‘now’
‘happens-before’
‘later’

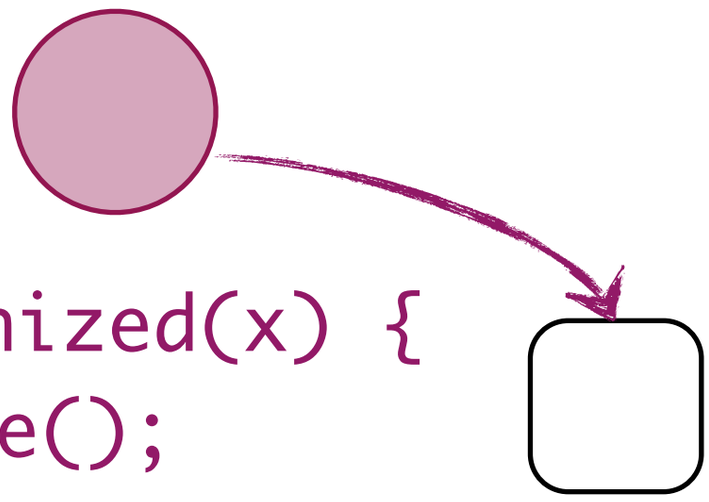
Christoph M. Angerer
Thomas R. Gross
ETH Zurich, Switzerland

Static Schedule Analysis with Explicit Happens-before Relationships

Example

Example

task A

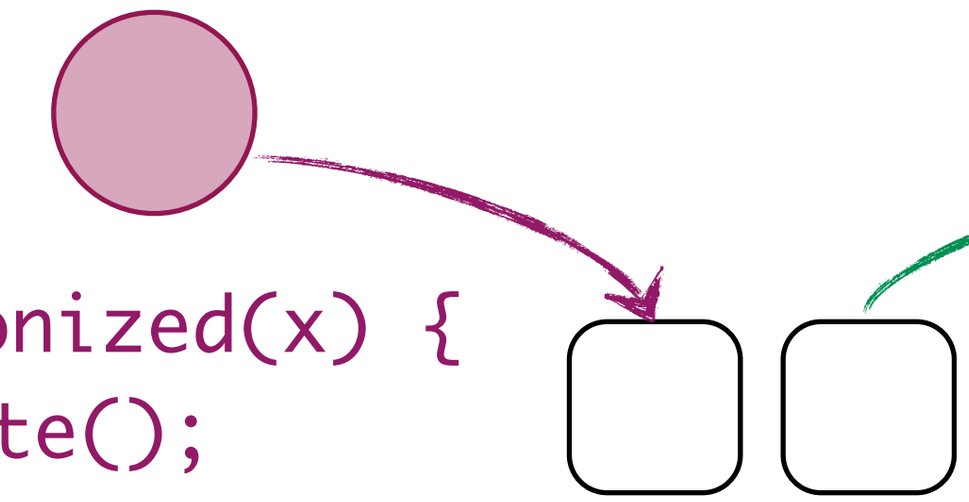


```
synchronized(x) {
  x.write();
}
```

Example

task A

task B

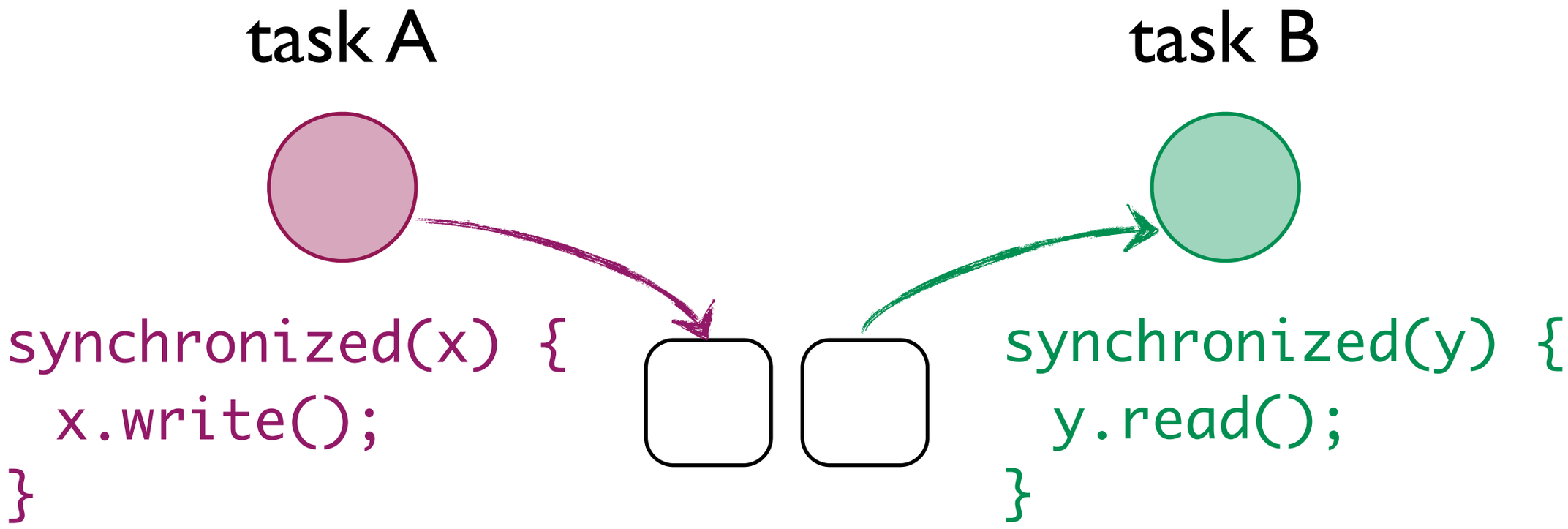


```
synchronized(x) {
  x.write();
}
```

```
synchronized(y) {
  y.read();
}
```

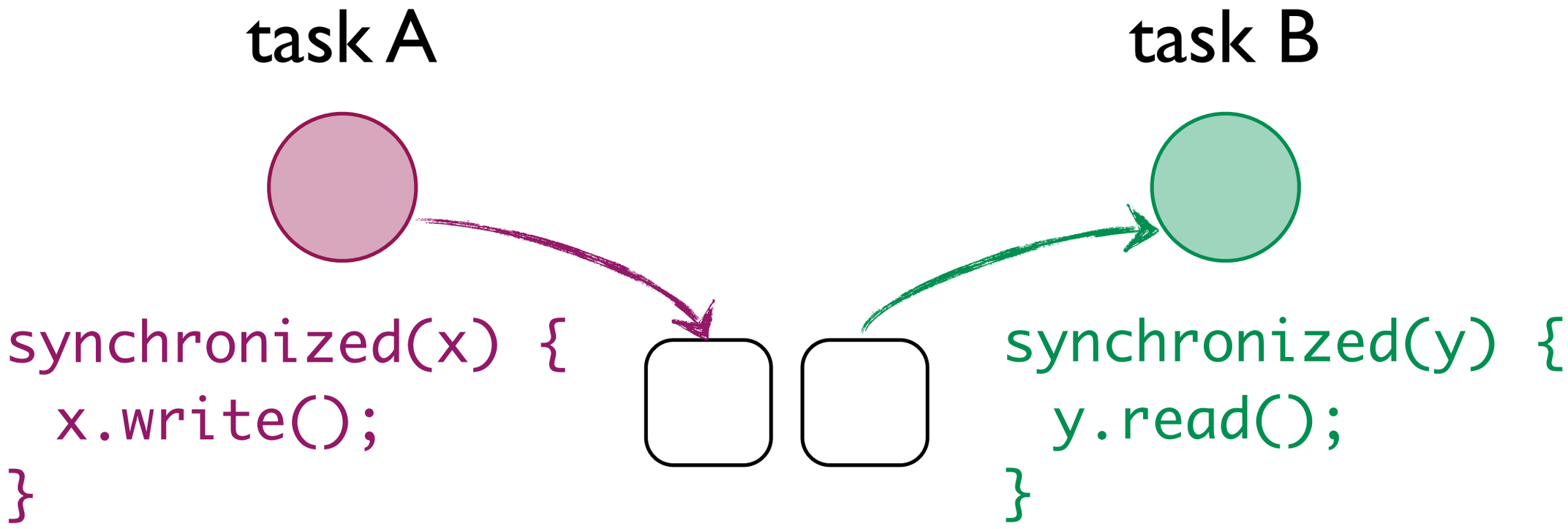
Example

- Can we remove synchronization for x, y?



Example

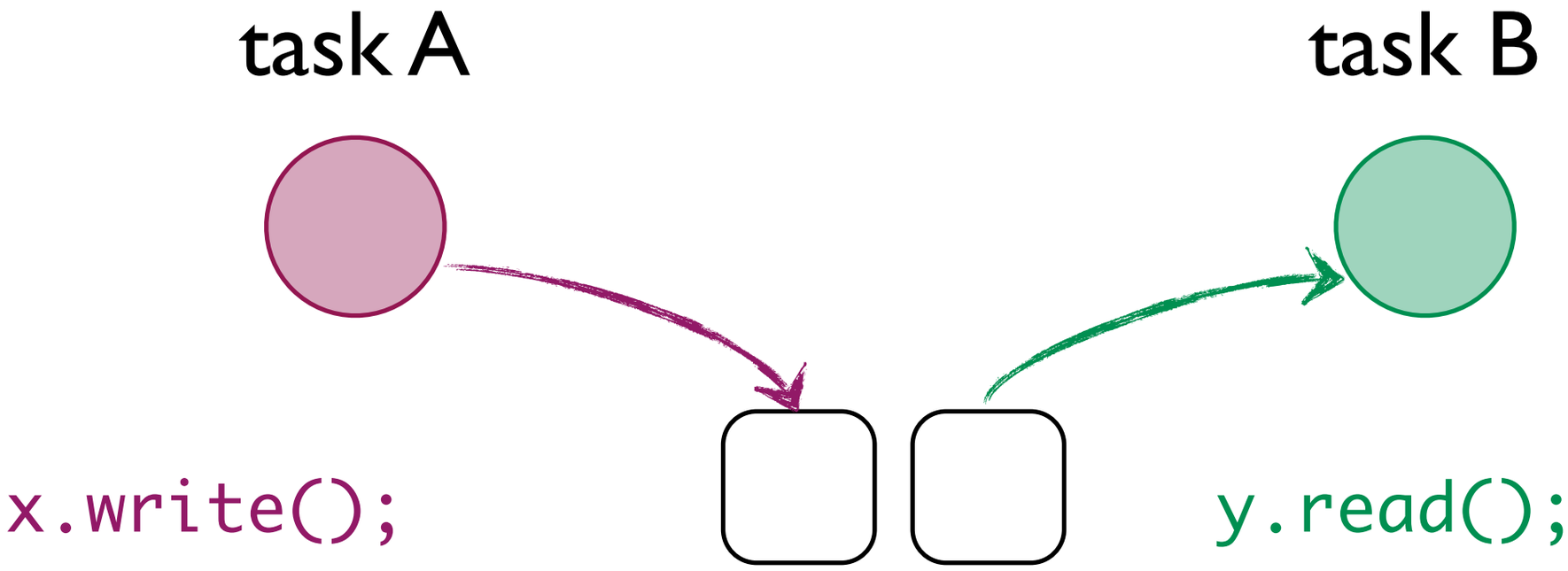
- Can we remove synchronization for x, y?



☑ Different Objects

Example

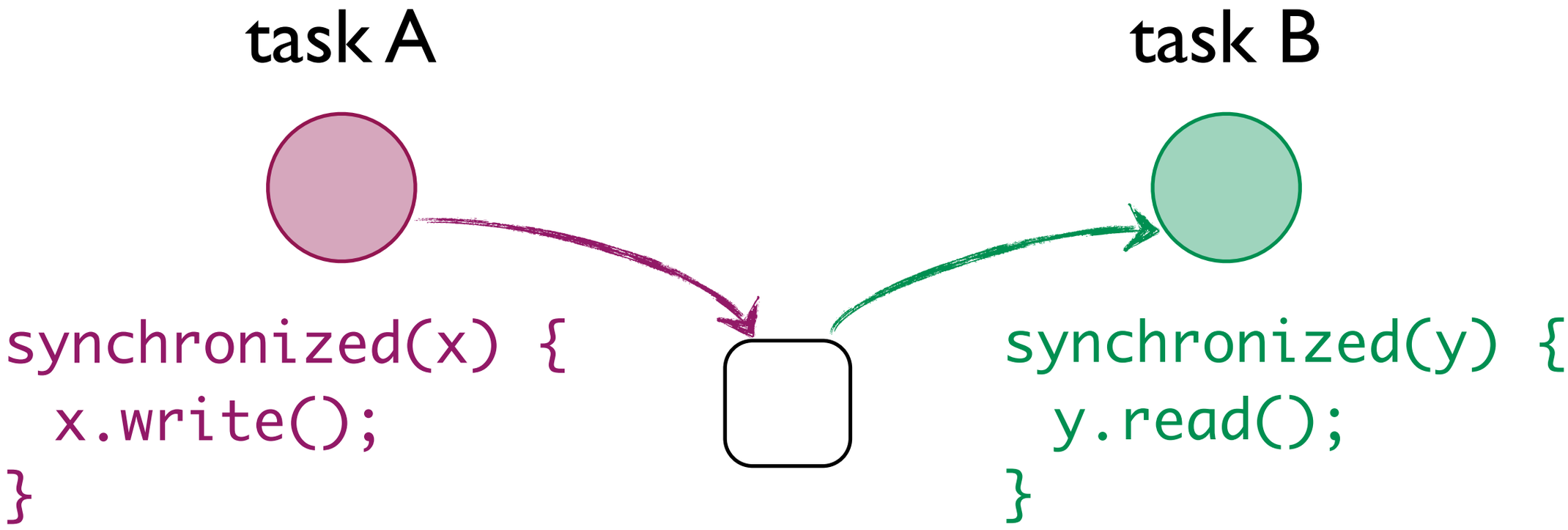
- Can we remove synchronization for x, y?



☑ Different Objects

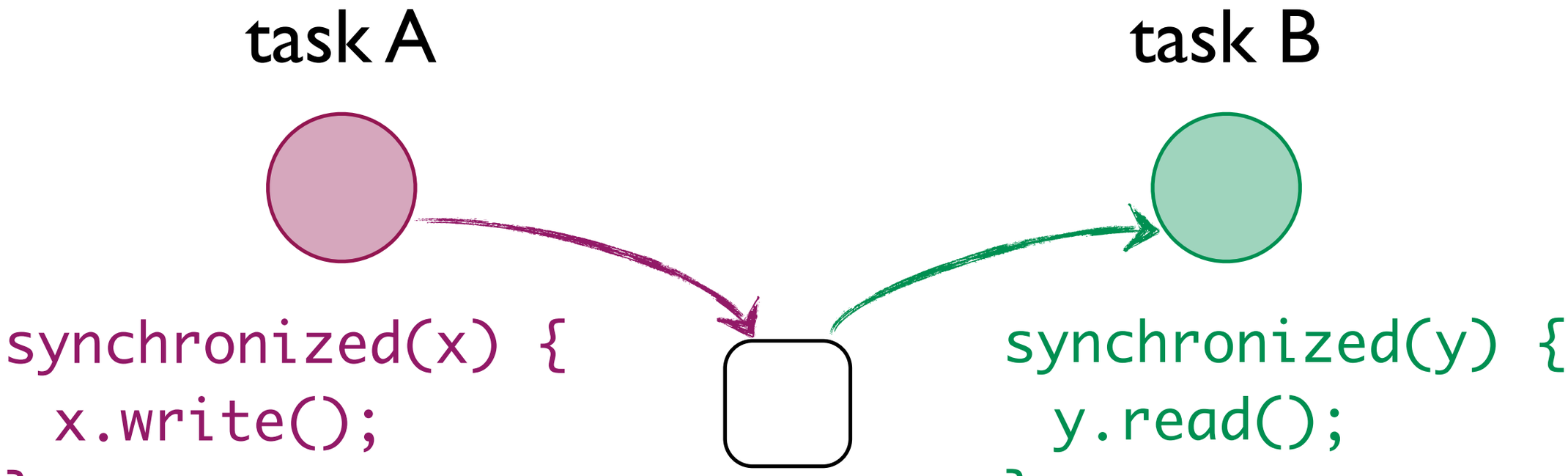
Example

- Can we remove synchronization for x, y?



Example

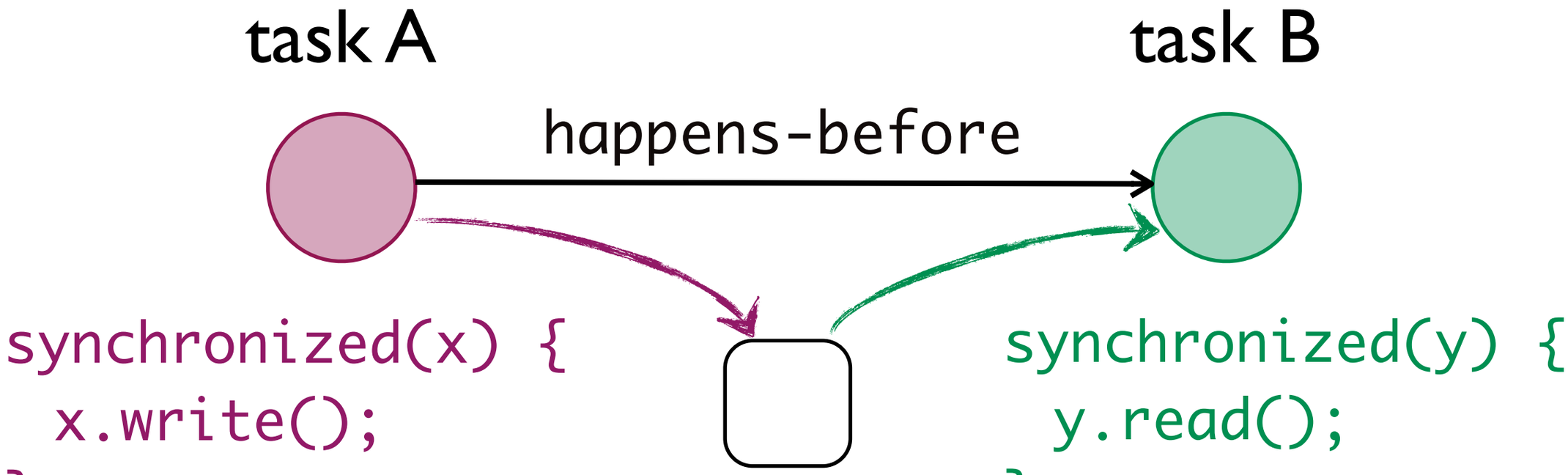
- Can we remove synchronization for x, y?



☒ Aliased

Example

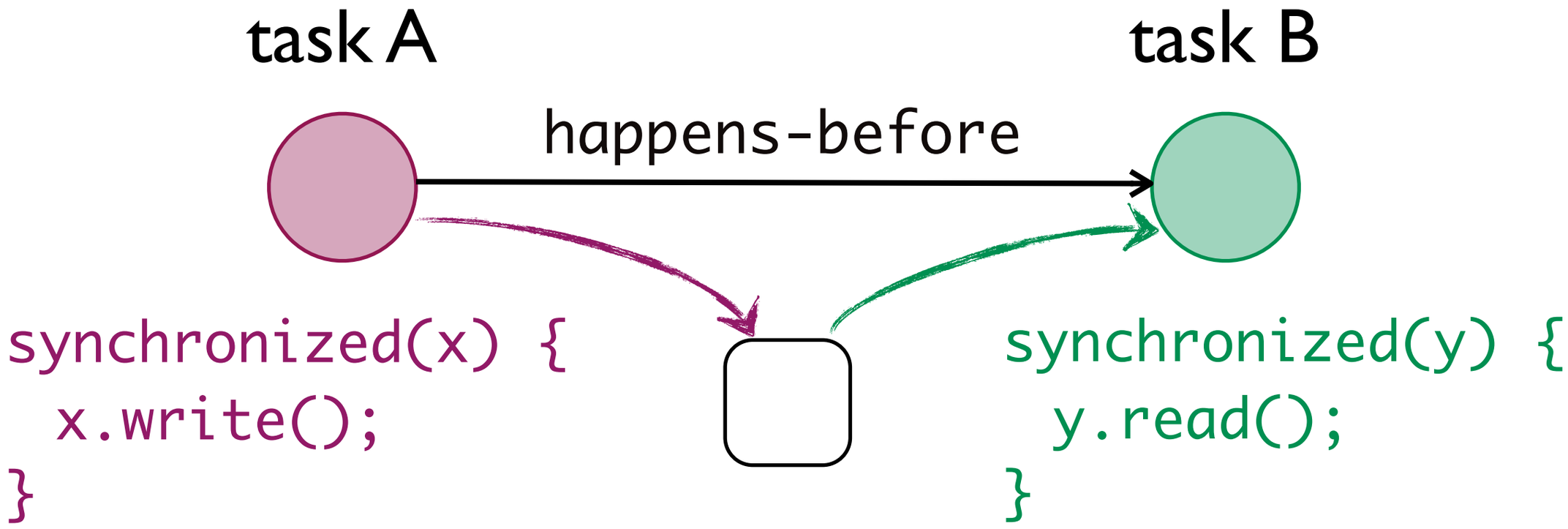
- Can we remove synchronization for x, y?



☒ Aliased

Example

- Can we remove synchronization for x, y?

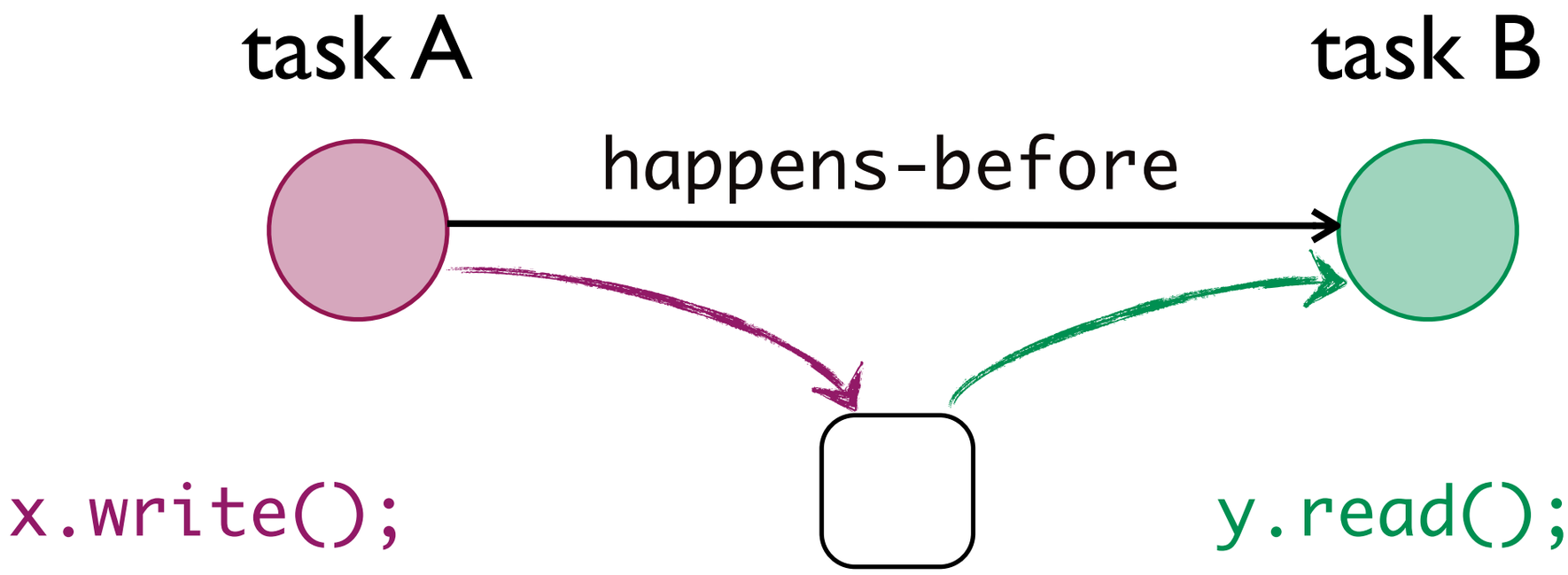


☒ Aliased

☑ Ordered memory access

Example

- Can we remove synchronization for x, y?



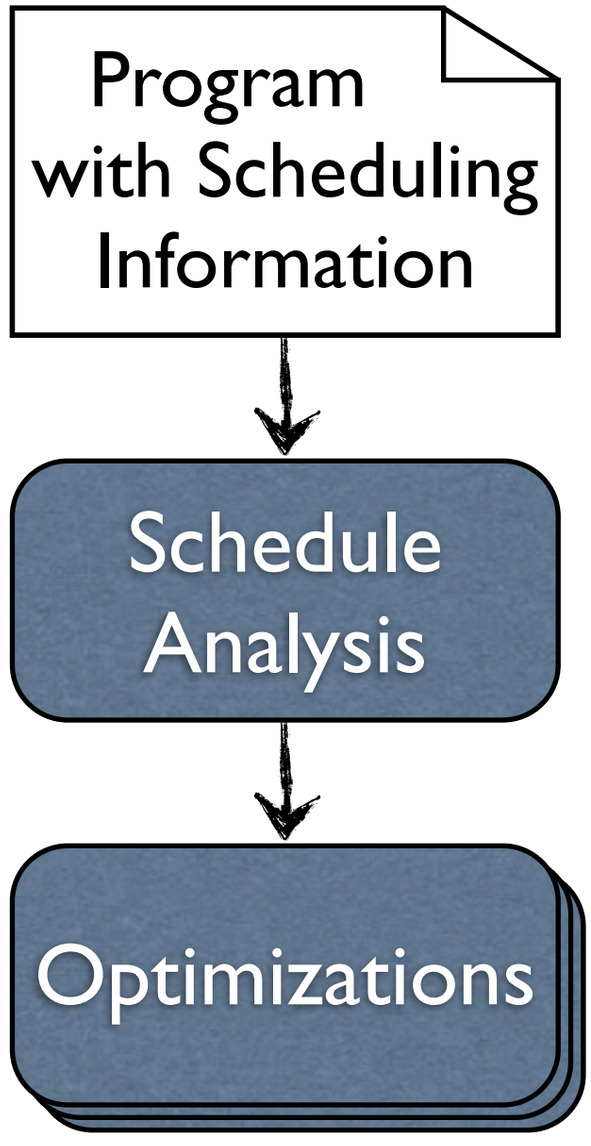
☒ Aliased

☑ Ordered memory access

Motivation

- Compilers profit from static knowledge about runtime schedules
- Optimizations today must reinvent own analyses
- Our goal: factor out analysis of task schedules
  - Simplification + integration of optimizations
  - Additional knowledge of happens-before relationships increases optimization potential

Schedule Analysis Overview



Outline

- Motivation
- Explicit Scheduling
- Genuine Edge Test
- Schedule Analysis
- Concluding Remarks

Explicit Scheduling Model

- A program representation that:
  - Contains explicit scheduling information
  - Allows for static reasoning
- General enough for *structured* (fork/join, Cilk, OpenMP) and *unstructured* parallelism (threads)
- Pre-processing step transforms traditional programs into programs with explicit scheduling

Explicit Scheduling

Explicit Scheduling

- A `task` method is similar to a regular method:
  - code that is executed in the context of `this`

Explicit Scheduling

- A `task` method is similar to a regular method:
  - code that is executed in the context of `this`
- Instead of *invoking* a task method, one *schedules* it for later execution:

```
Activation b = schedule obj.bar(42);
```

Explicit Scheduling

- A `task` method is similar to a regular method:
  - code that is executed in the context of `this`
- Instead of *invoking* a task method, one *schedules* it for later execution:

```
Activation b = schedule obj.bar(42);
```

- Keyword now references currently executing activation

Explicit Scheduling (2)

Explicit Scheduling (2)

- →-statement creates explicit happens-before relationship:

```
a → b;
```

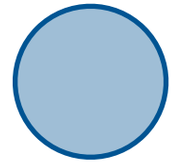
Explicit Scheduling (2)

- →-statement creates explicit happens-before relationship:

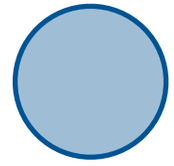
```
a → b;
```

- Implicit happens-before relationship between scheduling task and scheduled task
  - Gives scheduling task time to add happens-before relationships

```
class MyClass {
  task doWrite() {...}
  task doRead() {...}
  task doCompute() {
    Activation write = schedule doWrite();
    Activation read = schedule doRead();
    write → read;
  }
}
```

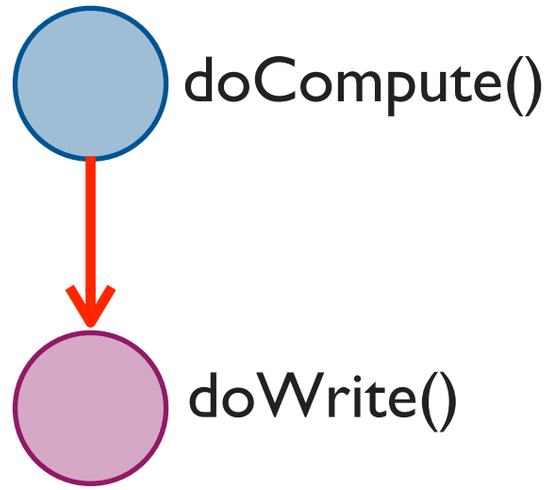



```
class MyClass {
  task doWrite() {...}
  task doRead() {...}
  task doCompute() {
    Activation write = schedule doWrite();
    Activation read = schedule doRead();
    write → read;
  }
}
```

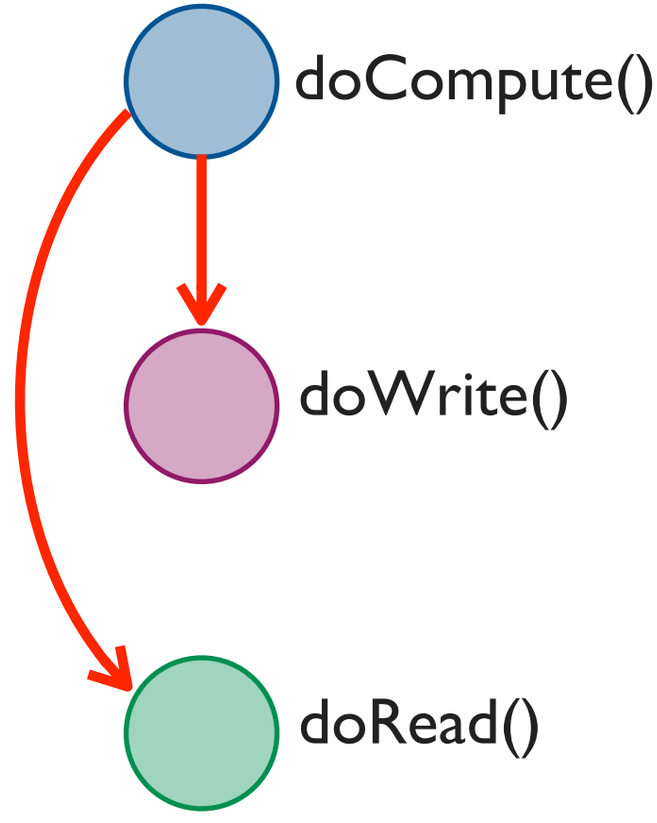



```
class MyClass {
  task doWrite() {...}
  task doRead() {...}
  task doCompute() {
    Activation write = schedule doWrite();
    Activation read = schedule doRead();
    write → read;
  }
}
```

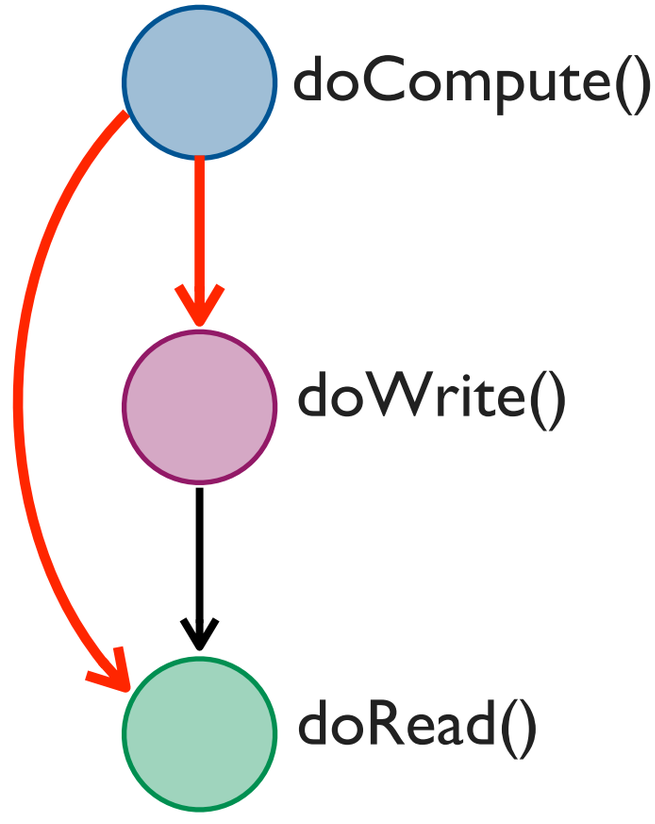
```
class MyClass {
  task doWrite() {...}
  task doRead() {...}
  task doCompute() {
    Activation write = schedule doWrite();
    Activation read = schedule doRead();
    write → read;
  }
}
```



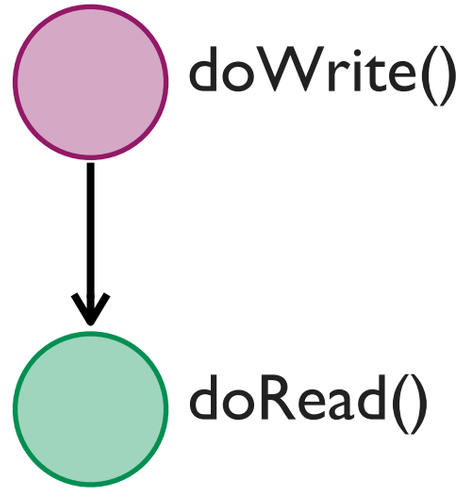
```
class MyClass {
  task doWrite() {...}
  task doRead() {...}
  task doCompute() {
    Activation write = schedule doWrite();
    Activation read = schedule doRead();
    write → read;
  }
}
```

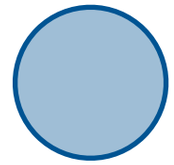


```
class MyClass {
  task doWrite() {...}
  task doRead() {...}
  task doCompute() {
    Activation write = schedule doWrite();
    Activation read = schedule doRead();
    write → read;
  }
}
```



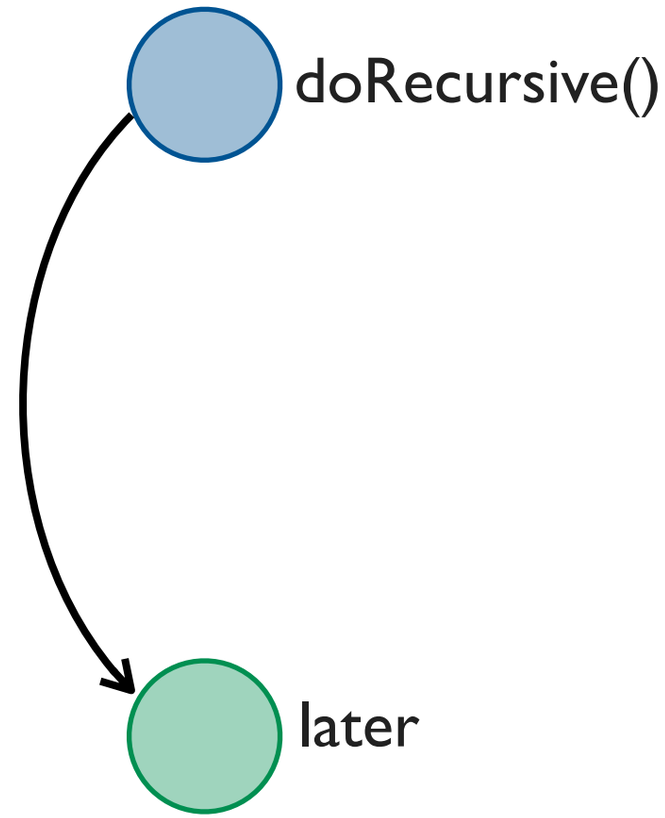
```
class MyClass {
  task doWrite() {...}
  task doRead() {...}
  task doCompute() {
    Activation write = schedule doWrite();
    Activation read = schedule doRead();
    write → read;
  }
}
```



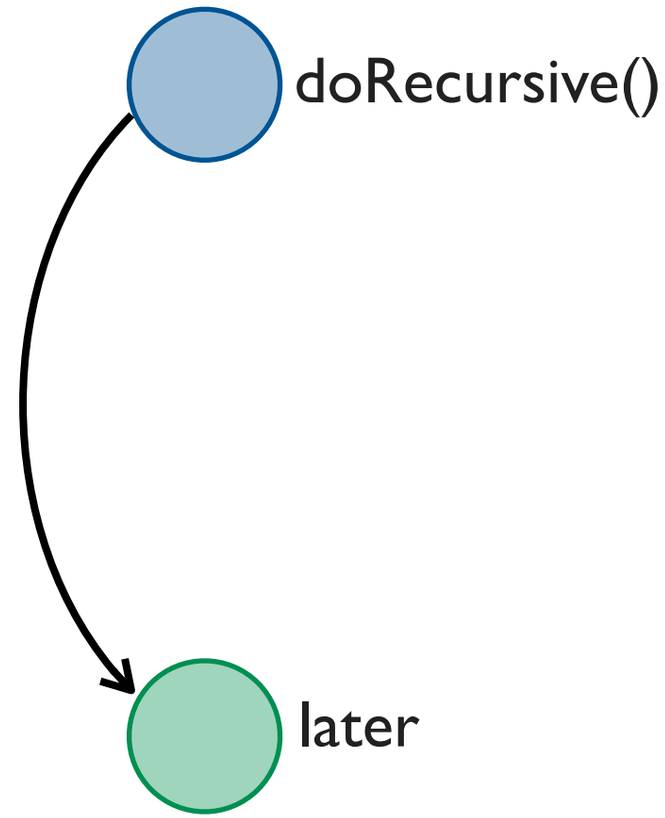



```
class MyClass {
  task doRecursive(Activation later) {
    //perform computation
    if (more()) {
      Activation next =
            schedule doRecursive(later);
      next → later;
    }
  }
}
```

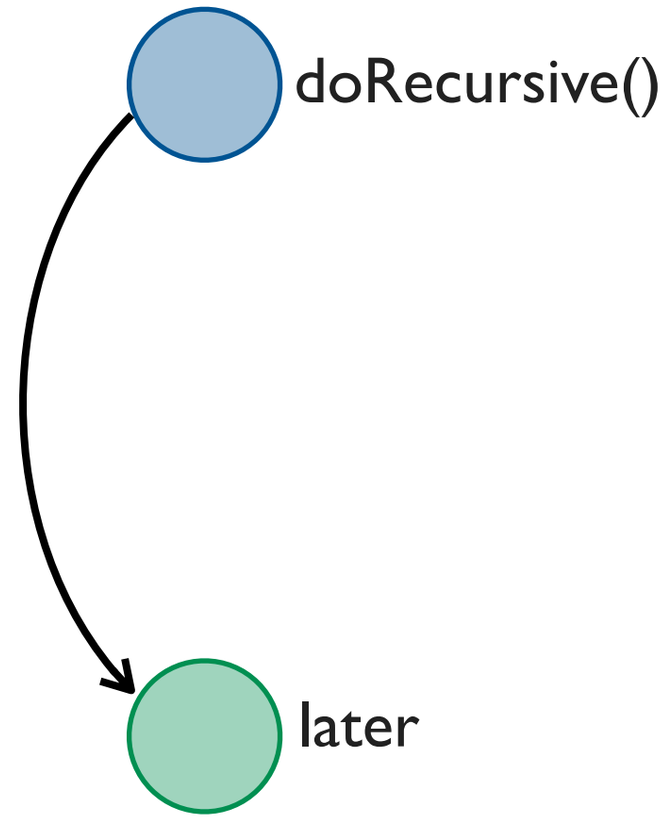
```
class MyClass {
  task doRecursive(Activation later) {
    //perform computation
    if (more()) {
      Activation next =
            schedule doRecursive(later);
      next → later;
    }
  }
}
```



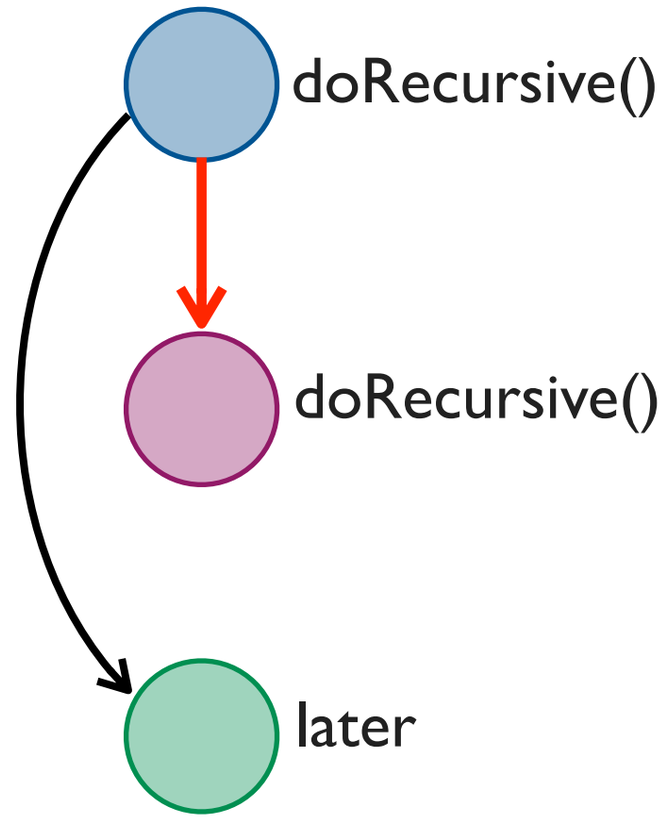
```
class MyClass {
  task doRecursive(Activation later) {
    //perform computation
    if (more()) {
      Activation next =
            schedule doRecursive(later);
      next → later;
    }
  }
}
```



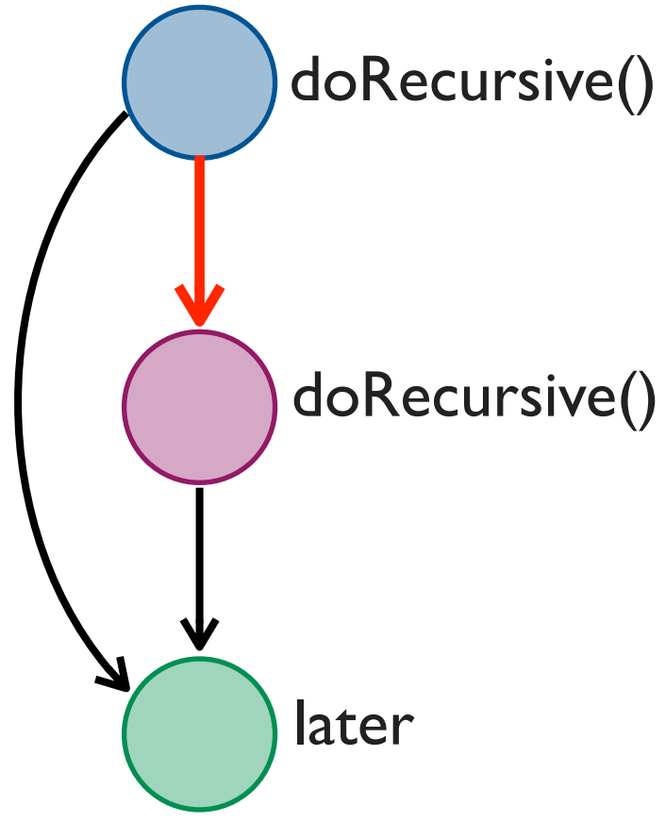
```
class MyClass {
  task doRecursive(Activation later) {
    //perform computation
    if (more()) {
      Activation next =
            schedule doRecursive(later);
      next → later;
    }
  }
}
```



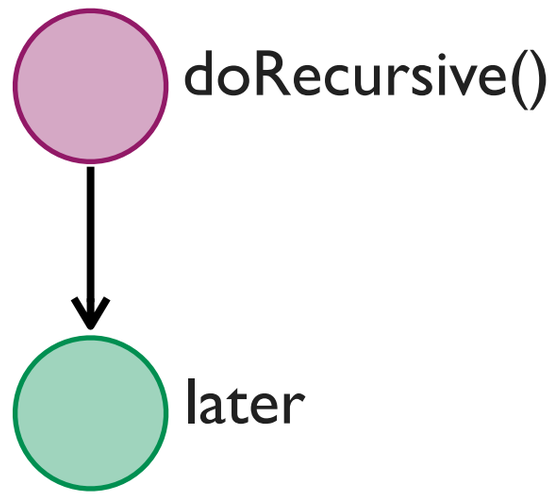
```
class MyClass {
  task doRecursive(Activation later) {
    //perform computation
    if (more()) {
      Activation next =
          schedule doRecursive(later);
      next → later;
    }
  }
}
```



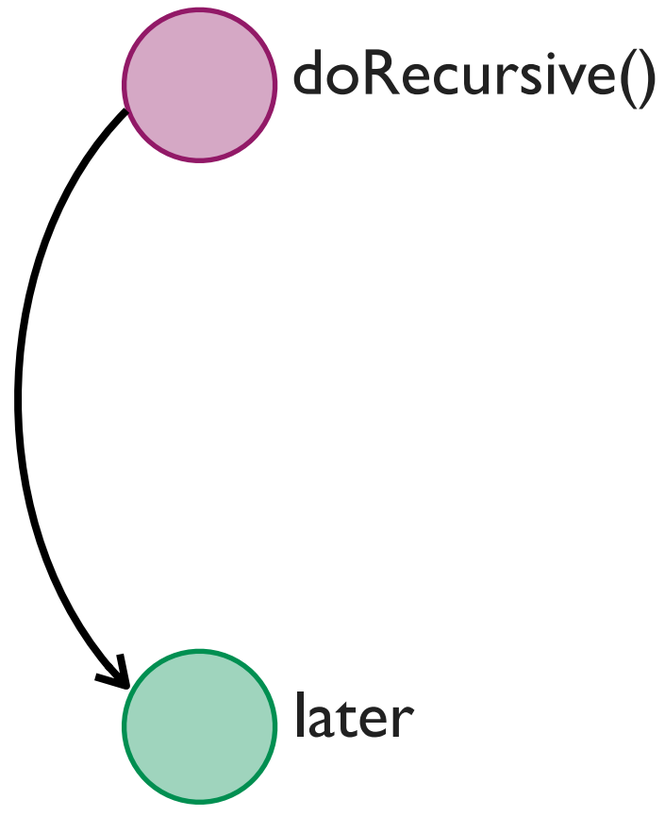
```
class MyClass {
  task doRecursive(Activation later) {
    //perform computation
    if (more()) {
      Activation next =
            schedule doRecursive(later);
      next → later;
    }
  }
}
```



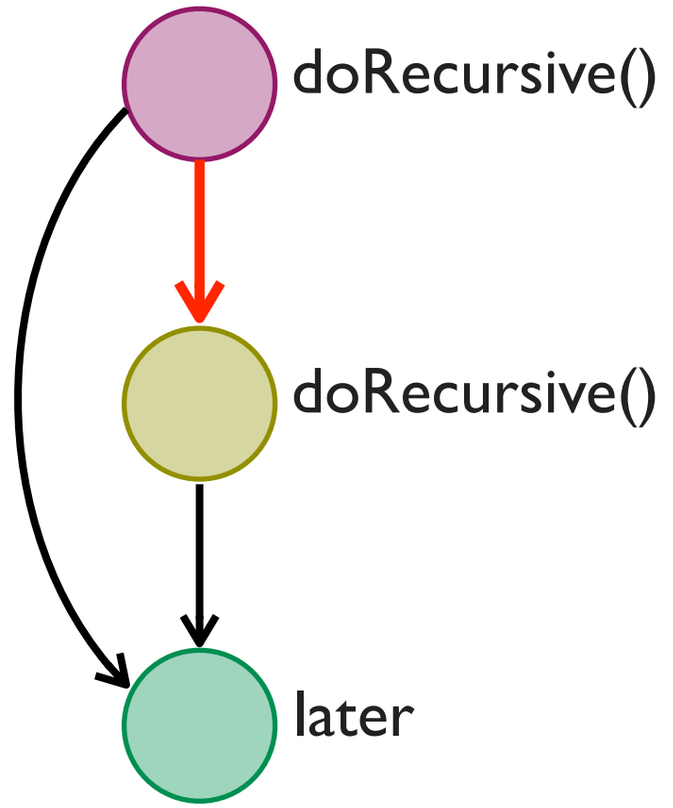
```
class MyClass {
  task doRecursive(Activation later) {
    //perform computation
    if (more()) {
      Activation next =
            schedule doRecursive(later);
      next → later;
    }
  }
}
```



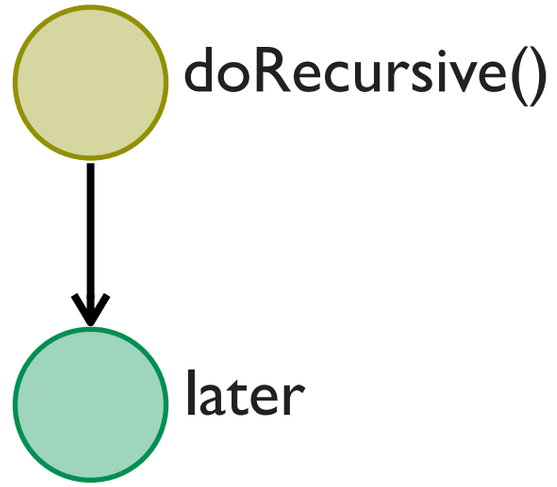
```
class MyClass {
  task doRecursive(Activation later) {
    //perform computation
    if (more()) {
      Activation next =
            schedule doRecursive(later);
      next → later;
    }
  }
}
```



```
class MyClass {
  task doRecursive(Activation later) {
    //perform computation
    if (more()) {
      Activation next =
            schedule doRecursive(later);
      next → later;
    }
  }
}
```



```
class MyClass {
  task doRecursive(Activation later) {
    //perform computation
    if (more()) {
      Activation next =
            schedule doRecursive(later);
      next → later;
    }
  }
}
```



```
class MyClass {
  task doRecursive(Activation later) {
    //perform computation
    if (more()) {
      Activation next =
            schedule doRecursive(later);
      next → later;
    }
  }
}
```

Outline

- Motivation
- Explicit Scheduling
- Genuine Edge Test
- Schedule Analysis
- Concluding Remarks

Unreliable Edges

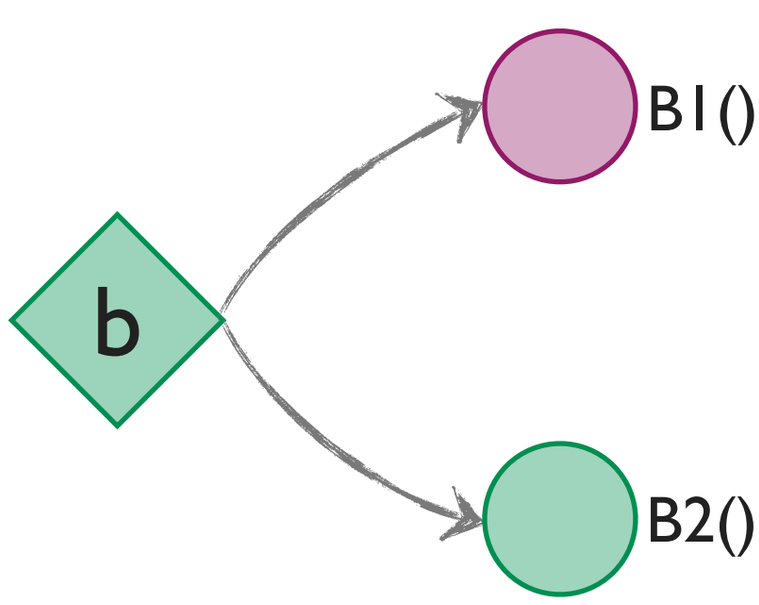
```
task doThings() {
  Activation a = schedule A();
  Activation b = schedule B1();
  if (random) {
    b = schedule B2();
  }
  a → b;
}
```

Unreliable Edges

```
task doThings() {
  Activation a = schedule A();
  Activation b = schedule B1();
  if (random) {
    b = schedule B2();
  }
  a → b;
}
```

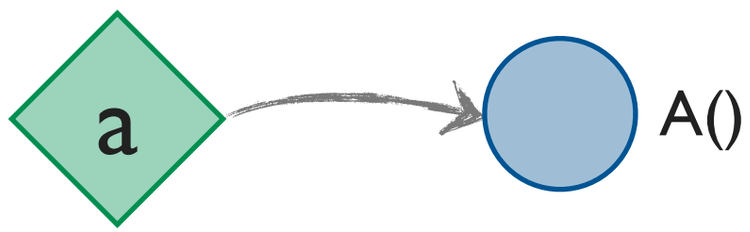
Unreliable Edges

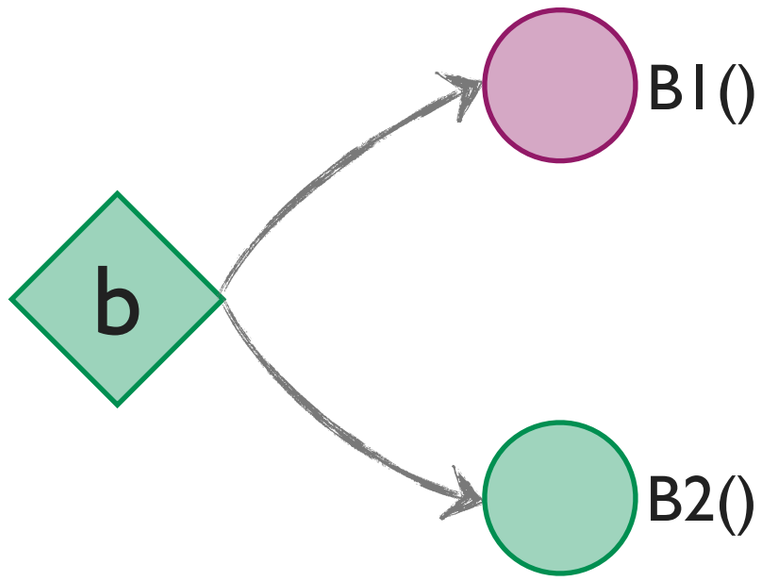
```
task doThings() {
  Activation a = schedule A();
  Activation b = schedule B1();
  if (random) {
    b = schedule B2();
  }
  a → b;
}
```



Unreliable Edges

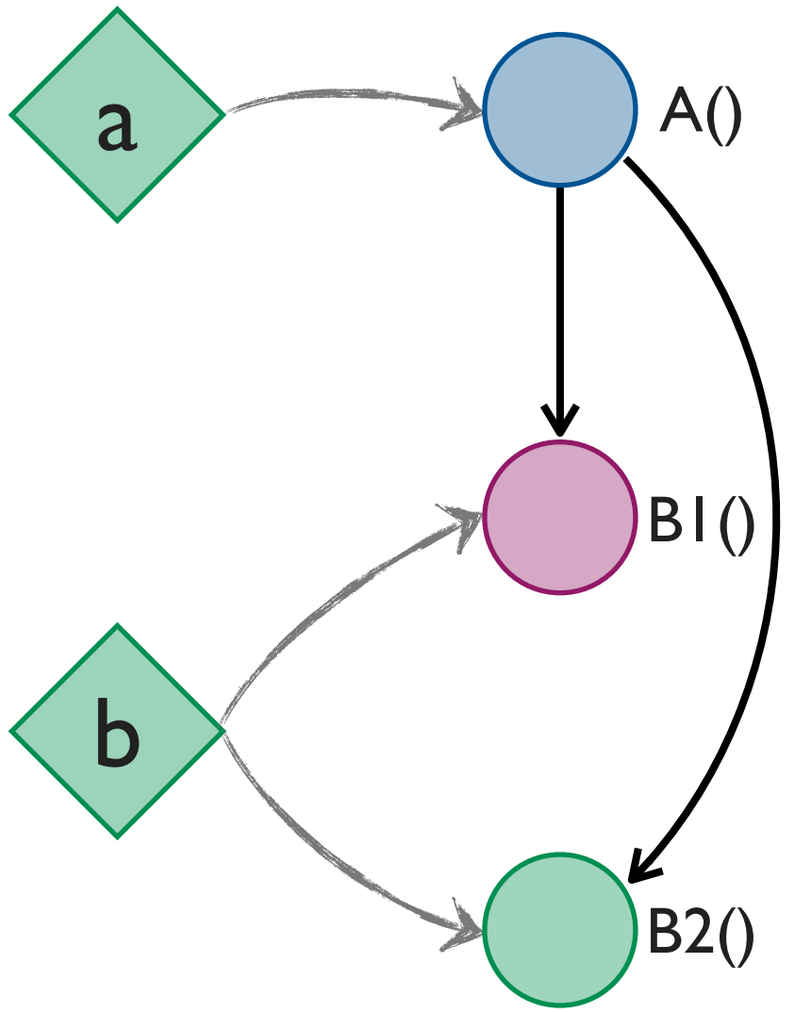
```
task doThings() {
  Activation a = schedule A();
  Activation b = schedule B1();
  if (random) {
    b = schedule B2();
  }
  a → b;
}
```





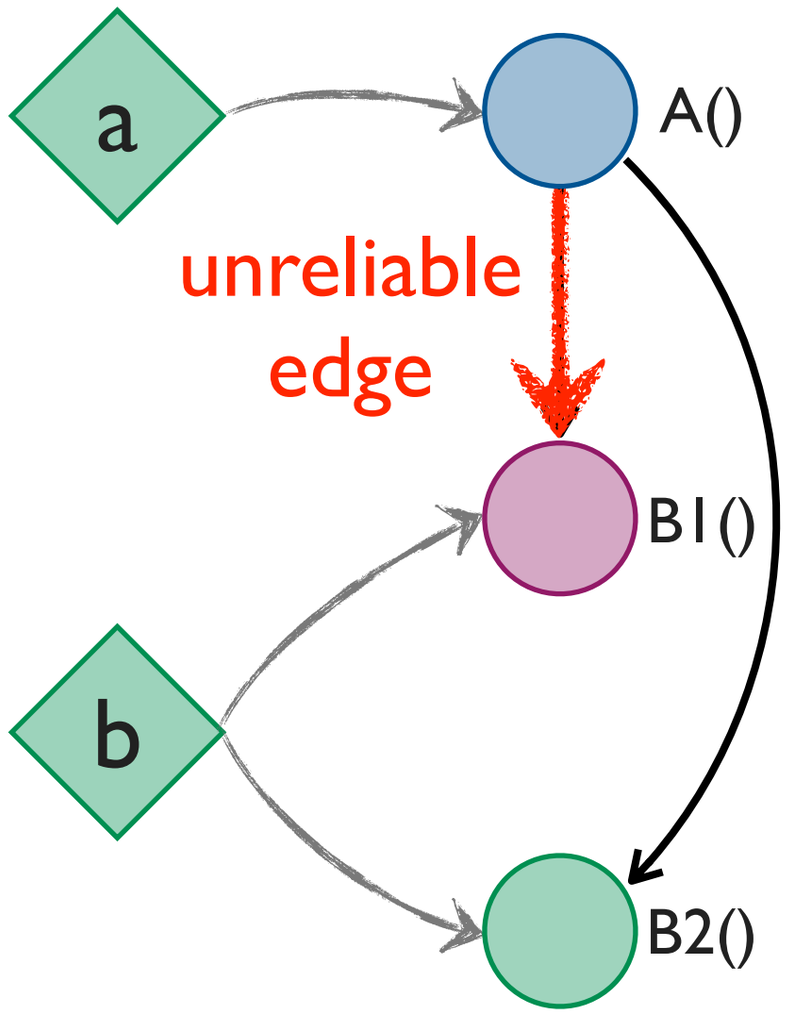
Unreliable Edges

```
task doThings() {
  Activation a = schedule A();
  Activation b = schedule B1();
  if (random) {
    b = schedule B2();
  }
  a → b;
}
```



Unreliable Edges

```
task doThings() {
  Activation a = schedule A();
  Activation b = schedule B1();
  if (random) {
    b = schedule B2();
  }
  a → b;
}
```



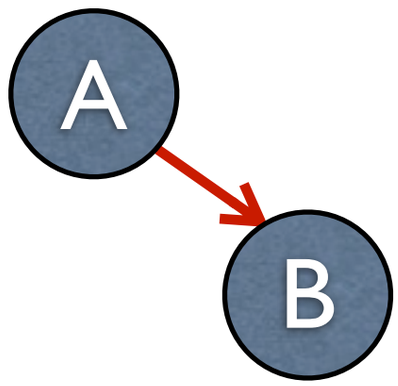
Genuine Edges

- An edge x → y is *genuine* **if activation** of x implies edge has been created

Types of Edges

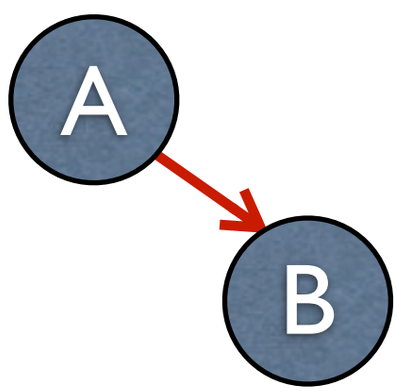
Types of Edges

- *creation edges:* **A schedules B**

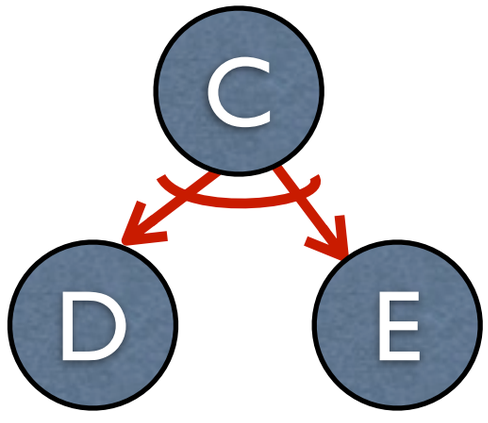


Types of Edges

- *creation edges:* **A schedules B**

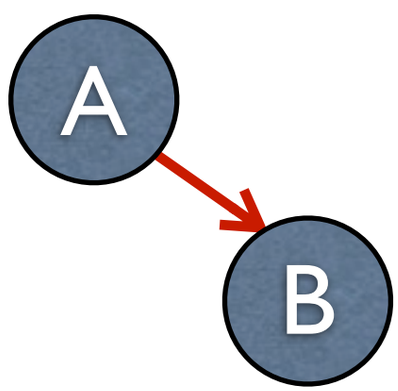


- *exclusive edges:* **C schedules** *either* **D** *or* **E**

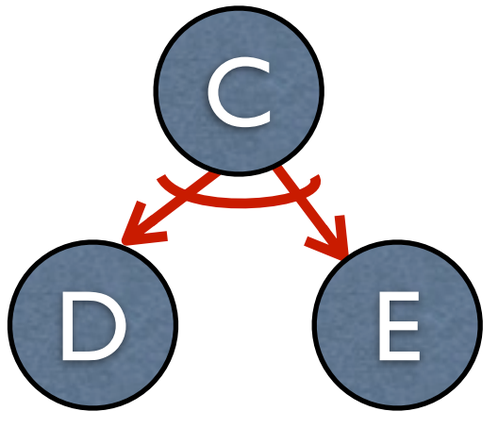


Types of Edges

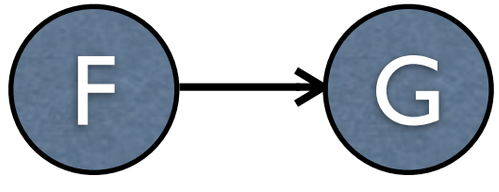
- *creation edges:* **A schedules B**



- *exclusive edges:* **C schedules** *either* **D** *or* **E**



- *explicit →-statement:* **F happens-before G**



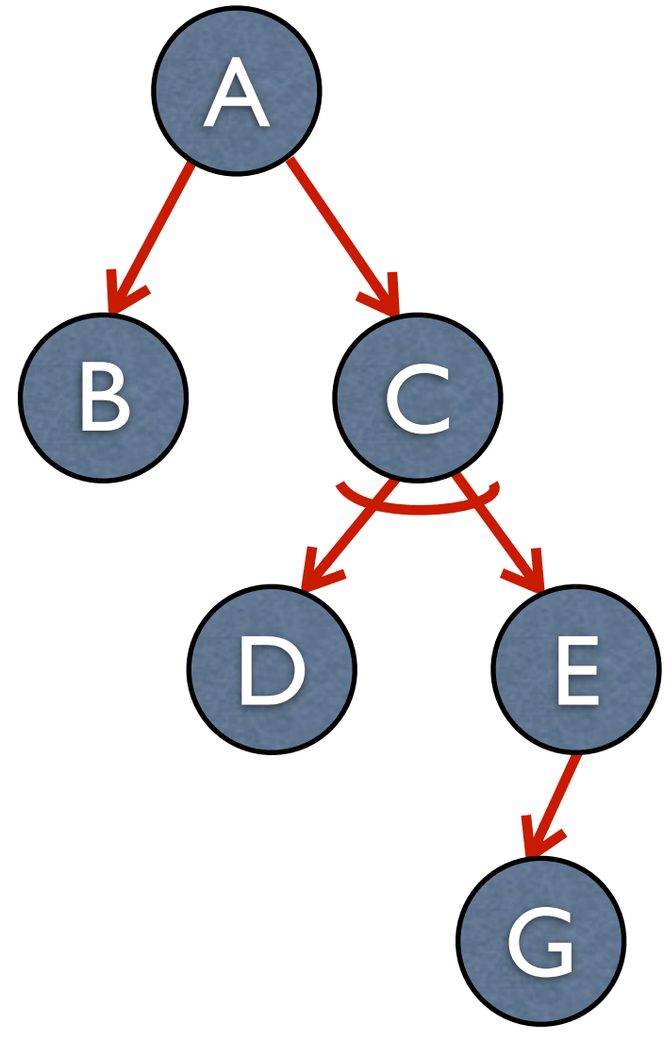
Creation Tree

Creation Tree

- Every activation is created by exactly one parent activation

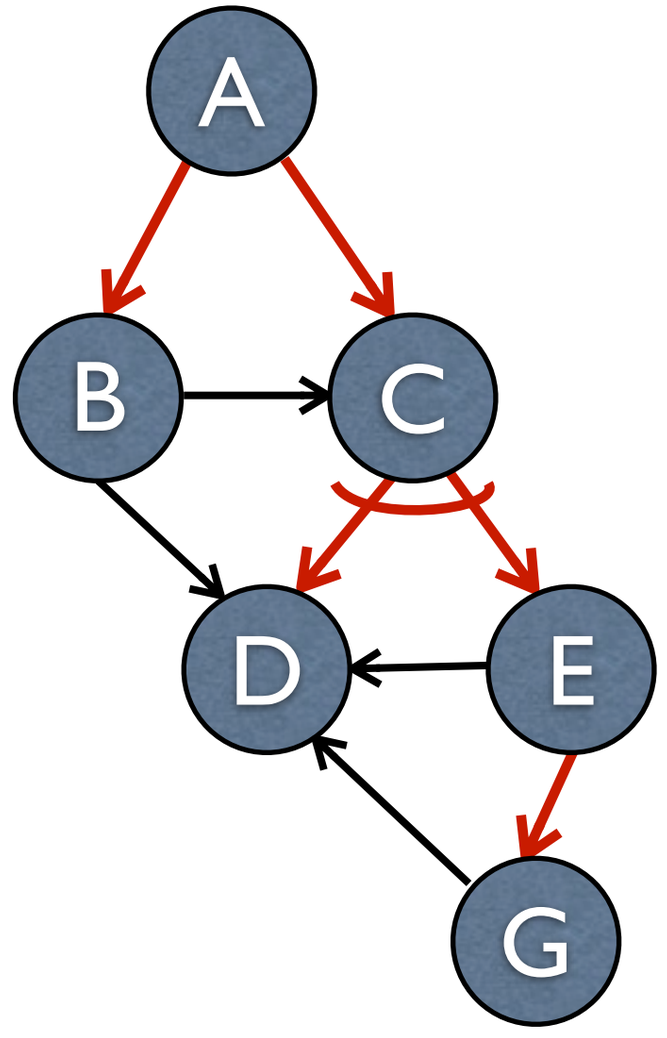
Creation Tree

- Every activation is created by exactly one parent activation



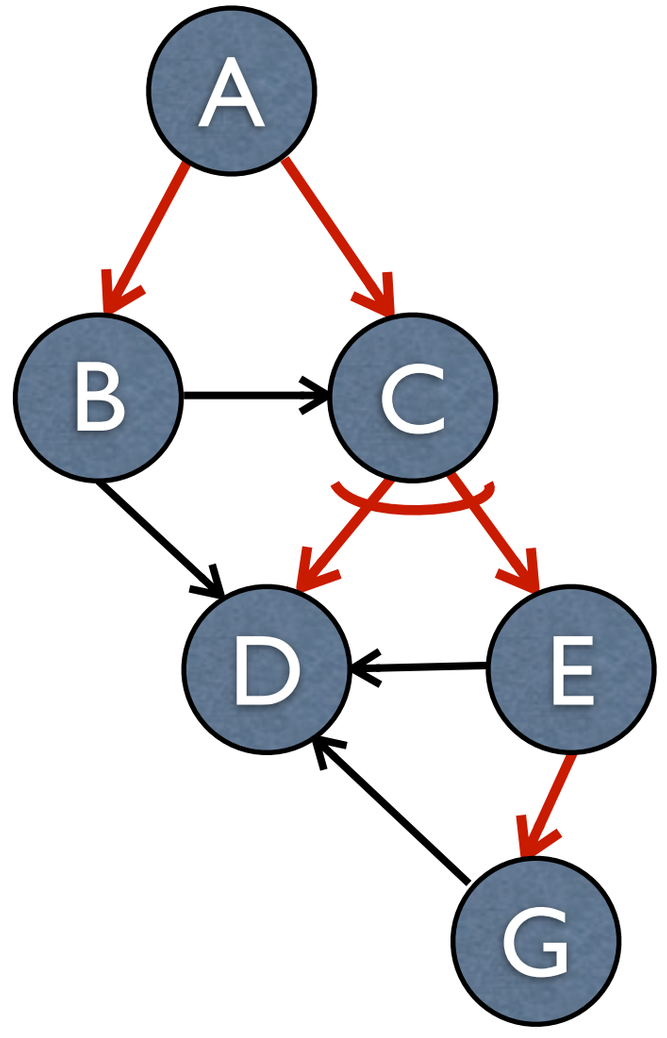
Creation Tree

- Every activation is created by exactly one parent activation



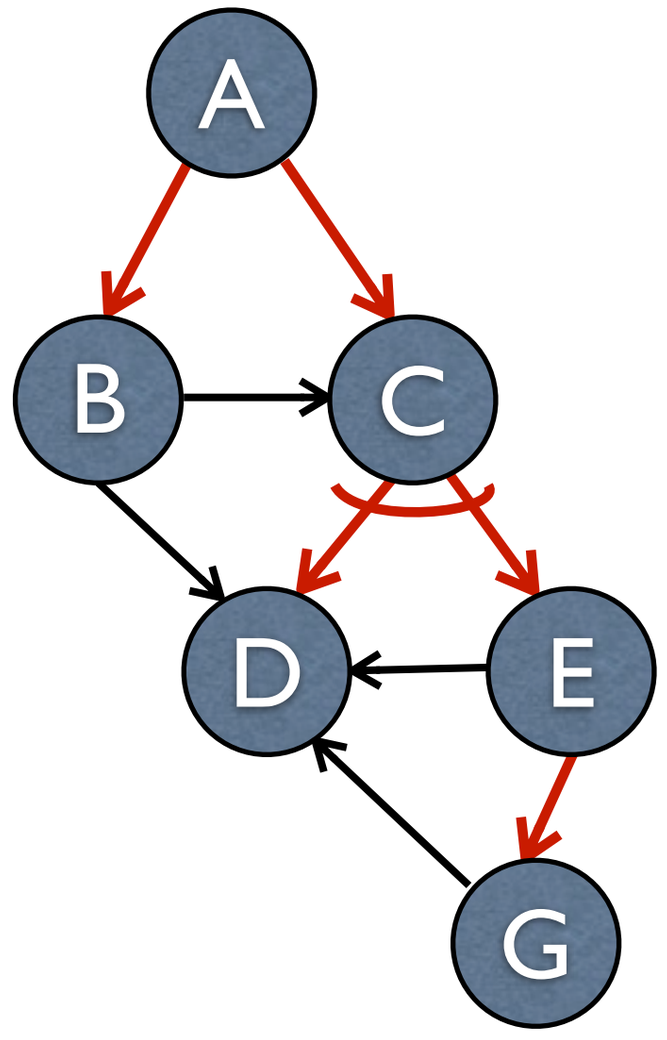
Creation Tree

- Every activation is created by exactly one parent activation
- Creation Tree: a *spanning tree* embedded in schedule

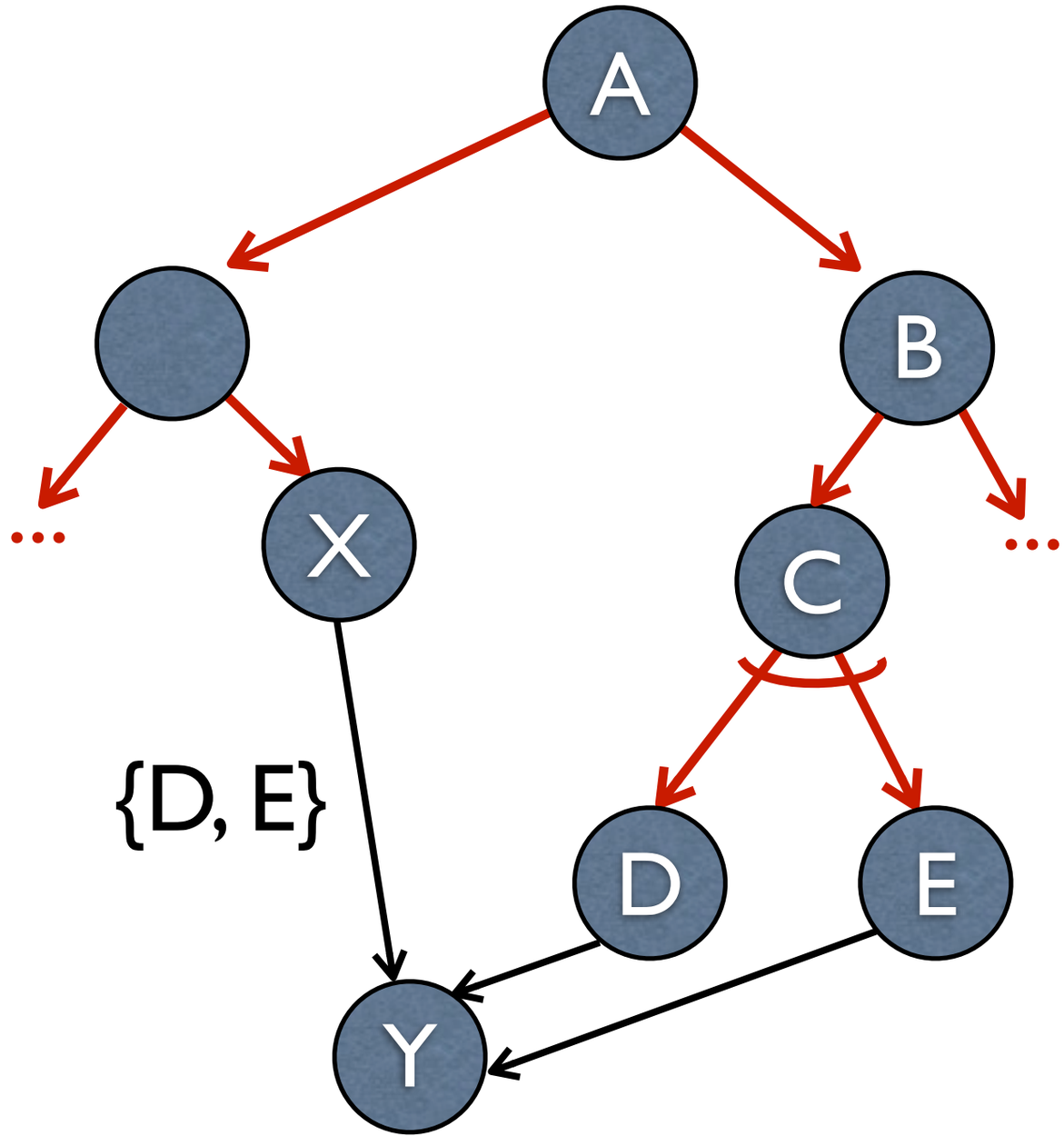


Creation Tree

- Every activation is created by exactly one parent activation
- Creation Tree: a *spanning tree* embedded in schedule
- Useful property:
  - Execution of child implies the completion of parent
  - Creation edges are *genuine*

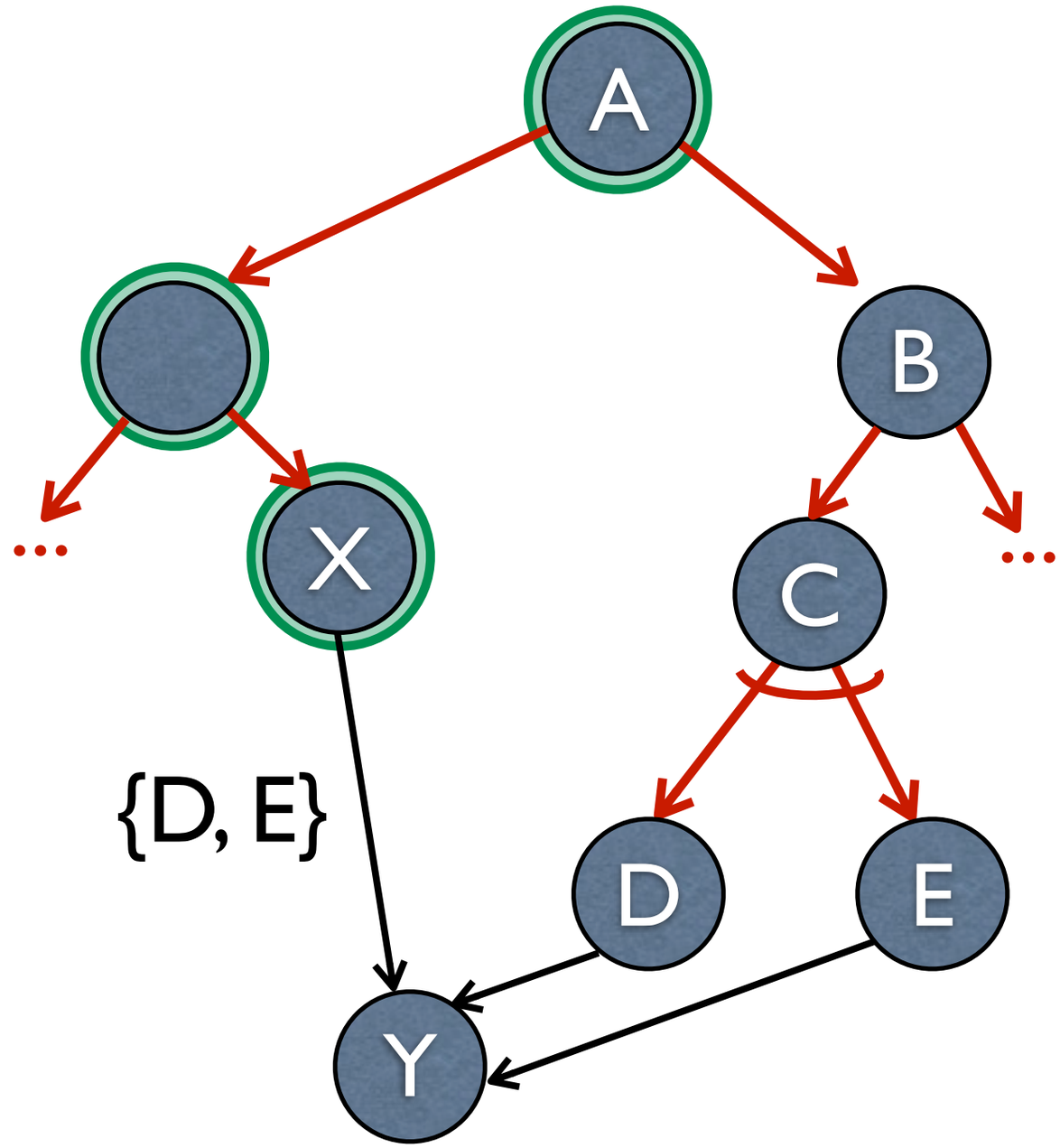


Computing Genuine Edges



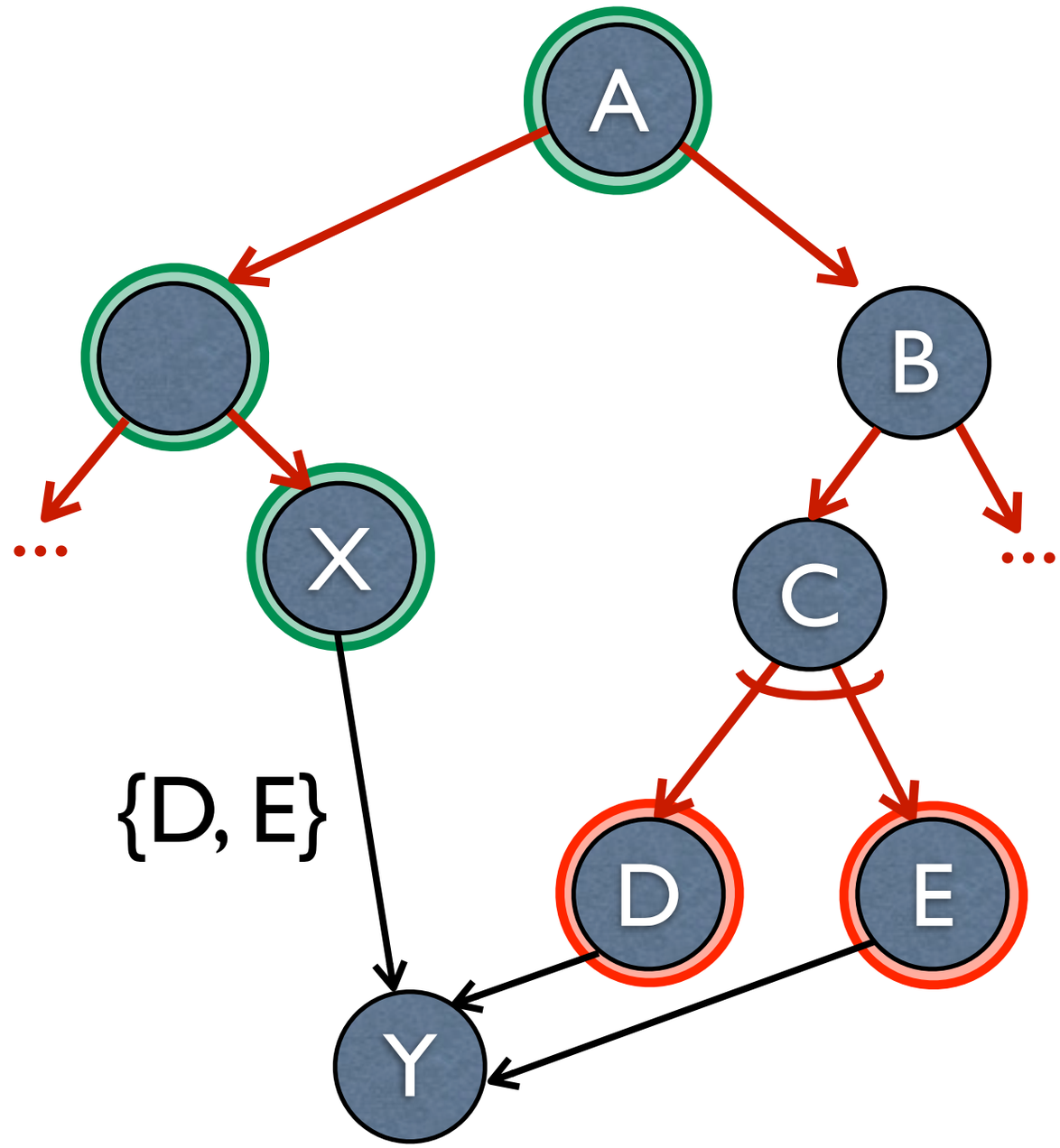
Computing Genuine Edges

1. Mark all parents of x ("the fence")



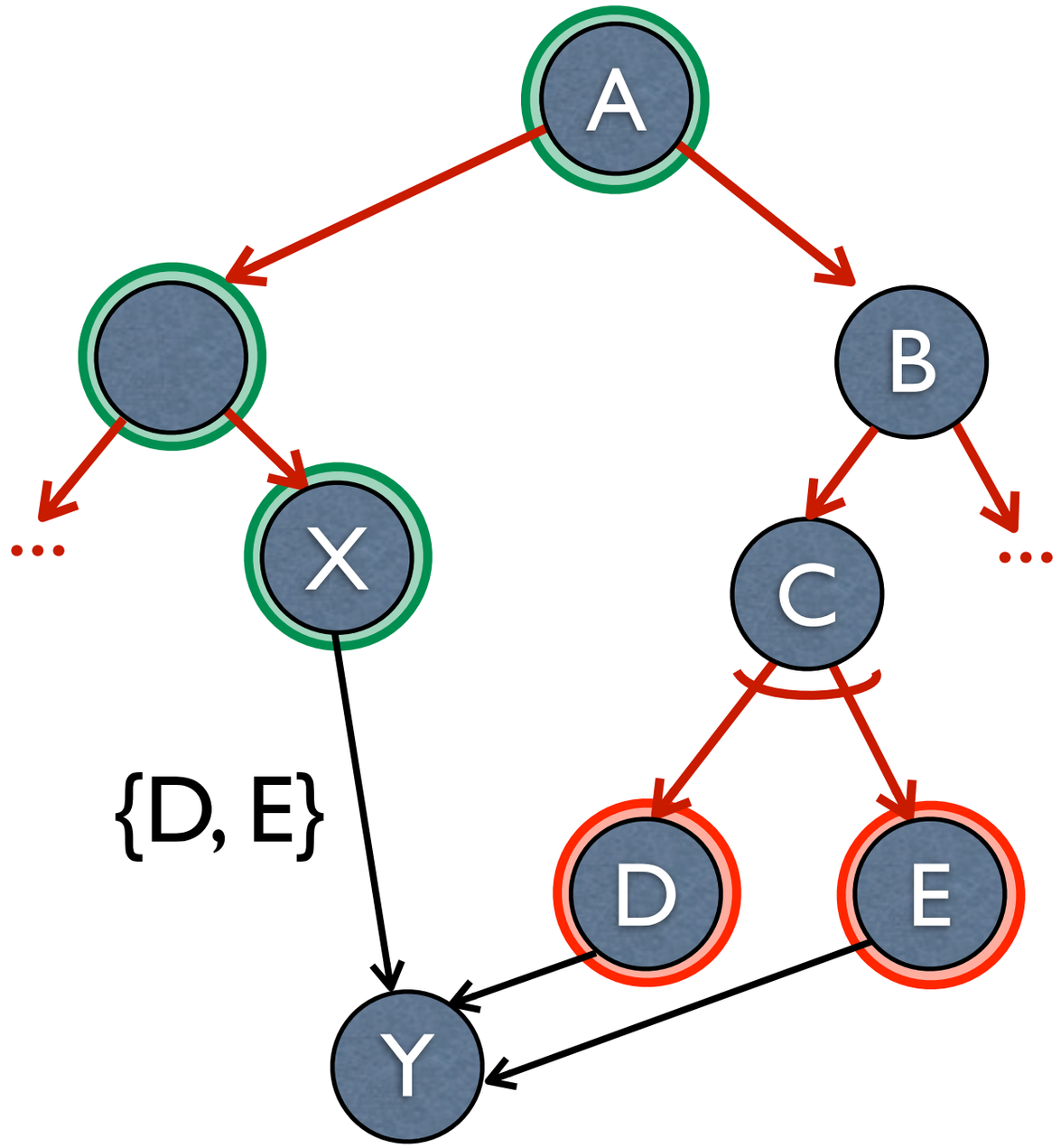
Computing Genuine Edges

1. Mark all parents of x ("the fence")
2. Mark edge creators



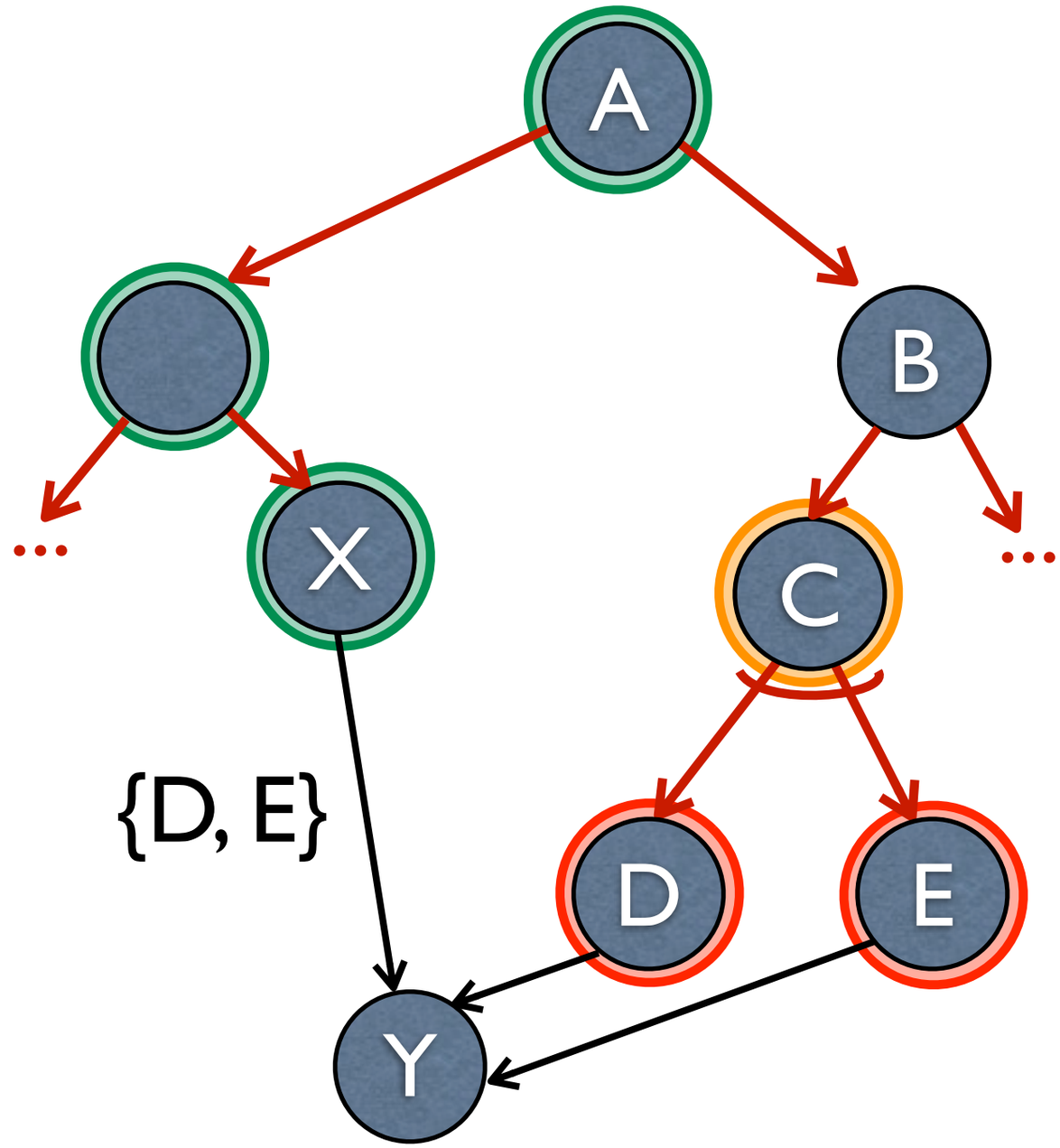
Computing Genuine Edges

1. Mark all parents of x ("the fence")
2. Mark edge creators
3. Mark parent nodes



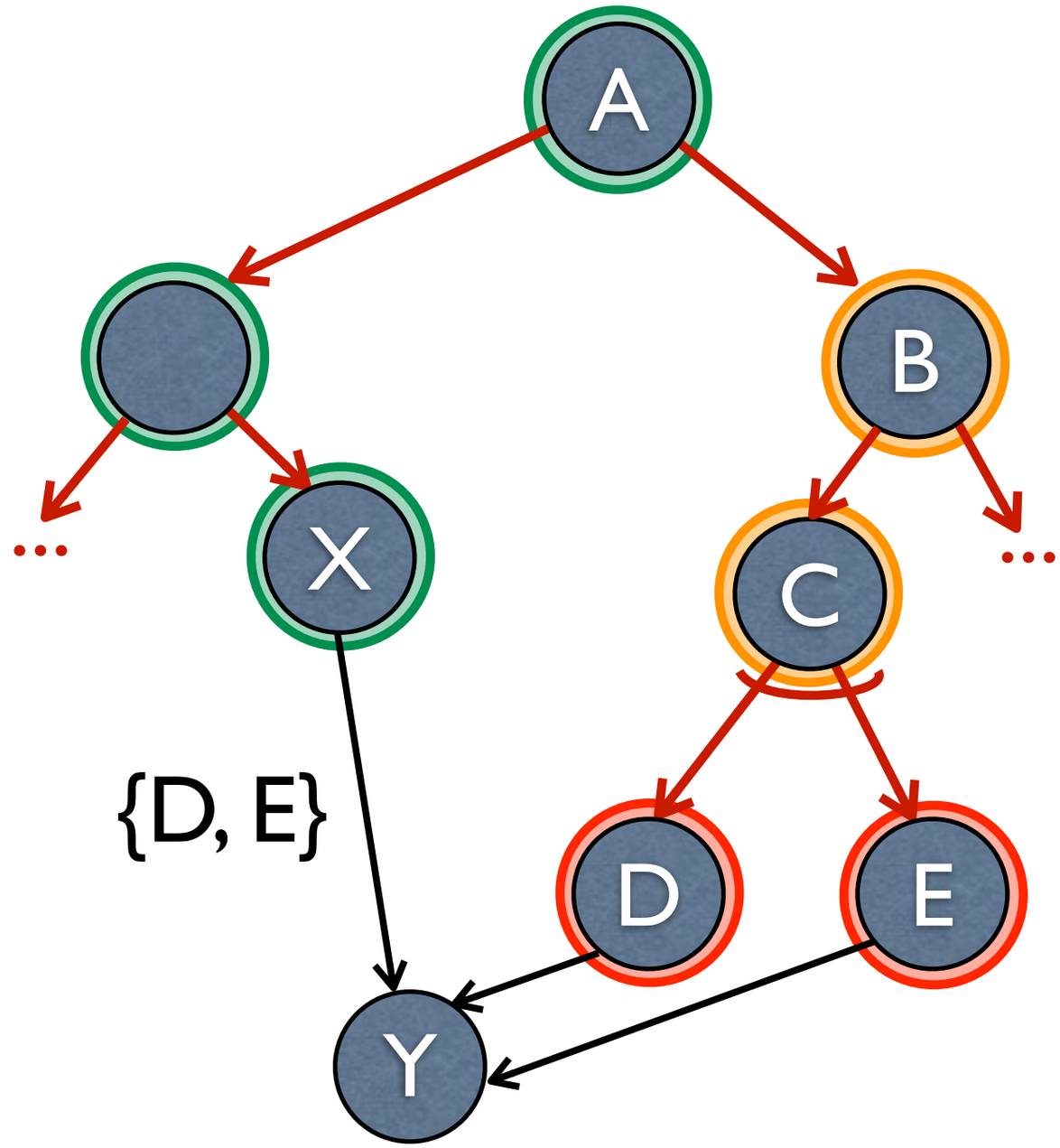
Computing Genuine Edges

1. Mark all parents of x ("the fence")
2. Mark edge creators
3. Mark parent nodes



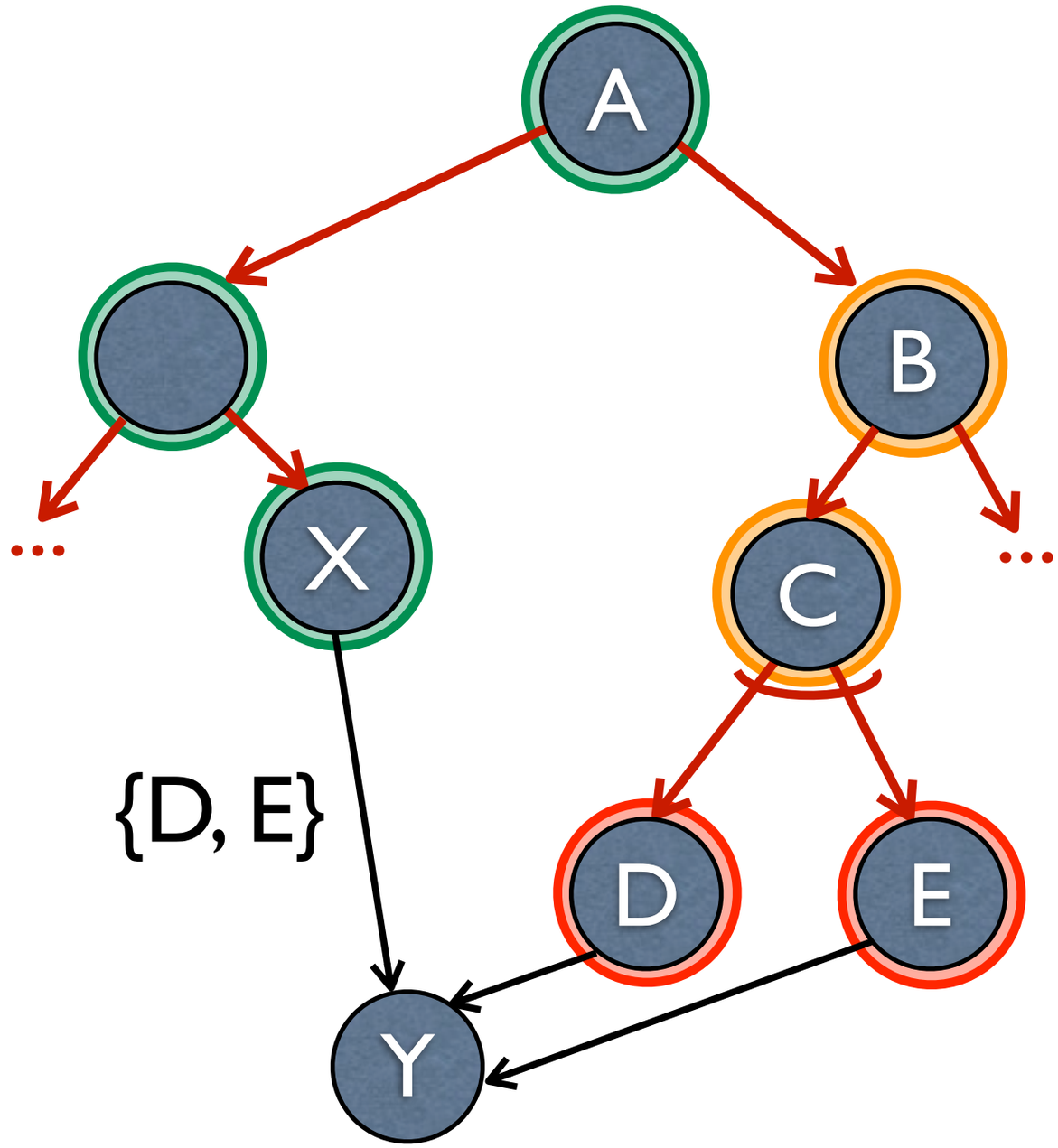
Computing Genuine Edges

1. Mark all parents of x ("the fence")
2. Mark edge creators
3. Mark parent nodes



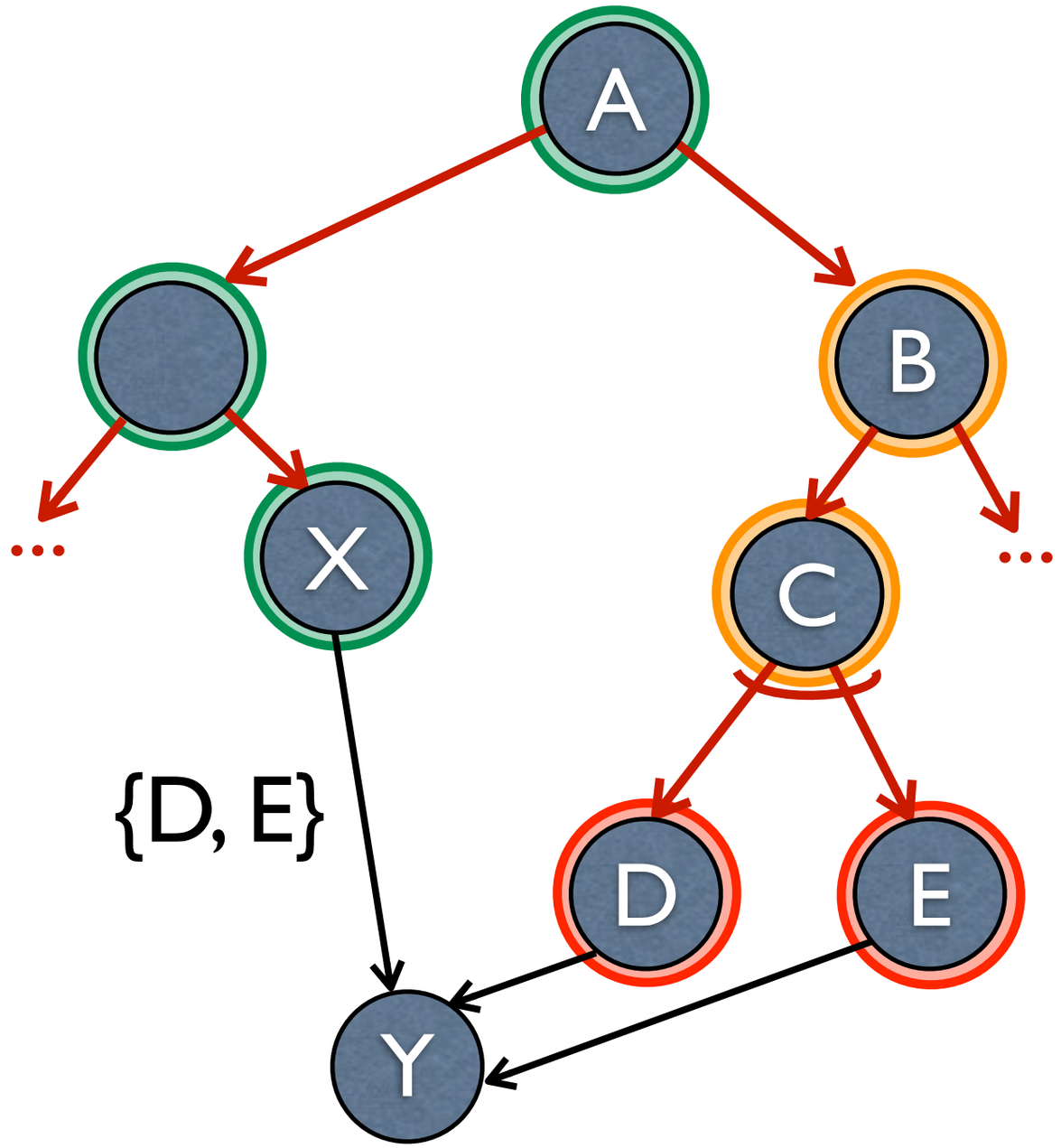
Computing Genuine Edges

1. Mark all parents of x ("the fence")
2. Mark edge creators
3. Mark parent nodes
4. Stop when a parent of x is touched



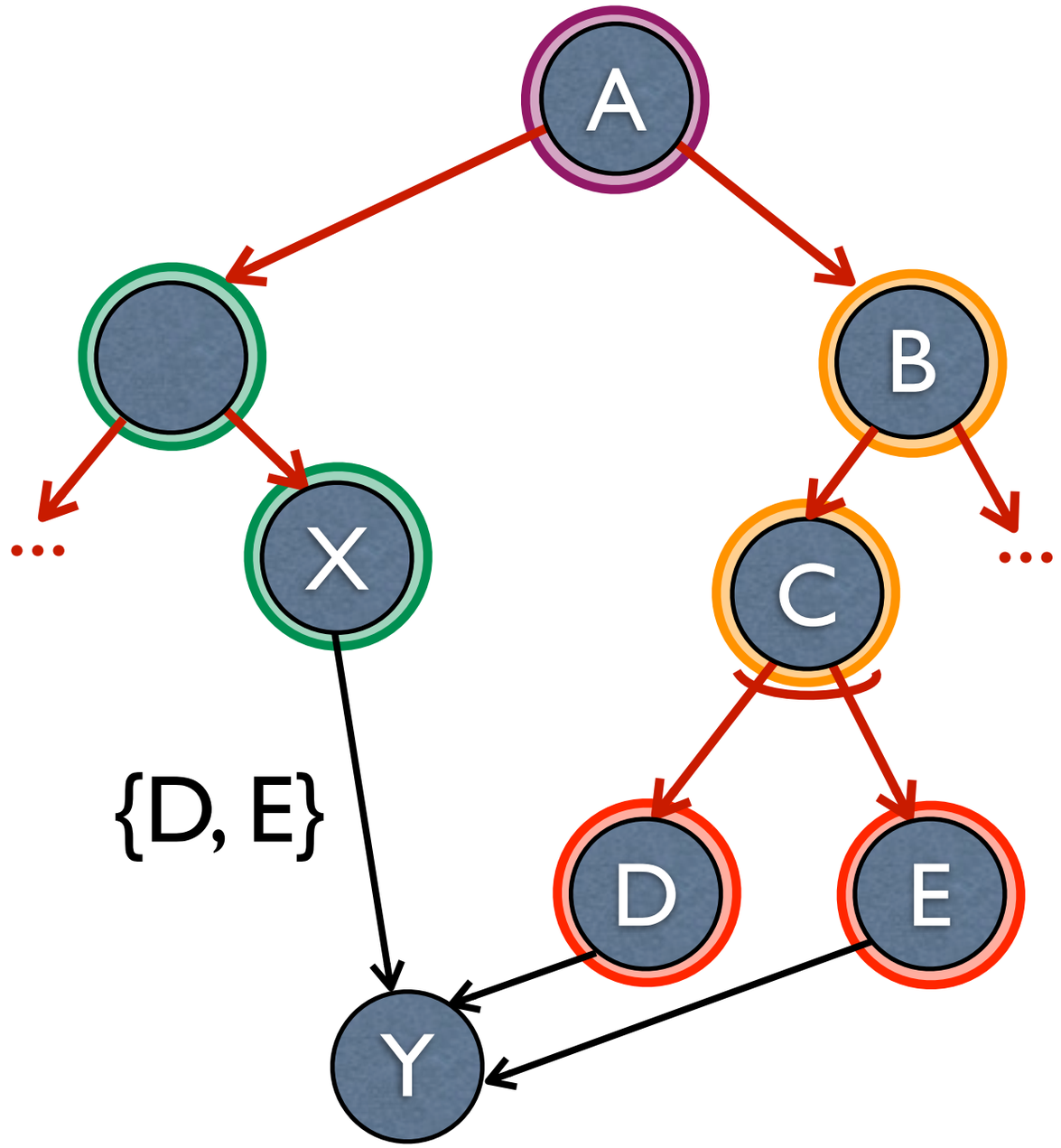
Computing Genuine Edges

1. Mark all parents of x ("the fence")
2. Mark edge creators
3. Mark parent nodes
4. Stop when a parent of x is touched
5. Else: edge not genuine



Computing Genuine Edges

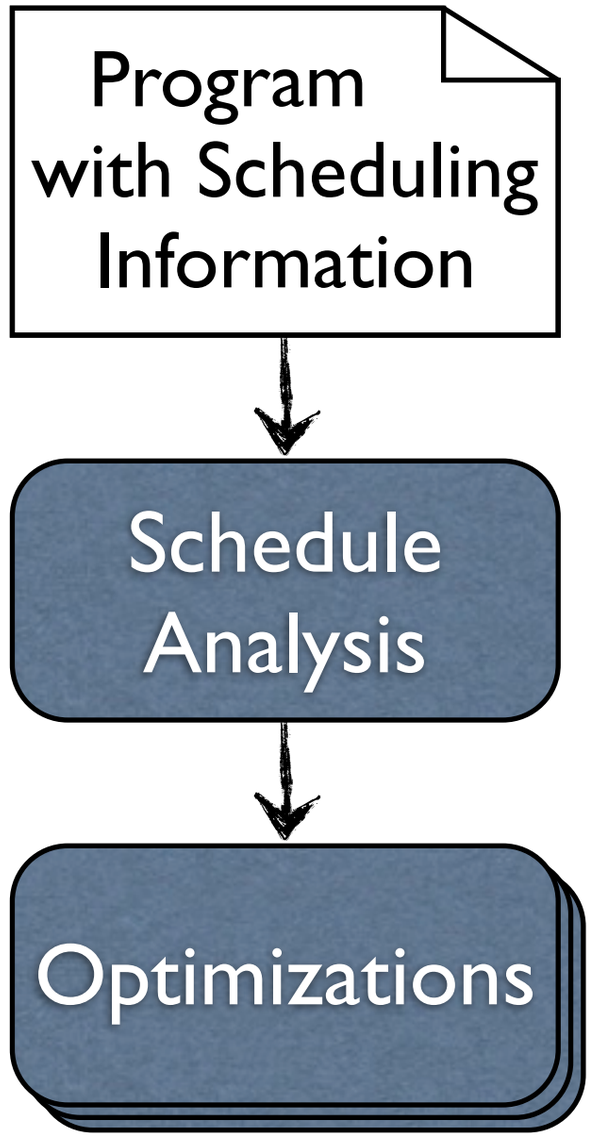
1. Mark all parents of x ("the fence")
2. Mark edge creators
3. Mark parent nodes
4. Stop when a parent of x is touched
5. Else: edge not genuine



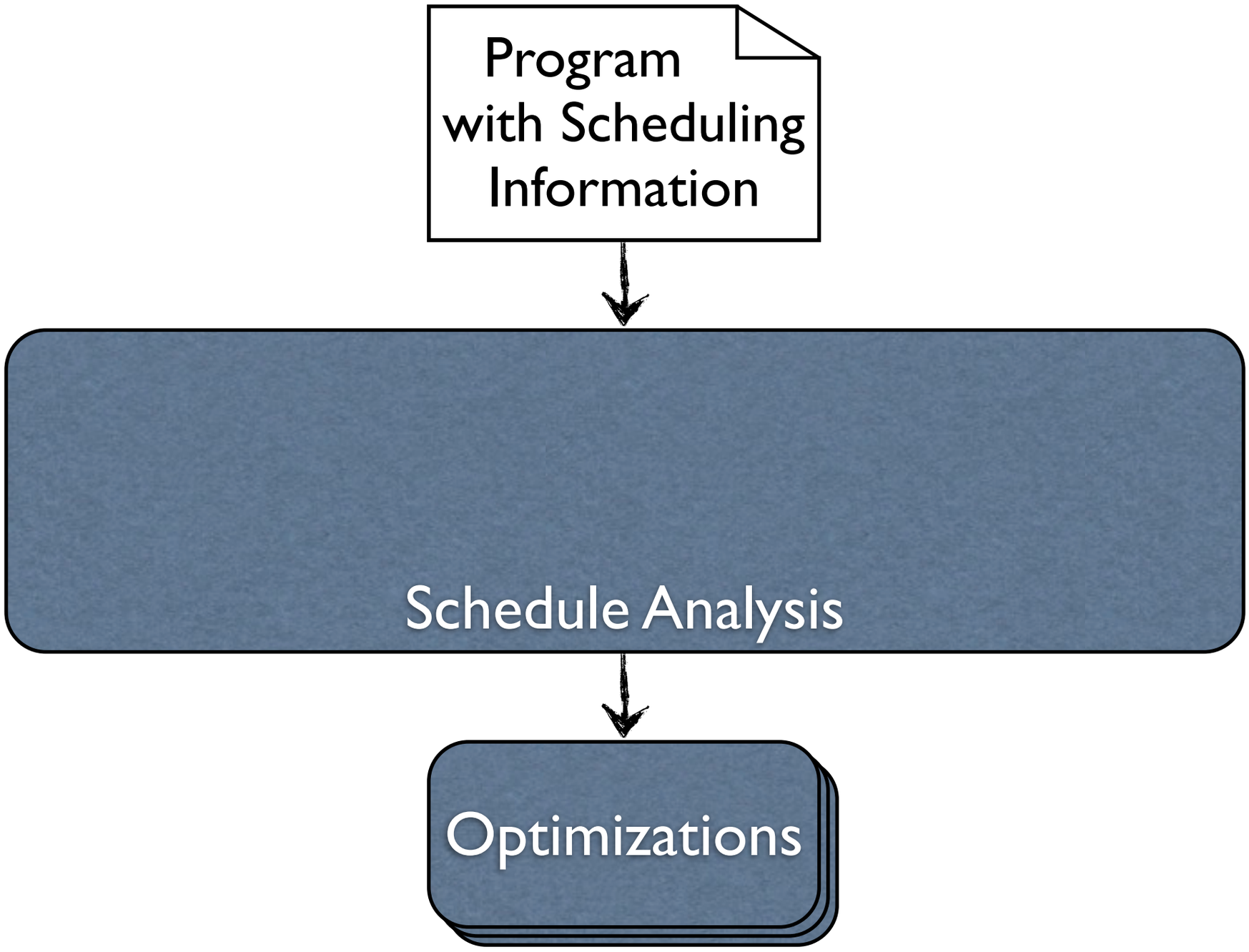
Outline

- Motivation
- Explicit Scheduling
- Genuine Edge Test
- Schedule Analysis
- Concluding Remarks

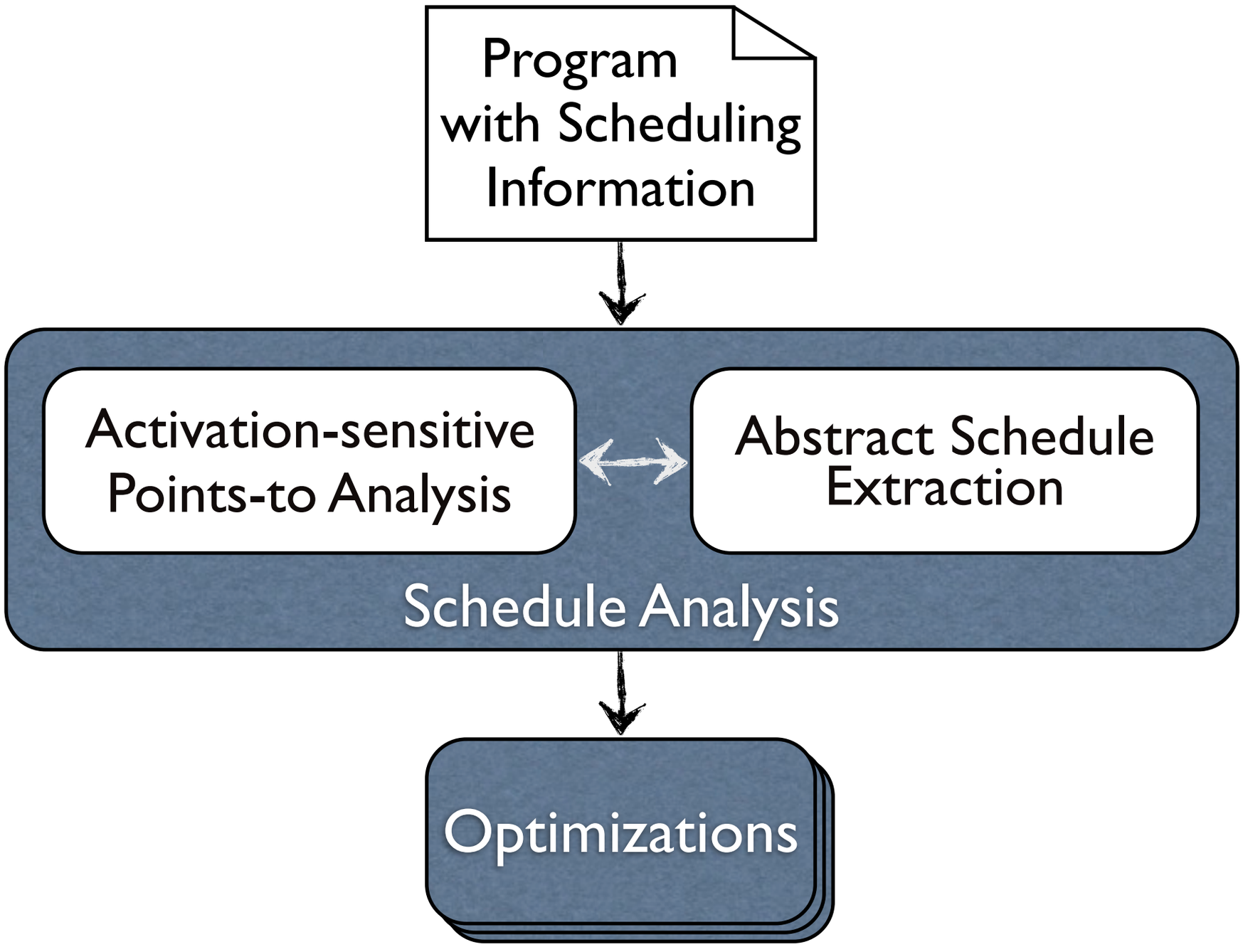
Schedule Analysis Overview



Schedule Analysis Overview



Schedule Analysis Overview



Points-to Analysis

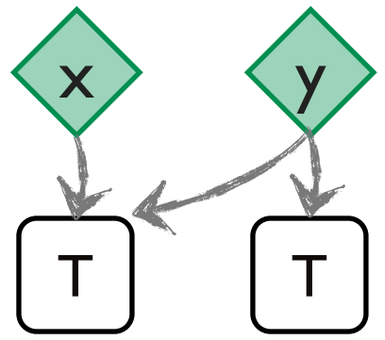
- Computes points-to sets for each program variable
- Activation-sensitive
- →-statements handled in Schedule Extraction phase
- Treats `schedule` statements as method calls
  - Parameters are bound at schedule-time
  - Flow-insensitive with respect to calls

Schedule Extraction

```
task A() {
  T x = new T();
  T y = random ? new T() : x;

  Activation b = schedule B(x);
  Activation c = schedule C(y);

  b → c;
}
```

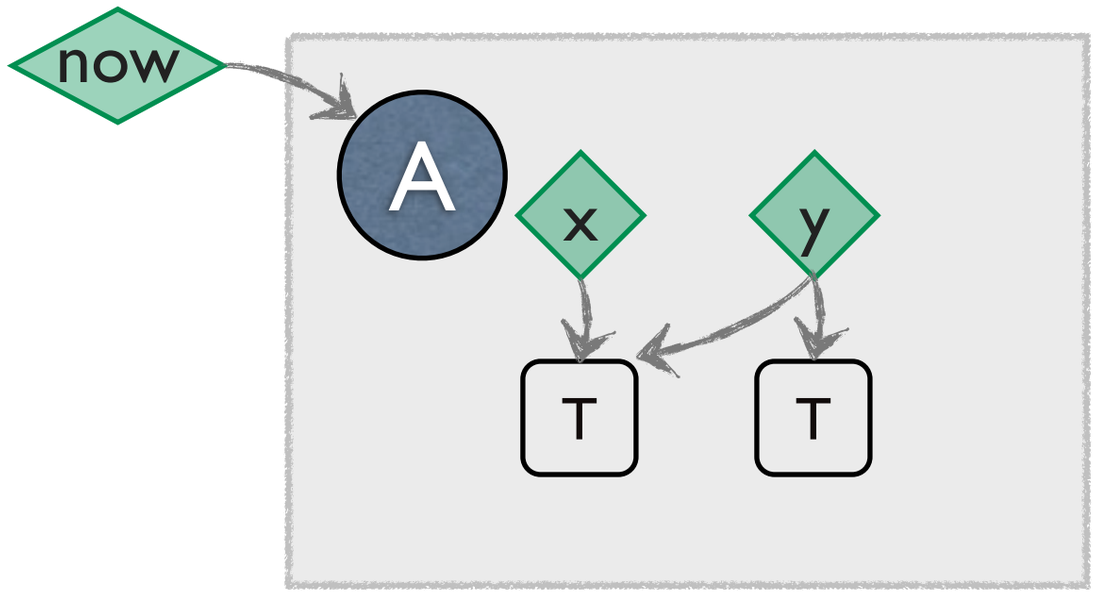
Schedule Extraction

```
task A() {
  T x = new T();
  T y = random ? new T() : x;

  Activation b = schedule B(x);
  Activation c = schedule C(y);

  b → c;
}
```



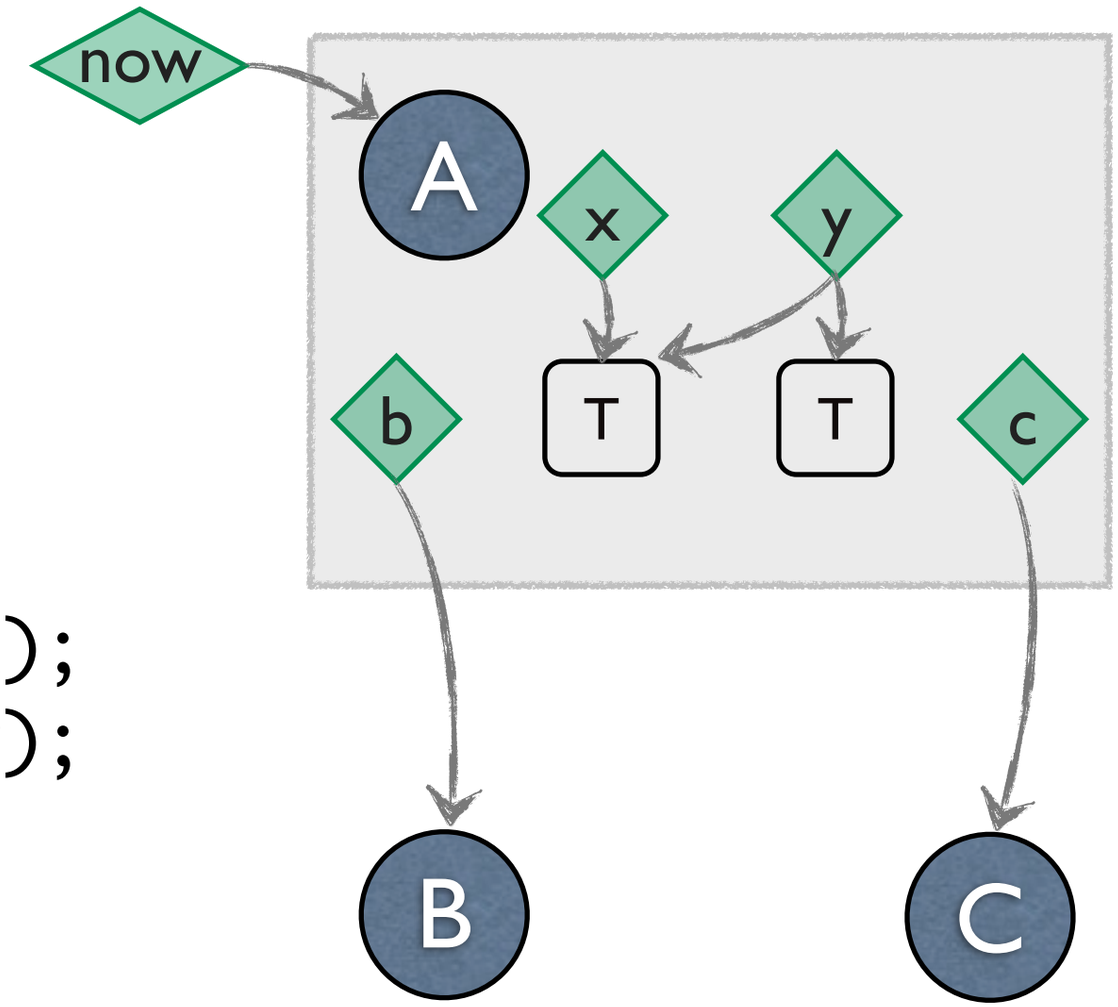
Schedule Extraction

```
task A() {
  T x = new T();
  T y = random ? new T() : x;

  Activation b = schedule B(x);
  Activation c = schedule C(y);

  b → c;
}
```



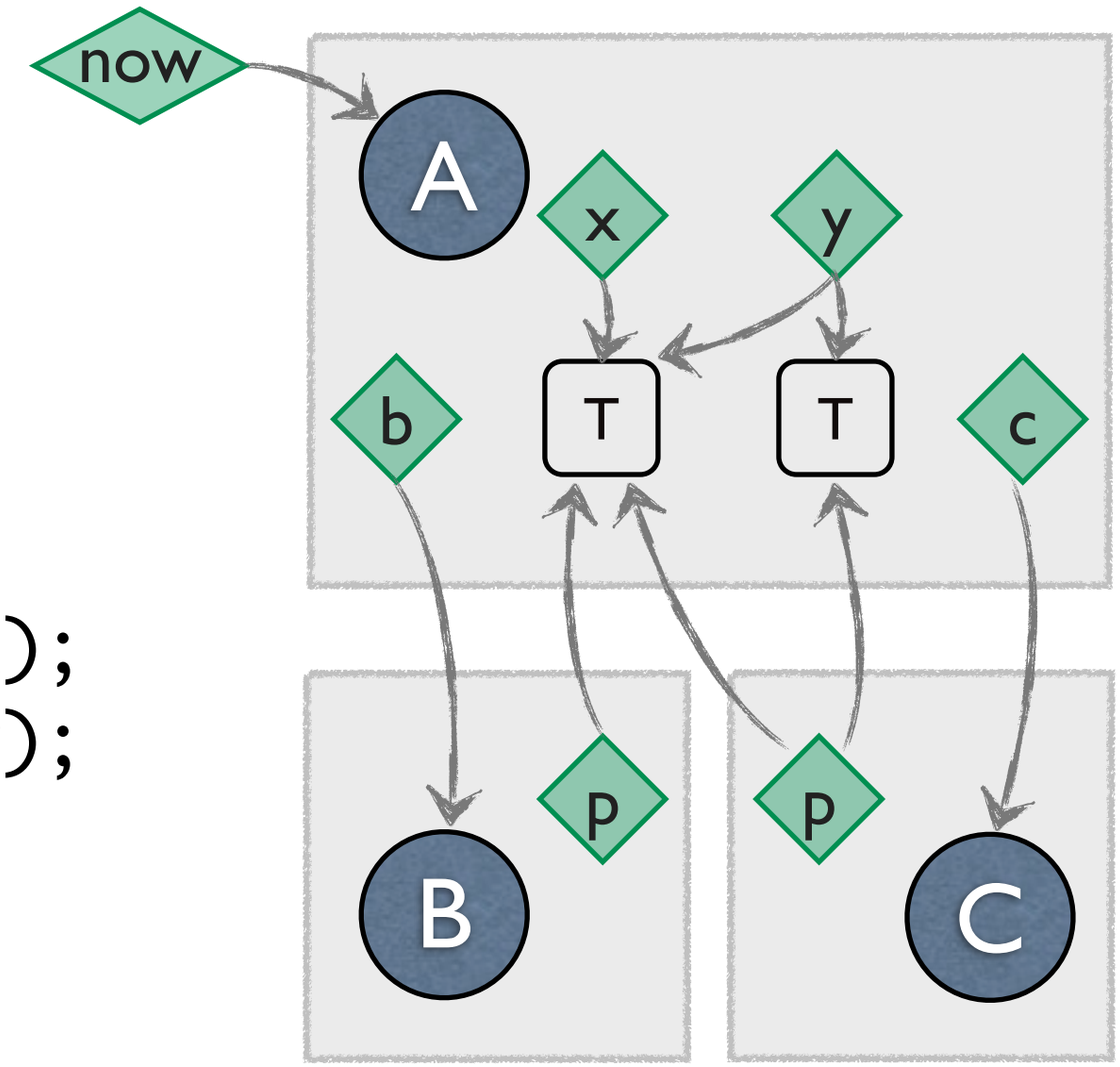
Schedule Extraction

```
task A() {
  T x = new T();
  T y = random ? new T() : x;

  Activation b = schedule B(x);
  Activation c = schedule C(y);

  b → c;
}
```



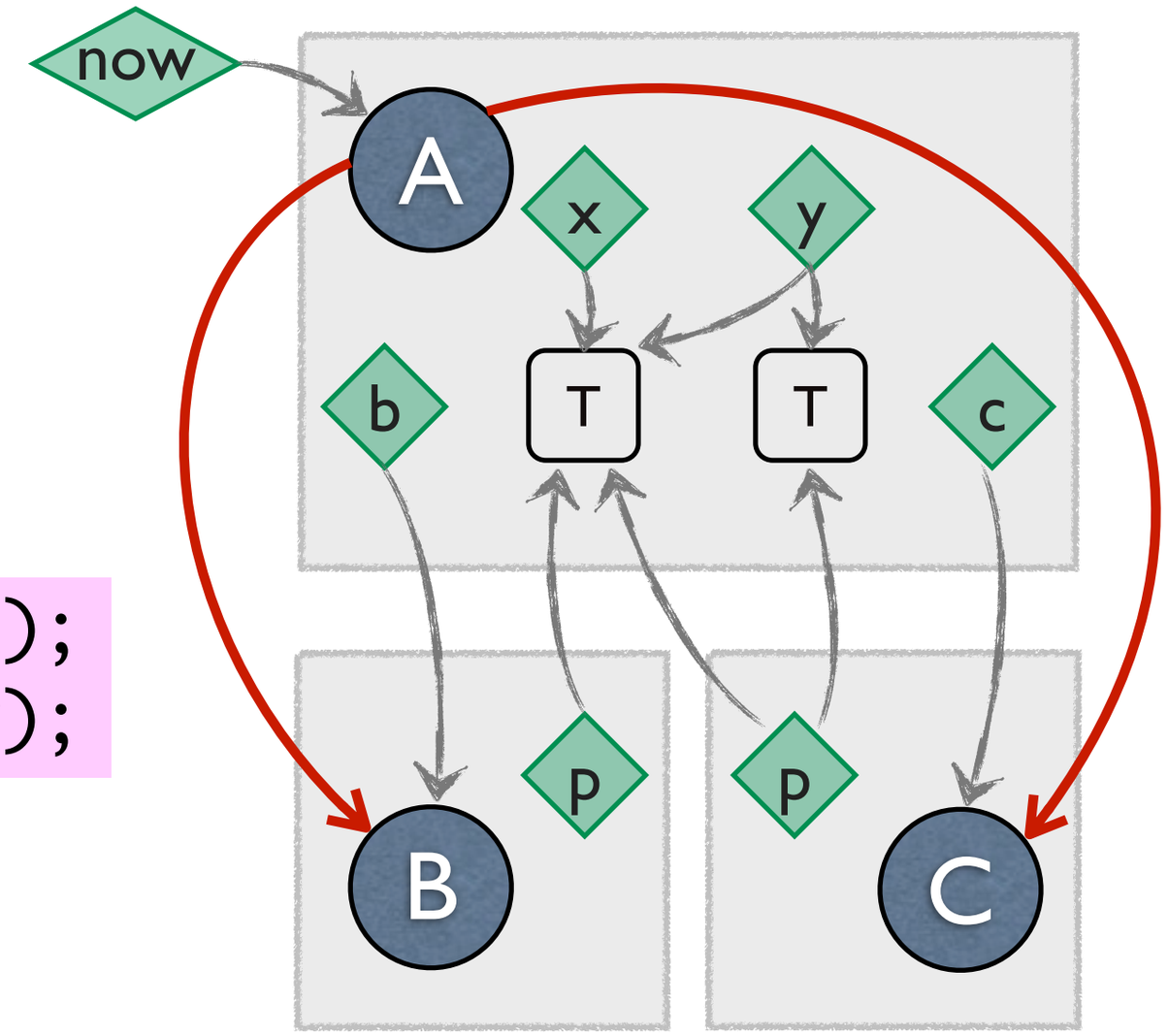
Schedule Extraction

```
task A() {
  T x = new T();
  T y = random ? new T() : x;

  Activation b = schedule B(x);
  Activation c = schedule C(y);

  b → c;
}
```



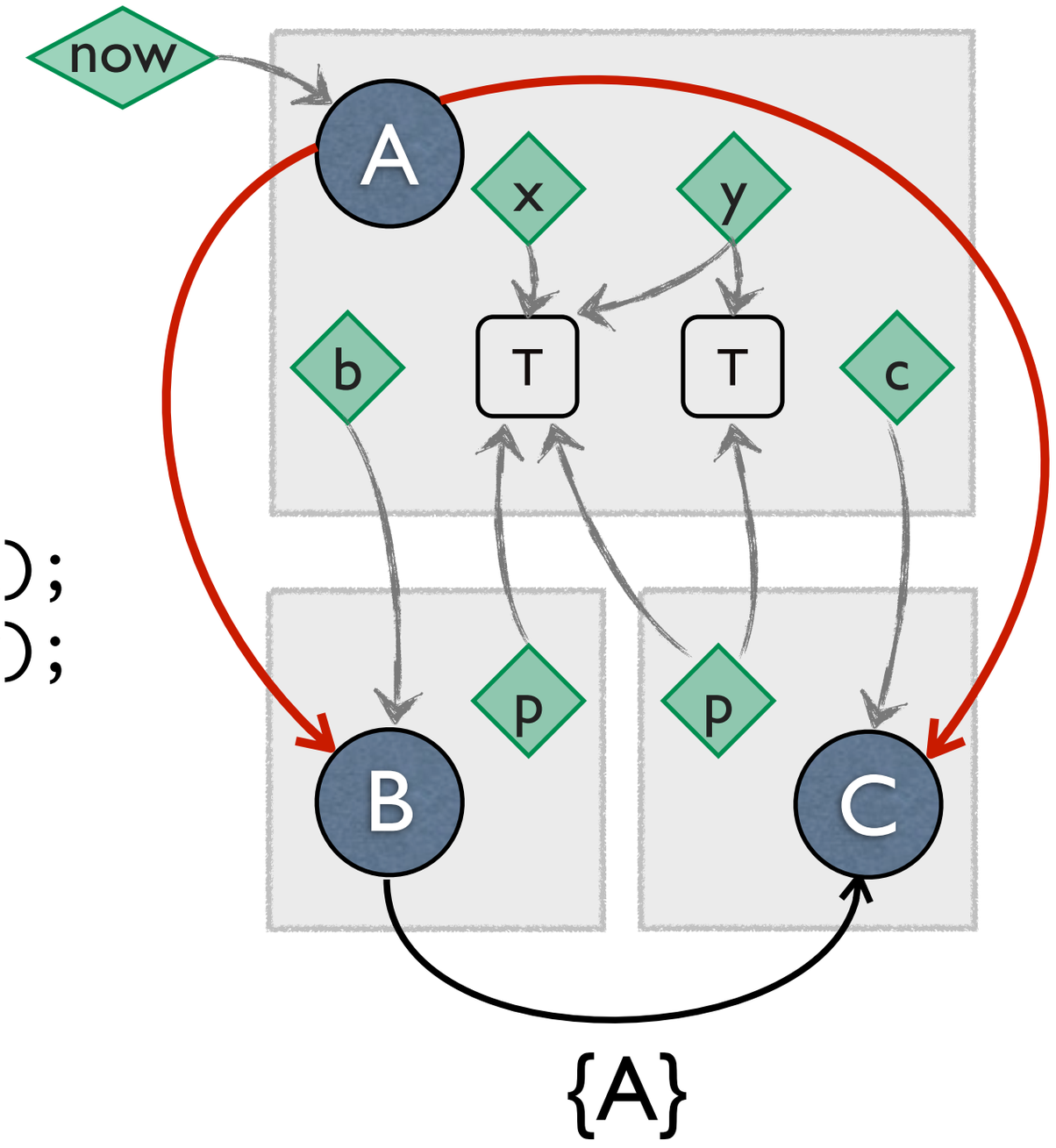
Schedule Extraction

```
task A() {
  T x = new T();
  T y = random ? new T() : x;

  Activation b = schedule B(x);
  Activation c = schedule C(y);

  b → c;
}
```



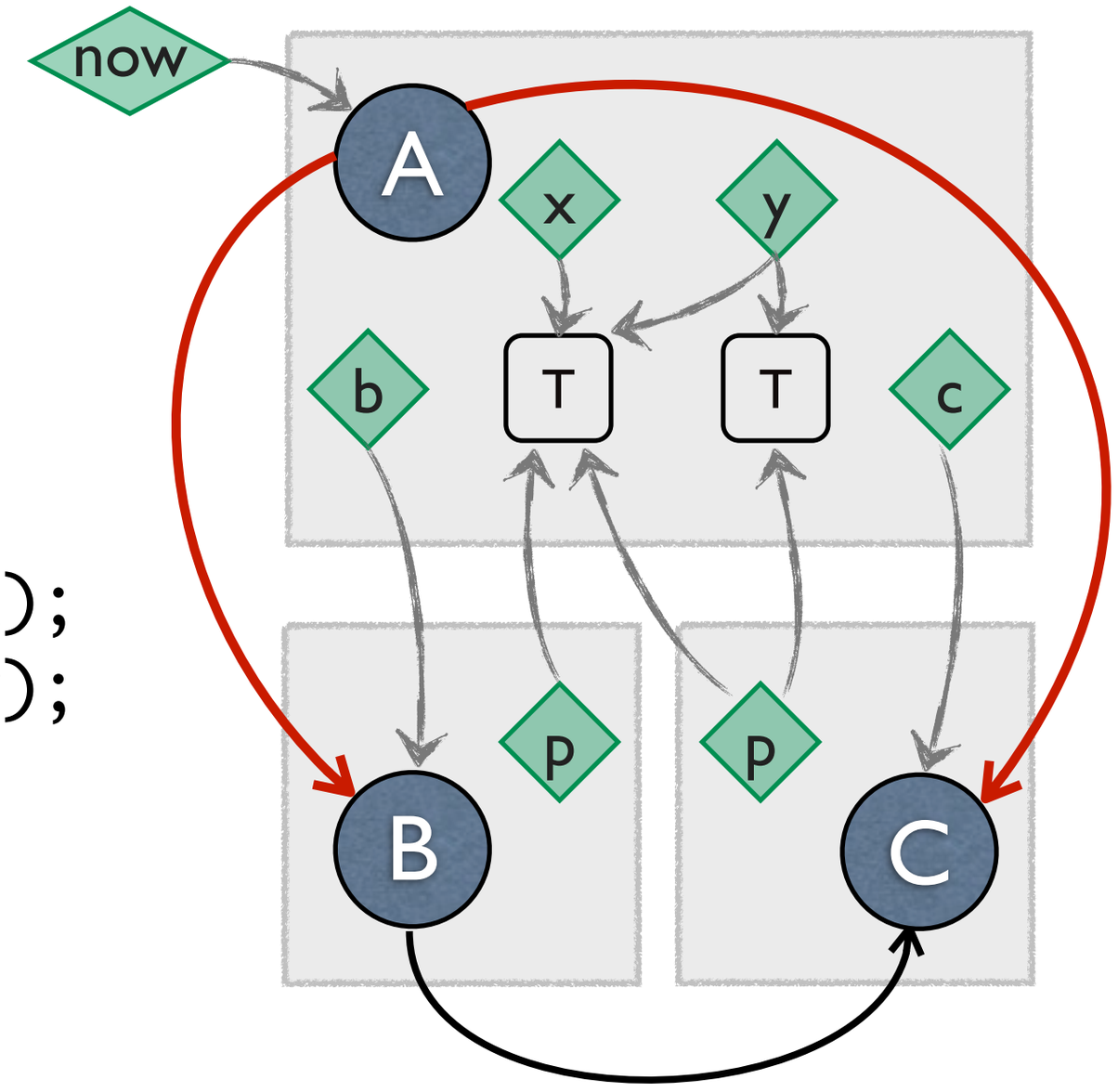
Schedule Extraction

```
task A() {
  T x = new T();
  T y = random ? new T() : x;

  Activation b = schedule B(x);
  Activation c = schedule C(y);

  b → c;
}
```



Schedule Extraction

```
task A() {
  T x = new T();
  T y = random ? new T() : x;

  Activation b = schedule B(x);
  Activation c = schedule C(y);

  b → c;
}
```



☑ Genuine {A}

Using Schedule Analysis

Using Schedule Analysis

- Analyze programs that mix different parallelism styles:
  - Threads, fork/join, intervals, ...

Using Schedule Analysis

- Analyze programs that mix different parallelism styles:
  - Threads, fork/join, intervals, ...
- Analyze programs that mix different synchronization:
  - e.g., lock-based and STM in the same program [LCPC 2010]

Using Schedule Analysis

- Analyze programs that mix different parallelism styles:
  - Threads, fork/join, intervals, ...
- Analyze programs that mix different synchronization:
  - e.g., lock-based and STM in the same program [LCPC 2010]
- Optimizations directly profit from improvements of the schedule analysis

Conclusion

Future compilers must understand the scheduling of tasks at runtime

Conclusion

Future compilers must understand the scheduling of tasks at runtime

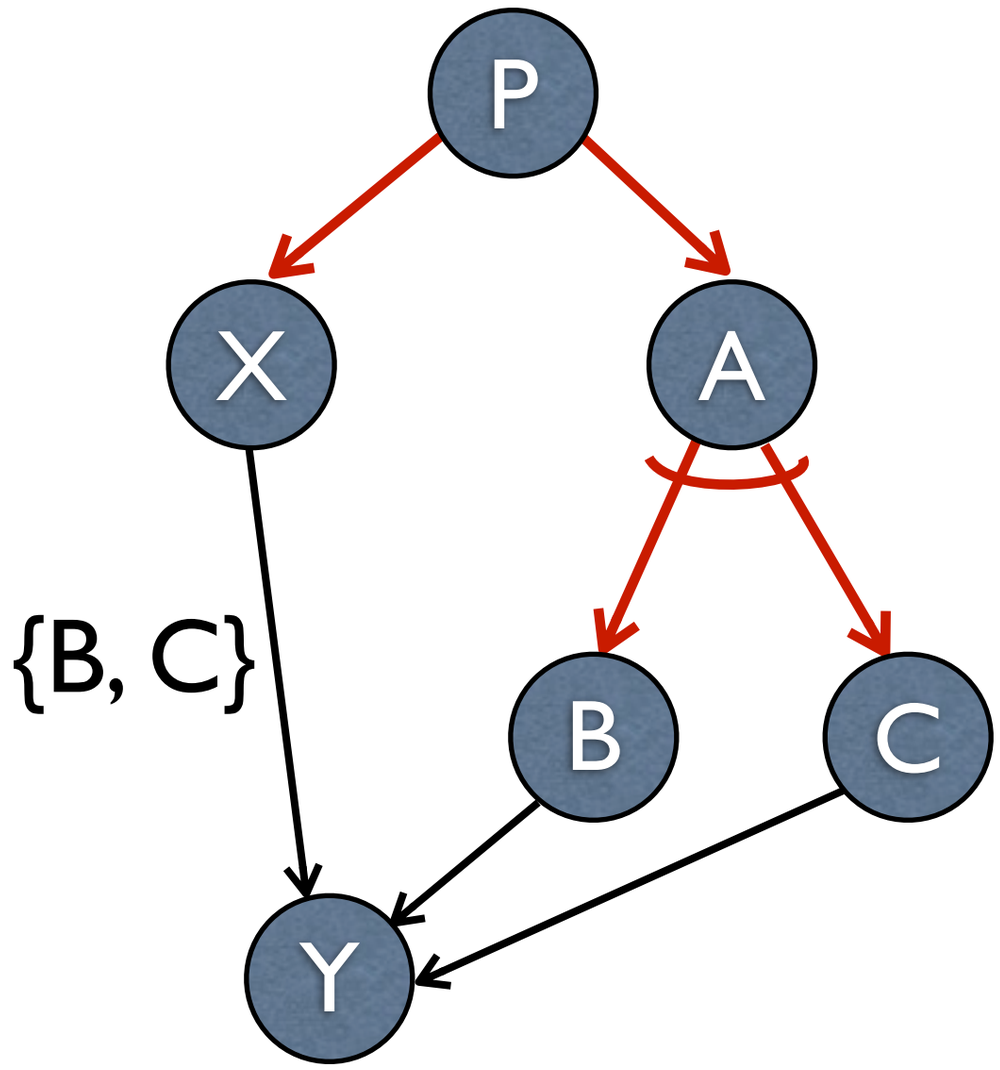
Think "points-to-analysis" for schedules

- What do do with that stuff now?
  - cite lcpc (showed sync removal, in addition lcpc shows stm)
  - think alias analysis for parallel programs

Concluding Remarks

- now→later: simple abstraction to capture explicit scheduling constraints
- Used by programmer or as intermediate representation in compiler
- Explicit happens-before relationships enable static analysis of runtime schedules
  - Schedule Analysis as basis for optimizations
  - Integration in a single optimizing compiler

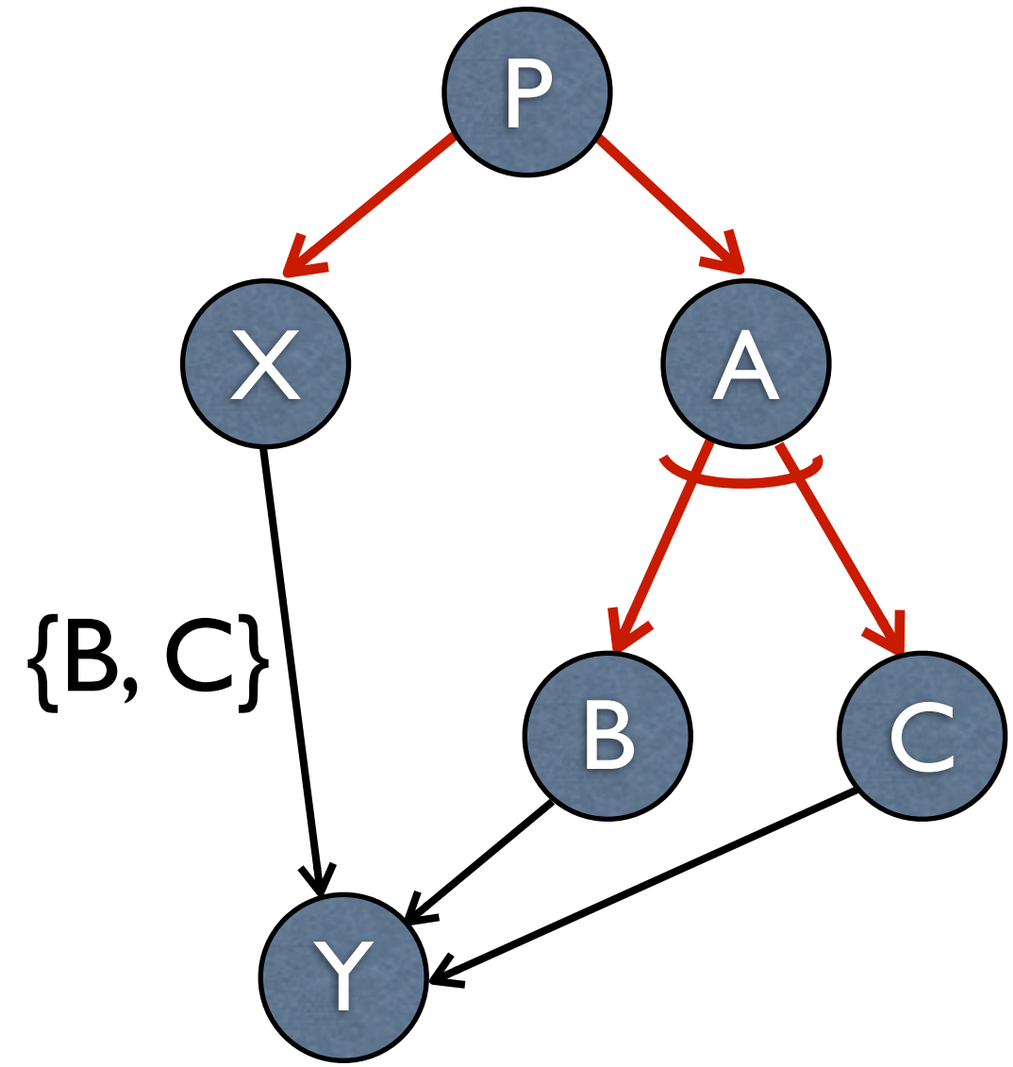
Genuine Explicit Edges



Genuine Explicit Edges

- An edge $x \rightarrow y$ is genuine if activation of x implies edge has been created:

$$x \Rightarrow x \rightarrow y$$



Conditional Edges

- Static analysis is imprecise
  - Conditional activation
  - Conditional creation of edges
- Question: can the analysis *rely* on a happens-before edge?
  - If *yes*, **we call the edge** *genuine*

Structural Properties

Structural Properties

- Schedules must form a directed acyclic graph (DAG)
  - For *progress* and *liveness* of activations

Structural Properties

- Schedules must form a directed acyclic graph (DAG)
  - For *progress* and *liveness* of activations
- Possible relations between two activations:
  - *sequential:* execution is strictly ordered
  - *exclusive:* activations can never co-exist at runtime
  - *(potentially) parallel:* neither sequential nor exclusive

Related Work

- Pointer Analysis for Parallel Programs [Rugina, Rinard '03]
  - Interference information for fork/join parallelism
- Combined with Escape Analysis [Salcianu, Rinard '01], [Nanda, Ramesh '03]
  - Compute Points-to sets, no ordering
- May-happen-in-parallel [Naumovich et al. '99]
  - For X10 (structured parallelism)
- May-happen-before [Barik '05]
  - Happens-before relations in thread creation trees