6 Specifying requirements

Overview In the previous chapters, we introduced models for cyberphysical systems that define their behaviors. The rest of this book is about techniques for verifying whether those behaviors are correct. To proceed with verification, one has to first *define* what it means for a behavior to be "correct". This is the subject of the current chapter. The definition of correct behavior usually comes from what are called the *design requirement documents* for the system or the product. These documents describe what the system does—capabilities, functions, operations—and may also contain information on how the system works, how it should interact with users, how it should be maintained, etc. We begin the chapter introducing *requirements analysis* which is the human process of pinning down the requirements. In the first part of this chapter, we discuss existing safety standards that are written informally, but often drive the safety-critical requirements of cyberphysical systems. In the second part, we present formal requirements in terms of safety, liveness, and more general temporal logic statements. In between, in section 6.3, we discuss the roles of human and computational resources in the formal verification process.

6.1 Requirements analysis

Requirements analysis for a product or a component is a set of tasks that ultimately lead to determination and documentation of the *design requirements* that the product must meet. Requirements are sometimes referred to as *technical specifications*, *specifications*, and *functional attributes*. For example, "0 to 60 mph in 2.8 seconds" is an example of a high-level specification for an electric vehicle. For a more realistic example, the 416 pages long software requirements document for the A-7 E aircraft by. Alspaugh et al. (1992),x will be instructive.

As products have many stakeholders—users, designers, regulators, and maintainers—they often have conflicting needs and incentives. Therefore, requirements analysis is an iterative human process for eliciting input from stakeholders, analyzing use cases, cross-validating possibly conflicting requirements, and then documenting the requirements in a contract. Beyond the core hardware and software functionalities (a.k.a. *behavioral requirements*), requirements can cover aspects such as performance, user interfaces, energy efficiency, environmental impact (such as emissions), and cost-effectiveness. Here, we will not discuss this topic in any detail (for more information, see, for example, the book Rajan and Wahl (2013)), but we note some of the common challenges in using requirements for the purpose of verification.

First, requirements are typically written using a combination of natural language, flowcharts, and pseudocode. As a result, requirements are ambiguous and under-constrained. There are requirement specification languages (RSL) that reduce some of those ambiguities. For example, variants of temporal logics (see Sections 6.4.2 and 6.5) have been proposed for

mathematically defining the requirements of cyberphysical systems. Another alternative for avoiding ambiguities is to use natural language processing (NLP) tools that convert the requirements to a machine-readable, and therefore consistent and unambiguous, form (Kof (2004)). Adoption of RSLs and NLP tools has been limited in part because they involve a learning curve, and are often seen as impractical.

6.2 Safety standards

Safety standards provide guidelines and processes for developing safety-critical systems. For example, the Federal Aviation Administration (FAA) of the United States uses the DO-178C standard (RTCA (2011)) as guidance for determining and certifying the airworthiness of aviation software, and it is enforced as part of the Federal Aviation Regulations. ISO 26262 ISO is the relevant standard for functional safety of electronic and software components in road vehicles, but in contrast to the situation for aviation systems, its adoption is voluntary. Other safety standards include MIL-STD-882E, the Department of Defense Standard Practice for System Safety; FMVSS, the Federal Motor Vehicle Safety Standard; AUTOSAR, the Automotive Open System Architecture; EN 50126, on Reliability, Availability, Maintainability and Safety of Railway Applications (RAMS); MISRA C, the Guidelines for the Use of the C Language in Critical Systems; the IEC 62304 Standard for Medical Devices; and SOTIF, on Safety of the Intended Function (SOTIF).

Most safety standards classify system components or functions into certain *integrity levels* and give guidelines for developing and testing components in each integrity level. The classification of a component into an integrity level is typically based on analysis of the consequences of the component's failure or malfunction. It is important to remember that

most of the standards are descriptive but not prescriptive, and that they leave a lot to the discretion of the suppliers and system builders.

6.2.1 DO-178C

For example, DO-178C has five *assurance levels* for software modules, also known as *Design Assurance Levels (DAL)* (see Table 6.1). A component's level is determined from the safety assessment process and hazard analysis through examination of the effects of a failure condition in the system. The failure conditions are categorized by their effects on the aircraft, crew, and passengers. For example, at the two extremes, **Level A** is assigned for "Catastrophic Outcome," and **Level E** is assigned for "No Safety Effect."

The DAL level classification then establishes the rigor necessary to demonstrate compliance with DO-178C. For example, components that command, control, or monitor safety-critical functions are classified as **Level A**. The standard requires any **Level A** software to be tested to cover every statement, branch, and function call, and also to pass the so-called *Modified Condition Decision Coverage (MC/DC)* tests. Roughly, an MC/DC test suite requires that (i) each entry and exit point in the code be invoked, (ii) each decision take every possible value, and (iii) each condition in a decision take every possible value. For certain levels, DO-178C requires that the testing, verification, and validation be performed by a team that is independent of the software development team.

Dozens of commercial tools (e.g., MATLAB, Esterel, Cantata, VectorCAST, Rapita Systems, and CodeSonar) can support DO-178C certification in different ways, such as by applying formal verification. The DO-333 supplement of DO-178C identifies aspects of airworthiness certification process that pertains to the production of software using formal

DAL	Failure Condition	Code Coverage Requirements	ASIL
A	Catastrophic	MC/DC unit tests with independence, branch, functional, call, and statement coverage	-
B	Hazardous	Branch and statement coverage with independence, MC/DC highly recommended	ASIL D
C	Major	Statement coverage MC/DC unit tests, branch coverage recommended	ASIL B/C
D	Minor	Statement coverage branch coverage recommended	ASIL A
E	No safety effect		-

Table 6.1
Design Assurance Levels (DAL) of DO-178C and ASILs of ISO26262.

methods. For example, DO-333 requires the soundness of each formal analysis method to be documented. Implementation of tool soundness issues is addressed separately as part of the tool qualification process described in DO-330. Use cases for classical theorem proving, model checking, and abstract interpretation in D0-178C certification, in accordance with DO-333, appear in Cofer and Miller (2014); however, applications of hybrid system verification are currently absent.

6.2.2 ISO 26262

Much like DO-178C, the ISO 26262 standard classifies components into levels; in this case, they are four *Automotive Safety Integrity Levels (ASIL)*, which are based on exposure to issues that affect the controllability of the vehicle. Roughly, an ASIL level captures a more general notion of *risk*, and is expressed as:

$$Risk = (Probability\ of\ accident) \times (Expected\ loss\ in\ case\ of\ accident) \tag{6.1}$$

$$ASIL = (Exposure \times Controllability) \times Severity. \tag{6.2}$$

The standard divides probabilities of exposure into five classes, ranging from "Very low probability" (E1) to "High probability" (E4). For severity, it has four classes, from "No injuries" to "fatal injuries." For controllability (by the driver), it has four classes, from "Controllable " to "Difficult to control." The standard shows how these variables must be combined to determine the required ASIL for an electronic subsystem or component in the vehicle. For example, a component that must be relied upon in a situation that has a medium probability of occurrence, and is considered normally controllable but can result in life-threatening injuries, requires an ASIL of B.

It would be a misconception to think that an ASIL can be attributed to a device; rather, an ASIL can only be attributed to a functionality or a property. For example, it would not make sense to talk about "ASIL B LIDAR." In contrast, a valid requirement might be of the form "With ASIL B, it is assured that in 99% of the cases in which there is an object of

s specified dimension in a specified range in front of the sensor, then the sensor will report it in the object list."

The ASIL level classification establishes the rigor necessary to demonstrate compliance with ISO 26262, again, in a manner similar to that of DO-178C. Although the two standards are similar in spirit, the requirements for meeting ASIL and DAL levels are not precisely comparable. ISO 26262 specifies testing requirements both at the unit level and at the architectural level, and there is no requirement for independence. Also, although MC/DC testing is highly recommended at the highest level (ASIL D), the requirement for how thats achieved is different from the requirements in DO-178C. Table 6.1 provides a rough alignment between DAL levels and ASIL levels.

Several case studies have been published that show how testing and formal verification can be applied for ISO 26262 compliance and risk analysis (Altinger et al. (2014); Rana et al. (2013)). A simulation-based analysis method for Automatic Emergency Braking (AEB) systems is proposed in Fabris (2012). More recently, that analysis has been recreated more efficiently using data-driven verification (Fan et al. (2018)). The broad idea is as follows. The question of whether or not the (automatic) braking profile of a sequence of cars on the highway is safe depends on several parameters: initial separation, initial speeds, vehicle dynamics, reaction times, road surface, etc. The analysis determines (through simulation or formal analysis) whether a given braking profile is safe for a set of scenarios characterized by the above parameters. That determination is then combined with statistical information about the distributions of the parameters (for example, from road traffic camera data) to obtain the probability of accidents. In the case of unsafe scenarios, the worst-case relative velocity of the collision is computed; it serves as a proxy for the sever-

ity of the accident. By combining the probability and the severity, one determines the overall risk associated with the braking profile. That type of analysis can be used as a design tool for tuning the braking profiles for different highway speeds, road conditions, etc.

6.2.3 Beyond current safety standards and requirements

Safety standards and requirements are useful for dealing with many potential design and implementation defects. As of early 2020, many areas of autonomous systems are not appropriately covered by existing standards Koopman and Wagner (2016). This realization is driving research activities and the creation of new standards and specifications that go beyond ISO 26262 and DO-178C. One such emerging standard for highly automated driving is Safety of the Intended Functionality (SOTIF) (ISO/WD (2018)). SOTIF accommodates the statistical correctness of functionality, such as in image-based object detection, and it covers unsafe situations that arise outside of hardware failures.

Responsibility Sensitive Safety (RSS) is another emerging model for safety introduced by Shalev-Shwartz et al. (2017b). RSS formalizes what it means for an autonomous vehicle to drive safely on its own and how it should exercise reasonable caution to protect against the unsafe driving behavior of others. RSS aims to satisfy the need for sound, useful (i.e., not overly conservative), and verifiable driving policies.

Autonomous systems that use test-driving data to train machine learning (ML) functions, such as deep neural networks (DNN), are also beyond the current safety standards. There is intense ongoing research on topics related to specification and verification of machine learning algorithms, and autonomous systems built with those ML algorithms. We briefly

point to some of the ongoing activities in Section 11.8.5, but dust is yet to settle, for us to venture out into these topics in this book.

6.3 From requirements to verification

Formally, a requirement defines a predicate over the set of system behaviors. It defines which behaviors are allowed and which ones are forbidden. More generally, for example for stochastic models, it makes sense to have quantitative requirements that define metrics over probability distributions of behaviors. For example, a requirement could be that the expected fuel efficiency of a vehicle model is within some range, for a given set of driving conditions. In this book, we will stick to the simpler binary requirements. In the rest of this chapter, we will discuss *formal* or mathematically precise requirements. Automatic tools need the requirements to be represented in a formal and machine-readable format. There are many such formats that cater to different modeling formalisms, application domains, and analysis approaches. Instead of getting into tool-specific formats, in the rest of this section, we discuss the dominant mathematical notions that define allowed system behaviors. Unless otherwise stated, our model of the system in question could be a discrete, continuous, or a hybrid automaton. The rest of the book is about verification techniques. At this point, we make a few remarks about algorithms, and the roles they play, in the overall verification and validation enterprise.

6.3.1 Formal verification algorithms

First and foremost, the goal of a verification process is to correctly check whether a given cyberphysical system meets a given requirement. Secondly, as users and developers of

verification techniques, we would prefer for the verification process to use resources optimally. More about resources and optimality in Section 6.3.2, but first let us define what it means for a verification process to be correct.

Recall that for an automaton $\mathcal{A}$, the set of all possible behaviors or executions is denoted by $\mathsf{Execs}_{\mathcal{A}}$ (see Sections 2.4, 3.4, and 4.4). A requirement R, therefore, can be seen as a subset of $\mathsf{Execs}_{\mathcal{A}}$. An execution $\alpha \in \mathsf{Execs}_{\mathcal{A}}$ *meets the requirement* R if $\alpha \in R$; otherwise, it is said to *violate* the requirement. An execution $\alpha \in \mathsf{Execs}_{\mathcal{A}}$ that violates a requirement R is called a *counterexample to* R.

In an ideal world, verification would be a fully automatic process carried out by an algorithm executing on a computer. Such a verification algorithm Alg would come with a warranty to work for a *class of models* and a *class of requirements*. Given an input model $\mathcal{A}$ from that model class and a requirement R from that class of requirements, Alg would decide whether *all* executions of $\mathcal{A}$ meet the requirement R. Indeed, this is a *decision problem* (defined in Appendix B.2.1). That is, Alg would produce one of two kinds of outputs: (a) if $\mathcal{A}$ violates R, then Alg finds a counterexample execution $\alpha \in \mathsf{Execs}_{\mathcal{A}} \setminus R$ that witnesses the violation; or (b) Alg gives a proof establishing that $\forall\ \alpha \in \mathsf{Execs}_{\mathcal{A}}, \alpha \in R$. Counterexamples can help designers fix design bugs, help product managers revise operating conditions and usage models, and help customers redefine requirements R. A proof establishing that $\mathcal{A}$ meets the requirement R can be used in the certification process (for example, for D0-178C); proofs can also be useful for explaining the correctness of the design of $\mathcal{A}$.

For model and requirement classes with such warrantied verification algorithms, we can happily shift our attention to investigating the *optimality* of the algorithm in terms of com-

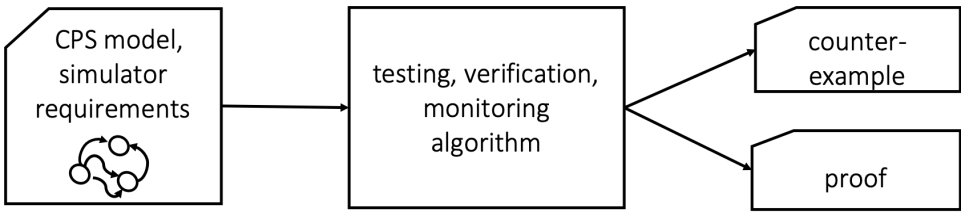

Figure 6.1

A verification algorithm takes as input an automaton $\mathcal{A}$ from a class of models and a requirement R from a class of requirements, and either gives a proof establishing that all behaviors of $\mathcal{A}$ meet R, or gives a particular behavior of $\mathcal{A}$ that violates R.

putational resource usage. For example, checking invariant requirements for the class of integer timed automaton, presents one such ideal situation, as we shall see in Chapter 9.

But, for general cyberphysical system mode classes, the situation is *known* to be grimmer. There is no perfect algorithm that always works. We will see an example of this in Section 9.5. The verification enterprise has to rely on imperfect algorithms that may be wrong—missing bugs or giving false alarms, that may not terminate on other inputs, or worse. In such situations, the verification team has to navigate the complex trade-offs between the expressive power of the model class and the precision and computational efficacy of the available imperfect verification algorithm (or tool) for that class.

A verification algorithm Alg is said to be *sound* if it answers the verification query correctly. That is, when it gives a proof, the proof is valid, and indeed all behaviors of $\mathcal{A}$ meet the requirement R; and when it gives a counterexample behavior, then the counterexample is indeed an actual behavior of $\mathcal{A}$ that violates R. An algorithm is said to be *complete* if it is guaranteed to terminate for all $\mathcal{A}, R$ inputs. Most of the verification algorithms and

techniques we discuss in the later chapters are sound. While some, like the reachability analysis algorithms of Chapter 9 and the data-driven verification algorithms of Chapter 11, are fully automatic, others, like the invariance and termination verification techniques of Chapters 7 and 10 can be partially automated with manual inputs.

6.3.2 Resource usage for verification and computational complexity

Even with perfect algorithms, the verification and validation enterprise requires human effort in (a) encoding the requirements and the models in a verification tool; (b) *validating* the models—which means checking that the models indeed correspond to the real system that has been designed or prototyped. (c) Effort is also needed in interpreting the output results of the verification tool. Usually, the models and requirements have to be manually refined several times, in a closed-loop iterative process, before the desired results are achived. As it happens, there is no agreed upon definition for precisely accounting for these human efforts.

For algorithms, on the other hand, the resources concerned are memory usage, computing cycles, bits needed for communication, etc.—things that can be measured. For any given algorithm Alg, the amount of resource (e.g., number of CPU cycles) used will obviously depend on its inputs $\mathcal{A}$ and R. Alg will require more cycles to verify a bigger $\mathcal{A}$ than a smaller one. Therefore, to study the efficiency of Alg, it makes sense more sense to see how the cycles used by Alg scales with the size of the inputs, rather than fixate on the actual number of cycles used by a particular input.

The computer science view of measuring resource usage is to count resources (number of computational steps, bits of memory, bits sent, random bits drawn, etc.) used by a Turing

machine representation of Alg. Turing machines and computational complexity concepts are reviewed in Appendix B.2. Owing to the *complexity theoretic Church-Turing thesis*, the actual resources used by Alg (running on any real computer, implemented with any programming language, operating system, etc.) is proportional to the resources used by this Turing machine representation. Since we are mainly interested in how the resource usage scales with the size of the input, we can rigorously derive the worst case resource usage of Alg—up to a constant factor—from any representation (e.g., pseudocode used in this book) without writing down a Turing machine.

It is worth noting that representing a real-valued output, even from a small input, may require infinite number of bits. Real-valued calculations are abound in verification algorithms for cyberphysical systems. This can complicate the Turing machine-based discipline of measuring complexity. One practical workaround is to use finite precision representations of real numbers, but this requires careful analysis of the of propagation of errors. A branch of complexity theory that using more powerful machines that can represent real numbers has been developed by Blum et al. (1997, 1988). These developments are theoretical and the connections with verification are yet to be drawn.

In contrast, the control theorists take a more abstract mathematical view of resource usage, without worrying too much about how the objects being computed are represented in a computer. For example, the resource usage of two algorithms would be compared in terms the number and dimension of the mathematical operations (e.g., matrix multiplication, constraint solving, optimization, etc.) that each of them invokes.

Finally, we remark that what can be called an "efficient" algorithm depends on the context. The commonly held interpretation, that a problem is efficiently solvable if there is

a polynomial time algorithm for solving it, does not make sense in some contexts. For example, in the context of data science problems, the inputs are huge and polynomial time algorithms would be uselessly inefficient. A logarithmic time algorithm or at least a sub-linear time algorithm would be considered efficient. In contrast, many important verification problems are known to be undecidable, and for such problems, even an exponential time algorithm for a relaxation of the problem might be considered a good start.

6.3.3 Invariants and safety requirements

By far the most common requirements are invariants or *safety requirements* (also known as *safety properties*). The idea of invariance for cyberphysical systems or programs comes from generalization of fundamental conserved quantities, like energy or momentum in physics—quantities that do not vary in a closed system. Roughly, an invariant asserts *requirements that must always hold*. They capture the idea that "some things *always* hold" or equivalently that "bad things *never* happen."

For an automaton $\mathcal{A}$ with a set of variables V and state space $val(V)$, a *candidate invariant* I is a subset of $val(V)$. A candidate invariant can be equivalently expressed as a predicate over V. A candidate invariant I is an invariant if all states along all executions of $\mathcal{A}$ satisfy I:

$$\forall \alpha \in \mathsf{Execs}_{\mathcal{A}}, \forall\, t, \alpha(t) \in I. \tag{6.3}$$

More generally, a *safety requirement* is defined by Alpern and Schneider (Alpern and Schneider (1987)) as a set $S \subseteq \mathsf{Execs}_{\mathcal{A}}$ such that for any $\alpha \in \mathsf{Execs}_{\mathcal{A}}$, $\alpha \in S \Leftrightarrow \forall\, \beta \in \mathsf{Frags}_{\mathcal{A}} : \beta \leq \alpha \Rightarrow \beta \in S$. That is, a safety requirement S is a requirement

such that if α satisfies S then so does any prefix of α. Invariant requirements are a popular subclass of safety requirements. Invariant requirements can be alternatively defined as a set $I \subseteq val(V)$ such that the reachable states of $\mathcal{A}$ (from a given set of initial states $\Theta \subseteq val(V)$) are contained in I, that is, $\mathsf{Reach}_{\mathcal{A}} \subseteq I$.

For the Dijkstra's token ring algorithm of 2.5, the statement "the system *always* has a single token" represents a candidate invariant that is the set of all the states in which the system has a single token. This set can be equivalently written as the predicate ϕ_{legal} (Section 2.5.1). As we saw in Section 2.5, the candidate invariant ϕ_{legal} is indeed an invariant if all initial states of DijkstraTR have a single token. The driving safety requirement "unless a turn signal is given, a car always stays inside lanes" can be expressed by the predicate:

$$\neg turnSignal \Rightarrow leftLane \leq x \leq rightLane,$$

where x is the lateral position of the car and *leftLane* and *rightLane* are positions of the left and right lane markers.

Invariant properties can also be specified negatively in terms of a bad thing or an unsafe thing that should *never* happen. An example of such a requirement in a traffic intersection would be, "The lights on intersecting lanes should *never* be green simultaneously." The invariant requirement in that case is indirectly specified by the set of bad states or unsafe states that should not be reached.

Exercise 6.1. Write five traffic safety rules as invariants. Give the English statement and the corresponding predicate over the involved state variables.

Exercise 6.2. If $I_1, I_2 \subseteq val(V)$ are invariants of $\mathcal{A}$, then show that $I_1 \cup I_2$ and $I_1 \cap I_2$ are also invariants.

Proposition 6.1. For any candidate invariant (or safety) requirement I, if $\mathcal{A}$ violates I, then there exists a finite witnessing counterexample.

Proof. Suppose $\mathcal{A}$ violates the candidate invariant I. Then, there exists a reachable state $\mathbf{v} \in \mathsf{Reach}_{\mathcal{A}}$ that is $\mathbf{v} \notin I$. That is, there exists a finite execution α that ends in $\mathbf{v}$. This α is a finite witnessing counterexample. □

A natural weakening of an invariant requirement is a *bounded invariant* or a *bounded safety requirement*. A candidate invariant $I \subseteq val(V)$ is a *bounded invariant up to time* T, for some time bound $T \geq 0$, provided the states reachable *within time* T are contained in I. In other words, $\mathsf{Reach}_{\mathcal{A}}(\Theta, T) \subseteq I$. The unbounded time invariant verification problem is usually hard if not impossible to solve exactly for most classes of models. That is the reason why recent research has concentrated around the more practical bounded time verification problem. We will discuss techniques for invariant verification in Chapters 9, 11, and 7.

6.3.4 Progress requirements

The second most common type of requirement, after invariants, is *progress requirements*. Roughly, a progress requirement captures the idea that "something good *eventually* happens." Achievement of safety requirements can be trivial, unless the system also has to meet some progress requirements.

Progress requirements come in several flavors. For an automaton $\mathcal{A}$ with a set of variables V and state space $val(V)$, a simple type of progress property can be specified by a subset $P \subseteq val(V)$. Automaton $\mathcal{A}$ is then said to meet this progress requirement (from a given set of initial states $\Theta \subseteq val(V)$) if every execution of $\mathcal{A}$ eventually reaches P. That

is,

$$\forall \alpha \in \mathsf{Execs}_{\mathcal{A}}, \exists\, t, \alpha(t) \in P. \tag{6.4}$$

The above is akin to *termination* of programs. More generally, a *liveness requirement* is defined in Alpern and Schneider (1987) as a set $L \subseteq \mathsf{Execs}_{\mathcal{A}}$ such that $\forall\, \beta \in \mathsf{Frags}_{\mathcal{A}}, \exists \beta' \in \mathsf{Frags}_{\mathcal{A}}, \beta \frown \beta' \in L$. That is, every finite execution can be extended (possibly by an infinite suffix) to meet such a requirement. Termination is a particularly familiar liveness requirement.

A stronger progress requirement would be to require every execution fragment of $\mathcal{A}$ eventually to reach P, regardless of the starting state. The *stabilization* property is an example of this type of requirement: after failures, the system can end up in *any* state, but then it recovers within finite time, that is, it comes back to a legal state. In Dijkstra's token ring algorithm, the stabilization requirement says that "the system *eventually* comes back to a state with a single token," and the predicate ϕ_{legal} defines this set (see Section 2.5). For automata on metric spaces, a more meaningful version of progress is captured by *asymptotic stability*. It requires all executions of $\mathcal{A}$ to converge to P as time goes to infinity. For a vehicle control system, for instance, it makes more sense to converge towards a waypoint than to hit the waypoint precisely. The reader may find it helpful at this point to revisit Sections 3.4 and 4.4.2 for the related definitions of global and local stability.

The above requirements do not say anything about how soon progress to P is achieved. A stronger version of progress bounds the time within which P must be achieved:

$$\forall \alpha \in \mathsf{Execs}_{\mathcal{A}}, \exists\, t \leq T_P, \alpha(t) \in P. \tag{6.5}$$

Here, T_P is a uniform upper bound within which P is achieved. Other versions of bounded progress allow T_P to be a function of the initial state of α. Bounded progress is actually an invariant in disguise. To make that clear, we rewrite Equation (6.5) as:

$$\forall \alpha \in \mathsf{Execs}_{\mathcal{A}'}, \forall\, t, \alpha(t) \lceil\ timer \leq T_P \ \vee\ \alpha(t) \lceil\ (V \setminus \{timer\}) \in P, \tag{6.6}$$

where $\mathcal{A}'$ is an augmented version of automaton $\mathcal{A}$ with a *timer* variable. The invariant predicate here says that either $timer \leq T_P$ or the state of $\mathcal{A}'$ (without *timer*) is in P.

Exercise 6.3. Write five traffic rules that are progress requirements. Give the English statement and the corresponding predicate over the involved state variables.

Exercise 6.4. Construct a finite automaton $\mathcal{A}$ and two progress predicates P_1 and P_2 such that $\mathcal{A}$ meets the progress requirements P_1 and P_2, but not $P_1 \wedge P_2$.

A counterexample for a progress requirement P is an *infinite* execution α such that $\forall\ t, \alpha(t) \notin P$. Alternatively, a pair of finite execution fragments α_1, α_2 can represent an infinite counterexample. For this counterexample we require that (i) $\alpha_1.\mathsf{fstate} \in \Theta$, (ii) $\alpha_1.\mathsf{lstate} = \alpha_2.\mathsf{fstate} = \alpha_2.\mathsf{lstate}$, and (iii) for all $t \in \alpha_1.\mathsf{dom}$, $\alpha_1(t) \notin P$ and for all $t \in \alpha_2.\mathsf{dom}$, $\alpha_2(t) \notin P$. Then we define $\alpha = \alpha_1 \frown \alpha_2 \frown \alpha_2 \frown \ldots$. The process for checking that α is a valid execution of $\mathcal{A}$ and that it violates the requirement P is straightforward. That type of counterexample, which has a finite initial execution followed by an infinitely repeating fragment, is called a *lasso*. We will discuss techniques for verification of progress properties in Chapter 10.

6.4 Linear temporal logic

Temporal logics are a family of formal languages for succinctly specifying complex requirements. For example, temporal logic formulas can express requirements such as "after the walk button is pressed, the red light and the walk sign eventually turn on" and "after a failure, the system may exit the safety envelope S, but it eventually re-enters and remains in S." The term *temporal* can be misleading here, as these requirements have nothing to do with time—at least, not in the sense of real-time in timed and hybrid automaton models[2]. *Temporal* refers to the sequential ordering of certain actions or predicates in the execution of the automaton in question. At the top level, there are two classes of temporal logics: (a) branching logics, which allow quantification over executions; and (b) linear logics, which do not. We will introduce Computational Tree Logic (CTL) and Linear Tree Logic (LTL) as prominent members of these two classes. Temporal logics can be discussed in the context of different types of automata—discrete state, timed, and hybrid; here, we focus on discrete models.

6.4.1 Background definitions

Recall Definition 2.1: An automaton is defined as a tuple $\mathcal{A} = (V, \Theta, A, \mathcal{D})$ where V is the set of variables, Θ is the set of initial states, A is the set of actions or transition labels, and $\mathcal{D}$ is the set of labeled transitions. In the discussion of temporal logics, it is usual to label states instead of transitions.

[2] Real-time extensions of temporal logics, such as metric temporal logic (MTL) (Koymans (1990a)), are not covered in any detail in this book.

Atomic propositions Let AP be a set of *atomic propositions*. Each $p \in AP$ is a basic (atomic) property or requirement that we care about. Examples of atomic propositions for the mutual exclusion protocol of Section 2.5 are p_1 ("only one process has a token"), p_2 ("process 2 does not have a token"), and p_3 ("process 2 has value k"). A *labeling function* AP assigns to each state $\mathbf{v} \in val(V)$ a set of atomic propositions that hold in $\mathbf{v}$. The labels for each state can be enumerated explicitly in the case of a finite state system or else they have to be defined symbolically. For example, for any state of the mutual exclusion protocol $\mathbf{v} \in val(V)$, $p_3 \in Lab(\mathbf{v})$ if and only if $\mathbf{v}.x[2] = k$.

Automaton with state labels If we replace the transition labels with state labels in Definition 2.1, we get the following variant of the discrete automaton model.

Definition 6.1. A *labeled transition system (LTS)* $\mathcal{A}$ is a tuple $(V, \Theta, Lab, \mathcal{D})$, where

(a) V is set of state variables.

(b) $\Theta \subseteq val(V)$ is a nonempty set of *start states*.

(c) $Lab : val(V) \to 2^{AP}$ is a labeling function that assigns each state a set of atomic propositions.

(d) $\mathcal{D} \subseteq val(V) \times val(V)$ is the set of *transitions*.

This type of automaton is also called a *Kripke structure*. A finite state example is shown in Figure 6.2.

Since a LTS does not have transition labels, we define executions of $\mathcal{A}$ to be a sequence of states. An *execution fragment* or *run* of $\mathcal{A}$ is a (finite or infinite) sequence $\alpha = \mathbf{v}_0, \mathbf{v}_1, \ldots$, such that for all i, $(\mathbf{v}_i, \mathbf{v}_{i+1}) \in \mathcal{D}$. Given an execution α, we will use the notation $\alpha[k]$ to denote the k^{th} state $\mathbf{v}_k$ in the sequence. An execution fragment for which $\alpha[0] \in \Theta_0$ is called

an *execution*. The set of all executions (fragments) of $\mathcal{A}$ is denoted by $\mathsf{Execs}_{\mathcal{A}}$ ($\mathsf{Frags}_{\mathcal{A}}$). The set of executions (fragments) starting from a given state $\mathbf{v} \in val(V)$ is denoted by $\mathsf{Execs}_{\mathcal{A}}(\mathbf{v})$ ($\mathsf{Frags}_{\mathcal{A}}(\mathbf{v})$).

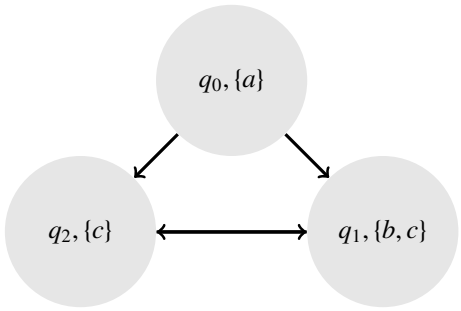

Figure 6.2

The states q_1, q_2, q_3 have labels $\{a\}, \{b, c\}$, and $\{c\}$, respectively.

6.4.2 LTL syntax

An LTL formula is an expression built from atomic propositions, Boolean connectives, and special temporal operators. There are five temporal operators: Next (**X**), Eventually (**F**), Always (**G**), Until (**U**), and Release (**R**). The syntax of an LTL formula f is given by the following grammar:

$$\begin{aligned} f \quad := \quad & true \mid p \mid \neg f_1 \mid f_1 \wedge f_2 \mid f_1 \vee f_2 \mid \\ & \mid \mathbf{X}\, f_1 \mid \mathbf{F}\, f_1 \mid \mathbf{G}\, f_1 \mid f_1\, \mathbf{U}\, f_2 \mid f_1\, \mathbf{R}\, f_2, \end{aligned}$$

where $p \in AP$ is an atomic proposition, and f_1, f_2 are LTL (sub-)formulas. Other logical operators, like $\Rightarrow$, $\Leftrightarrow$, and **xor**, though not included in the above grammar, can be expressed in terms of $\neg$ and $\wedge$, and therefore can be used to connect LTL sub-formulas. Some authors

use $\Box$ and $\Diamond$ to denote the Always and Eventually operators, respectively. Some examples of syntactically correct LTL formulas are $\mathbf{F}\ p_1$ ("eventually only one process has a token"), $\mathbf{FG}\ p_1$ ("eventually always p_1"), $p_1 \Rightarrow \mathbf{X}\ p_1$ ("if p_1, then the next step satisfies p_1"), and $p_2 \Rightarrow \Diamond \neg p_2$ ("if process 2 holds the token then eventually it does not").

6.4.3 LTL semantics

Broadly, any temporal logic formula or requirement R defines a set $[\![R]\!]$ of signals or sequences that satisfy R. The semantics of LTL is defined in terms of sequences labeled with atomic propositions. Here, we will define the semantics of LTL as sets of executions of an automaton $\mathcal{A}$. Given a state $\mathbf{v}$ and an LTL formula f, we write $\mathbf{v} \models_{\mathcal{A}} f$ if and only if all executions of $\mathcal{A}$ starting from state $\mathbf{v}$ *satisfy* f. When the underlying automaton $\mathcal{A}$ is clear from context, we write $\mathbf{v} \models_{\mathcal{A}} f$ as $\mathbf{v} \models f$. Since f is necessarily constructed from one or two LTL sub-formulas that use the above grammar, the definition of the relation $\models_{\mathcal{A}}$ ($\models$) will be inductive on the structure of the LTL formula f.

$$
\begin{array}{ll}
\mathbf{v} \models \mathit{true} & \iff \mathit{true} \\
\mathbf{v} \models p & \iff p \in \mathit{Lab}(q) \\
\mathbf{v} \models \neg f & \iff \mathbf{v} \not\models f \\
\mathbf{v} \models f_1 \wedge f_2 & \iff \mathbf{v} \models f_1 \textit{ and } \mathbf{v} \models f_2 \\
\mathbf{v} \models f_1 \vee f_2 & \iff \mathbf{v} \models f_1 \textit{ or } \mathbf{v} \models f_2 \\
\mathbf{v} \models \mathbf{X}\, f & \iff \forall \mathbf{v}' \in \mathit{val}(V), (\mathbf{v}, \mathbf{v}') \in \mathcal{D},\ \mathbf{v}' \models f \\
\mathbf{v} \models \mathbf{F}\, f & \iff \forall \alpha \in \mathsf{Frags}_{\mathcal{A}}(\mathbf{v}), \exists i \geq 0, \alpha[i] \models f_2 \\
\mathbf{v} \models \mathbf{G}\, f & \iff \forall \alpha \in \mathsf{Frags}_{\mathcal{A}}(\mathbf{v}), \forall i \geq 0, \alpha[i] \models f_2 \\
\mathbf{v} \models f_1\, \mathbf{U}\, f_2 & \iff \forall \alpha \in \mathsf{Frags}_{\mathcal{A}}(\mathbf{v}), \exists i \geq 0, \alpha[i] \models f_2 \ \wedge\ \forall j \leq i, \alpha[j] \models f_1 \\
\mathbf{v} \models f_1\, \mathbf{R}\, f_2 & \iff \forall \alpha \in \mathsf{Frags}_{\mathcal{A}}(\mathbf{v}), \exists i \geq 0, \alpha[i] \models f_2 \ \wedge\ \forall j \leq i, \alpha[j] \models f_1
\end{array}
$$

A few remarks about the definitions in the above box are needed. First, although here α is quantified over execution fragments of $\mathcal{A}$ starting from $\mathbf{v}$, a more general interpretation, independent of an automaton, is given in terms of infinite sequences of subsets of AP ($\alpha : \omega \to 2^{AP}$). In Section 6.5.1, we will see how branching temporal logics allow us to talk about *some* but not all executions starting from a state. A state $\mathbf{v}$ is defined to satisfy the LTL formula $\mathbf{X}\, f$ when *all* possible *next* states $\mathbf{v}'$ from $\mathbf{v}$ satisfy f, that is, for all $\mathbf{v}'$ such that $(\mathbf{v}, \mathbf{v}') \in \mathcal{D}$ is a valid transition, $\mathbf{v}' \models f$. A state $\mathbf{v}$ satisfies $f_1\, \mathbf{U}\, f_2$ if for *every* execution fragment α starting from $\mathbf{v}$, f_2 eventually becomes true, and until then f_1 holds. The $\mathbf{U}$ is also called the *strong until* operator to highlight the interpretation that f_2 does

indeed hold sometime in the future. In contrast, a *weak until* (**W**) operator does not require f_2 to occur. A state $\mathbf{v}$ satisfies f_1 **R** f_2 iff f_1 has to be true until and including the point when f_2 first becomes true; if f_2 never becomes true, then f_1 must remain true forever. An automaton $\mathcal{A}$ satisfies an LTL formula f if every execution, $\alpha \in \mathsf{Execs}_{\mathcal{A}}$ satisfies f. Thus, $[\![f]\!] = \{\alpha \in \mathsf{Execs}_{\mathcal{A}} \mid \alpha \models f\}$.

There is redundancy in the above set of temporal operators. It turns out that any LTL formula f can be translated into a *semantically equivalent* formula f' that uses only **X** and **U** operators. *Semantic equivalence* means that for any state $\mathbf{v}$, $\mathbf{v} \models f$ iff $\mathbf{v} \models f'$ and f' only uses the temporal operators **X** and **U**. For example, you can easily check the following:

$$\Diamond\, f \equiv \mathit{true}\ \mathbf{U}\ f$$

$$\Box\, f \equiv \neg(\mathit{true}\ \mathbf{U}\ \neg f).$$

A common LTL idiom is to write "always, eventually f", or $\Box\Diamond\, f$. It means that from any point in an execution, there is always a future point when f holds. That is equivalent to saying that f holds infinitely often. Another common idiom is $\Diamond\Box f$, which says that eventually f holds and remains true. That is equivalent to the stabilization property we mentioned in **??**.

Exercise 6.5. Write LTL formulas that capture the following traffic light properties. Use the atomic propositions r, y, and g, which correspond to the red, yellow, and green lights' being on.

1. Exactly one light is on at any given time step.
2. Eventually the green light turns on.

3. Always, the green light eventually turns on.
4. After red, there are at least three time steps of yellow, and then the light turns green.

Exercise 6.6. Prove the following identities.

1. $\Diamond\Diamond\, f \equiv \Diamond\, f$
2. $\Box\Box\, f \equiv \Box\, f$
3. $\neg\mathbf{X}\, f \equiv \mathbf{X}\, \neg f$
4. $\neg\Diamond\, f \equiv \Box\, \neg f$
5. $\neg\Box\, f \equiv \Diamond\, \neg f$
6. $f_1 \mathbf{U} f_2 \equiv f_2 \vee (f_1 \wedge \mathbf{X}(f_1 \mathbf{U} f_2))$
7. $\Diamond f_1 \equiv \Diamond f_1 \vee \mathbf{X}\, \Diamond f_1$
8. $\mathbf{X}(f_1 \mathbf{U} f_2) \equiv (\mathbf{X}\, f_1)\mathbf{U}(\mathbf{X}\, f_2)$
9. $\Diamond(f_1 \vee f_2) \equiv \Diamond f_1 \vee \Diamond f_2$
10. $\Box(f_1 \wedge f_2) \equiv \Box f_1 \wedge \Box f_2$

Exercise 6.7. Give counterexample automata for the following identities.

1. $\Diamond(f_1 \wedge f_2) \not\equiv \Diamond f_1 \wedge \Diamond f_2$
2. $\Box(f_1 \vee f_2) \not\equiv \Box f_1 \vee \Box f_2$

Semantics of LTL for discrete and hybrid automata A given automaton $\mathcal{A}$ *satisfies* an LTL formula if all execution fragments starting from every starting state Θ of $\mathcal{A}$ match the properties prescribed by the formula. That is, $\mathcal{A} \models f$ iff $\forall\ \mathbf{v}_0 \in \Theta$, $\mathbf{v}_0 \models f$. For the automaton in Figure 6.2, for $f_1 := \Diamond\Box c$, $\mathcal{A} \models f_1$, and for $f_2 := \Diamond\Box b$, $\mathcal{A} \not\models f_2$. In general, $\mathcal{A} \not\models f$ is not equivalent to $\mathcal{A} \models \neg f$, since it is possible that some executions of $\mathcal{A}$ satisfy f and others do not.

The above definition of the semantics of LTL can be applied, with a couple of modifications, to hybrid automata. First, the role of atomic propositions is taken by state predicates. Given a hybrid automaton $\mathcal{A} = (V, \Theta, A, \mathcal{D}, \mathcal{T})$, let $P_1, \ldots, P_k \subseteq val(V)$ be a set of predicates. Then, a state $\mathbf{v} \in val(V)$ is labeled P_i if $\mathbf{v} \in P_i$. Second, an execution α of $\mathcal{A}$ evolves over dense time, and there is no notion of next step or i^{th} step, as is needed in the above

definition of LTL semantics. The workaround is to *sample* α (Plaku et al. (2009); Cimatti et al. (2014)). Given an infinite, diverging, sampling time sequence $t_0, t_1, \ldots$ and an open execution α, we can define $\alpha[i] = \alpha(t_i)$. Using that sampled version of α, we can define $\alpha \models f$ much as it is defined for discrete automata above.

6.5 Computation tree logic (CTL)

Linear temporal logic (LTL) gives us a way to talk about linear or sequential requirements that have to hold in *every* execution of a system. In contrast, *computational tree logic (CTL)* allows us to describe requirements of computational trees. For example, we can state properties like "there exists an execution along which the aircraft returns to the safe holding zone."

6.5.1 CTL syntax

CTL has five *temporal operators*—Next (**X**), Eventually (**F**), Always (**G**), Until (**U**), and Release (**R**)—and two *path quantifiers*: the Existential quantifier (**E**) and the Universal quantifier (**A**). There are two types of formulas in CTL: *state formulas* are evaluated at a state, and *path formulas* are evaluated along state sequences (executions or paths). The syntax for state and path formulas is given by the following grammar:

$$\begin{aligned} \text{(state formula) } f &:= \mathit{true} \mid p \mid \neg f_1 \mid f_1 \wedge f_2 \mid f_1 \vee f_2 \mid \mathbf{E}\,\phi \mid \mathbf{A}\,\phi \\ \text{(path formula) } \phi &:= \mathbf{X}\,f_1 \mid \mathbf{F}\,f_1 \mid \mathbf{G}\,f_2 \mid f_1\,\mathbf{U}\,f_2 \mid f_1\mathbf{R}\,f_2 \end{aligned}$$

where $p \in AP$ is an atomic proposition, f_1, f_2 are state formulas, and ϕ, ϕ_1 are path formulas. Additional logical operators, like $\Rightarrow, \Leftrightarrow$, and **xor**, can be expressed in terms of $\neg$ and $\wedge$ to connect CTL state formulas. The depth of a CTL formula is the number of pro-

duction rules used in creating it. Some examples of valid CTL formulas and their depths are $\mathbf{EX}p$ (depth 3), $\mathbf{AX}(\mathbf{EX}p)$ (depth 5), and $(\mathbf{AX}(\mathbf{EX}p))\mathbf{U}q$ (depth 6). Note that $\mathbf{AXX}f$, $\mathbf{AEX}f$, and $(p\mathbf{U}q)\mathbf{U}p$ are not CTL formulas. The temporal operators must alternate with the path quantifiers in valid formulas. In other words, any occurrence of operators **U** and **X** is associated with **E** or **A**, and vice versa.

6.5.2 CTL semantics

As in the case of LTL, the semantics of CTL is defined in the context of sequences of labeled states, for example, executions of an automaton $\mathcal{A}$. Given a state $\mathbf{v}$ and a CTL state formula f, we write $\mathbf{v} \models f$ if and only if $\mathbf{v}$ *satisfies* f. Similarly, given a sequence of states α and a CTL path formula ϕ, we write $\alpha \models \phi$ if and only if α satisfies ϕ. The satisfaction relation for state formulas is defined inductively as follows.

$$
\begin{aligned}
&\mathbf{v} \models \mathit{true} && \iff \mathit{true}\\
&\mathbf{v} \models p && \iff p \in \mathit{Lab}(q)\\
&\mathbf{v} \models \neg f && \iff \mathbf{v} \not\models f\\
&\mathbf{v} \models f_1 \wedge f_2 && \iff \mathbf{v} \models f_1 \textit{ and } \mathbf{v} \models f_2\\
&\mathbf{v} \models f_1 \vee f_2 && \iff \mathbf{v} \models f_1 \textit{ or } \mathbf{v} \models f_2\\
&\mathbf{v} \models \mathbf{E}\,\phi && \iff \exists \alpha \in \mathsf{Frags}_{\mathcal{A}}(\mathbf{v}), \alpha \models \phi\\
&\mathbf{v} \models \mathbf{A}\,\phi && \iff \forall \alpha \in \mathsf{Frags}_{\mathcal{A}}(\mathbf{v}), \alpha \models \phi
\end{aligned}
$$

The satisfaction relation for path formulas is defined as follows.

$$
\begin{aligned}
\alpha \models \mathbf{X}\, f_1 &\iff \alpha[1] \models f_1 \\
\alpha \models \mathbf{F}\, f_1 &\iff \exists i \geq 0, \alpha[i] \models f_1 \\
\alpha \models \mathbf{G}\, f_1 &\iff \forall i \geq 0, \alpha[i] \models f_1 \\
\alpha \models f_1\, \mathbf{U}\, f_2 &\iff \exists i \geq 0, \alpha[i] \models f_2 \ \wedge\ \forall j \leq i, \alpha[j] \models f_1 \\
\alpha \models f_1\, \mathbf{R}\, f_2 &\iff \exists i \geq 0, \alpha[i] \models f_2 \ \wedge\ \forall j \leq i, \alpha[j] \models f_1
\end{aligned}
$$

As in the case of LTL, there is a minimal set of operators that is adequate for expressing any CTL formula. Specifically, any CTL formula f can be translated into a semantically equivalent formula f' that uses only the **E** quantifier and the temporal operators **X** and **U**.

Exercise 6.8. Convert the following CTL formulas to equivalent formulas that use only **E**, **X**, and **U**: **AX** f_1, **AF** f_1, **AG** f_1, **A**f_1 **U** f_2, **EF** f_1, **EG** f_1.

Exercise 6.9. Define finite state machines (Kripke structures) that satisfy each of these CTL formulas: **AG** f_1, **AF** f_1, **EG** f_1, **EF** f_1.

6.5.3 Expressiveness of LTL and CTL

Both CTL and LTL are languages for specifying the ordering of actions and predicates. It is natural to ask, which is more expressive? For any LTL formula f_1, does there exist a corresponding CTL formula that defines the same set of executions? Similarly, can any CTL formula be expressed as an LTL formula? It is easy to see that the last statement is

false. There is no LTL counterpart to the CTL formula **EF** f_1, because LTL does not allow path quantifiers. The following exercise shows that there are LTL formulas that cannot be expressed in CTL.

Exercise 6.10. Consider the finite state labeled transition system $\mathcal{A}$ defined as follows: (a) $val(V) = \{q_0, q_1, q_2\}$, (b) $\Theta = \{q_0\}$, (c) $Lab(q_0) = Lab(q_2) = p$, $Lab(q_1) = \neg p$, (d) $\mathcal{D} = \{(q_0, q_0), (q_0, q_1), (q_1, q_2), (q_2, q_2)\}$. Show that the LTL formula **FG** p is satisfied by all executions of $\mathcal{A}$. Is there a corresponding CTL formula that is satisfied by executions of $\mathcal{A}$? Note that **AFG** p is not a CTL formula; how about **AFAG** p?

From the above discussion, it follows that the expressive powers of LTL and CTL are incomparable. There is a temporal logic called CTL* that freely combines temporal operators and path quantifiers, and therefore has expressive power that includes those of both CTL and LTL.

6.6 Further reading

6.6.1 Temporal logic model checking

Given an LTL formula f, checking of whether $[\![f]\!] = \emptyset$ is called the *LTL satisfiability* problem. It is PSPACE-complete in the size of the formula f. Given an automaton $\mathcal{A}$ and an LTL formula f, we would like to check whether $\mathcal{A} \models f$. That is the LTL model-checking problem. To answer it, we check whether there exists an execution $\alpha \in \mathsf{Execs}_{\mathcal{A}}$ that satisfies $\neg f$. Thus, it suffices to be able to search for a path or execution that satisfies (the negation of) an LTL formula g. The original LTL model-checking algorithm of Lichtenstein and Pnueli (2000) uses the so-called *tableau construction*. The problem is

PSPACE-complete (the proof is quite technical), but one can easily show NP-hardness by reducing the Hamiltonian path problem to the above. Although the complexity is exponential in the size of the formula $|g|$, it is linear in $|\mathcal{D}|$, the size of the transition system. For detailed exploration of those ideas, we refer interested readers to the textbooks Clarke et al. (1999) and Baier and Katoen (2008). Temporal logic model checking is now part of several academic and commercial tools, such as NuSMV (Cimatti et al. (1999)), SPIN (Holzmann (1997)), and LTSmin (Kant et al. (2015)). These tools have been used successfully to verify a range of significant practical applications, from high-level descriptions of distributed algorithms and cache coherence protocols to detailed code for telephone exchanges, railway interlocking systems, and aircraft collision avoidance systems.

6.6.2 Planning and synthesis with temporal logics

LTL and CTL have been extensively used to specify requirements for the controller synthesis problem for cyberphysical and robotic systems (recall Definition 3.6). The problem as formulated by Kress-Gazit et al. (2009) is as follows: given (a) an LTL specification for the tasks of an agent (ψ_s), (b) a model of the agent's movements and actions, and (c) a temporal logic specification of how the environment reacts to agent's actions (ψ_e), construct (if possible) a controller so that the agent's resulting trajectories and actions satisfy the specification (ψ_s), in any environment satisfying ψ_e, from any possible initial state. A standard synthesis approach for this problem is based on first discretizing the environment of the agent. Constructing a discrete controller or an automaton that meets an LTL specification is a well-studied problem in computer science and Pnueli and Rosner (1989) show that generally the problem is doubly exponential in the size of the formula. However, by re-

stricting the specification to a special subclass of LTL called *generalized reactivity GR(1)*, one can use the algorithm of Piterman et al. (2006) which is polynomial in the size of the state space. Kress-Gazit et al. (2009) take this approach to the above synthesis problem and then uses low-level continuous state space controllers to implement the discrete transitions coming from Piterman et al. (2006). In order to gain computational tractability, variants of this approach exploiting specific properties of the environment such as planarity, restricted types of dynamics, and restricted types of specifications have been developed by Kloetzer and Belta (2008); Tabuada (2008); Sadraddini and Belta (2019); Wongpiromsarn et al. (2011); Raman et al. (2015) and several others.

6.6.3 Dense time, signal, and stochastic temporal logics

There are several variants of temporal logics and corresponding verification algorithms. *Timed CTL (TCTL)* adds a timed until operator, clock variables, and clock constraints in formulas, and thus enables reasoning about clocks and dense time in timed automata. TCTL is the logic of choice for the UPPAAL model-checking tool (Bengtsson et al. (1996)). *Metric Temporal Logic (MTL)* is a linear logic in which the modalities of LTL are augmented with timing constraints (Koymans (1990b)). For example, $\Box(\mathit{walkButton} \implies (\Diamond_{(0,10)}\mathit{lightRed}))$ in MTL means that *lightRed* will become *true* sometime in the next 10 time units when *walkButton* holds. The verification problem for timed automata with respect to MTL requirements has been shown to be undecidable (Alur et al. (1996)). That led to the creation of *Metric Interval Temporal Logic (MITL)* (Alur et al. (1996)); it forbids singleton or punctual intervals (for example, $\Diamond_{\{10\}}$), which are allowed in MTL.

Signal Temporal Logic (STL) introduced by Maler and Nickovic (2004) extends MTL further to include real-valued constraints. For example, the STL formula

$$\Box(x[t] < 5.5 \implies (\Diamond_{(0,10)} lightGreen))$$

allows the numerical predicate $x[t] < 5.5$ to play the role of atomic propositions. That generalization of the syntax has led several researchers to consider more continuous or quantitative semantics for temporal logics (Donzé and Maler (2010); Fainekos and Pappas (2009)). That is, instead of interpreting formulas to be merely *true* or *false*, we can consider the *distance* of a formula from being true. For example, the Boolean satisfaction of the inequality $x[t] \leq 5.5$ does not distinguish between $x = 5.5 + \epsilon$ and $x \gg 5.5$; both are classified as *false*. Similarly, we cannot distinguish between marginal satisfaction by $x = c - \epsilon$ and a more robust satisfaction by $x \ll c$. The broad idea is to associate a real number, often called the *robustness degree*, with a requirement-behavior pair (R, α) that measures the distance between α and the set $[\![R]\!]$ of all behaviors that satisfy R. The robustness degree is positive when α is deeper inside $[\![R]\!]$ and more negative the farther α is from $[\![R]\!]$. Donzé and Maler (2010) present an efficient dense-time algorithm for computing these measures for piecewise-linear signals and for computing the measures' sensitivity to parameter variations.

Along a different axis, *Probabilistic CTL (PCTL)* (Hansson and Jonsson (1994)) and *Continuous Stochastic Logic (CSL)* (Aziz et al. (2000)) add probabilistic operators on top of temporal logics and have been used for analyzing the performance and reliability of systems. For example, the PCTL formula $P_{\geq 0.5}(walkButton \implies (\Diamond_{(0,10)} lightRed))$ means that with a probability of at least 0.5, *lightRed* turns on within 10 time units of *walkButton*.

The time domain for PCTL is $\mathbb{N}$ and that for CSL is $\mathbb{R}_{\geq 0}$. The stochastic analog of the verification problem has to decide, given a PCTL (or CSL) formula f that describes the requirement and a probabilistic model $\mathcal{A}$ (such as a Markov chain or a Markov decision process), whether the model $\mathcal{A}$ satisfies the property f. There are two main approaches to solving the problem: numerical and statistical. In the numerical approach, symbolic and numerical methods are used to model-check the formal model $\mathcal{A}$ for correctness with respect to f. The alternative statistical approach is based on Monte Carlo simulation of the model and performance of sequential hypothesis testing on the generated samples (Younes and Simmons (2002); Sen et al. (2005a)). Each of those approaches has been implemented in several tools (Hermanns et al. (2000); Hinton et al. (2006); Sen et al. (2005b)).

6.6.4 Runtime verification and monitoring

Verification algorithms may also be used for monitoring cyberphysical systems Bartocci and Falcone (2018); Sistla et al. (2011); Duggirala et al. (2014). The problem setup here is for the monitoring algorithm to execute concurrently with the actual implementation of the cyberphysical system: it takes as input estimates of the current state of the system and has to detect or raise an alarm sometime before the requirement is violated. This online version of the verification problem is referred to as *runtime verification* or *monitoring*.

For this use case, the verification algorithm runs periodically, say every δ seconds, on the system being analyzed. At time t, the algorithm takes as input the system model $\mathcal{A}$ and the current state estimate $\Theta_t \subseteq val(V)$, and decides whether or not the requirements *will be* violated by any of the executions starting from Θ_t in the future, up to some lookahead time-bound of T seconds. For the analysis to be actionable, δ must be much smaller than

T. For an automotive application, the lookahead time-bound T may be 6 seconds, and the period of computation δ may be 10 to 100 milliseconds. For nondeterministic models with large errors in state estimation (Θ_t), the verification algorithms may have to check large sets of executions in a short amount of time. A survey of requirements-based monitoring techniques is presented by Bartocci et al. (2018).

6.7 Problems

Exercise 6.11. Consider the SLPlatoon automaton of Section 5.9 modeling a platoon of 3 vehicles on the highway. Define appropriate atomic propositions using the variables of SLPlatoon and express the following requirements in CTL.

(a) "All vehicles always maintain a separation of at least d_{min} and at most d_{max}."

(b) "If Car1 brakes then the other cars also eventually brake."

(c) "If any car goes from cruising to braking, then passes through reacting mode in between."

Exercise 6.12. Five philosophers are sitting around a table, taking turns thinking and eating. Let us assume that the system has the following atomic propositions: e_i, philosopher i is currently eating; f_i, philosopher i just finished eating. Express the following properties of the system in CTL using only the EU, EX, and EF operators. Start by using any operator to write the property, and then convert.

(a) "Philosophers 1 and 4 never eat at the same time."

(b) "If philosopher 4 has finished eating, she cannot eat again until philosopher 3 has begun eating.

(c) "Philosopher 2 will be the first to eat."