Concurrent Systems

3D4 ⟷ CS2016

Operating Systems

Andrew Butterfield
ORI.G39, Andrew.Butterfield@scss.tcd.ie

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

with thanks to Mike Brady

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 SPIN LTL notation here
  - there are more mathematical forms in the literature
  - LTL notation text notation is not standard - Wikipedia has a common but different variant.

Propositional Logic in LTL

- LTL contains propositional logic with all the usual parts:
  - Logical constants: `true`, `false`
  - Variables: a, b, .. p, q, `xx`, `yY`, ... - they must start with lowercase letter
  - Negation: `!` Logical-and: `&&` (also `/\`) Logical-or: `||` (also `\/`)
  - Logical Implication: `->`
  - Logical Equivalence: `<->`

`#define` vs. `inline`

- SPIN uses the C pre-processor (CPP) to process Promela files
  - So all the CPP facilities are available, such as #define, #if, #ifdef, etc.
- In the producer-consumer example (`prodcons2.pml`), we used something similar called `inline`.
  - What is the difference?
- If an error occurs in code produced by a macro defined using `#define`,
  - the error is reported at the point of use in the expanded macro text
- If an error occurs in code produced by a macro defined using `inline`,
  - the error is reported at the relevant line in the `inline` definition itself
    - Generally much more useful.

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 expresses in terms of this
- LTL starts with a notion of "weak until",
  which says that (`p 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 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" is weak until with an additional requirements 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 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`

Compiling LTL with SPIN

- SPIN can take a (quoted) LTL formula as an argument on the command-line and output the corresponding `never` claim on `stdout` :
  (Here, for example, we are claiming that p and q are never true together at the start)

```
:- spin -f "p && q"
never  {    /* p && q */
accept_init:
T0_init:
        do
        :: atomic { ((p) && (q)) -> assert(!((p) && (q))) }
        od;
accept_all:
        skip
}
```

- We can also pipe the formula from a file:

```
:- spin -f "$(< pandq.txt)"
```

Where are the comparisons?

- The LTL notation supported by SPIN does not contain useful comparison operators such as ==, <=, >=
- It also doesn't support general expressions like `x+1` or `y * (z - x)`, or even `x+1 < y`.
- Instead, we must used the C-preprocessor `#define` to use a single variable to denote a boolean expression:
  - `#define p (x+1<y).`
    The parentheses are recommended !
  - We can then use `p` in LTL to stand for `x+1<y`.
  - These `#define`s go in the Promela file into which the `never` claim is put.

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)