Chapter 13

Invariants and Temporal Logic

Every embedded system must be designed to meet certain requirements. Such system requirements are also termed as properties or specifications. The need for specifications is aptly captured by the following quotation (paraphrased from Young et al. (1985)):

“A design without specifications cannot be right or wrong, it can only be surprising!”

In present engineering practice, it is common to have system requirements stated in a natural language such as English. As an example, consider the SpaceWire communication protocol that is gaining adoption with several national space agencies (European Cooperation for Space Standardization, 2002). Here are two properties re-
produced from Section 8.5.2.2 of the specification document, stating conditions on the behavior of the system upon reset:

1. “The ErrorReset state shall be entered after a system reset, after link operation has been terminated for any reason or if there is an error during link initialization.”

2. “Whenever the reset signal is asserted the state machine shall move immediately to the ErrorReset state and remain there until the reset signal is de-asserted.”

It is important to precisely state requirements to avoid ambiguities inherent in natural languages. For example, consider the first property of the SpaceWire protocol stated above. Observe that there is no mention of when the ErrorReset state is to be entered. The systems that implement the SpaceWire protocol are synchronous, meaning that transitions of the state machine occur on ticks of a system clock. Given this, must the ErrorReset state be entered on the very next tick after one of the three conditions becomes true or on some subsequent tick of the clock? As it turns out, the document intends the system to make the transition to ErrorReset on the very next tick, but this is not made precise by the English language description.

This chapter will introduce techniques to specify system properties mathematically and precisely. The specific formalism we will use is called temporal logic. As the name suggests, temporal logic is a precise mathematical notation with associated rules for representing and reasoning about timing-related properties of systems. While temporal logic has been used by philosophers and logicians since the times of Aristotle, it is only in the last thirty years that it has found application as a mathematical notation for specifying system requirements.

One of the most common kinds of system property is an invariant. It is also one of the simplest forms of a temporal logic property. We will first introduce the notion of an invariant, and then generalize it to more expressive specifications in temporal logic.

13.1 Invariants

An invariant is a system property that remains true at all times.
Put another way, an invariant is true in the initial state of the system, and remains unchanged (true) as the system evolves, after every reaction, in every state.

In practice, most properties are invariants. Both properties of the SpaceWire protocol stated above are invariants, although this might not be immediately obvious. Both SpaceWire properties specify conditions that must hold true always.

Here is an example of an invariant property of a model we have encountered in Chapter 2.

**Example 13.1:** Consider the model of a traffic light controller given in Figure 2.10 and its environment as modeled in Figure 2.11. Consider the system formed by the asynchronous composition of these two state machines. An obvious property that the composed system must satisfy is that there is no pedestrian crossing when the traffic light is green (when cars are allowed to move). This property must always remain true of this system, and hence is a system invariant.

It is also desirable to specify invariant properties of software and hardware implementations of embedded systems. Some of these properties specify correct programming practice on language constructs. For example, the C language property

"Never dereference a null pointer"

is an invariant specifying good programming practice. Typically dereferencing a null pointer in a C program results in a segmentation fault, possibly leading to a system crash.

Similarly, several desirable properties of concurrent programs are invariants.

**Example 13.2:** Consider the following property regarding an absence of deadlock:

If a thread $A$ blocks while acquiring a mutex lock, then the thread $B$ that holds that lock must not be blocked attempting to acquire a lock held by $A$. 
This property is an invariant on the multi-threaded program constructed from threads A and B.

Many system invariants also impose requirements on program data, as illustrated in the example below.

Example 13.3: Consider the following example of a software task from the open source Paparazzi unmanned aerial vehicle (UAV) project (Nemer et al., 2006):

```c
void altitude_control_task(void) {
    if (pprz_mode == PPRZ_MODE_AUTO2
        || pprz_mode == PPRZ_MODE_HOME) {
        if (vertical_mode == VERTICAL_MODE_AUTO_ALT) {
            float err = estimator_z - desired_altitude;
            desired_climb
                = pre_climb + altitude_pgain * err;
            if (desired_climb < -CLIMB_MAX) {
                desired_climb = -CLIMB_MAX;
            } else if (desired_climb > CLIMB_MAX) {
                desired_climb = CLIMB_MAX;
            }
        }
    }
}
```

For this example, it is required that the value of the desired_climb variable at the end of altitude_control_task remains within the range [-CLIMB_MAX, CLIMB_MAX]. This is an example of a special kind of invariant, a post condition, that must be maintained every time altitude_control_task returns. Determining whether this is the case requires analyzing the control flow of the program.
13.2 Linear Temporal Logic

We now give a formal description of temporal logic and illustrate with examples of how it can be used to specify system behavior. In particular, we study a particular kind of temporal logic known as linear temporal logic, or LTL. There are other forms of temporal logic, some of which are briefly surveyed in sidebars.

Using LTL, one can express a property over a single, but arbitrary execution of a system. For instance, one can express the following kinds of properties in LTL:

- **Occurrence of an event and its properties.** For example, one can express the property that an event $A$ must occur at least once in every trace of the system, or that it must occur infinitely-many times.

- **Causal dependency between events.** An example of this is the property that if an event $A$ occurs in a trace, then event $B$ must also occur.

- **Ordering of events.** An example of this kind of property is one specifying that every occurrence of event $A$ is preceded by a matching occurrence of $B$.

We now formalize the above intuition about the kinds of properties expressible in linear temporal logic. In order to perform this formalization, it is helpful to fix a particular formal model of computation. We will use the theory of finite-state machines, introduced in Chapter 2.

Recall from Section 2.6 that an execution trace of a finite-state machine is a sequence of the form

$$q_0, q_1, q_2, q_3, \ldots,$$

where $q_j = (x_j, s_j, y_j)$ where $s_j$ is the state, $x_j$ is the input valuation, and $y_j$ is the output valuation at reaction $j$.

13.2.1 Propositional Logic Formulas

First, we need to be able to talk about conditions at each reaction, like whether an input or output is present, what the value of an input or output is, or what the state is. Let an atomic proposition be such a statement about the inputs, outputs, or states. It is a predicate (an expression that evaluates to true or false). Examples of atomic propositions that are relevant for the state machines in Figure 13.1 are:
true Always true.
false Always false.
x True if input \( x \) is present.
\( x = \text{present} \) True if input \( x \) is present.
\( y = \text{absent} \) True if \( y \) is absent.
b True if the FSM is in state \( b \)

In each case, the expression is true or false at a reaction \( q_i \). The proposition \( b \) is true at a reaction \( q_i \) if \( q_i = (x, b, y) \) for any valuations \( x \) and \( y \), which means that the machine is in state \( b \) at the start of the reaction. I.e., it refers to the current state, not the next state.

A propositional logic formula or (more simply) proposition is a predicate that combines atomic propositions using logical connectives: conjunction (logical AND, denoted \( \land \)), disjunction (logical OR denoted \( \lor \)), negation (denoted \( \neg \)), and implies (denoted \( \Rightarrow \)). Propositions for the state machines in Figure 13.1 include any of the above atomic proposition and expressions using the logical connectives together with atomic propositions. Here are some examples:

\[
\begin{align*}
x \land y & \quad \text{True if } x \text{ and } y \text{ are both present.} \\
x \lor y & \quad \text{True if either } x \text{ or } y \text{ is present.} \\
x = \text{present} \land y = \text{absent} & \quad \text{True if } x \text{ is present and } y \text{ is absent.} \\
\neg y & \quad \text{True if } y \text{ is absent.} \\
a \Rightarrow y & \quad \text{True if whenever the FSM is in state } a, \\
& \text{the output } y \text{ will be made present by the reaction}
\end{align*}
\]

**input:** \( x \): pure  
**output:** \( y \): pure

Figure 13.1: Two finite state machines used to illustrate LTL formulas.
Note that if \( p_1 \) and \( p_2 \) are propositions, the proposition \( p_1 \implies p_2 \) is true if and only if \( \neg p_2 \implies \neg p_1 \). In other words, if we wish to establish that \( p_1 \implies p_2 \) is true, it is equally valid to establish that \( \neg p_2 \implies \neg p_1 \) is true. In logic, the latter expression is called the **contrapositive** of the former.

Note further that \( p_1 \implies p_2 \) is true if \( p_1 \) is false. This is easy to see by considering the contrapositive. The proposition \( \neg p_2 \implies \neg p_1 \) is true regardless of \( p_2 \) is \( \neg p_1 \) is true. Thus, another proposition that is equivalent to \( p_1 \implies p_2 \) is

\[
\neg p_1 \lor (p_1 \land p_2).
\]

### 13.2.2 LTL Formulas

An **LTL formula**, unlike the above propositions, applies to an entire trace

\[ q_0, q_1, q_2, \ldots, \]

rather than to just one reaction \( q_i \). The simplest LTL formulas look just like the propositions above, but they apply to an entire trace rather than just a single element of the trace. If \( p \) is a proposition, then by definition, we say that LTL formula \( \phi = p \) holds for the trace \( q_0, q_1, q_2, \ldots \) if and only if \( p \) is true for \( q_0 \). It may seem odd to say that the formula holds for the entire trace even though the proposition only holds for the first element of the trace, but we will see that LTL provides ways to reason about the entire trace.

By convention, we will denote LTL formulas by \( \phi, \phi_1, \phi_2, \) etc. and propositions by \( p, p_1, p_2, \) etc.

Given a state machine \( M \) and an LTL formula \( \phi \), we say that \( \phi \) holds for \( M \) if \( \phi \) holds for all possible traces of \( M \). This typically requires considering all possible inputs.

---

**Example 13.4:** The LTL formula \( a \) holds for Figure 13.1(b), because all traces begin in state \( a \). It does not hold for Figure 13.1(a).

The LTL formula \( x \implies y \) holds for both machines. In both cases, in the first reaction, if \( x \) is present, then \( y \) will be present.
To demonstrate that an LTL formula is false for an FSM, it is sufficient to give one trace for which it is false. Such a trace is called a **counterexample**. To show that an LTL formula is true for an FSM, you must demonstrate that it is true for all traces, which is often much harder (although not so much harder when the LTL formula is a simple propositional logic formula, because in that case we only have to consider the first element of the trace).

**Example 13.5:** The LTL formula $y$ is false for both FSMs in Figure 13.1. In both cases, a counterexample is a trace where $x$ is absent in the first reaction.

In addition to propositions, LTL formulas can also have one or more special **temporal operators**. These make LTL much more interesting, because the enable reasoning about entire traces instead of just making assertions about the first element of a trace. There are four main temporal operators, which we describe next.

**G Operator**

The property $G\phi$ (which is read as “globally $\phi$”) holds for a trace if $\phi$ holds for every suffix of that trace.

In mathematical notation, $G\phi$ holds for the trace if and only if, for all $j \geq 0$, formula $\phi$ holds in the suffix $q_j, q_{j+1}, q_{j+2}, \ldots$.

**Example 13.6:** In Figure 13.1(b), $G(x \implies y)$ is true for all traces of the machine, and hence holds for the machine. $G(x \land y)$ does not hold for the machine, because it is false for any trace where $x$ is absent in any reaction. Such a trace provides a counterexample.

If $\phi$ is a propositional logic formula, then $G\phi$ simply means that $\phi$ holds in every reaction. We will see, however, that when we combine the $G$ operator with other temporal logic operators, we can make much more interesting statements about traces and about state machines.
F Operator

The property $F\phi$ (which is read as “eventually $\phi$” or “finally $\phi$”) holds for a trace if $\phi$ holds for some suffix of the trace.

Formally, $F\phi$ holds for the trace if and only if, for some $j \geq 0$, formula $\phi$ holds in the suffix $q_j, q_{j+1}, q_{j+2}, \ldots$.

**Example 13.7:** In Figure 13.1(a), $Fb$ is trivially true because the machine starts in state $b$, hence, for all traces, the proposition $b$ holds for the trace itself (the very first suffix).

More interestingly, $G(x \Rightarrow Fb)$ holds for Figure 13.1(a). This is because if $x$ is present in any reaction, then the machine will eventually be in state $b$. This is true even in suffixes that start in state $a$.

Notice that parentheses can be important in interpreting an LTL formula. For example, $(Gx) \Rightarrow (Fb)$ is trivially true because $Fb$ is true for all traces (since the initial state is $b$).

Notice that $F\neg \phi$ holds if and only if $\neg G\phi$. That is, stating that $\phi$ is eventually false is the same as stating that $\phi$ is not always true.

X Operator

The property $X\phi$ (which is read as “next state $\phi$”) holds for a trace $q_0, q_1, q_2, \ldots$ if and only if $\phi$ holds for the trace $q_1, q_2, q_3, \ldots$.

**Example 13.8:** In Figure 13.1(a), $x \Rightarrow Xa$ holds for the state machine, because if $x$ is present in the first reaction, then the next state will be $a$. $G(x \Rightarrow Xa)$ does not hold for the state machine because it does not hold for any suffix that begins in state $a$.

In Figure 13.1(b), $G(b \Rightarrow Xa)$ holds for the state machine.
**Safety and Liveness Properties**

Informally, a safety property is one specifying that “nothing bad happens” during an execution of the system. Similarly, a liveness property specifies that “something good will happen” during a system execution.

More formally, a property \( p \) is a **safety property** if one can show that the system does not satisfy \( p \) by exhibiting a finite-length execution of that system. We say \( p \) is a **liveness property** of a system if, for every finite-length prefix of a system execution that does not satisfy \( p \), it is possible to extend the execution so as to satisfy \( p \).

Invariants, discussed in Section 13.1, are safety properties. To show that an invariant does not hold, one needs only to exhibit an execution that leads to a state where the invariant does not hold. Such an execution must be finite. An invariant may be expressed in LTL as \( \mathbf{G}\phi \).

Liveness properties, on the other hand, specify performance or progress requirements on a system. For a state machine, a property of the form \( \mathbf{F}\phi \) is a liveness property. No finite execution can establish that this is false.

The following is a slightly more elaborate example of a liveness property:

“Whenever an interrupt is asserted, the corresponding interrupt service routine (ISR) is eventually executed.”

In temporal logic, if \( p_1 \) is the property than an interrupt is asserted, and \( p_2 \) is the property that the interrupt service routine is executed, then this property can be written

\[
\mathbf{G}(p_1 \implies \mathbf{F}p_2) .
\]

Note that in addition to being a liveness property, this is also a safety property. In general, invariants can be both safety and liveness properties. In the above example, \( p_1 \implies \mathbf{F}p_2 \) must hold whenever an interrupt is asserted.

Liveness properties can be either **bounded** or **unbounded**. A **bounded liveness** property specifies a time bound on when something desirable must happen. In the above example, if the ISR must be executed within 100 clock cycles of the interrupt being asserted, the property is a bounded liveness property; otherwise, if there is no such time bound on the occurrence of the ISR, it is an **unbounded liveness** property. LTL can express a limited form of bounded liveness properties using the \( \mathbf{X} \) operator, but it does not provide any mechanism for quantifying time directly.
U Operator

The property $\phi_1 U \phi_2$ (which is read as “$\phi_1$ until $\phi_2$”) holds for a trace if $\phi_2$ holds for some suffix of that trace, and $\phi_1$ holds until $\phi_2$ becomes true.

Formally, $\phi_1 U \phi_2$ holds for the trace if and only if there exists $j \geq 0$ such that $\phi_2$ holds in the suffix $q_j, q_{j+1}, q_{j+2}, \ldots$ and $\phi_1$ holds in suffixes $q_i, q_{i+1}, q_{i+2}, \ldots$, for all $i$ s.t. $0 \leq i < j$. $\phi_1$ may or may not hold for $q_j, q_{j+1}, q_{j+2}, \ldots$.

**Example 13.9:** In Figure 13.1(b), $a U x$ is true for any trace for which $F x$ holds. Since this does not include all traces, $a U x$ does not hold for the state machine.

Some authors define a weaker form of the $U$ operator that does not require $\phi_2$ to hold. Using our definition, this can be written

$$(G \neg \phi_2) \lor (\phi_1 U \phi_2).$$

This holds if either $\phi_2$ never holds (for any suffix) or, if $\phi_2$ holds for some suffix, then $\phi_1$ holds for all previous suffixes. This can equivalently be written

$$(F \phi_2) \implies (\phi_1 U \phi_2).$$

**Example 13.10:** In Figure 13.1(b), $(G \neg x) \lor (a U x)$ holds for the state machine.

13.2.3 Using LTL Formulas

Consider the following English descriptions of properties and their corresponding LTL formalizations:
Example 13.11: “Whenever the robot is facing an obstacle, eventually it moves at least 5 cm away from the obstacle.”

Let $p$ denote the condition that the robot is facing an obstacle, and $q$ denote the condition where the robot is at least 5 cm away from the obstacle. Then, this property can be formalized in LTL as

$$G(p \implies Fq) .$$

Example 13.12: Consider the SpaceWire property:

“Whenever the reset signal is asserted the state machine shall move immediately to the ErrorReset state and remain there until the reset signal is de-asserted.”

Let $p$ be true when the reset signal is asserted, and $q$ be true when the state of the FSM is ErrorReset. Then, the above English property is formalized in LTL as:

$$G(p \implies X(q U \neg p)) .$$

In the above formalization, we have interpreted “immediately” to mean that the state changes to ErrorReset in the very next time step. Moreover, the above LTL formula will fail to hold for any execution where the reset signal is asserted an not eventually de-asserted. It was probably the intent of the standard that the reset signal should be eventually de-asserted, but the English language statement does not make this clear.

Example 13.13: Consider the traffic light controller in Figure 2.10. A property of this controller is that the outputs always cycle through $sigG$, weather it is green or red.
\( \text{sig}Y \) and \( \text{sig}R \). We can express this in LTL as follows:

\[
\begin{align*}
\mathbf{G} \{} & (\text{sig}G \Longrightarrow \mathbf{X}((\neg \text{sig}R \land \neg \text{sig}G) \mathbf{U} \text{sig}Y)) \\
\land & (\text{sig}Y \Longrightarrow \mathbf{X}((\neg \text{sig}G \land \neg \text{sig}Y) \mathbf{U} \text{sig}R)) \\
\land & (\text{sig}R \Longrightarrow \mathbf{X}((\neg \text{sig}Y \land \neg \text{sig}R) \mathbf{U} \text{sig}G)) \\
\}.
\]

The following LTL formulas express commonly useful properties.

**Verification and Model Checking**

Amir Pnueli (1977) was the first to formalize temporal logic as a way of specifying program properties. Since his seminal paper, temporal logic has become widespread as a way of specifying properties for a range of systems, including hardware, software, and cyber-physical systems.

The introduction of temporal logic also led to several advances in techniques for automatically verifying that a system satisfies its specification. One of the main such techniques is **model checking**. Model checking is an algorithmic method for determining whether a system model satisfies a formal specification expressed as a temporal logic formula. It was introduced in 1981 by Clarke and Emerson (1981) and Queille and Sifakis (1981). We will study some of the key concepts in model checking in more detail in Chapter 15.

Several tools are available for checking that finite-state machines satisfy specifications in temporal logic. Chief among these are SMV (symbolic model verifier), which was first developed at Carnegie Mellon University by Kenneth McMillan-McMillan, Kenneth. Currently a few different versions of SMV are available online (McMillan; NuSMV Group). The SPIN model checker (Holzmann) developed by Gerard Holzmann, is another leading tool for model checking a system constructed from communicating FSMs.

The seminal contributions in temporal logic and model checking have been recognized by the ACM Turing award, the highest honor in Computer Science, in 1996 and 2007.
(a) **Infinitely-many occurrences:** This property is of the form $\text{GF}p$, meaning that it is always the case that $p$ is true eventually. Put another way, this means that $p$ is true **infinitely often**.

(b) **Steady-state property:** This property is of the form $\text{FG}p$, read as “from some point in the future, $p$ holds at all times.” This represents a steady-state property, indicating that after some point in time, the system reaches a **steady state** in which $p$ is always true.

(c) **Request-response property:** The formula $\text{G}(p \implies \text{F}q)$ can be interpreted to mean that a request $p$ will eventually produce a response $q$.

### 13.3 Summary

Dependability and correctness are central concerns in embedded systems design. Formal specifications, in turn, are central to achieving these goals. In this chapter, we have studied temporal logic, one of the main approaches for writing formal specifications.

We have only studied one of the versions of temporal logic, namely, *propositional linear temporal logic*. Many other variants of temporal logic exist, for example, for reasoning about models of computation other than finite-state machines. These include **branching-time temporal logic** (e.g., CTL), for reasoning about properties over a computation tree of a non-deterministic system; **real-time temporal logic**, for reasoning about real-time systems; and **probabilistic temporal logic**, for reasoning about probabilistic models such as Markov chains or Markov decision processes.