Software Model Checking: Theory and Practice

Lecture: Specification Checking - Temporal Logic
Objectives

- Understand why temporal logic can be a useful formalism for specifying properties of concurrent/reactive systems.
- Understand the intuition behind Computation Tree Logic (CTL) – the specification logic used e.g., in the well-known SMV model-checker.
- Be able to confidently apply Linear Temporal Logic (LTL) – the specification logic used in e.g., Bogor and SPIN – to specify simple properties of systems.
- Understand the formal semantics of LTL.
Outline

- CTL by example
- LTL by example
- LTL – formal definition
- Common properties to be stated for concurrent systems and how they can be specified using LTL
- Bogor’s support for LTL
Reasoning about Executions

- We’ve seen specifications that are about individual program states
  - e.g., assertions, invariants

- Sometimes we want to reason about the relationship between multiple states
  - Must one state always precede another?
  - Does seeing one state preclude the possibility of subsequently seeing another?

- We need to shift our thinking from states to paths in the state space
We want to reason about execution trees
- tree node = snapshot of the program’s state

Reasoning consists of two layers
- defining predicates on the program states (control points, variable values)
- expressing temporal relationships between those predicates
Examples

- A use of a variable must be preceded by a definition
- When a file is opened it must subsequently be closed
- You cannot shift from drive to reverse without passing through neutral
- The program will eventually terminate
Why Use Temporal Logic?

- Requirements of concurrent, distributed, and reactive systems are often phrased as constraints on *sequences of events or states* or constraints on *execution paths*.
- Temporal logic provides a formal, expressive, and compact notation for realizing such requirements.
- The temporal logics we consider are also strongly tied to various computational frameworks (e.g., automata theory) which provides a foundation for building verification tools.
Linear Time Logic

Restrict path quantification to "ALL" (no "EXISTS")
Linear Time Logic

Restrict path quantification to "ALL" (no "EXISTS")

Reason in terms of branching traces instead of branching trees
Linear Time Logic (LTL)

Syntax

\( \Phi ::= P \)
\( \mid \neg \Phi \mid \Phi \land \Phi \mid \Phi \lor \Phi \mid \Phi \rightarrow \Phi \)
\( \mid [] \Phi \mid <> \Phi \mid \Phi \mathbin{U} \Gamma \mid \mathcal{X} \Phi \)

...primitive propositions
...propositional connectives
...temporal operators

Semantic Intuition

\( [] \Phi \) ...always \( \Phi \)

\( <> \Phi \) ...eventually \( \Phi \)

\( \Phi \mathbin{U} \Gamma \) ...\( \Phi \) until \( \Gamma \)
# Modal vs. Temporal Logic

<table>
<thead>
<tr>
<th>Modal Logic</th>
<th>Temporal Logic (LTL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(G, R)</td>
<td>(G,&lt;)</td>
</tr>
<tr>
<td>Kripke Structures</td>
<td>Temporal Structures</td>
</tr>
<tr>
<td>World $g \in G$</td>
<td>Time point $g \in G$</td>
</tr>
<tr>
<td>$[\square]F$</td>
<td>$[\square]F$ (always in the future)</td>
</tr>
<tr>
<td>$&lt;&gt;F$</td>
<td>$&lt;&gt;F$ (sometimes in the future)</td>
</tr>
<tr>
<td></td>
<td>XF (next time point)</td>
</tr>
<tr>
<td></td>
<td>F U G (until)</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Specification Checking: Temporal Logic
Linear Time Logic

\[ []<>p \]

- "Along all paths, it must be the case that globally (i.e., in each state we come to) eventually \( p \) will hold"
- Expresses a form of fairness
  - \( p \) must occur infinitely often along the path
  - To check \( \Phi \) under the assumption of fair traces, check \( []<>p \rightarrow \Phi \)
Linear Time Logic

“Along all paths, eventually it is the case that p holds at each state)” (i.e., “eventually permanently p”)
“Any path contains only finitely many ¬p states”
### Linear Time Logic

#### Example

| p W q | = | []p || (p U q) |

- **p W q**
- **[]p || (p U q)**

- "p unless q", or "p waiting for q", or "p weak-until q"

---

**Specification Checking : Temporal Logic**
Semantics for LTL

- Semantics of LTL is given with respect to a (usually infinite) path or trace
  - \( \pi = s_1 s_2 s_3 \ldots \)

- We write \( \pi_i \) for the suffix starting at \( s_i \), e.g.,
  - \( \pi_3 = s_3 s_4 s_5 \ldots \)

- A system satisfies an LTL formula \( f \) if each path through the system satisfies \( f \).
Semantics of LTL

- For primitive propositions p:
  \[ \pi \models p \iff s_1 \models p \quad \pi \models \neg p \iff s_1 \models \neg p \]
- \[ \pi \models \phi \land \psi \iff \pi \models \phi \text{ and } \pi \models \psi \]
- \[ \pi \models \phi \lor \psi \iff \pi \models \phi \text{ or } \pi \models \psi \]
- \[ \pi \models X\phi \iff \pi_2 \models \phi \]
- \[ \pi \models \neg \phi \iff \exists i \geq 1. \pi_i \models \phi \]
- \[ \pi \models \neg \phi \iff \forall i \geq 1. \pi_i \models \phi \]
- \[ \pi \models (\phi \cup \psi) \iff \exists i \geq 1. \pi_i \models \psi \text{ and } \forall j : 1 \leq j < i. \pi_j \models \phi \]
LTL Notes

- Invented by Prior (1960’s), and first used to reason about concurrent systems by A. Pnueli, Z. Manna, etc.
- LTL model-checkers are usually explicit-state checkers due to connection between LTL and automata theory
- Most popular LTL-based checker is SPIN (G. Holzman)
Comparing LTL and CTL

- CTL is not strictly more expression than LTL (and vice versa)
- CTL* invented by Emerson and Halpern in 1986 to unify CTL and LTL
- We believe that almost all properties that one wants to express about software lie in intersection of LTL and CTL
A classic distinction ...

- **Safety properties**
  - “nothing bad ever happens”
  - are violated by a *finite* path prefix that ends in a bad thing
  - are fundamentally about the *history* of a computation up to a point

- **Liveness properties**
  - “something good eventually happens”
  - are violated by *infinite* path suffixes on which the good thing never happens
  - are fundamentally about the *future* of a computation from a point onward
Examples

- A use of a variable must be preceded by a definition
- When a file is opened it must subsequently be closed
- You cannot shift from drive to reverse without passing through neutral
- No pair of adjacent dining philosophers can be eating at the same time
- The program will eventually terminate
- The program is free of deadlock
Examples

- A use of a variable must be preceded by a definition -- **Safety**
- When a file is opened it must subsequently be closed -- **Liveness**
- You cannot shift from drive to reverse without passing through neutral -- **Safety**
- No pair of adjacent dining philosophers can be eating at the same time -- **Safety**
- The program will eventually terminate -- **Liveness**
- The program is free of deadlock -- **Safety**