Rely/Guarantee Reasoning for Asynchronous Programs

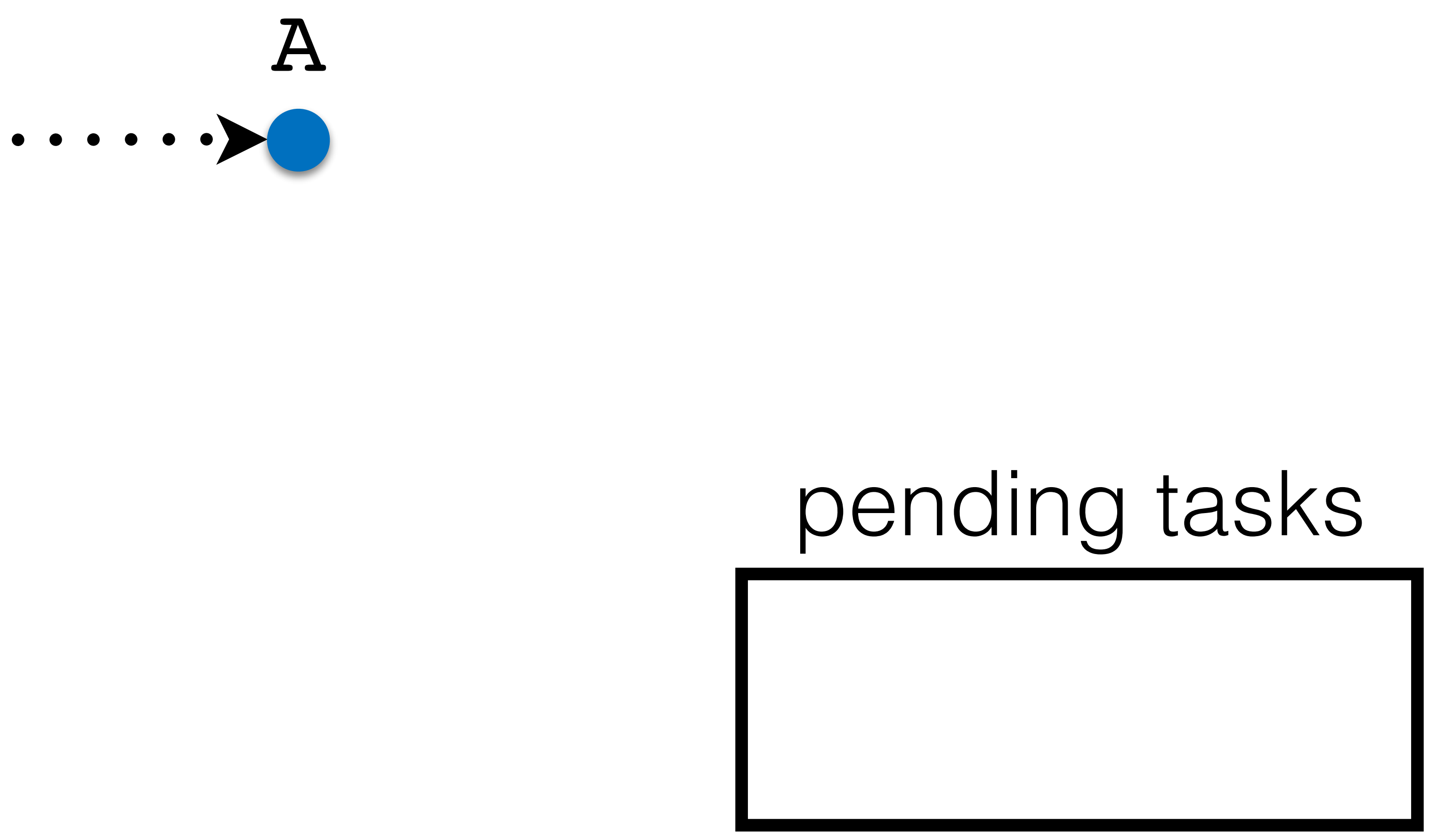
Ivan Gavran[1], **Filip Niksic**[1], Aditya Kanade[2], Rupak Majumdar[1], Viktor Vafeiadis[1]

[1] Max Planck Institute for Software Systems (MPI-SWS), Germany
[2] Indian Institute of Science, Bangalore, India

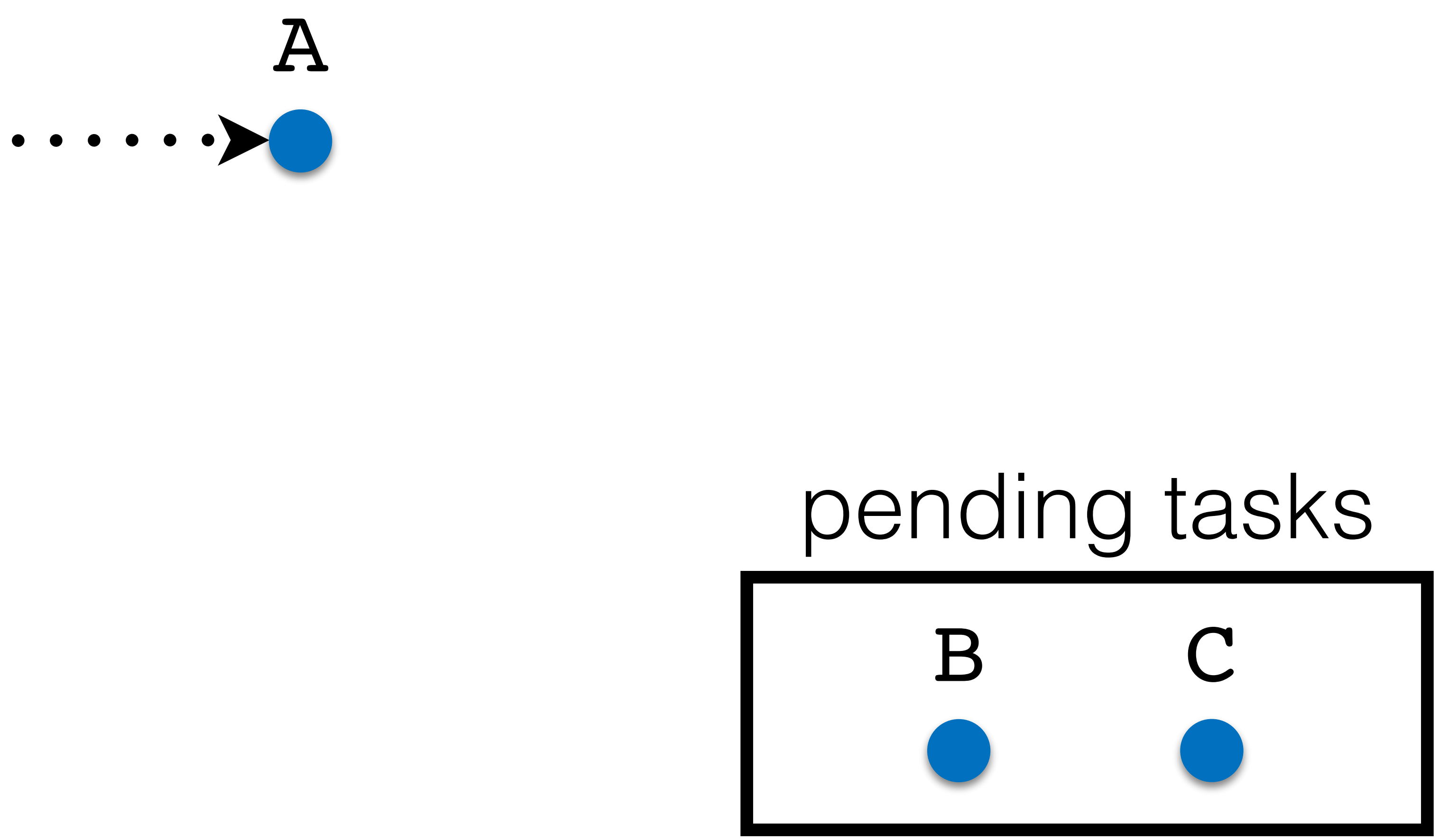
Asynchronous programming is widespread

- **Web apps**: AJAX, jQuery, XMLHttpRequest
- **Smartphone apps**: AsyncTask, dispatch_async
- **Server-side**: node.js, java.nio
- **Systems**: kqueue, epoll, Libevent
- **Other**: async/await in Scala

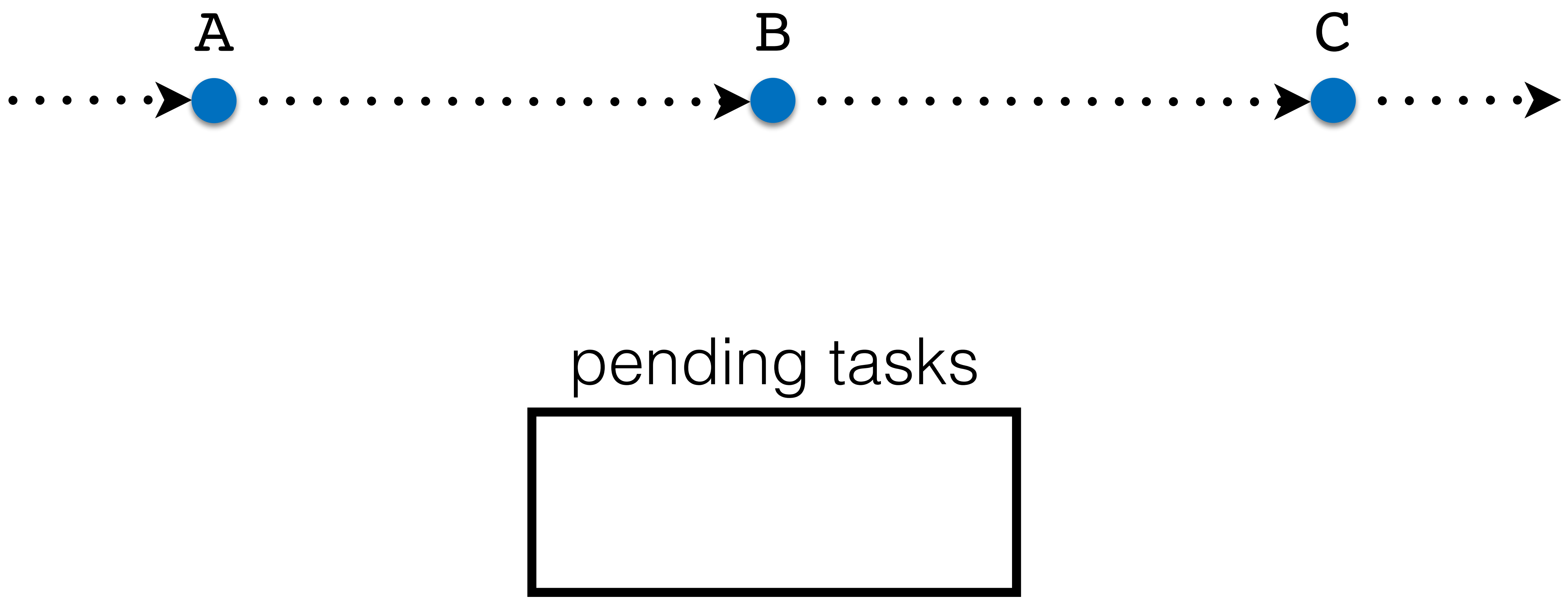
Common feature:
Posting tasks for later execution



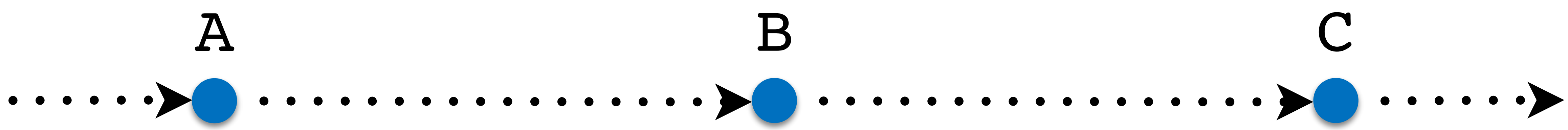
Common feature:
Posting tasks for later execution



Common feature:
Posting tasks for later execution



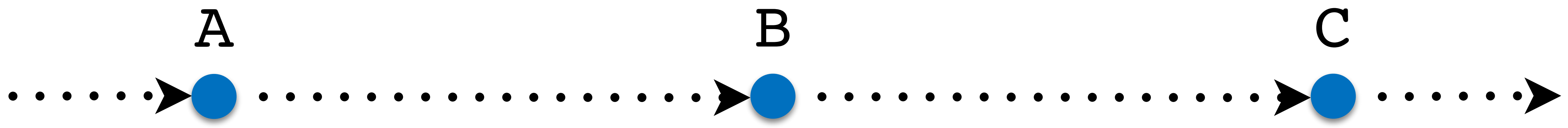
Common feature:
Posting tasks for later execution



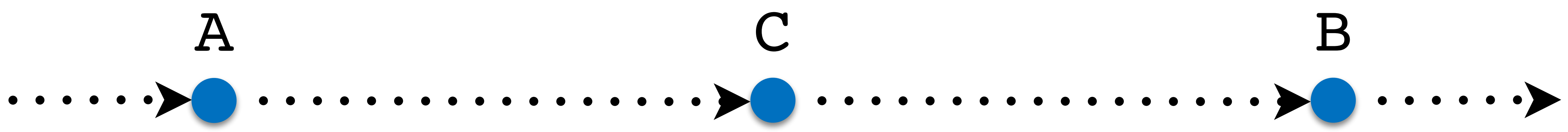
Tasks may be executed when

- triggered by external events
  (mouse click, response ready, socket ready, …)
- dispatched by a scheduler

Drawback:
Obscured control-flow

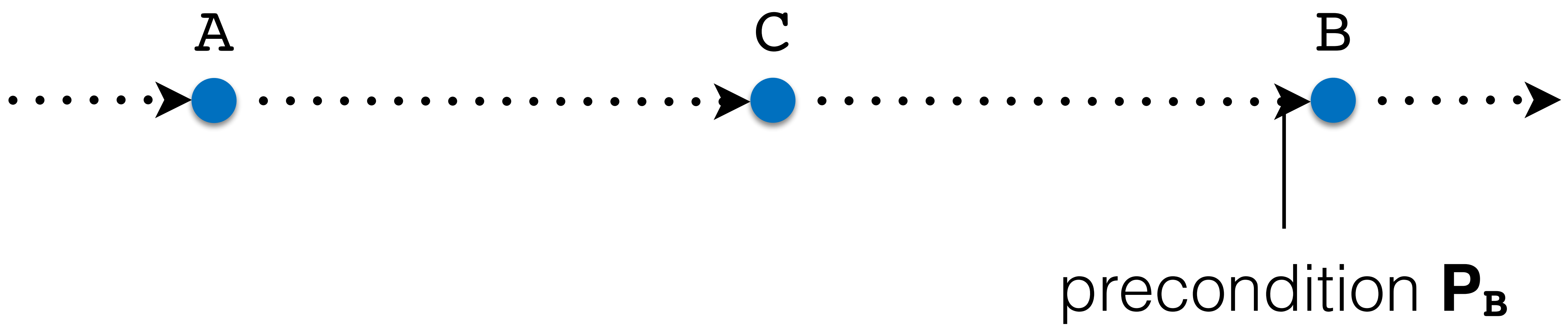


Drawback:
Obscured control-flow

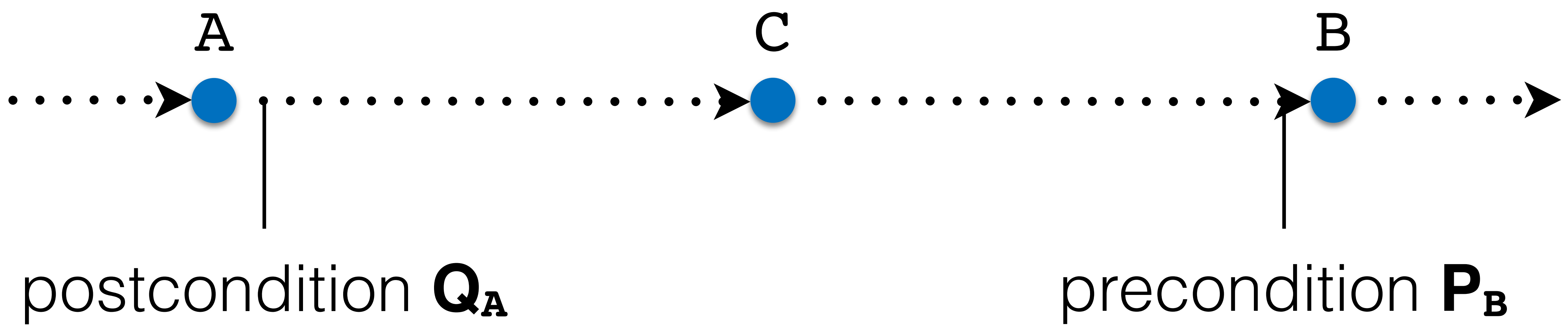


Multiple **pending tasks** may be executed **in any order**.

Drawback: Obscured control-flow

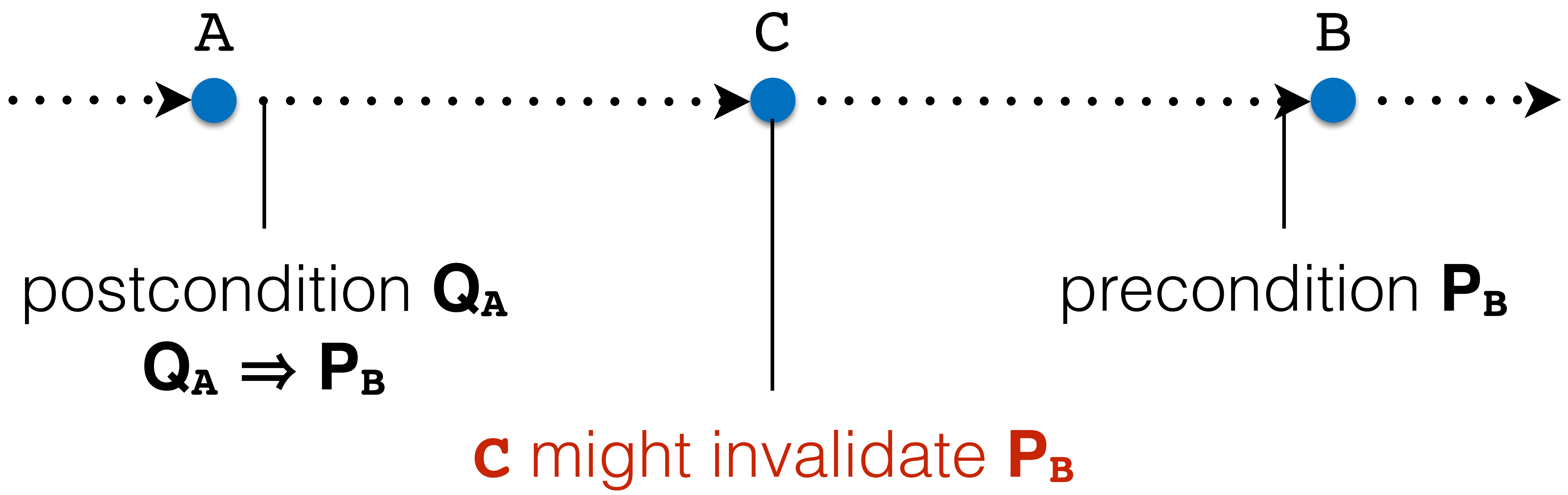


Drawback:
Obscured control-flow

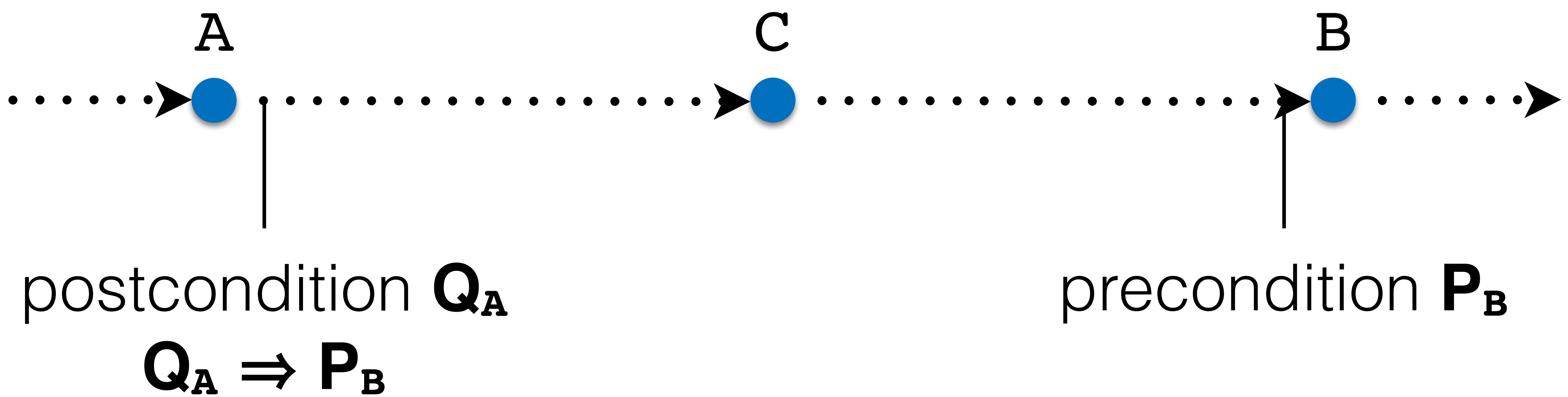


$\mathbf{Q}_{\mathtt{A}} \Rightarrow \mathbf{P}_{\mathtt{B}}$

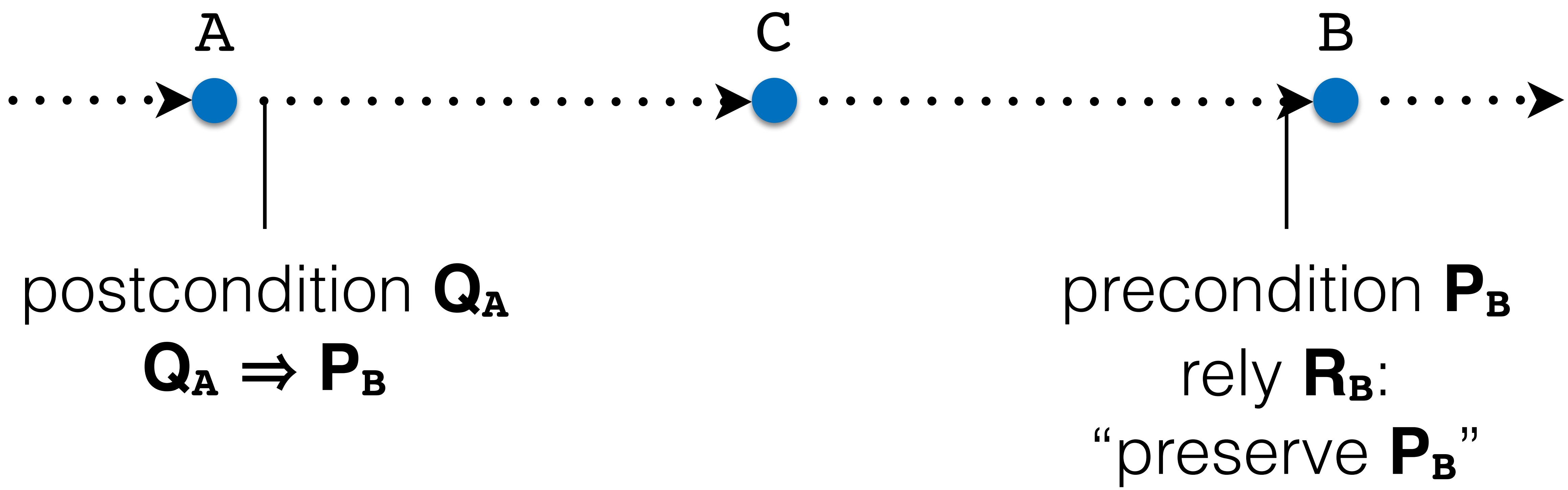
Drawback: Obscured control-flow



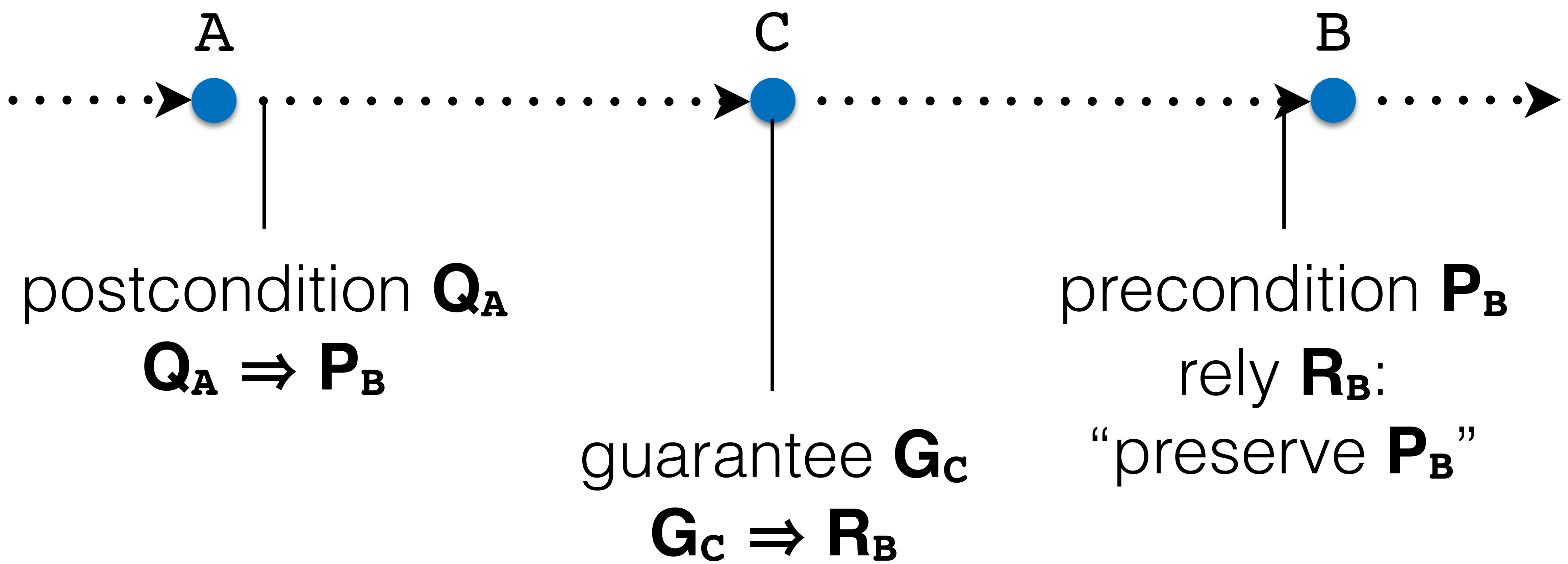
Adapting rely/guarantee reasoning [Jones83]



Adapting rely/guarantee reasoning [Jones83]



Adapting rely/guarantee reasoning [Jones83]



Soundness of rely/guarantee reasoning

Given a program with specification in terms of predicates P, Q, R, G, if

- the predicates satisfy “natural rely/guarantee conditions”
- each task meets its rely/guarantee specification

then the program is correct.

Rely/guarantee reasoning is modular

Sufficient to **verify** each task **in isolation**,

using a verifier for sequential software.

Contributions

We have:

- Identified the “natural rely/guarantee conditions”
- Proved soundness of rely/guarantee reasoning
- Demonstrated the approach on two C programs that use Libevent (done using Frama-C)

The rest of the talk: Rely/guarantee…

… by example

… in theory

… in practice

The rest of the talk: Rely/guarantee…

… by example

… in theory

… in practice

Modeling asynchronous tasks

Extend an imperative language with **asynchronous procedures**, together with constructs:

```
post f(v1, …, vk)

delete f(v1, …, vk)
```

Maintain a **set of pending procedure instances**.

Execute instances atomically in a **non-deterministic order**.

Example: ROT13 server

```
async main() {
    int socket = prepare_socket();
    post accept(socket);
}

async accept(int socket) {
    struct client *c = malloc(…);
    client_setup(c);
    c->fd = accept_connection(socket);
    post read(c);
    post accept(socket);
}

async read(struct client *c) { … }

async write(struct client *c) { … }
```

Example: ROT13 server

```
async read(struct client *c) {
    if (…) { // c->fd is ready
        receive_chunk(c);
        post write(c);
        post read(c);
    }
    else { // connection is closed
        delete write(c);
        free(c);
    }
}
```

```
async write(struct client *c) {
    if (…) { // c->fd is ready
        send_chunk(c);
        if (more_to_send(c))
            post write(c);
    }
    else { // connection is closed
        delete read(c);
        free(c);
    }
}
```

Example: ROT13 server

```
//@ requires valid(c);
async read(struct client *c) {
    if (…) { // c->fd is ready
        receive_chunk(c);
        post write(c);
        post read(c);
    }
    else { // connection is closed
        delete write(c);
        free(c);
    }
}
```

```
//@ requires valid(c);
async write(struct client *c) {
    if (…) { // c->fd is ready
        send_chunk(c);
        if (more_to_send(c))
            post write(c);
    }
    else { // connection is closed
        delete read(c);
        free(c);
    }
}
```

Introducing predicate $\text{posted}_{\texttt{f}}$

For each asynchronous procedure `f(x1,…,xk)`, we introduce a predicate

$$\textbf{posted}_{\texttt{f}}(\mathbf{x_1, \ldots, x_k})$$

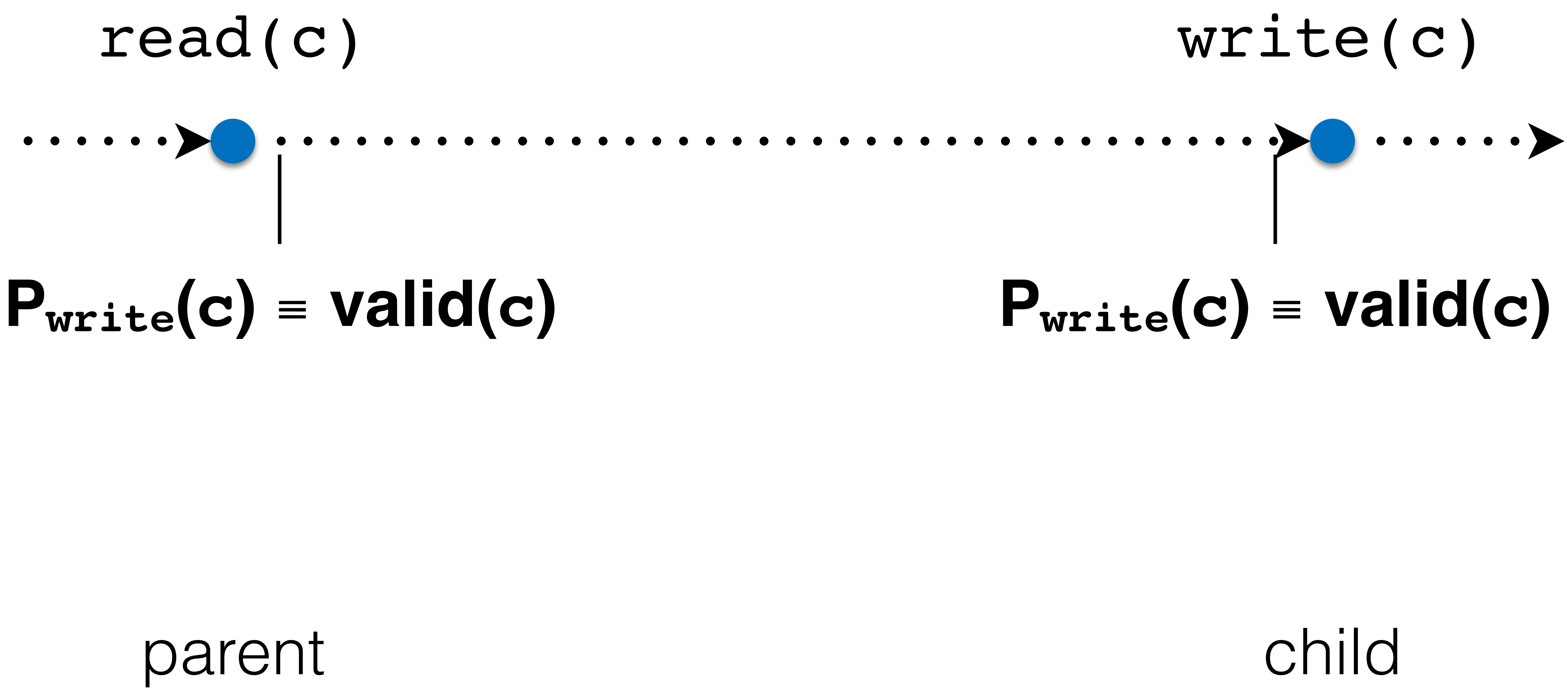
True iff **`f`** has been posted with arguments $\mathbf{x_1, \ldots, x_k}$ during the execution of the **current asynchronous procedure**.

Example: ROT13 server

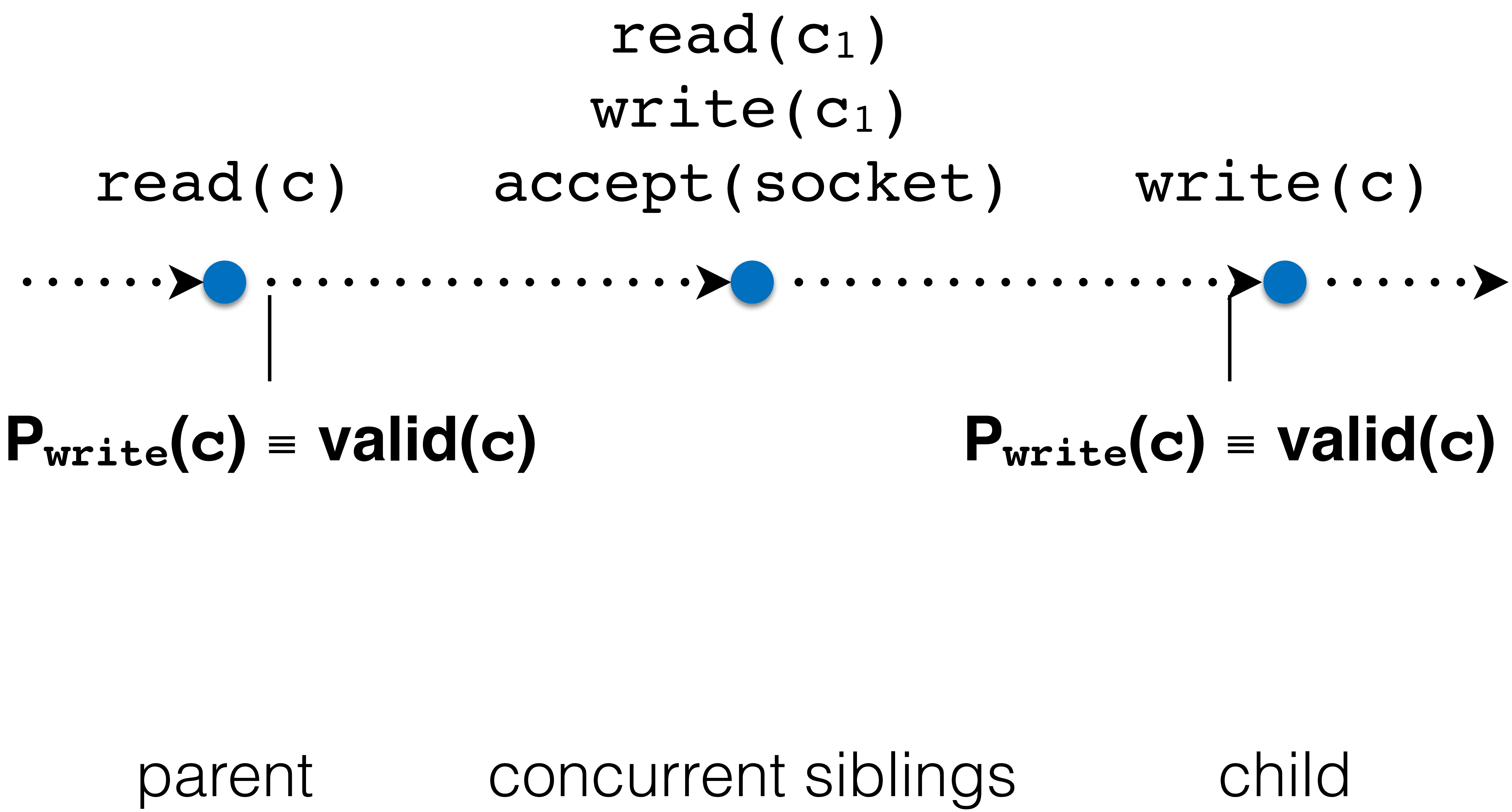
```
/*@ requires valid(c);
  @ ensures ∀c₁;
  @ posted_read(c₁) ⇒ valid(c₁);
  @ ensures ∀c₁;
  @ posted_write(c₁) ⇒ valid(c₁);
  @*/
async read(struct client *c) {
    if (…) { // c->fd is ready
        receive_chunk(c);
        post write(c);
        post read(c);
    }
    else { // connection is closed
        delete write(c);
        free(c);
    }
}
```

```
/*@ requires valid(c);
  @ ensures ∀c₁;
  @ posted_write(c₁) ⇒ valid(c₁);
  @*/
async write(struct client *c) {
    if (…) { // c->fd is ready
        send_chunk(c);
        if (more_to_send(c))
            post write(c);
    }
    else { // connection is closed
        delete read(c);
        free(c);
    }
}
```

Preserving the precondition



Preserving the precondition

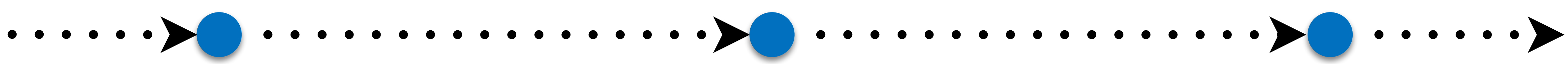


Preserving the precondition

`read(c)`

`read(c₁)` `write(c₁)` `accept(socket)`

`write(c)`



guarantee $\mathbf{P}_{\texttt{write}}$ is preserved

$\mathbf{G}_{\texttt{read}} \Rightarrow \mathbf{R}_{\texttt{write}}$

$\mathbf{G}_{\texttt{write}} \Rightarrow \mathbf{R}_{\texttt{write}}$

$\mathbf{G}_{\texttt{accept}} \Rightarrow \mathbf{R}_{\texttt{write}}$

rely on $\mathbf{P}_{\texttt{write}}$ being preserved

parent

concurrent siblings

child

Introducing predicate pending$_{\mathtt{f}}$

For each asynchronous procedure `f(x1,…,xk)`, we introduce a predicate

$$\mathbf{pending}_{\mathtt{f}}(x_1, \ldots, x_k)$$

True iff `f` with arguments $x_1, \ldots, x_k$ is **pending**, i.e. is in the set of pending procedure instances.

write's rely predicate $R_{\texttt{write}}$

$$\mathbf{R}_{\texttt{write}} \equiv \forall c.\ (\text{pending}'_{\texttt{write}}(c) \wedge \text{pending}_{\texttt{write}}(c) \wedge \text{valid}'(c)) \Rightarrow \text{valid}(c)$$

(prime means at the beginning of execution)

`write`'s global invariant

With `write`'s parents ensuring:

$\forall c.\ \text{posted}_{\texttt{write}}(c) \Rightarrow \text{valid}(c)$

and `write`'s concurrent siblings ensuring:

$\forall c.\ (\text{pending'}_{\texttt{write}}(c) \wedge \text{pending}_{\texttt{write}}(c)$

$\wedge\ \text{valid'}(c)) \Rightarrow \text{valid}(c)$

rely/guarantee ensures a **global invariant**:

$\boldsymbol{\forall c.\ \text{pending}_{\texttt{write}}(c) \Rightarrow \text{valid}(c)}$

Final specification of `write`

```
/*@ requires Precondition:
  @     valid(c);
  @ ensures Parent_child_condition:
  @     ∀c1; posted_write(c1) ⇒ valid(c1);
  @ ensures Guarantee:
  @     (∀c1; (pending_read’(c1) ∧ pending_read(c1)
  @             ∧ valid’(c1)) ⇒ valid(c1))
  @     ∧ (∀c1; (pending_write’(c1) ∧ pending_write(c1)
  @               ∧ valid’(c1)) ⇒ valid(c1));
  @*/
async write(struct client *c) { … }
```

Final specification of `write`

```
/*@ requires Precondition:
  @     valid(c);
  @ ensures Parent_child_condition:
  @     ∀c1; posted_write(c1) ⇒ valid(c1);
  @ ensures Guarantee:
  @     (∀c1; (pending_read'(c1) ∧ pending_read(c1)           } R_read
  @                ∧ valid'(c1)) ⇒ valid(c1))
  @     ∧ (∀c1; (pending_write'(c1) ∧ pending_write(c1)        } R_write
  @                ∧ valid'(c1)) ⇒ valid(c1));
  @*/
async write(struct client *c) { … }
```

Final specification of `write`

```
/*@ requires Precondition:
  @     valid(c);
  @ ensures Parent_child_condition:
  @     ∀c₁; posted_write(c₁) ⇒ valid(c₁);
  @ ensures Guarantee:
  @     (∀c₁; (pending_read'(c₁) ∧ pending_read(c₁)          }  R_read
  @             ∧ valid'(c₁)) ⇒ valid(c₁))
  @     ∧ (∀c₁; (pending_write'(c₁) ∧ pending_write(c₁)      }  R_write
  @               ∧ valid'(c₁)) ⇒ valid(c₁));
  @*/
async write(struct client *c) { … }
```

`write` can now be verified in isolation using a standard verification tool (in our case Frama-C)

The rest of the talk: Rely/guarantee…

… by example

… in theory

… in practice

Rely/guarantee decomposition

For each asynchronous procedure `f` we require:

- $P_{\texttt{f}}$ — precondition predicate
- $R_{\texttt{f}}$ — rely predicate
- $G_{\texttt{f}}$ — guarantee predicate
- $Q_{\texttt{f}}$ — postcondition predicate

First-order formulas; may include predicates $\text{posted}_{\texttt{g}}$ and $\text{pending}_{\texttt{g}}$ (in negative positions)

Rely/guarantee conditions

Given a rely/guarantee decomposition, for each asynchronous procedure **`f`**:

(1) $Q_{\mathtt{f}} \Rightarrow G_{\mathtt{f}}$

(2) $Q_{\mathtt{g}} \Rightarrow (\mathrm{posted}_{\mathtt{f}} \Rightarrow P_{\mathtt{f}})$, for each $\mathtt{g} \in \mathrm{parents}(\mathtt{f})$

(3) $R_{\mathtt{f}} \Rightarrow ((\mathrm{pending'}_{\mathtt{f}} \wedge \mathrm{pending}_{\mathtt{f}} \wedge P'_{\mathtt{f}}) \Rightarrow P_{\mathtt{f}})$

(4) $G_{\mathtt{g}} \Rightarrow R_{\mathtt{f}}$, for each $\mathtt{g} \in \mathrm{siblings}(\mathtt{f})$

Soundness of rely/guarantee reasoning

Theorem. Given an asynchronous program with a rely/guarantee decomposition, if

- the decomposition satisfies the rely/guarantee conditions
- each procedure meets its specification (P and Q)

then the program is correct.

Key lemma

Lemma. For each asynchronous procedure `f`, at every schedule point we have

$$\mathbf{pending}_{\mathtt{f}} \Rightarrow \mathbf{P}_{\mathtt{f}}$$

The rest of the talk: Rely/guarantee…

… by example

… in theory

… in practice

Generic rely/guarantee predicates

Given preconditions P_f, the **weakest predicates** that satisfy the rely/guarantee conditions:

- $R_f \equiv (\text{pending}'_f \wedge \text{pending}_f \wedge P'_f) \Rightarrow P_f$

- $G_f \equiv \bigwedge_{g \in \text{siblings}(f)} R_g$

- $Q_f \equiv G_f \wedge \bigwedge_{g \in \text{children}(f)} \text{posted}_g \Rightarrow P_g$

Generic rely/guarantee predicates

Sufficient for the ROT13 example:

```
//@ requires valid(c);
async read(struct client *c) {
    if (…) { // c->fd is ready
        receive_chunk(c);
        post write(c);
        post read(c);
    }
    else { // connection is closed
        delete write(c);
        free(c);
    }
}
```

```
//@ requires valid(c);
async write(struct client *c) {
    if (…) { // c->fd is ready
        send_chunk(c);
        if (more_to_send(c))
            post write(c);
    }
    else { // connection is closed
        delete read(c);
        free(c);
    }
}
```

Generic rely/guarantee predicates

Sufficient for the ROT13 example:

```
//@ requires valid(c);
async read(struct client *c) {
    if (…) { // c->fd is ready
        receive_chunk(c);
        post write(c);
        post read(c);
    }
    else { // connection is closed
        delete write(c);
        free(c);
    }
}
```

```
//@ requires valid(c);
async write(struct client *c) {
    if (…) { // c->fd is ready
        send_chunk(c);
        if (more_to_send(c))
            post write(c);
    }
    else { // connection is closed
        delete read(c);
        free(c);
    }
}
```

Not sufficient in general.

Implementation for Libevent

- Focused on C programs that use Libevent
- Low-level usage of Libevent replaced with calls to

  $\texttt{post\_f}(x_1, \ldots, x_k)$

  $\texttt{delete\_f}(x_1, \ldots, x_k)$

- Verification done using Frama-C (WP, Z3)

  good: utilizing a mature and stable tool

  bad: utilizing a mature and stable tool (!)

Summary

We have:

- Identified the "natural rely/guarantee conditions"
- Proved soundness of rely/guarantee reasoning
- Demonstrated the approach on two C programs that use Libevent (done using Frama-C)

http://www.mpi-sws.org/~fniksic/ **fniksic@mpi-sws.org**

Race example

```
struct device {
    int owner;
    …
} dev;

async main() {
    dev.owner = 0;
    int socket = prepare_socket();
    post accept(socket);
}

async accept(int socket) {
    int id = new_client_id(); // positive, unique
    int fd = accept_connection(socket);
    post new_client(id, fd);
    post accept(socket);
}

async new_client(int id, int fd) { … }

async write(int id, int fd) { … }
```

Race example

```
async new_client(int id, int fd) {
    if (dev.owner > 0) {
        post new_client(id, fd);
    }
    else {
        dev.owner = id;
        post write(id, fd);
    }
}
```

```
async write(int id, int fd) {
    if (transfer(fd, dev)) {
        post write(id, fd);
    }
    else { // write complete
        dev.owner = 0;
    }
}
```

Race example

```
/*@ requires Precondition:
  @   id > 0;
  @*/
async new_client(int id, int fd) {
    if (dev.owner > 0) {
        post new_client(id, fd);
    }
    else {
        dev.owner = id;
        post write(id, fd);
    }
}
```

```
/*@ requires Precondition:
  @   id > 0 ∧
  @   dev.owner == id ∧
  @   ∀ id1, fd1;
  @     pending_write(id1, fd1)
  @       ⇒ id == id1 ∧ fd == fd1;
  @*/
async write(int id, int fd) {
    if (transfer(fd, dev)) {
        post write(id, fd);
    }
    else { // write complete
        dev.owner = 0;
    }
}
```

Race example

```
/*@ requires Precondition:
  @   id > 0;
  @ ensures Parent_child_write:
  @   ∀ id₁, fd₁;
  @     posted_write(id₁, fd₁)
  @       ⇒ P_write(id₁, fd₁);
  @*/
async new_client(int id, int fd) {
    if (dev.owner > 0) {
        post new_client(id, fd);
    }
    else {
        dev.owner = id;
        post write(id, fd);
    }
}
```

```
/*@ requires Precondition:
  @   id > 0 ∧
  @   dev.owner == id ∧
  @   ∀ id₁, fd₁;
  @     pending_write(id₁, fd₁)
  @       ⇒ id == id₁ ∧ fd == fd₁;
  @*/
async write(int id, int fd) {
    if (transfer(fd, dev)) {
        post write(id, fd);
    }
    else { // write complete
        dev.owner = 0;
    }
}
```

Race example

```
/*@ requires Precondition:
  @   id > 0;
  @ ensures Parent_child_write: X
  @   ∀ id₁, fd₁;
  @     posted_write(id₁, fd₁)
  @       ⇒ P_write(id₁, fd₁);
  @*/
async new_client(int id, int fd) {
    if (dev.owner > 0) {
        post new_client(id, fd);
    }
    else {
        dev.owner = id;
        post write(id, fd);
    }
}
```

```
/*@ requires Precondition:
  @   id > 0 ∧
  @   dev.owner == id ∧
  @   ∀ id₁, fd₁;
  @     pending_write(id₁, fd₁)
  @       ⇒ id == id₁ ∧ fd == fd₁;
  @*/
async write(int id, int fd) {
    if (transfer(fd, dev)) {
        post write(id, fd);
    }
    else { // write complete
        dev.owner = 0;
    }
}
```

Race example

```
/*@ requires Precondition:
  @   id > 0;
  @ requires Global_inv_write:
  @   ∀ id₁, fd₁;
  @     pending_write(id₁, fd₁)
  @       ⇒ P_write(id₁, fd₁);
  @ ensures Parent_child_write:
  @   ∀ id₁, fd₁;
  @     posted_write(id₁, fd₁)
  @       ⇒ P_write(id₁, fd₁);
  @*/
async new_client(int id, int fd) {
    if (dev.owner > 0) {
        post new_client(id, fd);
    }
    else {
        dev.owner = id;
        post write(id, fd);
    }
}
```

```
/*@ requires Precondition:
  @   id > 0 ∧
  @   dev.owner == id ∧
  @   ∀ id₁, fd₁;
  @     pending_write(id₁, fd₁)
  @       ⇒ id == id₁ ∧ fd == fd₁;
  @*/
async write(int id, int fd) {
    if (transfer(fd, dev)) {
        post write(id, fd);
    }
    else { // write complete
        dev.owner = 0;
    }
}
```

Race example

```
/*@ requires Precondition:
  @   id > 0;
  @ requires Global_inv_write:
  @   ∀ id₁, fd₁;
  @     pending_write(id₁, fd₁)
  @       ⇒ P_write(id₁, fd₁);
  @ ensures Parent_child_write: ✓
  @   ∀ id₁, fd₁;
  @     posted_write(id₁, fd₁)
  @       ⇒ P_write(id₁, fd₁);
  @*/
async new_client(int id, int fd) {
    if (dev.owner > 0) {
        post new_client(id, fd);
    }
    else {
        dev.owner = id;
        post write(id, fd);
    }
}
```

```
/*@ requires Precondition:
  @   id > 0 ∧
  @   dev.owner == id ∧
  @   ∀ id₁, fd₁;
  @     pending_write(id₁, fd₁)
  @       ⇒ id == id₁ ∧ fd == fd₁;
  @*/
async write(int id, int fd) {
    if (transfer(fd, dev)) {
        post write(id, fd);
    }
    else { // write complete
        dev.owner = 0;
    }
}
```