pthread_create()

- Prototype:
```
int pthread_create( pthread_t *
                  , const pthread_attr_t *
                  , void *(*)(void *)
                  , void * );
```

- Call:
```
rc = pthread_create(&thread_data,NULL,ThreadCode,threadarg);
```

- Precondition: none relevant to this course... (resource availability, permissions,..)

- Postcondition: `thread_data` contains the pthread ID. `ThreadCode(threadarg)` is ready to execute.

- Invariant: none obvious but note that `ThreadCode()` may execute some of its code before `pthread_create()` returns to its caller.

pthread_exit()

- Prototype: `void pthread_exit( void * );`
- Call: `pthread_exit( NULL );`
- Precondition: Formally, none, but it's a bad idea for the calling thread to own any mutex locks.
- Postcondition: terminates the calling thread, so nothing is observed "afterwards" in the caller. In the surrounding pthread environment, any mutex locks it holds are still held afterwards. Values of variables local to the calling thread are undefined.
- Invariant: thread code that ends because the `ThreadCode` function ends normally, performs an implicit `pthread_exit(NULL);` call.

pthread_join()

- Prototype:

```
int pthread_join( pthread_t
                , void ** );
```

- Call:

```
rc = pthread_join(thread_data,NULL);
```

- Precondition: `pthread_data` refers to a existing thread; there is no other live call to `pthread_join()` with the same `pthread_data` value.
- Postcondition: The thread identified by `thread_data` has terminated.
- Invariant: `pthread_join()` does not return to the caller until the thread identified by `thread_data` has terminated; `pthread_join()` waits for termination, and does not force it.

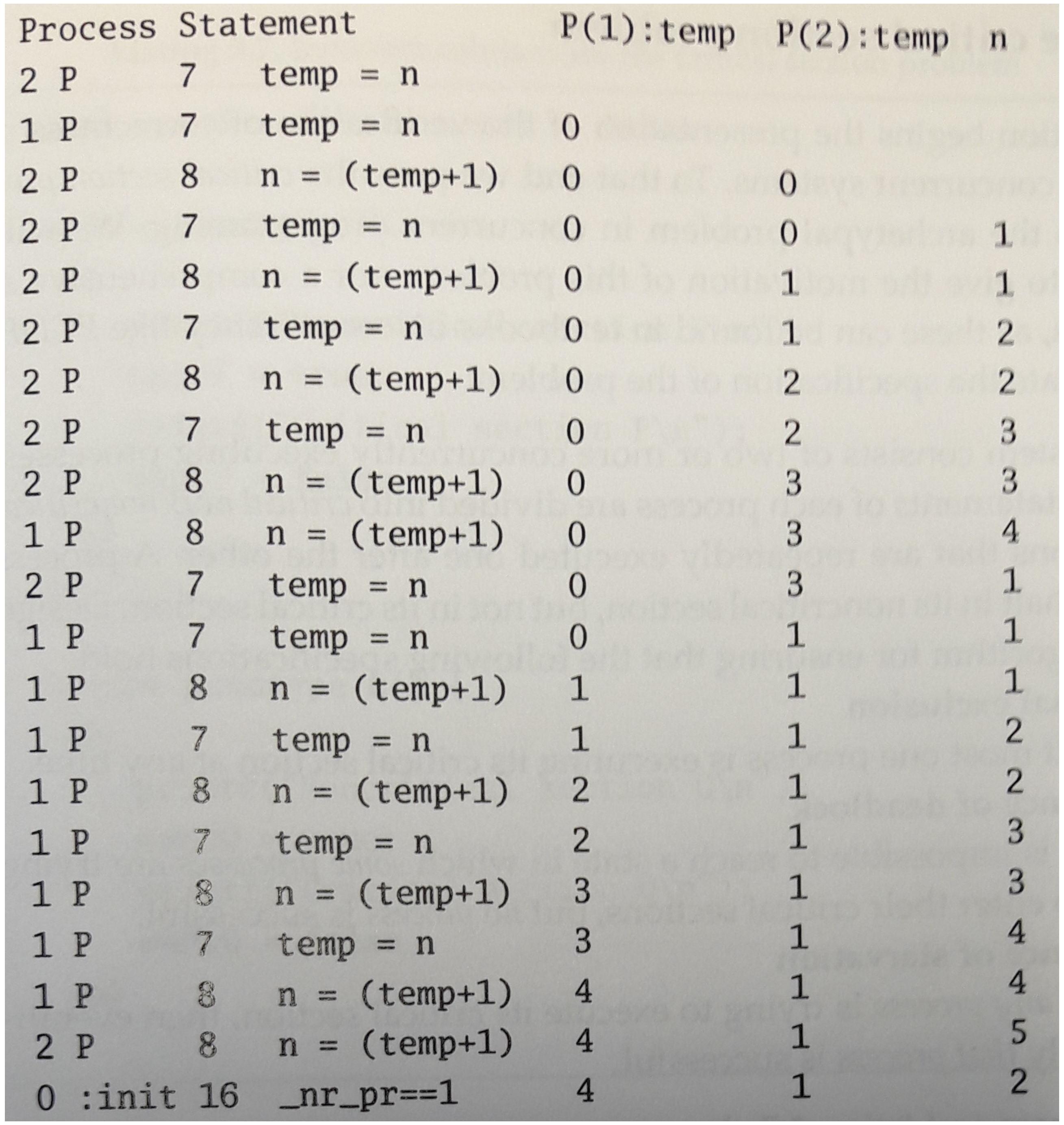
Concurrent Counting Algorithm

Example: Concurrent Counting Algorithm	
integer n ← 0;	
p	**q**
integer temp	integer temp
p1: do 10 times	q1: do 10 times
p2: temp ← n	q2: temp ← n
p3: n ← temp + 1	q3: n ← temp + 1

- Increments a global variable *n* 20 times, thus *n* should be 20 after execution.
- But, the program is faulty.
    - Proof: construct a scenario where *n is 2* afterwards.
- Wouldn't it be nice to get a program to do this analysis?

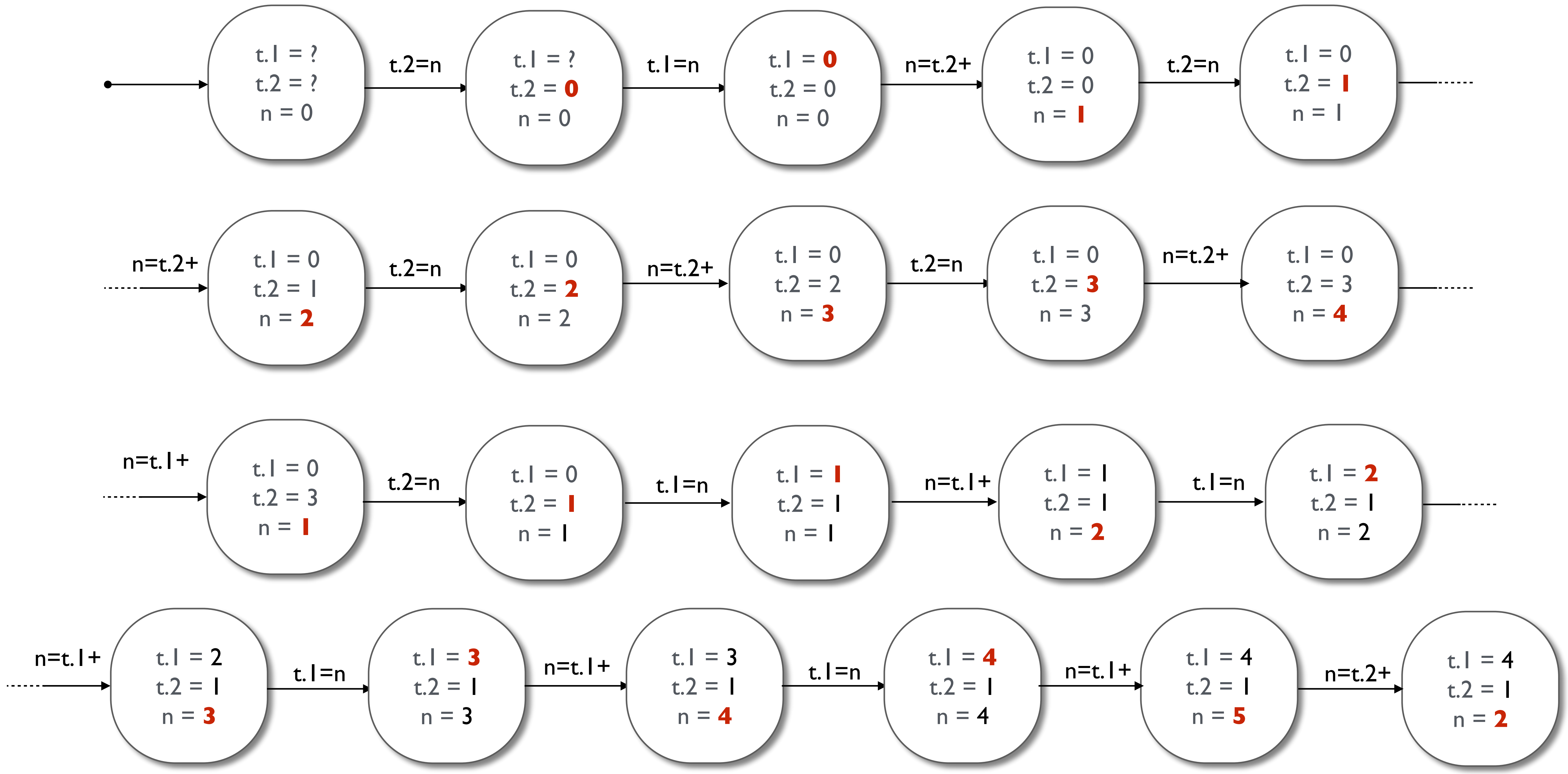
????

- Discovered by M. Ben-Ari during his concurrency course
    - Student puzzled him by observing a sum equal to 9
    - He modelled it and found it could be as low as 2, but no lower
    - On the right, running Promela
        - with a loop of length 5 rather than 10
        - a final assertion that n>2
        - This is the counterexample resulting in not(n>2), i.e., n=2.



```
Process Statement          P(1):temp  P(2):temp  n
2 P      7   temp = n
1 P      7   temp = n      0
2 P      8   n = (temp+1)  0          0
2 P      7   temp = n      0          0          1
2 P      8   n = (temp+1)  0          1          1
2 P      7   temp = n      0          1          2
2 P      8   n = (temp+1)  0          2          2
2 P      7   temp = n      0          2          3
2 P      8   n = (temp+1)  0          3          3
1 P      8   n = (temp+1)  0          3          4
2 P      7   temp = n      0          3          1
1 P      7   temp = n      0          1          1
1 P      8   n = (temp+1)  1          1          1
1 P      7   temp = n      1          1          2
1 P      8   n = (temp+1)  2          1          2
1 P      7   temp = n      2          1          3
1 P      8   n = (temp+1)  3          1          3
1 P      7   temp = n      3          1          4
1 P      8   n = (temp+1)  4          1          4
2 P      8   n = (temp+1)  4          1          5
0 :init 16   _nr_pr==1     4          1          2
```

n is 2 scenario



Correctness, fairness

- We can use state diagram descriptions of concurrent computations to explore whether a concurrent program is *correct* according to some criterion, or whether it is *fair*.

Correctness

- (Correctness in Sequential Programs)
  - Partial Correctness: The answer is correct if the program halts
  - Total Correctness: The program does halt and the answer is correct.
- Correctness in Concurrent Programs
  - Safety Property: a property must be *always* true
  - Liveness Property: a property must be *eventually* true

Using State machines

- We can describe concurrent programs using states and state transitions.
- We can construct a state transition diagram (a *"model"*) which captures every possible scenario a concurrent program is capable of executing.
- By reasoning with the model (*"model checking"*), we can prove whether or not a particular property holds for a concurrent program.
- Much more powerful idea than merely testing a concurrent program.

Weak Fairness

- Recall a concurrent program is modelled as the interleaving of statements in any possible way in a number of sequential programs.
  - If a statement is ready for execution, then if it is *guaranteed* that it will *eventually* be executed (i.e. it will appear in a scenario), the system is said to be *weakly fair*.
- Another way of thinking about this is that in a weakly fair system, the arbitrary interleaving of statements does not include scenarios in which available and ready statements will never be executed.
- Sometimes we assume weak fairness for something to work.

Verification of concurrent programs

- Manual construction and inspection of state diagrams
  - Only suitable for the most trivial programs with few states
- Automated construction and inspection of state diagrams
  - The task of constructing the state diagram and using it to check the correctness of the concurrent program can be automated using a *model checker*
  - We still need a way to specify models and logical correctness properties
- Formal specification and verification using temporal logic

Linear Temporal Logic

- Logic that reasons about state-change over time
    - Allows us to write properties that apply to an entire scenario, i.e., sequences of states
- LTL is the simplest of a large family of temporal (and "modal") logics
    - Also often used to reason about computing systems are CTL (Computational Tree Logic) and CTL*
- LTL is an extension of standard propositional ("digital") logic with temporal operators
- We use the notation of the SPIN model-checker* here
    - there are more mathematical forms in the literature
    - LTL text notation is not standard - Wikipedia has a common but different variant.

* coming soon!

Propositional Logic in LTL

- LTL contains propositional logic with all the usual parts:
  - Logical constants: `true`, `false`
    - Also numeric constants: 0, 1, 2,..
  - Variables: `a`, `b`, .. p, q, `xx`, `yY`, ... - they must start with lowercase letter
    - Also boolean-valued expressions over variables and constants
  - Propositional Operators:
    - Negation: `!`   Logical-and: `&&` (also `/\`)   Logical-or: `||` (also \/)
    - Logical Implication: `->`
    - Logical Equivalence: `<->`
- We also assume that have numeric constants, and operators, and comparisons

State(s) in LTL

- A state is defined by the values of a given set of variables.
  - propositional expressions evaluate to true or false based on those values in a given ***single*** state
- LTL in general is interested in sequences (a.k.a. paths) of states.
  - In general, an LTL expression is deemed to be true of a *given starting* state in such a path:
    - if it holds true for the path from that state onwards.
- We will denote such a path as a sequence of indexed state:s
  - $\mathbf{S}_0$, $\mathbf{S}_1$, $\mathbf{S}_2$, ..., $\mathbf{S}_{i-1}$, $\mathbf{S}_i$, $\mathbf{S}_{i+1}$, ...
- A temporal property is true "at" state $\mathbf{S}_i$ if it is true for the path starting with $\mathbf{S}_i$.

Linear Temporal Operators (Until)

- For most applications, all we need to do is to define one temporal operator ("until")
  - The rest of the operators can be expressed in terms of this
- LTL starts with a notion of "weak until",
  which says that (`p w-until q`) is true at starting state s_i if
  - `q` is true at s_i, or
  - `p` is true at s_i and (`p w-until q`) is true at s_{i+1}
  - Note that there is no requirement for `q` to ever be true, but if it is never so, then `p` must always be true
- "Strong until" (`s-until`) is "weak until" with an additional requirement that q must become true after a finite number of steps.
- SPIN uses the notation `U` to denote strong until - it does not have the weak operator.

Linear Temporal Operators (derived)

- We can now derive two other useful operators
  - one using "weak-until", the other using "strong-until"
- Always p (`[]p`) is true if p is true in every state in the path
  - It can be defined using weak-until as `[]p = p w-until false.`
- Eventually p (`<>p`) is true if p is true at least once, somewhere along path, after a finite number of steps
  - It is defined using strong-until as `<>p = true U p`

Always

- Always p (`[ ]p`) is true at S_i if p is true in every state in the path from S_i onwards
  - It can be defined using weak-until as `[ ]p = p w-until false.`
- (`p w-until false`) is true at starting state S_i if
  - `false` is true at S_i, (`false` is never true anywhere, anytime!) or
  - `p` is true at S_i and (`p w-until false`) is true at S_{i+1}
  - Note that there is no requirement for `false` to ever be true, but if it is never so, then `p` must always be true
- So `p` must always be true!

Eventually

- Eventually q (`<>q`) is true at state **S**i if q is true at least once, somewhere along the path from **S**i, after a finite number of steps
  - It is defined using strong-until as `<>q = true U q`
- which says that (`true U q`) is true at starting state **S**i if
  - q is true at **S**i, or
  - `true` is true at **S**i (it always is!) and (`true U q`) is true at **S**i+1
  - q must be true after a finite number of steps along the path.
- So q must be true after a finite number of steps.

predicates without temporal operators

- We refer to a predicate without temporal operator as a "state predicate"
  - It is a predicate using constants, expressions, and the proposition operators.
- State predicate p is true "at **S**i" if it is true of **S**i .
  - There is no reference to any subsequent states.

Nested temporal operators

- The ps and qs used previously to talk about arguments to operators like "until", "always", "eventually" can themselves be built using such operators: [](\[]p) <>(\[]P -> <>Q)
- We can define a systematic set of rules that can always determine if a given path of states is satisfied by a given temporal logic predicate.
  - These rules apply recursively
  - They are easy to automate

Common LTL Predicates

LTL	Reads as...	Property
`[]p`	always p	invariance
`<>p`	eventually p	guarantee
`p -> <>q`	p implies eventually q	response
`p -> q U r`	p implies q until r	precedence
`[]<>p`	always eventually p	recurrence (progress)
`<>[]p`	eventually always p	stability (non-progress)
`<>p -> <>q`	eventually p implies eventually q	correlation

("The Spin Model-Checker", p137)