SWEN90004
Modelling Complex Software Systems

Model checking with temporal logic

Nic Geard

Lecture Con.10

Semester 1, 2019

Introduction

In lectures the last two lectures, we saw how to specify properties that hold true for every execution of a concurrent system. These safety and liveness properties are useful for describing certain attributes of systems. However, they are not as powerful as logical properties.

A logical property is a description composed of propositions, and logical operators, such as and, not, and or. In previous subjects, you will have already encountered propositional logic, which allows us to describe properties about the state of a system at a given instant in time. However, FSP models have no state. Instead, they consist of actions that occur in time.

LTL

Here we go beyond propositional logic and safety/liveness properties to linear temporal logic (LTL).

LTL predicates give us far greater flexibility than either propositional logic or safety/liveness properties, and allow us to specify more intricate system properties, which can then be checked using LTSA.

Propositional and predicate logic

We assume you are comfortable with propositional and first-order predicate logic.

We recommend chapter 8 (and some of 9) from Makinson, *Sets, Logic and Maths for Computing*, Springer 2008. An online version is available through the University Library.

Another electronic resource is the Magnus, *Forall X: An Introduction to Formal Logic*, `http://www.fecundity.com/logic/`. Aim to understand the first four chapters.

Logic for Actions

Atomic propositions in logic are concerned with the state of a system. In software analysis, this is usually basic propositions such as `i >= 0` and `i < j`.

However, in models of concurrency, typically we are concerned with actions, and the order in which they occur. We can adopt the propositional logic idea and say that the proposition `a` is true when the action `a` executes, and false at all other times. The problem is that only one proposition is true at a time.

Really, we are interested in properties about sequences of actions over time, not just one instant of time.

Temporal logics allow us to talk about durations and the relative times of events (as opposed to absolute time).

FSP – Fluents

A fluent is a property that can change over time. Fluents allow us to describe properties of a system over its lifetime, rather than at just one instant. Fluents are used heavily in logic, especially artificial intelligence.

In FSP, a fluent is described as:

$$\texttt{fluent FL = <\{s1,...,sN\}, \{e1,...,eN\}>}$$

in which $\texttt{s1,...,sN}$ and $\texttt{e1,...,eN}$ are actions.

The fluent $\texttt{FL}$ is the proposition, and $\texttt{FL}$ becomes true when any of the actions in $\{\texttt{s1,...,sN}\}$ occur, and then false again when any of the actions in $\{\texttt{e1,...,eN}\}$ occur. $\texttt{FL}$ is initially false.

FSP – Fluents

If we want a fluent that is initially true, we can describe it using:

```
fluent FL = <{s1,...,sN}, {e1,...,eN}> initially 1
```

in which `1` represents *true*. `0` represents *false*.

As an example, consider the traffic light system from Lecture Con.05. We can say when we expect the light to be green:

```
fluent GREEN = <{green}, {yellow,red}> initially 1
```

That is, the fluent `GREEN` becomes true when the `green` action occurs, and false when `yellow` or `red` occur. The fluent is therefore not an action: it remains true when actions other than `yellow` or `red` occur (say, actions `wander` or `none`), even though the action `green` is not currently "true".

FSP – Indexed fluents

As with processes, we can define indexed fluents for two lights:

```
fluent GREEN[i:1..2]
  = <{green[i]}, {yellow[i],red[i]}> initially 1
```

FSP – Fluent expressions

Fluents can be combined using the propositional logic connectives:

FSP	Logic	Meaning	
`&&`	$\wedge$	conjunction	(and)
`||`	$\vee$	disjunction	(or)
`!`	$\neg$	negation	(not)
`->`	$\rightarrow$	implication	(`A->B` $\equiv$ `!A || B`)
`<->`	$\leftrightarrow$	equivalence	(`A<->B` $\equiv$ `(A->B)&&(B->A)`)

In addition, we have (bounded) universal and existential quantifiers:

```
forall[i:1..2] GREEN[i]
exists[i:1..2] GREEN[i]
```

FSP – Fluent expressions

Now we can express that we do not want the two lights to be green at the same time:

```
!forall[i:1..2] GREEN[i]
```

which is equivalent to

```
!(GREEN[1] && GREEN[2])
```

and is also equivalent to:

```
exists[i:1..2] !GREEN[i]
```

Temporal logic – “always” and “eventually”

So far we specified properties of a single point in time. Their truth/falsehood depended on the trace up to that point.

To express properties with respect to an entire timeline, we have temporal operators.

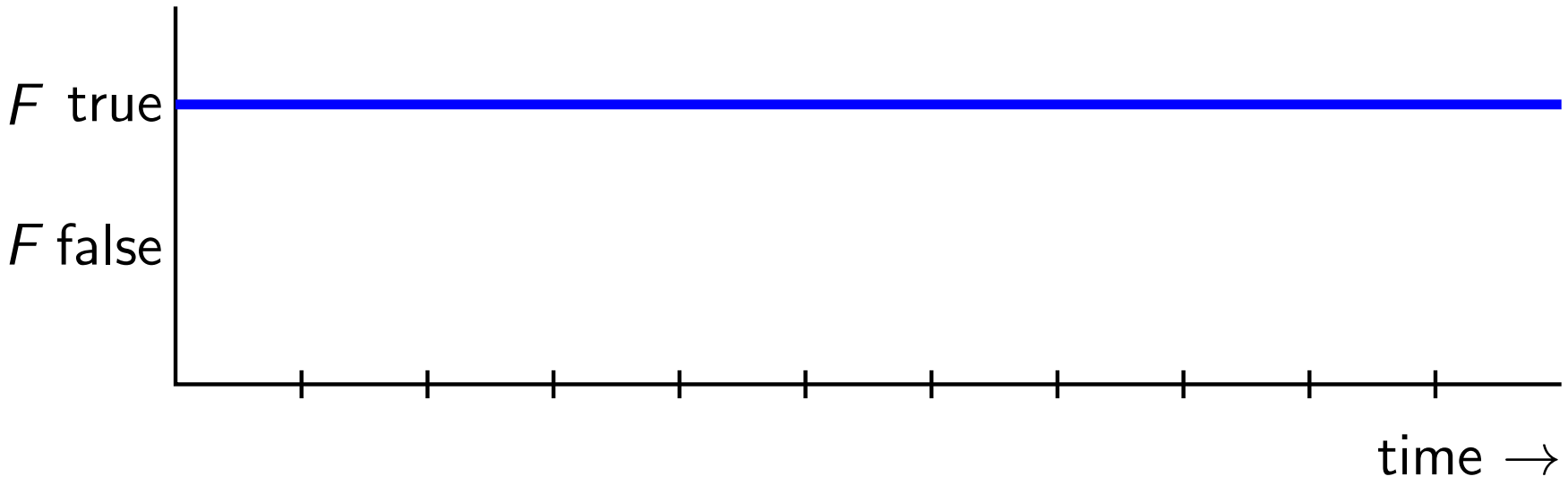
Temporal operators allow us to specify properties about all traces in our model, and thus of our model itself.

The two most basic operators are always and eventually.

Temporal logic – “always”

The linear temporal logic formula $[]F$ *(read always* F*) is true iff the formula* F *is true at the current instant and at every instant in the future.*

In the literature, the “always” operator is a square: $\Box F$, but [] is a close approximation in typewriter font.



Temporal logic – “always”

Let us define some fluents for our traffic light:

```
fluent GREEN  = <{green}, {yellow,red}> initially 1
fluent YELLOW = <{yellow}, {green,red}> initially 0
fluent RED    = <{red}, {yellow,green}> initially 0
```

Now to say that the light is always green, yellow, or red:

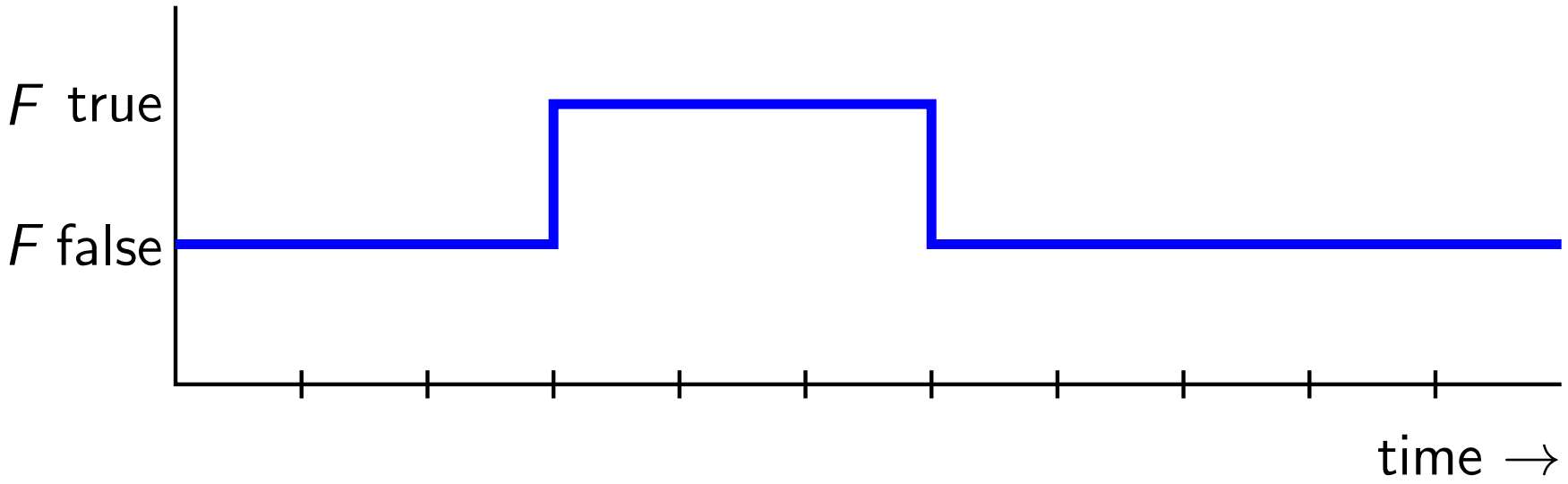
```
assert ALWAYS_A_COLOUR = [](GREEN || YELLOW || RED)
```

This shows the difference between actions and fluents. The property `[](green||yellow||red)` does not hold for the traffic light system, because `button` and `none` can occur. At those points in time, none of `green`, `yellow`, or `red` hold, but one of our fluents will.

Temporal logic – “eventually”

The linear temporal logic formula <>F (read eventually F) is true iff the formula F is true at the current instant or at some instant in the future.

In the literature, this operator is a diamond: $\Diamond F$, but <> is a close approximation in typewriter font.



Temporal logic – “eventually”

This fluent says that the light will eventually become red:

```
assert EVENTUALLY_RED = <>RED
```

To check these properties using LTSA, select
Check → `LTL Property` → *ALWAYS_A_COLOUR*
(or whatever property you want to choose).

FSP – Safety and liveness

The always operator is used to describe safety properties, while the eventually operator is used to describe liveness properties.

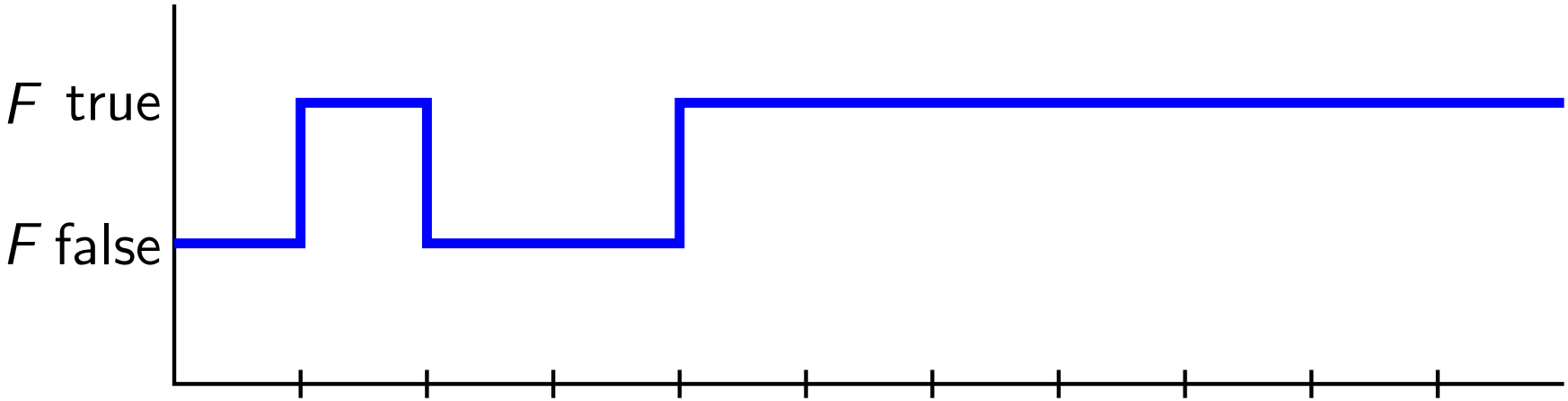
However, these operators are not the same as using the `property` keyword, as we did in Lectures Con.08 and Con.09. Temporal operators offer far more flexibility. For example, consider this terminating system:

```
A = (a -> b -> END | c -> b -> END).
```

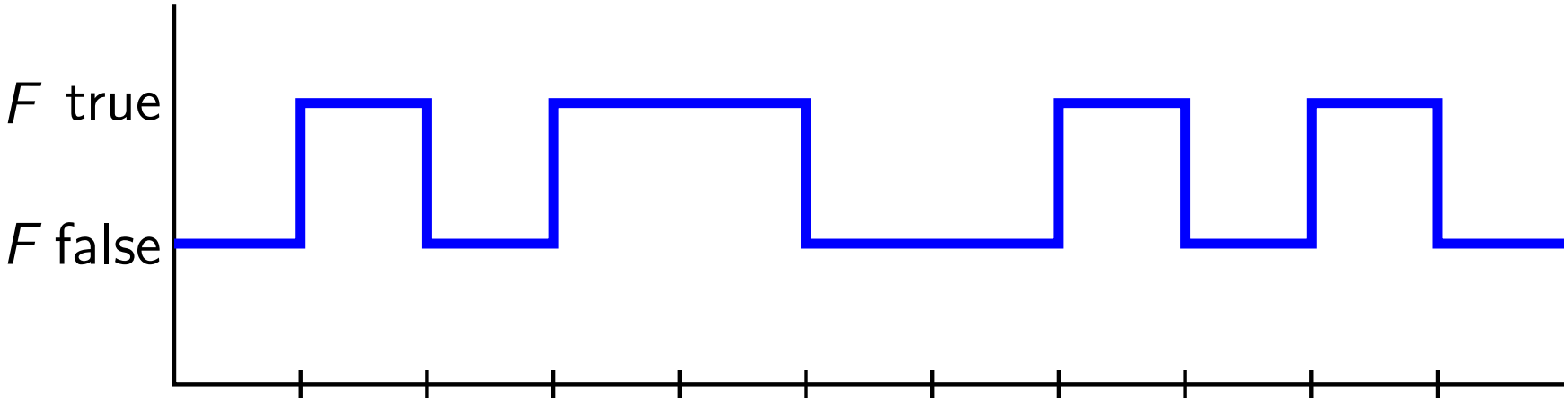
We can specify the property that `b` will eventually occur using `assert B = <>b`. Using LTSA, this will report no violations. However, `b` does not occur infinitely often, which is what a progress property specifies. With LTL we can express that `b` eventually occurs, even if it is only once.

Combining temporal operators

Note that $\Diamond F$ says that F will become true, but not that F remains true from some point on. For that, we need $\Diamond\Box F$:

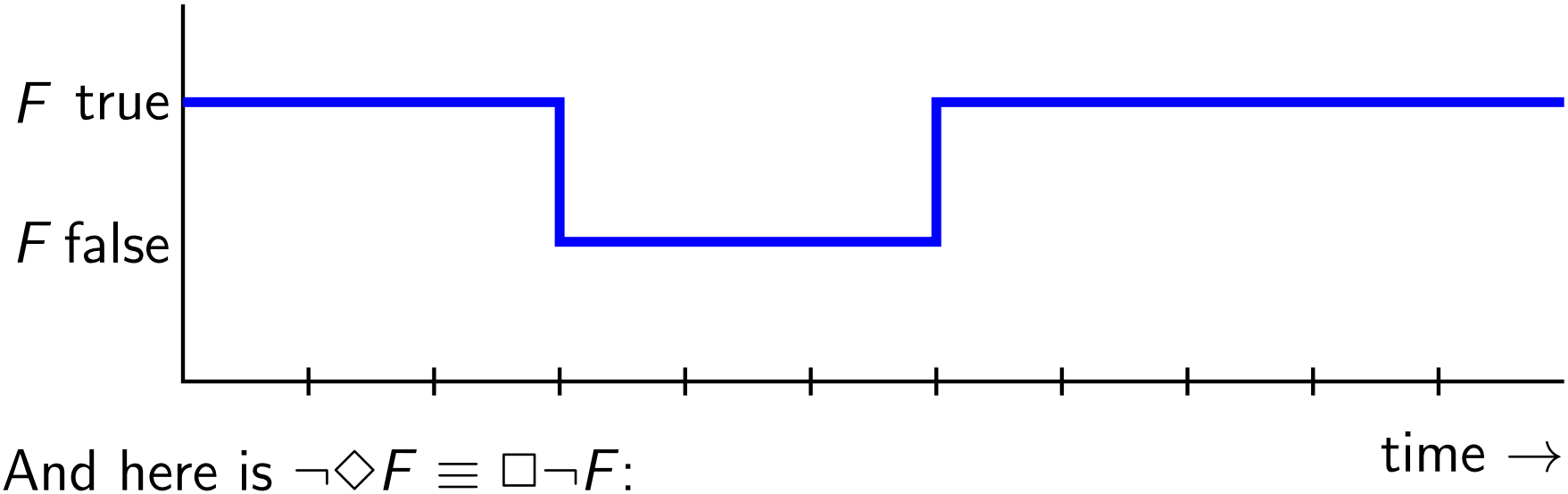


This is very different from $\Box\Diamond F$:

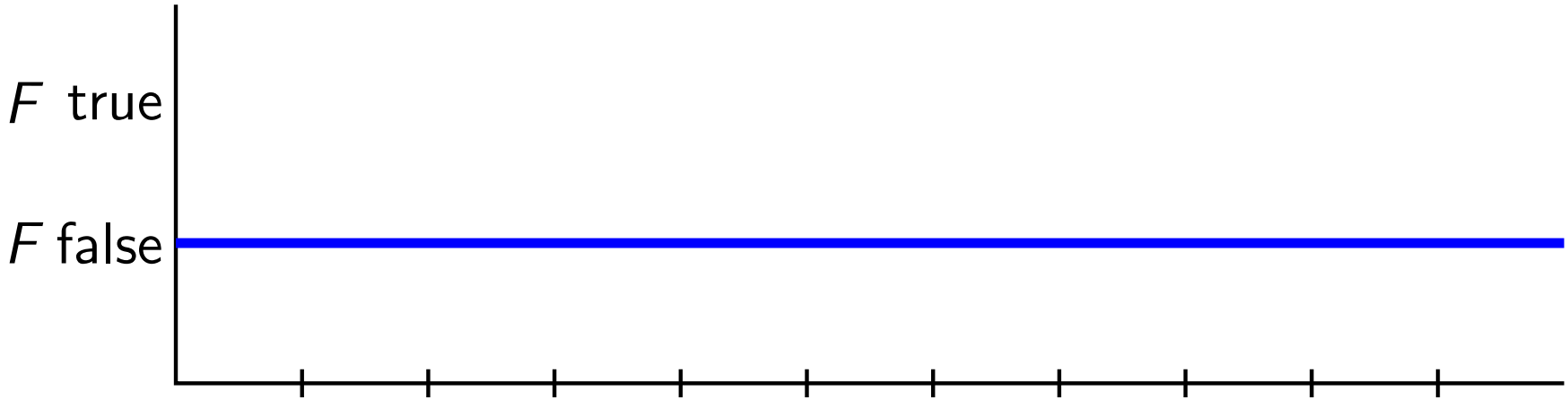


Temporal logic laws

It should be clear that $\Box$ and $\Diamond$ are dual operators, just like universal and existential quantification. Here is $\neg\Box F \equiv \Diamond\neg F$:



And here is $\neg\Diamond F \equiv \Box\neg F$:



Temporal logic expressiveness

The temporal operators can be combined arbitrarily with the usual connectives. This allows us to express intricate properties of a concurrent system.

Suppose process P and process Q run concurrently, vying for some resource. Say that p is the action that P performs when it is in its critical section (having the resource), and q is what Q performs in its critical section.

Then $p \rightarrow \Diamond q$ means that if P is currently in its critical section then Q will eventually get to be in its critical section.

Hence $\Box(p \rightarrow \Diamond q)$ expresses that Q will not be starved.

Temporal logic – the "until" operator

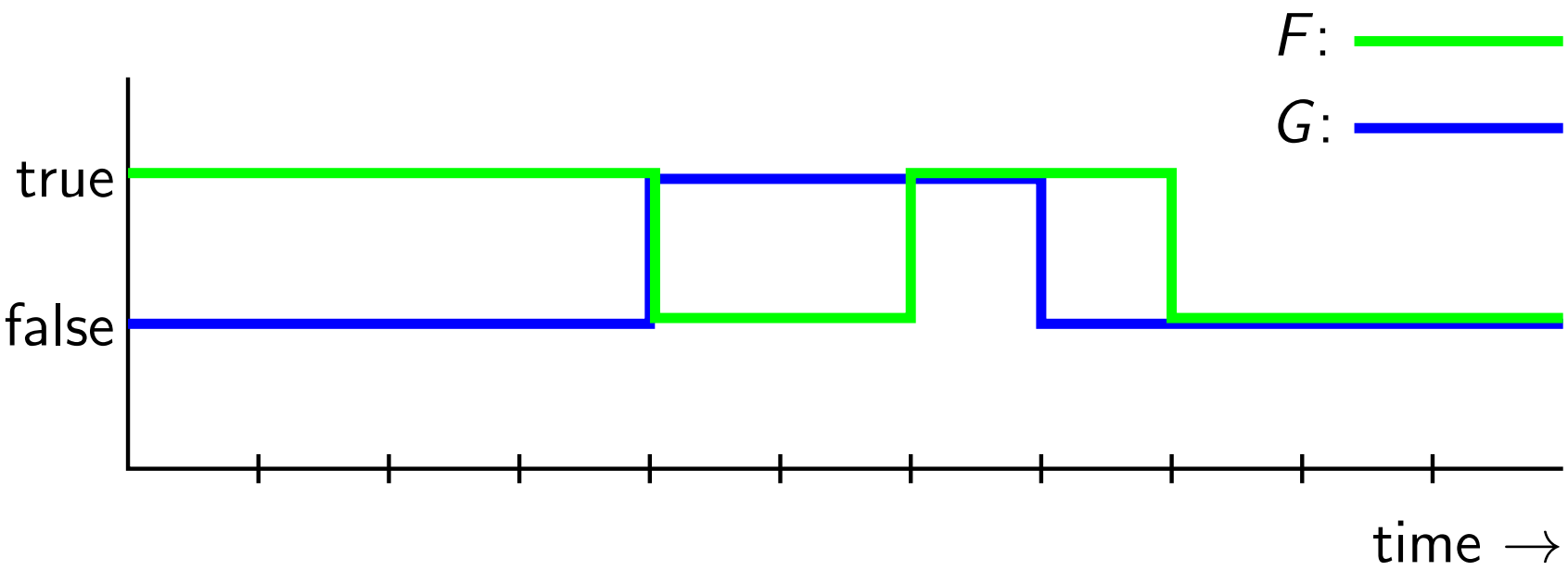
Still, the "always" and "eventually" operators have limited expressive power in that they are monadic—each is applied to a single formula.

Often correctness specifications involve relating several propositions. We may want to say that P enters its critical section at most once before Q enters its critical section.

The until operator allows us to specify that a certain property is true until another property becomes true.

Temporal logic – the "until" operator

F $\mathtt{U}$ G is true iff G eventually becomes true, and F is true until that instant.



Temporal logic – the "until" operator

In our traffic light system, the light is initially green, and stays green until the button is pushed:

```
assert INITIALLY_GREEN = (GREEN U button)
```

It is important to note that, once the button is pushed, the light can remain green while still preserving this property. Nothing in the assertion says that `green` will become false. Once the button is pushed, the truth of `GREEN` (and in fact, `INITIALLY_GREEN`) is no longer of interest.

Temporal logic – the "next" operator

The next operator allows us to specify that a certain property is true at the next instant:

> *The linear temporal logic formula* `X` *F is true iff F is true at the next instant.*

By "next instant", we mean when the next action occurs.

As an example, we can specify that when the button is pushed, the light will go yellow at the next time instant:

```
assert BUTTON_TO_YELLOW = (button -> X YELLOW)
```

Temporal logic – the "next" operator

```
assert BUTTON_TO_YELLOW = (button -> X YELLOW)
```

Here we could just as easily replace the fluent `YELLOW` with the action `yellow` and the result would be the same. This is because the fluent `YELLOW` becomes true when `yellow` occurs. It remains true until the `green` or `red` actions occur, but in this case, we care only about the next action after `button`, so the longevity of `YELLOW` is irrelevant.

Mutual exclusion revisited

Let us again consider the modelling of mutual exclusion using semaphores from Lecture Con.08.

Recall we had N processes who wanted to `enter` then `exit` a critical region:

```
LOOP
 = (mutex.down -> enter -> exit -> mutex.up -> LOOP).

||N_LOOPS
  = (  p[1..N]:LOOP
    || {p[1..N]}::mutex:SEMAPHORE(1)
    ).
```

where `up` and `down` were the semaphore operations.

Mutual exclusion revisited

First, we define a fluent that is true when a process is in its critical section, and define this for all processes. A process is in its critical section when it calls `enter`, and jumps out when it calls `exit`:

```
fluent IN_CRITICAL[i:1..N]
  = <{p[i].enter}, {p[i].exit}>
```

We want to specify two important properties: a safety property that states that only one process can enter the critical section at a time; and a liveness property that says no process starves.

Expressing Mutex safety and liveness

```
//only one thread in its critical section at one time
assert MUTEX_N
  = []!(exists [i:1..N-1]
          (IN_CRITICAL[i] && IN_CRITICAL[i+1..N]))

//all processes eventually enter the critical section
assert EVENTUALLY_ENTER
  = forall[i:1..N] <>p[i].enter
```

The first property states that it is always the case that there is no process `i`, such that that process is in its critical section, and another process is also in its critical section.

Expressing Mutex safety and liveness

These are the two most important properties for mutual exclusion. However, we may also want to verify certain other properties.

The following specifies that no process enters its critical section before locking the mutex:

```
assert NO_ENTER_BEFORE_MUTEX
  = forall[i:1..N]
      (!IN_CRITICAL[i] U p[i].mutex.down)
```

Expressing Mutex safety and liveness

We may also specify that, when a process locks the mutex, it will be in its critical section at the next tick:

```
assert LOCKED_OUT
  = forall[i:1..N]
      (p[i].mutex.down -> X IN_CRITICAL[i])
```

What this really says is that, when a process locks the mutex, no other process can do anything related to the mutex: the very next action is the locking process entering its critical section.

Expressing Mutex safety and liveness

Finally, we may want the following response property, which states that, if a thread enters the critical section, it must eventually exit:

```
assert MUST_EXIT
  = forall[i:1..N]
      [](p[i].enter -> <>p[i].exit)
```

These properties all hold on the original model.

But again, if we can change the semaphore from a binary semaphore to one that allows values 0, 1 and 2, we experience violation of the `MUTEX_N` and `LOCKED_OUT` properties.

Expressing Mutex safety and liveness

One nice thing about LTSA is the report it gives for LTL properties. If we check the `MUTEX_N` property for the 3-valued semaphore, LTSA reports:

```
Trace to property violation in MUTEX_N:
    p.1.mutex.down
    p.1.enter           IN_CRITICAL.1
    p.2.mutex.down      IN_CRITICAL.1
    p.2.enter           IN_CRITICAL.1 && IN_CRITICAL.2
```

On the left is the trace, as seen earlier. On the right are the related fluents that are true. After `p.1.enter`, fluent `IN_CRITICAL.1` becomes true. After `p.2.enter`, fluents `IN_CRITICAL.1` and `IN_CRITICAL.2` are both true; that is, process 1 and 2 are in the critical section.

Expressing Mutex safety and liveness

To save us having to check each property individually, we could create a proposition conjoining all the desired properties:

```
assert ALL_PROPERTIES
  = (  EVENTUALLY_ENTER
    && MUTEX_N
    && NO_ENTER_BEFORE_MUTEX
    && LOCKED_OUT
    && MUST_EXIT
    )
```

Now, we just need to check `ALL_PROPERTIES`.

If a violation is found, then we need to check each property individually, to find out which is violated, but as a form of regression testing, this approach is useful.

Model checking in the real world

Model checking specifications (written in temporal logic variants) has become an important tool in practice, used to prove the correctness of concurrent systems, communication protocols, and hardware aspects such as cache coherence.

A major breakthrough came around 1990 when Ken McMillan found a way of using binary decision diagrams and associated powerful algorithms to perform what he called symbolic model checking. Suddenly it became possible to reason automatically about systems with very large numbers of states, often in the order of 10^{20} or more.

McMillan used his technology to verify the correctness of the cache protocol of the Encore Gigamax multiprocessor. Random simulation is not effective for this, but McMillan was able to track down a potential processor deadlock and fix it. Since then, many other circuits and protocols have been verified using model checking.

References / Recommended reading

- J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*, 2nd edition, John Wiley and Sons, 2006. Available at `http://flylib.com/books/en/2.752.1.1/1/`
- Dov M. Gabbay, A. Kurucz, F. Wolter, M. Zakharyaschev, *Many-Dimensional Modal Logics: Theory and Applications*, Elsevier, 2003.
- K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publ., 1993.