An Abstraction Technique for Describing Concurrent Program Behaviour

Wytse Oortwijn[1] Stefan Blom[1] Dilian Gurov[2]
Marieke Huisman[1] Marina Zaharieva-Stojanovski[1]

University of Twente, the Netherlands

KTH Royal Institute of Technology, Sweden

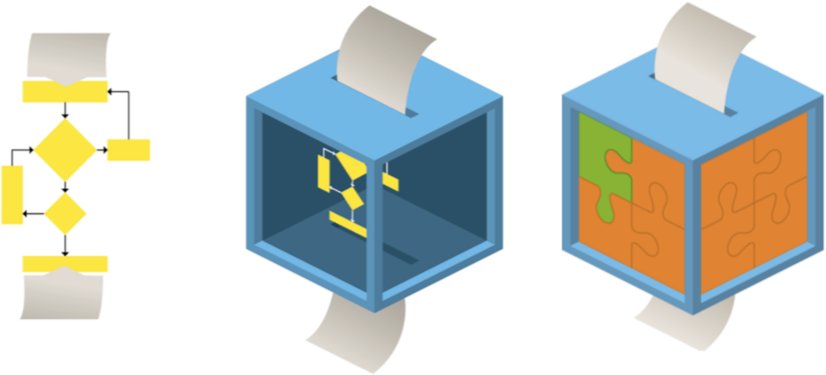
July 23, 2017

Outline

Outline

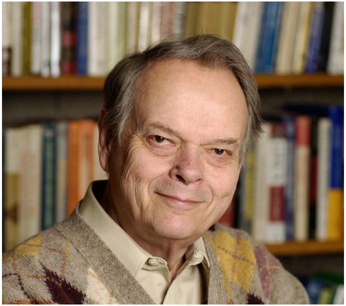
Therac-25 radiation therapy machine

Automated program verification



Deductive verification: underlying theory

Tony Hoare



John Reynolds

Hoare Triples *(Tony Hoare)*

$$\{\mathcal{P}\}\ S\ \{\mathcal{Q}\}$$

Hoare Triples *(Tony Hoare)*

$$\{\mathcal{P}\}\ S\ \{\mathcal{Q}\}$$

Formal meaning:

$$\forall h, \sigma \,.\; h, \sigma \models \mathcal{P}$$

Hoare Triples *(Tony Hoare)*

$$\{\mathcal{P}\}\ S\ \{\mathcal{Q}\}$$

Formal meaning:

$$\forall h, \sigma \ .\ \ h, \sigma \models \mathcal{P} \rightarrow (S, h, \sigma) \rightsquigarrow^* (\mathsf{skip}, h', \sigma')$$

Hoare Triples *(Tony Hoare)*

$$\{\mathcal{P}\}\ S\ \{\mathcal{Q}\}$$

Formal meaning:

$$\forall h, \sigma \,.\; h, \sigma \models \mathcal{P} \rightarrow (S, h, \sigma) \rightsquigarrow^* (\text{skip}, h', \sigma') \rightarrow h', \sigma' \models \mathcal{Q}$$

Separation Logic: ownership predicates *(Boyland)*

$$h, \sigma \models E \overset{\pi}{\hookrightarrow} E'$$

Separation Logic: ownership predicates *(Boyland)*

$$h, \sigma \models E \overset{\pi}{\hookrightarrow} E'$$

if and only if

$$h([\![E]\!]\sigma) = ([\![E']\!]\sigma, \pi') \text{ and } \pi \leq \pi'$$

**heaps are associated with permissions* $\pi \in (0, \ldots, 1]$

Separation Logic: separating conjunction

$$h, \sigma \models \mathcal{P}_1 * \mathcal{P}_2$$

Separation Logic: separating conjunction

$$h, \sigma \models \mathcal{P}_1 * \mathcal{P}_2$$

if and only if

Separation Logic: separating conjunction

$$h, \sigma \models \mathcal{P}_1 * \mathcal{P}_2$$

if and only if

$$h = h_1 \uplus h_2,$$

Separation Logic: separating conjunction

$$h, \sigma \models \mathcal{P}_1 * \mathcal{P}_2$$

if and only if

$h = h_1 \uplus h_2$, such that $h_1, \sigma \models \mathcal{P}_1$

Separation Logic: separating conjunction

$$h, \sigma \models \mathcal{P}_1 * \mathcal{P}_2$$

if and only if

$$h = h_1 \uplus h_2, \text{ such that } h_1, \sigma \models \mathcal{P}_1 \text{ and } h_2, \sigma \models \mathcal{P}_2$$

Separation Logic: the "small axioms"

Reading from shared memory:

$$\frac{x \notin \mathsf{fv}(E, E')}{\vdash \{\mathcal{P}[x/E'] \wedge E \xhookrightarrow{\pi} E'\}\, x := [E]\, \{\mathcal{P} \wedge E \xhookrightarrow{\pi} E'\}}$$

Separation Logic: the "small axioms"

Reading from shared memory:

$$\frac{x \notin \mathsf{fv}(E, E')}{\vdash \{\mathcal{P}[x/E'] \wedge E \overset{\pi}{\hookrightarrow} E'\}\; x := [E]\; \{\mathcal{P} \wedge E \overset{\pi}{\hookrightarrow} E'\}}$$

Writing to shared memory:

$$\frac{}{\vdash \{E \overset{1}{\hookrightarrow} -\}\; [E] := E'\; \{E \overset{1}{\hookrightarrow} E'\}}$$

Separation Logic: the "small axioms"

Reading from shared memory:

$$\frac{x \notin \mathsf{fv}(E, E')}{\vdash \{\mathcal{P}[x/E'] \wedge E \xhookrightarrow{\pi} E'\}\, x := [E]\, \{\mathcal{P} \wedge E \xhookrightarrow{\pi} E'\}}$$

Writing to shared memory:

$$\frac{}{\vdash \{E \xhookrightarrow{1} -\}\, [E] := E'\, \{E \xhookrightarrow{1} E'\}}$$

Splitting and merging ownership predicates:

$$E \xhookrightarrow{\pi_1 + \pi_2} E' \Leftrightarrow E \xhookrightarrow{\pi_1} E' * E \xhookrightarrow{\pi_2} E'$$

CSL: Concurrent Separation Logic *(Peter O'Hearn)*

$$\frac{\{\mathcal{P}_1\}\ S_1\ \{\mathcal{Q}_1\} \qquad \{\mathcal{P}_2\}\ S_2\ \{\mathcal{Q}_2\}}{\{\mathcal{P}_1 * \mathcal{P}_2\}\ S_1 \parallel S_2\ \{\mathcal{Q}_1 * \mathcal{Q}_2\}}$$

**plus some extra conditions, which are omitted*

CSL: Counting example

$$\overline{\qquad\qquad\qquad\qquad\qquad\qquad}$$

$$x := x + 2 \parallel y := y + 3$$

CSL: Counting example

$$\frac{}{\begin{array}{c}\{x \hookrightarrow X * y \hookrightarrow Y\} \\ x := x + 2 \parallel y := y + 3 \\ \{x \hookrightarrow X{+}2 * y \hookrightarrow Y{+}3\}\end{array}}$$

CSL: Counting example

$$\frac{\{x \hookrightarrow X\}\, x := x + 2\, \{x \hookrightarrow X{+}2\} \qquad \{y \hookrightarrow Y\}\, y := y + 3\, \{y \hookrightarrow Y{+}3\}}{\{x \hookrightarrow X * y \hookrightarrow Y\}\;\; x := x + 2 \parallel y := y + 3\;\; \{x \hookrightarrow X{+}2 * y \hookrightarrow Y{+}3\}}$$

Outline

Owicki-Gries: by auxiliary state

How to verify this program?

$$x := x + 1; \parallel x := x + 1;$$

Owicki-Gries: by auxiliary state

How to verify this program?

$$x := x + 1; \parallel x := x + 1;$$

$$\{x = 0\}$$
$$\{x = A + B \land A = 0 \land B = 0\}$$

$\{x = A + B \land A = 0\}$	$\parallel$	$\{x = A + B \land B = 0\}$
$x := x + 1;$	$\parallel$	$x := x + 1;$
ghost $\{ A := 1; \}$	$\parallel$	**ghost** $\{ B := 1; \}$
$\{x = A + B \land A = 1\}$	$\parallel$	$\{x = A + B \land B = 1\}$

$$\{x = A + B \land A = 1 \land B = 1\}$$
$$\{x = 2\}$$

Model-based Abstraction

Process algebras *(based on mCRL2)*

$$\mathsf{P} ::= \varepsilon \mid a(\overline{E}) \mid \mathsf{P} \cdot \mathsf{P} \mid \mathsf{P} + \mathsf{P} \mid \mathsf{P} \parallel \mathsf{P} \mid B \rightarrow P \diamond P \mid p(\overline{E})$$

Model-based Abstraction

Process algebras *(based on mCRL2)*

$$\mathrm{P} ::= \varepsilon \mid a(\overline{E}) \mid \mathrm{P} \cdot \mathrm{P} \mid \mathrm{P} + \mathrm{P} \mid \mathrm{P} \parallel \mathrm{P} \mid B \rightarrow P \diamond P \mid p(\overline{E})$$

Extending separation logic

- Process ownership predicates:

$$\mathrm{Proc}_{\pi}(m, p, \mathrm{P})$$

Model-based Abstraction

Process algebras *(based on mCRL2)*

$$P ::= \varepsilon \mid a(\overline{E}) \mid P \cdot P \mid P + P \mid P \parallel P \mid B \rightarrow P \diamond P \mid p(\overline{E})$$

Extending separation logic

- Process ownership predicates:

$$\mathsf{Proc}_{\pi}(m, p, \mathsf{P})$$

- Splitting & merging:

$$\mathsf{Proc}_{\pi_1+\pi_2}(m, p, \mathsf{P}_1 \parallel \mathsf{P}_2) \Leftrightarrow \mathsf{Proc}_{\pi_1}(m, p, \mathsf{P}_1) * \mathsf{Proc}_{\pi_2}(m, p, \mathsf{P}_2)$$

Owicki-Gries: by model-based abstraction

Program abstractions

- Process algebra with data *(based on mCRL2)*
- Capturing shared program state
- Built from *actions* with contracts *(guards and effects)*

Owicki-Gries: by model-based abstraction

Program abstractions

- Process algebra with data *(based on mCRL2)*
- Capturing shared program state
- Built from *actions* with contracts *(guards and effects)*

Action definition

action incr;

Owicki-Gries: by model-based abstraction

Program abstractions

- Process algebra with data *(based on mCRL2)*
- Capturing shared program state
- Built from *actions* with contracts *(guards and effects)*

Action definition

action incr;

Process definition

process parincr := incr $\|$ incr;

Owicki-Gries: by model-based abstraction

Program abstractions

- Process algebra with data *(based on mCRL2)*
- Capturing shared program state
- Built from *actions* with contracts *(guards and effects)*

Action definition

guard `true`;
effect $x = \textbf{old}(x) + 1$;
action incr;

Process definition

process parincr := incr ‖ incr;

Owicki-Gries: by model-based abstraction

Program abstractions

- Process algebra with data *(based on mCRL2)*
- Capturing shared program state
- Built from *actions* with contracts *(guards and effects)*

Action definition

guard `true`;

effect $\mathtt{x} = \textbf{old}(\mathtt{x}) + 1$;

action incr;

Process definition

requires `true`;

ensures $\mathtt{x} = \textbf{old}(\mathtt{x}) + 2$;

process parincr := incr ∥ incr;

Owicki-Gries: by model-based abstraction

$$\{x \overset{1}{\hookrightarrow} 0\}$$

$$m := \textbf{init}\ \text{parincr}\ \textbf{over}\ x;$$

$$\{\text{Proc}_1(m, \text{parincr})\}$$

$$\{\text{Proc}_1(m, \text{incr} \parallel \text{incr})\}$$

$$\{\text{Proc}_{1/2}(m, \text{incr}) * \text{Proc}_{1/2}(m, \text{incr})\}$$

$$\{\text{Proc}_{1/2}(m, \text{incr})\} \quad \parallel \quad \{\text{Proc}_{1/2}(m, \text{incr})\}$$

$$\textbf{action}\ m.\text{incr}\ \{\ x := x + 1;\ \} \quad \parallel \quad \textbf{action}\ m.\text{incr}\ \{\ x := x + 1;\ \}$$

$$\{\text{Proc}_{1/2}(m, \varepsilon)\} \quad \parallel \quad \{\text{Proc}_{1/2}(m, \varepsilon)\}$$

$$\{\text{Proc}_{1/2}(m, \varepsilon) * \text{Proc}_{1/2}(m, \varepsilon)\}$$

$$\{\text{Proc}_1(m, \varepsilon \parallel \varepsilon)\}$$

$$\{\text{Proc}_1(m, \varepsilon)\}$$

$$\textbf{destroy}\ m;$$

$$\{x \overset{1}{\hookrightarrow} 2\}$$

Model-based verification

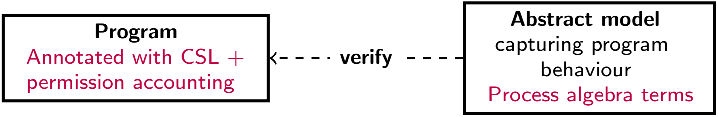
Program
Annotated with CSL +
permission accounting

Model-based verification

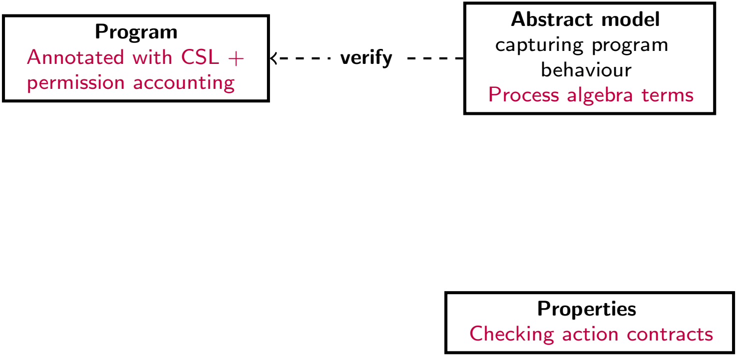
Program
Annotated with CSL + permission accounting

Abstract model
capturing program behaviour
Process algebra terms

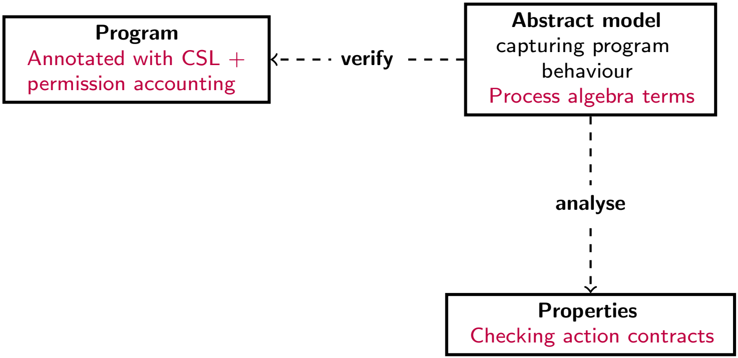
Model-based verification



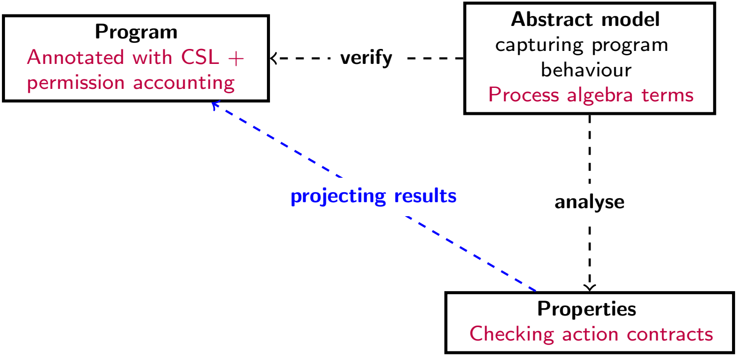
Model-based verification



Model-based verification



Model-based verification



CSL with model abstractions

Points-to predicates

- *Standard* points-to predicates: $E \overset{\pi}{\hookrightarrow}_{\mathsf{n}} E$
- *Process* points-to predicates: $E \overset{\pi}{\hookrightarrow}_{\mathsf{p}} E$
- *Action* points-to predicates: $E \overset{\pi}{\hookrightarrow}_{\mathsf{a}} E$

CSL with model abstractions

Points-to predicates

- *Standard* points-to predicates: $E \xhookrightarrow{\pi}_{\mathsf{n}} E$ *(reading)*
- *Process* points-to predicates: $E \xhookrightarrow{\pi}_{\mathsf{p}} E$ *(reading)*
- *Action* points-to predicates: $E \xhookrightarrow{\pi}_{\mathsf{a}} E$ *(reading)*

$$\frac{x \notin \mathsf{fv}(E, E')}{\vdash \{\mathcal{P}[x/E'] \wedge E \xhookrightarrow{\pi}_{t} E'\}\, x := [E]\, \{\mathcal{P} \wedge E \xhookrightarrow{\pi}_{t} E'\}}$$

CSL with model abstractions

Points-to predicates

- *Standard* points-to predicates: $E \xhookrightarrow{\pi}_{\mathsf{n}} E$ *(reading + writing)*
- *Process* points-to predicates: $E \xhookrightarrow{\pi}_{\mathsf{p}} E$ *(read-only!)*
- *Action* points-to predicates: $E \xhookrightarrow{\pi}_{\mathsf{a}} E$ *(reading + writing)*

$$\frac{x \notin \mathsf{fv}(E, E')}{\vdash \{\mathcal{P}[x/E'] \wedge E \xhookrightarrow{\pi}_{t} E'\}\, x := [E]\, \{\mathcal{P} \wedge E \xhookrightarrow{\pi}_{t} E'\}}$$

$$\frac{t \neq \mathsf{p}}{\vdash \{E \xhookrightarrow{1}_{t} -\}\, [E] := E'\, \{E \xhookrightarrow{1}_{t} E'\}}$$

CSL extensions: INIT rule *(simplified)*

$$\frac{B = \text{precondition}(p)}{\begin{array}{c}\vdash \{\ast_{i=0..n} E_i \overset{1}{\hookrightarrow} E_i' \ast B\} \\ m := \textbf{init } p() \textbf{ over } E_0, \ldots, E_n \\ \{\ast_{i=0..n} E_i \overset{1}{\hookrightarrow}_{\mathsf{p}} E_i' \ast \text{Proc}_1(m, p, \text{body}(p))\}\end{array}}$$

CSL extensions: Init rule *(simplified)*

$$\frac{B = \text{precondition}(p)}{\begin{array}{c}\vdash \{\ast_{i=0..n} E_i \overset{1}{\hookrightarrow} E'_i \ast B\} \\ m := \textbf{init}\ p()\ \textbf{over}\ E_0, \ldots, E_n \\ \{\ast_{i=0..n} E_i \overset{1}{\hookrightarrow}_{\mathsf{p}} E'_i \ast \text{Proc}_1(m, p, \text{body}(p))\}\end{array}}$$

Rule details

1. Standard points-to predicates are converted

CSL extensions: Init rule *(simplified)*

$$\frac{B = \text{precondition}(p)}{\begin{array}{c}\vdash \{\ast_{i=0..n} E_i \xhookrightarrow{1} E_i' \ast B\} \\ m := \textbf{init}\ p()\ \textbf{over}\ E_0, \ldots, E_n \\ \{\ast_{i=0..n} E_i \xhookrightarrow{1}_{\mathsf{p}} E_i' \ast \text{Proc}_1(m, p, \text{body}(p))\}\end{array}}$$

Rule details

1. Standard points-to predicates are converted
2. Precondition of the process must hold

CSL extensions: INIT rule *(simplified)*

$$\frac{B = \text{precondition}(p)}{\begin{array}{c}\vdash \{\ast_{i=0..n} E_i \overset{1}{\hookrightarrow} E_i' \ast B\} \\ m := \textbf{init } p() \textbf{ over } E_0, \ldots, E_n \\ \{\ast_{i=0..n} E_i \overset{1}{\hookrightarrow}_{\mathsf{p}} E_i' \ast \mathsf{Proc}_1(m, p, \mathsf{body}(p))\}\end{array}}$$

Rule details

1. Standard points-to predicates are converted
2. Precondition of the process must hold
3. Process ownership predicate is constructed

CSL extensions: FINISH rule *(simplified)*

$$\frac{\text{accessible}(p) = E_0, \ldots, E_n \qquad B = \text{postcondition}(p)}{\vdash \{\ast_{i=0..n} E_i \overset{1}{\hookrightarrow}_{\mathsf{p}} E_i' \ast \text{Proc}_1(m, p, \varepsilon)\}\ \textbf{finish}\ m\, \{\ast_{i=0..n} E_i \overset{1}{\hookrightarrow}_{\mathsf{n}} E_i' \ast B\}}$$

CSL extensions: FINISH rule *(simplified)*

$$\frac{\text{accessible}(p) = E_0, \ldots, E_n \qquad B = \text{postcondition}(p)}{\vdash \{\ast_{i=0..n} E_i \overset{1}{\hookrightarrow}_{\mathsf{p}} E_i' \ast \text{Proc}_1(m, p, \varepsilon)\}\ \textbf{finish}\ m\ \{\ast_{i=0..n} E_i \overset{1}{\hookrightarrow}_{\mathsf{n}} E_i' \ast B\}}$$

Rule details

1. Process points-to predicates are converted back;

CSL extensions: Finish rule *(simplified)*

$$\frac{\text{accessible}(p) = E_0, \ldots, E_n \qquad B = \text{postcondition}(p)}{\vdash \{\ast_{i=0..n} E_i \overset{1}{\hookrightarrow}_{\mathsf{p}} E_i' \ast \mathsf{Proc}_1(m, p, \varepsilon)\} \ \textbf{finish} \ m \ \{\ast_{i=0..n} E_i \overset{1}{\hookrightarrow}_{\mathsf{n}} E_i' \ast B\}}$$

Rule details

1. Process points-to predicates are converted back;
2. Full process predicate is handed in; and

CSL extensions: FINISH rule *(simplified)*

$$\frac{\text{accessible}(p) = E_0, \ldots, E_n \qquad B = \text{postcondition}(p)}{\vdash \{\ast_{i=0..n} E_i \overset{1}{\hookrightarrow}_{\mathsf{p}} E_i' \ast \text{Proc}_1(m, p, \varepsilon)\} \ \textbf{finish} \ m \ \{\ast_{i=0..n} E_i \overset{1}{\hookrightarrow}_{\mathsf{n}} E_i' \ast B\}}$$

Rule details

1. Process points-to predicates are converted back;
2. Full process predicate is handed in; and
3. The process postcondition is ensured!

CSL extensions: Action rule *(simplified)*

$$\frac{\text{modifies}(S) = E_0, \ldots, E_n \qquad B_1 = \text{guard}(a) \qquad B_2 = \text{effect}(a) \\ \vdash \{\ast_{i=0..n} E_i \xrightarrow{\pi_i}_{\mathsf{a}} E_i' \ast B_1\}\, S\, \{\ast_{i=0..n} E_i \xrightarrow{\pi_i}_{\mathsf{a}} E_i'' \ast B_2\}}{\begin{array}{c}\vdash \{\ast_{i=0..n} E_i \xrightarrow{\pi_i}_{\mathsf{p}} E_i' \ast \text{Proc}_\pi(m, p, a(\overline{E}) \cdot \mathsf{P}) \ast B_1\} \\ \textbf{action } m.a(\overline{E})\ \{\ S\ \} \\ \{\ast_{i=0..n} E_i \xrightarrow{\pi_i}_{\mathsf{p}} E_i'' \ast \text{Proc}_\pi(m, p, \mathsf{P}) \ast B_2\}\end{array}}$$

CSL extensions: ACTION rule *(simplified)*

$$\frac{\text{modifies}(S) = E_0, \ldots, E_n \qquad B_1 = \text{guard}(a) \qquad B_2 = \text{effect}(a) \\ \vdash \{\mathop{\ast}_{i=0..n} E_i \overset{\pi_i}{\hookrightarrow}_{\mathsf{a}} E_i' \ast B_1\}\ S\ \{\mathop{\ast}_{i=0..n} E_i \overset{\pi_i}{\hookrightarrow}_{\mathsf{a}} E_i'' \ast B_2\}}{\vdash \{\mathop{\ast}_{i=0..n} E_i \overset{\pi_i}{\hookrightarrow}_{\mathsf{p}} E_i' \ast \text{Proc}_\pi(m, p, a(\overline{E}) \cdot \mathrm{P}) \ast B_1\} \\ \textbf{action}\ m.a(\overline{E})\ \{\ S\ \} \\ \{\mathop{\ast}_{i=0..n} E_i \overset{\pi_i}{\hookrightarrow}_{\mathsf{p}} E_i'' \ast \text{Proc}_\pi(m, p, \mathrm{P}) \ast B_2\}}$$

Rule details

1. Process points-to predicates are converted;

CSL extensions: ACTION rule *(simplified)*

$$\frac{\text{modifies}(S) = E_0, \ldots, E_n \qquad B_1 = \text{guard}(a) \qquad B_2 = \text{effect}(a) \qquad \vdash \{\ast_{i=0..n} E_i \xhookrightarrow{\pi_i}_{\mathsf{a}} E_i' \ast B_1\}\ S\ \{\ast_{i=0..n} E_i \xhookrightarrow{\pi_i}_{\mathsf{a}} E_i'' \ast B_2\}}{\vdash \{\ast_{i=0..n} E_i \xhookrightarrow{\pi_i}_{\mathsf{p}} E_i' \ast \text{Proc}_\pi(m, p, a(\overline{E}) \cdot \mathsf{P}) \ast B_1\}\ \textbf{action}\ m.a(\overline{E})\ \{\ S\ \}\ \{\ast_{i=0..n} E_i \xhookrightarrow{\pi_i}_{\mathsf{p}} E_i'' \ast \text{Proc}_\pi(m, p, \mathsf{P}) \ast B_2\}}$$

Rule details

1. Process points-to predicates are converted;
2. The action guard and effect must hold; and

CSL extensions: Action rule *(simplified)*

$$\frac{\text{modifies}(S) = E_0, \ldots, E_n \qquad B_1 = \text{guard}(a) \qquad B_2 = \text{effect}(a) \\ \vdash \{\bigstar_{i=0..n} E_i \overset{\pi_i}{\hookrightarrow}_{\text{a}} E_i' * B_1\}\ S\ \{\bigstar_{i=0..n} E_i \overset{\pi_i}{\hookrightarrow}_{\text{a}} E_i'' * B_2\}}{\vdash \{\bigstar_{i=0..n} E_i \overset{\pi_i}{\hookrightarrow}_{\text{p}} E_i' * \text{Proc}_\pi(m, p, a(\overline{E}) \cdot \text{P}) * B_1\} \\ \textbf{action}\ m.a(\overline{E})\ \{\ S\ \} \\ \{\bigstar_{i=0..n} E_i \overset{\pi_i}{\hookrightarrow}_{\text{p}} E_i'' * \text{Proc}_\pi(m, p, \text{P}) * B_2\}}$$

Rule details

1. Process points-to predicates are converted;
2. The action guard and effect must hold; and
3. The process predicate is updated.

Outline

Verification example: greatest common divisor

Standard Euclidean algorithm

Given two positive integers x and y:

$$\begin{aligned} \gcd(x, x) &= x \\ \gcd(x, y) &= \gcd(x - y, y) \text{ if } x > y \\ \gcd(x, y) &= \gcd(x, y - x) \text{ if } y > x \end{aligned}$$

Parallel GCD *(from VerifyThis 2015)*

```
shared int x, y;

void threadx() {
   bool stop := false;
   while ¬stop do {
      acquire lock;
      if (x > y) { x := x − y; }
      stop := x = y;
      release lock;
   }
}

void thready() {
   bool stop := false;
   while ¬stop do {
      acquire lock;
      if (y > x) { y := y − x; }
      stop := x = y;
      release lock;
   }
}

int startgcd(int a, int b) {
   x := a;  y := b;
   init lock;
   handle t1 := fork threadx();
   handle t2 := fork thready();
   join t1;
   join t2;
   destroy lock;
   return x;
}
```

Parallel GCD: process algebraic description

```
shared int x, y;

guard x > 0 ∧ y > x
effect x = old(x) ∧ y = old(y) − old(x)
action decrx;

guard y > 0 ∧ x > y
effect x = old(x) − old(y) ∧ y = old(y)
action decry;

guard x = y
action done;

process tx() := decrx · tx() + done;
process ty() := decry · ty() + done;

requires x > 0 ∧ y > 0
ensures x = y
ensures x = gcd(old(x), old(y))
process pargcd() := tx() ∥ ty();
```

Parallel GCD: entry point

```
resource lock := ∃v1, v2 : v1 > 0 ∗
    v2 > 0 ∗ x ↪(1)p v1 ∗ y ↪(1)p v2;

requires a > 0 ∧ b > 0
ensures x = y ∧ x = gcd(a, b)
void startgcd(int a, int b) {
    x := a; y := b;
    m := init pargcd over x, y;
    init lock;
    handle t1 := fork threadx(m);
    handle t2 := fork thready(m);
    join t1;
    join t2;
    destroy lock;
    finish m;
}
```

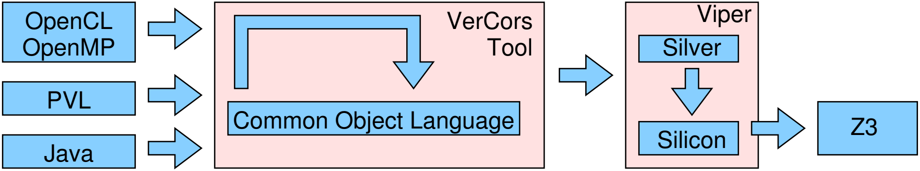
Parallel GCD: thread annotations

```
requires Lock_π(lock)
requires Proc_1/2(m, tx())
ensures Lock_π(lock)
ensures Proc_1/2(m, ε)
void threadx(model m) {
    bool stop := false;
    loop-inv Lock_π(lock);
    loop-inv ¬stop ⇒ Proc_1/2(m, tx());
    loop-inv stop ⇒ Proc_1/2(m, ε);
    while ¬stop do {
      acquire lock;
      if (x > y) {
        action m.decrx() { x := x − y; }
      }
      if (x = y) {
        action m.done() { stop := true; }
      }
      release lock;
    }
}
```

```
requires Lock_π(lock)
requires Proc_1/2(m, ty())
ensures Lock_π(lock)
ensures Proc_1/2(m, ε)
void thready(model m) {
    bool stop := false;
    loop-inv Lock_π(lock);
    loop-inv ¬stop ⇒ Proc_1/2(m, ty());
    loop-inv stop ⇒ Proc_1/2(m, ε);
    while ¬stop do {
      acquire lock;
      if (y > x) {
        action m.decry() { y := y − x; }
      }
      if (x = y) {
        action m.done() { stop := true; }
      }
      release lock;
    }
}
```

Outline

The VerCors Toolset: Overview



The VerCors Toolset: Achievements

Current support

- Reasoning about compiler directives *(OpenMP for C)*
- Reasoning about GPU kernels *(OpenCL)*
- Reasoning with program abstractions

The VerCors Toolset: Achievements

Current support

- Reasoning about compiler directives *(OpenMP for C)*
- Reasoning about GPU kernels *(OpenCL)*
- Reasoning with program abstractions

Ongoing work

- Inferring permissions/ownership predicates
- Building support for distributed software *(e.g. MPI)*
- Runtime verification

The VerCors Toolset: Future directions

requires $\cdots$
requires Process(P(k) $\cdot$ Q)
ensures $\cdots$
ensures Process(Q)
void `main`(**int** k):
    int $v \leftarrow$ `MPI_Recv`($\star$)
    `MPI_Send`(0, $v + k$)

The VerCors Toolset: Future directions

requires $\cdots$
requires Process(P(k) $\cdot$ Q)
ensures $\cdots$
ensures Process(Q)
void main(**int** k):
    int $v \leftarrow$ MPI_Recv($\star$)
    MPI_Send(0, $v + k$)

— 1 →

The VerCors Toolset: Future directions

process P(**int** k) $\equiv$

$\Sigma_{i:\textbf{int}} \cdot \textbf{recv}(\star, i) \cdot \textbf{send}(0, i + k)$

requires · · ·
requires Process(P(k) · Q)
ensures · · ·
ensures Process(Q)
void main(**int** k):
  int v ← MPI_Recv(⋆)
  MPI_Send(0, $v + k$)

1 →

2 ↓

The VerCors Toolset: Future directions

process P(**int** k) $\equiv$

$\Sigma_{i:\textbf{int}} \cdot \textbf{recv}(\star, i) \cdot \textbf{send}(0, i + k)$

2

requires $\cdots$
requires Process(P(k) $\cdot$ Q)
ensures $\cdots$
ensures Process(Q)
void `main`(**int** `k`):
    int $v \leftarrow$ `MPI_Recv`$(\star)$
    `MPI_Send`$(0,\ v + k)$

1

3

Verdict
Does the program correctly executes the process term?
Hoare logic reasoning

The VerCors Toolset: Future directions

process P(**int** k) $\equiv$

$\Sigma_{i:\mathbf{int}} \cdot$ **recv**$(\star, i) \cdot$ **send**$(0, i + k)$

2

requires $\cdots$
requires Process(P(k) $\cdot$ Q)
ensures $\cdots$
ensures Process(Q)
void `main`(**int** k):
    int $v \leftarrow$ `MPI_Recv`$(\star)$
    `MPI_Send`$(0, v + k)$

1

3

Verdict
Does the program correctly executes the process term?
Hoare logic reasoning

$\mathsf{P}(k_1) \parallel \mathsf{P}(k_2) \parallel \cdots \parallel \mathsf{MPI}$

The VerCors Toolset: Future directions

process P(**int** k) $\equiv$

$\Sigma_{i:\textbf{int}} \cdot \textbf{recv}(\star, i) \cdot \textbf{send}(0, i + k)$

2

requires $\cdots$
requires Process(P(k) · Q)
ensures $\cdots$
ensures Process(Q)
void `main`(**int** k):
  int $v \leftarrow$ `MPI_Recv`($\star$)
  `MPI_Send`(0, $v + k$)

1

3

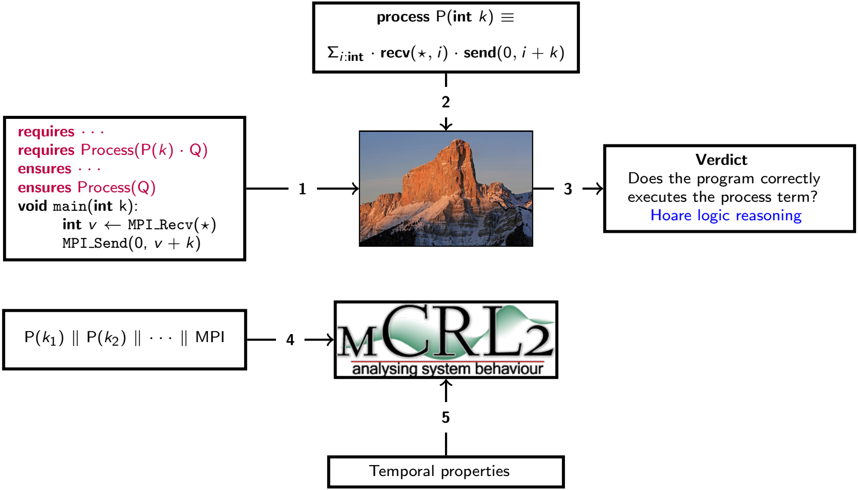
Verdict
Does the program correctly executes the process term?
Hoare logic reasoning

$P(k_1) \parallel P(k_2) \parallel \cdots \parallel MPI$

4

mCRL2
analysing system behaviour

The VerCors Toolset: Future directions



The VerCors Toolset: Future directions

process P(**int** k) $\equiv$

$\Sigma_{i:\textbf{int}} \cdot \textbf{recv}(\star, i) \cdot \textbf{send}(0, i + k)$

2

requires · · ·
requires Process(P(k) · Q)
ensures · · ·
ensures Process(Q)

```
void main(int k):
    int v ← MPI_Recv(⋆)
    MPI_Send(0, v + k)
```

1

3

Verdict
Does the program correctly executes the process term?
Hoare logic reasoning

6

$P(k_1) \parallel P(k_2) \parallel \cdots \parallel MPI$

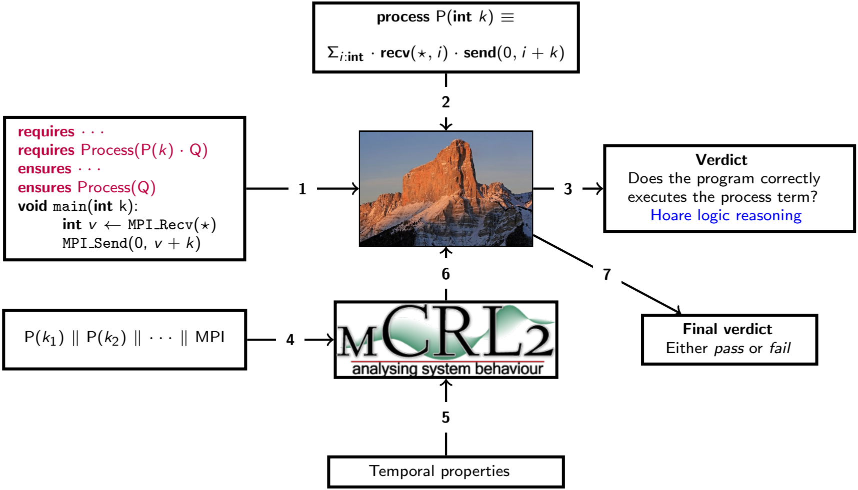
4

mCRL2 analysing system behaviour

5

Temporal properties

The VerCors Toolset: Future directions



Outline

Conclusion

Describing Concurrent Program Behaviour

- Using process algebras to describe changes to shared state.
- Combining deductive verification and algorithmic reasoning.

VerCors Verification Toolset

- Automated verification of parallel and concurrent software
- More information: `http://utwente.nl/vercors`
- Download: `https://github.com/utwente-fmt/vercors`