Lab 1: Specification

Formal Methods: Linear Temporal Logic (LTL)

RICARDO ANDRES CALVO MENDEZ, Purdue University, USA

This is the lab for the module on Specification. The topic is the use of Linear Temporal Logic as an example formal method. See IEC 61508, Part 3, Table A.1, row 1b.

This lab has three parts: pre-lab, in-class component, and take-home component. All students enrolled in the course are expected to complete the pre-lab and the in-class component. The take-home component is one of your options for the "complete 3 take-home labs" aspect of the course.

1 Pre-Lab: Purpose and Context

This lab introduces *formal specification* as used in high-assurance and safety-critical software engineering. Specifically, the lab uses *Linear Temporal Logic (LTL)* and a *model checker* to make software requirements precise and mechanically checkable.

The goal is not to master a new programming language, but to understand how engineers express and evaluate safety and liveness requirements using formal methods, as recommended in standards such as IEC 61508. You will see how subtle differences in specifications or models can lead to different verification outcomes, and how to interpret those outcomes responsibly.

2 Learning Outcomes

After completing this lab, you should be able to:

- Distinguish between *safety* and *liveness* properties.
- Read and interpret LTL formulas that express common engineering requirements.
- Understand how LTL is used to specify system properties across a range of formal verification tools.
- Encode LTL properties in a model-checking tool (using SPIN as a concrete example).
- Run a model checker and interpret verification results and counterexample traces.
- Diagnose whether a verification failure is due to a modeling error or a specification error.

The emphasis of this lab is on *Linear Temporal Logic* as a formal specification language. SPIN is used as a representative, executable context in which LTL properties can be checked, but the concepts and skills developed here apply broadly to other model checkers and formal verification tools used in practice, including in IEC 61508–compliant development workflows.

3 Tools Introduced

This lab makes use of the following tools and formalisms:

- **Linear Temporal Logic (LTL)** for specifying temporal properties of system behavior.
- **SPIN**, a widely used model checker, for verifying LTL properties over finite-state models.
- **Promela**, SPIN's modeling language, used here to encode simple automata.

You are expected to have SPIN installed on your machine and to bring a laptop to class. No prior experience with LTL or SPIN is assumed. On Brightspace, under Readings, you can find the relevant chapter from the SPIN book.

Author's Contact Information: Ricardo Andres Calvo Mendez, Purdue University, West Lafayette, Indiana, USA, rclavome@purdue.edu.

In-Class Lab: Formal Specification with LTL and SPIN

Linear Temporal Logic (LTL) with SPIN

1 Objectives and Setup

1.1 In-Class Goals

The goal of the in-class portion of this lab is to apply Linear Temporal Logic (LTL) in a runnable verification context and to practice interpreting the results of model checking. You will work with small automaton models and a fixed set of LTL properties, using SPIN to determine whether the properties hold and, when they do not, why.

The emphasis in this portion of the lab is on:

(1) understanding what a given LTL property actually states,
(2) relating that property to the modeled system behavior, and
(3) diagnosing whether a verification failure reflects a genuine system defect or a problem with the specification.

This workflow reflects common practice in IEC 61508–compliant development contexts, where formally stated safety requirements are often developed or reviewed by engineers in a safety or assurance role and then passed to a software engineering team for implementation and verification. In such settings, software engineers are frequently responsible for operationalizing formal requirements in specific tools, evaluating verification evidence, and determining whether discrepancies indicate design flaws, modeling assumptions, or issues with the requirements themselves.

1.2 Assumptions and Scope

This in-class lab assumes that you have completed the pre-lab and are familiar with the basic purpose of LTL and model checking.

To keep the activity focused and tractable within the available time:

- Each system is modeled as a small, finite-state automaton.
- Models are finite-state-machine automata. They are sequential and avoid concurrency, message passing, or scheduling concerns.
- LTL properties are limited to simple safety and liveness patterns.

1.3 Provided Files and Artifacts

You are provided with a GitHub repository containing Promela models for each system under study.

2 Using SPIN in This Lab

Here are the commands used to run SPIN during this lab. All commands assume that `spin` is installed and available on your command line, per the pre-lab instructions. For additional help interpreting errors or output, you may consult documentation or chat with your preferred LLM assistant.

2.1 Running the Model Checker

Given a Promela model file `model.pml` that includes one or more LTL properties, you can ask SPIN to check the model using:

```
$ spin -run model.pml
```

This command instructs SPIN to:

- translate the LTL properties into internal correctness checks, and
- explore the state space of the model to determine whether the properties hold.

If all checked properties hold, SPIN will report successful verification. If a property is violated, SPIN will report an error and generate a counterexample trace.

2.2 Interpreting Verification Outcomes

SPIN's output falls into two broad categories:

- **No errors found**: The model satisfies the checked LTL properties under the explored state space.
- **Error found**: At least one LTL property is violated by the model.

An error report does not, by itself, indicate whether the underlying issue is: (i) a defect in the system model, or (ii) a problem with the stated property. Determining which of these applies is a central task in this lab.

2.3 Reading Counterexample Traces

When SPIN reports a violation, it produces a counterexample trace showing an execution that leads to the error. You can replay this trace using:

```
$ spin -t model.pml
```

The counterexample trace represents a concrete sequence of states and transitions that violates the property. Your task is to examine this trace and relate it back to:

- the intended behavior of the system, and
- the meaning of the violated LTL property.

You will use these traces to decide whether a reported violation reflects a genuine system defect or a mismatch between the property and the modeled assumptions.

2.4 Reminder of LTL and SPIN notation

Table 1 summarizes the Linear Temporal Logic (LTL) operators that appear in this lab, together with their encodings in SPIN. The table is organized to emphasize the distinction between *state-level* predicates and Boolean connectives, which are evaluated within a single system state, and *temporal* operators, which quantify over future states along an execution trace.

When reading or writing LTL properties, it is important to remember that implication ($\rightarrow$) and Boolean connectives are evaluated at a single point in time, while temporal operators such as $\Box$ (always), $\Diamond$ (eventually), and $\bigcirc$ (next) are the constructs that introduce temporal structure. Combinations of temporal operators, such as $\Box\Diamond$ (infinitely often) and $\Diamond\Box$ (eventually always), express stronger long-run behavioral expectations and are particularly sensitive to modeling assumptions.

Table 1. LTL operators used in this lab and their SPIN encodings. The first group lists state predicates and Boolean connectives, which are evaluated within a single system state. The second group introduces temporal operators that refer to future states along an execution trace. The final rows show common operator combinations that express recurring behavior (□◇) and stabilization (◇□).

LTL syntax	Name	Intuitive semantic	Encoding in SPIN notation
φ	State predicate	"φ holds in the current state"	`phi`
$\neg\varphi$	Negation	"not φ"	`! (phi)`
$\varphi \wedge \psi$	Conjunction	"φ and ψ"	`(phi) && (psi)`
$\varphi \vee \psi$	Disjunction	"φ or ψ"	`(phi) || (psi)`
$\varphi \rightarrow \psi$	Implication	"if φ then ψ" (same state)	`(! (phi)) || (psi)`
$\Box\varphi$	Always	"φ holds at every step"	`[] (phi)`
$\Diamond\varphi$	Eventually	"φ holds at some future step"	`<> (phi)`
$\bigcirc\varphi$	Next (also denoted X)	"φ holds at the next step"	`X (phi)`
$\varphi\ U\ \psi$	Until	"φ holds until ψ holds"	`(phi) U (psi)`
$\Box\Diamond\varphi$	Infinitely often	"φ occurs infinitely often"	`[] <> (phi)`
$\Diamond\Box\varphi$	Eventually always	"from some point onward, φ always holds"	`<> [] (phi)`
$\Box(\varphi \rightarrow \psi)$	Invariant implication	"whenever φ holds, ψ also holds in the same state"	`[] ( (! (phi)) || (psi) )`
$\Box(\varphi \rightarrow \Diamond\psi)$	Response / leads-to	"whenever φ holds, ψ eventually holds"	`[] ( (! (phi)) || <> (psi) )`
$\Diamond(\varphi \wedge \psi)$	Eventual coincidence	"at some future point, both φ and ψ hold simultaneously"	`<> ( (phi) && (psi) )`

Throughout the in-class exercises, you should use Table 1 as a reference when translating properties into SPIN notation and when interpreting verification results.

3 Problem 1: Traffic Light Controller

3.1 System Description

This exercise models a simplified two-direction intersection controller for *North–South* (NS) and *East–West* (EW) traffic. At any time, each direction is intended to have one of the usual signal indications (e.g., Green, Yellow, Red). The controller cycles through phases that grant right-of-way to one direction at a time, with an intermediate all-red interlock used to avoid conflicting greens.

3.2 Automaton Model

The Promela automaton model for this problem can be found in the `traffic/` subdirectory of the GitHub repository provided for this lab. (If multiple variants are provided, use the filenames in that directory to select the intended model.)

3.3 Given Properties

The following properties are expressed in Linear Temporal Logic (LTL). Your task is to encode each property using SPIN's `ltl` syntax, run SPIN on the provided model(s), and interpret the results.

Safety Properties. The following properties are intended to express safety-style requirements, stating that certain undesirable situations never occur.

- **(T-S1) No conflicting greens.**

$$\Box\,\neg\big(ns\_green \wedge ew\_green\big)$$

- **(T-S2) If NS is green, then EW is red.**

$$\square\,(ns\_green \rightarrow ew\_red)$$

- **(T-S3) If EW is green, then NS is red.**

$$\square\,(ew\_green \rightarrow ns\_red)$$

- **(T-S4) No simultaneous yellow.**

$$\square\,\neg(ns\_yellow \wedge ew\_yellow)$$

Liveness Properties. The following properties express liveness-style expectations about progress or eventual behavior.

- **(T-L1) Each direction eventually gets a green.** *(Requires a cyclic or fair controller.)*

$$\square\Diamond\, ns\_green \;\wedge\; \square\Diamond\, ew\_green$$

- **(T-L2) If NS is green, then eventually EW will be green.**

$$\square\,(ns\_green \rightarrow \Diamond\, ew\_green)$$

- **(T-L3) If EW is green, then eventually NS will be green.**

$$\square\,(ew\_green \rightarrow \Diamond\, ns\_green)$$

- **(T-L4) NS eventually stabilizes in a recurring green state.** *(Strong; often not intended in practice.)*

$$\Diamond\square\, ns\_green$$

3.4 Student Analysis and Encoding

For each property listed above, complete the following steps *before* running SPIN.

(1) **Assess reasonableness.** Based on the problem context and system description, indicate whether the property appears to express a reasonable requirement for the system. Briefly justify your assessment.
(2) **Assess restrictiveness.** Determine whether the property could reasonably be violated by a legitimate system design. In particular, consider whether the property makes implicit assumptions about scheduling, fairness, or timing that may not be guaranteed.
(3) **Encode the property.** Translate the LTL formula into SPIN notation using the syntax summarized in Table 1. Record the resulting `ltl` declaration exactly as checked.

Do not modify the provided automaton models during this step. Your goal here is to reason about the properties themselves and to ensure that their formal encodings accurately reflect their intended meaning.

Table 2. Student analysis and SPIN encoding of traffic light properties.

Original LTL rule	**Reasonable requirement?**	**May be violated by a legitimate design?**	**SPIN encoding**
Safety properties			
$\square \neg(ns\_green \wedge ew\_green)$	Yes. Only one road may be active at a time.	No. This is a **must** in this traffic light system	`ltl T_S1 {[] (!(ns_green && ew_green))}`
$\square(ns\_green \rightarrow ew\_red)$	Yes. Same as before, Only one road may be active at a time	Yes. It may be violated if one light is *green* and the other is *yellow*	`ltl T_S2 {[] (ns_green -> ew_red)}`
$\square(ew\_green \rightarrow ns\_red)$	Yes. Same as before, Only one road may be active at a time	Yes. Same as before, it may be violated if one light is *green* and the other is *yellow*	`ltl T_S3 {[] (ew_green -> ns_red)}`
$\square \neg(ns\_yellow \wedge ew\_yellow)$	Yes. During *double-yellow*, vehicles from both roads may enter the intersection simultaneously. This must be prohibited in the system.	No. The system should never allow vehicles from both roads simultaneously.	`ltl T_S4 {[] (!(ns_yellow && ew_yellow))}`
Liveness properties			
$\square\diamond\, ns\_green \wedge \square\diamond\, ew\_green$	Yes. It ensures that both roads will have access.	No. It is necessary. Otherwise, one road would never have access.	`ltl T_L1 {([] <> ns_green) && ([] <> ew_green)}`
$\square\left(ns\_green \rightarrow \diamond\, ew\_green\right)$	Yes. Similar to the previous one, it ensures that both roads will have access.	No. An alternation phase is required in a traffic light system.	`ltl T_L2 {[] (ns_green -> <> ew_green)}`
$\square\left(ew\_green \rightarrow \diamond\, ns\_green\right)$	Yes. Similar to the previous one, it ensures that both roads will have access.	No. An alternation phase is required in a traffic light system.	`ltl T_L3 {[] (ew_green -> <> ns_green)}`
$\diamond\square\, ns\_green$	No. A traffic light should not stay green indefinitely, since that would keep the other direction red indefinitely.	Yes. Both lights should keep changing and not remain green indefinitely.	`ltl T_L4 {<> [] ns_green}`

3.5 SPIN Results

For each property analyzed and encoded above, evaluate the property against the provided automaton model using SPIN.

For each property:

(1) Insert the SPIN encoding of the property into the Promela model using the `ltl` construct.
(2) Run SPIN on the model.
(3) Record whether SPIN reports that the property *holds* or reports a *violation*.

If SPIN reports a violation, replay and inspect the counterexample trace generated by the model checker. Use this trace to understand how the system execution leads to the violation of the property.

For each property, report:

- the verification outcome (holds or violated),
- whether a counterexample trace was produced, and
- a brief explanation of what the trace shows.

When a violation is reported, you should also assess whether the observed behavior represents:

- a genuine defect in the system model, or
- a mismatch between the stated property and the modeled assumptions or intended design.

Your discussion should explicitly reference the behavior shown in the counterexample trace, rather than relying solely on intuition or informal reasoning.

3.5.1 Rule 1: (T-S1) No conflicting greens.

Rule: $\square \neg (ns\_green \wedge ew\_green)$

Verification outcome: Holds

Analysis and justification (model vs. property). The model successfully represents this property by strictly forbidding both lights from being green at the same time. This is achieved by setting both lights to red and allowing only one of them to turn green. This behavior is implemented in the `(phase == ALL_R)` transition.

3.5.2 Rule 2: (T-S2) If NS is green, then EW is red.

Rule: $\square (ns\_green \rightarrow ew\_red)$

Verification outcome: Holds

Analysis and justification (model vs. property). The model successfully represents this property: for green lights, only one state may be active at a time, either `NS_G` or `EW_G`. These states are mutually exclusive, allowing only one green light at a time.

3.5.3 Rule 3: (T-S3) If EW is green, then NS is red.

Rule: $\square (ew\_green \rightarrow ns\_red)$

Verification outcome: Holds

Analysis and justification (model vs. property). This is similar to the previous state, the model successfully represents this property: for green lights, only one state may be active at a time, either `NS_G` or `EW_G`. These states are mutually exclusive, allowing only one green light at a time.

3.5.4 Rule 4: (T-S4) No simultaneous yellow.

Rule: $\square\, \neg(ns\_yellow \wedge ew\_yellow)$

Verification outcome: Violation

Analysis and justification (model vs. property). The studied model includes a state that explicitly sets both lights to yellow (the state `BOTH_Y` which clearly violates the property). This appears to be an intentional component of the original design that does not correspond to the behavior of a real-world traffic light system.

Fix proposed: Remove the state `BOTH_Y` and create two new states `NS_G_and_SW_Y` and `SW_G_and_NS_Y`. In this way, both lights cannot be yellow at the same time.

3.5.5 Rule 5: (T-L1) Each direction eventually gets a green.

Rule: $\square\Diamond\, ns\_green \wedge \square\Diamond\, ew\_green$

Verification outcome: Violation

Analysis and justification (model vs. property). This property is clearly violated when we observe the self-transitions $NS\_G \rightarrow NS\_G$ and $EW\_G \rightarrow EW\_G$. This leads to the possibility that a green light remains permanently green, completely preventing the other direction from turning green.

Fix proposed: To achieve this, completely remove the self-transitions $NS\_G \rightarrow NS\_G$ and $EW\_G \rightarrow EW\_G$, and use the previously proposed intermediate states $NS\_G\_and\_SW\_Y$ and $SW\_G\_and\_NS\_Y$ to represent transitions from green states.

3.5.6 Rule 6: (T-L2) If NS is green, then eventually EW will be green.

Rule: $\square\, (ns\_green \rightarrow \Diamond\, ew\_green)$

Verification outcome: Violation

Analysis and justification (model vs. property). The analysis for this is exactly the same as the previous one. This property is clearly violated when we observe the self-transitions $NS\_G \rightarrow NS\_G$ and $EW\_G \rightarrow EW\_G$. This leads to the possibility that a green light remains permanently green, completely preventing the other direction from turning green.

Fix proposed: To achieve this, the analysis is similar to the previous one, completely remove the self-transitions $NS\_G \rightarrow NS\_G$ and $EW\_G \rightarrow EW\_G$, and use the previously proposed intermediate states $NS\_G\_and\_SW\_Y$ and $SW\_G\_and\_NS\_Y$ to represent transitions from green states.

3.5.7 Rule 7: (T-L3) If EW is green, then eventually NS will be green.

Rule: $\square\, (ew\_green \rightarrow \Diamond\, ns\_green)$

Verification outcome: Violation

Analysis and justification (model vs. property). The analysis for this is exactly the same as the previous ones. This property is clearly violated when we observe the self-transitions $NS\_G \rightarrow NS\_G$ and $EW\_G \rightarrow EW\_G$. This leads to the possibility that a green light remains permanently green, completely preventing the other direction from turning green.

Fix proposed: To achieve this, the analysis is similar to the previous ones, completely remove the self-transitions $NS\_G \rightarrow NS\_G$ and $EW\_G \rightarrow EW\_G$, and use the previously proposed intermediate states $NS\_G\_and\_SW\_Y$ and $SW\_G\_and\_NS\_Y$ to represent transitions from green states.

3.5.8 Rule 8: (T-L4) NS eventually stabilizes in a recurring green state.

Rule: $\Diamond\Box$ *ns_green*

Verification outcome: Violation

Analysis and justification (model vs. property). This property is violated, and it should indeed be violated in a correct design. If one of the lights in a traffic light system remains permanently green, it implies that the other road will never have the opportunity to pass, which is unintended behavior in a traffic light system.

Fix proposed: In this scenario this property is not well-designed because the light should never stabilize in a state, we can modify the property to the opposite as follows:

(T-L4') Never NS eventually stabilizes in a recurring green state.

$$!(\Diamond\Box\ ns\_green)$$