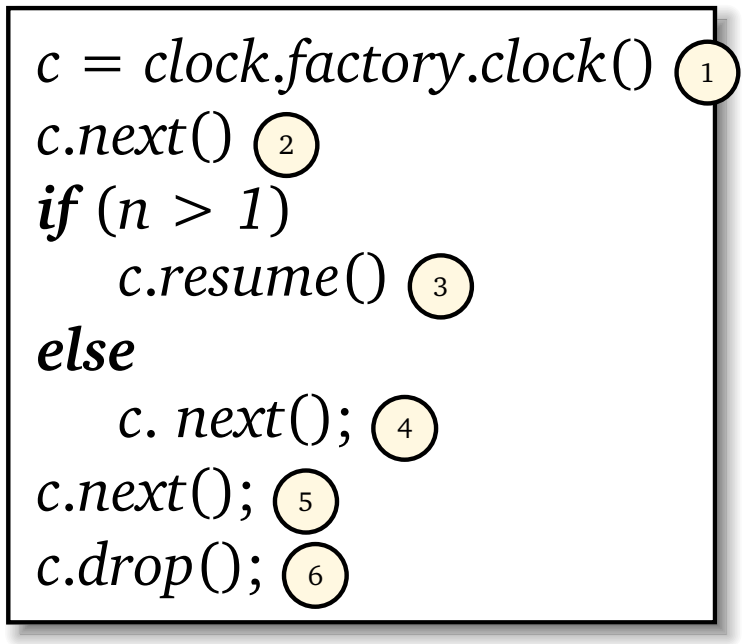
- Extract model from IR
- One automaton per clock
- Use NuSMV

- X10 compiler plug-in to choose clock implementation based on analysis result
- Each clock specialized independently

Building the Automaton

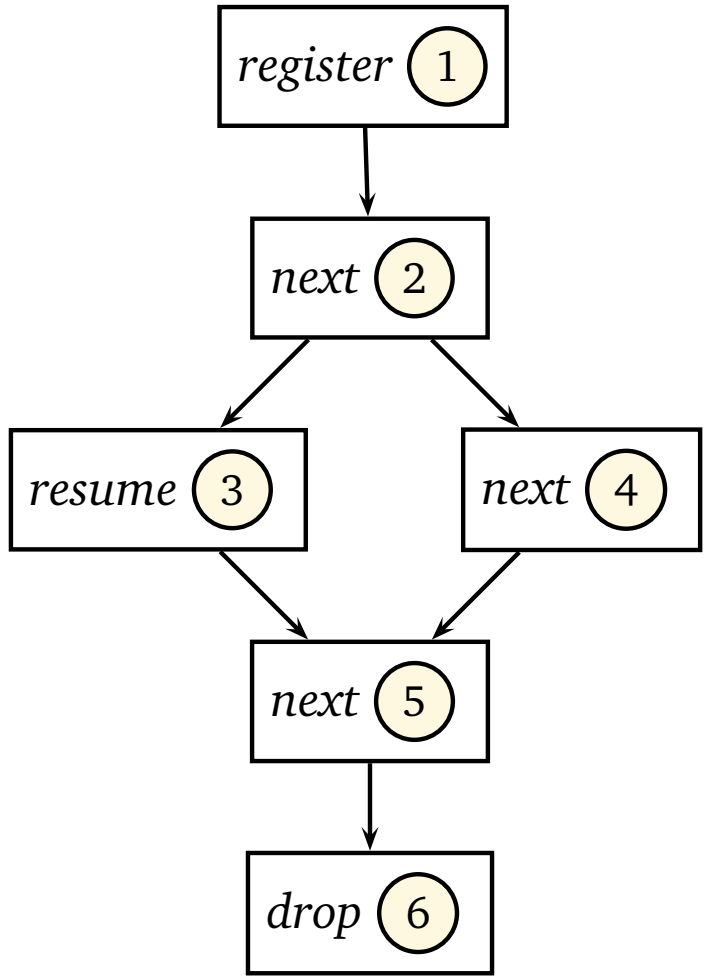
```
c = clock.factory.clock()
c.next()
if (n > 1)
    c.resume()
else
    c. next();
c.next();
c.drop();
```

Building the Automaton

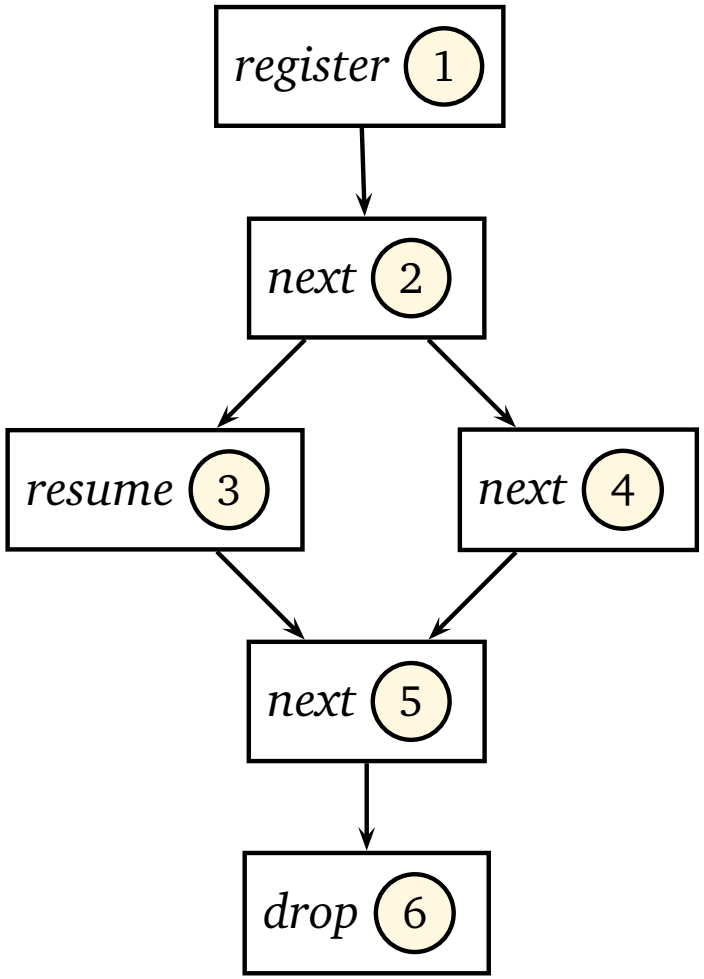


Building the Automaton

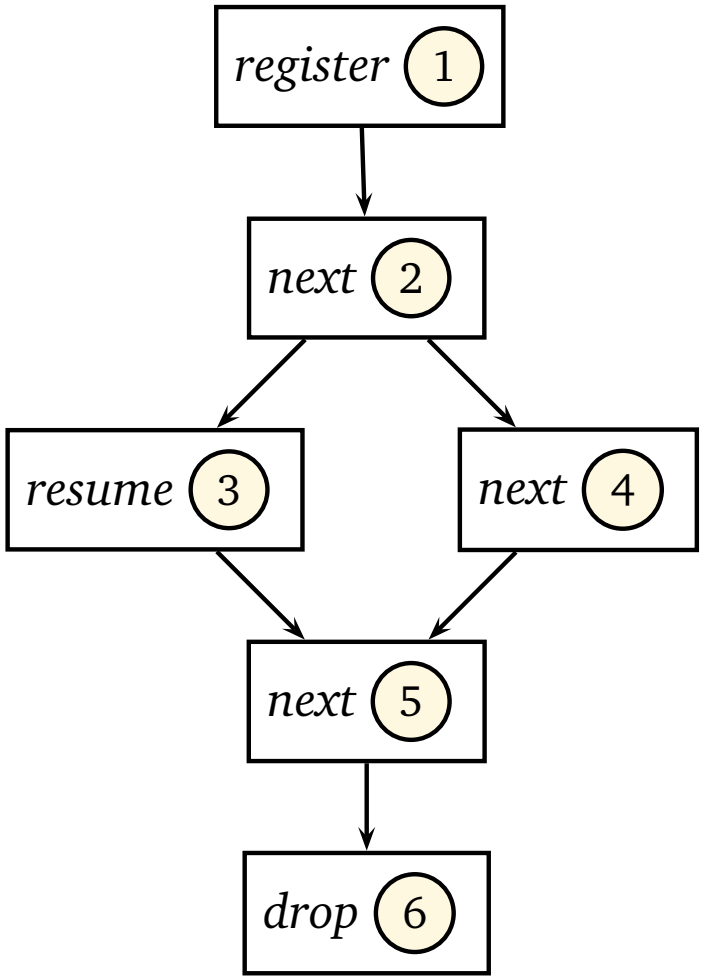
```
c = clock.factory.clock()  1
c.next()  2
if (n > 1)
    c.resume()  3
else
    c. next();  4
c.next();  5
c.drop();  6
```



Building the Automaton

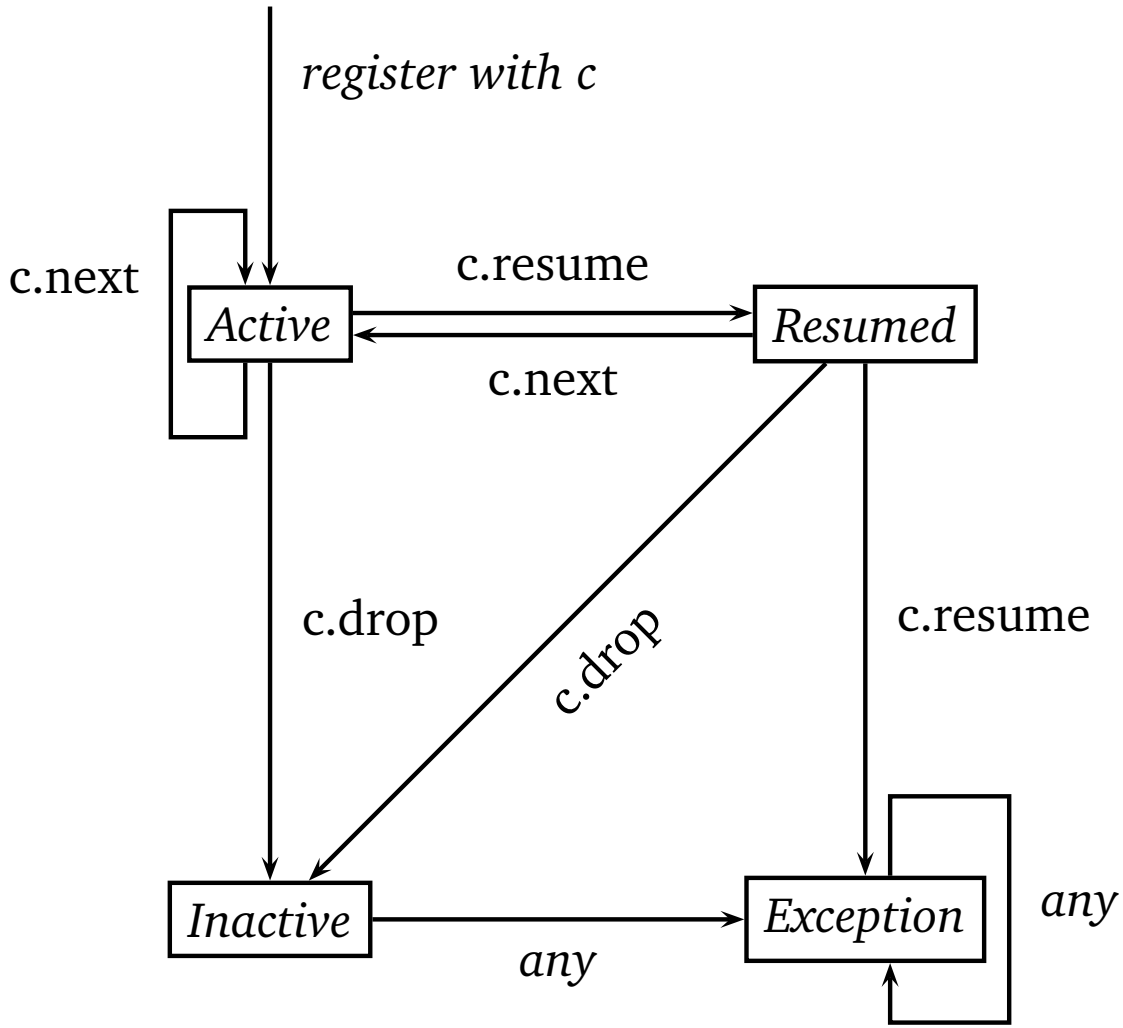


Building the Automaton




```
init (clock) = register;
next(clock) :=
case
   (clock = register) : next_2;
   (clock = next_2) : {resume_3, next_4};
   (clock = resume_3) : next_5;
   (clock = next_4) : next_5;
   (clock = next_5) : drop_6;
   1: clock;
esac;
DEFINE clock_next = clock in
                        {next_2, next_4, next_5}
DEFINE clock_resume = clock in {resume_3}
..
```

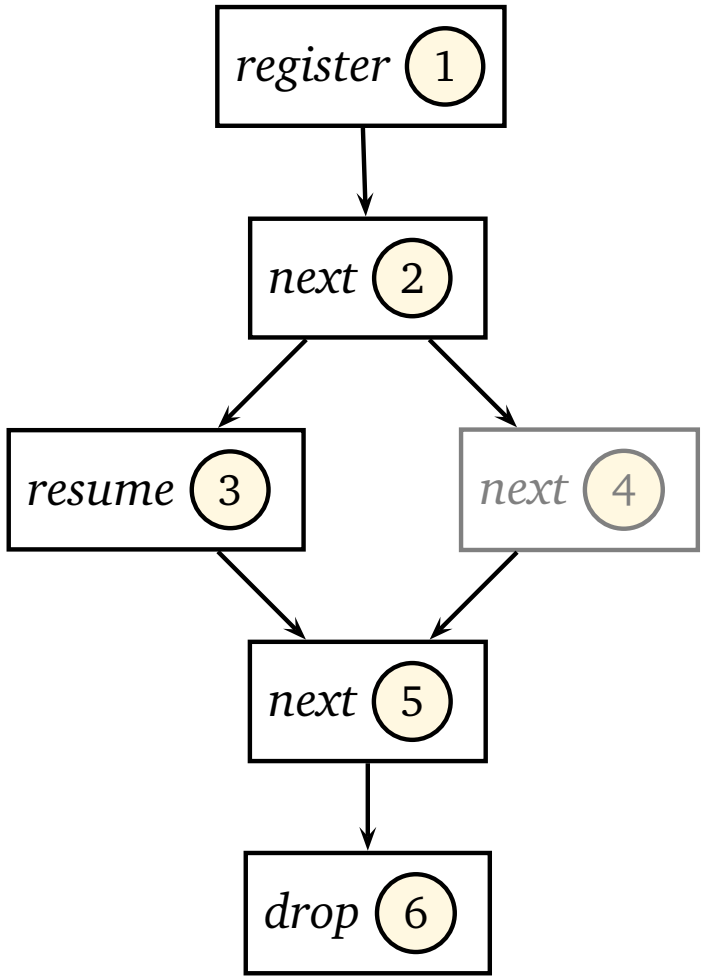
The Protocol



The Monitor

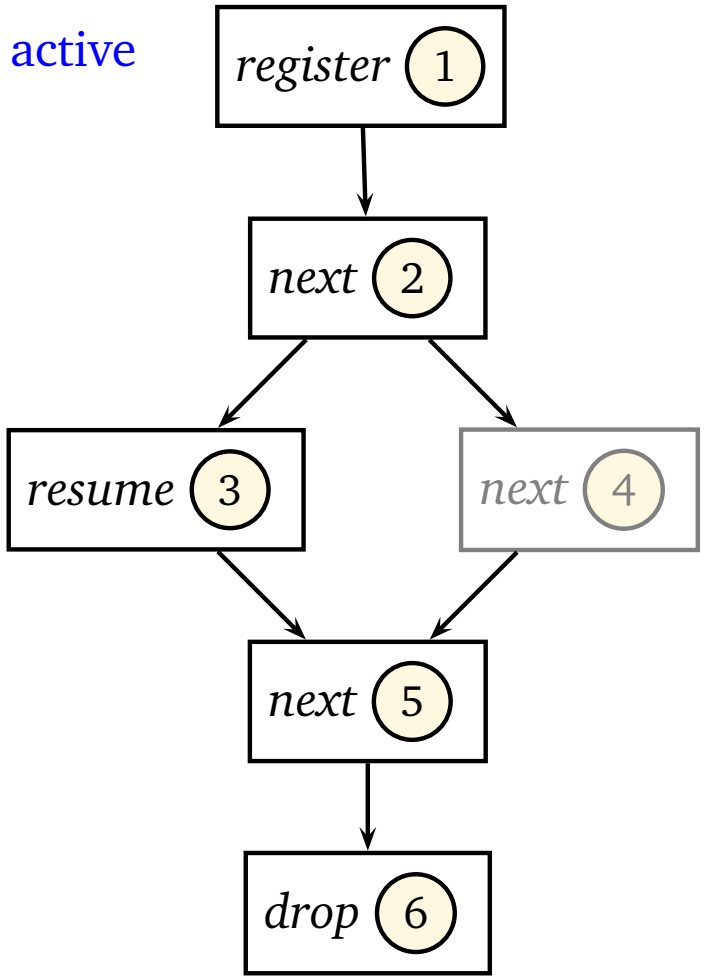
```
init (state) = inactive;
next(state) :=
case
   (state = inactive) & (clock_register) : active;
   (state = active) & (clock_drop) : inactive;
   (state = active) & (clock_resume): resumed;
   (state = resumed) & (clock_next): active;
   ..
-- Exception cases
   (state = resumed) & (clock_resume): exception;
   (state = inactive) & (clock_next) : exception;
```

The Monitor

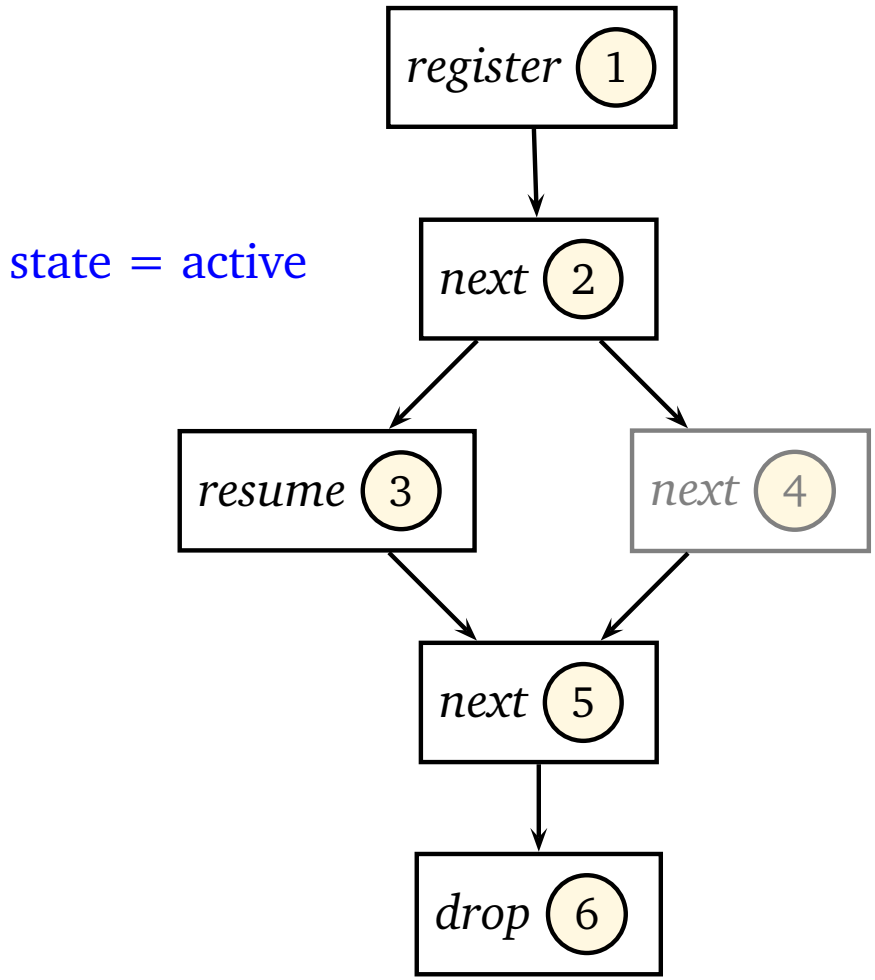


The Monitor

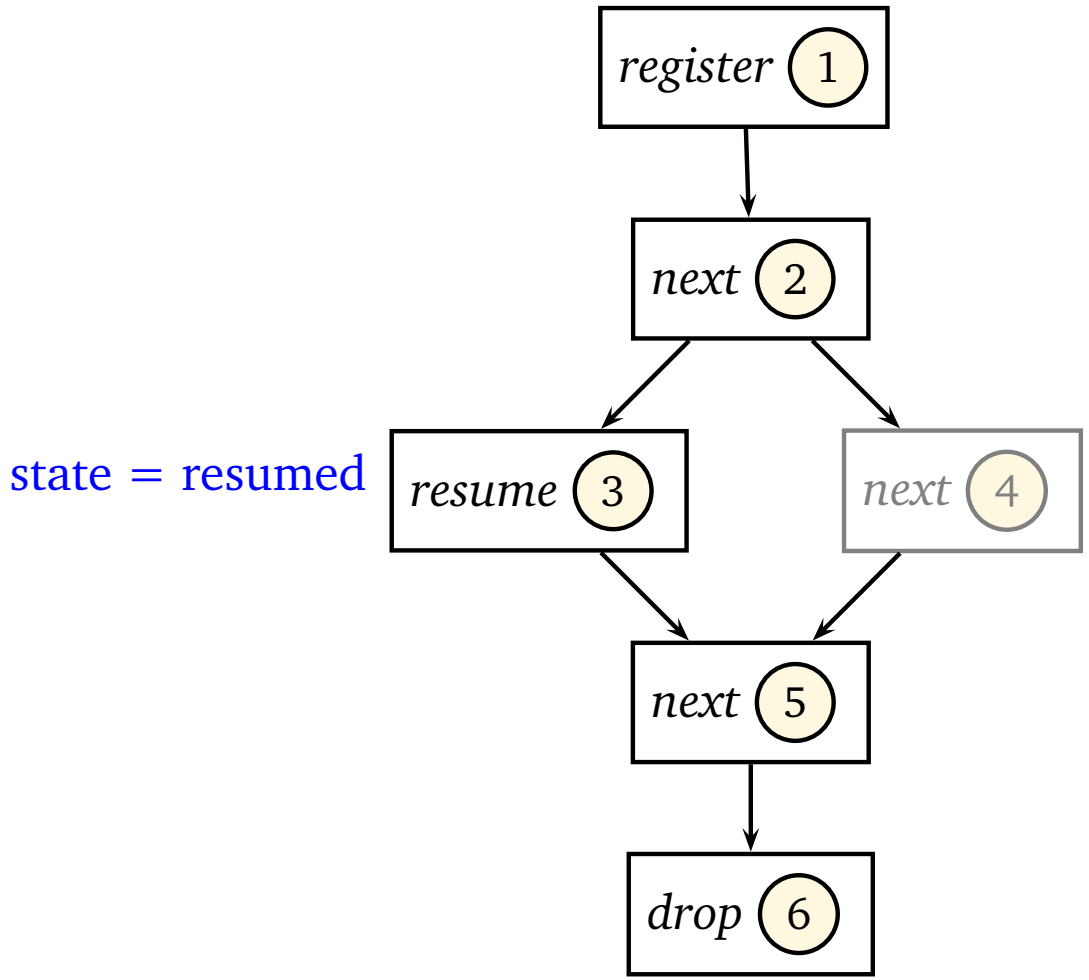
state = active



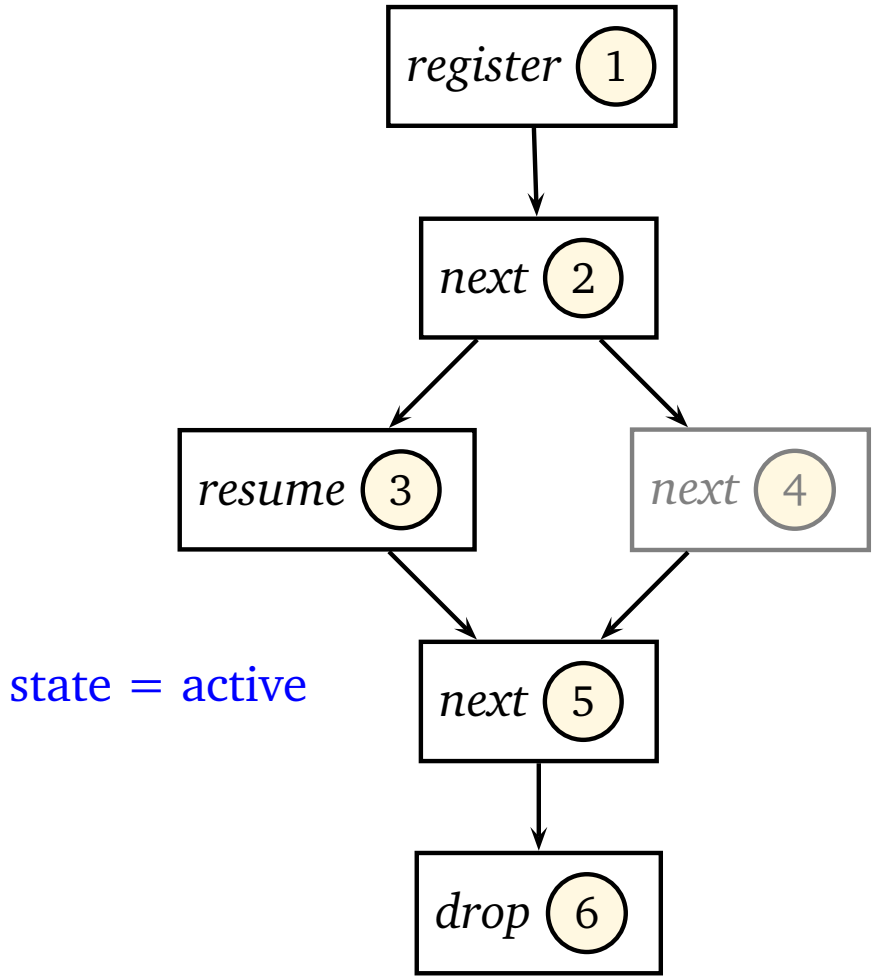
The Monitor



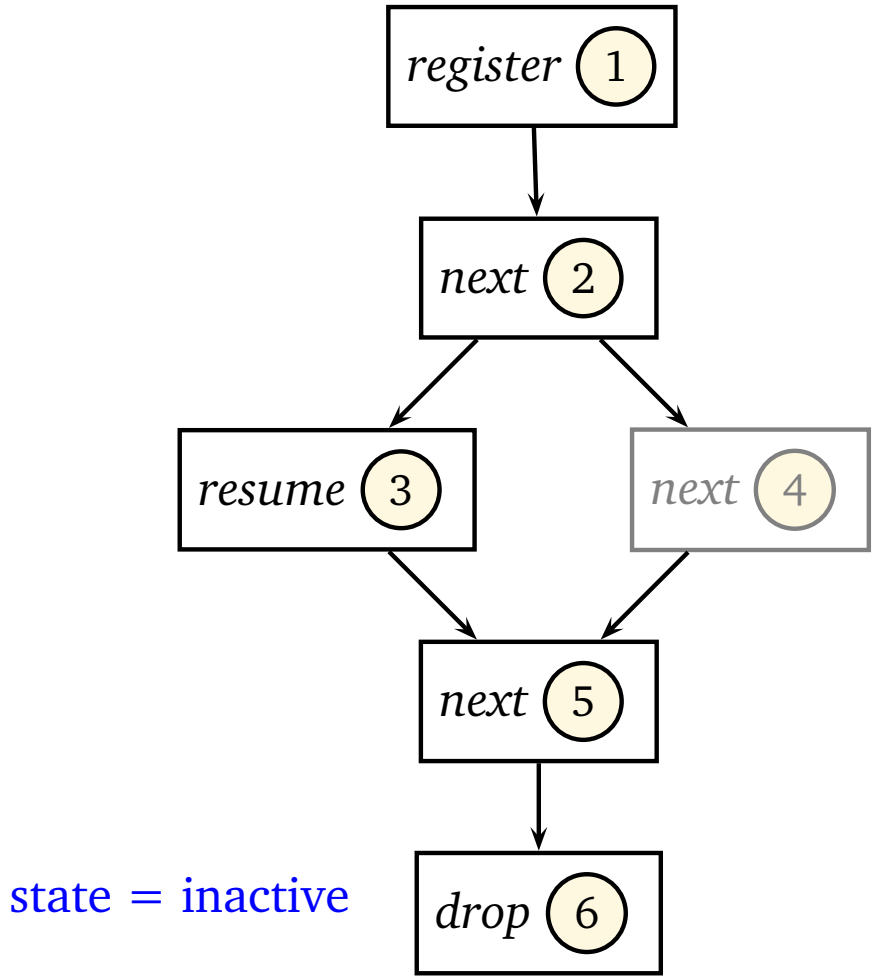
The Monitor



The Monitor



The Monitor



Dealing with *async*

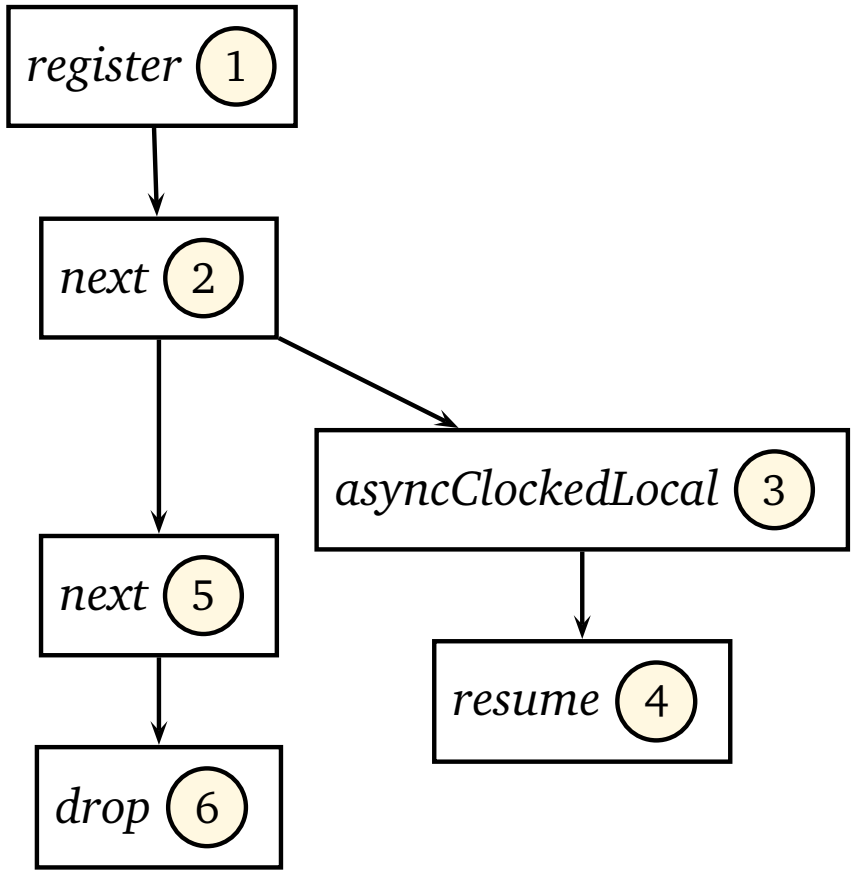
```
c = clock.factory.clock()  1
c.next();  2

async clocked (c) {  3
    c.resume();  4
}
c.next();  5
c.drop();  6
```

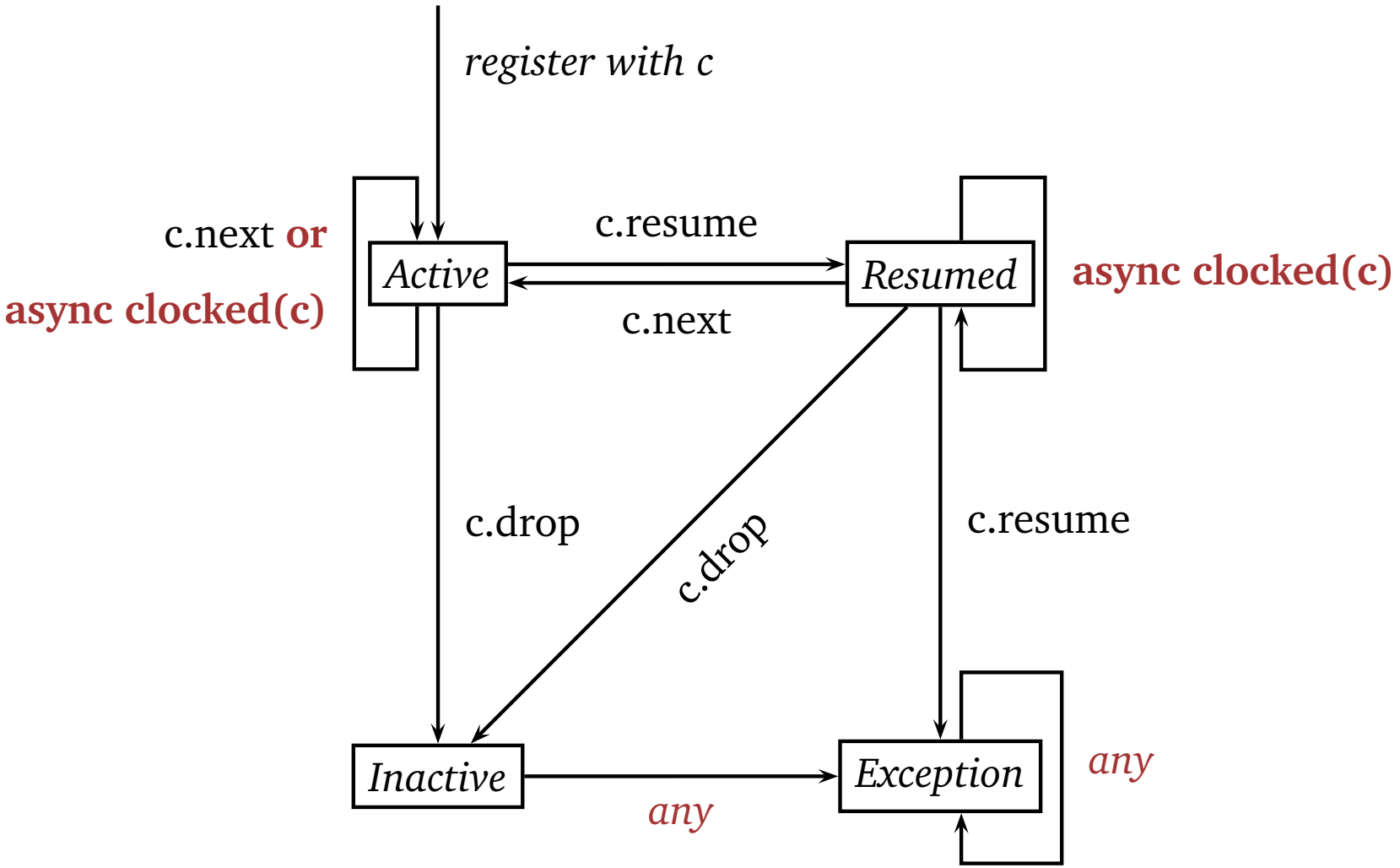
Dealing with *async*

```
c = clock.factory.clock()  1
c.next();  2

async clocked (c) {  3
    c.resume();  4
}
c.next();  5
c.drop();  6
```



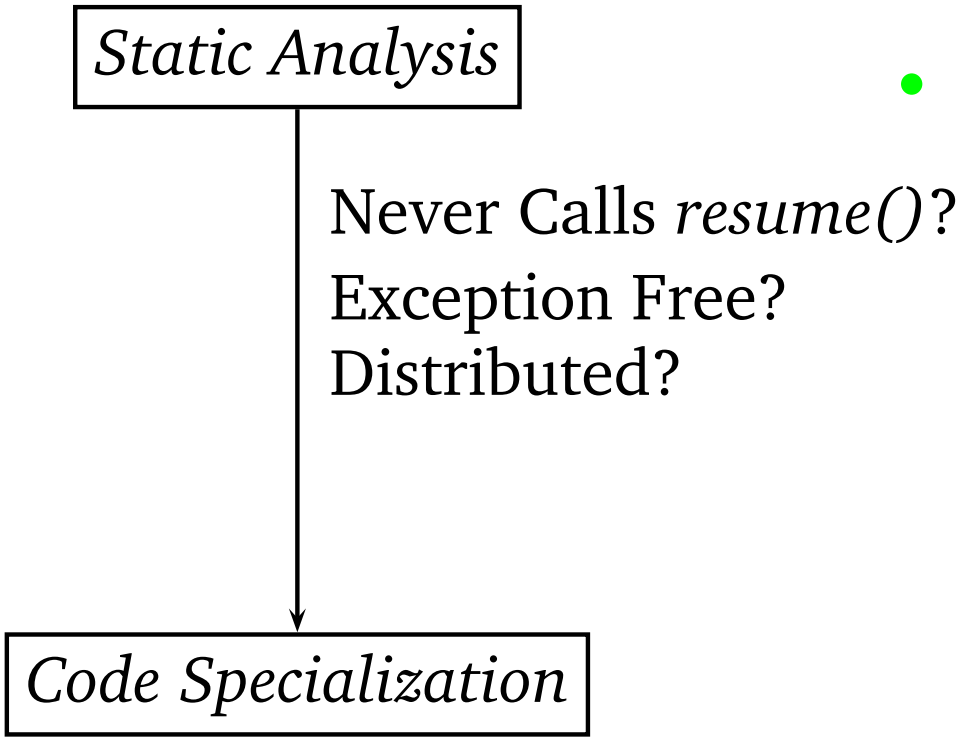
The Protocol



Checking for properties

- Is the clock protocol violation free?
  AG (!($state = exception$))
- Does the clock ever call resume()?
  AG (!($state = resumed$))
- Is the clock used in multiple places?
  AG(!$clock\_asyncClockedRemote$)
- Do the spawned activities ever call next?
  G($clock\_next- >$
  H(!($clock\_asyncClockedLocal|clock\_asyncClockedRemote$)))

Code Specialization



- We generate different code for each clock
- For eg., we remove run-time exception checks for exception free clocks

Experimental Results

Example	Clocks	Lines	Result	Speed Up	Analysis Time Base	Analysis Time NuSMV
Linear Search	1	35	EF, NR, L	35.2%	33.5s	0.4s
Relaxation	1	55	EF, NR, L	87.6	6.7	0.3
All Reduction Barrier	1	65	EF, NR	1.5	27.2	0.1
Pascal's Triangle	1	60	EF, L	20.5	25.8	0.4
Prime Number Sieve	1	95	NR, L	213.9	34.7	0.4
N-Queens	1	155	EF, NR, ON, L	1.3	24.3	0.5
LU Factorization	1	210	EF, NR	5.7	20.6	0.9
MolDyn JGF Bench.	1	930	NR	2.3	35.1	0.5
Pipeline	2	55	Clock 1: EF, NR, L Clock 2: EF, NR, L	31.4	7.5	0.5
Edmiston	2	205	Clock 1: NR, L Clock 2: NR, L	14.2	29.9	0.5

EF: No ClockUseException
NR: No Resume
ON: Only the activity that created the clock calls *next* on it
L: Clocked used locally (in a single place)

Conclusion

- Sequential analysis for concurrency optimization
- Our analysis is safe
- Combined with alias analysis
- Future work
  - Modular techniques
  - Inter-activity analysis
  - Refining alias analysis using clock information

Beginning of Research

Combining with Alias Analysis

```
final clock c = clock.factory.clock();
final clock d = clock.factory.clock();
final clock x = (n > 1)? c: d;
x.next();
x.resume();
```

Combining with Alias Analysis

```
final clock c = clock.factory.clock();
final clock d = clock.factory.clock();
final clock x = (n > 1)? c: d;
x.next();
x.resume();
```

```
final clock c = clock.factory.clock();
final clock d = clock.factory.clock();
if (*) {
    c.next();
    c.resume();
} else {
    d.next();
    d.resume();
}
```

Combining with Alias Analysis

```
final clock c = clock.factory.clock();
final clock d = clock.factory.clock();
final clock x = (n > 1)? c: d;
x.next();
x.resume();
```

```
final clock c = clock.factory.clock();
final clock d = clock.factory.clock();
if (*)
    c.next();
else
    d.next();
if (*)
    c.resume();
else
    d.resume();
}
```

Related Work

- Typestate analysis
- Model checking concurrent programs
- Code specialization
  - Specialization of CML message-passing techniques [Reppy 2007]
- Analysis of X10 programs
  - May-happen-in-parallel analysis [Agarwal 2007]