Verification of concurrent programs

Lecture 13 of TDA384/DIT391
(Principles of Concurrent Programming)

Carlo A. Furia

Chalmers University of Technology – University of Gothenburg

SP3 2017/2018

Today's menu

Finite-state models of concurrency: recap

Specification

Verification

Testing

Model checking

Finite-state models of concurrency: recap

State/transition diagrams
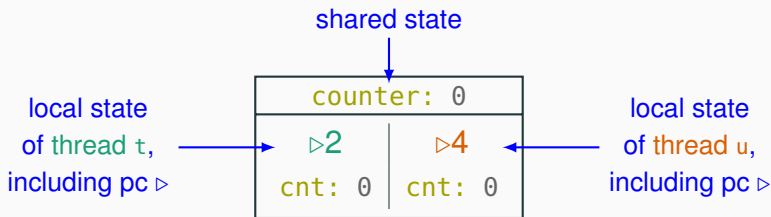
We capture the essential elements of concurrent programs using state/transition diagrams (also called: (finite) state automata, (finite) state machines, or transition systems).

- states in a diagram capture possible program states
- transitions connect states according to execution order

Structural properties of a diagram capture semantic properties of the corresponding program.

States

A state captures the shared and local states of a concurrent program:

shared state

counter: 0	
▷2	▷4
cnt: 0	cnt: 0

local state
of thread t,
including pc ▷

local state
of thread u,
including pc ▷

int counter = 0;

thread t	thread u	
int cnt;	**int** cnt;	
1 cnt = counter;	cnt = counter;	4
2 counter = cnt + 1;	counter = cnt + 1;	5
3 // terminates	// terminates	6

A state captures the shared and local states of a concurrent program:

counter: 0	
▷2	▷4
cnt: 0	cnt: 0

When unambiguous, we simplify a state with only the essential information:

0	
▷2	▷4
0	0

Initial states

The initial state of a computation is marked with an incoming arrow:
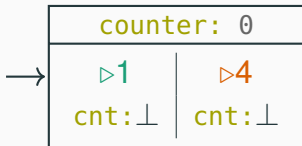
$$\longrightarrow \boxed{\begin{array}{c} \texttt{counter: 0} \\ \hline \begin{array}{c|c} \triangleright 1 & \triangleright 4 \\ \texttt{cnt:} \bot & \texttt{cnt:} \bot \end{array} \end{array}}$$

int counter = 0;

thread t	thread u	
int cnt;	**int** cnt;	
1 cnt = counter;	cnt = counter;	4
2 counter = cnt + 1;	counter = cnt + 1;	5
3 *// terminates*	*// terminates*	6

Final states

The final states of a computation – where the program terminates –
are marked with double-line edges:
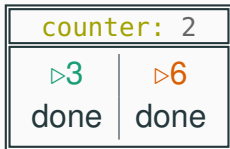

```
                    int counter = 0;
```

thread t	thread u	
`int cnt;`	`int cnt;`	
1 `cnt = counter;`	`cnt = counter;`	4
2 `counter = cnt + 1;`	`counter = cnt + 1;`	5
3 `// terminates`	`// terminates`	6

A transition corresponds to the execution of one atomic instruction, and it is an arrow connecting two states (or a state to itself):


```
                        int counter = 0;
        thread t                         thread u
    int cnt;                         int cnt;
1   cnt = counter;                   cnt = counter;            4
2   counter = cnt + 1;               counter = cnt + 1;        5
3   // terminates                    // terminates            6
```
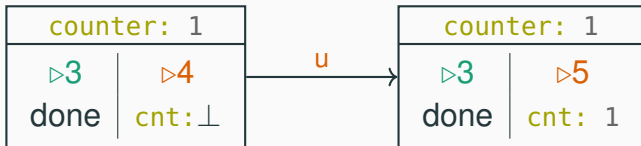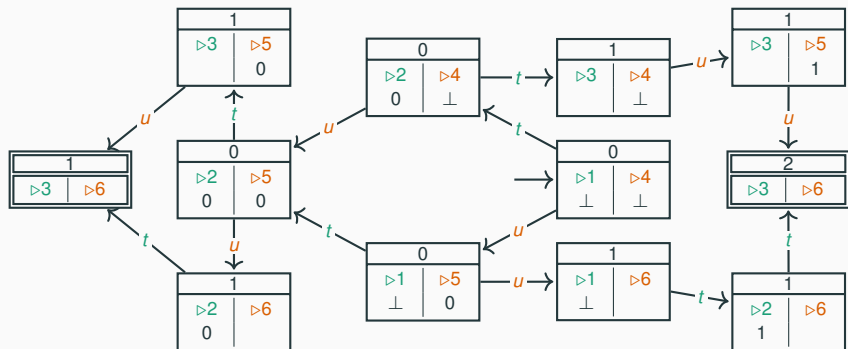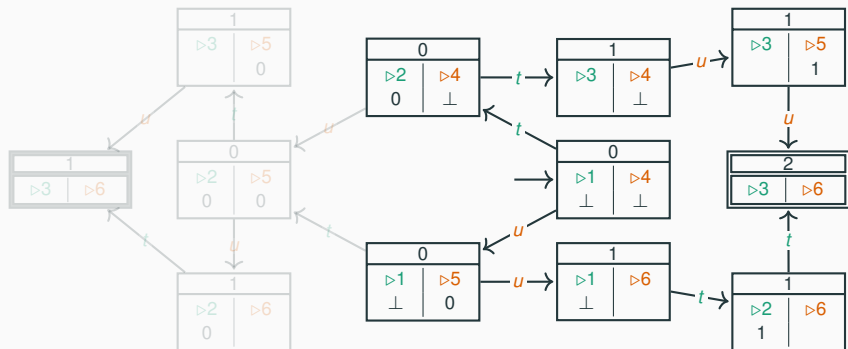
A complete state/transition diagram

The complete state/transition diagram for the concurrent counter example explicitly shows all possible interleavings:

Counter with locks: state/transition diagram

The state/transition diagrams of the concurrent counter example using locks shows the absence of race conditions:

Reasoning about program properties

The structural properties of a diagram capture semantic properties of
the corresponding program:

mutual exclusion: there are no states where two threads are in their
critical region

deadlock freedom: for every (non-final) state, there is an outgoing
transition

starvation freedom: there is no (looping) path such that a thread
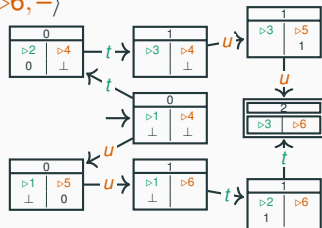never enters its critical region while trying to do so

no race conditions: all the final states have the same result

Building and analyzing state/transition diagrams quickly becomes
tedious. That's where formal verification techniques such as model
checking can help.

Transition tables

Transition tables are <u>equivalent representations</u> of the information of state/transition diagrams.

CURRENT	NEXT WITH t	NEXT WITH u
$\langle 0, \triangleright 1, \perp, \triangleright 4, \perp \rangle$	$\langle 0, \triangleright 2, 0, \triangleright 4, \perp \rangle$	$\langle 0, \triangleright 1, \perp, \triangleright 5, 0 \rangle$
$\langle 0, \triangleright 2, 0, \triangleright 4, \perp \rangle$	$\langle 1, \triangleright 3, -, \triangleright 4, \perp \rangle$	—
$\langle 0, \triangleright 1, \perp, \triangleright 5, 0 \rangle$	—	$\langle 1, \triangleright 1, \perp, \triangleright 6, - \rangle$
$\langle 1, \triangleright 3, -, \triangleright 4, \perp \rangle$	—	$\langle 1, \triangleright 3, -, \triangleright 5, 1 \rangle$
$\langle 1, \triangleright 1, \perp, \triangleright 6, - \rangle$	$\langle 1, \triangleright 2, 1, \triangleright 6, - \rangle$	—
$\langle 1, \triangleright 3, -, \triangleright 5, 1 \rangle$	—	$\langle 2, \triangleright 3, -, \triangleright 6, - \rangle$
$\langle 1, \triangleright 2, 1, \triangleright 6, - \rangle$	$\langle 2, \triangleright 3, -, \triangleright 6, - \rangle$	—
$\langle 2, \triangleright 3, -, \triangleright 6, - \rangle$	—	—

Specification

Writing correct programs

Programming means writing instructions that achieve a certain
functionality. How do we know if a program is correct? And what
does it even mean that a program is correct?

To this end, we distinguish between implementation and specification:

- The implementation is the code that is written, compiled, and
  executed
- The specification is a description of what the program should do,
  usually at a more abstract level than the implementation

Implementation:
```
void withdraw(int amount) {
  balance -= amount;
}
```

Specification:
method withdraw takes a positive
integer amount not exceeding
balance, and decreases balance
by amount

Functional specifications

In sequential programming, we are mainly interested in functional – or input/output – specifications of individual methods. Such specifications consist of two parts:

1. precondition: a constraint that defines the method's valid inputs
2. postcondition: a functional description of the expected output after executing the method

In object-oriented programs, the input and output of a method also include the object state before and after executing the method.

Implementation:

```java
void withdraw(int amount) {
  balance -= amount;
}
```

Specification:

1. precondition:
   `0 < amount && amount <= balance`
2. postcondition:
   "after" `balance ==`
        "before" `balance - amount`

Pre/postconditions in Java

Java does not have support for writing pre/postcondition specifications in the source file.

JML (Java Modeling Language) is a system for annotating Java programs in special comments.

```java
class BankAccount {
  int balance;

  //@ requires 0 < amount && amount <= balance;
  //@ ensures balance == \old(balance) - amount;
  void withdraw(int amount) {
    balance -= amount;
  }
}
```

Invariants

In addition to pre- and postconditions of individual methods, functional specifications include class invariants, which specify properties of the state of objects of that class that should always hold between method calls.

```
class BankAccount {
  int balance;
  invariant { balance >= 0 } // balance never negative
        // (holds if withdraw is called with amount <= balance)
  void withdraw(int amount) {
    balance -= amount;
  }

  void deposit(int amount) {
    balance += amount;
  }
}
```

Specifications of concurrent programs

The specification of concurrent programs should cover two parts:

- a functional specification defines the correct input/output behavior
- a temporal specification defines the absence of undesired behavior, such as no race conditions, deadlock, and starvation

Functional specification techniques such as pre- and postconditions, and class invariants are also applicable to concurrent programs. Class invariants are particularly useful for shared-memory concurrency, where invariants characterize the valid states of shared objects.

Temporal specifications require new notations and techniques.

Temporal logic

Temporal logic was invented by philosophers and later brought to computer science by Pnueli in the 1970s.

Temporal logic is a notation to specify behavior over time. More precisely, it formally defines properties of traces of states, like those that originate from the execution of a (concurrent) program.

Out of the many variants of temporal logic that have been developed, we present the widely used LTL (Linear Temporal Logic).

LTL operators

LTL includes all the usual Boolean operators of propositional logic:

FORMULA	MEANING
p	p is true
$\neg p$	p is not true (i.e., false)
$p \wedge q$	p and q are true
$p \vee q$	p or q is true (or both)
$p \Rightarrow q$	p true implies that q true (if p then q too)

LTL operators

LTL includes all the usual Boolean operators of propositional logic:

FORMULA	MEANING
p	p is true
$\neg p$	p is not true (i.e., false)
$p \wedge q$	p and q are true
$p \vee q$	p or q is true (or both)
$p \Rightarrow q$	p true implies that q true (if p then q too)

In addition, it has a few temporal operators:

FORMULA	MEANING
$\Diamond p$	p is eventually true (from now on)
$\Box p$	p is always true (from now on)
$p \cup q$	p is true (from now on) until q is true
Xp	p is true in the next step

LTL specifications

When we use LTL to specify properties of concurrent programs, propositions (like p and q) represent properties of a program's global state – including shared memory, and threads' local memory and program counters. For example:

PROPOSITION	STATE PROPERTY
c_t	thread t is in its critical section
c_u	thread u is in its critical section
e_t	thread t is trying to enter its critical section
n_t	thread t has terminated

With this convention, we can rigorously specify temporal properties.

LTL specifications: example

In our running example of concurrent increment of counter:

- each thread's critical section is the whole code it executes
- the global state includes: the value of `counter`, the values of the local `cnt`, and the program counter of each thread

<div align="center">

int counter = 0;

</div>

thread t	thread u	
int cnt;	**int** cnt;	
1 cnt = counter;	cnt = counter;	4
2 counter = cnt + 1;	counter = cnt + 1;	5
3 *// terminates*	*// terminates*	6

In our running example of concurrent increment of counter:

- each thread's critical section is the whole code it executes
- the global state includes: the value of counter, the values of the local cnt, and the program counter of each thread

<div align="center">

int counter = 0;

thread t	thread u	
int cnt;	**int** cnt;	
1 cnt = counter;	cnt = counter;	4
2 counter = cnt + 1;	counter = cnt + 1;	5
3 *// terminates*	*// terminates*	6

</div>

PROPOSITION	MEANING
$t \triangleright k$	thread t is at line k
$u \triangleright k$	thread u is at line k

FORMULA	DEFINITION
e_t	$t \triangleright 1$
c_t	$t \triangleright 2$
n_t	$t \triangleright 3$

Mutual exclusion means that no two threads are in the critical section at the same time. For a program with two threads *t* and *u*:

$$\Box \neg (c_t \wedge c_u)$$

"Always (in every state), it is not the case that
both *t* and *u* are in their critical section."

Deadlock freedom in LTL

A deadlock occurs when no thread makes progress; thus deadlock freedom is when some thread makes progress. For a program with two threads t and u:

PROPOSITION	STATE PROPERTY
e_t	thread t is trying to enter its critical section
e_u	thread u is trying to enter its critical section

$$\Box\Big((e_t \wedge e_u) \Rightarrow \Diamond(c_t \vee c_u) \Big)$$

"Always, if both t and u are trying to enter their critical sections,
then t or u will eventually (in some future state)
be inside its critical section."
Or, equivalently: "Not all threads get stuck out forever."

Starvation occurs when one thread does not make progress; thus starvation freedom is when all threads make progress. For a program with two threads *t* and *u*, using the same propositions as before:
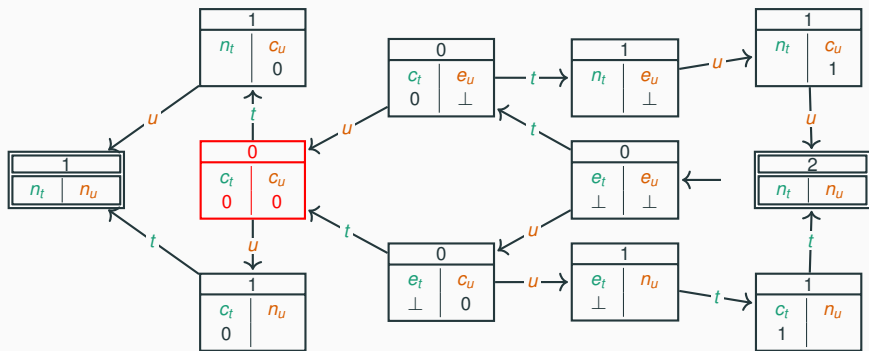
$$\Box\Big(e_t \Rightarrow \Diamond c_t\Big) \;\wedge\; \Box\Big(e_u \Rightarrow \Diamond c_u\Big)$$

"Always, if *t* is trying to enter its critical sections, then *t* will eventually be inside its critical section; and the same holds for *u*."
Or, equivalently: "No threads get stuck out forever."

Mutual exclusion in writing to `counter`: $\Box \neg (c_t \wedge c_u)$, with n_t denoting that t is not in its critical section, and n_u denoting that u is not in its critical section.
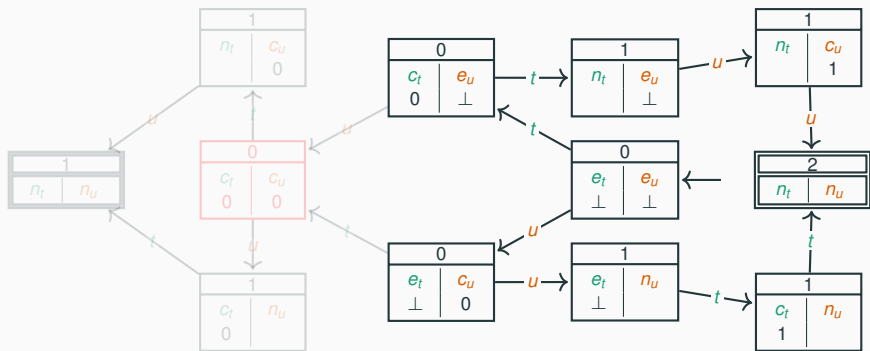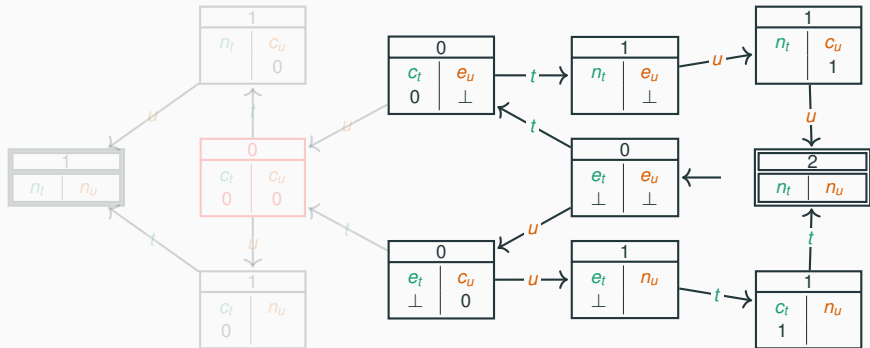
Counter with locks: mutual exclusion

Mutual exclusion in writing to `counter`: $\Box\neg\,(c_t \wedge c_u)$, with n_t denoting that t is not in its critical section, and n_u denoting that u is not in its critical section.

Deadlock freedom: $\square\Big((e_t \wedge e_u) \Rightarrow \Diamond(c_t \vee c_u) \Big)$

Starvation freedom: $\square\Big(e_t \Rightarrow \Diamond c_t \Big) \ \wedge \ \square\Big(e_u \Rightarrow \Diamond c_u \Big)$

Verification

Verification

Verification is the process of checking that a program is correct. This means that, in addition to the implementation, there is also some form of specification (possibly only informal).

Two main techniques to do verification:

- testing: run the program using many different inputs and check that every run satisfies the specification
- formal verification: mathematically prove that every possible run of the program satisfies the specification

Verification

Testing

Testing

Testing in a nutshell:

- run the program using many different inputs
- check that every run satisfies the specification

Method deposit under test:

```java
class BankAccount {
  int balance;

  void deposit(int amount);
  void withdraw(int amount);
}
```

Testing code:

```java
BankAccount ba = new BankAccount();
ba.deposit(100);
check(ba.balance == 100);
ba.deposit(20);
check(ba.balance == 100 + 20);
ba.withdraw(11);
check(ba.balance == 100 + 20 - 11);
// ...
```

Testing is unreliable to find error in concurrent programs because of nondeterminism: a correct run does not guarantee that some other run with the same input will also be correct!

```java
public class Counter
  implements Runnable
{
  // thread's computation:
  public void run() {
    int cnt = counter;
    counter = cnt + 1;
  }
}
```

```java
Counter c = new Counter();
Thread t = new Thread(c);
Thread u = new Thread(c);
t.start();
u.start();
t.join();
u.join();
check(c.count() == 2);
```

↑
sometimes it holds, sometimes it fails!

Testing deadlock freedom?

Besides nondeterminism, there is another problem that occurs if we try to test temporal properties.

- Testing mutual exclusion: if we run the program and detect that two threads are in their critical section at the same time, we know that there is a bug
- Testing deadlock freedom: if we run the program and detect that all threads are blocked for, say, one hour, we still cannot be sure that they will be blocked there forever

In simple examples, setting an arbitrary timeout may be enough, but in large systems with massive workloads it may be hard to figure out how much waiting time is to be expected.

Verification

Model checking

Safety and liveness properties

The difference between properties such as mutual exclusion and deadlock freedom is captured by two classes of temporal properties:

Safety properties are violated by a finite trace:

- informally: "nothing bad ever happens"
- example: <u>mutual exclusion</u> – a trace where, at some given time, two threads are both in their critical section shows that mutual exclusion does not hold

Liveness properties are violated only by an infinite trace:

- informally: "something good eventually happens"
- example: <u>deadlock freedom</u> – a trace where, from some time on, all threads are in the same state forever shows that deadlock freedom does not hold

Formal verification

Since testing is inadequate to reliably verify concurrent programs, formal verification is more widely used even if it is generally more difficult and expensive.

Remember that the specification of concurrent programs consists of two parts: functional and temporal. Verification proceeds as follows:

- first, prove that the temporal specification is always satisfied
- then, assume the temporal specification and prove that the functional specification is always satisfied

The advantages of this approach include:

- verifying a temporal specification alone often is feasible on abstract models of programs which ignore details such as the precise value of all variables
- if a strong temporal specification holds, we can often verify the functional specification as if the program were sequential (because concurrent executions are free from race conditions!)

Formal verification of the counter

Verifying the concurrent counter:

- to prove mutual exclusion, we only analyze the locking behavior and ignore the exact value of `counter`
- if mutual exclusion holds, the two threads execute `run` sequentially, thus we analyze the program as if it were sequential

```java
public class Counter
  implements Runnable
{
  // thread's computation:
  public void run() {
    lock();
    int cnt = counter;
    counter = cnt + 1;
    unlock();
  }
}
```

```java
Counter c = new Counter();
Thread t = new Thread(c);
Thread u = new Thread(c);
t.start();
u.start();
t.join();
u.join();
check(c.count() == 2);
```

Model checking

Model checking is an effective technique to verify concurrent programs, first developed in the 1980s.

Model checking mainly targets the verification of temporal specifications – expressed in temporal logic – about the behavior of state/transition diagrams – also called transition systems or finite-state automata:

1. given a concurrent program, build a state/transition diagram using finitely many states that captures its concurrent behavior

2. model checking algorithms analyze all infinitely many traces of the state/transition diagram and check whether a given temporal logic specification holds:

   - if model checking is successful, we have verified that all executions of the program satisfy the temporal specification
   - if model checking is unsuccessful, it returns a counterexample – a concrete trace that shows that the temporal specification is violated

Model checking in practice

Building a state/transition diagram that correctly captures the behavior of a concurrent program is something that cannot always be done automatically. Model checking tools provide convenient languages to formalize concisely complex state/transition diagrams.

For example, this is a model of the concurrent behavior of the shared counter in ProMeLa – the input language of the Spin model checker:

```
int count = 0;

proctype IncThread() {
  int tmp;  tmp = count;  count = tmp + 1;
}

init { // spawn two threads running in parallel
  run IncThread(); run IncThread();
}
```

Model checking techniques and tools

There are two large families of model-checking techniques and tools:

- Explicit-state model checking works by explicitly exploring the state space generated by a given state/transition diagram. Spin is the most popular explicit-state model checker.
- Symbolic model checking works by encoding a given state/transition diagram using logic formulas (or other specialized data structures), and then expressing the temporal properties as logic properties of the encoding. NuSMV is a state-of-the-art symbolic model checker.

To know more about model checking: course "Formal methods in software development"

These slides' license

© 2016–2018 Carlo A. Furia