Principles of Concurrency

Week 13
Testing and Verification

slides adapted from
https://www.research.ibm.com/haifa/Workshops/padtad2008/present/musuvathi.ppt
Iterative Context Bounding for Systematic Testing of Multithreaded Programs

1. Introduction
Multithreaded programs are difficult to get right because specific thread interleavings, unexpected even to an expert programmer, lead to crashes that occur late in the software development cycle or even after the software is released. The traditional method for testing concurrent software in the industry is stress testing, in which the software is executed under heavy loads with the hope of producing an erroneous interaction. Empirical evidence clearly demonstrates that this form of testing is inadequate: it does not prove any notion of coverage with respect to concurrent anomalies. A scheduled for concurrency testing, the method in which the tool under test runs under heavy load with the hope of producing an erroneous interaction. Empirical evidence clearly demonstrates that this form of testing is inadequate: it does not prove any notion of coverage with respect to concurrent anomalies. A scheduled for concurrency testing, the method in which the tool under test runs under heavy load with the hope of producing an erroneous interaction. Empirical evidence clearly demonstrates that this form of testing is inadequate: it does not prove any notion of coverage with respect to concurrent anomalies.

Iterative context bounding (ICB) is a substantial improvement of traditional context bounding. ICB addresses the state-explosion problem by a different approach. Instead of running for an exponential number of iterations and checking for the same bugs, ICB systematically explores a bounded set of thread interleavings. Our tool, Iterative Context Bounding (ICB), uses a randomized scheduler to explore the system state space. ICB has several advantages over traditional context bounding methods: it is simple to implement, it requires no modification to the system under test, and it can be used in conjunction with traditional context bounding methods.

ICB works by randomly scheduling a fixed number of instructions from the program memory at a time. The scheduler explores a bounded set of thread interleavings, and if it finds a bug, it reports it. If it cannot find the bug within the time limit, it returns a negative result.

ICB is simple to implement because it requires no modification to the system under test. It can be used in conjunction with traditional context bounding methods, and it can be used to explore the entire state space of a program.

ICB requires no modification to the system under test. It can be used in conjunction with traditional context bounding methods, and it can be used to explore the entire state space of a program.

ICB requires no modification to the system under test. It can be used in conjunction with traditional context bounding methods, and it can be used to explore the entire state space of a program.

ICB requires no modification to the system under test. It can be used in conjunction with traditional context bounding methods, and it can be used to explore the entire state space of a program.

ICB requires no modification to the system under test. It can be used in conjunction with traditional context bounding methods, and it can be used to explore the entire state space of a program.

ICB requires no modification to the system under test. It can be used in conjunction with traditional context bounding methods, and it can be used to explore the entire state space of a program.

ICB requires no modification to the system under test. It can be used in conjunction with traditional context bounding methods, and it can be used to explore the entire state space of a program.

ICB requires no modification to the system under test. It can be used in conjunction with traditional context bounding methods, and it can be used to explore the entire state space of a program.

ICB requires no modification to the system under test. It can be used in conjunction with traditional context bounding methods, and it can be used to explore the entire state space of a program.

ICB requires no modification to the system under test. It can be used in conjunction with traditional context bounding methods, and it can be used to explore the entire state space of a program.

ICB requires no modification to the system under test. It can be used in conjunction with traditional context bounding methods, and it can be used to explore the entire state space of a program.
The Heisenbug problem

Concurrent executions are highly nondeterministic

Rare thread interleavings result in Heisenbugs
  ▶ Difficult to find, reproduce, and debug

Observing the bug can “fix” it
  ▶ Likelihood of interleavings changes, say, when you add printfs

A huge productivity problem
  ▶ Developers and testers can spend weeks chasing a single Heisenbug
Thread interleavings

Thread 1

```
x++;  
x++;  
```

Thread 2

```
x*=2;  
x*=2;  
```
Thread interleavings

Thread 1
x++;  
x++;  

Thread 2
x*=2;  
x*=2;  

Diagram:

1 -> 0  (Thread 1)

0 -> 1  (Thread 2)
Thread interleavings

Thread 1
- x++;
- x++;

Thread 2
- x*=2;
- x*=2;

Diagram:
- Thread 1 operations:
  - x++;
  - x++;
- Thread 2 operations:
  - x*=2;
  - x*=2;

Diagram nodes:
- 0
- 1
- 2
- 3

Execution order:
1. Thread 1: x++ -> x++
2. Thread 2: x*=2 -> x*=2

Execution:
- 1. x starts at 0
- 2. x becomes 1
- 3. x becomes 2
- 4. x becomes 0

Possible interleavings:
- 0, 1, 2, 0
- 0, 2, 1, 0
- 0, 1, 0, 2
- 0, 2, 0, 1

Principles of Concurrency, Spring 2023
Thread interleavings

Thread 1
- x++;
- x++;

Thread 2
- x*=2;
- x*=2;
CHESS in a nutshell

CHESS is a user-mode scheduler
Controls all scheduling nondeterminism
  ➤ Replace the OS scheduler

Guarantees:
  ➤ Every program run takes a different thread interleaving
  ➤ Reproduce the interleaving for every run
High level goals

Scale to large programs

Any error found by CHESS is possible in the wild
  • CHESS does not introduce any new behaviors

Any error found in the wild can be found by CHESS
  • Need to capture all sources of nondeterminism
  • Exhaustively explore the nondeterminism (state explosion)
    e.g. Enumerate all thread interleavings
  • Hard to achieve

Practical goal: beat stress testing
CHESS architecture

```
While(not done) {
  TestScenario()
}
```

- Every run takes a different interleaving
- Every run is repeatable
CHESS runs the scenario in a loop
• Every run takes a different interleaving
• Every run is repeatable

Intercept synch. & threading calls
• To control and introduce nondeterminism

Principles of Concurrency, Spring 2023
CHESS architecture

CHESS runs the scenario in a loop
- Every run takes a different interleaving
- Every run is repeatable

Intercept synch. & threading calls
- To control and introduce nondeterminism

Detect
- Assertion violations
- Deadlocks
- Dataraces
- Livelocks
Running Example

Thread 1
- Lock (l);
- bal += x;
- Unlock(l);

Thread 2
- Lock (l);
- t = bal;
- Unlock(l);
- Lock (l);
- bal = t - y;
- Unlock(l);
Introduce Schedule() points

- Instrument calls to the CHESS scheduler
- Each call is a potential preemption point
**First-cut solution: Random sleeps**

Introduce random sleep at schedule points

Does not introduce new behaviors

- Sleep models a possible preemption at each location
- Sleeping for a finite amount guarantees starvation-freedom

```
Thread 1
Sleep(rand());
Lock (l);
bal += x;
Sleep(rand());
Unlock(l);

Thread 2
Sleep(rand());
Lock (l);
t = bal;
Sleep(rand());
Unlock(l);

Sleep(rand());
Lock (l);
bal = t - y;
Sleep(rand());
Unlock(l);
```
**Improvement 1:**

**Thread 1**
- Schedule();
- Lock(l);
- bal += x;
- Schedule();
- Unlock(l);

**Thread 2**
- Schedule();
- Lock(l);
- t = bal;
- Schedule();
- Unlock(l);
- Schedule();
- Lock(l);
- bal = t - y;
- Schedule();
- Unlock(l);
Improvement 1:

Thread 1

Schedule();
Lock (l);
bal += x;
Schedule();
Unlock(l);

Schedule(); Lock (l);
t = bal;
Schedule();
Unlock(l);

Schedule(); Lock (l);
b = t - y;
Schedule();
Unlock(l);
Improvement 1:

- Delays that result in the same “happens-before” graph are equivalent
- Avoid exploring equivalent interleavings
Improvement 2:

Thread 1

Schedule();
Lock (l);
bal += x;
Schedule();
Unlock(l);

Thread 2

Schedule(); Lock (l);
t = bal;
Schedule();
Unlock(l);

Schedule(); Lock (l);
bal = t - y;
Schedule();
Unlock(l);
Improvement 2:

Thread 1:
- Schedule();
- Lock (l);
- bal += x;
- Schedule();
- Unlock(l);

Thread 2:
- Schedule();
- Lock (l);
- t = bal;

- Schedule();
- Unlock(l);

- Schedule();  Lock (l);
- bal = t - y;
- Schedule();
- Unlock(l);
Improvement 2:

- Avoid exploring delays that are impossible
- Identify when threads can make progress
- CHESS maintains a run queue and a wait queue
  - Mimics OS scheduler state

Thread 1:
Schedule();  
Lock (l);  
bal += x;  
Schedule();  
Unlock(l);

Thread 2:
Schedule();  
Lock (l);  
t = bal;

Schedule();  
Unlock(l);  
Schedule();  
Lock (l);  
bal = t - y;  
Schedule();  
Unlock(l);
Emulate execution on a uniprocessor

- Enable only one thread at a time
- Linearizes a partial-order into a total-order
- Controls the order of data-races
Emulate execution on a uniprocessor

Thread 1

Schedule(); Lock (l);
t = bal;
Schedule();
Unlock(l);

Thread 2

Schedule(); Lock (l);
bal += x;
Schedule();
Unlock(l);

Schedule(); Lock (l);
bal = t - y;
Schedule();
Unlock(l);

- Enable only one thread at a time
- Linearizes a partial-order into a total-order
- Controls the order of data-races
Capture all sources of nondeterminism?

Scheduling nondeterminism? Yes

Timing nondeterminism? Yes
  ▶ Controls when and in what order the timers fire

Nondeterministic system calls? Mostly
  ▶ CHESS uses precise abstractions for many system calls

Input nondeterminism? No
  ▶ Rely on users to provide inputs
    Program inputs, return values of system calls, files read, packets received, ...
    ▶ Good tradeoff in the short term
    But can’t find race-conditions on error handling code
    ▶ Future extensions using symbolic execution?
      (DART, jCUTE, SAGE, PEX)
State space explosion

Number of executions

\[ = O\left( n^{nk} \right) \]

Exponential in both \( n \) and \( k \)

- Typically: \( n < 10 \quad k > 100 \)

Limits scalability to large programs

Thread 1

\[ x = 1; \]
\[ \ldots \]
\[ \ldots \]
\[ \ldots \]
\[ y = k; \]

Thread \( n \)

\[ x = 1; \]
\[ \ldots \]
\[ \ldots \]
\[ \ldots \]
\[ y = k; \]

k steps each

n threads
State space explosion

Number of executions

\[ \text{Number of executions} = O(n^{nk}) \]

Exponential in both \( n \) and \( k \)

Typically: \( n < 10 \quad k > 100 \)

Limits scalability to large programs

Goal: Scale CHESS to large programs (large \( k \))
Preemption bounding

- Prioritize executions with small number of preemptions

- Two kinds of context switches:
  - Preemptions – forced by the scheduler
    - e.g. Time-slice expiration
  - Non-preemptions – a thread voluntarily yields
    - e.g. Blocking on an unavailable lock, thread end
Preemption bounding

- Prioritize executions with small number of preemptions

- Two kinds of context switches:
  - Preemptions – forced by the scheduler
    - e.g. Time-slice expiration
  - Non-preemptions – a thread voluntarily yields
    - e.g. Blocking on an unavailable lock, thread end

```c
x = 1;
if (p != 0) {
    x = p->f;
}
```
Preemption bounding

- Prioritize executions with small number of preemptions

- Two kinds of context switches:
  - Preemptions – forced by the scheduler
    - e.g. Time-slice expiration
  - Non-preemptions – a thread voluntarily yields
    - e.g. Blocking on an unavailable lock, thread end

```c
x = 1;
if (p != 0) {
    p = 0;
    preemption
}
x = p->f;
```
So, is CHESS is unsound?

Soundness: prove that the program is correct for a given input test harness
- Need to exhaustively explore all interleavings

For small programs, CHESS is sound
- Iteratively increase the preemption bound

Preemption bounding helps scale to large programs
- A good “knob” to trade resources for coverage

Better search algorithms $\rightarrow$ more coverage faster
- Partial-order reduction
- Modular testing of loosely-coupled programs
Concurrent programs have cyclic state spaces

### Thread 1

L1: while(!done) {
    L2: Sleep();
}

### Thread 2

M1: done = 1;
Concurrent programs have cyclic state spaces

Thread 1

L1: while(!done) {
    L2: Sleep();
}

Thread 2

M1: done = 1;

! done
    L1

! done
    L2
Concurrent programs have cyclic state spaces

Thread 1

L1: while( ! done) {
L2: Sleep();
}

Thread 2

M1: done = 1;
Concurrent programs have cyclic state spaces

```
L1: while(!done) {
L2:   Sleep();
}
```

```
M1: done = 1;
```
Concurrent programs have cyclic state spaces

- Spinlocks
- Non-blocking algorithms
- Implementations of synchronization primitives
- Periodic timers
- ...

```
L1: while (! done) {
L2:    Sleep();
}
```

```
M1: done = 1;
```
A demonic scheduler unrolls any cycle ad-infinitum

Thread 1

while( ! done) {
    Sleep();
}

Thread 2

done = 1;
A demonic scheduler unrolls any cycle ad-infinitum

```c
while(!done) {
    Sleep();
}
done = 1;
```

Thread 1

Thread 2
A demonic scheduler unrolls any cycle ad-infinitum

Thread 1

```c
while( ! done )
{
    Sleep();
}
```

Thread 2

```c
done = 1;
```

![Diagram showing the unrolling of a cycle with two threads and variables ! done and done.](image-url)
A demonic scheduler unrolls any cycle ad-infinitum

Thread 1

while( ! done) 
{ 
    Sleep();
} 

Thread 2

done = 1;

! done

! done

! done

done

done

done
A demonic scheduler unrolls any cycle ad-infinitum

Thread 1

while(!done)
{
    Sleep();
}

Thread 2

done = 1;

! done

! done

! done

! done

done

done

done
Depth bounding

Prune executions beyond a bounded number of steps
Problem 1: Ineffective state coverage

Bound has to be large enough to reach the deepest bug
  - Typically, greater than 100 synchronization operations

Every unrolling of a cycle redundantly explores reachable state space
Problem 2: Cannot find livelocks

Livelocks: lack of progress in a program

Thread 1

```plaintext
temp = done;
while(!temp)
{
    Sleep();
}
```

Thread 2

```plaintext
done = 1;
```
Key idea

while(!done) {
    Sleep();
}

done = 1;
Key idea

Thread 1

while( ! done)  
{  
    Sleep();  
}  

done = 1;

Thread 2

! done

done

! done

done
Key idea

- This test terminates only when the scheduler is fair
- Fairness is assumed by programmers

All cycles in correct programs are unfair
A fair cycle is a livelock

```c
while(!done)
{
    Sleep();
}
done = 1;
```
We need a fair demonic scheduler
We need a fair demonic scheduler

Avoid unrolling unfair cycles
  ▶ Effective state coverage
We need a fair demonic scheduler

- Avoid unrolling unfair cycles
  - Effective state coverage

- Detect fair cycles
  - Find livelocks (violations of fair termination)
Fair termination allows CHESS to check for arbitrary liveness properties

Example: Good Samaritan assumption
- For all threads $t: \text{scheduled}(t) \rightarrow \text{yield}(t)$
- A thread when scheduled infinitely often yields the processor infinitely often

Examples of yield:
- `Sleep()`, `ScheduleThread()`, `asm \{rep nop;\}`
- Thread completion
Probabilistic Concurrency Testing

- Also a random scheduler, but uses randomization sparingly
  - Repeated independent runs increase probability of finding a bug
- Naive approach is exponential (n threads with k instructions has $n^k$ possible thread schedules)
- But, bugs in practice are not adversarial
  - small number of instructions executed by small number of threads
  - goal: schedule these instructions correctly
- The depth of a concurrency bug is the minimum number of scheduling constraints sufficient to find the bug
  - PCT focusses on probabilistically finding bugs at a given depth
  - Can find a bug at depth $d$ in $O(nk^{d-1})$ independent runs
### Ordering Edges and Depth

#### (a) manifests whenever the conditional in T2 executes before T1
- (a) manifests whenever the conditional in T2 executes before T1.

#### (b) manifests when two ordering constraints are satisfied
- (b) manifests when two ordering constraints are satisfied.

#### (c) also manifests under two ordering constraints
- (c) also manifests under two ordering constraints.
Three of the bugs reported by prior work as atomicity violations [24]. For instance, not all atomicity violations have a depth of 2, and in fact, by circular lock acquisition [15] have depth.

Concurrency bugs correspond to bugs of low depth. For example, ordering constraints sufficient to find the bug. Any schedule that satisfies these ordering constraints is sufficient to find the bug. It is possible for different sets of ordering constraints to provide better guarantees for bugs with smaller depth.

We classify concurrency bugs according to a metric. Intuitively, deeper bugs are inherently harder to find. PCT is designed to further improve PCT.

PCT is orthogonal to heuristic-directed testing methods above, in particular, deeper bugs are inherently harder to find. PCT is designed to provide a guaranteed probability of finding bugs in every run of the program with complex control flow, the depth of a bug might not be consistent across different runs. Thus, the presence of control flow increases the bug depth to 2.

Consider a naive randomized scheduler that flips a coin in each scheduling decision. Using a randomized scheduler may appear like an obvious choice.

For example, Fig. 1 shows examples of common concurrency bugs. Concurrency bugs happen when instructions are scheduled in an order not envisioned by the programmer. We identify a set of these instructions not relevant to the bug. For the examples shown, a single ordering constraint (black arrow) is sufficient to find the bug. Any schedule that satisfies these ordering constraints is sufficient to find the bugs. However, it is not a priori clear how to design such a scheduler with these constraints explicitly. It relies on the mere fact that there are cycles in the scheduling graph.

Another characterization of a concurrency bug is its depth, which is defined as the minimum number of ordering constraints that are sufficient to trigger the same bug. In such a case, we identify a set of these constraints explicitly. It relies on the mere fact that there are cycles in the scheduling graph.

Figure 2. Figure 4 shows a slight modification to Fig. 1(a). In this example, the thread preempts the main thread.

Figure 3.

Figure 4.

- Freeing the mutex before it is acquired.
- Setting a global variable in one thread and accessing it in another.
- Initializing a thread that accesses a variable.
- Accessing a variable that is initialized in another thread.

For the examples shown, a single ordering constraint (black arrow) is sufficient to find the bug. Any schedule that satisfies these ordering constraints is sufficient to find the bugs. However, it is not a priori clear how to design such a scheduler with these constraints explicitly. It relies on the mere fact that there are cycles in the scheduling graph.

Another characterization of a concurrency bug is its depth, which is defined as the minimum number of ordering constraints that are sufficient to trigger the same bug. In such a case, we identify a set of these constraints explicitly. It relies on the mere fact that there are cycles in the scheduling graph.

For example, Fig. 1 shows examples of common concurrency bugs. Concurrency bugs happen when instructions are scheduled in an order not envisioned by the programmer. We identify a set of these instructions not relevant to the bug. For the examples shown, a single ordering constraint (black arrow) is sufficient to find the bug. Any schedule that satisfies these ordering constraints is sufficient to find the bugs. However, it is not a priori clear how to design such a scheduler with these constraints explicitly. It relies on the mere fact that there are cycles in the scheduling graph.

Another characterization of a concurrency bug is its depth, which is defined as the minimum number of ordering constraints that are sufficient to trigger the same bug. In such a case, we identify a set of these constraints explicitly. It relies on the mere fact that there are cycles in the scheduling graph.

For example, Fig. 1 shows examples of common concurrency bugs. Concurrency bugs happen when instructions are scheduled in an order not envisioned by the programmer. We identify a set of these instructions not relevant to the bug. For the examples shown, a single ordering constraint (black arrow) is sufficient to find the bug. Any schedule that satisfies these ordering constraints is sufficient to find the bugs. However, it is not a priori clear how to design such a scheduler with these constraints explicitly. It relies on the mere fact that there are cycles in the scheduling graph.
Randomization

Two bugs of depth 1 difficult to find using a pure randomized scheduler

The PCT Algorithm
- instead, use a priority-based scheduler
- the scheduler schedules a low priority thread only when all higher priority threads are blocked
- threads change priorities when they pass a priority change point
- randomly assign initial priority values and randomly pick priority change points
Verification vs. Testing

- Testing provides assurance by exploring (sampling) points in an application’s input space
  - Premised on the assumption that this sample is representative of the entire space
  - Challenge is determining how best to define the sampler

- Verification provides assurance by demonstrating that a property holds for all elements in the input space
  - Proofs provide stronger guarantees than tests
  - But, the challenge is identifying a proof method that scales (e.g., even in the presence of an unbounded input space)
Properties

- Application is free from deadlock
- Application is free from livelock, starvation
- Application satisfies performance (realtime) guarantees
- There is no path through the application’s control-flow that results in an assertion violation
- Application adheres to a protocol specification

Hard to realize in a sequential setting

Exponentially harder in a concurrent setting because of the additional (non-deterministic) interleavings that manifest among threads
Model checking is an automated technique that, given a finite-state model of a system and a logical property, systematically checks whether this property holds for (a given initial state in) that model. (Clarke and Emerson, 1981)

Goal: verify $M \models \phi$

where $M$ is a finite-state model and $\phi$ is a property stated in some formal logic

Problem: exponential state-space explosion
- Prove the correctness of the model with respect to an implementation
- Alternatively, find errors in the model prior to implementation
The Model Checker SPIN

Gerard J. Holzmann

Abstract—SPIN is an efficient verification system for models of distributed software systems. It has been used to detect design errors in applications ranging from high-level descriptions of distributed algorithms to detailed code for controlling telephone exchanges. This paper gives an overview of the design and structure of the verifier, reviews its theoretical foundation, and gives an overview of significant practical applications.

Index Terms—Formal methods, program verification, design verification, model checking, distributed systems, concurrency.

1 INTRODUCTION

SPIN is a generic verification system that supports the design and verification of asynchronous process systems [36], [38]. SPIN verification models are focused on proving the correctness of process interactions, and they attempt to abstract as much as possible from internal sequential computations. Process interactions can be specified in SPIN with rendezvous primitives, with asynchronous messages passing through buffered channels, through access to shared variables, or with any combination of these. In focusing on asynchronous control in software systems, rather than synchronous control in hardware systems, SPIN distinguishes itself from other well-known approaches to model checking, e.g., [12], [49], [53].

As a formal methods tool, SPIN aims to provide:

1) an intuitive, program-like notation for specifying design choices unambiguously, without implementation detail;
2) a powerful, concise notation for expressing general correctness requirements, and
3) a methodology for establishing the logical consistency of the design choices from 1) and the matching correctness requirements from 2).

Many formalisms have been suggested to address the first two items, but rarely are the language choices directly related to a basic feasibility requirement for the third item. In SPIN the notations are chosen in such a way that the logical consistency of a design can be demonstrated mechanistically by the tool. SPIN accepts design specifications written in the verification language Promela (a Process Meta Language) [36], and it accepts correctness claims specified in the syntax of standard Linear Temporal Logic (LTL) [60].

There are no general decision procedures for unbounded systems, and one could well question the soundness of a design that would assume unbounded growth. Models that can be specified in Promela are, therefore, always required to be bounded, and have only countably many distinct behaviors. This means that all correctness properties automatically become formally decidable, within the constraints that are set by problem size and the computational resources that are available to the model checker to render the proofs. All verification systems, of course, do have physical limitations that are set by problem size, machine memory size, and the maximum runtime that the user is willing, or able, to endure. These constraints are an often neglected issue in formal verification. We study the limitations of the model checker explicitly and offer relief strategies for problems that are outside the normal domain of exhaustive proof. Such strategies are discussed in Sections 3.3 and 3.4 of this paper.

1.1 Structure

The basic structure of the SPIN model checker is illustrated in Fig. 1. The typical mode of working is to start with the specification of a high level model of a concurrent system, or distributed algorithm, typically using SPIN’s graphical front-end XSPIN. After fixing syntax errors, interactive simulation is performed until basic confidence is gained that the design behaves as intended. Then, in a third step, SPIN is used to generate an optimized on-the-fly verification program from the high level specification. This verifier is compiled, with possible compile-time choices for the types of reduction algorithms to be used, and executed. If any counterexamples to the correctness claims are detected, these can be fed back into the interactive simulator and inspected in detail to establish, and remove, their cause.

The remainder of this paper consists of three main parts. Section 2 gives an overview of the basic verification method that SPIN employs. Section 3 summarizes the basic algorithms and complexity management techniques that have been implemented. Section 4 gives three examples of typical applications of the SPIN model checker to design and verification problems. The first example is the problem of devising a correct process scheduler for a distributed operating system; the second problem is the verification of a leader election protocol for a distributed ring; the third problem is the proof of correctness of a standard sliding window flow control protocol. Section 4 concludes with a summary of a range of other significant verification problems to which SPIN has been applied. Section 5 concludes the paper.

An automated tool for checking the logical consistency of asynchronous systems
Uses a specification/modeling language called Promela (Protocol/Process MetaLanguage)
- communication messaging
  - synchronous
  - asynchronous

---

© 1996 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
A model consists of:
- Declarations (types, channel, processes)
- Definition of a finite-state transition system
- No notion of “unboundedness”

**Promela model**

- Process
  - Local-state
  - Communication via channels and global variables
  - Can be created arbitrarily

```promela
proctype Sender(chan in; chan out) {
  bit sndB, rcvB;
  do
    :: out ! MSG, sndB ->
    in ? ACK, rcvB;
    if
      :: sndB == rcvB -> sndB = 1-sndB
    else -> skip
    fi
  od
}
```

The body consists of a sequence of statements.
Process Statements

- The body of a process consists of a sequence of statements
- A statement is either executable or blocked
- Assignment and assert statements are always executable

```c
int x;
proctype Aap()
{
    int y=1;
    skip;
    run Noot();
    x=2;
    x>2 && y==1;
    skip;
}
```

Executable if **Noot** can be created...

Can only become executable if a **some other process** makes `x` greater than 2.
Promela statements

- **skip** always executable
- **assert(<expr>)** always executable
- **expression** executable if not zero
- **assignment** always executable
- **if** executable if at least one guard is executable
- **do** executable if at least one guard is executable
- **break** always executable (exits do-statement)
- **send (ch!)** executable if channel ch is not full
- **receive (ch?)** executable if channel ch is not empty
Interleaving Semantics

- Promela processes execute concurrently.
- Non-deterministic scheduling of the processes.
- Processes are interleaved (statements of different processes do not occur at the same time).
  - Exception: rendez-vous communication.
- All statements are atomic; each statement is executed without interleaving with other processes.
- Each process may have several different possible actions enabled at each point of execution.
  - Only one choice is made, non-deterministically.

(X)SPIN Architecture

- Promela model $M$ is translated into Xspin.
- Xspin is compiled into $\phi$.
- $\phi$ is translated into LTL.
- LTL is translated into SPIN.
- SPIN is simulated and verified.
- C program is generated.
- M |= $\phi$

- Deadlocks
- Safety properties
- Liveness properties

- Editing window
- Simulation options
- Verification options
- MSC simulation window

- Random, guided, interactive

- Counter example
- Checker
- $\mid = \mid$

- Pan.*
**Mutual Exclusion**

```c
bit flag; /* signal entering/leaving the section */
byte mutex; /* # procs in the critical section */

proctype P(bit i) {
    flag != 1;
    flag = 1;
    mutex++;
    printf("MSC: P(%d) has entered section.\n", i);
    mutex--;
    flag = 0;
}

proctype monitor() {
    assert(mutex != 2);
}

init {
    atomic { run P(0); run P(1); run monitor(); } 
}
```

Problem: **assertion violation! Both processes can pass the flag != 1 “at the same time”, i.e. before flag is set to 1.**

starts two instances of process P
Mutual Exclusion

```cpp
bit x, y;      /* signal entering/leaving the section */
byte mutex;    /* # of procs in the critical section. */

active proctype A() {
  x = 1;
  y == 0;
  mutex++;
  mutex--;
  x = 0;
}

active proctype B() {
  y = 1;
  x == 0;
  mutex++;
  mutex--;
  y = 0;
}

active proctype monitor() {
  assert(mutex != 2);
}
```

Process A waits for process B to end.

Problem: invalid-end-state!
Both processes can pass execute \( x = 1 \) and \( y = 1 \) “at the same time”, and will then be waiting for each other.
Mutual Exclusion

```
bit x, y;    /* signal entering/leaving the section */
byte mutex;  /* # of procs in the critical section. */
byte turn;   /* who's turn is it? */

active proctype A() {
    x = 1;
    turn = B_TURN;
    y == 0 ||
        (turn == A_TURN);
    mutex++;
    mutex--;
    x = 0;
}

active proctype B() {
    y = 1;
    turn = A_TURN;
    x == 0 ||
        (turn == B_TURN);
    mutex++;
    mutex--;
    y = 0;
}

active proctype monitor() {
    assert(mutex != 2);
}
```

Can be generalised to a single process.

First "software-only" solution to the mutex problem (for two processes).

Theo C. Ruys - SPIN Beginners' Tutorial
Mutual Exclusion

byte turn[2]; /* who's turn is it? */
byte mutex;  /* # procs in critical section */

proctype P(bit i) {
  do ::
    turn[i] = 1;
    turn[i] = turn[1-i] + 1;
    (turn[1-i] == 0) || (turn[i] < turn[1-i]);
    mutex++;
    mutex--;
  od
  turn[i] = 0;
}

proctype monitor() { assert(mutex != 2); }
init { atomic { run P(0); run P(1); run monitor();} }

Problem (in Promela/SPIN): turn[i] will overrun after 255.

More mutual exclusion algorithms in (good-old) [Ben-Ari 1990].
Safety and Liveness

Safety:
- “Nothing bad ever happens”
- invariants
- deadlock-freedom
- Model-checker: try to find a violating trace

Liveness:
- “Something good eventually happens”
- termination
- reactivity
- Model-checker: search for an infinite loop in which something good does not happen
• **SPIN** uses a depth first search algorithm (DFS) to generate and explore the complete state space.

```plaintext
procedure dfs(s: state) {
    if error(s)
        reportError();
    foreach (successor t of s) {
        if (t not in Statespace)
            dfs(t);
    }
}
```

- States are stored in a hash table.
- Requires state matching.
- The old states $s$ are stored on a stack, which corresponds with a complete execution path.
- Only works for state properties.

• Note that the **construction and error checking** happens at the same time: SPIN is an on-the-fly model checker.
SPIN combats exponential search space using a number of reduction techniques:
- partial order reduction
- bitstate hashing
- state vector compression
- dataflow analysis
- slicing

Partial order reduction:
- intuition: exploit conditions in which interleaved operations do not affect final outcome (e.g., commutativity)
- Example: if in state, process P executes only “local” statements, then the actions of other processes can be deferred until P finishes
- local: access only local variables
- receive or send data from/to an exclusive queue
Suppose the statements of P1 and P2 are all local.
Liveness Specifications

LTL formulae: propositional formula with temporal operators:

- □ P - formula P is true now and forever into the future
- ◊ P - formula P is satisfied at some point in the future

Linear temporal logic refers to the underlying nature of time and the choices possible in the future:
- linear: each point in time has a well-defined successor
- branching: each point in time has multiple possible futures
- think of time in terms of ordering of events

• p → ◊q p implies eventually q (response)
• p → q U r p implies q until r (precedence)
• □◊p always eventually p (progress)
• ◊□p eventually always p (stability)
• ◊p → ◊q eventually p implies eventually q (correlation)

• Eventually ◊φ := true U φ
• Always □φ := ¬◊¬φ
**Example**

**System description**
- Focus on lights in on particular direction
- Light can be any of three colors: green, yellow, read
- Atomic propositions = light color

**Ordering specifications**
- Liveness: “traffic light is green infinitely often”
  \[ □ \Diamond \text{green} \]
- Chronological ordering: “once red, the light cannot become green immediately”
  \[ □ (\text{red} \rightarrow \neg \Diamond \text{green}) \]
- More detailed: “once red, the light always becomes green eventually after being yellow for some time”
  \[ □ (\text{red} \rightarrow (\Diamond \text{green} \land (\neg \Diamond \text{green} \cup \text{yellow}))) \]
  \[ □ (\text{red} \rightarrow \Diamond (\text{red} \cup (\text{yellow} \land (\Diamond \text{yellow} \cup \Diamond \text{green})))) \]

**Progress property**
- Every request will eventually lead to a response
  \[ □ (\text{request} \rightarrow \Diamond \text{response}) \]
Temporal Logic of Actions

- Formulas (aka specifications) are built using:
  - values, variables, states, functions, and actions
  - an action is a Boolean-valued expression that relates states
- Temporal logic is a formalism that reasons about behavior in terms of sequences of states (time progresses in a linear order)
- Used to describe the dynamic behavior of concurrent/reactive systems
- Focussed on catching design (rather than implementation) errors
- Used in industry

How Amazon Web Services Uses Formal Methods
Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff
Communications of the ACM
April 2015, Vol. 58, No. 4, pages 66–73
Specifications

- A mathematical description language, compiler, and model-checker for describing and proving properties of concurrent and distributed systems

Example: \( \text{CHOOSE } x \in S : \forall y \in S : y \leq x \).

Built on top of propositional logic

\( \land \) conjunction (and) \( \Rightarrow \) implication (implies)

\( \lor \) disjunction (or) \( \equiv \) equivalence (is equivalent to)

\( \neg \) negation (not)

Sets, and the following two primitive predicates:

\( \forall \) universal quantification (for all)

\( \exists \) existential quantification (there exists)
Abstraction

- An execution is represented as a sequence of discrete steps
  - simulate a concurrent program as a sequential interleaving of its threads
- A step denotes a change in state
- A sequence of states is a behavior
  - Goal: describe all possible behaviors of a design/system
  - Abstract behaviors as state machines:
    - All possible initial states
    - Actions: next states
    - States:
      - variables
      - initial values
      - relations among them in the current state
      - relations between values in the current state and next state
Example

In English

```c
int i;
void main()
{
  i = someNumber();
  i = i + 1;
}
```

In TLA+

```plaintext
EXTENDS Integers

VARIABLES i, pc

Init ≜ (pc = "start") \land (i = 0)

Next ≜ \forall \land pc = "start"
  \land i' \in 0..1000
  \land pc' = "middle"
\lor \land pc = "middle"
  \land i' = i + 1
  \land pc' = "done"
```

In English:

If the current value of \(pc\) equals "start"

- then the next value of \(i\) in \{0, 1, \ldots, 1000\}
- next value of \(pc\) equals "middle"

else if the current value of \(pc\) equals "middle"

- then the next value of \(i\) equals the current value of \(i + 1\)
- next value of \(pc\) equals "done"

else no next values

OK. This isn’t very easy to read.

It’s simpler and more elegant in TLA+,
because it’s written as a mathematical formula.

But we’ll get to that later.
Example: Specifying an Hour Clock

$$HCini \triangleq hr \in \{1, \ldots, 12\}$$

$$HCnxt \triangleq hr' = \text{IF } hr \neq 12 \text{ THEN } hr + 1 \text{ ELSE } 1$$

$$HC \triangleq HCini \land \Box [HCnxt]_{hr}$$

admissible behavior (a possibly infinite collection of states):

$$[hr = 10] \rightarrow [hr = 11] \rightarrow [hr = 11] \rightarrow [hr = 11] \rightarrow \cdots$$
--- MODULE HourClock ---

EXTENDS Naturals
VARIABLE hr
HCini  ==  hr \in (1 .. 12)
HCnxt  ==  hr' = IF hr # 12 THEN hr + 1 ELSE 1
HC    ==  HCini \[ HCnxt ]_hr

THEOREM HC => []HCini

---
Specifying Asynchrony

Module AsynchInterface

EXTENDS Naturals

CONSTANT Data

VARIABLES val, rdy, ack

TypeInvariant \( \triangleq \) \( \land \) val \( \in \) Data
\( \land \) rdy \( \in \) \{0, 1\}
\( \land \) ack \( \in \) \{0, 1\}

Init \( \triangleq \) \( \land \) val \( \in \) Data
\( \land \) rdy \( \in \) \{0, 1\}
\( \land \) ack = rdy

Send \( \triangleq \land \) rdy = ack
\( \land \) val' \( \in \) Data
\( \land \) rdy' = 1 \(-\) rdy
\( \land \) UNCHANGED ack

Recv \( \triangleq \land \) rdy \( \neq \) ack
\( \land \) ack' = 1 \(-\) ack
\( \land \) UNCHANGED \( \langle \) val, rdy \( \rangle \)

Next \( \triangleq \) Send \( \lor \)Recv
Spec \( \triangleq \) Init \( \land \) □[Next] \( \langle \) val, rdy, ack \( \rangle \)

THEOREM Spec \( \Rightarrow \) □TypeInvariant
A FIFO Queue

**MODULE Channel**

EXTENDS Naturals, Sequences

CONSTANT Message

VARIABLE chan

\[ \text{TypeInvariant} \triangleq \text{chan } \in \{ \text{val : Data, rd} y : \{0, 1\}, \text{ack} : \{0, 1\} \} \]

**Init** \( \triangleq \text{TypeInvariant} \)
\[ \wedge \text{chan.ack } = \text{chan.rdy} \]

\[ \text{Send(d)} \triangleq \text{chan.rdy } = \text{chan.ack} \]
\[ \wedge \text{chan'} = [\text{chan} \text{ EXCEPT !}.\text{val } = d, \!\text{.rdy } = 1 - \text{@}] \]

**Rcv** \( \triangleq \text{chan.rdy } \neq \text{chan.ack} \)
\[ \wedge \text{chan'} = [\text{chan} \text{ EXCEPT !}.\text{ack } = 1 - \text{@}] \]

**Next** \( \triangleq (\exists d \in \text{Data} : \text{Send(d)}) \lor \text{Rcv} \)

\[ \text{Spec} \triangleq \text{Init} \wedge [\text{Next}]_{\text{chan}} \]

**THEOREM** Spec \( \Rightarrow \Box \text{TypeInvariant} \)

**MODULE FIFO**

EXTENDS Naturals, Sequences

CONSTANT Message

VARIABLES in, out

**Inner(q)** \( \triangleq \text{instance InnerFIFO} \)

\[ \text{Spec} \triangleq \exists q : \text{Inner(q)}!\text{Spec} \]

**THEOREM** Spec \( \Rightarrow \Box \text{TypeInvariant} \)
- Handles specifications of the form:

\[ \text{Init} \land \Box[\text{Next}]_{\text{vars}} \land \text{Temporal} \]

- Without specifications, TLC checks for:
  - type errors
  - deadlock, i.e., violation of:
    \[ \Box(\text{ENABLED Next}). \]
- TLC generates behaviors that satisfy a specification based on a model (i.e., an assignment of values to variables in the specification)
  - check all reachable states i.e., find all states that can occur in behaviors satisfying:
    \[ \text{Init} \land \Box[\text{Next}]_{\text{vars}}. \]
- Checking all states is usually impossible (e.g., arbitrarily many messages) - infinitely many states/paths.
  - In practice, “finitize” the model