15
Formal Methods

The safety and explainability of robotic systems has become increasingly important as applications for robotic systems transition to more unstructured and interactive environments. While one component to developing safe robots is to design robust and high-performing autonomy algorithms, another critical component is the analysis of the system's design. System analysis could come in several forms, including unit, component, or system-level testing, but one challenging aspect of testing is the determination of appropriate success criteria. It is also highly desirable to develop systems that are *provably correct* or *correct-by-construction* with respect to the stated success criteria.

This chapter introduces a set of rigorous mathematical tools and concepts for specifying desired behavior (i.e. requirements or specifications), proving that the system achieves the desired behavior, and synthesizing robot systems to be correct-by-construction. These mathematical tools are known as *formal methods*[1].

Formal Methods

Formal methods provide a mathematical framework for reasoning about a system's specifications as well as analyzing whether the system's behavior guarantees their satisfaction. Approaches for synthesizing provably correct behavior can also be built on top of these tools and are commonly incorporated within the umbrella of formal methods. Historically these techniques have been developed within the computer science community, and have been used to study problems related to logic, automatically proving properties of algorithms, checking the correctness of properties of circuits, and more. However in this chapter formal methods will be explored within the context of autonomous systems.

Definition 15.0.1 (Formal Methods). *Formal methods are mathematically based techniques for the specification, development, and verification of software and hardware systems.*

It is important to note that formal methods are not just particular solutions or algorithms but rather are a class of tools and formalisms. Accordingly, this chapter will not focus on a particular algorithm (solution) for applying formal

[1] E. M. Clarke et al. *Model Checking*. 2nd ed. MIT Press, 2018

methods to problems in robotics, but will rather introduce several broadly used tools and techniques. Specifically, Section 15.1 will introduce *linear temporal logic*, which is a specialized language for writing specifications. Then the concept of the robotic system as a reactive model that can be *verified* to satisfy the stated specifications will be introduced in Section 15.2. Finally, the ability to *synthesize* robot systems to provably be able to satisfy a specification will be explored in Section 15.3.

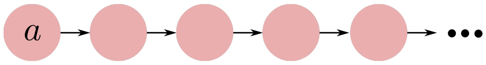
15.1 Linear Temporal Logic

Linear temporal logic (LTL) is a mathematical *language* for formally expressing specifications or requirements on the system's behavior[2]. LTL is useful in robotics applications because it extends propositional logic[3] to handle *temporal* components (assuming discrete time steps), which are common in sequential decision making problems and other robotics tasks. For example propositional logic can be used to write a specification that "proposition a and proposition b must both be true", while LTL extends the possible specifications to include temporal constraints such as "proposition a must be true *until* proposition b is true".

The language of LTL can be expressed in terms of several atomic operators, the first few of which are inherited from propositional logic:

1. *true* or *false* (Boolean values)[4] represent Boolean constants.

2. $a, b, \dots$ (propositional symbols) denote single variables that can either be true or false at the current time step.



3. $\neg\psi$ (negation operator[5]) denotes the negation of ψ.

4. $\psi_1 \wedge \psi_2$ (conjunction "and" operator) which can be read as "ψ_1 and ψ_2".

5. $\psi_1 \vee \psi_2$ (disjunction "or" operator) which can be read as "ψ_1 or ψ_2". This operator can be expressed in terms of "and" and "not" as $\psi_1 \vee \psi_2 = \neg(\neg\psi_1 \wedge \neg\psi_2)$.

6. $\psi_1 \rightarrow \psi_2$ (implication operator) denotes that ψ_1 implies ψ_2. This operator can be expressed in terms of "not' and "or" as $\psi_1 \rightarrow \psi_2 = \neg\psi_1 \vee \psi_2$.

Additional operators that are fundamental to LTL provide the capability for expressing temporal constraints:

7. $X\psi$ ("next" operator) denotes that ψ happens next (at the next time step).

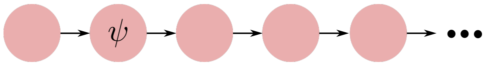
8. $\psi_1 U \psi_2$ ("until" operator) denotes that ψ_1 should happen until ψ_2 happens.

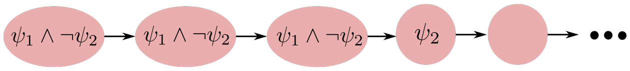
[2] Similar to how ordinary differential equations provide a mathematical "language" that is useful for modeling the kinematics and dynamics of physical systems.

[3] A formalism for expressing logical operations including conjunction (and), disjunction (or), and negation (not).

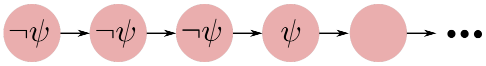
[4] Technically false can also be written as ¬true, where ¬ is the "not" operator.

[5] Technically *false* can be written as ¬*true*.




$\psi_1 \wedge \neg\psi_2$
$\psi_1 \wedge \neg\psi_2$
$\psi_1 \wedge \neg\psi_2$
ψ_2
•••

9. $F\psi$ ("eventually" operator) denotes that ψ happens at some point in the *future*. This operator can be expressed in terms of the eventually operation as $F\psi = \text{true}\, U\, \psi$.



10. $G\psi$ ("always" operator) denotes that ψ happens globally (at all times). This operator can be expressed in terms of the negation and future operations as $G\psi = \neg F \neg \psi$.

From these atomic operators it is possible to define many new specifications through *composition*, and they can become arbitrarily complex as needed. A couple of common and useful compositions include:

11. $GF\psi$ ("infinitely often" composition) denotes that ψ will eventually happen an infinite number of times (i.e. globally, ψ will happen eventually). In other words there is always a ψ in the future.

12. $FG\psi$ ("stability"[6] composition) denotes that at some point in time ψ will be true for all time thereafter (i.e. eventually ψ will happen globally).

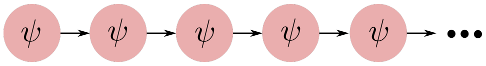
13. $G(\psi_1 \to F\psi_2)$ ("response" composition) denotes that for all time, whenever ψ_1 occurs then ψ_2 will occur sometime in the future. There are other useful variations on this composition, such as by replacing the F operator with X operator.

Linear temporal logic provides a very powerful tool[7] for abstractly talking about time, and in general the specifications written using LTL can in a way be more "vague". For example the eventually operator F does not explicitly state *when* something must occur, just that at *some point* it will. It is also important to keep in mind that LTL is not an algorithm or technique for solving problems, but rather a language for *formulating* problems (i.e. for expressing properties of interest such as system specifications).

Example 15.1.1 (Coffee Machine Specification)**.** Consider a simple robot that makes coffee. This robot has a button that a user can press, and has two functions: grinding coffee beans and brewing coffee. The desired behavior of this robot could be expressed by the designer as: *if the start button is pressed, the robot will immediately start grinding beans for the next two cycles, and then brew the coffee for the next two cycles after that*. This specification, denoted as ϕ, could be

[6] This notion of stability is similar but not directly the same as the notions of stability from control theory.

[7] There are also alternatives to LTL that provide even more powerful features, for example by not requiring a "linear" temporal structure but rather allowing for temporal "branching".



expressed via linear temporal logic as:

$$\phi : G\big(\text{button} \rightarrow \text{grind} \wedge X\text{grind} \wedge XX(\neg\text{grind} \wedge \text{brew}) \wedge XXX(\neg\text{grind} \wedge \text{brew})\big).$$

Note that the entire statement needs to be wrapped in the "always" operator to ensure this behavior can occur at any arbitrary time that a user presses the button.

15.2 *Verification*

System verification is the process of *proving* that the system's behavior will satisfy the stated requirements and specifications (often expressed using linear temporal logic). In the terminology of formal methods, this system is typically referred to as the *model*[8] and the output of the verification[9] procedure is a simple yes/no stating whether the model satisfies the specification. In this chapter the model will be denoted by P and the specification by ϕ, and the notation for stating that model P satisfies specification ϕ is $P \models \phi$ (which can be read as "P models ϕ").

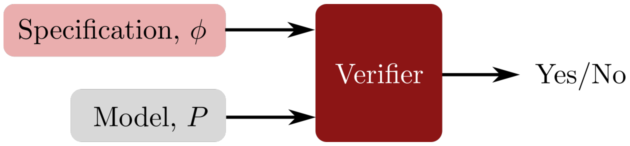


Figure 15.1: Given a system (model) and a specification, the process of verification proves whether the system satisfies the specification.

In this chapter it is assumed that the model P is a *reactive system*, meaning that its behavior is defined based on inputs i which effect the system's outputs o[10]. In contrast to robotic control problems where the "inputs" are generally the control inputs determined by the control algorithm, the inputs i within this context refer to signals coming from the environment. The outputs o of the model can then be thought of as the result of the system's decision making or underlying algorithm/process. As was mentioned previously, the model P can take on many forms depending on whether the system is a hardware component, software component, algorithm, finite state machine, or even simply a mathematical function such as a machine learning model or control law.

For a given model P the specification ϕ is assumed to be written in terms of the input and output sequences (i.e. the behavior is defined by the inputs and outputs of the system). Specifically, these sequences will be denoted as $\hat{i} = (i_0, i_1, \dots)$ and $\hat{o} = (o_0, o_1, \dots)$. With these definitions, the expression that P satisfies ϕ can equivalently be written as $P \models \phi$ or $\hat{i} \cup \hat{o} \models \phi$.

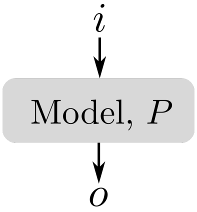


Figure 15.2: The model P is a system with inputs i and outputs o. These inputs and outputs are used to express the model's specification ϕ.

To summarize, the problem of model verification is to simply determine whether the input-output behavior of the model P guarantees that ϕ is satisfied

[8] The model could refer to a piece of software, a hardware component, an individual algorithm, or even an entire robot.

[9] Also commonly referred to as *model checking*.

[10] The inputs and outputs occur at each time step, and the behavior is assumed to be non-terminating.

for all possible input sequences. There are several existing techniques that can perform system verification, and they may be tailored to the specific model form[11,12].

15.3 Reactive Synthesis

Given a reactive model P and a LTL specification ϕ, the problem of verification is to determine whether the behavior of P satisfies ϕ for all possible input sequences $\hat{i}$. But several important questions remain: how should the system be designed, and what should be changed if the verification step shows that $P \not\models \phi$? *Reactive synthesis* addresses these problems by *synthesizing* the system model P to be correct-by-construction. In other words, in reactive synthesis the specification ϕ is first defined and then a model is constructed from scratch to satisfy the specification.

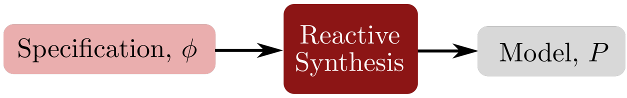


Figure 15.3: Given a specification ϕ, the process of reactive synthesis generates a model P that realizes the specification under all possible environmental inputs.

15.3.1 Specification Satisfiability and Realizability

The first step in reactive synthesis is to determine whether a model[13] even exists which can satisfy the LTL specification ϕ for all possible input sequences. If no such model exists, then the system designer should reevaluate the specification itself.

In the nomenclature of formal methods, the specification ϕ is said to be *realizable* if it can be satisfied for all possible input sequences, and it is *satisfiable* if there exists at least one input sequence leading to satisfaction. These properties can be more rigorously defined in terms of the input and output sequences that describe the system's behavior:

Definition 15.3.1 (Satisfiability)**.** *A specification ϕ is satisfiable if for some input sequence there exists an output sequence that satisfies the specification. Mathematically:*

$$\exists\, \hat{i} = (i_0, i_1, \dots), \quad \exists\, \hat{o} = (o_0, o_1, \dots), \quad s.t.\ \hat{i} \cup \hat{o} \models \phi.$$

Definition 15.3.2 (Realizability)**.** *A specification ϕ is realizable if for all possible input sequences there exists an output sequence that satisfies the specification. Mathematically:*

$$\forall\, \hat{i} = (i_0, i_1, \dots), \quad \exists\, \hat{o} = (o_0, o_1, \dots), \quad s.t.\ \hat{i} \cup \hat{o} \models \phi.$$

Obviously the property of satisfiability is weaker than realizability, and realizability is much more important in practice. For example in order to guarantee safety in a rigorous way it is not sufficient to show that the system will be safe

[11] M. Kwiatkowska, G. Normal, and D. Parker. "PRISM 4.0: Verification of Probabilistic Real-time Systems". In: *Proceedings of the 23rd International Conference on Computer Aided Verification*. 2011, pp. 585–591

[12] G. Katz et al. "The Marabou Framework for Verification and Analysis of Deep Neural Networks". In: *Computer Aided Verification*. 2019, pp. 443–452

[13] Technically speaking, a *finite state model*.

under a single scenario, but rather it should be shown for all scenarios. However, designing specifications that are realizable can be quite challenging, even in seemingly simple problems. As an example consider again the coffee machine robot from Example 15.1.1.

Example 15.3.1 (Coffee Machine Realizability)**.** The coffee machine robot from Example 15.1.1 had inputs $I = \{\text{button}\}$ and outputs $O = \{\text{grind}, \text{brew}\}$. For simplicity let $i_{\text{button}} = 1$ and $i_{\text{button}} = 0$ denote the button is pressed and not pressed, respectively. Additionally let $o_{\text{grind}} = 1$ and $o_{\text{brew}} = 1$ denote that the actions are occurring and let them be zero otherwise.

Recall that the linear temporal logic specification for the robot's behavior was defined as:

$$\phi : G\big(\text{button} \rightarrow \text{grind} \wedge X\text{grind} \wedge XX(\neg\text{grind} \wedge \text{brew}) \wedge XXX(\neg\text{grind} \wedge \text{brew})\big).$$

This specification can now be analyzed to determine whether it is realizable. First, notice that one possible sequence of inputs and outputs that satisfies this specification is:

$$\begin{aligned}(i_{\text{button}}, o_{\text{grind}}, o_{\text{brew}})_k = &(0, 0, 0)_0,\\ &(1, 1, 0)_1,\\ &(0, 1, 0)_2,\\ &(0, 0, 1)_3,\\ &(0, 0, 1)_4,\\ &(0, 0, 0)_5,\\ &\quad\vdots\end{aligned}$$

Because there exists a sequence that satisfies the specification ϕ it is by definition *satisfiable*. However, consider a second input sequence $\hat{i} = (0, 1, 1, 0, 0, \dots)$ where the coffee machine's button is pressed twice in a row:

$$\begin{aligned}(i_{\text{button}}, o_{\text{grind}}, o_{\text{brew}})_k = &(0, 0, 0)_0,\\ &(1, 1, 0)_1,\\ &(1, 1, 0)_2,\\ &(0, ?, 1)_3,\end{aligned}$$

At time step $k = 3$ there is no combination of outputs that will satisfy the specification, since the first button press requires that $o_{\text{grind},3} = 0$ but the second button press requires that $o_{\text{grind},3} = 1$! Therefore by definition this specification is not *realizable*[14].

15.3.2 Synthesis for Realizable LTL Specifications

If a LTL specification ϕ is realizable, then the synthesis problem seeks to find a finite state system that satisfies ϕ under all possible inputs. This can be accomplished by formulating the problem as a two-player game where the objective

[14] Does this mean it is impossible to automate a coffee maker? No! It just demonstrates that writing *specifications* can be challenging.

is for the system to generate "winning" outputs while the environment generates adversarial inputs. The two-player game formulation can be expressed mathematically by defining the following components:

1. With the inputs I and outputs O, at each time step the environment gets to choose from a set of $2^{|I|}$ actions and the system gets to choose from a set of $2^{|O|}$ actions.

2. The strategy of the system is expressed as a function $f : (2^{|I|})^* \to 2^{|O|}$, where f is a function from a finite sequence of environmental inputs to a specific output.

3. The linear temporal logic specification ϕ is defined by the input and output sequences.

4. The game is played for an infinitely long horizon, generating sequences $\hat{i} = (i_0, i_1, \dots)$ and $\hat{o} = (o_0, o_1, \dots)$.

5. The game is won if $\hat{i} \cup \hat{o} \models \phi$.

The process of converting a problem specification into this two-player game follows two main steps. First, the specification is converted into a *non-deterministic Büchi automaton* and then the automaton is determinized to yield the game. Once the game is appropriately formulated, it can be solved using existing algorithms to generate the policy f that defines the system's behavior. Unfortunately, converting the specification into the automaton is computationally very challenging! In fact the computational complexity is *doubly-exponential* in the size of the specification[15], which significantly limits the complexity of the problems that can be considered[16].

While the precise details for converting a specification into a two-player game and solving the game are beyond the scope of this chapter, the process can be explored visually through the following example.

Example 15.3.2 (Simple Reactive Synthesis Problem)**.** Consider the LTL specification $\phi : G(r \to Xg)$ which states that whenever a request r is received the system should provide a grant g in the next time step. In this problem $I = \{r\}$ where r denotes a request or no request and $O = \{g\}$ where g specifies if a grant was made or not. The first step in transforming this specification into the two-player synthesis game is to generate the following Büchi automaton representation, as shown in Figure 15.4. In Figure 15.4 the variables q_0 and q_1 represent states of the automaton, and the transitions between the states are dependent on the environmental inputs and the system's behavior.

The two-player game is generated from this Büchi automaton[17] by introducing intermediate states as well as the unsafe "contradiction" state (denoted in Figure 15.5 as $\perp$). This game is represented graphically in Figure 15.5 where the $*$ denotes that any action could be taken by the model and the small grey circles represent the intermediate states. The system can "win" this two-player game

[15] A doubly exponential function has the form $f(x) = a^{b^x}$.

[16] This is one of the most significant limitations of formal methods in practical robotic settings, and approaches to overcome this complexity are still a topic of research.

[17] This automaton is already deterministic, so no determinizing step is needed.

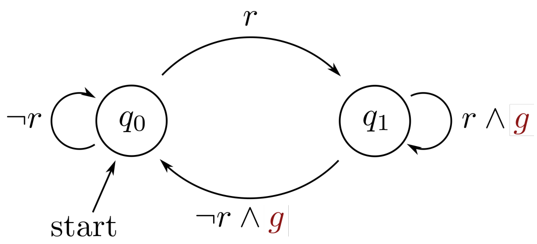


Figure 15.4: The Büchi automaton representation of the LTL specification $\phi : G(r \rightarrow Xg)$.

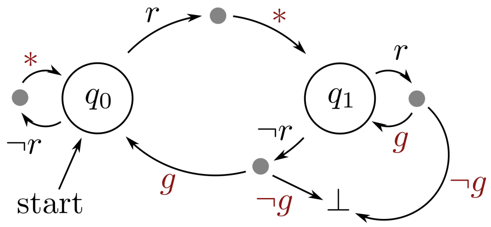


Figure 15.5: The two-player game representation derived from the Büchi automaton in Figure 15.4.

by ensuring that the contradiction state is never reached, which then defines the system's behavior! By analyzing Figure 15.5, it turns out that one "winning" strategy strategy (behavior) is for the system to *always* provide a grant! This strategy is shown graphically in Figure 15.6.

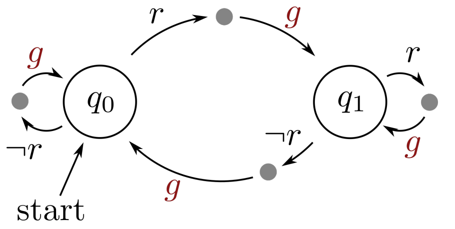


Figure 15.6: A strategy for the system in Example 15.3.2 to ensure the specification is met is to *always* provide grants, which is guaranteed to avoid the contradiction state in the two-player game shown in Figure 15.5.