The VerCors Tool Set

Verification of Parallel and Concurrent Software

Stefan Blom Saeed Darabi
Marieke Huisman Wytse Oortwijn

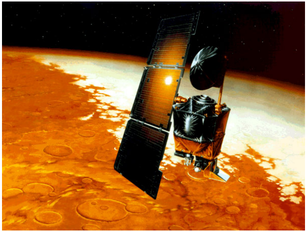
Formal Methods and Tools, University of Twente

September 20, 2017

Outline

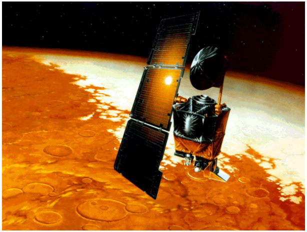
Outline

Concurrent programming is *error-prone*



Mars climate orbiter

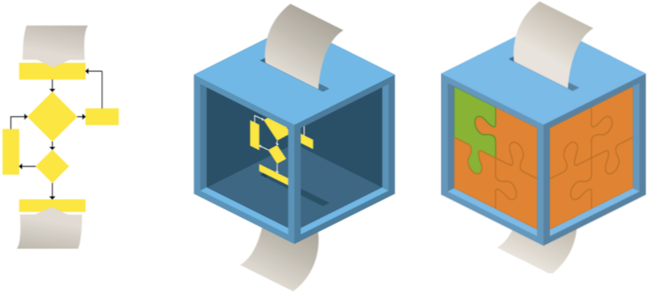
Concurrent programming is *error-prone*



Mars climate orbiter

THERAC-25

Automated program verification



The VerCors Tool Set

Verifying Java programs: *example*

```
class Counter {
    int count;

    public void incr(int y) {
        this.count := this.count + y;
    }
}
```

Verifying Java programs: *example*

```
class Counter {
    int count;

    requires Perm(this.count, write);
    public void incr(int y) {
      this.count := this.count + y;
    }
}
```

Writing permission for *this.count* is required

Verifying Java programs: *example*

```
class Counter {
    int count;

    requires Perm(this.count, write);
    ensures Perm(this.count, write);
    public void incr(int y) {
        this.count := this.count + y;
    }
}
```

Writing permission for *this.count* is ensured

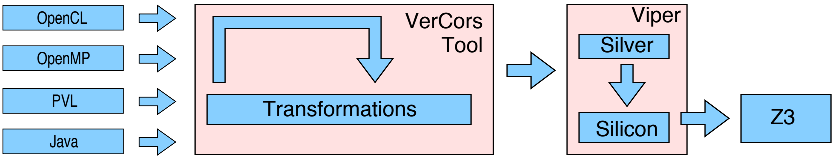
Verifying Java programs: *example*

```
class Counter {
    int count;

    requires Perm(this.count, write);
    ensures Perm(this.count, write);
    ensures this.count = old(this.count) + y;
    public void incr(int y) {
        this.count := this.count + y;
    }
}
```

this.count is conrrectly incremented

The VerCors Tool Set - *Overview*



Outline

Deterministic parallel programming

Concurrency as optimisation

Sequential program annotated by *compiler directives*	*compiled into*	Multi-threaded programs or GPU kernels

Deterministic parallel programming

Concurrency as optimisation

Sequential program annotated by *compiler directives*	*compiled into*	Multi-threaded programs or GPU kernels

Compiler directives

Give hints to the compiler to know *where* and *how* to parallelise sequential code.

Deterministic parallel programming

Concurrency as optimisation

Sequential program annotated by *compiler directives*	*compiled into*	Multi-threaded programs or GPU kernels

Compiler directives

Give hints to the compiler to know *where* and *how* to parallelise sequential code. Some implementations are:

- OpenACC
- OpenMP
- PENCIL, etc.

Deterministic parallel programming

Concurrency as optimisation

Sequential program annotated by *compiler directives*	*compiled into*	Multi-threaded programs or GPU kernels

Compiler directives

Give hints to the compiler to know *where* and *how* to parallelise sequential code. Some implementations are:

- OpenACC
- OpenMP *(for C)*
- PENCIL, etc.

Verifying OpenMP programs - *NFM 2017*

for (**int** $i = 0$; $i < N$; i++) {
  $C[i] = A[i]$;
}

for (**int** $i = 0$; $i < N$; i++) {
  $D[i] = C[i+1] + B[i]$;
}

Verifying OpenMP programs - *NFM 2017*

```
for (int i = 0; i < N; i++) {
    C[i] = A[i];
}

for (int i = 0; i < N; i++) {
    D[i] = C[i + 1] + B[i];
}
```

```
for (int i = 0; i < N; i++) {
    C[i] = A[i];
}

for (int i = 0; i < N; i++) {
    D[i] = C[i] + B[i];
}
```

Verifying OpenMP programs - *NFM 2017*

```
#pragma omp parallel {
    #pragma omp for
    for (int i = 0; i < N; i++) {
        C[i] = A[i];
    }

    #pragma omp for
    for (int i = 0; i < N; i++) {
        D[i] = C[i + 1] + B[i];
    }
}
```

```
for (int i = 0; i < N; i++) {
    C[i] = A[i];
}

for (int i = 0; i < N; i++) {
    D[i] = C[i] + B[i];
}
```

Verifying OpenMP programs - *NFM 2017*

```
#pragma omp parallel {
    #pragma omp for
    for (int i = 0; i < N; i++) {
        C[i] = A[i];
    }

    #pragma omp for
    for (int i = 0; i < N; i++) {
        D[i] = C[i + 1] + B[i];
    }
}
```

```
#pragma omp parallel {
    #pragma omp for
        schedule(static) nowait
    for (int i = 0; i < N; i++) {
        C[i] = A[i];
    }

    #pragma omp for
        schedule(static) nowait
    for (int i = 0; i < N; i++) {
        D[i] = C[i] + B[i];
    }
}
```

Verifying OpenMP programs - *NFM 2017*

```
#pragma omp parallel {
    #pragma omp for
    for (int i = 0; i < N; i++) {
        C[i] = A[i];
    }

    #pragma omp for
    for (int i = 0; i < N; i++) {
        D[i] = C[i + 1] + B[i];
    }
}
```

```
#pragma omp parallel {
    #pragma omp for
        schedule(static) nowait
    for (int i = 0; i < N; i++) {
        C[i] = A[i];
    }

    #pragma omp for
        schedule(static) nowait
    for (int i = 0; i < N; i++) {
        D[i] = C[i] + B[i];
    }
}
```

Verifying loop parallelisations - *FASE 2015*

Specifying a contract for every loop iteration, capturing its resources

Verifying loop parallelisations - *FASE 2015*

Specifying a contract for every loop iteration, capturing its resources

$$\frac{}{\vdash\ \textbf{for}\,(\textbf{int}\ i = 0; i < N; i\,{+}{+})\{\ S_i\ \}}$$

Verifying loop parallelisations - *FASE 2015*

Specifying a contract for every loop iteration, capturing its resources

$$\frac{\vdash \{P_0\}\ S_0\ \{Q_0\} \qquad \cdots \qquad \vdash \{P_{N-1}\}\ S_{N-1}\ \{Q_{N-1}\}}{\vdash\ \textbf{for}\,(\textbf{int}\ i = 0; i < N; i{+}{+})\{\ S_i\ \}}$$

Verifying loop parallelisations - *FASE 2015*

Specifying a contract for every loop iteration, capturing its resources

$$\frac{\vdash \{P_0\}\ S_0\ \{Q_0\} \qquad \cdots \qquad \vdash \{P_{N-1}\}\ S_{N-1}\ \{Q_{N-1}\}}{\vdash \{P_0 * \cdots * P_{N-1}\}\ \textbf{for}\,(\textbf{int}\ i = 0; i < N; i{+}{+})\{\ S_i\ \}\ \{Q_0 * \cdots * Q_{N-1}\}}$$

Deterministic parallelism: *iteration contracts*

```
void clear(int A[], int len) {
    for (int i = 0; i < len; i++) {
        A[i] = 0;
    }
}
```

Deterministic parallelism: *iteration contracts*

$\textbf{requires}\ A \neq \texttt{null} \wedge \textbf{length}(A) = len;$

```
void clear(int A[], int len) {
    for (int i = 0; i < len; i++) {
        A[i] = 0;
    }
}
```

The input array A may not be `null`

Deterministic parallelism: *iteration contracts*

```
requires A ≠ null ∧ length(A) = len;
requires (forall* int k; 0 ≤ k < len; Perm(A[k], write));
void clear(int A[], int len) {
    for (int i = 0; i < len; i++) {
        A[i] = 0;
    }
}
```

Permission is required to write to every element of A

Deterministic parallelism: *iteration contracts*

$$
\begin{array}{l}
\textbf{requires } A \neq \texttt{null} \wedge \textbf{length}(A) = len; \\
\textbf{requires } (\textbf{forall}{*}\ \textbf{int } k;\ 0 \leq k < len;\ \text{Perm}(A[k], \text{write})); \\
\textbf{ensures } (\textbf{forall}{*}\ \textbf{int } k;\ 0 \leq k < len;\ \text{Perm}(A[k], \text{write})); \\
\textbf{ensures } (\textbf{forall int } k;\ 0 \leq k < len;\ A[k] = 0); \\
\textbf{void } \texttt{clear}(\textbf{int } A[\,], \textbf{int } len)\ \{ \\
\quad \textbf{for } (\textbf{int } i = 0;\ i < len;\ i{+}{+})\ \{ \\
\quad\quad A[i] = 0; \\
\quad \} \\
\}
\end{array}
$$

A "cleared" array is ensured

Deterministic parallelism: *iteration contracts*

$\textbf{requires}\ A \neq \texttt{null} \wedge \textbf{length}(A) = \mathit{len};$
$\textbf{requires}\ (\textbf{forall}{*}\ \textbf{int}\ k;\ 0 \leq k < \mathit{len};\ \text{Perm}(A[k], \text{write}));$
$\textbf{ensures}\ (\textbf{forall}{*}\ \textbf{int}\ k;\ 0 \leq k < \mathit{len};\ \text{Perm}(A[k], \text{write}));$
$\textbf{ensures}\ (\textbf{forall}\ \textbf{int}\ k;\ 0 \leq k < \mathit{len};\ A[k] = 0);$
$\textbf{void}\ \texttt{clear}(\textbf{int}\ A[\,],\ \textbf{int}\ \mathit{len})\ \{$
$\quad \textbf{for}\ (\textbf{int}\ i = 0;\ i < \mathit{len};\ i{+}{+})$
$\qquad \textbf{requires}\ \text{Perm}(A[i], \text{write})\quad \{$
$\qquad A[i] = 0;$
$\quad \}$
$\}$

Iteration i requires permission to write to $A[i]$

Deterministic parallelism: *iteration contracts*

$$
\begin{array}{l}
\textbf{requires } A \neq \texttt{null} \wedge \textbf{length}(A) = len; \\
\textbf{requires } (\textbf{forall}{*}\ \textbf{int } k;\ 0 \leq k < len;\ \mathrm{Perm}(A[k], \mathrm{write})); \\
\textbf{ensures } (\textbf{forall}{*}\ \textbf{int } k;\ 0 \leq k < len;\ \mathrm{Perm}(A[k], \mathrm{write})); \\
\textbf{ensures } (\textbf{forall int } k;\ 0 \leq k < len;\ A[k] = 0); \\
\textbf{void } \texttt{clear}(\textbf{int } A[\,],\ \textbf{int } len)\ \{ \\
\quad \textbf{for } (\textbf{int } i = 0;\ i < len;\ i{+}{+}) \\
\qquad \textbf{requires } \mathrm{Perm}(A[i], \mathrm{write}); \\
\qquad \textbf{ensures } \mathrm{Perm}(A[i], \mathrm{write})\ \ \{ \\
\qquad A[i] = 0; \\
\quad \} \\
\}
\end{array}
$$

Iteration i ensures permission to write to $A[i]$

Deterministic parallelism: *iteration contracts*

$$
\begin{array}{l}
\textbf{requires } A \neq \texttt{null} \wedge \textbf{length}(A) = len;\\
\textbf{requires } (\textbf{forall}{*}\ \textbf{int } k;\ 0 \leq k < len;\ \mathrm{Perm}(A[k], \mathrm{write}));\\
\textbf{ensures } (\textbf{forall}{*}\ \textbf{int } k;\ 0 \leq k < len;\ \mathrm{Perm}(A[k], \mathrm{write}));\\
\textbf{ensures } (\textbf{forall int } k;\ 0 \leq k < len;\ A[k] = 0);\\
\textbf{void } \texttt{clear}(\textbf{int } A[\,],\ \textbf{int } len)\ \{\\
\quad \textbf{for } (\textbf{int } i = 0;\ i < len;\ i{+}{+})\\
\quad\quad \textbf{requires } \mathrm{Perm}(A[i], \mathrm{write});\\
\quad\quad \textbf{ensures } \mathrm{Perm}(A[i], \mathrm{write});\\
\quad\quad \textbf{ensures } A[i] = 0\ \{\\
\quad\quad A[i] = 0;\\
\quad \}\\
\}
\end{array}
$$

Iteration i “cleared” $A[i]$

Deterministic parallelism: *verified examples*

1. **Histogram**: calculating the *histogram* of a matrix.
2. **Loop dependencies**: several examples with forward/backward *loop dependencies*.
3. **Reductions**: several examples with *reduction patterns* (e.g. summations).
4. **Loop vectorisation**: several examples with *loop vectorisations*.

Deterministic parallelism: *verified examples*

1. **Histogram**: calculating the *histogram* of a matrix.
2. **Loop dependencies**: several examples with forward/backward *loop dependencies*.
3. **Reductions**: several examples with *reduction patterns* (e.g. summations).
4. **Loop vectorisation**: several examples with *loop vectorisations*.

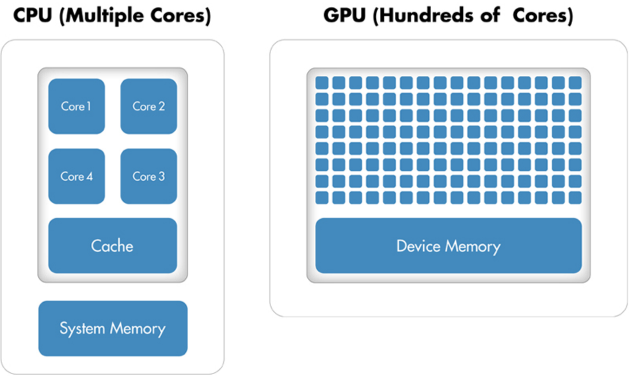
Deterministic parallelism: *verified examples*

1. **Histogram**: calculating the *histogram* of a matrix.
2. **Loop dependencies**: several examples with forward/backward *loop dependencies*.
3. **Reductions**: several examples with *reduction patterns* (e.g. summations).
4. **Loop vectorisation**: several examples with *loop vectorisations*.

See our *FASE2015* and *NFM2017* papers for more details

Outline

GPU computing model



source https://www.mathworks.com

Single-Instruction-Multiple-Data *(SIMD)*

```
__kernel void add1(__global int A[]) {
    // get thread identifier
    int tid = get_global_id(0);

    // increase entry by one
    A[tid] = A[tid] + 1;
}
```

Single-Instruction-Multiple-Data *(SIMD)*

```
__kernel void add1(__global int A[]) {
  // get thread identifier
  int tid = get_global_id(0);

  // increase entry by one
  A[tid] = A[tid] + 1;
}
```


```
__kernel void add1(__global int A[]) {
  // get thread identifier
  int tid = get_global_id(0);

  // increase entry by one
  A[tid] = A[tid] + 1;
}
```

```
__kernel void add1(__global int A[]) {
  // get thread identifier
  int tid = get_global_id(0);

  // increase entry by one
  A[tid] = A[tid] + 1;
}
```

```
__kernel void add1(__global int A[]) {
  // get thread identifier
  int tid = get_global_id(0);

  // increase entry by one
  A[tid] = A[tid] + 1;
}
```

```
__kernel void add1(__global int A[]) {
  // get thread identifier
  int tid = get_global_id(0);

  // increase entry by one
  A[tid] = A[tid] + 1;
}
```

```
__kernel void add1(__global int A[]) {
  // get thread identifier
  int tid = get_global_id(0);

  // increase entry by one
  A[tid] = A[tid] + 1;
}
```

```
__kernel void add1(__global int A[]) {
  // get thread identifier
  int tid = get_global_id(0);

  // increase entry by one
  A[tid] = A[tid] + 1;
}
```

```
__kernel void add1(__global int A[]) {
  // get thread identifier
  int tid = get_global_id(0);

  // increase entry by one
  A[tid] = A[tid] + 1;
}
```

```
__kernel void add1(__global int A[]) {
  // get thread identifier
  int tid = get_global_id(0);

  // increase entry by one
  A[tid] = A[tid] + 1;
}
```

```
__kernel void add1(__global int A[]) {
  // get thread identifier
  int tid = get_global_id(0);

  // increase entry by one
  A[tid] = A[tid] + 1;
}
```

```
__kernel void add1(__global int A[]) {
  // get thread identifier
  int tid = get_global_id(0);

  // increase entry by one
  A[tid] = A[tid] + 1;
}
```

```
__kernel void add1(__global int A[]) {
  // get thread identifier
  int tid = get_global_id(0);

  // increase entry by one
  A[tid] = A[tid] + 1;
}
```

```
__kernel void add1(__global int A[]) {
  // get thread identifier
  int tid = get_global_id(0);

  // increase entry by one
  A[tid] = A[tid] + 1;
}
```

```
__kernel void add1(__global int A[]) {
  // get thread identifier
  int tid = get_global_id(0);

  // increase entry by one
  A[tid] = A[tid] + 1;
}
```

```
__kernel void add1(__global int A[]) {
  // get thread identifier
  int tid = get_global_id(0);

  // increase entry by one
  A[tid] = A[tid] + 1;
}
```

```
__kernel void add1(__global int A[]) {
  // get thread identifier
  int tid = get_global_id(0);

  // increase entry by one
  A[tid] = A[tid] + 1;
}
```

```
__kernel void add1(__global int A[]) {
  // get thread identifier
  int tid = get_global_id(0);

  // increase entry by one
  A[tid] = A[tid] + 1;
}
```

```
__kernel void add1(__global int A[]) {
  // get thread identifier
  int tid = get_global_id(0);

  // increase entry by one
  A[tid] = A[tid] + 1;
}
```

```
__kernel void add1(__global int A[]) {
  // get thread identifier
  int tid = get_global_id(0);

  // increase entry by one
  A[tid] = A[tid] + 1;
}
```

```
__kernel void add1(__global int A[]) {
  // get thread identifier
  int tid = get_global_id(0);

  // increase entry by one
  A[tid] = A[tid] + 1;
}
```

Single-Instruction-Multiple-Data *(SIMD)*

$tid = 0$	$tid = 1$	$tid = 2$	$tid = 3$	$tid = 4$
$tid = 5$	$tid = 6$	$tid = 7$	$tid = 8$	$tid = 9$
$tid = 10$	$tid = 11$	$tid = 12$	$tid = 13$	$tid = 14$
$tid = 15$	$tid = 16$	$tid = 17$	$tid = 18$	$tid = 19$

Single-Instruction-Multiple-Data *(SIMD)*

$A[0] = A[0] + 1$	$A[1] = A[1] + 1$	$A[2] = A[2] + 1$	$A[3] = A[3] + 1$	$A[4] = A[4] + 1$
$A[5] = A[5] + 1$	$A[6] = A[6] + 1$	$A[7] = A[7] + 1$	$A[8] = A[8] + 1$	$A[9] = A[9] + 1$
$A[10] = A[10] + 1$	$A[11] = A[11] + 1$	$A[12] = A[12] + 1$	$A[13] = A[13] + 1$	$A[14] = A[14] + 1$
$A[15] = A[15] + 1$	$A[16] = A[16] + 1$	$A[17] = A[17] + 1$	$A[18] = A[18] + 1$	$A[19] = A[19] + 1$

Thread specifications

```
__kernel void add1(__global int A[]) {
    // get thread identifier
    int tid = get_global_id(0);

    // increase entry by one
    A[tid] = A[tid] + 1;
}
```

Thread specifications

```
requires A ≠ null;
__kernel void add1(__global int A[]) {
    // get thread identifier
    int tid = get_global_id(0);

    // increase entry by one
    A[tid] = A[tid] + 1;
}
```

The input array may not be `null`

Thread specifications

```
requires A ≠ null;
requires Perm(A[get_global_id(0)], write);
__kernel void add1(__global int A[]) {
    // get thread identifier
    int tid = get_global_id(0);

    // increase entry by one
    A[tid] = A[tid] + 1;
}
```

Thread with ID i requires *write permission* for $A[i]$

Thread specifications

```
requires A ≠ null;
requires Perm(A[get_global_id(0)], write);
ensures Perm(A[get_global_id(0)], write);
__kernel void add1(__global int A[]) {
    // get thread identifier
    int tid = get_global_id(0);

    // increase entry by one
    A[tid] = A[tid] + 1;
}
```

Thread with ID *i* ensures *write permission* for *A*[*i*]

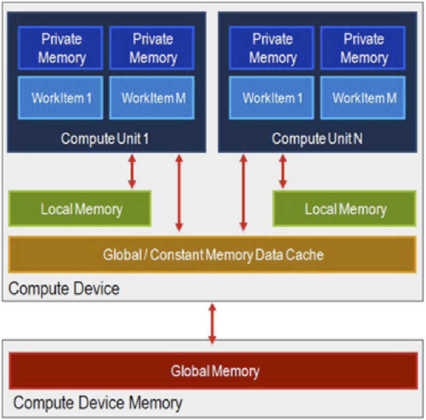
Thread specifications

requires $A \neq$ `null`;
requires Perm(A[`get_global_id(0)`], write);
ensures Perm(A[`get_global_id(0)`], write);
ensures A[`get_global_id(0)`] $=$ **old**(A[`get_global_id(0)`]) $+ 1$;
__**kernel** **void** `add1`(__**global** **int** A[ ]) {
    // get thread identifier
    int *tid* = `get_global_id(0)`;

    // increase entry by one
    $A[tid] = A[tid] + 1$;
}

Thread with ID i has incremented $A[i]$

GPU architecture: *threads & workgroups*



Barrier specifications

```
__kernel void addcomplex(__global int A[]) {
    // get thread identifier
    int tid = get_global_id(0);

    // perform the updates
    A[tid] = A[tid] + 1;
    if (tid > 0) {
        A[tid − 1] = 2 ∗ A[tid − 1];
    }
}
```

Barrier specifications

```
__kernel void addcomplex(__global int A[]) {
    // get thread identifier
    int tid = get_global_id(0);

    // perform the updates
    A[tid] = A[tid] + 1;

    barrier(global);

    if (tid > 0) {
        A[tid - 1] = 2 * A[tid - 1];
    }
}
```

A barrier is required here for correctness!

Barrier specifications

```
requires Perm(A[get_global_id(0)], write);
__kernel void addcomplex(__global int A[]) {
    // get thread identifier
    int tid = get_global_id(0);

    // perform the updates
    A[tid] = A[tid] + 1;

    barrier(global);

    if (tid > 0) {
        A[tid − 1] = 2 ∗ A[tid − 1];
    }
}
```

Thread with ID i can *write* to $A[i]$ and *read* from $B[i]$

Barrier specifications

```
requires Perm(A[get_global_id(0)], write);
__kernel void addcomplex(__global int A[]) {
    // get thread identifier
    int tid = get_global_id(0);

    // perform the updates
    A[tid] = A[tid] + 1;

    barrier(global) {
        requires Perm(A[tid], write);
        ensures tid > 0 ⇒ Perm(A[tid − 1], write);
    };

    if (tid > 0) {
        A[tid − 1] = 2 ∗ A[tid − 1];
    }
}
```

The barrier *consumes* all read permissions

Barrier specifications

```
requires Perm(A[get_global_id(0)], write);
__kernel void addcomplex(__global int A[]) {
  // get thread identifier
  int tid = get_global_id(0);

  // perform the updates
  A[tid] = A[tid] + 1;

  barrier(global) {
    requires Perm(A[tid], write);
    ensures tid > 0 ⇒ Perm(A[tid − 1], write);
  };

  if (tid > 0) {
    A[tid − 1] = 2 ∗ A[tid − 1];
  }
}
```

The barrier *redistributes* all consumed permissions

GPU kernels: *verified examples*

1. **Parallel prefix sum**: data-race freedom of a *paralell prefix sum* algorithm.
2. **Summations**: kernels with *atomic*, *barriers*, and *reductions*.
3. **Floats**: support for *floating point* arithmetics.

GPU kernels: *verified examples*

1. **Parallel prefix sum**: data-race freedom of a *paralell prefix sum* algorithm.
2. **Summations**: kernels with *atomic*, *barriers*, and *reductions*.
3. **Floats**: support for *floating point* arithmetics.

Outline

Static verification: *deductive verification*

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

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

Static verification: *deductive verification*

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

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

Strong correctness guarantees

Static verification: *deductive verification*

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

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

Hard to construct a proof

Static verification: *model checking*

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

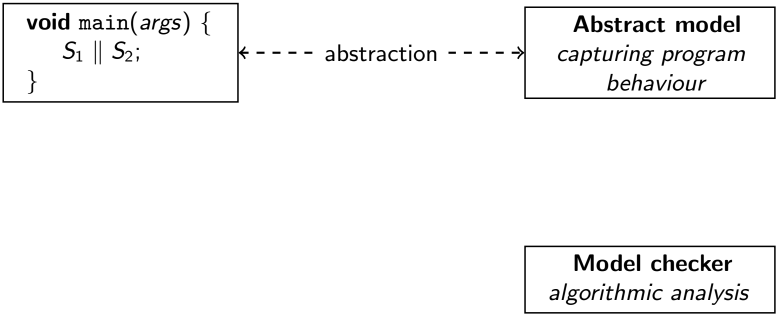
Static verification: *model checking*

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

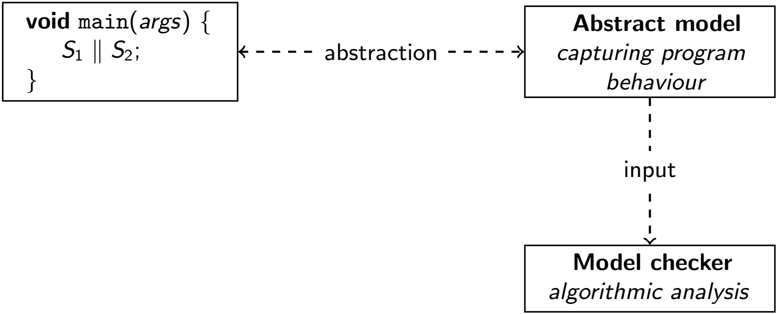
Abstract model
capturing program behaviour

Static verification: *model checking*

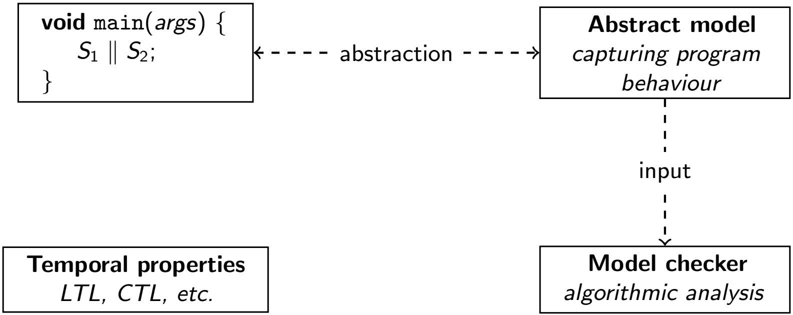
Static verification: *model checking*



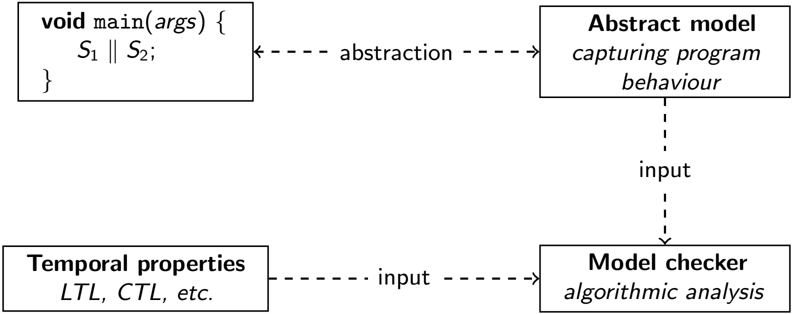
Static verification: *model checking*



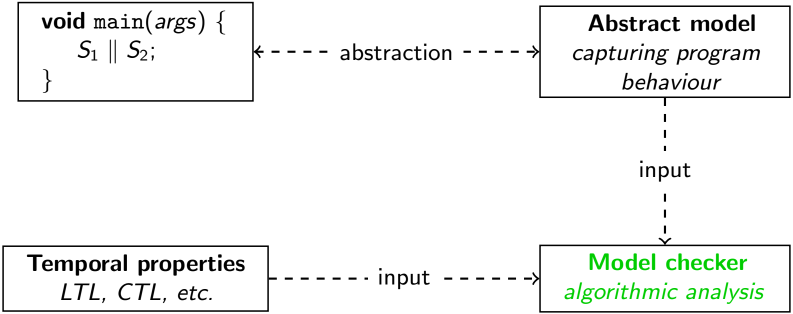
Static verification: *model checking*



Static verification: *model checking*

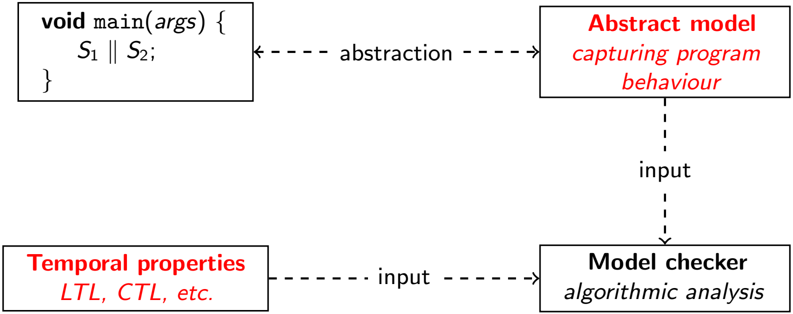


Static verification: *model checking*



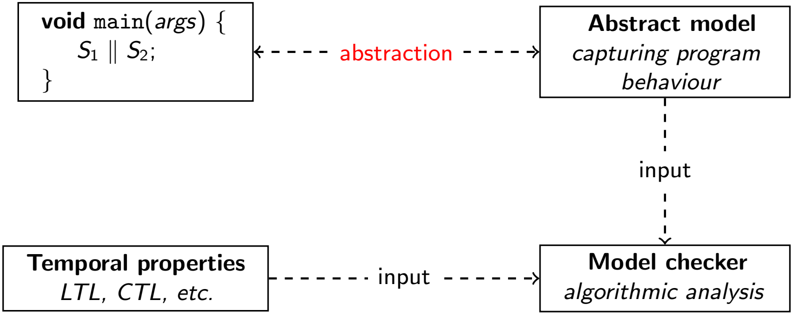
Good automation

Static verification: *model checking*



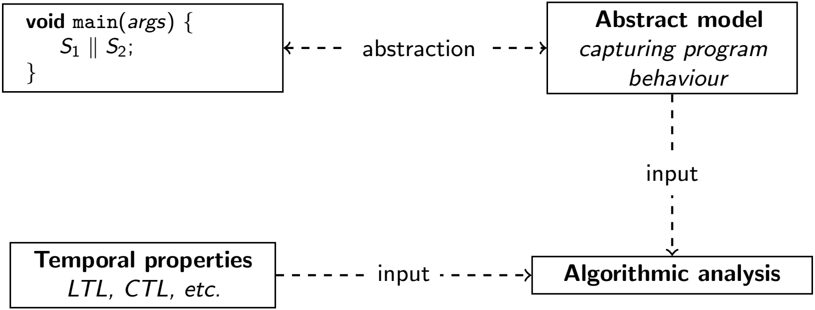
State-space explosion problems

Static verification: *model checking*

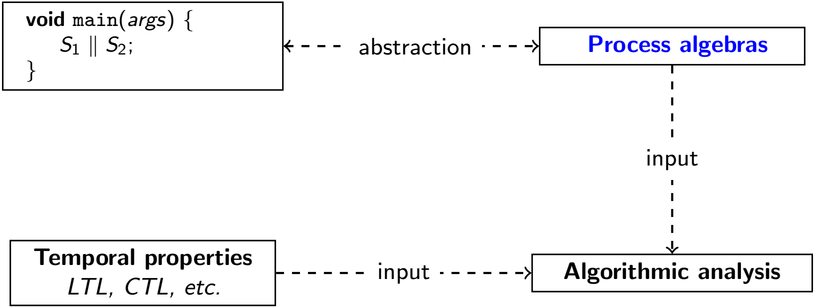


Is the program abstraction correct?

Our approach: *deductive + algorithmic verification*

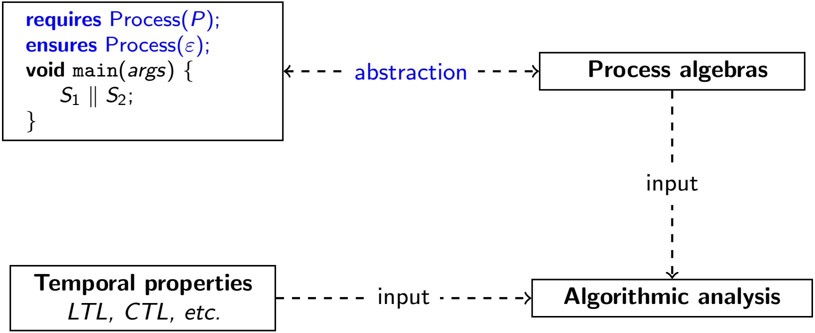


Our approach: *deductive + algorithmic verification*



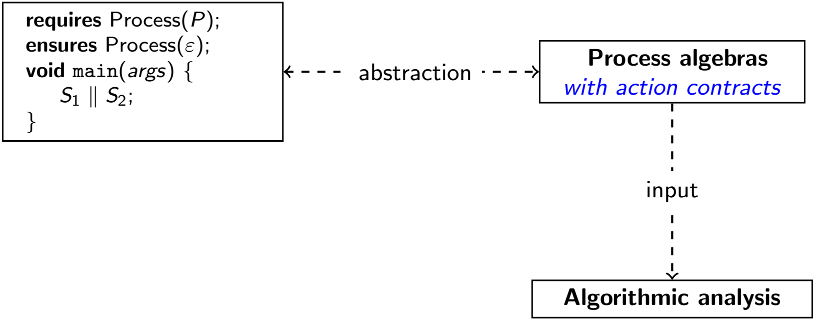
Abstract models are *process algebra terms*

Our approach: *deductive + algorithmic verification*



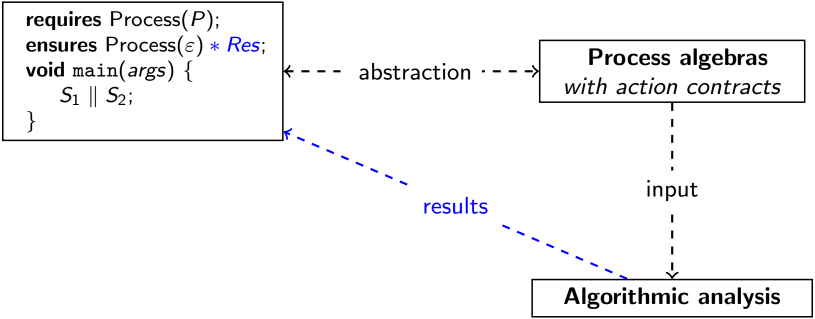
Deductively verifying *correctness* of the abstraction

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

```
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);
      }
    }
}
```

Program abstraction

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	Program abstraction

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 $\mathtt{x} = \mathbf{old}(\mathtt{x}) + 1$;
action incr;

requires true;
ensures $\mathtt{x} = \mathbf{old}(\mathtt{x}) + 2$;
process $\mathtt{parincr}() := \mathtt{incr} \parallel \mathtt{incr}$;

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	Program abstraction
class Counter {	
public int *val*;	
requires Process(`parincr`);	**guard** true;
requires *c*.*val* ∼ `x`;	**effect** `x` = **old**(`x`) + 1;
static void add2(Counter *c*) {	**action** `incr`;
parallel {	
act `incr` **do** `atomic-add`(*c*.*val*, 1);	**requires** true;
} **and** {	**ensures** `x` = **old**(`x`) + 2;
act `incr` **do** `atomic-add`(*c*.*val*, 1);	**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: *underlying theory*

Formalised using the Coq proof assistant

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.
4. **Locking protocol**: verifying that a lock implementation adheres to the intended locking protocol.

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.
4. **Locking protocol**: verifying that a lock implementation adheres to the intended locking protocol.

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.
4. **Locking protocol**: verifying that a lock implementation adheres to the intended locking protocol.

See our *VSTTE2017* paper for more details

Outline

Conclusion

Future directions

1. Verifying *distributed software*.
2. Automatic generation of *annotations*.
3. Generating *meaningful* error messages.

Conclusion

Future directions

1. Verifying *distributed software*.
2. Automatic generation of *annotations*.
3. Generating *meaningful* error messages.

VerCors Verification Tool Set

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