Temporal Logic and Model Checking

Fraser Brown and Ian Dardik

Fall 2024

0.1 Motivating Example

Previously, we introduced Hoare Logic for reasoning about partial correctness of a program. Hoare Logic is a powerful formalism for reasoning about sequential programs, but fall short for concurrent programs. For example, consider the following example (due to Lamport [**?**]):

Program 1	Program 2
$\{true\}$	$\{true\}$
$x := x + 1$	$x := x + y + 1$
$\{x = x + 1\}$	$x := x - y$
	$\{x = x + 1\}$

Both programs above satisfy the exact same Hoare triples; moreover, they are semantically equivalent. However, consider the result of running each program above in parallel with the following program:

Program 3

$y := y - 7$

When Program 1 and 3 are run in parallel, the postcondition is unaffected. However, when program 2 and 3 are run in parallel, the correct postcondition changes to $\{x \in \{x + 1, x + 8\}\}$. The key point of this example is that Hoare style pre- and postconditions are insufficient for reasoning about concurrent programs; it is necessary to consider what happens *during* the program execution. Next, we will introduce *linear temporal logic* to address this concern.

0.2 Linear Temporal Logic (LTL)

Temporal logics are types of logics that reason about program state over time, and hence are able to specify correctness conditions during program execution. There are many different temporal logic languages, for instance Linear Temporal Logic (LTL) [**?**], Computation Tree Logic, and the μ-Calculus. We will exclusively focus on LTL in this text, due to its intuitiveness and popularity.

Linear time logics, including LTL, specify properties over *behaviors*. Behaviors are similar to program traces (which were introduced in the textbook), except we define behaviors to be *inifinite* sequences of program state (whereas traces were finite sequences). For example, consider the following behavior:

$$[x \mapsto 0][x \mapsto 1][x \mapsto 2] \dots$$

This program behavior satisfies the invariant $x \geqslant 0$. We can express this in LTL using the syntax $\mathbf{G}(x \geqslant 0)$. The **G** temporal operator is pronounced "always" or "globally" and means that the enclosed formula is true at all time steps. The behavior also satisfies the LTL formula $\mathbf{X}(x = 1)$, which means that $x = 1$ in the second step of the behavior and the **X** operator is pronounced "next". The behavior also satisfies the LTL formula $(x = 0) \wedge \mathbf{XX}(x = 2)$. The final operator we will discuss is **F** which is pronounced "eventually". For example, the behavior above satisfies the formula $\mathbf{F}(x = 1)$ because the behavior eventually satisfies $x = 1$ (at the second time step).

The syntax for an LTL formula ϕ is as follows:

$$\phi ::= \text{non-temporal formula} \mid \phi_1 \wedge \phi_2 \mid \neg\phi' \mid \mathbf{G}\phi' \mid \mathbf{F}\phi' \mid \mathbf{X}\phi'$$

We now define the semantics of LTL in terms of whether a behavior σ satisfies a given LTL formula. We will use $\sigma[i]$ to denote the ith state in the behavior, and $\sigma[i\ldots]$ to denote the behavior $\sigma[i]\sigma[i+1]\ldots$

$\sigma \models \psi$	iff $\sigma[0] \models \psi$ (where ψ is a non-temporal formula)
$\sigma \models \phi \wedge \psi$	iff $\sigma \models \phi$ and $\sigma \models \psi$
$\sigma \models \neg\phi$	iff $\sigma \not\models \phi$
$\sigma \models \mathbf{G}\phi$	iff $\forall i \in \mathbb{N}, \sigma[i\ldots] \models \phi$
$\sigma \models \mathbf{F}\phi$	iff $\exists i \in \mathbb{N}, \sigma[i\ldots] \models \phi$
$\sigma \models \mathbf{X}\phi$	iff $\sigma[1\ldots] \models \phi$

LTL allows us to specify partial correctness properties, e.g. invariants, that are familiar to us from Hoare Logic. Such properties are called *safety properties*; intuitively, safety properties specify that something bad does not happen. However, LTL also allows us to specify an additional class of properties called *liveness properties*. Intuitively, liveness properties specify that something good eventually happens, and allow us to express properties such as termination (full correctness). For example, if the end of a program occurs when the program counter is at location END, then we can specify termination with the formula $\mathbf{F}(pc = END)$. In general, all linear temporal properties are the intersection of a safety and a liveness property [**?**].

There are many LTL patterns that are commonly used. For example, ϕ happens *infinitely often* is specified as $\mathbf{GF}\phi$. Additionally, *stability* is specified as $\mathbf{FG}\phi$. As a final example, ϕ *leads to* ψ is specified as $\mathbf{G}(\phi \Rightarrow \mathbf{F}\psi)$, which means that whenever ϕ happens, ψ must eventually follow.

0.3 Model Checking

0.3.1 Introduction

The semantics above define when a particular behavior satisfies an LTL property. More importantly, however, we are interested to know when an entire program M satisfies a property, written as $M \models P$ for some LTL formula P. To define property satisfaction, we will use the semantics operator $[\![.]\!]$ that denotes the set of all behaviors of a given program or formula. We define $[\![M]\!]$ to be the set of every behavior of M, given by the usual semantics of our programming languages. For LTL formulas, we define $[\![P]\!] = \{\sigma \mid \sigma \models P\}$. We now define $M \models P$ to be exactly when $[\![M]\!] \subseteq [\![P]\!]$, i.e. when every behavior of M satisfies P. In the case that $M \models P$, we say that the program M *satisfies* the property P, or alternatively M is a *model* of P.

Model Checking is the act of using an automated approach to check (and prove) whether a program satisfies a given property. In general, the model checking problem is undecidable, due to Rice's Theorem. However, in the 1980's, Ed Clarke, Allen Emerson, and (separately) Joseph Sifakis showed that model checking is decidable for finite-state programs and specifications. Since its inception, model checking has flourished into an important and active research area in computer science.

The main challenge of model checking (finite state systems) is known as the *state explosion problem* for parallel systems. This problem refers to the fact that the state space of the overall system increases exponentially with the number of parallel processes. The state explosion problem makes it infeasible to model check most complicated programs, although model checking for programs is an active area of research. More often, however, model checking is applied to prove the correctness of a formal specification of a program or protocol, rather than the implementation itself.

Defining and verifying formal specifications is a form of *lightweight formal methods*. The idea is to apply verification at a higher level of abstraction than the program itself. One of

the key advantages to lightweight formal methods is that it makes model checking feasible in many cases. A disadvantage, however, is that model checking proves correctness of the specification, and not the actual implementation. Although verifying a specification–rather than the implementation–may seem to defeat the very purpose of verification, it has many practical advantages for software development. For instance, an engineer can ensure that a complicated algorithm, such as a distributed or concurrent algorithm, is bug-free before even starting to implement it in code. Lghtweight formal methods has enjoyed many success stories across several domains, including distributed systems, operating systems, and security.

0.3.2 Running Example: ConcurrentAdd

Suppose we have a concurrent program where several processes read and write to the same shared variable x. We want each process's source code to look approximately like:

modifications to local state variables

$\vdots$

$x := x + 6$

However, it is extremely important that x never equals 555; if this is the case, the entire system will have a meltdown and explode. Our goal, as the engineer tasked to write the code, is to make sure the code is correct, i.e. the system satisfies the invariant $\mathbf{G}(x \neq 555)$. However, there is a lot of code to write which makes verification tough. Therefore, we will use lightweight formal methods and create a formal specification of the tough, concurrent part of the code, and only prove this part correct. Once we have proved the formal specification to be correct, we can write the code to implement the algorithm. In the following section, we will specify the algorithm in the TLA$^+$ formal specification language.

0.3.3 Specifying ConcurrentAdd in TLA$^+$

In this section we will introduce the TLA$^+$ formal specification language and use it to encode the running example. The TLA$^+$ language allows us to specify our system as a *symbolic transition system*. A symbolic transition system is a tuple $(Init, Next)$, where $Init$ is the *initial predicate* that describes the set of initial states, and $Next$ is the transition relation predicate that describes the transitions that the system can make. It is easiest to understand TLA$^+$ and symbolic transitions systems through and example, so we will proceed to model the ConcurrentAdd system in TLA$^+$. Using the principle of lightweight formal methods, we will only specify the tough, concurrent parts of the code. We show the TLA$^+$ specification in Fig. 1.

The specification begins by declaring a constant $Proc$, also known as a parameter to the system. $Proc$ is the set of processes in the system; since the specification should work for any number of processes, we let $Proc$ be given. A system–such as ConcurrentAdd–that declares parameters is also called a *parameterized system*. For now, it is convenient to think of $Proc$ as the set of two process, e.g. $Proc = \{p_1, p_2\}$.

The specification also declares two state variables x and $memory$. The variable x represents the global shared variable that each process accesses, while $memory$ represents the local memory for each process. In the initial state of the system, x as well as the local memory of each process is set to 0, as specified by the initial state predicate $Init$. The transition relation, defined by $Next$, is the disjunction of three possible actions that a processes can take: $Read$, Inc, and $Write$. In each action, primed state variables (e.g. x') represent the value of a variable in the *next* state, while unprimed variables (e.g. x) represent the current value of a variable.

MODULE *ConcurrentAdd*

CONSTANT $Proc$
VARIABLES $x, memory$

$Init \triangleq$
$\land\ x = 0$
$\land\ memory = [p \in Proc \mapsto 0]$

$Next \triangleq$
$\exists p \in Proc :$
$\lor\ Read(p)$
$\lor\ Inc(p)$
$\lor\ Write(p)$

$Inc(p) \triangleq$
$\land\ memory' = [memory \text{ EXCEPT } ![p] = memory[p] + 6]$
$\land$ UNCHANGED x

$Read(p) \triangleq$
$\land\ memory' = [memory \text{ EXCEPT } ![p] = x]$
$\land$ UNCHANGED x

$Write(p) \triangleq$
$\land\ x' = memory[p]$
$\land$ UNCHANGED $memory$

$Safety \triangleq \Box(x \neq 555)$

Figure 1: TLA$^+$ specification for the ConcurrentAdd example

An example behavior of this system (with two processes) is:

$$[x \mapsto 0, memory \mapsto [p_1 \mapsto 0, p_2 \mapsto 0]],$$
$$[x \mapsto 0, memory \mapsto [p_1 \mapsto 6, p_2 \mapsto 0]],$$
$$[x \mapsto 6, memory \mapsto [p_1 \mapsto 6, p_2 \mapsto 0]],$$
$$[x \mapsto 6, memory \mapsto [p_1 \mapsto 6, p_2 \mapsto 6]],$$
$$\vdots$$

This behavior corresponds to the sequence of actions: $Inc(p_1)$, $Write(p_1)$, $Read(p_2), \ldots$.

The key safety property is also encoded in the specification as $Safety$. In TLA$^+$, the $\Box$ operator is the **G** temporal operator introduced above. Therefore, $Safety$ defines an invariant that says x will never equal 555.

0.3.4 Explicit-State Model Checking

Our goal is to verify that the TLA$^+$ specification satisfies its key safety property $Safety$. In this section, we will explore a verification technique called *explicit-state model checking*.

Explicit-state model checking is a style of model checking that enumerates every reachable state in the underlying transition system. At each state we visit, we check whether the invariant $x \neq 555$ holds. If the invariant does not hold, then model checking reports a safety violation along with a violating behavior. In the case that the invariant holds, we continue to enumerate more states and build the transition system. If no error is detected by the time the entire system is built, then model checking succeeds and the system is verified.

Unfortunately, explicit-state model checking cannot prove that ConcurrentAdd satisfies $Safety$. ConcurrentAdd is an infinite-state transition system, and hence the explicit-state approach will never terminate. Furthermore, the explicit-state approach is not sufficient for proving safety for parameterized systems, such as ConcurrentAdd. Explicit-state model checking can show that *finite instances* of a system are safe (e.g. for $Proc = \{p_1, p_2\}$), but cannot show that the entire family of transition systems (for arbitrary choice of $Proc$) for ConcurrentAdd is safe. Many real-world specifications, e.g. distributed protocols, are specified as infinite-state

parameterized systems. We therefore introduce a new technique that can be used to prove safety for such systems.

0.3.5 The Inductive Invariant Approach

To prove safety for systems that are infinite-state and/or parameterized, we must use the *inductive invariant approach*. An inductive invariant $IndInv$ is a formula that has the following two properties:

$$Init \Rightarrow IndInv \tag{1}$$

$$IndInv \wedge Next \Rightarrow IndInv' \tag{2}$$

Property (1) shows that every initial state satisfies $IndInv$, while (2) shows that IndInv is closed with respect to the transition relation $Next$. Together, these two properties imply that $IndInv$ is an overapproximation of the set of states in the transition system. Inductive invariants that imply safety, i.e. $IndInv \Rightarrow Safety$, are powerful due to the following proof rule:

$$\frac{Init \Rightarrow IndInv \qquad IndInv \wedge Next \Rightarrow IndInv' \qquad IndInv \Rightarrow Safety}{Spec \Rightarrow \Box Safety} \text{ INDINVSAFE}$$

Where $Spec$ is the specification with initial state predicate $Init$ and transition relation $Next$. This rule shows that inductive invariants can be used to prove that a system satisfies a given invariant. The inductive invariant method is especially powerful because it can be used to show safety for infinite-state and parameterized systems.

Are you able to find an inductive invariant that proves ConcurrentAdd is correct? In general, discovering inductive invariants is a tough problem and an active area of research; however, it is feasible to write an inductive invariant for ConcurrentAdd that is relatively concise and elegant.