Rely/Guarantee Reasoning for Asynchronous Programs

Ivan Gavran\textsuperscript{1}, Filip Niksic\textsuperscript{1}, Aditya Kanade\textsuperscript{2}, Rupak Majumdar\textsuperscript{1}, Viktor Vafeiadis\textsuperscript{1}

\textsuperscript{1} Max Planck Institute for Software Systems (MPI-SWS), Germany
\textsuperscript{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

pending tasks

A

B
C
Common feature: Posting tasks for later execution

pending tasks
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

precondition $P_B$
Drawback: Obscured control-flow

precondition \( P_B \)

postcondition \( Q_A \)

\[ Q_A \Rightarrow P_B \]
Drawback: Obscured control-flow

postcondition $Q_A$ 

$Q_A \Rightarrow P_B$

precondition $P_B$

C might invalidate $P_B$
Adapting rely/guarantee reasoning [Jones83]
Adapting rely/guarantee reasoning [Jones83]

postcondition $Q_A \Rightarrow P_B$

precondition $P_B$

rely $R_B$: “preserve $P_B$”
Adapting rely/guarantee reasoning [Jones83]

- **A**
  - postcondition: $Q_A \Rightarrow P_B$
  - precondition: rely $R_B$: “preserve $P_B$”

- **C**
  - guarantee: $G_C \Rightarrow R_B$

- **B**

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(v_1, ..., v_k)
```

```
delete f(v_1, ..., v_k)
```

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

Execute instances atomically in a **non-deterministic order**.
Example: ROT13 server

```c
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

```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);
    }
}
```

```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

```c
//@ 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 $\text{posted}_f$

For each asynchronous procedure $f(x_1, \ldots, x_k)$, we introduce a predicate

$$\text{posted}_f(x_1, \ldots, x_k)$$

True iff $f$ has been posted with arguments $x_1, \ldots, x_k$ during the execution of the current asynchronous procedure.
Example: ROT13 server

```c
/*@ requires valid(c);
 ensure ∀c1;
 ensure ∀c1;
 ensure ∀c1;
 ensure ∀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);
 ensure ∀c1;
 ensure ∀c1;
 ensure ∀c1;
 ensure ∀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

\[ \text{read}(c) \]

\[ \text{write}(c) \]

\[ P_{\text{write}}(c) \equiv \text{valid}(c) \]

parent

child
Preserving the precondition

read(c₁)
write(c₁)
read(c) accept(socket) write(c)

\[ P_{\text{write}}(c) \equiv \text{valid}(c) \]

parent concurrent siblings child
Preserving the precondition

read\(c_1\)  
write\(c_1\)  
read\(c\)  
accept(socket)  
write\(c\)

guarantee \(P_{\text{write}}\) is preserved  

\[G_{\text{read}} \Rightarrow R_{\text{write}}\]  
\[G_{\text{write}} \Rightarrow R_{\text{write}}\]  
\[G_{\text{accept}} \Rightarrow R_{\text{write}}\]  

rely on \(P_{\text{write}}\) being preserved

parent  
concurrent siblings  
child
Introducing predicate pending\(_f\)

For each asynchronous procedure \(f(x_1, \ldots, x_k)\), we introduce a predicate

\[
pending_f(x_1, \ldots, x_k)
\]

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 $R_{write}$

$$R_{write} \equiv \forall c. (\text{pending'}_{write}(c) \land \text{pending}_{write}(c) \land \text{valid'}(c)) \Rightarrow \text{valid}(c)$$

(prime means at the beginning of execution)
write’s global invariant

With write’s parents ensuring:
\[ \forall c. \text{posted}_{\text{write}}(c) \Rightarrow \text{valid}(c) \]

and write’s concurrent siblings ensuring:
\[ \forall c. (\text{pending'}_{\text{write}}(c) \land \text{pending}_{\text{write}}(c) \land \text{valid'}(c)) \Rightarrow \text{valid}(c) \]

rely/guarantee ensures a global invariant:
\[ \forall c. \text{pending}_{\text{write}}(c) \Rightarrow \text{valid}(c) \]
Final specification of \texttt{write}

```c
/*@ \textbf{requires Precondition:} \\
@ \quad \texttt{valid}(c); \\
@ \textbf{ensures Parent\_child\_condition:} \\
@ \quad \forall c_1; \texttt{posted\_write}(c_1) \implies \texttt{valid}(c_1); \\
@ \textbf{ensures Guarantee:} \\
@ \quad (\forall c_1; (\texttt{pending\_read'}(c_1) \land \texttt{pending\_read}(c_1) \\
@ \quad \land \texttt{valid'}(c_1))) \implies \texttt{valid}(c_1)) \\
@ \land (\forall c_1; (\texttt{pending\_write'}(c_1) \land \texttt{pending\_write}(c_1) \\
@ \quad \land \texttt{valid'}(c_1))) \implies \texttt{valid}(c_1)); \\
@*/

\texttt{async write(\texttt{struct} client *c) \{ ... \}}
```
Final specification of write

```c
/*@ 
requires Precondition:
  @   valid(c);
  @
ensures Parent_child_condition:
  @   \forall c_1; posted_write(c_1) \Rightarrow valid(c_1);
  @
ensures Guarantee:
  @   (\forall c_1; (pending_read'(c_1) \land pending_read(c_1)
  @       \land valid'(c_1)) \Rightarrow valid(c_1))
  @
  @   \land (\forall c_1; (pending_write'(c_1) \land pending_write(c_1)
  @       \land valid'(c_1)) \Rightarrow valid(c_1));
  @*/

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

Final specification of `write`

```c
/*@ 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) { ... }
```

`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:

- $P_f$ — precondition predicate
- $R_f$ — rely predicate
- $G_f$ — guarantee predicate
- $Q_f$ — postcondition predicate

First-order formulas; may include predicates $\text{posted}_g$ and $\text{pending}_g$ (in negative positions)
Rely/guarantee conditions

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

(1) $Q_f \Rightarrow G_f$

(2) $Q_g \Rightarrow (\text{posted}_f \Rightarrow P_f)$, for each $g \in \text{parents}(f)$

(3) $R_f \Rightarrow ((\text{pending'}_f \land \text{pending}_f \land P'_f) \Rightarrow P_f)$

(4) $G_g \Rightarrow R_f$, for each $g \in \text{siblings}(f)$
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

\[
pending_f \Rightarrow P_f
\]
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:

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

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

- $Q_f \equiv G_f \land \bigwedge_{g \in \text{children}(f)} \text{posted}_g \Rightarrow P_g$
Generic rely/guarantee predicates

Sufficient for the ROT13 example:

```c
//@ 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:

```c
//@ 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);
    }
}
```

```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

\[ \text{post}_f(x_1, \ldots, x_k) \]

\[ \text{delete}_f(x_1, \ldots, x_k) \]

• 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

```c
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) { ... }
```
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;
    }
}
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

```plaintext
/*@ requires Precondition: 
   @ id > 0; 
   @ ensures Parent_child_write: 
   @    ∀ id_1, fd_1; 
   @      posted_write(id_1, fd_1) 
   @      ⇒ P_write(id_1, fd_1); 
   @*/
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 ∧ 
   @ ∀ id_1, fd_1; 
   @ pending_write(id_1, fd_1) 
   @ ⇒ id == id_1 ∧ fd == fd_1; 
   @*/
async write(int id, int fd) {
   if (transfer(fd, dev)) {
      post write(id, fd);
   } else { // write complete
      dev.owner = 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);
    }
}

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

```c
/*@ requires Precondition:
 @   id > 0;
 @ requires Global_inv_write:
 @   ∀ id, fd;
 @     pending_write(id, fd) ⇒ P_write(id, fd);
 @ ensures Parent_child_write:
 @   ∀ id, fd;
 @     posted_write(id, fd) ⇒ P_write(id, fd);
 @*/
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 ∧
 @   ∀ id, fd;
 @   pending_write(id, fd) ⇒ id == id ∧ fd == 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;
   @ 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;
    }
}