Modern Concurrent Separation Logics

Pedro da Rocha Pinto

Thomas
Dinsdale-Young

Philippa Gardner

Resource Reasoning 2016

Counter Module

```
function read(x) {
  r := [x];
  return r;
}
```

```
function incr(x) {
  do {
    r := [x];
    b := CAS(x, r, r + 1);
  } while (b = 0);
  return r;
}
```

```
function wkincr(x) {
  r := [x];
  [x] := r + 1;
}
```

Ticket Lock Client

```
function lock(x) {
  t := incr(x.next);
  do {
    v := read(x.owner)
  } while (v ≠ t);
}
```

```
function unlock(x) {
  wkincr(x.owner);
}
```

Sequential Specification

Abstract predicate $\mathsf{C}(\mathtt{x}, n)$ describes a counter with value $n \in \mathbb{N}$ at address $\mathtt{x}$.

$$\{\mathsf{C}(\mathtt{x}, n)\}\ \mathtt{read}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n) \wedge \mathsf{ret} = n\}$$
$$\{\mathsf{C}(\mathtt{x}, n)\}\ \mathtt{incr}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n+1) \wedge \mathsf{ret} = n\}$$
$$\{\mathsf{C}(\mathtt{x}, n)\}\ \mathtt{wkincr}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n+1)\}$$

Pros: The specification captures sequential behaviour.

Cons: The specification does not support concurrency.

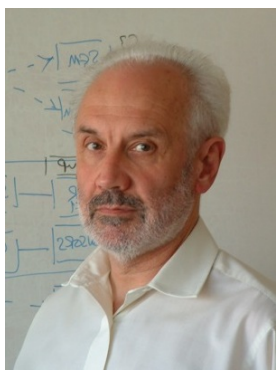
Specification of Concurrent Modules

Modular specifications of concurrent modules require:

- auxiliary state: Owicki, Gries
- interference abstraction: Jones
- resource ownership: O'Hearn
- atomicity: Herlithy

Susan Owicki

David Gries



Cliff Jones

Peter O'Hearn

Maurice Herlihy

Specification of Concurrent Modules

Modular specifications of concurrent modules require:

- auxiliary state
- interference abstraction
- resource ownership
- atomicity

Auxiliary State: Owicki-Gries

Parallel Rule

$$\frac{\{P_1\}\ \mathbb{C}_1\ \{Q_1\} \quad \{P_2\}\ \mathbb{C}_2\ \{Q_2\} \quad \text{non-interference}}{\{P_1 \wedge P_2\}\ \mathbb{C}_1 \parallel \mathbb{C}_2\ \{Q_1 \wedge Q_2\}}$$

Auxiliary State: Owicki-Gries

Concurrent Specification using Invariants

$$\{\exists n.\, \mathsf{C}(\mathtt{x}, n)\}\ \mathtt{read}(\mathtt{x})\ \{\exists n, m.\, \mathsf{C}(\mathtt{x}, n) \wedge \mathsf{ret} = m\}$$
$$\{\exists n.\, \mathsf{C}(\mathtt{x}, n)\}\ \mathtt{incr}(\mathtt{x})\ \{\exists n, m.\, \mathsf{C}(\mathtt{x}, n) \wedge \mathsf{ret} = m\}$$
$$\{\exists n.\, \mathsf{C}(\mathtt{x}, n)\}\ \mathtt{wkincr}(\mathtt{x})\ \{\exists n.\, \mathsf{C}(\mathtt{x}, n)\}$$

Pros: The specification captures concurrent behaviour.

Cons: Using invariants leads to very weak specifications. No information about the actual value of the counter.

Auxiliary State: Owicki-Gries

Stronger specifications require auxiliary state:

$$\left\{\ \mathtt{C}(\mathtt{x},0)\ \right\}$$

$$\mathtt{y} := 0;\ \mathtt{z} := 0;$$

$$\left\{\ \mathtt{C}(\mathtt{x},0) \wedge \mathtt{y} = 0 \wedge \mathtt{z} = 0\ \right\}$$

$$\begin{array}{l|l}
\left\{\ \mathtt{C}(\mathtt{x},\mathtt{y}+\mathtt{z}) \wedge \mathtt{y} = 0\ \right\} & \left\{\ \mathtt{C}(\mathtt{x},\mathtt{y}+\mathtt{z}) \wedge \mathtt{z} = 0\ \right\} \\
\mathtt{do}\ \{ & \mathtt{do}\ \{ \\
\quad \mathtt{r} := [\mathtt{x}]; & \quad \mathtt{r}' := [\mathtt{x}]; \\
\quad \langle \mathtt{b} := \mathtt{CAS}(\mathtt{x},\mathtt{r},\mathtt{r}+1); & \quad \langle \mathtt{b}' := \mathtt{CAS}(\mathtt{x},\mathtt{r}',\mathtt{r}'+1); \\
\qquad \mathtt{if}\ (\mathtt{b})\ \mathtt{y{+}{+}};\rangle & \qquad \mathtt{if}\ (\mathtt{b}')\ \mathtt{z{+}{+}};\rangle \\
\}\ \mathtt{while}\ (\mathtt{b} = 0); & \}\ \mathtt{while}\ (\mathtt{b}' = 0); \\
\left\{\ \mathtt{C}(\mathtt{x},\mathtt{y}+\mathtt{z}) \wedge \mathtt{y} = 1\ \right\} & \left\{\ \mathtt{C}(\mathtt{x},\mathtt{y}+\mathtt{z}) \wedge \mathtt{z} = 1\ \right\}
\end{array}$$

$$\left\{\ \mathtt{C}(\mathtt{x},2) \wedge \mathtt{y} = 1 \wedge \mathtt{z} = 1\ \right\}$$

$$\left\{\ \mathtt{C}(\mathtt{x},2)\ \right\}$$

Auxiliary State: Owicki-Gries

Pros: The specification is strong.

Cons: The proof method, introducing auxiliary code, is not modular.

Auxiliary State: Owicki-Gries

Auxiliary state, introduced in the Owicki-Gries method, is important for specifying concurrent modules.

Specification of Concurrent Modules

Modular specifications of concurrent modules require:

- auxiliary state
- interference abstraction
- resource ownership
- atomicity

Interference Abstraction: Rely/guarantee

Parallel Rule

The Rs and Gs are called rely and guarantee relations respectively.

$$\frac{R \cup G_2, G_1 \vdash \{P_1\}\ \mathbb{C}_1\ \{Q_1\} \quad R \cup G_1, G_2 \vdash \{P_2\}\ \mathbb{C}_2\ \{Q_2\}}{R, G_1 \cup G_2 \vdash \{P_1 \wedge P_2\}\ \mathbb{C}_1 \parallel \mathbb{C}_2\ \{Q_1 \wedge Q_2\}}$$

Interference Abstraction: Rely/guarantee

Specification: `read(x)` and `incr(x)`

$$A, \emptyset \vdash \{\exists n.\, \mathsf{C}(\mathtt{x}, n)\}\ \mathtt{read}(\mathtt{x})\ \{\exists n.\, \mathsf{C}(\mathtt{x}, n) \wedge \mathsf{ret} \leq n\}$$
$$A, A \vdash \{\exists n.\, \mathsf{C}(\mathtt{x}, n)\}\ \mathtt{incr}(\mathtt{x})\ \{\exists n.\, \mathsf{C}(\mathtt{x}, n) \wedge \mathsf{ret} \leq n\}$$

where $A = \{\mathsf{C}(\mathtt{x}, n) \rightsquigarrow \mathsf{C}(\mathtt{x}, n+1) \mid n \in \mathbb{N}\}$.

Pros: The behaviour that the environment might do is specified.

Cons: This specification does not restrict the increment operation to perform a single increment.

Ticket Lock Client

```
function lock(x) {
  t := incr(x.next);
  do {
    v := read(x.owner)
  } while (v ≠ t);
}
```

```
function unlock(x) {
  wkincr(x.owner);
}
```

Interference Abstraction: Rely/guarantee

Specification: `wkincr(x)`

$$R, G \vdash \left\{\mathsf{C}(\mathtt{x}, n)\right\}\ \mathtt{wkincr(x)}\ \left\{\exists n' \geq n+1.\, \mathsf{C}(\mathtt{x}, n')\right\}$$

where $R = \{\mathsf{C}(\mathtt{x}, m) \rightsquigarrow \mathsf{C}(\mathtt{x}, m+1) \mid m > n\}$ and G is as before.

Pros: This specification allows weak increments to occur concurrently.

Cons: This specification is too weak to reason about the ticket lock.

Interference Abstraction: Rely/guarantee

Interference abstraction, introduced in the rely/guarantee method, is important for specifying concurrent modules.

Specification of Concurrent Modules

Modular specifications of concurrent modules require:

- auxiliary state
- interference abstraction
- resource ownership
- atomicity

Resource Ownership: Concurrent Separation Logics

Parallel Rule

The disjoint concurrency rule from concurrent separation logic:

$$\frac{\{P_1\}\ \mathbb{C}_1\ \{Q_1\} \quad \{P_2\}\ \mathbb{C}_2\ \{Q_2\}}{\{P_1 * P_2\}\ \mathbb{C}_1 \parallel \mathbb{C}_2\ \{Q_1 * Q_2\}}$$

Ownership embodies specialised notions of auxiliary state and interference abstraction. If a thread owns a resource, then the environment cannot manipulate it.

Resource Ownership: Concurrent Separation Logics

Fractional Permissions

$$\mathsf{C}(\mathtt{x}, n_1 + n_2, \pi_1 + \pi_2) \iff \mathsf{C}(\mathtt{x}, n_1, \pi_1) * \mathsf{C}(\mathtt{x}, n_2, \pi_2)$$

for $n_1, n_2 \in \mathbb{N}$ and $\pi_1, \pi_2 \in (0, 1]$ and $\pi_1 + \pi_2 \leq 1$.

The abstract predicate $\mathsf{C}(\mathtt{x}, n, \pi_1)$ now means that this resource contributes value n to the counter at $\mathtt{x}$ with permission π_1.

Resource Ownership: Concurrent Separation Logics

Specification using Fractional Permissions

$$\{\mathsf{C}(\mathtt{x}, n, \pi)\}\ \mathtt{read}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n, \pi) \wedge \mathsf{ret} \geq n\}$$
$$\{\mathsf{C}(\mathtt{x}, n, 1)\}\ \mathtt{read}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n, 1) \wedge \mathsf{ret} = n\}$$
$$\{\mathsf{C}(\mathtt{x}, n, \pi)\}\ \mathtt{incr}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n+1, \pi) \wedge \mathsf{ret} \geq n\}$$
$$\{\mathsf{C}(\mathtt{x}, n, 1)\}\ \mathtt{wkincr}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n+1, 1)\}$$

Pros: This specification allows concurrent reads and increments.

Cons: The specification enforces sequential weak increments. The return values are no longer guaranteed to be the actual value of the counter.

Resource Ownership: Concurrent Separation Logics

$$
\begin{array}{c}
\left\{\ \mathsf{C}(\mathtt{x},0,1)\ \right\} \\
\left\{\ \mathsf{C}(\mathtt{x},0,0.5) * \mathsf{C}(\mathtt{x},0,0.5)\ \right\} \\
\begin{array}{l|l}
\left\{\ \mathsf{C}(\mathtt{x},0,0.5)\ \right\} & \left\{\ \mathsf{C}(\mathtt{x},0,0.5)\ \right\} \\
\mathtt{incr}(\mathtt{x}) & \mathtt{incr}(\mathtt{x}) \\
\left\{\ \mathsf{C}(\mathtt{x},1,0.5)\ \right\} & \left\{\ \mathsf{C}(\mathtt{x},1,0.5)\ \right\}
\end{array} \\
\left\{\ \mathsf{C}(\mathtt{x},1,0.5) * \mathsf{C}(\mathtt{x},1,0.5)\ \right\} \\
\left\{\ \mathsf{C}(\mathtt{x},2,1)\ \right\}
\end{array}
$$

Resource Ownership: Concurrent Separation Logics

Resource ownership, developed by concurrent separation logic and its successors, is important for specifying concurrent modules.

Specification of Concurrent Modules

Modular specifications of concurrent modules require:

- auxiliary state
- interference abstraction
- resource ownership
- atomicity

Atomicity

Atomicity provides a way to abstract interference. It gives the abstraction that an operation takes effect at a single, discrete instant in time.

Atomicity: Linearisability

Sequential Specification for `read(x)` and `incr(x)`

$$\{\mathsf{C}(\mathtt{x}, n)\}\ \mathtt{read}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n) \wedge \mathsf{ret} = n\}$$
$$\{\mathsf{C}(\mathtt{x}, n)\}\ \mathtt{incr}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n+1) \wedge \mathsf{ret} = n\}$$

Pros: Strong concurrent reasoning about `read(x)` and `incr(x)`.

Cons: Linearisability does not allow us to control the interference like rely-guarantee.

Atomicity: Linearisability

Sequential Specification with `wkincr(x)` not possible

$$\{\mathsf{C}(\mathtt{x}, n)\}\ \mathtt{read}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n) \wedge \mathsf{ret} = n\}$$
$$\{\mathsf{C}(\mathtt{x}, n)\}\ \mathtt{incr}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n+1) \wedge \mathsf{ret} = n\}$$
$$\{\mathsf{C}(\mathtt{x}, n)\}\ \mathtt{wkincr}(\mathtt{x})\ \{\mathsf{C}(\mathtt{x}, n+1)\}$$

Cons: The `wkincr` is not atomic when other increments occur.

Atomicity: Linearisability

Atomicity, put forward by linearisability, is important for specifying concurrent modules.

Synthesis

Now we combine auxiliary state, interference abstraction, resource reasoning and atomicity to provide expressive specifications for concurrent modules.

Synthesis

- higher-order: Birkedal, Dreyer, Jacobs, Turon
- history-based: Nanevski, Sergey
- first-order: da Rocha Pinto, Dinsdale-Young, Gardner


Lars Birkedal


Derek Dreyer


Bart Jacobs


Aleks Nanevski


Ilya Sergey


Aaron Turon

Synthesis: First-order Approach

A simple atomic triple has the form

$$\mathbb{A}x \in X.\, \langle P(x) \rangle\ \mathbb{C}\ \langle Q(x) \rangle$$

Synthesis: First-order Approach

Specification using Atomic Triples

$$\forall n.\, \langle \mathsf{C}(s, \mathtt{x}, n) \rangle\ \mathtt{read}(\mathtt{x})\ \langle \mathsf{C}(s, \mathtt{x}, n) \wedge \mathsf{ret} = n \rangle$$
$$\forall n.\, \langle \mathsf{C}(s, \mathtt{x}, n) \rangle\ \mathtt{incr}(\mathtt{x})\ \langle \mathsf{C}(s, \mathtt{x}, n+1) \wedge \mathsf{ret} = n \rangle$$
$$\langle \mathsf{C}(s, \mathtt{x}, n) \rangle\ \mathtt{wkincr}(\mathtt{x})\ \langle \mathsf{C}(s, \mathtt{x}, n+1) \rangle$$

Pros: Atomic triples specify operations with respect to an abstraction; each operation can be verified independently.

The specification is strong. It allows concurrent reads and concurrent increments, and concurrent reads and one weak increment.

Verfication of Implementations and Clients

Verification of the counter implementation.

Specification and verification of the ticket-lock client.

Pros: A specification of ticket lock whose implementation is based on the atomic counter commands.

Cons: The specification of ticket lock is not atomic. It requires helping (on-going).

Total Correctness using Ordinals

$$\forall \alpha. \vdash_\tau \{\mathsf{emp}\}\ \texttt{makeCounter}()\ \{\exists s. \mathsf{C}(s, \mathsf{ret}, 0, \alpha)\}$$

$$\vdash_\tau \Vdash\!\forall n \in \mathbb{N}, \alpha. \langle \mathsf{C}(s, \mathtt{x}, n, \alpha) \rangle\ \texttt{read}(\mathtt{x})\ \langle \mathsf{C}(s, \mathtt{x}, n, \alpha) \wedge \mathsf{ret} = n \rangle$$

$$\forall \beta. \vdash_\tau \Vdash\!\forall n \in \mathbb{N}, \alpha. \langle \mathsf{C}(s, \mathtt{x}, n, \alpha) \wedge \alpha > \beta(\alpha) \rangle\ \texttt{incr}(\mathtt{x})\ \langle \mathsf{C}(s, \mathtt{x}, n+1, \beta(\alpha)) \rangle$$

$$\forall \alpha > \beta. \mathsf{C}(s, x, n, \alpha) \implies \mathsf{C}(s, x, n, \beta)$$

Total Correctness using Ordinals

$$\forall \alpha. \vdash_\tau \{\mathsf{emp}\}\ \texttt{makeCounter}()\ \{\exists s. \mathsf{C}(s, \mathsf{ret}, 0, \alpha)\}$$

$$\vdash_\tau \Vdash\!\forall n \in \mathbb{N}, \alpha. \langle \mathsf{C}(s, \mathtt{x}, n, \alpha) \rangle\ \texttt{read}(\mathtt{x})\ \langle \mathsf{C}(s, \mathtt{x}, n, \alpha) \wedge \mathsf{ret} = n \rangle$$

$$\forall \beta. \vdash_\tau \Vdash\!\forall n \in \mathbb{N}, \alpha. \langle \mathsf{C}(s, \mathtt{x}, n, \alpha) \wedge \alpha > \beta(\alpha) \rangle\ \texttt{incr}(\mathtt{x})\ \langle \mathsf{C}(s, \mathtt{x}, n+1, \beta(\alpha)) \rangle$$

$$\forall \alpha > \beta. \mathsf{C}(s, x, n, \alpha) \implies \mathsf{C}(s, x, n, \beta)$$

Non-impedance relationship in the counter module:

Total Correctnesss: Example

$$
\begin{array}{l}
\{\ \mathsf{emp}\ \} \\
\texttt{x} := \texttt{makeCounter}(); \\
\{\ \exists s.\, \mathsf{C}(s, \texttt{x}, 0, \omega \oplus \omega)\ \} \\
\begin{array}{l|l}
\texttt{n} := \texttt{random}(); & \texttt{m} := \texttt{random}(); \\
\texttt{i} := 0; & \texttt{j} := 0; \\
\texttt{while}\ (\texttt{i} < \texttt{n})\ \{ & \texttt{while}\ (\texttt{j} < \texttt{m})\ \{ \\
\quad \texttt{incr}(\texttt{x}); & \quad \texttt{incr}(\texttt{x}); \\
\quad \texttt{i} := \texttt{i} + 1; & \quad \texttt{j} := \texttt{j} + 1; \\
\} & \}
\end{array} \\
\{\ \mathsf{C}(s, \texttt{x}, \texttt{n} + \texttt{m}, 0)\ \}
\end{array}
$$

A Personal History with Thomas and Pedro (and others)

- RGSep specification of concurrent B-trees
- Introduction of CAP to give abstraction: concurrent set/index module specified; B-tree implementation verified
- Specification not applicable to clients from e.g. `java.util.concurrent`!
- Introduction of TaDA to give abstract atomic, general specifications
- B-tree implementation and skip lists in `java.util.concurrent` verified
- Introduction of Total-TaDA for total correctness.

Pedro and Thomas