Getting it Right for HPC:

An Introduction to Testing and Model Checking

Model Checking I

Stephen F. Siegel

Department of Computer and Information Sciences
University of Delaware

December 18, 2023

Motivation: The limitations of testing

1. **lack of coverage** of input space
   - only a tiny fraction of inputs can be tested
   - no sure way to select the tests guaranteed to find all bugs

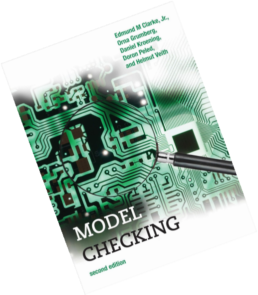
Motivation: The limitations of testing

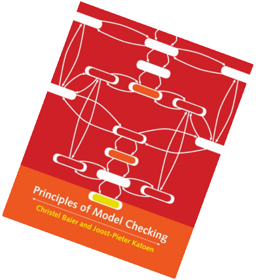
1. **lack of coverage** of input space
   - only a tiny fraction of inputs can be tested
   - no sure way to select the tests guaranteed to find all bugs
2. **nondeterminism**
   - a program can behave differently on two executions with the same input
     - concurrency: interleaving of statements from different threads/processes may impact observable behavior
     - compilers, MPI implementation, OpenMP runtime, . . . : behavior is (intentionally) not fully specified
   - correct result on one execution does not even guarantee correct result on another execution with the same input
   - repeating test failures is non-trivial

"...testing can be used to show the presence of bugs, but never to show their absence!"

— E. Dijkstra

What is Model Checking?





A wide variety of techniques, all involving three essential tasks:

1. Represent the program as a mathematically precise finite state model
   - automata, program graph, labeled transition system, . . .
2. Formulate program properties as logical properties of the model
   - first order logic, temporal logic, . . .
3. Use automated algorithmic techniques to check the model satisfies the properties
   - graph search algorithms, computer algebra, automated theorem proving, . . .

Model Checking Example: Shared Resource

```
  boolean x;
  proc rw0 {
    while (true) {
0     x := 0;
1     synch();
2     if (x == 0)
3       use_resource();
    }
  }
  proc rw1 {
    while (true) {
0     x := 1;
1     synch();
2     if (x == 1)
3       use_resource();
    }
  }
```

Model Checking Example: Shared Resource

Property 1: Freedom from deadlock
The program does not deadlock.

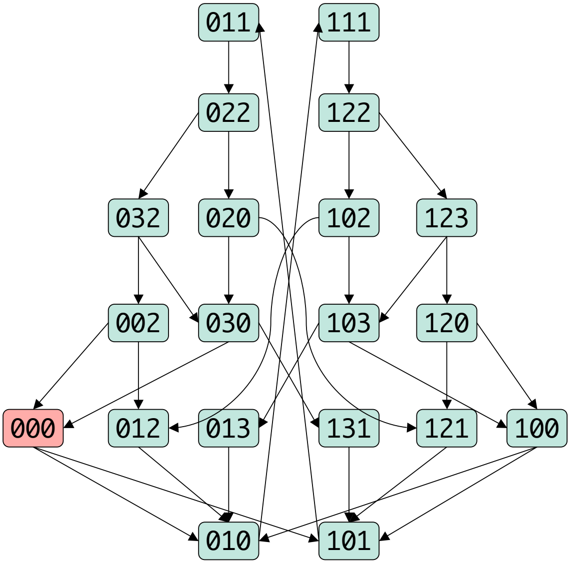
Property 2: Mutual exclusion
It is never the case that both processes use the resource at the same time.

Property 3: Liveness
The resource will eventually be used.

State: $[x, \text{pc}_0, \text{pc}_1]$

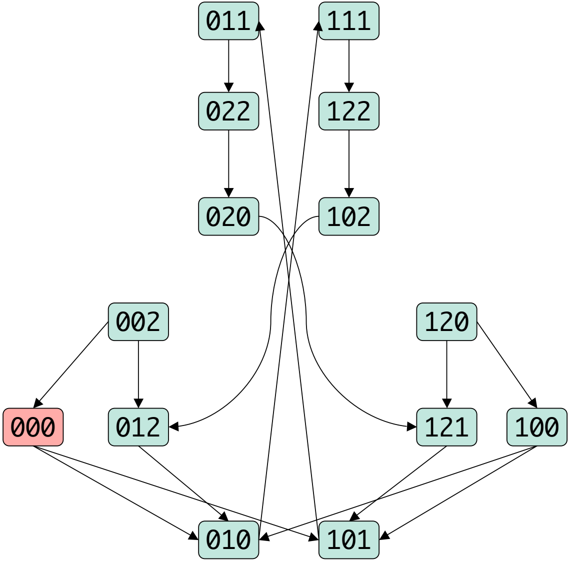
```
  boolean x;
  proc rw0 {
    while (true) {
0     x := 0;
1     synch();
2     if (x == 0)
3       use_resource();
    }
  }
  proc rw1 {
    while (true) {
0     x := 1;
1     synch();
2     if (x == 1)
3       use_resource();
    }
  }
```

Model Checking Example: Shared Resource $[x,\ \mathsf{pc}_0,\ \mathsf{pc}_1]$



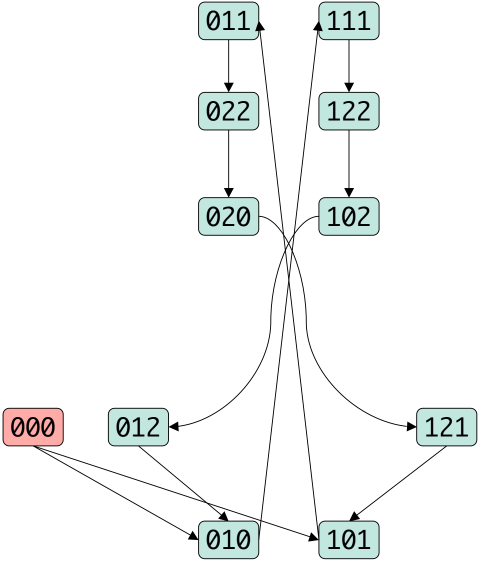

```
    boolean x;
    proc rw0 {
      while (true) {
0       x := 0;
1       synch();
2       if (x == 0)
3         use_resource();
      }
    }
    proc rw1 {
      while (true) {
0       x := 1;
1       synch();
2       if (x == 1)
3         use_resource();
      }
    }
```

Model Checking Example: Shared Resource $[x,\ \mathrm{pc}_0,\ \mathrm{pc}_1]$



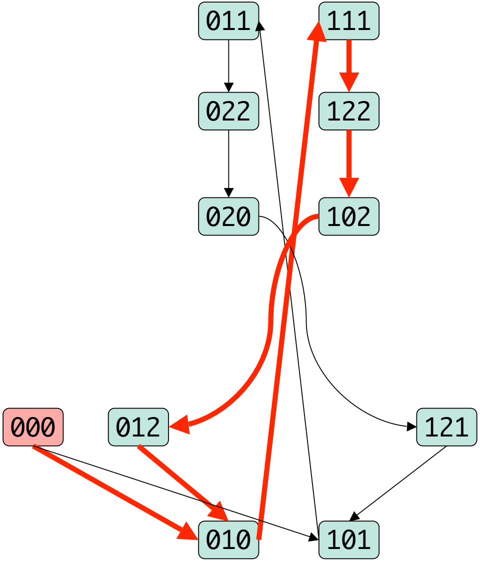

```
    boolean x;
    proc rw0 {
      while (true) {
0       x := 0;
1       synch();
2       if (x == 0)
3         use_resource();
      }
    }
    proc rw1 {
      while (true) {
0       x := 1;
1       synch();
2       if (x == 1)
3         use_resource();
      }
    }
```

Model Checking Example: Shared Resource $[x,\ \mathrm{pc}_0,\ \mathrm{pc}_1]$



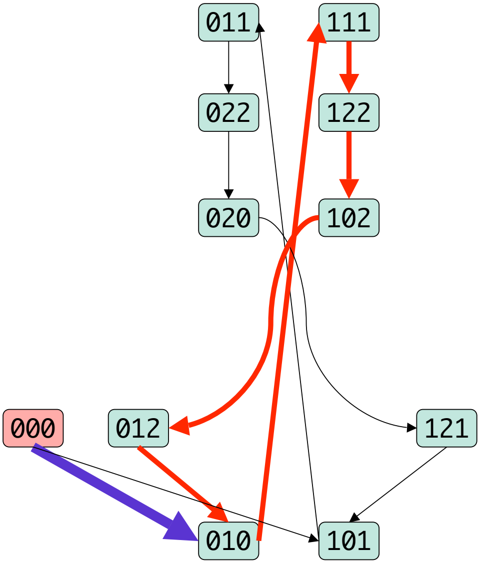

```
    boolean x;
    proc rw0 {
      while (true) {
0       x := 0;
1       synch();
2       if (x == 0)
3         use_resource();
      }
    }
    proc rw1 {
      while (true) {
0       x := 1;
1       synch();
2       if (x == 1)
3         use_resource();
      }
    }
```

Model Checking Example: Shared Resource $[x, \mathrm{pc}_0, \mathrm{pc}_1]$



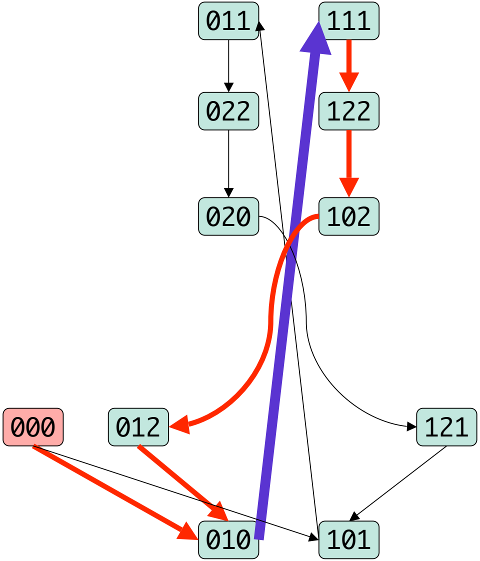

```
    boolean x;
    proc rw0 {
      while (true) {
0       x := 0;
1       synch();
2       if (x == 0)
3         use_resource();
      }
    }
    proc rw1 {
      while (true) {
0       x := 1;
1       synch();
2       if (x == 1)
3         use_resource();
      }
    }
```

Model Checking Example: Shared Resource $[x,\ \mathrm{pc}_0,\ \mathrm{pc}_1]$



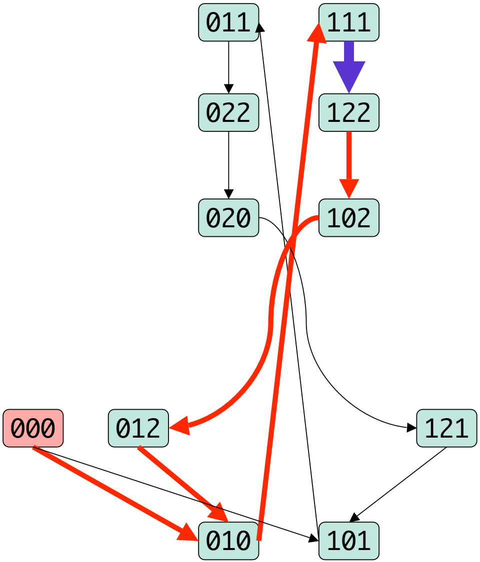

```
  boolean x;
  proc rw0 {
    while (true) {
0     x := 0;
1     synch();
2     if (x == 0)
3       use_resource();
    }
  }
  proc rw1 {
    while (true) {
0     x := 1;
1     synch();
2     if (x == 1)
3       use_resource();
    }
  }
```

Model Checking Example: Shared Resource $[x,\ \mathrm{pc}_0,\ \mathrm{pc}_1]$



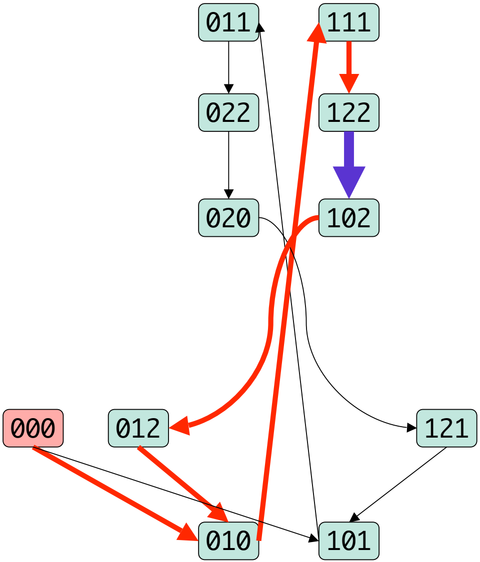

```
     boolean x;
     proc rw0 {
       while (true) {
0        x := 0;
1        synch();
2        if (x == 0)
3          use_resource();
       }
     }
     proc rw1 {
       while (true) {
0        x := 1;
1        synch();
2        if (x == 1)
3          use_resource();
       }
     }
```

Model Checking Example: Shared Resource $[x, \mathrm{pc}_0, \mathrm{pc}_1]$



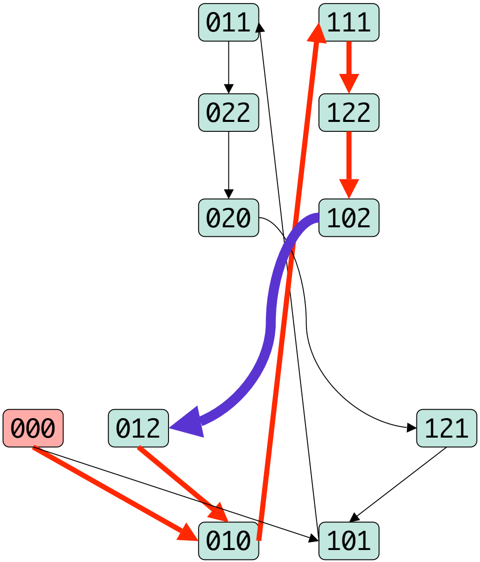

```
   boolean x;
   proc rw0 {
     while (true) {
0      x := 0;
1      synch();
2      if (x == 0)
3        use_resource();
     }
   }
   proc rw1 {
     while (true) {
0      x := 1;
1      synch();
2      if (x == 1)
3        use_resource();
     }
   }
```

Model Checking Example: Shared Resource $[x, \mathrm{pc}_0, \mathrm{pc}_1]$



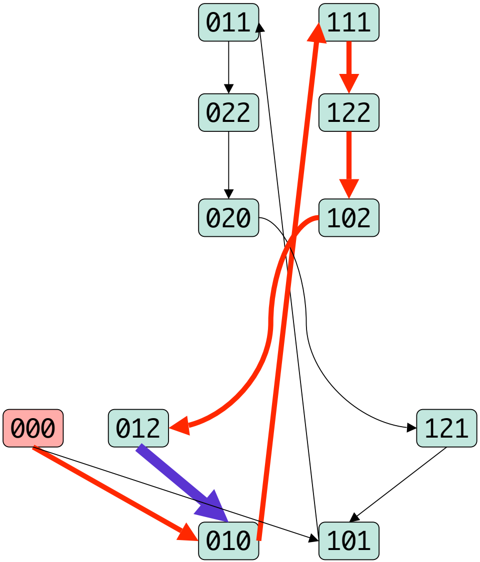

```
  boolean x;
  proc rw0 {
    while (true) {
0     x := 0;
1     synch();
2     if (x == 0)
3       use_resource();
    }
  }
  proc rw1 {
    while (true) {
0     x := 1;
1     synch();
2     if (x == 1)
3       use_resource();
    }
  }
```

Model Checking Example: Shared Resource $[x, \mathrm{pc}_0, \mathrm{pc}_1]$



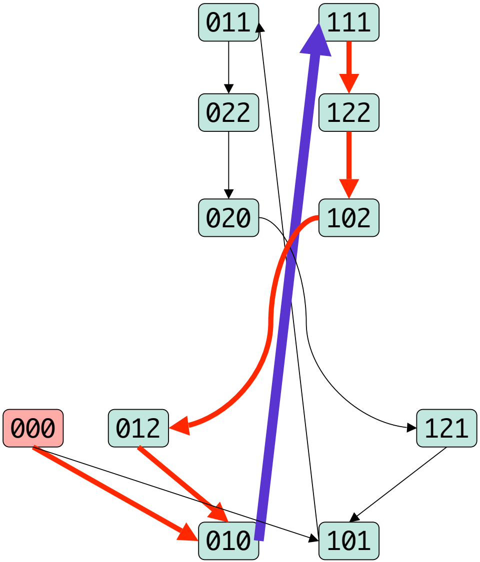

```
    boolean x;
    proc rw0 {
      while (true) {
0       x := 0;
1       synch();
2       if (x == 0)
3         use_resource();
      }
    }
    proc rw1 {
      while (true) {
0       x := 1;
1       synch();
2       if (x == 1)
3         use_resource();
      }
    }
```

Model Checking Example: Shared Resource $[x, \text{pc}_0, \text{pc}_1]$




```
    boolean x;
    proc rw0 {
      while (true) {
0       x := 0;
1       synch();
2       if (x == 0)
3         use_resource();
      }
    }
    proc rw1 {
      while (true) {
0       x := 1;
1       synch();
2       if (x == 1)
3         use_resource();
      }
    }
```

Model Checking Example: Shared Resource $[x, \mathrm{pc}_0, \mathrm{pc}_1]$




```
    boolean x;
    proc rw0 {
      while (true) {
0       x := 0;
1       synch();
2       if (x == 0)
3         use_resource();
      }
    }
    proc rw1 {
      while (true) {
0       x := 1;
1       synch();
2       if (x == 1)
3         use_resource();
      }
    }
```

Adding Symbolic Execution

- James C. King, **Symbolic execution and program testing**
  - Communications of the ACM, 1976
- intuition: many tests can be combined into one test

Adding Symbolic Execution

- James C. King, **Symbolic execution and program testing**
  - Communications of the ACM, 1976
- intuition: many tests can be combined into one test
- execute program abstractly, using symbolic constants for inputs
  - $X_1, X_2, \ldots$

Adding Symbolic Execution

- James C. King, **Symbolic execution and program testing**
  - Communications of the ACM, 1976
- intuition: many tests can be combined into one test
- execute program abstractly, using symbolic constants for inputs
  - $X_1, X_2, \ldots$
- operations result in symbolic expressions
  - $3X_1^2 + X_1X_2$

Adding Symbolic Execution

- James C. King, **Symbolic execution and program testing**
  - Communications of the ACM, 1976
- intuition: many tests can be combined into one test
- execute program abstractly, using symbolic constants for inputs
  - $X_1, X_2, \ldots$
- operations result in symbolic expressions
  - $3X_1^2 + X_1X_2$
- to deal with branches, introduce a path condition variable pc
  - boolean-valued symbolic expression
    - $X_1 > 0$

Adding Symbolic Execution

- James C. King, **Symbolic execution and program testing**
  - Communications of the ACM, 1976
- intuition: many tests can be combined into one test
- execute program abstractly, using symbolic constants for inputs
  - $X_1, X_2, \ldots$
- operations result in symbolic expressions
  - $3X_1^2 + X_1X_2$
- to deal with branches, introduce a path condition variable pc
  - boolean-valued symbolic expression
    - $X_1 > 0$
  - represents a subset of the input domain
  - records the assumptions on the input that must hold in order for a particular path to have been followed

Adding Symbolic Execution

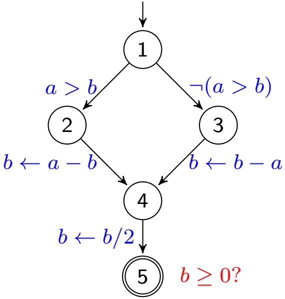
- James C. King, **Symbolic execution and program testing**
  - Communications of the ACM, 1976
- intuition: many tests can be combined into one test
- execute program abstractly, using symbolic constants for inputs
  - $X_1, X_2, \ldots$
- operations result in symbolic expressions
  - $3X_1^2 + X_1X_2$
- to deal with branches, introduce a path condition variable pc
  - boolean-valued symbolic expression
    - $X_1 > 0$
  - represents a subset of the input domain
  - records the assumptions on the input that must hold in order for a particular path to have been followed
  - at a branch on condition e, split into two cases
    1. pc $\leftarrow$ pc $\wedge\, e$, follow *true* branch
    2. pc $\leftarrow$ pc $\wedge\, \neg e$, follow *false* branch

Symbolic Execution Example

```
   input real a, b;
1  if (a>b)
2     b=a-b;
   else
3     b=b-a;
4  b=b/2.0;
5  assert b>=0;
```

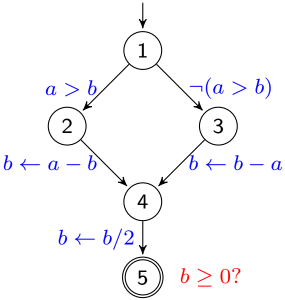
Symbolic Execution Example

```
   input real a, b;
1  if (a>b)
2     b=a-b;
   else
3     b=b-a;
4  b=b/2.0;
5  assert b>=0;
```



Symbolic Execution Example

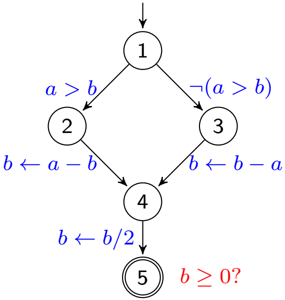
```
   input real a, b;
1  if (a>b)
2     b=a-b;
   else
3     b=b-a;
4  b=b/2.0;
5  assert b>=0;
```



State:
$\langle$pc, location, $a, b\rangle$

Symbolic Execution Example

```
   input real a, b;
1  if (a>b)
2     b=a-b;
   else
3     b=b-a;
4  b=b/2.0;
5  assert b>=0;
```



State:
$\langle \mathsf{pc}, \mathsf{location}, a, b \rangle$

$$\langle \mathsf{true}, 1, X_1, X_2 \rangle$$

$$\langle X_1 > X_2, 2, X_1, X_2 \rangle \qquad \langle \neg(X_1 > X_2), 3, X_1, X_2 \rangle$$

$$\langle X_1 > X_2, 4, X_1, X_1 - X_2 \rangle \qquad \langle \neg(X_1 > X_2), 4, X_1, X_2 - X_1 \rangle$$

$$\langle X_1 > X_2, 5, X_1, (X_1 - X_2)/2 \rangle \qquad \langle \neg(X_1 > X_2), 5, X_1, (X_2 - X_1)/2 \rangle$$

$$X_1 > X_2 \implies (X_1 - X_2)/2 \geq 0? \qquad \neg(X_1 > X_2) \implies (X_2 - X_1)/2 \geq 0?$$

Some properties checked by CIVL

- generic properties of correct C programs: absence of ...
  - out-of-bound array indexing
  - dereferences of a NULL or undefined pointer
  - pointer arithmetic beyond the bounds of an object
  - reading a variable before it is initialized
  - memory leaks, double frees, divisions by zero, integer overflow for signed integer types

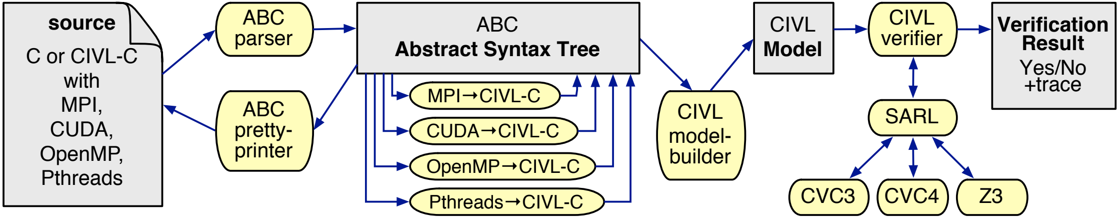
Some properties checked by CIVL

- generic properties of correct C programs: absence of . . .
  - out-of-bound array indexing
  - dereferences of a NULL or undefined pointer
  - pointer arithmetic beyond the bounds of an object
  - reading a variable before it is initialized
  - memory leaks, double frees, divisions by zero, integer overflow for signed integer types
- API-specific properties
  - each process in an MPI communicator makes the same sequence of collective calls
  - any message received will fit in the specified receive buffer
  - the program will be free of potential and absolute deadlocks
  - in OpenMP, shared variable accesses will avoid data-races

Some properties checked by CIVL

- generic properties of correct C programs: absence of ...
  - out-of-bound array indexing
  - dereferences of a NULL or undefined pointer
  - pointer arithmetic beyond the bounds of an object
  - reading a variable before it is initialized
  - memory leaks, double frees, divisions by zero, integer overflow for signed integer types
- API-specific properties
  - each process in an MPI communicator makes the same sequence of collective calls
  - any message received will fit in the specified receive buffer
  - the program will be free of potential and absolute deadlocks
  - in OpenMP, shared variable accesses will avoid data-races
- application-specific properties
  - on any execution of a program, no assertion will be violated
  - given two programs with the same input-output signature:
    - on any execution of the two programs starting from the same inputs
    - the two programs will produce the same output

The CIVL framework



- an intermediate verification language for concurrency: CIVL-C
- an analysis and verification framework centered around that language
- front-end: ABC (extended C compiler front-end) and transformers for each dialect
- back-end: a CIVL-C verifier based on model checking and symbolic execution
- Open source (L/GPL), 100% Java 17, documentation!, unit tests, coverage analysis, . . .