An Abstraction Technique for Describing Concurrent Program Behaviour

Wytse Oortwijn
(joint with Stefan Blom and Marieke Huisman)

Formal Methods and Tools, University of Twente

September 19, 2017

Outline

Outline

Concurrent programming is *error-prone*

THERAC-25

Concurrent programming is *error-prone*

THERAC-25

Intel TSX

Concurrent programming is *error-prone*

THERAC-25

Intel TSX

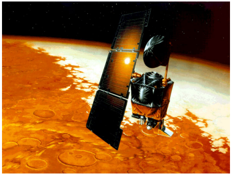
Toyota car acceleration

Concurrent programming is *error-prone*

THERAC-25

Intel TSX

Toyota car acceleration



Mars climate orbiter

Concurrent system behaviour

Single traffic light

only a *<u>few</u>* *possible behaviours:*

red, orange, and green

Concurrent system behaviour

Single traffic light

only a few possible behaviours:

red, orange, and green

Multiple traffic lights

the number of possible behaviours is exponential in the number of lights

Static verification: *deductive verification*

```
void main(args) {
    S1 || S2;
}
```

How to verify a concurrent program?

Static verification: *deductive verification*

$$\begin{array}{l} \textbf{void}\ \texttt{main}(args)\ \{ \\ \quad S_1 \parallel S_2; \\ \} \end{array}$$

$$\dfrac{\dfrac{\dfrac{\cdots}{\vdash \{P_1\}\, S_1\, \{Q_1\}} \qquad \dfrac{\cdots}{\vdash \{P_2\}\, S_2\, \{Q_2\}}}{\vdash \{P_1 * P_2\}\, S_1 \parallel S_2\, \{Q_1 * Q_2\}}}{\vdash \{P_1 * P_2\}\, \texttt{main}(args)\, \{Q_1 * Q_2\}}$$

How to verify a concurrent program?

Using program logics, like: Hoare logic, separation logic, etc.

Static verification: *deductive verification*

requires $P_1 * P_2$;
ensures $Q_1 * Q_2$;
void `main`(*args*) {
    $S_1 \parallel S_2$;
}

$$\dfrac{\dfrac{\dfrac{\cdots}{\vdash \{P_1\}\, S_1\, \{Q_1\}} \qquad \dfrac{\cdots}{\vdash \{P_2\}\, S_2\, \{Q_2\}}}{\vdash \{P_1 * P_2\}\, S_1 \parallel S_2\, \{Q_1 * Q_2\}}}{\vdash \{P_1 * P_2\}\, \texttt{main}(args)\, \{Q_1 * Q_2\}}$$

How to verify a concurrent program? (using automated tools)

Using program logics, like: Hoare logic, separation logic, etc.

Static verification: *deductive verification*

requires $P_1 * P_2$;
ensures $Q_1 * Q_2$;
void `main`(*args*) {
$S_1 \parallel S_2$;
}

$$\dfrac{\dfrac{\dfrac{\cdots}{\vdash \{P_1\}\, S_1\, \{Q_1\}} \qquad \dfrac{\cdots}{\vdash \{P_2\}\, S_2\, \{Q_2\}}}{\vdash \{P_1 * P_2\}\, S_1 \parallel S_2\, \{Q_1 * Q_2\}}}{\vdash \{P_1 * P_2\}\, \texttt{main}(args)\, \{Q_1 * Q_2\}}$$

How to verify a concurrent program? (using automated tools)

Using program logics, like: Hoare logic, separation logic, etc.

Strong correctness guarantees

Static verification: *deductive verification*

```
requires P1 * P2;
ensures Q1 * Q2;
void main(args) {
    S1 || S2;
}
```

$$\dfrac{\dfrac{\dfrac{\cdots}{\vdash \{P_1\}\, S_1\, \{Q_1\}} \qquad \dfrac{\cdots}{\vdash \{P_2\}\, S_2\, \{Q_2\}}}{\vdash \{P_1 * P_2\}\, S_1 \parallel S_2\, \{Q_1 * Q_2\}}}{\vdash \{P_1 * P_2\}\, \mathtt{main}(args)\, \{Q_1 * Q_2\}}$$

How to verify a concurrent program? (using automated tools)

Using program logics, like: Hoare logic, separation logic, etc.

Hard to construct a proof

Static verification: *model checking*

```
void main(args) {
    S1 || S2;
}
```

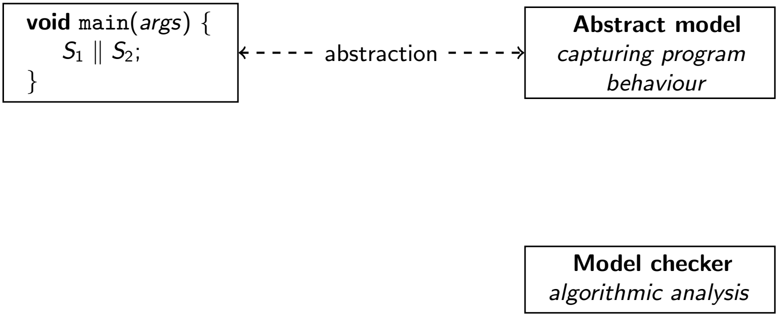
Static verification: *model checking*

```
void main(args) {
    S1 || S2;
}
```

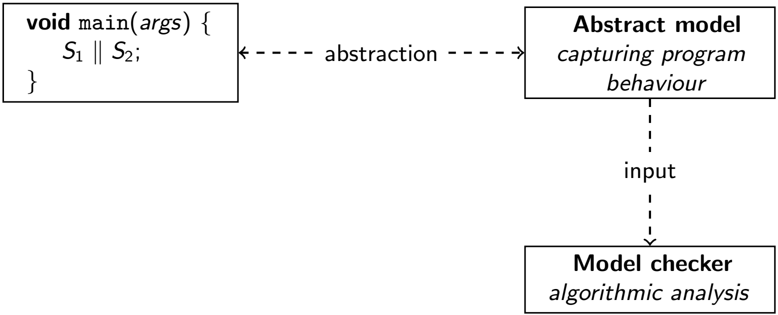
Abstract model
capturing program behaviour

Static verification: *model checking*

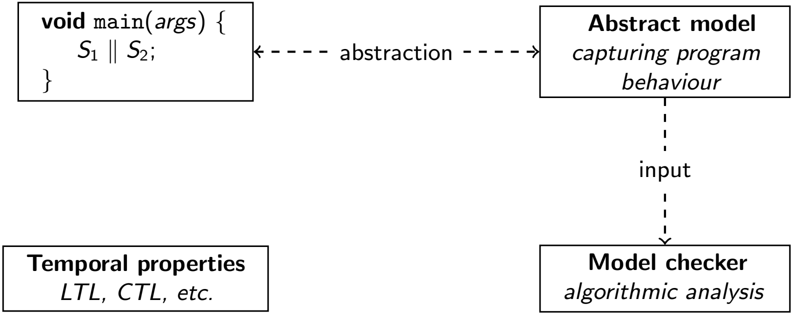
Static verification: *model checking*



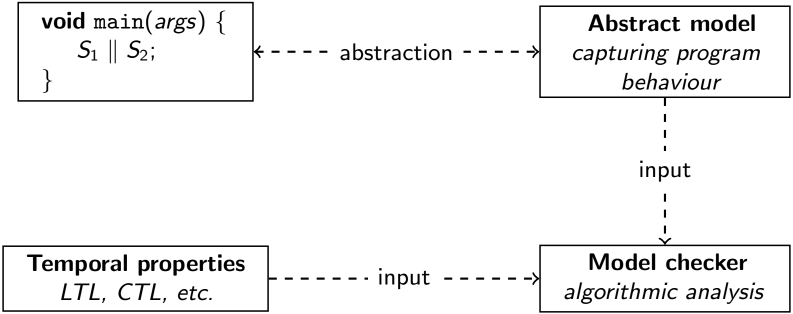
Static verification: *model checking*



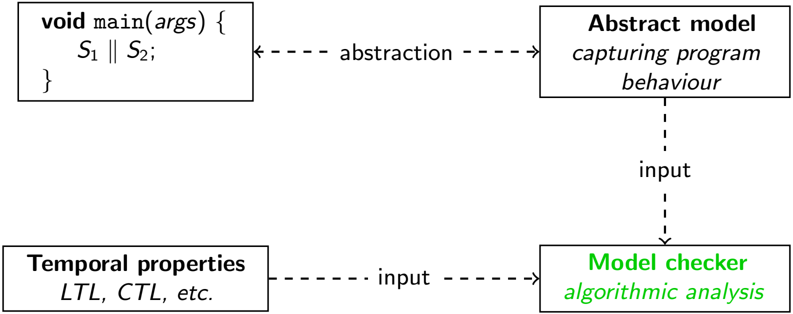
Static verification: *model checking*



Static verification: *model checking*

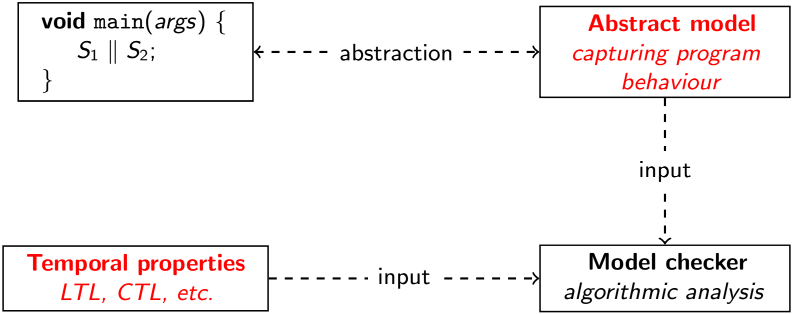


Static verification: *model checking*



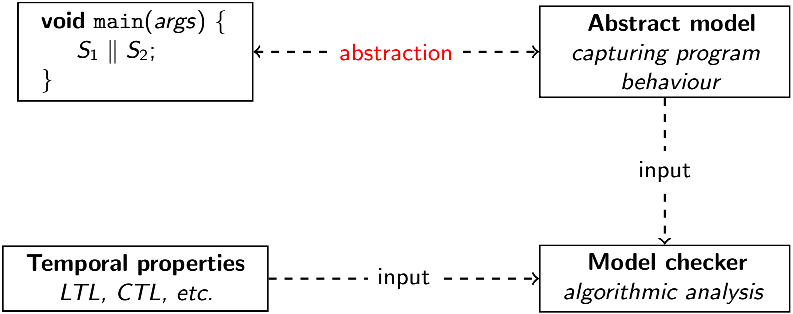
Good automation

Static verification: *model checking*



State-space explosion problems

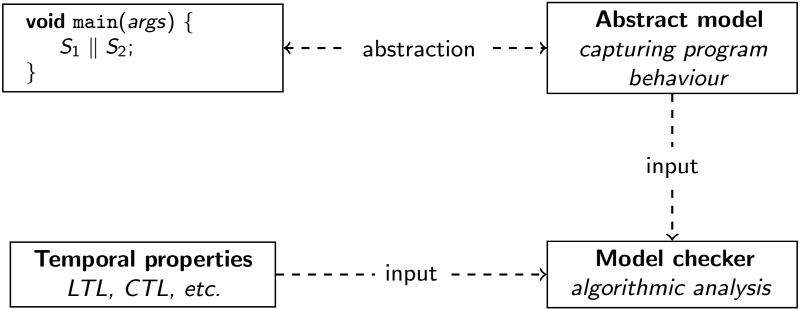
Static verification: *model checking*



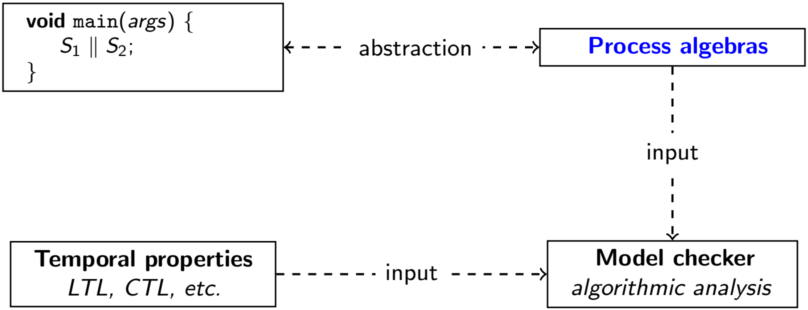
Is the program abstraction correct?

Outline

Our approach: *deducative + algorithmic verification*

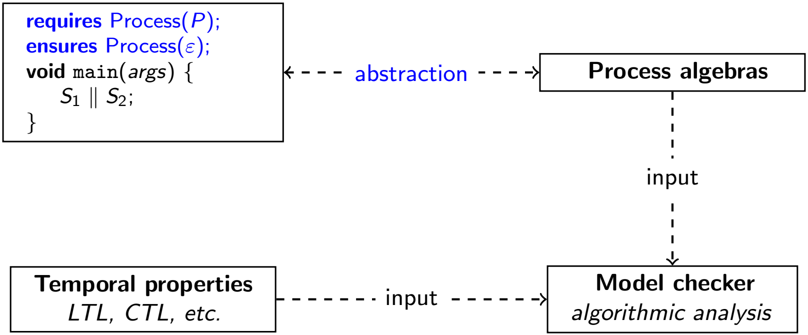


Our approach: *deductive + algorithmic verification*



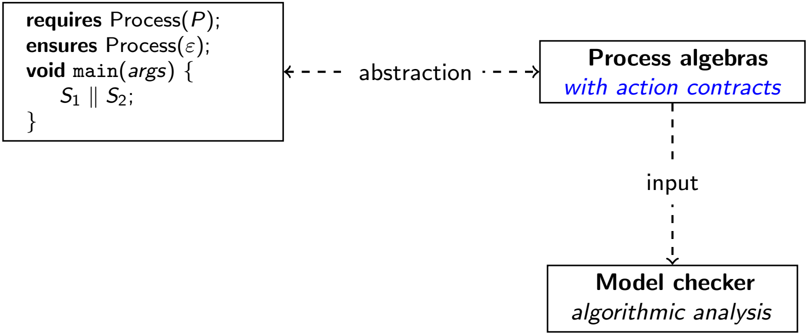
Abstract models are *process algebra terms*

Our approach: *deducative + algorithmic verification*



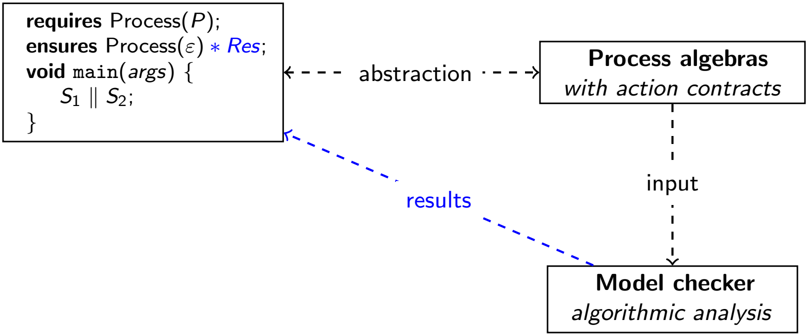
Deductively verifying *correctness* of the abstraction

Our approach: *deductive + algorithmic verification*



The properties are encoded as *process/action contracts*

Our approach: *deductive + algorithmic verification*



Applying model checking result *in the program logic*

Example program: *concurrent counting*

Concrete program

```
class Counter {
    public int val;

    static void add2(Counter c) {
       parallel {
          atomic-add(c.val, 1);
       } and {
          atomic-add(c.val, 1);
       }
    }
}
```

Program abstraction

Example program: *concurrent counting*

Concrete program | Program abstraction

```
class Counter {
    public int val;

   ensures c.val = old(c.val) + 2;
    static void add2(Counter c) {
       parallel {
          atomic-add(c.val, 1);
       } and {
          atomic-add(c.val, 1);
       }
    }
}
```

How to verify this property?

Example program: *concurrent counting*

Concrete program

```
class Counter {
    public int val;

    static void add2(Counter c) {
       parallel {
          atomic-add(c.val, 1);
       } and {
          atomic-add(c.val, 1);
       }
    }
}
```

Program abstraction

```
action incr;
```

The `incr` action *abstracts* the `atomic-add`'s.

Example program: *concurrent counting*

Concrete program

```
class Counter {
    public int val;

    static void add2(Counter c) {
        parallel {
            atomic-add(c.val, 1);
        } and {
            atomic-add(c.val, 1);
        }
    }
}
```

Program abstraction

```
action incr;

process parincr() := incr || incr;
```

The parincr process *abstracts* the add2 program.

Example program: *concurrent counting*

Concrete program

```
class Counter {
    public int val;

    static void add2(Counter c) {
        parallel {
            atomic-add(c.val, 1);
        } and {
            atomic-add(c.val, 1);
        }
    }
}
```

Program abstraction

```
guard true;
effect x = old(x) + 1;
action incr;

process parincr() := incr || incr;
```

Contracts are used to encode the *properties of interest*.

Example program: *concurrent counting*

Concrete program

```
class Counter {
    public int val;

    static void add2(Counter c) {
      parallel {
          atomic-add(c.val, 1);
       } and {
          atomic-add(c.val, 1);
       }
    }
}
```

Program abstraction

```
guard true;
effect x = old(x) + 1;
action incr;

requires true;
ensures x = old(x) + 2;
process parincr() := incr || incr;
```

Contracts are used to encode the *properties of interest*.

Example program: *concurrent counting*

Concrete program

```
class Counter {
    public int val;

   requires Process(parincr);
   requires c.val ~ x;
    static void add2(Counter c) {
       parallel {
           atomic-add(c.val, 1);
       } and {
           atomic-add(c.val, 1);
       }
    }
}
```

Program abstraction

```
guard true;
effect x = old(x) + 1;
action incr;

requires true;
ensures x = old(x) + 2;
process parincr() := incr || incr;
```

Linking concrete program to abstract model.

Example program: *concurrent counting*

Concrete program

```
class Counter {
    public int val;

   requires Process(parincr);
   requires c.val ~ x;
    static void add2(Counter c) {
      parallel {
        act incr do atomic-add(c.val, 1);
      } and {
        act incr do atomic-add(c.val, 1);
      }
    }
}
```

Program abstraction

```
guard true;
effect x = old(x) + 1;
action incr;

requires true;
ensures x = old(x) + 2;
process parincr() := incr || incr;
```

Linking abstract actions to concrete implementation.

Example program: *concurrent counting*

Concrete program

```
class Counter {
    public int val;

   requires Process(parincr);
   requires c.val ~ x;
   ensures Process(ε);
    static void add2(Counter c) {
       parallel {
          act incr do atomic-add(c.val, 1);
       } and {
          act incr do atomic-add(c.val, 1);
       }
    }
}
```

Program abstraction

```
guard true;
effect x = old(x) + 1;
action incr;

requires true;
ensures x = old(x) + 2;
process parincr() := incr || incr;
```

Linking concrete program to abstract model.

Example program: *concurrent counting*

Concrete program

```
class Counter {
    public int val;

   requires Process(parincr);
   requires c.val ~ x;
   ensures Process(ε);
   ensures c.val = old(c.val) + 2;
    static void add2(Counter c) {
       parallel {
          act incr do atomic-add(c.val, 1);
       } and {
          act incr do atomic-add(c.val, 1);
       }
    }
}
```

Program abstraction

```
guard true;
effect x = old(x) + 1;
action incr;

requires true;
ensures x = old(x) + 2;
process parincr() := incr || incr;
```

Using the result from analyzing all traces of parincr.

Model abstractions: *theory*

Formalised using the Coq proof assistant

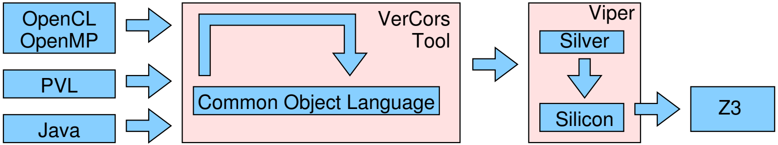
(see our VSTTE2017 paper for more details)

Model abstractions: *tool support*

As extension to the VerCors verification tool set

(see our iFM2017 paper+talk for more details)

The VerCors verification tool set



VerCors can be tried and downloaded via `http://utwente.nl/vercors`.

Model abstractions: *verified examples*

1. **Concurrent counting**: verifying that a concurrent counter computes the correct result.
2. **No-send-after-read**: verifying that confidential data can not be send after being received.
3. **Parallel GCD**: verifying that a parallel GCD algorithm computes the correct result *(VSTTE2017)*.
4. **Locking protocol**: verifying that a lock implementation adheres to the intended locking protocol.

Outline

Message Passing Interface (MPI)

MPI primitives

- `MPI_Send`(i, m) - sending a message m to process i;
- $m :=$ `MPI_Send`(i) - receiving a message m from process i;
- `MPI_Bcast`(m) - broadcasting a message m to every process, etc.

Message Passing Interface (MPI)

MPI primitives

- `MPI_Send`(i, m) - sending a message m to process i;
- $m :=$ `MPI_Send`(i) - receiving a message m from process i;
- `MPI_Bcast`(m) - broadcasting a message m to every process, etc.

Challenges

- Message exchanges are *concurrent*;
- Number of processes often *unknown*; and
- Number of possible behaviours often *unbounded*

Modeling MPI primitives

Abstracting MPI primitives

$\texttt{MPI\_Send}(i, m);$	$\rightsquigarrow$	$\textbf{action}\ \texttt{send}(i, m);$
$m := \texttt{MPI\_Send}(i);$	$\rightsquigarrow$	$\textbf{action}\ \texttt{recv}(i, m);$
$\texttt{MPI\_Bcast}(m);$	$\rightsquigarrow$	$\textbf{action}\ \texttt{bcast}(m);$
$\texttt{MPI\_Barrier}();$	$\rightsquigarrow$	$\textbf{action}\ \texttt{barrier};$

Modeling MPI primitives

Abstracting MPI primitives

$$\begin{array}{lcl}
\texttt{MPI_Send}(i, m); & \rightsquigarrow & \textbf{action } \texttt{send}(i, m); \\
m := \texttt{MPI_Send}(i); & \rightsquigarrow & \textbf{action } \texttt{recv}(i, m); \\
\texttt{MPI_Bcast}(m); & \rightsquigarrow & \textbf{action } \texttt{bcast}(m); \\
\texttt{MPI_Barrier}(); & \rightsquigarrow & \textbf{action } \texttt{barrier};
\end{array}$$

Introducing Hoare logic axioms

$$\frac{}{\vdash \{\texttt{send}(i, m) \cdot P\}\ \texttt{MPI\_Send(i, m)}\ \{P\}}$$

$$\frac{}{\vdash \{\texttt{recv}(i, m) \cdot P\}\ \texttt{m := MPI\_Recv(i)}\ \{P\}} \qquad \cdots$$

Linking abstractions to MPI programs

MPI program

```
void main(int k) {
    int v := MPI_Recv(*);
    MPI_Send(0, v + k);
}
```

Linking abstractions to MPI programs

MPI program

```
void main(int k) {
    int v := MPI_Recv(∗);
    MPI_Send(0, v + k);
}
```

Program abstraction

$$\textbf{process}\ \texttt{p}(\textbf{int}\ k) :=$$
$$\Sigma_{i:\textbf{int}} \cdot \textbf{recv}(*, i) \cdot \textbf{send}(0, i + k)$$

Linking abstractions to MPI programs

MPI program

requires Process($\mathrm{p}(k) \cdot \epsilon$);
void main(**int** k) {
    int v := MPI_Recv($*$);
    MPI_Send(0, $v + k$);
}

Program abstraction

$$\textbf{process}\ \mathrm{p}(\textbf{int}\ k) :=$$
$$\Sigma_{i:\textbf{int}} \cdot \textbf{recv}(*, i) \cdot \textbf{send}(0, i + k)$$

Linking abstractions to MPI programs

MPI program

$\textbf{requires}$ Process(p(k) $\cdot$ ϵ);
$\textbf{ensures}$ Process(ε);
void main(**int** k) {
    int v := MPI_Recv($*$);
    MPI_Send(0, $v + k$);
}

Program abstraction

$$\textbf{process}\ \mathrm{p}(\textbf{int}\ k) :=$$
$$\Sigma_{i:\textbf{int}} \cdot \textbf{recv}(*, i) \cdot \textbf{send}(0, i + k)$$

Linking abstractions to MPI programs

MPI program

```
requires Process(p(k) · ϵ);
ensures Process(ε);
void main(int k) {
    int v := MPI_Recv(∗);
    MPI_Send(0, v + k);
}
```

Program abstraction

$$\textbf{process}\ \mathrm{p}(\textbf{int}\ k) :=$$

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

Does the program correctly execute its abstraction?

Linking abstractions to MPI programs

MPI program

requires Process($\mathtt{p}(k) \cdot \epsilon$);
ensures Process(ε);
void main(**int** k) {
  $\{\mathtt{P}(k) \cdot \epsilon\}$
  int v := MPI_Recv($*$);
  MPI_Send(0, $v + k$);
}

Program abstraction

$$\textbf{process}\ \mathtt{p}(\textbf{int}\ k) :=$$

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

Does the program correctly execute its abstraction?

Linking abstractions to MPI programs

MPI program

$$\textbf{requires}\ \text{Process}(\texttt{p}(k) \cdot \epsilon);$$
$$\textbf{ensures}\ \text{Process}(\varepsilon);$$

```
void main(int k) {
    {P(k) · ϵ}
    {Σ_{i:int} · recv(∗, i) · send(0, i + k) · ϵ}
    int v := MPI_Recv(∗);
    MPI_Send(0, v + k);
}
```

Program abstraction

$$\textbf{process}\ \texttt{p}(\textbf{int}\ k) :=$$
$$\Sigma_{i:\textbf{int}} \cdot \textbf{recv}(*, i) \cdot \textbf{send}(0, i + k)$$

Does the program correctly execute its abstraction?

Linking abstractions to MPI programs

MPI program

$\textbf{requires}\ \text{Process}(\text{p}(k) \cdot \epsilon);$
$\textbf{ensures}\ \text{Process}(\varepsilon);$
$\textbf{void}\ \texttt{main}(\textbf{int}\ \text{k})\ \{$
$\quad \{\text{P}(k) \cdot \epsilon\}$
$\quad \{\Sigma_{i:\textbf{int}} \cdot \textbf{recv}(*, i) \cdot \textbf{send}(0, i + k) \cdot \epsilon\}$
$\quad \textbf{int}\ v := \texttt{MPI\_Recv}(*);$
$\quad \{\textbf{send}(0, v + k) \cdot \epsilon\}$
$\quad \texttt{MPI\_Send}(0,\ v + k);$
$\}$

Program abstraction

$$\textbf{process}\ \text{p}(\textbf{int}\ k) :=$$

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

Does the program correctly execute its abstraction?

Linking abstractions to MPI programs

MPI program

requires Process($\mathrm{p}(k) \cdot \epsilon$);
ensures Process(ε);
void main(**int** k) {
  $\{\mathrm{P}(k) \cdot \epsilon\}$
  $\{\Sigma_{i:\mathbf{int}} \cdot \mathbf{recv}(*, i) \cdot \mathbf{send}(0, i+k) \cdot \epsilon\}$
  int v := MPI_Recv($*$);
  $\{\mathbf{send}(0, v+k) \cdot \epsilon\}$
  MPI_Send(0, $v + k$);
  $\{\varepsilon\}$
}

Program abstraction

$$\mathbf{process}\ \mathrm{p}(\mathbf{int}\ k) :=$$

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

Does the program correctly execute its abstraction?

Linking abstractions to MPI programs

MPI program

requires Process($\mathrm{p}(k) \cdot \epsilon$);
ensures Process(ε);
void `main`(**int** `k`) {
  $\{\mathrm{P}(k) \cdot \epsilon\}$
  $\{\Sigma_{i:\mathbf{int}} \cdot \mathbf{recv}(*, i) \cdot \mathbf{send}(0, i+k) \cdot \epsilon\}$
  int v := `MPI_Recv`($*$);
  $\{\mathbf{send}(0, v+k) \cdot \epsilon\}$
  `MPI_Send`($0,\ v+k$);
  $\{\varepsilon\}$
}

Program abstraction

$$\mathbf{process}\ \mathrm{p}(\mathbf{int}\ k) :=$$

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

Does the program correctly execute its abstraction? **Yes!**

Toolchain overview

```
requires Process(p(k) · P);
ensures Process(P);
void main(int k) {
    int v := MPI_Recv(∗);
    MPI_Send(0, v + k);
}
```

Toolchain overview

```
requires Process(p(k) · P);
ensures Process(P);
void main(int k) {
      int v := MPI_Recv(*);
      MPI_Send(0, v + k);
}
```

→ 1 →

Toolchain overview

process $\mathtt{p}$(**int** k) :=

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

2

requires Process($\mathtt{p}(k) \cdot P$);
ensures Process(P);
void `main`(**int** `k`) {
    int v := `MPI_Recv`($*$);
    `MPI_Send`(0, $v + k$);
}

1

Toolchain overview

process p(**int** k) :=

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

2

requires Process(p(k) $\cdot$ P);
ensures Process(P);

```
void main(int k) {
    int v := MPI_Recv(*);
    MPI_Send(0, v + k);
}
```

1

3

Verdict
Does the program correctly executes its abstraction?
Hoare logic reasoning

Toolchain overview

process p(**int** k) :=

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

2

requires Process(p(k) · P);
ensures Process(P);
void main(**int** k) {
    int v := MPI_Recv(∗);
    MPI_Send(0, $v + k$);
}

1

3

Verdict
Does the program correctly executes its abstraction?
Hoare logic reasoning

P(k_1) ∥ P(k_2) ∥ · · · ∥ MPI

Toolchain overview

process p(**int** k) :=

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

2

requires Process(p(k) $\cdot$ P);
ensures Process(P);

```
void main(int k) {
    int v := MPI_Recv(*);
    MPI_Send(0, v + k);
}
```

1

3

Verdict
Does the program correctly executes its abstraction?
Hoare logic reasoning

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

4

mCRL2 analysing system behaviour

Toolchain overview

process p(**int** k) :=

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

2

```
requires Process(p(k) · P);
ensures Process(P);
void main(int k) {
    int v := MPI_Recv(*);
    MPI_Send(0, v + k);
}
```

1

3

Verdict
Does the program correctly executes its abstraction?
Hoare logic reasoning

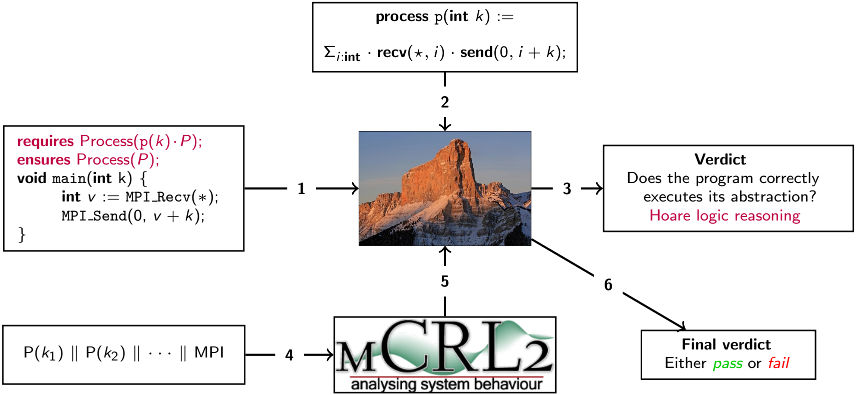
5

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

4

mCRL2 analysing system behaviour

Toolchain overview



Outline

Conclusion

VerCors verification tool set

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