## Part III

# **Analysis and Verification**

This part of this text studies analysis of embedded systems, with emphasis on methods for specifying desired and undesired behaviors and verifying that an implementation conforms to its specification. Chapter 13 covers temporal logic, a formal notation that can express families of input/output behaviors and the evolution of the state of a system over time. This notation can be used to specify unambiguously desired and undesired behaviors. Chapter 14 explains what it means for one specification. Chapter 15 shows how to check algorithmically whether a design correctly implements a specification. Chapter 16 illustrates how to analyze designs for quantitative properties, with emphasis on execution time analysis for software. Such analysis is essential to achieving real-time behavior in software. Chapter 17 introduces the basics of security and privacy with a focus on concepts relevant to embedded, cyber-physical systems.

# Invariants and Temporal Logic

13.1	Invariants	9
13.2	Linear Temporal Logic	2
	13.2.1 Propositional Logic Formulas	2
	13.2.2 LTL Formulas	4
	Sidebar: Probing Further: Alternative Temporal Logics	7
	13.2.3 Using LTL Formulas	9
13.3	Summary	0
	Sidebar: Safety and Liveness Properties	1
Exercises		

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

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

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

- 1. "The *ErrorReset* state shall be entered after a system reset, after link operation has been terminated for any reason or if there is an error during link initialization."
- 2. "Whenever the reset signal is asserted the state machine shall move immediately to the *ErrorReset* state and remain there until the reset signal is de-asserted."

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

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

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

## 13.1 Invariants

An **invariant** is a property that holds for a system if it remains true at all times during operation of the system. Put another way, an invariant holds for a system if it is true in the

initial state of the system, and it remains true as the system evolves, after every reaction, in every state.

In practice, many properties are invariants. Both properties of the SpaceWire protocol stated above are invariants, although this might not be immediately obvious. Both SpaceWire properties specify conditions that must remain true always. Below is an example of an invariant property of a model that we have encountered in Chapter 3.

**Example 13.1:** Consider the model of a traffic light controller given in Figure 3.10 and its environment as modeled in Figure 3.11. Consider the system formed by the asynchronous composition of these two state machines. An obvious property that the composed system must satisfy is that *there is no pedestrian crossing when the traffic light is green* (when cars are allowed to move). This property must always remain true of this system, and hence is a system invariant.

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

"The program never dereferences a null pointer"

is an invariant specifying good programming practice. Typically dereferencing a null pointer in a C program results in a segmentation fault, possibly leading to a system crash. Similarly, several desirable properties of concurrent programs are invariants, as illustrated in the following example.

**Example 13.2:** Consider the following property regarding an absence of dead-lock:

If a thread A blocks while trying to acquire a mutex lock, then the thread B that holds that lock must not be blocked attempting to acquire a lock held by A.

This property is required to be an invariant on any multithreaded program constructed from threads A and B. The property may or may not hold for a particular program. If it does not hold, there is risk of deadlock.

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

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

```
void altitude_control_task(void) {
1
    if (pprz mode == PPRZ MODE AUTO2
2
        || pprz_mode == PPRZ_MODE_HOME) {
3
     if (vertical_mode == VERTICAL_MODE_AUTO_ALT) {
4
      float err = estimator_z - desired_altitude;
5
      desired_climb
6
            = pre_climb + altitude_pgain * err;
7
      if (desired_climb < -CLIMB_MAX) {</pre>
8
       desired_climb = -CLIMB_MAX;
9
10
      if (desired_climb > CLIMB_MAX) {
11
       desired climb = CLIMB MAX;
12
13
      }
14
     }
15
    }
  }
16
```

For this example, it is required that the value of the desired\_climb variable at the end of altitude\_control\_task remains within the range [-CLIMB\_MAX, CLIMB\_MAX]. This is an example of a special kind of invariant, a **postcondition**, that must be maintained every time altitude\_control\_task returns. Determining whether this is the case requires analyzing the control flow of the program.

## 13.2 Linear Temporal Logic

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

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

- Occurrence of an event and its properties. For example, one can express the property that an event A must occur at least once in every trace of the system, or that it must occur infinitely many times.
- *Causal dependency between events*. An example of this is the property that if an event *A* occurs in a trace, then event *B* must also occur.
- Ordering of events. An example of this kind of property is one specifying that every occurrence of event A is preceded by a matching occurrence of B.

We now formalize the above intuition about the kinds of properties expressible in linear temporal logic. In order to perform this formalization, it is helpful to fix a particular formal model of computation. We will use the theory of finite-state machines, introduced in Chapter 3.

Recall from Section 3.6 that an execution trace of a finite-state machine is a sequence of the form

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

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

## 13.2.1 Propositional Logic Formulas

First, we need to be able to talk about conditions at each reaction, such as whether an input or output is present, what the value of an input or output is, or what the state is. Let an **atomic proposition** be such a statement about the inputs, outputs, or states. It is a predicate (an expression that evaluates to true or false). Examples of atomic propositions that are relevant for the state machines in Figure 13.1 are:

true	Always true.
false	Always false.
x	True if input x is present.
x = present	True if input x is present.
y = absent	True if y is absent.
b	True if the FSM is in state b

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

A **propositional logic formula** or (more simply) **proposition** is a predicate that combines atomic propositions using **logical connectives**: conjunction (logical AND, denoted  $\land$ ), disjunction (logical OR, denoted  $\lor$ ), negation (logical NOT, denoted  $\neg$ ), and **implies** (logical implication, denoted  $\Longrightarrow$ ). Propositions for the state machines in Figure 13.1 include any of the above atomic proposition and expressions using the logical connectives together with atomic propositions. Here are some examples:

$x \wedge y$	True if $x$ and $y$ are both <i>present</i> .
$x \lor y$	True if either x or y is present.
$x = present \land y = absent$	True if x is present and y is absent.
eg y	True if y is absent.
$a \implies y$	True if whenever the FSM is in state a, the
	output $y$ will be made present by the reaction





Figure 13.1: Two finite-state machines used to illustrate LTL formulas.

Note that if  $p_1$  and  $p_2$  are propositions, the proposition  $p_1 \implies p_2$  is true if and only if  $\neg p_2 \implies \neg p_1$ . In other words, if we wish to establish that  $p_1 \implies p_2$  is true, it is equally valid to establish that  $\neg p_2 \implies \neg p_1$  is true. In logic, the latter expression is called the **contrapositive** of the former.

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

 $\neg p_1 \lor p_2$ .

## 13.2.2 LTL Formulas

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

 $q_0, q_1, q_2, \ldots,$ 

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

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

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

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

The LTL formula  $x \implies y$  holds for both machines. In both cases, in the first reaction, if x is *present*, then y will be *present*.

To demonstrate that an LTL formula is false for an FSM, it is sufficient to give one trace for which it is false. Such a trace is called a **counterexample**. To show that an LTL formula is true for an FSM, you must demonstrate that it is true for all traces, which is often much harder (although not so much harder when the LTL formula is a simple propositional logic formula, because in that case we only have to consider the first element of the trace).

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

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

## **G** Operator

The property  $\mathbf{G}\phi$  (which is read as "globally  $\phi$ ") holds for a trace if  $\phi$  holds for *every* suffix of that trace. (A suffix is a tail of a trace beginning with some reaction and including all subsequent reactions.)

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

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

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

## **F** Operator

The property  $\mathbf{F}\phi$  (which is read as "eventually  $\phi$ " or "finally  $\phi$ ") holds for a trace if  $\phi$  holds for *some* suffix of the trace.

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

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

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

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

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

## ${\bf X}$ Operator

The property  $\mathbf{X}\phi$  (which is read as "**next state**  $\phi$ ") holds for a trace  $q_0, q_1, q_2, \ldots$  if and only if  $\phi$  holds for the trace  $q_1, q_2, q_3, \ldots$ 

**Example 13.8:** In Figure 13.1(a),  $x \implies Xa$  holds for the state machine, because if x is *present* in the first reaction, then the next state will be a.  $G(x \implies x)$ 

## **Probing Further: Alternative Temporal Logics**

Amir Pnueli (1977) was the first to formalize temporal logic as a way of specifying program properties. For this he won the 1996 ACM Turing Award, the highest honor in Computer Science. Since his seminal paper, temporal logic has become widespread as a way of specifying properties for a range of systems, including hardware, software, and cyber-physical systems.

In this chapter, we have focused on LTL, but there are several alternatives. LTL formulas apply to individual traces of an FSM, and in this chapter, by convention, we assert than an LTL formula holds for an FSM if it holds for all possible traces of the FSM. A more general logic called **computation tree logic** (**CTL**<sup>\*</sup>) explicitly provides quantifiers over possible traces of an FSM (Emerson and Clarke (1980); Ben-Ari et al. (1981)). For example, we can write a CTL<sup>\*</sup> expression that holds for an FSM if there exists *any* trace that satisfies some property, rather than insisting that the property must hold *for all* traces. CTL<sup>\*</sup> is called a **branching-time logic** because whenever a reaction of the FSM has a nondeterministic choice, it will simultaneously consider all options. LTL, by contrast, considers only one trace at a time, and hence it is called a **linear-time logic**. Our convention of asserting that an LTL formula holds for an FSM if it holds for all traces." We have to step outside the logic to apply this convention. With CTL<sup>\*</sup>, this convention is expressible directly in the logic.

Several other temporal logic variants have found practical use. For instance, **real-time temporal logics** (e.g., **timed computation tree logic** or **TCTL**), is used for reasoning about real-time systems (Alur et al., 1991; Alur and Henzinger, 1993) where the passage of time is not in discrete steps, but is continuous. Similarly, **probabilistic temporal logics** are useful for reasoning about probabilistic models such as Markov chains or Markov decision processes (see, for example, Hansson and Jonsson (1994)), and **signal temporal logic** has proved effective for reasoning about real-time behavior of hybrid systems (Maler and Nickovic, 2004).

Techniques for inferring temporal logic properties from traces, also known as **spec-ification mining**, have also proved useful in industrial practice (see Jin et al. (2015)).

**X**a) does not hold for the state machine because it does not hold for any suffix that begins in state a. In Figure 13.1(b),  $G(b \implies Xa)$  holds for the state machine.

#### **U** Operator

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

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

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

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

$$(\mathbf{G}\phi_1) \lor (\phi_1 \mathbf{U}\phi_2)$$
.

This holds if either  $\phi_1$  always holds (for any suffix) or, if  $\phi_2$  holds for some suffix, then  $\phi_1$  holds for all previous suffixes. This can equivalently be written

$$(\mathbf{F} \neg \phi_1) \implies (\phi_1 \mathbf{U} \phi_2).$$

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

## 13.2.3 Using LTL Formulas

Consider the following English descriptions of properties and their corresponding LTL formalizations:

**Example 13.11:** *"Whenever the robot is facing an obstacle, eventually it moves at least 5 cm away from the obstacle."* 

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

$$\mathbf{G}\left(p\implies\mathbf{F}q
ight)$$
 .

**Example 13.12:** Consider the SpaceWire property:

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

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

 $\mathbf{G}\left(p \implies \mathbf{X}(q \, \mathbf{U} \, \neg p)\right).$ 

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

**Example 13.13:** Consider the traffic light controller in Figure 3.10. A property of this controller is that the outputs always cycle through sigG, sigY and sigR. We

can express this in LTL as follows:

The following LTL formulas express commonly useful properties.

- (a) *Infinitely many occurrences:* This property is of the form  $\mathbf{G} \mathbf{F} p$ , meaning that it is always the case that p is *true* eventually. Put another way, this means that p is true **infinitely often**.
- (b) Steady-state property: This property is of the form F Gp, read as "from some point in the future, p holds at all times." This represents a steady-state property, indicating that after some point in time, the system reaches a steady state in which p is always true.
- (c) Request-response property: The formula  $\mathbf{G}(p \implies \mathbf{F}q)$  can be interpreted to mean that a request p will eventually produce a response q.

## 13.3 Summary

Dependability and correctness are central concerns in embedded systems design. Formal specifications, in turn, are central to achieving these goals. In this chapter, we have studied temporal logic, one of the main approaches for writing formal specifications. This chapter has provided techniques for precisely stating properties that must hold over time for a system. It has specifically focused on linear temporal logic, which is able to express many safety and liveness properties of systems.

## Safety and Liveness Properties

System properties may be **safety** or **liveness** properties. Informally, a safety property is one specifying that "nothing bad happens" during execution. Similarly, a liveness property specifies that "something good will happen" during execution.

More formally, a property p is a **safety property** if a system execution does not satisfy p if and only if there exists a finite-length prefix of the execution that cannot be extended to an infinite execution satisfying p. We say p is a **liveness property** if every finite-length execution trace can be extended to an infinite execution that satisfies p. See Lamport (1977) and Alpern and Schneider (1987) for a theoretical treatment of safety and liveness.

The properties we have seen in Section 13.1 are all examples of safety properties. Liveness properties, on the other hand, specify performance or progress requirements on a system. For a state machine, a property of the form  $\mathbf{F}\phi$  is a liveness property. No finite execution can establish that this property is not satisfied.

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

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

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

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

Note that both safety and liveness properties can constitute system invariants. For example, the above liveness property on interrupts is also an invariant;  $p_1 \implies \mathbf{F}p_2$  must hold in *every state*.

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

## **Exercises**

- 1. For each of the following questions, give a short answer and justification.
  - (a) TRUE or FALSE: If  $\mathbf{GF}p$  holds for a state machine A, then so does  $\mathbf{FG}p$ .
  - (b) TRUE or FALSE: G(Gp) holds for a trace if and only if Gp holds.
- 2. Consider the following state machine:



(Recall that the dashed line represents a default transition.) For each of the following LTL formulas, determine whether it is true or false, and if it is false, give a counterexample:

- (a)  $x \implies \mathbf{Fb}$
- (b)  $\mathbf{G}(x \implies \mathbf{F}(y=1))$
- (c)  $(\mathbf{G}x) \implies \mathbf{F}(y=1)$
- (d)  $(\mathbf{G}x) \implies \mathbf{GF}(y=1)$
- (e)  $\mathbf{G}((\mathbf{b} \wedge \neg x) \implies \mathbf{FGc})$
- (f)  $\mathbf{G}((\mathbf{b} \land \neg x) \implies \mathbf{Gc})$

(g) 
$$(\mathbf{GF}\neg x) \implies \mathbf{FGc}$$

Consider the synchronous feedback composition studied in Exercise 6 of Chapter
 Determine whether the following statement is true or false:

The following temporal logic formula is satisfied by the sequence w for every possible behavior of the composition and is not satisfied by any sequence that is not a behavior of the composition:

$$(\mathbf{G}w) \lor (w\mathbf{U}(\mathbf{G}\neg w))$$

Justify your answer. If you decide it is false, then provide a temporal logic formula for which the assertion is true.

4. This problem is concerned with specifying in linear temporal logic tasks to be performed by a robot. Suppose the robot must visit a set of n locations l<sub>1</sub>, l<sub>2</sub>,..., l<sub>n</sub>. Let p<sub>i</sub> be an atomic formula that is *true* if and only if the robot visits location l<sub>i</sub>.

Give LTL formulas specifying the following tasks:

- (a) The robot must eventually visit at least one of the n locations.
- (b) The robot must eventually visit all n locations, but in any order.
- (c) The robot must eventually visit all n locations, in the order  $l_1, l_2, \ldots, l_n$ .
- 5. Consider a system *M* modeled by the hierarchical state machine of Figure 13.2, which models an interrupt-driven program. *M* has two modes: Inactive, in which the main program executes, and Active, in which the interrupt service routine (ISR) executes. The main program and ISR read and update a common variable *timer-Count*.

Answer the following questions:

- (a) Specify the following property  $\phi$  in linear temporal logic, choosing suitable atomic propositions:
  - $\phi$ : The main program eventually reaches program location C.
- (b) Does M satisfy the above LTL property? Justify your answer by constructing the product FSM. If M does not satisfy the property, under what conditions would it do so? Assume that the environment of M can assert the interrupt at any time.
- 6. Express the postcondition of Example 13.3 as an LTL formula. State your assumptions clearly.
- 7. Consider the program fragment shown in Figure 11.6, which provides procedures for threads to communicate asynchronously by sending messages to one another. Please answer the following questions about this code. Assume the code is running on a single processor (not a multicore machine). You may also assume that only the code shown accesses the static variables that are shown.



Figure 13.2: Hierarchical state machine modeling a program and its interrupt service routine.

- (a) Let *s* be an atomic proposition asserting that send releases the mutex (i.e. executes line 24). Let *g* be an atomic proposition asserting that get releases the mutex (i.e. executes line 38). Write an LTL formula asserting that *g* cannot occur before *s* in an execution of the program. Does this formula hold for the first execution of any program that uses these procedures?
- (b) Suppose that a program that uses the send and get procedures in Figure 11.6 is aborted at an arbitrary point in its execution and then restarted at the beginning. In the new execution, it is possible for a call to get to return before any call to send has been made. Describe how this could come about. What value will get return?
- (c) Suppose again that a program that uses the send and get procedures above is aborted at an arbitrary point in its execution and then restarted at the beginning. In the new execution, is it possible for deadlock to occur, where neither

a call to get nor a call to send can return? If so, describe how this could come about and suggest a fix. If not, give an argument.

14

# **Equivalence and Refinement**

14.1	Models as Specifications	377
14.2	Type Equivalence and Refinement	378
	Sidebar: Abstraction and Refinement	378
14.3	Language Equivalence and Containment	381
	Sidebar: Finite Sequences and Accepting States	384
	Sidebar: Regular Languages and Regular Expressions	385
	Sidebar: Probing Further: Omega Regular Languages	386
14.4	Simulation	387
	14.4.1 Simulation Relations	389
	14.4.2 Formal Model	391
	14.4.3 Transitivity	392
	14.4.4 Non-Uniqueness of Simulation Relations	393
	14.4.5 Simulation vs. Language Containment	393
14.5	Bisimulation	395
14.6	Summary	398
Exer	cises	399

This chapter discusses some fundamental ways to compare state machines and other modal models, such as trace equivalence, trace containment, simulation, and bisimulation. These mechanisms can be used to check conformance of a state machine against a specification.

## 14.1 Models as Specifications

The previous chapter provided techniques for unambiguously stating properties that a system must have to be functioning properly and safely. These properties were expressed using linear temporal logic, which can concisely describe requirements that the trace of a finite-state machine must satisfy. An alternative way to give requirements is to provide a model, a specification, that exhibits expected behavior of the system. Typically, the specification is quite abstract, and it may exhibit more behaviors than a useful implementation of the system would. But the key to being a useful specification is that it explicitly excludes undesired or dangerous behaviors.

**Example 14.1:** A simple specification for a traffic light might state: "The lights should always be lighted in the order green, yellow, red. It should never, for example, go directly from green to red, or from yellow to green." This requirement can be given as a temporal logic formula (as is done in Example 13.13) or as an abstract model (as is done in Figure 3.12).

The topic of this chapter is on the use of abstract models as specifications, and on how such models relate to an implementation of a system and to temporal logic formulas.

**Example 14.2:** We will show how to demonstrate that the traffic light model shown in Figure 3.10 is a valid implementation of the specification in Figure 3.12. Moreover, all traces of the model in Figure 3.10 satisfy the temporal logic formula in Example 13.13, but not all traces of the specification in Figure 3.12 do. Hence, these two specifications are not the same.

This chapter is about comparing models, and about being able to say with confidence that one model can be used in place of another. This enables an engineering design process where we start with abstract descriptions of desired and undesired behaviors, and successively refine our models until we have something that is detailed enough to provide a complete implementation. It also tells when it is safe to change an implementation, replacing it with another that might, for example, reduce the implementation cost.

## 14.2 Type Equivalence and Refinement

We begin with a simple relationship between two models that compares only the data types of their communication with their environment. Specifically, the goal is to ensure that a model B can be used in any environment where a model A can be used without causing any conflicts about data types. We will require that B can accept any inputs that

## **Abstraction and Refinement**

This chapter focuses on relationships between models known as **abstraction** and **re-finement**. These terms are symmetric in that the statement "model A is an abstraction of model B" means the same thing as "model B is a refinement of model A." As a general rule, the refinement model B has more detail than the abstraction A, and the abstraction is simpler, smaller, or easier to understand.

An abstraction is **sound** (with respect to some formal system of properties) if properties that are true of the abstraction are also true of the refinement. The formal system of properties could be, for example, a type system, linear temporal logic, or the languages of state machines. If the formal system is LTL, then if every LTL formula that holds for A also holds for B, then A is a sound abstraction of B. This is useful when it is easier to prove that a formula holds for A than to prove that it holds for B, for example because the state space of B may be much larger than the state space of A.

An abstraction is **complete** (with respect to some formal system of properties) if properties that are true of the refinement are also true of the abstraction. For example, if the formal system of properties is LTL, then A is a complete abstraction of B if every LTL formula that holds for B also holds for A. Useful abstractions are usually sound but not complete, because it is hard to make a complete abstraction that is significantly simpler or smaller.

Consider for example a program B in an imperative language such as C that has multiple threads. We might construct an abstraction A that ignores the values of variables and replaces all branches and control structures with nondeterministic choices. The abstraction clearly has less information than the program, but it may be sufficient for proving some properties about the program, for example a mutual exclusion property. A can accept from the environment, and that any environment that can accept any output A can produce can also accept any output that B can produce.

To make the problem concrete, assume an actor model for A and B, as shown in Figure 14.1. In that figure, A has three ports, two of which are input ports represented by the set  $P_A = \{x, w\}$ , and one of which is an output port represented by the set  $Q_A = \{y\}$ . These ports represent communication between A and its environment. The inputs have type  $V_x$  and  $V_w$ , which means that at a reaction of the actor, the values of the inputs will be members of the sets  $V_x$  or  $V_w$ .

If we want to replace A by B in some environment, the ports and their types impose four constraints:

1. The first constraint is that B does not require some input signal that the environment does not provide. If the input ports of B are given by the set  $P_B$ , then this is guaranteed by

$$P_B \subseteq P_A. \tag{14.1}$$



Figure 14.1: Summary of type refinement. If the four constraints on the right are satisfied, then B is a type refinement of A.

The ports of B are a subset of the ports of A. It is harmless for A to have more input ports than B, because if B replaces A in some environment, it can simply ignore any input signals that it does not need.

2. The second constraint is that B produces all the output signals that the environment may require. This is ensured by the constraint

$$Q_A \subseteq Q_B, \tag{14.2}$$

where  $Q_A$  is the set of output ports of A, and  $Q_B$  is the set of output ports of B. It is harmless for B to have additional output ports because an environment capable of working with A does not expect such outputs and hence can ignore them.

The remaining two constraints deal with the types of the ports. Let the type of an input port  $p \in P_A$  be given by  $V_p$ . This means that an acceptable input value v on p satisfies  $v \in V_p$ . Let  $V'_p$  denote the type of an input port  $p \in P_B$ .

3. The third constraint is that if the environment provides a value  $v \in V_p$  on an input port p that is acceptable to A, then if p is also an input port of B, then the value is also acceptable to B; i.e.,  $v \in V'_p$ . This constraint can be written compactly as follows,

$$\forall \ p \in P_B, \quad V_p \subseteq V_p'. \tag{14.3}$$

Let the type of an output port  $q \in Q_A$  be  $V_q$ , and the type of the corresponding output port  $q \in Q_B$  be  $V'_q$ .

4. The fourth constraint is that if B produces a value  $v \in V'_q$  on an output port q, then if q is also an output port of A, then the value must be acceptable to any environment in which A can operate. In other words,

$$\forall q \in Q_A, \quad V_q' \subseteq V_q. \tag{14.4}$$

The four constraints of equations (14.1) through (14.4) are summarized in Figure 14.1. When these four constraints are satisfied, we say that *B* is a **type refinement** of *A*. If *B* is a type refinement of *A*, then replacing *A* by *B* in any environment will not cause type system problems. It could, of course, cause other problems, since the behavior of *B* may not be acceptable to the environment, but that problem will be dealt with in subsequent sections.

If B is a type refinement of A, and A is a type refinement of B, then we say that A and B are **type equivalent**. They have the same input and output ports, and the types of the ports are the same.

**Example 14.3:** Let A represent the nondeterministic traffic light model in Figure 3.12 and B represent the more detailed deterministic model in Figure 3.10. The ports and their types are identical for both machines, so they are type equivalent. Hence, replacing A with B or vice versa in any environment will not cause type system problems.

Notice that since Figure 3.12 ignores the *pedestrian* input, it might seem reasonable to omit that port. Let A' represent a variant of Figure 3.12 without the *pedestrian* input. It is not safe to replace A' with B in all environments, because B requires an input *pedestrian* signal, but A' can be used in an environment that provides no such input.

## 14.3 Language Equivalence and Containment

To replace a machine A with a machine B, looking at the data types of the inputs and outputs alone is usually not enough. If A is a specification and B is an implementation, then normally A imposes more constraints than just data types. If B is an optimization of A (e.g., a lower cost implementation or a refinement that adds functionality or leverages new technology), then B normally needs to conform in some way with the functionality of A.

In this section, we consider a stronger form of equivalence and refinement. Specifically, equivalence will mean that given a particular sequence of input valuations, the two machines produce the same output valuations.

**Example 14.4:** The garage counter of Figure 3.4, discussed in Example 3.4, is type equivalent to the extended state machine version in Figure 3.8. The actor model is shown below:



However, these two machines are equivalent in a much stronger sense than simply type equivalence. These two machines behave in exactly the same way, as viewed from the outside. Given the same input sequence, the two machines will produce the same output sequence.

Consider a port p of a state machine with type  $V_p$ . This port will have a sequence of values from the set  $V_p \cup \{absent\}$ , one value at each reaction. We can represent this sequence as a function of the form

 $s_p \colon \mathbb{N} \to V_p \cup \{absent\}.$ 

This is the signal received on that port (if it is an input) or produced on that port (if it is an output). Recall that a behavior of a state machine is an assignment of such a signal to each port of such a machine. Recall further that the language L(M) of a state machine M is the set of all behaviors for that state machine. Two machines are said to be **language** equivalent if they have the same language.

**Example 14.5:** A behavior of the garage counter is a sequence of *present* and *absent* valuations for the two inputs, *up* and *down*, paired with the corresponding output sequence at the output port, *count*. A specific example is given in Example 3.16. This is a behavior of both Figures 3.4 and 3.8. All behaviors of Figure 3.4 are also behaviors of 3.8 and vice versa. These two machines are language equivalent.

In the case of a nondeterministic machine M, two distinct behaviors may share the same input signals. That is, given an input signal, there is more than one possible output se-



Figure 14.2: Three state machines where (a) and (b) have the same language, and that language is contained by that of (c).

quence. The language L(M) includes all possible behaviors. Just like deterministic machines, two nondeterministic machines are language equivalent if they have the same language.

Suppose that for two state machines A and B,  $L(A) \subset L(B)$ . That is, B has behaviors that A does not have. This is called **language containment**. A is said to be a **language refinement** of B. Just as with type refinement, language refinement makes an assertion about the suitability of A as a replacement for B. If every behavior of B is acceptable to an environment, then every behavior of A will also be acceptable to that environment. A can substitute for B.

## **Finite Sequences and Accepting States**

A complete execution of the FSMs considered in this text is infinite. Suppose that we are interested in only the finite executions. To do this, we introduce the notion of an **accepting state**, indicated with a double outline as in state b in the example below:





Let  $L_a(M)$  denote the subset of the language L(M) that results from executions that terminate in an accepting state. Equivalently,  $L_a(M)$  includes only those behaviors in L(M) with an infinite tail of stuttering reactions that remain in an accepting state. All such executions are effectively finite, since after a finite number of reactions, the inputs and outputs will henceforth be *absent*, or in LTL, FG $\neg p$  for every port p.

We call  $L_a(M)$  the **language accepted by an FSM** M. A behavior in  $L_a(M)$  specifies for each port p a finite **string**, or a finite sequence of values from the type  $V_p$ . For the above example, the input strings (1), (1,0,1), (1,0,1,0,1), etc., are all in  $L_a(M)$ . So are versions of these with an arbitrary finite number of *absent* values between any two present values. When there is no ambiguity, we can write these strings 1, 101, 10101, etc.

In the above example, in all behaviors in  $L_a(M)$ , the output is present a finite number of times, in the same reactions when the input is present.

The state machines in this text are receptive, meaning that at each reaction, each input port p can have any value in its type  $V_p$  or be *absent*. Hence, the language L(M) of the machine above includes all possible sequences of input valuations.  $L_a(M)$  excludes any of these that do not leave the machine in an accepting state. For example, any input sequence with two 1's in a row and the infinite sequence  $(1, 0, 1, 0, \cdots)$  are in L(M) but not in  $L_a(M)$ .

Note that it is sometimes useful to consider language containment when referring to the language *accepted* by the state machine, rather than the language that gives all behaviors of the state machine.

Accepting states are also called **final states**, since for any behavior in  $L_a(M)$ , it is the last state of the machine. Accepting states are further explored in Exercise 2.

## **Regular Languages and Regular Expressions**

A **language** is a set of sequences of values from some set called its **alphabet**. A language accepted by an FSM is called a **regular language**. A classic example of a language that is not regular has sequences of the form  $0^n 1^n$ , a sequence of n zeros followed by n ones. It is easy to see that no *finite* state machine can accept this language because the machine would have to count the zeros to ensure that the number of ones matches. And the number of zeros is not bounded. On the other hand, the input sequences accepted by the FSM in the box on page 384, which have the form  $10101 \cdots 01$ , are regular.

A **regular expression** is a notation for describing regular languages. A central feature of regular expressions is the **Kleene star** (or **Kleene closure**), named after the American mathematician Stephen Kleene (who pronounced his name KLAY-nee). The notation  $V^*$ , where V is a set, means the set of all finite sequences of elements from V. For example, if  $V = \{0, 1\}$ , then  $V^*$  is a set that includes the **empty sequence** (often written  $\lambda$ ), and every finite sequence of zeros and ones.

The Kleene star may be applied to sets of sequences. For example, if  $A = \{00, 11\}$ , then A\* is the set of all finite sequences where zeros and ones always appear in pairs. In the notation of regular expressions, this is written (00|11)\*, where the vertical bar means "or." What is inside the parentheses defines the set A.

Regular expressions are sequences of symbols from an alphabet and sets of sequences. Suppose our alphabet is  $A = \{a, b, \dots, z\}$ , the set of lower-case characters. Then grey is a regular expression denoting a single sequence of four characters. The expression grey | gray denotes a set of two sequences. Parentheses can be used to group sequences or sets of sequences. For example, (grey) | (gray) and gr (e|a) y mean the same thing.

Regular expressions also provide convenience notations to make them more compact and readable. For example, the + operator means "one or more," in contrast to the Kleene star, which means "zero or more." For example, a+ specifies the sequences a, aa, aaa, etc.; it is the same as a(a\*). The ? operator species "zero or one." For example, colou?r specifies a set with two sequences, color and colour; it is the same as  $colo(\lambda | u)$  r, where  $\lambda$  denotes the empty sequence.

Regular expressions are commonly used in software systems for pattern matching. A typical implementation provides many more convenience notations than the ones illustrated here.

**Example 14.6:** Machines  $M_1$  and  $M_2$  in Figure 14.2 are language equivalent. Both machines produce output  $1, 1, 0, 1, 1, 0, \cdots$ , possibly interspersed with *absent* if the input is absent in some reactions.

Machine  $M_3$ , however, has more behaviors. It can produce any output sequence that  $M_1$  and  $M_2$  can produce, but it can also produce other outputs given the same inputs. Thus,  $M_1$  and  $M_2$  are both language refinements of  $M_3$ .

Language containment assures that an abstraction is sound with respect to LTL formulas about input and output sequences. That is, if A is a language refinement of B, then any LTL formula about inputs and outputs that holds for B also holds for A.

**Example 14.7:** Consider again the machines in Figure 14.2.  $M_3$  might be a specification. For example, if we require that any two output values 0 have at

## **Probing Further: Omega Regular Languages**

The regular languages discussed in the boxes on pages 384 and 385 contain only finite sequences. But embedded systems most commonly have infinite executions. To extend the idea of regular languages to infinite runs, we can use a **Büchi automaton**, named after Julius Richard Büchi, a Swiss logician and mathematician. A Büchi automaton is a possibly nondeterministic FSM that has one or more accepting states. The language accepted by the FSM is defined to be the set of behaviors that visit one or more of the accepting states infinitely often; in other words, these behaviors satisfy the LTL formula  $\mathbf{GF}(s_1 \lor \cdots \lor s_n)$ , where  $s_1, \cdots, s_n$  are the accepting states. Such a language is called an **omega-regular language** or  $\omega$ -regular language, a generalization of regular languages. The reason for using  $\omega$  in the name is because  $\omega$  is used to construct infinite sequences, as explained in the box on page 500.

As we will see in Chapter 15, many model checking questions can be expressed by giving a Büchi automaton and then checking to see whether the  $\omega$ -regular language it defines contains any sequences.

least one intervening 1, then  $M_3$  is a suitable specification of this requirement. This requirement can be written as an LTL formula as follows:

$$\mathbf{G}((y=0) \Rightarrow \mathbf{X}((y\neq 0)\mathbf{U}(y=1))).$$

If we prove that this property holds for  $M_3$ , then we have implicitly proved that it also holds for  $M_1$  and  $M_2$ .

We will see in the next section that language containment is *not sound* with respect to LTL formulas that refer to states of the state machines. In fact, language containment does not require the state machines to have the same states, so an LTL formula that refers to the states of one machine may not even apply to the other machine. A sound abstraction that references states will require simulation.

Language containment is sometimes called **trace containment**, but here the term "trace" refers only to the observable trace, not to the execution trace. As we will see next, things get much more subtle when considering execution traces.

## 14.4 Simulation

Two nondeterministic FSMs may be language equivalent but still have observable differences in behavior in some environments. Language equivalence merely states that given the same sequences of input valuations, the two machines are *capable* of producing the same sequences of output valuations. However, as they execute, they make choices allowed by the nondeterminism. Without being able to see into the future, these choices could result in one of the machines getting into a state where it can no longer match the outputs of the other.

When faced with a nondeterministic choice, each machine is free to use any policy to make that choice. Assume that the machine cannot see into the future; that is, it cannot anticipate future inputs, and it cannot anticipate future choices that any other machine will make. For two machines to be equivalent, we will require that each machine be able to make choices that allow it to match the reaction of the other machine (producing the same outputs), and further allow it to continue to do such matching in the future. It turns out that language equivalence is not strong enough to ensure that this is possible.

**Example 14.8:** Consider the two state machines in Figure 14.3. Suppose that  $M_2$  is acceptable in some environment (every behavior it can exhibit in that environment is consistent with some specification or design intent). Is it safe for  $M_1$  to replace  $M_2$ ? The two machines are language equivalent. In all behaviors, the output is one of two finite strings, 01 or 00, for both machines. So it would seem that  $M_1$  can replace  $M_2$ . But this is not necessarily the case.

Suppose we compose each of the two machines with its own copy of the environment that finds  $M_2$  acceptable. In the first reaction where x is *present*,  $M_1$  has no choice but to take the transition to state b and produce the output y = 0. However,  $M_2$  must choose between f and h. Whichever choice it makes,  $M_2$  matches the output y = 0 of  $M_1$  but enters a state where it is no longer able to always match the outputs of  $M_1$ . If  $M_1$  can observe the state of  $M_2$  when making its choice, then in the second reaction where x is *present*, it can choose a transition that  $M_2$ can *never* match. Such a policy for  $M_1$  ensures that the behavior of  $M_1$ , given the same inputs, is never the same as the behavior of  $M_2$ . Hence, it is not safe to replace  $M_2$  with  $M_1$ .

On the other hand, if  $M_1$  is acceptable in some environment, is it safe for  $M_2$  to replace  $M_1$ ? What it means for  $M_1$  to be acceptable in the environment is that whatever decisions it makes are acceptable. Thus, in the second reaction where x is *present*, both outputs y = 1 and y = 0 are acceptable. In this second reaction,  $M_2$  has no choice but to produce one or the other these outputs, and it will inevitably transition to a state where it continues to match the outputs of  $M_1$  (henceforth forever *absent*). Hence it is safe for  $M_2$  to replace  $M_1$ .

In the above example, we can think of the machines as maliciously trying to make  $M_1$  look different from  $M_2$ . Since they are free to use any policy to make choices, they are free to use policies that are contrary to our goal to replace  $M_2$  with  $M_1$ . Note that the machines do not need to know the future; it is sufficient to simply have good visibility of the present. The question that we address in this section is: under what circumstances can we assure that there is no policy for making nondeterministic choices that can make machine  $M_1$  observably different from  $M_2$ ? The answer is a stronger form of equivalence called bisimulation and a refinement relation called simulation. We begin with the simulation relation.



Figure 14.3: Two state machines that are language equivalent but where  $M_2$  does not simulate  $M_1$  ( $M_1$  does simulate  $M_2$ ).

## 14.4.1 Simulation Relations

First, notice that the situation given in Example 14.8 is not symmetric. It is safe for  $M_2$  to replace  $M_1$ , but not the other way around. Hence,  $M_2$  is a refinement of  $M_1$ , in a sense that we will now establish.  $M_1$ , on the other hand, is not a refinement of  $M_2$ .

The particular kind of refinement we now consider is a **simulation refinement**. The following statements are all equivalent:

- $M_2$  is a simulation refinement of  $M_1$ .
- $M_1$  simulates  $M_2$ .
- $M_1$  is a simulation abstraction of  $M_2$ .

Simulation is defined by a matching game. To determine whether  $M_1$  simulates  $M_2$ , we play a game where  $M_2$  gets to move first in each round. The game starts with both machines in their initial states.  $M_2$  moves first by reacting to an input valuation. If this involves a nondeterministic choice, then it is allowed to make any choice. Whatever it choses, an output valuation results and  $M_2$ 's turn is over.

It is now  $M_1$ 's turn to move. It must react to the same input valuation that  $M_2$  reacted to. If this involves a nondeterministic choice, then it must make a choice that matches the output valuation of  $M_2$ . If there are multiple such choices, it must select one without knowledge of the future inputs or future moves of  $M_2$ . Its strategy should be to choose one that enables it to continue to match  $M_2$ , regardless of what future inputs arrive or future decisions  $M_2$  makes.

Machine  $M_1$  "wins" this matching game  $(M_1 \text{ simulates } M_2)$  if it can always match the output symbol of machine  $M_2$  for all possible input sequences. If in any reaction  $M_2$  can produce an output symbol that  $M_1$  cannot match, then  $M_1$  does not simulate  $M_2$ .

**Example 14.9:** In Figure 14.3,  $M_1$  simulates  $M_2$  but not vice versa. To see this, first play the game with  $M_2$  moving first in each round.  $M_1$  will always be able to match  $M_2$ . Then play the game with  $M_1$  moving first in each round.  $M_2$  will not always be able to match  $M_1$ . This is true even though the two machines are language equivalent.

Interestingly, if  $M_1$  simulates  $M_2$ , it is possible to compactly record all possible games over all possible inputs. Let  $S_1$  be the states of  $M_1$  and  $S_2$  be the states of  $M_2$ . Then a simulation relation  $S \subseteq S_2 \times S_1$  is a set of pairs of states occupied by the two machines in each round of the game for all possible inputs. This set summarizes all possible plays of the game.

**Example 14.10:** In Figure 14.3,

$$S_1 = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$$

and

$$S_2 = \{\mathsf{e},\mathsf{f},\mathsf{g},\mathsf{h},\mathsf{i}\}.$$

The simulation relation showing that  $M_1$  simulates  $M_2$  is

 $S = \{(\mathsf{e},\mathsf{a}), (\mathsf{f},\mathsf{b}), (\mathsf{h},\mathsf{b}), (\mathsf{g},\mathsf{c}), (\mathsf{i},\mathsf{d})\}$ 

First notice that the pair (e, a) of initial states is in the relation, so the relation includes the state of the two machines in the first round. In the second round,  $M_2$  may be in either f or h, and  $M_1$  will be in b. These two possibilities are also accounted for. In the third round and beyond,  $M_2$  will be in either g or i, and  $M_1$  will be in c or d.

There is no simulation relation showing that  $M_2$  simulates  $M_1$ , because it does not.

A simulation relation is complete if it includes all possible plays of the game. It must therefore account for all reachable states of  $M_2$ , the machine that moves first, because  $M_2$ 's moves are unconstrained. Since  $M_1$ 's moves are constrained by the need to match  $M_2$ , it is not necessary to account for all of its reachable states.

## 14.4.2 Formal Model

Using the formal model of nondeterministic FSMs given in Section 3.5.1, we can formally define a simulation relation. Let

 $M_1 = (States_1, Inputs, Outputs, possible Updates_1, initialState_1),$ 

and

 $M_2 = (States_2, Inputs, Outputs, possible Updates_2, initialState_2).$ 

Assume the two machines are type equivalent. If either machine is deterministic, then its *possibleUpdates* function always returns a set with only one element in it. If  $M_1$  simulates  $M_2$ , the simulation relation is given as a subset of  $States_2 \times States_1$ . Note the ordering here; the machine that moves first in the game,  $M_2$ , the one being simulated, is first in  $States_2 \times States_1$ .

To consider the reverse scenario, if  $M_2$  simulates  $M_1$ , then the relation is given as a subset of  $States_1 \times States_2$ . In this version of the game  $M_1$  must move first.

We can state the "winning" strategy mathematically. We say that  $M_1$  simulates  $M_2$  if there is a subset  $S \subseteq States_2 \times States_1$  such that

1.  $(initialState_2, initialState_1) \in S$ , and

2. If (s<sub>2</sub>, s<sub>1</sub>) ∈ S, then ∀ x ∈ Inputs, and ∀ (s'<sub>2</sub>, y<sub>2</sub>) ∈ possibleUpdates<sub>2</sub>(s<sub>2</sub>, x), there is a (s'<sub>1</sub>, y<sub>1</sub>) ∈ possibleUpdates<sub>1</sub>(s<sub>1</sub>, x) such that:
(a) (s'<sub>2</sub>, s'<sub>1</sub>) ∈ S, and

(b) 
$$y_2 = y_1$$
.

This set S, if it exists, is called the **simulation relation**. It establishes a correspondence between states in the two machines. If it does not exist, then  $M_1$  does not simulate  $M_2$ .

#### 14.4.3 Transitivity

Simulation is **transitive**, meaning that if  $M_1$  simulates  $M_2$  and  $M_2$  simulates  $M_3$ , then  $M_1$  simulates  $M_3$ . In particular, if we are given simulation relations  $S_{2,1} \subseteq States_2 \times States_1$ ( $M_1$  simulates  $M_2$ ) and  $S_{3,2} \subseteq States_3 \times States_2$  ( $M_2$  simulates  $M_3$ ), then

$$S_{3,1} = \{(s_3, s_1) \in States_3 \times States_1 \mid \text{ there exists } s_2 \in States_2 \text{ where } (s_3, s_2) \in S_{3,2} \text{ and } (s_2, s_1) \in S_{2,1} \}$$

**Example 14.11:** For the machines in Figure 14.2, it is easy to show that (c) simulates (b) and that (b) simulates (a). Specifically, the simulation relations are

$$S_{a,b} = \{(\mathsf{a}, \mathsf{ad}), (\mathsf{b}, \mathsf{be}), (\mathsf{c}, \mathsf{cf}), (\mathsf{d}, \mathsf{ad}), (\mathsf{e}, \mathsf{be}), (\mathsf{f}, \mathsf{cf})\}.$$

and

 $S_{b,c} = \{(ad, ad), (be, bcef), (cf, bcef)\}.$ 

By transitivity, we can conclude that (c) simulates (a), and that the simulation relation is

$$S_{a,c} = \{(a, ad), (b, bcef), (c, bcef), (d, ad), (e, bcef), (f, bcef)\},\$$

which further supports the suggestive choices of state names.
#### 14.4.4 Non-Uniqueness of Simulation Relations

When a machine  $M_1$  simulates another machine  $M_2$ , there may be more than one simulation relation.

**Example 14.12:** In Figure 14.4, it is easy to check that  $M_1$  simulates  $M_2$ . Note that  $M_1$  is nondeterministic, and in two of its states it has two distinct ways of matching the moves of  $M_2$ . It can arbitrarily choose from among these possibilities to match the moves. If from state b it always chooses to return to state a, then the simulation relation is

$$S_{2,1} = \{(ac, a), (bd, b)\}.$$

Otherwise, if from state c it always chooses to return to state b, then the simulation relation is

 $S_{2,1} = \{(\mathsf{ac}, \mathsf{a}), (\mathsf{bd}, \mathsf{b}), (\mathsf{ac}, \mathsf{c})\}.$ 

Otherwise, the simulation relation is

$$S_{2,1} = \{(ac, a), (bd, b), (ac, c), (bd, d)\}.$$

All three are valid simulation relations, so the simulation relation is not unique.

#### 14.4.5 Simulation vs. Language Containment

As with all abstraction-refinement relations, simulation is typically used to relate a simpler specification  $M_1$  to a more complicated realization  $M_2$ . When  $M_1$  simulates  $M_2$ , then the language of  $M_1$  contains the language of  $M_2$ , but the guarantee is stronger than language containment. This fact is summarized in the following theorem.

**Theorem 14.1.** Let  $M_1$  simulate  $M_2$ . Then

$$L(M_2) \subseteq L(M_1).$$

# **input:** *x*: pure **output:** *y*: {0,1}



Figure 14.4: Two state machines that simulate each other, where there is more than one simulation relation.

**Proof.** This theorem is easy to prove. Consider a behavior  $(x, y) \in L(M_2)$ . We need to show that  $(x, y) \in L(M_1)$ .

Let the simulation relation be S. Find all possible execution traces for  $M_2$ 

$$((x_0, s_0, y_0), (x_1, s_1, y_1), (x_2, s_2, y_2), \cdots),$$

that result in behavior (x, y). (If  $M_2$  is deterministic, then there will be only one execution trace.) The simulation relation assures us that we can find an execution trace for  $M_1$ 

 $((x_0, s'_0, y_0), (x_1, s'_1, y_1), (x_2, s'_2, y_2), \cdots),$ 

where  $(s_i, s'_i) \in S$ , such that given input valuation  $x_i$ ,  $M_1$  produces  $y_i$ . Thus,  $(x, y) \in L(M_1)$ .

One use of this theorem is to show that  $M_1$  does not simulate  $M_2$  by showing that  $M_2$  has behaviors that  $M_1$  does not have.

**Example 14.13:** For the examples in Figure 14.2,  $M_2$  does not simulate  $M_3$ . To see this, just note that the language of  $M_2$  is a strict subset of the language of  $M_3$ ,

$$L(M_2) \subset L(M_3),$$

meaning that  $M_3$  has behaviors that  $M_2$  does not have.

It is important to understand what the theorem says, and what it does not say. It does not say, for example, that if  $L(M_2) \subseteq L(M_1)$  then  $M_1$  simulates  $M_2$ . In fact, this statement is not true, as we have already shown with the examples in Figure 14.3. These two machines have the same language. The two machines are observably different despite the fact that their input/output behaviors are the same.

Of course, if  $M_1$  and  $M_2$  are deterministic and  $M_1$  simulates  $M_2$ , then their languages are identical and  $M_2$  simulates  $M_1$ . Thus, the simulation relation differs from language containment only for nondeterministic FSMs.

## 14.5 Bisimulation

It is possible to have two machines  $M_1$  and  $M_2$  where  $M_1$  simulates  $M_2$  and  $M_2$  simulates  $M_1$ , and yet the machines are observably different. Note that by the theorem in the previous section, the languages of these two machines must be identical.

**Example 14.14:** Consider the two machines in Figure 14.5. These two machines simulate each other, with simulation relations as follows:

$$S_{2,1} = \{(\mathsf{e},\mathsf{a}), (\mathsf{f},\mathsf{b}), (\mathsf{h},\mathsf{b}), (\mathsf{j},\mathsf{b}), (\mathsf{g},\mathsf{c}), (\mathsf{i},\mathsf{d}), (\mathsf{k},\mathsf{c}), (\mathsf{m},\mathsf{d})\}$$

 $(M_1 \text{ simulates } M_2)$ , and

$$S_{1,2} = \{(\mathsf{a},\mathsf{e}),(\mathsf{b},\mathsf{j}),(\mathsf{c},\mathsf{k}),(\mathsf{d},\mathsf{m})\}$$

 $(M_2 \text{ simulates } M_1)$ . However, there is a situation in which the two machines will be observably different. In particular, suppose that the policies for making the



Figure 14.5: An example of two machines where  $M_1$  simulates  $M_2$ , and  $M_2$  simulates  $M_1$ , but they are not bisimilar.

nondeterministic choices for the two machines work as follows. In each reaction, they flip a coin to see which machine gets to move first. Given an input valuation, that machine makes a choice of move. The machine that moves second must be able to match all of its possible choices. In this case, the machines can end up in a state where one machine can no longer match all the possible moves of the other.

Specifically, suppose that in the first move  $M_2$  gets to move first. It has three possible moves, and  $M_1$  will have to match all three. Suppose it chooses to move to f or h. In the next round, if  $M_1$  gets to move first, then  $M_2$  can no longer match all of its possible moves.

Notice that this argument does not undermine the observation that these machines simulate each other. If in each round,  $M_2$  always moves first, then  $M_1$  will always be able to match its every move. Similarly, if in each round  $M_1$  moves first, then  $M_2$  can always match its every move (by always choosing to move to j in the first round). The observable difference arises from the ability to alternate which machines moves first.

To ensure that two machines are observably identical in all environments, we need a stronger equivalence relation called **bisimulation**. We say that  $M_1$  is **bisimilar** to  $M_2$  (or  $M_1$  **bisimulates**  $M_2$ ) if we can play the matching game modified so that in each round either machine can move first.

As in Section 14.4.2, we can use the formal model of nondeterministic FSMs to define a bisimulation relation. Let

$$M_1$$
 = (States<sub>1</sub>, Inputs, Outputs, possibleUpdates<sub>1</sub>, initialState<sub>1</sub>), and  $M_2$  = (States<sub>2</sub>, Inputs, Outputs, possibleUpdates<sub>2</sub>, initialState<sub>2</sub>).

Assume the two machines are type equivalent. If either machine is deterministic, then its *possibleUpdates* function always returns a set with only one element in it. If  $M_1$ bisimulates  $M_2$ , the simulation relation is given as a subset of  $States_2 \times States_1$ . The ordering here is not important because if  $M_1$  bisimulates  $M_2$ , then  $M_2$  bisimulates  $M_1$ .

We say that  $M_1$  bisimulates  $M_2$  if there is a subset  $S \subseteq States_2 \times States_1$  such that

- 1.  $(initialState_2, initialState_1) \in S$ , and
- 2. If  $(s_2, s_1) \in S$ , then  $\forall x \in Inputs$ , and  $\forall (s'_2, y_2) \in possibleUpdates_2(s_2, x)$ , there is a  $(s'_1, y_1) \in possibleUpdates_1(s_1, x)$  such that:
  - (a)  $(s'_2, s'_1) \in S$ , and
  - (b)  $y_2 = y_1$ , and
- 3. If  $(s_2, s_1) \in S$ , then  $\forall x \in Inputs$ , and  $\forall (s'_1, y_1) \in possibleUpdates_1(s_1, x)$ , there is a  $(s'_2, y_2) \in possibleUpdates_2(s_2, x)$  such that:
  - (a)  $(s'_2, s'_1) \in S$ , and

(b) 
$$y_2 = y_1$$
.

This set S, if it exists, is called the **bisimulation relation**. It establishes a correspondence between states in the two machines. If it does not exist, then  $M_1$  does not bisimulate  $M_2$ .

## 14.6 Summary

In this chapter, we have considered three increasingly strong abstraction-refinement relations for FSMs. These relations enable designers to determine when one design can safely replace another, or when one design correctly implements a specification. The first relation is type refinement, which considers only the existence of input and output ports and their data types. The second relation is language refinement, which considers the sequences of valuations of inputs and outputs. The third relation is simulation, which considers the state trajectories of the machines. In all three cases, we have provided both a refinement relation and an equivalence relation. The strongest equivalence relation is bisimulation, which ensures that two nondeterministic FSMs are indistinguishable from each other.

## **Exercises**

- 1. In Figure 14.6 are four pairs of actors. For each pair, determine whether
  - A and B are type equivalent,
  - A is a type refinement of B,
  - B is a type refinement of A, or
  - none of the above.



Figure 14.6: Four pairs of actors whose type refinement relationships are explored in Exercise 1.

- 2. In the box on page 384, a state machine M is given that accepts finite inputs x of the form (1), (1, 0, 1), (1, 0, 1, 0, 1), etc.
  - (a) Write a regular expression that describes these inputs. You may ignore stuttering reactions.
  - (b) Describe the output sequences in  $L_a(M)$  in words, and give a regular expression for those output sequences. You may again ignore stuttering reactions.
  - (c) Create a state machine that accepts *output* sequences of the form (1), (1, 0, 1), (1, 0, 1, 0, 1), etc. (see box on page 384). Assume the input x is pure and that whenever the input is present, a present output is produced. Give a deterministic solution if there is one, or explain why there is no deterministic solution. What *input* sequences does your machine accept?
- 3. The state machine in Figure 14.7 has the property that it outputs at least one 1 between any two 0's. Construct a two-state nondeterministic state machine that simulates this one and preserves that property. Give the simulation relation. Are the machines bisimilar?
- 4. Consider the FSM in Figure 14.8, which recognizes an input code. The state machine in Figure 14.9 also recognizes the same code, but has more states than the one in Figure 14.8. Show that it is equivalent by giving a bisimulation relation with the machine in Figure 14.8.
- 5. Consider the state machine in Figure 14.10. Find a bisimilar state machine with only two states, and give the bisimulation relation.



Figure 14.7: Machine that outputs at least one 1 between any two 0's.

input: x: {0,1}
output: recognize: pure



Figure 14.8: A machine that implements a code recognizer. It outputs *recognize* at the end of every input subsequence 1100; otherwise it outputs *absent*.



Figure 14.9: A machine that implements a recognizer for the same code as in Figure 14.8, but has more states.

#### Lee & Seshia, Introduction to Embedded Systems

**input:** *x*: {0,1}

- 6. You are told that state machine A has one input x, and one output y, both with type  $\{1, 2\}$ , and that it has states  $\{a, b, c, d\}$ . You are told nothing further. Do you have enough information to construct a state machine B that simulates A? If so, give such a state machine, and the simulation relation.
- 7. Consider a state machine with a pure input x, and output y of type  $\{0, 1\}$ . Assume the states are

$$States = \{a, b, c, d, e, f\}$$

and the initial state is *a*. The *update* function is given by the following table (ignoring stuttering):

(currentState, input)	(nextState, output)
(a,x)	(b, 1)
(b,x)	(c, 0)
(c,x)	(d,0)
(d,x)	(e,1)
(e,x)	(f,0)
(f,x)	(a, 0)

- (a) Draw the state transition diagram for this machine.
- (b) Ignoring stuttering, give all possible behaviors for this machine.
- (c) Find a state machine with three states that is bisimilar to this one. Draw that state machine, and give the bisimulation relation.
- 8. For each of the following questions, give a short answer and justification.

**input:** *x*: pure **output:** *y*: {0,1}



Figure 14.10: A machine that has more states than it needs.

- (a) TRUE or FALSE: Consider a state machine A that has one input x, and one output y, both with type {1, 2} and a single state s, with two self loops labeled true/1 and true/2. Then for any state machine B which has exactly the same inputs and outputs (along with types), A simulates B.
- (b) TRUE or FALSE: Suppose that f is an arbitrary LTL formula that holds for state machine A, and that A simulates another state machine B. Then we can safely assert that f holds for B.
- (c) TRUE or FALSE: Suppose that A are B are two type-equivalent state machines, and that f is an LTL formula where the atomic propositions refer only to the inputs and outputs of A and B, not to their states. If the LTL formula f holds for state machine A, and A simulates state machine B, then f holds for B.

# Reachability Analysis and Model Checking

15.1	Open a	and Closed Systems	 •	•	 •	• •	405
15.2	Reacha	ability Analysis		•	 •	• •	406
	15.2.1	Verifying $\mathbf{G}p$					407
	15.2.2	Explicit-State Model Checking					409
	15.2.3	Symbolic Model Checking					411
15.3	Abstra	ction in Model Checking			 •	• •	413
15.4	Model	Checking Liveness Properties			 •	• •	417
	15.4.1	Properties as Automata					418
	15.4.2	Finding Acceptance Cycles					420
15.5	Summ	ary		•			423
	Sideba	: Probing Further: Model Checking in Practice					424
Exer	cises .		 •	•	 •	• •	425

Chapters 13 and 14 have introduced techniques for formally specifying properties and models of systems, and for comparing such models. In this chapter, we will study algorithmic techniques for **formal verification** — the problem of checking whether a system satisfies its formal specification in its specified operating environment. In particular, we study a technique called **model checking**. Model checking is an algorithmic method for determining whether a system satisfies a formal specification expressed as a temporal

logic formula. It was introduced by Clarke and Emerson (1981) and Queille and Sifakis (1981), which earned the creators the 2007 ACM Turing Award, the highest honor in Computer Science.

Central to model checking is the notion of the set of reachable states of a system. **Reachability analysis** is the process of computing the set of reachable states of a system. This chapter presents basic algorithms and ideas in reachability analysis and model checking. These algorithms are illustrated using examples drawn from embedded systems design, including verification of high-level models, sequential and concurrent software, as well as control and robot path planning. Model checking is a large and active area of research, and a detailed treatment of the subject is out of the scope of this chapter; we refer the interested reader to Clarke et al. (1999) and Holzmann (2004) for an in-depth introduction to this field.

# 15.1 Open and Closed Systems

A **closed system** is one with no inputs. An **open system**, in contrast, is one that maintains an ongoing interaction with its environment by receiving inputs and (possibly) generating output to the environment. Figure 15.1 illustrates these concepts.

Techniques for formal verification are typically applied to a model of the closed system M obtained by composing the model of the system S that is to be verified with a model of its environment E. S and E are typically open systems, where all inputs to S are generated by E and vice-versa. Thus, as shown in Figure 15.2, there are three inputs to the verification process:

- A model of the system to be verified, S;
- A model of the environment, E, and



Figure 15.1: Open and closed systems.



Figure 15.2: Formal verification procedure.

• The property to be verified  $\Phi$ .

The verifier generates as output a YES/NO answer, indicating whether or not S satisfies the property  $\Phi$  in environment E. Typically, a NO output is accompanied by a counterexample, also called an **error trace**, which is a trace of the system that indicates how  $\Phi$  is violated. Counterexamples are very useful aids in the debugging process. Some formal verification tools also include a proof or certificate of correctness with a YES answer; such an output can be useful for **certification** of system correctness.

The form of composition used to combine system model S with environment model E depends on the form of the interaction between system and environment. Chapters 5 and 6 describe several ways to compose state machine models. All of these forms of composition can be used in generating a verification model M from S and E. Note that M can be nondeterministic.

For simplicity, in this chapter we will assume that system composition has already been performed using one of the techniques presented in Chapters 5 and 6. All algorithms discussed in the following sections will operate on the combined verification model M, and will be concerned with answering the question of whether M satisfies property  $\Phi$ . Additionally, we will assume that  $\Phi$  is specified as a property in linear temporal logic.

# 15.2 Reachability Analysis

We consider first a special case of the model checking problem which is useful in practice. Specifically, we assume that M is a finite-state machine and  $\Phi$  is an LTL formula of the form  $\mathbf{G}p$ , where p is a proposition. Recall from Chapter 13 that  $\mathbf{G}p$  is the temporal logic formula that holds in a trace when the proposition p holds in every state of that trace. As we have seen in Chapter 13, several system properties are expressible as Gp properties.

We will begin in Section 15.2.1 by illustrating how computing the reachable states of a system enables one to verify a Gp property. In Section 15.2.2 we will describe a technique for reachability analysis of finite-state machines based on explicit enumeration of states. Finally, in Section 15.2.3, we will describe an alternative approach to analyze systems with very large state spaces.

## 15.2.1 Verifying Gp

In order for a system M to satisfy  $\mathbf{G}p$ , where p is a proposition, every trace exhibitable by M must satisfy  $\mathbf{G}p$ . This property can be verified by enumerating all states of M and checking that every state satisfies p.

When M is finite-state, in theory, such enumeration is always possible. As shown in Chapter 3, the state space of M can be viewed as a directed graph where the nodes of the graph correspond to states of M and the edges correspond to transitions of M. This graph is called the **state graph** of M, and the set of all states is called its state space. With this graph-theoretic viewpoint, one can see that checking  $\mathbf{G}p$  for a finite-state system M corresponds to traversing the state graph for M, starting from the initial state and checking that every state reached in this traversal satisfies p. Since M has a finite number of states, this traversal must terminate.

**Example 15.1:** Let the system S be the traffic light controller of Figure 3.10 and its environment E be the pedestrian model shown in Figure 3.11. Let M be the synchronous composition of S and E as shown in Figure 15.3. Observe that M is a closed system. Suppose that we wish to verify that M satisfies the property

$$\mathbf{G} \neg (\mathsf{green} \land \mathsf{crossing})$$

In other words, we want to verify that it is never the case that the traffic light is green while pedestrians are crossing.

The composed system M is shown in Figure 15.4 as an extended FSM. Note that M has no inputs or outputs. M is finite-state, with a total of 188 states (using a similar calculation to that in Example 3.12). The graph in Figure 15.4 is not



Figure 15.3: Composition of traffic light controller (Figure 3.10) and pedestrian model (Figure 3.11).

the full state graph of M, because each node represents a set of states, one for each different value of *count* in that node. However, through visual inspection of this graph we can check for ourselves that no state satisfies the proposition (green  $\land$  crossing), and hence every trace satisfies the LTL property  $\mathbf{G} \neg (\text{green} \land$ crossing).

In practice, the seemingly simple task of verifying whether a finite-state system M satisfies a Gp property is not as straightforward as in the previous example for the following reasons:

• Typically, one starts only with the initial state and transition function, and the state graph must be constructed on the fly.



Figure 15.4: Extended state machine obtained from synchronous-reactive composition of traffic light controller and pedestrian models. Note that this is nondeterministic.

• The system might have a huge number of states, possibly exponential in the size of the syntactic description of M. As a consequence, the state graph cannot be represented using traditional data structures such as an adjacency or incidence matrix.

The next two sections describe how these challenges can be handled.

## 15.2.2 Explicit-State Model Checking

In this section, we discuss how to compute the reachable state set by generating and traversing the state graph on the fly.

First, recall that the system of interest M is closed, finite-state, and can be nondeterministic. Since M has no inputs, its set of possible next states is a function of its current state alone. We denote this transition relation of M by  $\delta$ , which is only a function of the current state of M, in contrast to the *possibleUpdates* function introduced in Chapter 3 which is also a function of the current input. Thus,  $\delta(s)$  is the set of possible next states from state s of M.

Algorithm 15.1 computes the set of reachable states of M, given its initial state  $s_0$  and transition relation  $\delta$ . Procedure **DFS\_Search** performs a depth-first traversal of the state graph of M, starting with state  $s_0$ . The graph is generated on-the-fly by repeatedly applying  $\delta$  to states visited during the traversal.

 $\begin{array}{ll} {\rm Input} & : {\rm Initial state} \; s_0 \; {\rm and} \; {\rm transition} \; {\rm relation} \; \delta \; {\rm for} \; {\rm closed} \; {\rm finite-state} \\ & {\rm system} \; M \\ {\rm Output:} \; {\rm Set} \; R \; {\rm of} \; {\rm reachable} \; {\rm states} \; {\rm of} \; M \end{array}$ 

**1 Initialize:** Stack  $\Sigma$  to contain a single state  $s_0$ ; Current set of reached states  $R := \{s_0\}$ .

2 DFS\_Search() { **3 while** *Stack*  $\Sigma$  *is not empty* **do** Pop the state s at the top of  $\Sigma$ 4 Compute  $\delta(s)$ , the set of all states reachable from s in one 5 transition for each  $s' \in \delta(s)$  do 6 if  $s' \notin R$  then 7  $R := R \cup \{s'\}$ 8 Push s' onto  $\Sigma$ 9 end 10 end 11 12 end 13 }

Algorithm 15.1: Computing the reachable state set by depth-first explicit-state search.

The main data structures required by the algorithm are  $\Sigma$ , the stack storing the current path in the state graph being explored from  $s_0$ , and R, the current set of states reached during traversal. Since M is finite-state, at some point all states reachable from  $s_0$  will be in R, which implies that no new states will be pushed onto  $\Sigma$  and thus  $\Sigma$  will become empty. Hence, procedure **DFS\_Search** terminates and the value of R at the end of the procedure is the set of all reachable states of M.

The space and time requirements for this algorithm are linear in the size of the state graph (see Appendix B for an introduction to such complexity notions). However, the number of nodes and edges in the state graph of M can be exponential in the size of the descriptions of S and E. For example, if S and E together have 100 Boolean state variables (a small

number in practice!), the state graph of M can have a total of  $2^{100}$  states, far more than what contemporary computers can store in main memory. Therefore, explicit-state search algorithms such as **DFS\_Search** must be augmented with **state compression** techniques. Some of these techniques are reviewed in the sidebar on page 424.

A challenge for model checking concurrent systems is the **state-explosion problem**. Recall that the state space of a composition of k finite-state systems  $M_1, M_2, \ldots, M_k$  (say, using synchronous composition), is the cartesian product of the state spaces of  $M_1, M_2, \ldots, M_k$ . In other words, if  $M_1, M_2, \ldots, M_k$  have  $n_1, n_2, \ldots, n_k$  states respectively, their composition can have  $\prod_{i=1}^k n_i$  states. It is easy to see that the number of states of a concurrent composition of k components grows exponentially with k. Explicitly representing the state space of the composite system does not scale. In the next section, we will introduce techniques that can mitigate this problem in some cases.

## 15.2.3 Symbolic Model Checking

The key idea in **symbolic model checking** is to represent a set of states *symbolically* as a propositional logic formula, rather than explicitly as a collection of individual states. Specialized data structures are often used to efficiently represent and manipulate such formulas. Thus, in contrast to explicit-state model checking, in which individual states are manipulated, symbolic model checking operates on sets of states.

Algorithm 15.2 (Symbolic\_Search) is a symbolic algorithm for computing the set of reachable states of a closed, finite-state system M. This algorithm has the same inputoutput specification as the previous explicit-state algorithm DFS\_Search; however, all operations in Symbolic\_Search are set operations.

In algorithm **Symbolic\_Search**, R represents the entire set of states reached at any point in the search, and  $R_{new}$  represents the *new* states generated at that point. When no more new states are generated, the algorithm terminates, with R storing all states reachable from  $s_0$ . The key step of the algorithm is line 5, in which  $R_{new}$  is computed as the set of all states s' reachable from any state s in R in one step of the transition relation  $\delta$ . This operation is called **image computation**, since it involves computing the image of the function  $\delta$ . Efficient implementations of image computation that directly operate on propositional logic formulas are central to symbolic reachability algorithms. Apart from image computation, the key set operations in **Symbolic\_Search** include set union and emptiness checking.

Input : Initial state s<sub>0</sub> and transition relation δ for closed finite-state system M, represented symbolically
Output: Set R of reachable states of M, represented symbolically
1 Initialize: Current set of reached states R = {s<sub>0</sub>}

2 Symbolic\_Search() { 3  $R_{\text{new}} = R$ 4 while  $R_{\text{new}} \neq \emptyset$  do 5  $| R_{\text{new}} := \{s' \mid \exists s \in R \text{ s.t. } s' \in \delta(s) \land s' \notin R\}$ 6  $| R := R \cup R_{\text{new}}$ 7 end 8 }

Algorithm 15.2: Computing the reachable state set by symbolic search.

**Example 15.2:** We illustrate symbolic reachability analysis using the finite-state system in Figure 15.4.

To begin with, we need to introduce some notation. Let  $v_l$  be a variable denoting the state of the traffic light controller FSM S at the start of each reaction; i.e.,  $v_l \in \{\text{green}, \text{yellow}, \text{red}, \text{pending}\}$ . Similarly, let  $v_p$  denote the state of the pedestrian FSM E, where  $v_p \in \{\text{crossing}, \text{none}, \text{waiting}\}$ .

Given this notation, the initial state set  $\{s_0\}$  of the composite system M is represented as the following propositional logical formula:

$$v_l = \text{red} \land v_p = \text{crossing} \land count = 0$$

From  $s_0$ , the only enabled outgoing transition is the self-loop on the initial state of the extended FSM in Figure 15.4. Thus, after one step of reachability computation, the set of reached states R is represented by the following formula:

 $v_l = \mathsf{red} \land v_p = \mathsf{crossing} \land 0 \le count \le 1$ 

After two steps, R is given by

 $v_l = \mathsf{red} \land v_p = \mathsf{crossing} \land 0 \le count \le 2$ 

and after k steps,  $k \leq 60$ , R is represented by the formula

 $v_l = \operatorname{red} \land v_p = \operatorname{crossing} \land 0 \le count \le k$ 

On the 61st step, we exit the state (red, crossing), and compute R as

 $v_l = \text{red} \land v_p = \text{crossing} \land 0 \le count \le 60$  $\lor v_l = \text{green} \land v_p = \text{none} \land count = 0$ 

Proceeding similarly, the set of reachable states R is grown until there is no further change. The final reachable set is represented as:

 $\begin{array}{l} v_l = \operatorname{red} \wedge v_p = \operatorname{crossing} \wedge 0 \leq count \leq 60 \\ \lor v_l = \operatorname{green} \wedge v_p = \operatorname{none} \wedge 0 \leq count \leq 60 \\ \lor v_l = \operatorname{pending} \wedge v_p = \operatorname{waiting} \wedge 0 < count \leq 60 \\ \lor v_l = \operatorname{yellow} \wedge v_p = \operatorname{waiting} \wedge 0 \leq count \leq 5 \end{array}$ 

In practice, the symbolic representation is much more compact than the explicit one. The previous example illustrates this nicely because a large number of states are compactly represented by inequalities like  $0 < count \le 60$ . Computer programs can be designed to operate directly on the symbolic representation. Some examples of such programs are given in the box on page 424.

Symbolic model checking has been used successfully to address the state-explosion problem for many classes of systems, most notably for hardware models. However, in the worst case, even symbolic set representations can be exponential in the number of system variables.

# 15.3 Abstraction in Model Checking

A challenge in model checking is to work with the simplest abstraction of a system that will provide the required proofs of safety. Simpler abstractions have smaller state spaces and can be checked more efficiently. The challenge, of course, is to know what details to omit from the abstraction.

The part of the system to be abstracted away depends on the property to be verified. The following example illustrates this point.

**Example 15.3:** Consider the traffic light system M in Figure 15.4. Suppose that, as in Example 15.1 we wish to verify that M satisfies the property

 $\mathbf{G} \neg (\text{green} \land \text{crossing})$ 

Suppose we abstract the variable *count* away from M by hiding all references to *count* from the model, including all guards mentioning it and all updates to it. This generates the abstract model  $M_{\rm abs}$  shown in Figure 15.5.

We observe that this abstract  $M_{\rm abs}$  exhibits more behaviors than M. For instance, from the state (yellow, waiting) we can take the self-loop transition forever, staying in that state perennially, even though in the actual system M this state must be exited within five clock ticks. Moreover, every behavior of M can be exhibited by  $M_{\rm abs}$ .

The interesting point is that, even with this approximation, we can prove that  $M_{abs}$  satisfies **G**  $\neg$ (green  $\land$  crossing). The value of *count* is irrelevant for this property.

Notice that while M has 188 states,  $M_{abs}$  has only 4 states. Reachability analysis on  $M_{abs}$  is far easier than for M as we have far fewer states to explore.

There are several ways to compute an abstraction. One of the simple and extremely useful approaches is called **localization reduction** or **localization abstraction** (Kurshan (1994)). In localization reduction, parts of the design model that are irrelevant to the property being checked are abstracted away by hiding a subset of state variables. Hiding a variable corresponds to freeing that variable to evolve arbitrarily. It is the form of abstraction used in Example 15.3 above, where *count* is allowed to change arbitrarily, and all transitions are made independent of the value of *count*.

**Example 15.4:** Consider the multithreaded program given below (adapted from Ball et al. (2001)). The procedure lock\_unlock executes a loop within which it acquires a lock, then calls the function randomCall, based on whose result it either releases the lock and executes another loop iteration, or it quits the loop (and then releases the lock). The execution of another loop iteration is



Figure 15.5: Abstraction of the traffic light system in Figure 15.4.

ensured by incrementing new, so that the condition old != new evaluates to true.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
1
    unsigned int old, new;
2
3
    void lock_unlock() {
4
      do {
5
        pthread_mutex_lock(&lock);
6
        old = new;
7
        if (randomCall()) {
8
          pthread_mutex_unlock(&lock);
9
          new++;
10
11
      } while (old != new)
      pthread_mutex_unlock(&lock);
13
    }
14
```

Suppose the property we want to verify is that the code does not attempt to call pthread\_mutex\_lock twice in a row. Recall from Section 11.2.4 how the system can deadlock if a thread becomes permanently blocked trying to acquire a lock. This could happen in the above example if the thread, already holding lock lock, attempts to acquire it again.

If we model this program exactly, without any abstraction, then we need to reason about all possible values of old and new, in addition to the remaining state of the program. Assuming a word size of 32 in this system, the size of the state space is roughly  $2^{32} \times 2^{32} \times n$ , where  $2^{32}$  is the number of values of old and new, and n denotes the size of the remainder of the state space.

However, it is not necessary to reason about the precise values of old and new to prove that this program is correct. Assume, for this example, that our programming language is equipped with a boolean type. Assume further that the program can perform nondeterministic assignments. Then, we can generate the following abstraction of the original program, written in C-like syntax, where the Boolean variable b represents the predicate old == new.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
1
    boolean b; // b represents the predicate (old == new)
2
    void lock_unlock() {
3
4
      do {
          pthread_mutex_lock(&lock);
5
          b = true;
6
          if (randomCall()) {
7
            pthread_mutex_unlock(&lock);
8
            b = false;
9
10
          }
11
      } while (!b)
      pthread_mutex_unlock(&lock);
12
13
   }
```

It is easy to see that this abstraction retains just enough information to show that the program satisfies the desired property. Specifically, the lock will not be acquired twice because the loop is only iterated if b is set to false, which implies that the lock was released before the next attempt to acquire.

Moreover, observe that size of the state space to be explored has reduced to simply 2n. This is the power of using the "right" abstraction.

A major challenge for formal verification is to *automatically* compute simple abstractions. An effective and widely-used technique is **counterexample-guided abstraction refinement** (**CEGAR**), first introduced by Clarke et al. (2000). The basic idea (when using localization reduction) is to start by hiding almost all state variables except those referenced by the temporal logic property. The resulting abstract system will have more behaviors than the original system. Therefore, if this abstract system satisfies an LTL formula  $\Phi$  (i.e., each of its behaviors satisfies  $\Phi$ ), then so does the original. However, if the abstract system does not satisfy  $\Phi$ , the model checker generates a counterexample. If this counterexample is a counterexample for the original system, the process terminates, having found a genuine counterexample. Otherwise, the CEGAR approach analyzes this counterexample to infer which hidden variables must be made visible, and with these additional variables, recomputes an abstraction. The process continues, terminating either with some abstract system being proven correct, or generating a valid counterexample for the original system.

The CEGAR approach and several follow-up ideas have been instrumental in driving progress in the area of software model checking. We review some of the key ideas in the sidebar on page 424.

# 15.4 Model Checking Liveness Properties

So far, we have restricted ourselves to verifying properties of the form  $\mathbf{G}p$ , where p is an atomic proposition. An assertion that  $\mathbf{G}p$  holds for all traces is a very restricted kind of safety property. However, as we have seen in Chapter 13, several useful system properties are not safety properties. For instance, the property stating that "the robot must visit location A" is a liveness property: if visiting location A is represented by proposition q, then this property is an assertion that  $\mathbf{F}q$  must hold for all traces. In fact, several problems, including path planning problems for robotics and progress properties of distributed and concurrent systems can be stated as liveness properties. It is therefore useful to extend model checking to handle this class of properties.

Properties of the form  $\mathbf{F}p$ , though liveness properties, can be partially checked using the techniques introduced earlier in this chapter. Recall from Chapter 13 that  $\mathbf{F}p$  holds for a trace if and only if  $\neg \mathbf{G} \neg p$  holds for the same trace. In words, "p is true some time in the future" iff " $\neg p$  is always false." Therefore, we can attempt to verify that the system satisfies  $\mathbf{G} \neg p$ . If the verifier asserts that  $\mathbf{G} \neg p$  holds for all traces, then we know that  $\mathbf{F}p$  does not hold for any trace. On the other hand, if the verifier outputs "NO", then the accompanying counterexample provides a witness exhibiting how p may become true eventually. This witness provides one trace for which  $\mathbf{F}p$  holds, but it does not prove that  $\mathbf{F}p$  holds for all traces (unless the machine is deterministic).

More complete checks and more complicated liveness properties require a more sophisticated approach. Briefly, one approach used in explicit-state model checking of LTL properties is as follows:

- 1. Represent the negation of the property  $\Phi$  as an automaton *B*, where certain states are labeled as accepting states.
- 2. Construct the synchronous composition of the property automaton B and the system automaton M. The accepting states of the property automaton induce accepting states of the product automaton  $M_B$ .
- 3. If the product automaton  $M_B$  can visit an accepting state infinitely often, then it indicates that M does not satisfy  $\Phi$ ; otherwise, M satisfies  $\Phi$ .

The above approach is known as the **automata-theoretic approach to verification**. We give a brief introduction to this subject in the rest of this section. Further details may be found in the seminal papers on this topic (Wolper et al. (1983); Vardi and Wolper (1986)) and the book on the SPIN model checker (Holzmann (2004)).

## 15.4.1 Properties as Automata

Consider the first step of viewing properties as automata. Recall the material on omegaregular languages introduced in the box on page 386. The theory of Büchi automata and omega-regular languages, briefly introduced there, is relevant for model checking liveness properties. Roughly speaking, an LTL property  $\Phi$  has a one-to-one correspondence with a set of behaviors that satisfy  $\Phi$ . This set of behaviors constitutes the language of the Büchi automaton corresponding to  $\Phi$ .

For the LTL model checking approach we describe here, if  $\Phi$  is the property that the system must satisfy, then we represent its negation  $\neg \Phi$  as a Büchi automaton. We present some illustrative examples below.

**Example 15.5:** Suppose that an FSM  $M_1$  models a system that executes forever and produces a pure output h (for heartbeat), and that it is required to produce this output at least once every three reactions. That is, if in two successive reactions it fails to produce the output h, then in the third it must.

We can formulate this property in LTL as the property  $\Phi_1$  below:

$$\mathbf{G}(h \lor \mathbf{X}h \lor \mathbf{X}^2h)$$

and the negation of this property is

$$\mathbf{F}(\neg h \land \mathbf{X} \neg h \land \mathbf{X}^2 \neg h)$$

The Büchi automaton  $B_1$  corresponding to the negation of the desired property is given below:



Let us examine this automaton. The language accepted by this automaton includes all behaviors that enter and stay in state d. Equivalently, the language includes all behaviors that produce a *present* output on f in some reaction. When we compose the above machine with  $M_1$ , if the resulting composite machine can never produce f = present, then the language accepted by the composite machine is empty. If we can prove that the language is empty, then we have proved that M produces the heartbeat h at least once every three reactions.

Observe that the property  $\Phi_1$  in the above example is in fact a safety property. We give an example of a liveness property below.

**Example 15.6:** Suppose that the FSM  $M_2$  models a controller for a robot that must locate a room and stay there forever. Let p be the proposition that becomes true when the robot is in the target room. Then, the desired property  $\Phi_2$  can be expressed in LTL as FGp.

The negation of this property is  $\mathbf{GF}\neg p$ . The Büchi automaton  $B_2$  corresponding to this negated property is given below:



Notice that all accepting behaviors of  $B_2$  correspond to those where  $\neg p$  holds infinitely often. These behaviors correspond to a cycle in the state graph for the product automaton where state b of  $B_2$  is visited repeatedly. This cycle is known as an **acceptance cycle**.

Liveness properties of the form  $\mathbf{G} \mathbf{F} p$  also occur naturally as specifications. This form of property is useful in stating **fairness** properties which assert that certain desirable properties hold infinitely many times, as illustrated in the following example.

**Example 15.7:** Consider a traffic light system such as that in Example 3.10. We may wish to assert that the traffic light becomes green infinitely many times in any execution. In other words, the state green is visited infinitely often, which can be expressed as  $\Phi_3 = \mathbf{GF}$  green.

The automaton corresponding to  $\Phi_3$  is identical to that for the negation of  $\Phi_2$  in Example 15.6 above with  $\neg p$  replaced by green. However, in this case the accepting behaviors of this automaton are the desired behaviors.

Thus, from these examples we see that the problem of detecting whether a certain accepting state *s* in an FSM can be visited infinitely often is the workhorse of explicit-state model checking of LTL properties. We next present an algorithm for this problem.

## 15.4.2 Finding Acceptance Cycles

We consider the following problem:

Given a finite-state system M, can an accepting state  $s_a$  of M be visited infinitely often?

Put another way, we seek an algorithm to check whether (i) state  $s_a$  is reachable from the initial state  $s_0$  of M, and (ii)  $s_a$  is reachable from itself. Note that asking whether a state *can* be visited infinitely often is not the same as asking whether it *must* be visited infinitely often.

The graph-theoretic viewpoint is useful for this problem, as it was in the case of  $\mathbf{G}p$  discussed in Section 15.2.1. Assume for the sake of argument that we have the entire state graph constructed a priori. Then, the problem of checking whether state  $s_a$  is reachable from  $s_0$  is simply a graph traversal problem, solvable for example by depth-first search (DFS). Further, the problem of detecting whether  $s_a$  is reachable from itself amounts to checking whether there is a cycle in the state graph containing that state.

The main challenges for solving this problem are similar to those discussed in Section 15.2.1: we must perform this search on-the-fly, and we must deal with large state spaces.

The **nested depth-first search** (nested DFS) algorithm, which is implemented in the SPIN model checker (Holzmann (2004)), solves this problem and is shown as Algorithm 15.3. The algorithm begins by calling the procedure called Nested\_DFS\_Search with argument 1, as shown in the Main function at the bottom.  $M_B$  is obtained by composing the original closed system M with the automaton B representing the negation of LTL formula  $\Phi$ .

As the name suggests, the idea is to perform two depth-first searches, one nested inside the other. The first DFS identifies a path from initial state  $s_0$  to the target accepting state  $s_a$ . Then, from  $s_a$  we start another DFS to see if we can reach  $s_a$  again. The variable mode is either 1 or 2 depending on whether we are performing the first DFS or the second. Stacks  $\Sigma_1$  and  $\Sigma_2$  are used in the searches performed in modes 1 and 2 respectively. If  $s_a$  is encountered in the second DFS, the algorithm generates as output the path leading from  $s_0$  to  $s_a$  with a loop on  $s_a$ . The path from  $s_0$  to  $s_a$  is obtained simply by reading off the contents of stack  $\Sigma_1$ . Likewise, the cycle from  $s_a$  to itself is obtained from stack  $\Sigma_2$ . Otherwise, the algorithm reports failure.

Search optimization and state compression techniques that are used in explicit-state reachability analysis can be used with nested DFS also. Further details are available in Holzmann (2004).

**Input** : Initial state  $s_0$  and transition relation  $\delta$  for automaton  $M_B$ ; Target accepting state  $s_a$  of  $M_B$ 

**Output**: Acceptance cycle containing  $s_a$ , if one exists

**1 Initialize:** (i) Stack  $\Sigma_1$  to contain a single state  $s_0$ , and stack  $\Sigma_2$  to be empty; (ii) Two sets of reached states  $R_1 := R_2 := \{s_0\}$ ; (iii) Flag found := false.

```
2 Nested_DFS_Search(Mode mode) {
 3 while Stack \Sigma_{mode} is not empty do
        Pop the state s at the top of \Sigma_{mode}
 4
        if (s = s_a \text{ and } mode = 1) then
 5
             Push s onto \Sigma_2
 6
             Nested_DFS_Search(2)
 7
             return
 8
        end
 9
        Compute \delta(s), the set of all states reachable from s in one transition
10
        for each s' \in \delta(s) do
11
             if (s' = s_a \text{ and } mode = 2) then
12
                 Output path to s_a with acceptance cycle using contents of
13
                  stacks \Sigma_1 and \Sigma_2
                  found := true
14
                  return
15
             end
16
             if s' \notin R_{mode} then
17
                  R_{\text{mode}} := R_{\text{mode}} \cup \{s'\}
18
                  Push s' onto \Sigma_{mode}
19
                  Nested_DFS_Search(mode)
20
             end
21
        end
22
23 end
24 }
25 Main() {
        Nested_DFS_Search(1)
26
        if (found = false) then Output "no acceptance cycle with s_a" end }
27
28
```



# 15.5 Summary

This chapter gives some basic algorithms for formal verification, including model checking, a technique for verifying if a finite-state system satisfies a property specified in temporal logic. Verification operates on closed systems, which are obtained by composing a system with its operating environment. The first key concept is that of reachability analysis, which verifies properties of the form  $\mathbf{G}p$ . The concept of abstraction, central to the scalability of model checking, is also discussed in this chapter. This chapter also shows how explicit-state model checking algorithms can handle liveness properties, where a crucial concept is the correspondence between properties and automata.

### **Probing Further: Model Checking in Practice**

Several tools are available for computing the set of reachable states of a finite-state system and checking that they satisfy specifications in temporal logic. One such tool is **SMV** (symbolic model verifier), which was first developed at Carnegie Mellon University by Kenneth McMillan. SMV was the first model checking tool to use binary decision diagrams (**BDD**s), a compact data structure introduced by Bryant (1986) for representing a Boolean function. The use of BDDs has proved instrumental in enabling analysis of more complex systems. Current symbolic model checkers also rely heavily on Boolean satisfiability (SAT) solvers (see Malik and Zhang (2009)), which are programs for deciding whether a propositional logic formula can evaluate to true. One of the first uses of SAT solvers in model checking was for **bounded model checking** (see Biere et al. (1999)), where the transition relation of the system is unrolled only a bounded number of times. A few different versions of SMV are available online (see for example http://nusmv.fbk.eu/).

The SPIN model checker (Holzmann, 2004) developed in the 1980's and 1990's at Bell Labs by Gerard Holzmann and others, is another leading tool for model checking (see http://www.spinroot.com/). Rather than directly representing models as communicating FSMs, it uses a specification language (called Promela, for process meta language) that enables specifications that closely resemble multithreaded programs. SPIN incorporates state-compression techniques such as **hash compaction** (the use of hashing to reduce the size of the stored state set) and **partial-order reduction** (a technique to reduce the number of reachable states to be explored by considering only a subset of the possible process interleavings).

Automatic abstraction has played a big role in applying model checking directly to software. An example of abstraction-based software model checking is the **SLAM** system developed at Microsoft Research (Ball and Rajamani, 2001; Ball et al., 2011). SLAM combines CEGAR with a particular form of abstraction called predicate abstraction, in which predicates in a program are abstracted to Boolean variables. A key step in these techniques is checking whether a counterexample generated on the abstract model is in fact a true counterexample. This check is performed using satisfiability solvers for logics richer than propositional logic. These solvers are called **SAT-based decision procedures** or **satisfiability modulo theories (SMT**) solvers (for more details, see Barrett et al. (2009)).

More recently, techniques based on **inductive learning**, that is, generalization from sample data, have started playing an important role in formal verification (see Seshia (2015) for an exposition of this topic).

## **Exercises**

1. Consider the system M modeled by the hierarchical state machine of Figure 13.2, which models an interrupt-driven program.

Model M in the modeling language of a verification tool (such as SPIN). You will have to construct an environment model that asserts the interrupt. Use the verification tool to check whether M satisfies  $\phi$ , the property stated in Exercise 5:

 $\phi$ : The main program eventually reaches program location C.

Explain the output you obtain from the verification tool.

2. Figure 15.3 shows the synchronous-reactive composition of the traffic light controller of Figure 3.10 and the pedestrian model of Figure 3.11.

Consider replacing the pedestrian model in Figure 15.3 with the alternative model given below where the initial state is nondeterministically chosen to be one of none or crossing:



- (a) Model the composite system in the modeling language of a verification tool (such as SPIN). How many reachable states does the combined system have? How many of these are initial states?
- (b) Formulate an LTL property stating that every time a pedestrian arrives, eventually the pedestrian is allowed to cross (i.e., the traffic light enters state red).
- (c) Use the verification tool to check whether the model constructed in part (a) satisfies the LTL property specified in part (b). Explain the output of the tool.

3. The notion of reachability has a nice symmetry. Instead of describing all states that are reachable from some initial state, it is just as easy to describe all states from which some state can be reached. Given a finite-state system M, the **backward** reachable states of a set F of states is the set B of all states from which some state in F can be reached. The following algorithm computes the set of backward reachable states for a given set of states F:

**Input** : A set F of states and transition relation  $\delta$  for closed finite-state system M **Output:** Set B of backward reachable states from F in M

```
1 Initialize: B := F

2 B_{\text{new}} := B

3 while B_{\text{new}} \neq \emptyset do

4 \begin{vmatrix} B_{\text{new}} := \{s \mid \exists s' \in B \text{ s.t. } s' \in \delta(s) \land s \notin B \}

5 \begin{vmatrix} B := B \cup B_{\text{new}} \end{vmatrix}

6 end
```

Explain how this algorithm can check the property  $\mathbf{G}p$  on M, where p is some property that is easily checked for each state s in M. You may assume that M has exactly one initial state  $s_0$ .

**16** Quantitative Analysis

16.1	Proble	ms of Interest
	16.1.1	Extreme-Case Analysis
	16.1.2	Threshold Analysis
	16.1.3	Average-Case Analysis
16.2	Progra	ums as Graphs
	16.2.1	Basic Blocks
	16.2.2	Control-Flow Graphs
	16.2.3	Function Calls
16.3	Factor	s Determining Execution Time
	16.3.1	Loop Bounds
	16.3.2	Exponential Path Space
	16.3.3	Path Feasibility
	16.3.4	Memory Hierarchy
16.4	<b>Basics</b>	of Execution Time Analysis
	16.4.1	Optimization Formulation
	16.4.2	Logical Flow Constraints
	16.4.3	Bounds for Basic Blocks
16.5	Other	Quantitative Analysis Problems
	16.5.1	Memory-bound Analysis
	16.5.2	Power and Energy Analysis
16.6	Summ	ary
	Sideba	r: Tools for Execution-Time Analysis
Exer	cises .	

Will my brake-by-wire system actuate the brakes within one millisecond? Answering this question requires, in part, an **execution-time analysis** of the software that runs on the electronic control unit (ECU) for the brake-by-wire system. Execution time of the software is an example of a **quantitative property** of an embedded system. The constraint that the system actuate the brakes within one millisecond is a **quantitative constraint**. The analysis of quantitative properties for conformance with quantitative constraints is central to the correctness of embedded systems and is the topic of the present chapter.

A quantitative property of an embedded system is any property that can be measured. This includes physical parameters, such as position or velocity of a vehicle controlled by the embedded system, weight of the system, operating temperature, power consumption, or reaction time. Our focus in this chapter is on properties of software-controlled systems, with particular attention to execution time. We present program analysis techniques that can ensure that execution time constraints will be met. We also discuss how similar techniques can be used to analyze other quantitative properties of software, particularly resource usage such as power, energy, and memory.

The analysis of quantitative properties requires adequate models of both the software components of the system and of the environment in which the software executes. The environment includes the processor, operating system, input-output devices, physical components with which the software interacts, and (if applicable) the communication network. The environment is sometimes also referred to as the platform on which the software executes. Providing a comprehensive treatment of execution time analysis would require much more than one chapter. The goal of this chapter is more modest. We illustrate key features of programs and their environment that must be considered in quantitative analysis, and we describe qualitatively some analysis techniques that are used. For concreteness, we focus on a single quantity, *execution time*, and only briefly discuss other resource-related quantitative properties.

## 16.1 Problems of Interest

The typical quantitative analysis problem involves a software task defined by a program P, the environment E in which the program executes, and the quantity of interest q. We assume that q can be given by a function of  $f_P$  as follows,

$$q = f_P(x, w)$$
where x denotes the inputs to the program P (such as data read from memory or from sensors, or data received over a network), and w denotes the environment parameters (such as network delays or the contents of the cache when the program begins executing). Defining the function  $f_P$  completely is often neither feasible nor necessary; instead, practical quantitative analysis will yield extreme values for q (highest or lowest values), average values for q, or proofs that q satisfies certain threshold constraints. We elaborate on these next.

### 16.1.1 Extreme-Case Analysis

In extreme-case analysis, we may want to estimate the *largest value* of q for all values of x and w,

$$\max_{x,w} f_P(x,w). \tag{16.1}$$

Alternatively, it can be useful to estimate the *smallest value* of q:

$$\min_{x,w} f_P(x,w). \tag{16.2}$$

If q represents execution time of a program or a program fragment, then the largest value is called the **worst-case execution time (WCET)**, and the smallest value is called the **best-case execution time (BCET)**. It may be difficult to determine these numbers exactly, but for many applications, an upper bound on the WCET or a lower bound on the BCET is all that is needed. In each case, when the computed bound equals the actual WCET or BCET, it is said to be a **tight bound**; otherwise, if there is a considerable gap between the actual value and the computed bound, it is said to be a **loose bound**. Computing loose bounds may be much easier than finding tight bounds.

### 16.1.2 Threshold Analysis

A **threshold property** asks whether the quantity q is always bounded above or below by a threshold T, for any choice of x and w. Formally, the property can be expressed as

$$\forall x, w, \quad f_P(x, w) \le T \tag{16.3}$$

or

$$\forall x, w, \quad f_P(x, w) \ge T \tag{16.4}$$

#### Lee & Seshia, Introduction to Embedded Systems

429

Threshold analysis may provide assurances that a quantitative constraint is met, such as the requirement that a brake-by-wire system actuate the brakes within one millisecond.

Threshold analysis may be easier to perform than extreme-case analysis. Unlike extremecase analysis, threshold analysis does not require us to determine the maximum or minimum value exactly, or even to find a tight bound on these values. Instead, the analysis is provided some guidance in the form of the target value T. Of course, it might be possible to use extreme-case analysis to check a threshold property. Specifically, Constraint 16.3 holds if the WCET does not exceed T, and Constraint 16.4 holds if the BCET is not less than T.

## 16.1.3 Average-Case Analysis

Often one is interested more in typical resource usage rather than in worst-case scenarios. This is formalized as average-case analysis. Here, the values of input x and environment parameter w are assumed to be drawn randomly from a space of possible values X and W according to probability distributions  $\mathcal{D}_x$  and  $\mathcal{D}_w$  respectively. Formally, we seek to estimate the value

$$\mathbb{E}_{\mathcal{D}_x, \mathcal{D}_w} f_P(x, w) \tag{16.5}$$

where  $\mathbb{E}_{\mathcal{D}_x, \mathcal{D}_w}$  denotes the expected value of  $f_P(x, w)$  over the distributions  $\mathcal{D}_x$  and  $\mathcal{D}_w$ .

One difficulty in average-case analysis is to define realistic distributions  $\mathcal{D}_x$  and  $\mathcal{D}_w$  that capture the true distribution of inputs and environment parameters with which a program will execute.

In the rest of this chapter, we will focus on a single representative problem, namely, WCET estimation.

# 16.2 Programs as Graphs

A fundamental abstraction used often in program analysis is to represent a program as a graph indicating the flow of control from one code segment to another. We will illustrate this abstraction and other concepts in this chapter using the following running example:

**Example 16.1:** Consider the function modexp that performs modular exponentiation, a key step in many cryptographic algorithms. In modular exponentiation, given a base b, an exponent e, and a modulus m, one must compute  $b^e \mod m$ . In the program below, base, exponent and mod represent b, e and m respectively. EXP\_BITS denotes the number of bits in the exponent. The function uses a standard shift-square-accumulate algorithm, where the base is repeatedly squared, once for each bit position of the exponent, and the base is accumulated into the result only if the corresponding bit is set.

```
#define EXP_BITS 32
1
2
   typedef unsigned int UI;
3
4
   UI modexp(UI base, UI exponent, UI mod) {
5
     int i;
6
     UI result = 1;
7
8
     i = EXP_BITS;
g
     while(i > 0) {
10
       if ((exponent & 1) == 1) {
11
12
          result = (result * base) % mod;
13
        }
       exponent >>= 1;
14
       base = (base * base) % mod;
15
       i--;
16
17
     }
18
     return result;
  }
19
```

### 16.2.1 Basic Blocks

A **basic block** is a sequence of consecutive program statements in which the flow of control enters only at the beginning of this sequence and leaves only at the end, without halting or the possibility of branching except at the end.

**Example 16.2:** The following three statements from the modexp function in Example 16.1 form a basic block:

```
14 exponent >>= 1;
15 base = (base * base) % mod;
16 i--;
```

Another example of a basic block includes the initializations at the top of the function, comprising lines 7 and 9:

```
7 result = 1;
8
9 i = EXP_BITS;
```

## 16.2.2 Control-Flow Graphs

A control-flow graph (CFG) of a program P is a directed graph G = (V, E), where the set of vertices V comprises basic blocks of P, and the set of edges E indicates the flow of control between basic blocks. Figure 16.1 depicts the CFG for the modexp program of Example 16.1. Each node of the CFG is labeled with its corresponding basic block. In most cases, this is simply the code as it appears in Example 16.1. The only exception is for conditional statements, such as the conditions in while loops and if statements; in these cases, we follow the convention of labeling the node with the condition followed by a question mark to indicate the conditional branch.

Although our illustrative example of a control-flow graph is at the level of C source code, it is possible to use the CFG representation at other levels of program representation as well, including a high-level model as well as low-level assembly code. The level of representation employed depends on the level of detail required by the context. To make them easier to follow, our control-flow graphs will be at the level of source code.

## 16.2.3 Function Calls

Programs are typically decomposed into several functions in order to systematically organize the code and promote reuse and readability. The control-flow graph (CFG) representation can be extended to reason about code with function calls by introducing special **call** and **return** edges. These edges connect the CFG of the **caller function** – the one making the function call – to that of the **callee function** – the one being called. A **call edge** indicates a transfer of control from the caller to the callee. A **return edge** indicates a transfer of control from the caller.



Figure 16.1: Control-flow graph for the modexp function of Example 16.1. All incoming edges at a node indicate transfer of control to the start of the basic block for that node, and all outgoing edges from a node indicate an exit from the end of the basic block for that node. For clarity, we label the outgoing edges from a branch statement with 0 or 1 indicating the flow of control in case the branch evaluates to false or true, respectively. An ID number for each basic block is noted above the node for that block; IDs range from 1 to 6 for this example.



Figure 16.2: Control-flow graphs for the  $modexp_call$  and update functions in Example 16.3. Call/return edges are indicated with dashed lines.

**Example 16.3:** A slight variant shown below of the modular exponentation program of Example 16.1 uses function calls and can be represented by the CFG with call and return edges in Figure 16.2.

```
1 #define EXP_BITS 32
2 typedef unsigned int UI;
3 UI exponent, base, mod;
4
5 UI update(UI r) {
6 UI res = r;
7 if ((exponent & 1) == 1) {
8 res = (res * base) % mod;
9 }
```

```
10
     exponent >>= 1;
11
     base = (base * base) % mod;
     return res;
12
  }
13
14
15 UI modexp_call() {
    UI result = 1; int i;
16
     i = EXP_BITS;
17
     while(i > 0) {
18
        result = update(result);
19
        i--;
20
21
     }
22
     return result;
23
  }
```

In this modified example, the variables base, exponent, and mod are global variables. The update to base and exponent in the body of the while loop, along with the computation of result is now performed in a separate function named update.

Non-recursive function calls can also be handled by **inlining**, which is the process of copying the code for the callee into that of the caller. If inlining is performed transitively for all functions called by the code that must be analyzed, the analysis can be performed on the CFG of the code resulting from inlining, without using call and return edges.

# 16.3 Factors Determining Execution Time

There are several issues one must consider in order to estimate the worst-case execution time of a program. This section outlines some of the main issues and illustrates them with examples. In describing these issues, we take a programmer's viewpoint, starting with the program structure and then considering how the environment can impact the program's execution time.

## 16.3.1 Loop Bounds

The first point one must consider when bounding the execution time of a program is whether the program terminates. Non-termination of a sequential program can arise from non-terminating loops or from an unbounded sequence of function calls. Therefore, while writing real-time embedded software, the programmer must ensure that all loops are guaranteed to terminate. In order to guarantee this, one must determine for each loop a bound on the number of times that loop will execute in the worst case. Similarly, all function calls must have bounded recursion depth. The problems of determining bounds on loop iterations or recursion depth are undecidable in general, since the halting problem for Turing machines can be reduced to either problem. (See Appendix B for an introduction to Turing machines and decidability.)

In this section, we limit ourselves to reasoning about loops. In spite of the undeciable nature of the problem, progress has been made on automatically determining loop bounds for several patterns that arise in practice. Techniques for determining loop bounds are a current research topic and a full survey of these methods is out of the scope of this chapter. We will limit ourselves to presenting illustrative examples for loop bound inference.

The simplest case is that of for loops that have a specified constant bound, as in Example 16.4 below. This case occurs often in embedded software, in part due to a discipline of programming enforced by designers who must program for real-time constraints and limited resources.

**Example 16.4:** Consider the function modexpl below. It is a slight variant of the function modexp introduced in Example 16.1 that performs modular exponentiation, in which the while loop has been expressed as an equivalent for loop.

```
#define EXP_BITS 32
1
2
  typedef unsigned int UI;
3
4
  UI modexp1(UI base, UI exponent, UI mod) {
5
     UI result = 1; int i;
6
7
     for(i=EXP_BITS; i > 0; i--) {
8
       if ((exponent & 1) == 1) {
9
10
         result = (result * base) % mod;
```

```
11  }
12  exponent >>= 1;
13  base = (base * base) % mod;
14  }
15  return result;
16  }
```

In the case of this function, it is easy to see that the for loop will take exactly EXP\_BITS iterations, where EXP\_BITS is defined as the constant 32.

In many cases, the loop bound is not immediately obvious (as it was for the above example). To make this point, here is a variation on Example 16.4.

**Example 16.5:** The function listed below also performs modular exponentiation, as in Example 16.4. However, in this case, the for loop is replaced by a while loop with a different loop condition – the loop exits when the value of exponent reaches 0. Take a moment to check whether the while loop will terminate (and if so, why).

```
typedef unsigned int UI;
1
2
  UI modexp2(UI base, UI exponent, UI mod) {
3
     UI result = 1;
4
5
     while (exponent != 0) {
6
7
       if ((exponent & 1) == 1) {
         result = (result * base) % mod;
8
q
       }
       exponent >>= 1;
10
       base = (base * base) % mod;
11
12
     }
     return result;
13
  }
14
```

Now let us analyze the reason that this loop terminates. Notice that exponent is an unsigned int, which we will assume to be 32 bits wide. If it starts out equal to 0, the loop terminates right away and the function returns result = 1. If not, in each iteration of the loop, notice that line 10 shifts exponent one bit

to the right. Since exponent is an unsigned int, after the right shift, its most significant bit will be 0. Reasoning thus, after at most 32 right shifts, all bits of exponent must be set to 0, thus causing the loop to terminate. Therefore, we can conclude that the loop bound is 32.

Let us reflect on the reasoning employed in the above example. The key component of our "proof of termination" was the observation that the number of bits of exponent decreases by 1 each time the loop executes. This is a standard argument for proving termination – by defining a **progress measure** or **ranking function** that maps each state of the program to a mathematical structure called a **well order**. Intuitively, a well order is like a program that counts down to zero from some initial value in the natural numbers.

## 16.3.2 Exponential Path Space

Execution time is a path property. In other words, the amount of time taken by the program is a function of how conditional statements in the program evaluate to true or false. A major source of complexity in execution time analysis (and other program analysis problems as well) is that the number of program paths can be very large — exponential in the size of the program. We illustrate this point with the example below.

**Example 16.6:** Consider the function count listed below, which runs over a two-dimensional array, counting and accumulating non-negative and negative elements of the array separately.

```
#define MAXSIZE 100
1
2
3 int Array[MAXSIZE][MAXSIZE];
4 int Ptotal, Pcnt, Ntotal, Ncnt;
5 . . .
6
 void count() {
     int Outer, Inner;
7
     for (Outer = 0; Outer < MAXSIZE; Outer++) {</pre>
8
       for (Inner = 0; Inner < MAXSIZE; Inner++)</pre>
                                                     {
9
         if (Array[Outer][Inner] >= 0) {
10
           Ptotal += Array[Outer][Inner];
11
```

```
Pcnt++;
12
13
           } else {
              Ntotal += Array[Outer][Inner];
14
             Ncnt++;
15
           }
16
        }
17
      }
18
   }
19
```

The function includes a nested loop. Each loop executes MAXSIZE (100) times. Thus, the inner body of the loop (comprising lines 10–16) will execute 10,000 times – as many times as the number of elements of Array. In each iteration of the inner body of the loop, the conditional on line 10 can either evaluate to true or false, thus resulting in  $2^{10000}$  possible ways the loop can execute. In other words, this program has  $2^{10000}$  paths.

Fortunately, as we will see in Section 16.4.1, one does not need to explicitly enumerate all possible program paths in order to perform execution time analysis.

## 16.3.3 Path Feasibility

Another source of complexity in program analysis is that all program paths may not be executable. A computationally expensive function is irrelevant for execution time analysis if that function is never executed.

A path p in program P is said to be **feasible** if there exists an input x to P such that P executes p on x. In general, even if P is known to terminate, determining whether a path p is feasible is a computationally intractable problem. One can encode the canonical NP-complete problem, the Boolean satisfiability problem (see Appendix B), as a problem of checking path feasibility in a specially-constructed program. In practice, however, in many cases, it is possible to determine path feasibility.

**Example 16.7:** Recall Example 13.3 of a software task from the open source Paparazzi unmanned aerial vehicle (UAV) project (Nemer et al., 2006):

```
1 #define PPRZ_MODE_AUTO2 2
2 #define PPRZ_MODE_HOME 3
3 #define VERTICAL MODE AUTO ALT 3
4 #define CLIMB MAX 1.0
5 . . .
  void altitude_control_task(void) {
6
     if (pprz_mode == PPRZ_MODE_AUTO2
7
         || pprz_mode == PPRZ_MODE_HOME) {
8
       if (vertical mode == VERTICAL MODE AUTO ALT) {
9
         float err = estimator_z - desired_altitude;
10
         desired climb
11
                = pre_climb + altitude_pgain * err;
12
         if (desired_climb < -CLIMB_MAX) {</pre>
13
           desired climb = -CLIMB MAX;
14
15
         }
         if (desired climb > CLIMB MAX) {
16
           desired_climb = CLIMB_MAX;
17
18
         }
19
       }
20
     }
   }
21
```

This program has 11 paths in all. However, the number of *feasible* program paths is only 9. To see this, note that the two conditionals desired\_climb < -CLIMB\_MAX on line 13 and desired\_climb > CLIMB\_MAX on line 16 cannot both be true. Thus, only three out of the four paths through the two innermost conditional statements are feasible. This infeasible inner path can be taken for two possible evaluations of the outermost conditional on lines 7 and 8: either if pprz\_mode == PPRZ\_MODE\_AUTO2 is true, or if that condition is false, but pprz\_mode == PPRZ\_MODE\_HOME is true.

## 16.3.4 Memory Hierarchy

The preceding sections have focused on properties of programs that affect execution time. We now discuss how properties of the execution platform, specifically of cache memories, can significantly impact execution time. We illustrate this point using Example 16.8.<sup>1</sup> The material on caches introduced in Sec. 9.2.3 is pertinent to this discussion.

<sup>&</sup>lt;sup>1</sup>This example is based on a similar example in Bryant and O'Hallaron (2003).

**Example 16.8:** Consider the function dot\_product listed below, which computes the dot product of two vectors of floating point numbers. Each vector is of dimension n, where n is an input to the function. The number of iterations of the loop depends on the value of n. However, even if we know an upper bound on n, hardware effects can still cause execution time to vary widely for similar values of n.

```
1 float dot_product(float *x, float *y, int n) {
2  float result = 0.0;
3  int i;
4  for(i=0; i < n; i++) {
5     result += x[i] * y[i];
6   }
7  return result;
8 }</pre>
```

Suppose this program is executing on a 32-bit processor with a direct-mapped cache. Suppose also that the cache can hold two sets, each of which can hold 4 floats. Finally, let us suppose that x and y are stored contiguously in memory starting with address 0.

Let us first consider what happens if n = 2. In this case, the entire arrays x and y will be in the same block and thus in the same cache set. Thus, in the very first iteration of the loop, the first access to read x[0] will be a cache miss, but thereafter every read to x[i] and y[i] will be a cache hit, yielding best case performance for loads.

Consider next what happens when n = 8. In this case, each x[i] and y[i] map to the same cache set. Thus, not only will the first access to x[0] be a miss, the first access to y[0] will also be a miss. Moreover, the latter access will evict the block containing x[0]-x[3], leading to a cache miss on x[1], x[2], and x[3] as well. The reader can see that every access to an x[i] or y[i] will lead to a cache miss.

Thus, a seemingly small change in the value of n from 2 to 8 can lead to a drastic change in execution time of this function.

# 16.4 Basics of Execution Time Analysis

Execution time analysis is a current research topic, with many problems still to be solved. There have been over two decades of research, resulting in a vast literature. We cannot provide a comprehensive survey of the methods in this chapter. Instead, we will present some of the basic concepts that find widespread use in current techniques and tools for WCET analysis. Readers interested in a more detailed treatment may find an overview in a recent survey paper (Wilhelm et al., 2008) and further details in books (e.g., Li and Malik (1999)) and book chapters (e.g., Wilhelm (2005)).

## 16.4.1 Optimization Formulation

An intuitive formulation of the WCET problem can be constructed using the view of programs as graphs. Given a program P, let G = (V, E) denote its control-flow graph (CFG). Let n = |V| be the number of nodes (basic blocks) in G, and m = |E| denote the number of edges. We refer to the basic blocks by their index i, where i ranges from 1 to n.

We assume that the CFG has a unique *start* or *source* node s and a unique *sink* or *end* node t. This assumption is not restrictive: If there are multiple start or end nodes, one can add a dummy start/end node to achieve this condition. Usually we will set s = 1 and t = n.

Let  $x_i$  denote the number of times basic block *i* is executed. We call  $x_i$  the **execution** count of basic block *i*. Let  $\mathbf{x} = (x_1, x_2, \dots, x_n)$  be a vector of variables recording execution counts. Not all valuations of **x** correspond to valid program executions. We say that **x** is valid if the elements of **x** correspond to a (valid) execution of the program. The following example illustrates this point.

**Example 16.9:** Consider the CFG for the modular exponentiation function modexp introduced in Example 16.1. There are six basic blocks in this function, labeled 1 to 6 in Figure 16.1. Thus,  $\mathbf{x} = (x_1, x_2, \dots, x_6)$ . Basic blocks 1 and 6, the start and end, are each executed only once. Thus,  $x_1 = x_6 = 1$ ; any other valuation cannot correspond to any program execution.

Next consider basic blocks 2 and 3, corresponding to the conditional branches i > 0 and (exponent & 1) == 1. One can observe that  $x_2$  must equal  $x_3 + 1$ , since the block 3 is executed every time block 2 is executed, except when the loop exits to block 6.

Along similar lines, one can see that basic blocks 3 and 5 must be executed an equal number of times.

#### **Flow Constraints**

The intuition expressed in Example 16.9 can be formalized using the theory of **network flow**, which finds use in many contexts including modeling traffic, fluid flow, and the flow of current in an electrical circuit. In particular, in our problem context, the flow must satisfy the following two properties:

1. Unit Flow at Source: The control flow from source node s = 1 to sink node t = n is a single execution and hence corresponds to unit flow from source to sink. This property is captured by the following two constraints:

$$x_1 = 1$$
 (16.6)

$$x_n = 1 \tag{16.7}$$

2. *Conservation of Flow:* For each node (basic block) *i*, the incoming flow to *i* from its predecessor nodes equals the outgoing flow from *i* to its successor nodes.

To capture this property, we introduce additional variables to record the number of times that each edge in the CFG is executed. Following the notation of Li and Malik (1999), let  $d_{ij}$  denote the number of times the edge from node *i* to node *j* in the CFG is executed. Then we require that for each node *i*,  $1 \le i \le n$ ,

$$x_i = \sum_{j \in P_i} d_{ji} = \sum_{j \in S_i} d_{ij},$$
(16.8)

where  $P_i$  is the set of predecessors to node *i* and  $S_i$  is the set of successors. For the source node,  $P_1 = \emptyset$ , so the sum over predecessor nodes is omitted. Similarly, for the sink node,  $S_n = \emptyset$ , so the sum over successor nodes is omitted.

Taken together, the two sets of constraints presented above suffice to implicitly define all source-to-sink execution paths of the program. Since this constraint-based representation is an *implicit* representation of program paths, this approach is also referred to in the literature as **implicit path enumeration** or **IPET**.

We illustrate the generation of the above constraints with an example.

**Example 16.10:** Consider again the function modexp of Example 16.1, with CFG depicted in Figure 16.1.

The constraints for this CFG are as follows:

$$\begin{array}{rcrcrcrcrc} x_1 &=& 1 \\ x_6 &=& 1 \\ x_1 &=& d_{12} \\ x_2 &=& d_{12} + d_{52} = d_{23} + d_{26} \\ x_3 &=& d_{23} = d_{34} + d_{35} \\ x_4 &=& d_{34} = d_{45} \\ x_5 &=& d_{35} + d_{45} = d_{52} \\ x_6 &=& d_{26} \end{array}$$

Any solution to the above system of equations will result in integer values for the  $x_i$  and  $d_{ij}$  variables. Furthermore, this solution will generate valid execution counts for basic blocks. For example, one such valid solution is

$$x_1 = 1, d_{12} = 1, x_2 = 2, d_{23} = 1, x_3 = 1, d_{34} = 0, d_{35} = 1,$$
  
 $x_4 = 0, d_{45} = 0, x_5 = 1, d_{52} = 1, x_6 = 1, d_{26} = 1.$ 

Readers are invited to find and examine additional solutions for themselves.

#### **Overall Optimization Problem**

We are now in a position to formulate the overall optimization problem to determine worst-case execution time. The key assumption we make in this section is that we know an upper bound  $w_i$  on the execution time of the basic block *i*. (We will later see in Section 16.4.3 how the execution time of a single basic block can be bounded.) Then the WCET is given by the maximum  $\sum_{i=1}^{n} w_i x_i$  over valid execution counts  $x_i$ .

Putting this together with the constraint formulation of the preceding section, our goal is to find values for  $x_i$  that give

$$\max_{x_i, \ 1 \le i \le n} \sum_{i=1}^n w_i x_i$$

subject to

$$x_1 = x_n = 1$$
  
$$x_i = \sum_{j \in P_i} d_{ji} = \sum_{j \in S_i} d_{ij}$$

This optimization problem is a form of a **linear programming** (**LP**) problem (also called a **linear program**), and it is solvable in polynomial time.

However, two major challenges remain:

- This formulation assumes that all source to sink paths in the CFG are feasible and does not bound loops in paths. As we have already seen in Section 16.3, this is not the case in general, so solving the above maximization problem may yield a pessimistic loose bound on the WCET. We will consider this challenge in Section 16.4.2.
- The upper bounds  $w_i$  on execution time of basic blocks *i* are still to be determined. We will briefly review this topic in Section 16.4.3.

## 16.4.2 Logical Flow Constraints

In order to ensure that the WCET optimization is not too pessimistic by including paths that cannot be executed, we must add so-called **logical flow constraints**. These constraints rule out infeasible paths and incorporate bounds on the number of loop iterations. We illustrate the use of such constraints with two examples.

### Loop Bounds

For programs with loops, it is necessary to use bounds on loop iterations to bound execution counts of basic blocks.

**Example 16.11:** Consider the modular exponentiation program of Example 16.1 for which we wrote down flow constraints in Example 16.10.

Notice that those constraints impose no upper bound on  $x_2$  or  $x_3$ . As argued in Examples 16.4 and 16.5, the bound on the number of loop iterations in this example is 32. However, without imposing this additional constraint, since there is no upper bound on  $x_2$  or  $x_3$ , the solution to our WCET optimization will be infinite, implying that there is no upper bound on the WCET. The following single constraint suffices:

 $x_3 \leq 32$ 

From this constraint on  $x_3$ , we derive the constraint that  $x_2 \leq 33$ , and also upper bounds on  $x_4$  and  $x_5$ . The resulting optimization problem will then return a finite solution, for finite values of  $w_i$ .

Adding such bounds on values of  $x_i$  does not change the complexity of the optimization problem. It is still a linear programming problem.

#### **Infeasible Paths**

Some logical flow constraints rule out combinations of basic blocks that cannot appear together on a single path.

**Example 16.12:** Consider a snippet of code from Example 16.7 describing a software task from the open source Paparazzi unmanned aerial vehicle (UAV) project (Nemer et al., 2006):

```
#define CLIMB_MAX 1.0
```

```
2
3
  void altitude_control_task(void) {
4
      . . .
      err = estimator_z - desired_altitude;
5
      desired_climb
6
             = pre_climb + altitude_pgain * err;
7
      if (desired_climb < -CLIMB_MAX) {</pre>
8
        desired_climb = -CLIMB_MAX;
9
      }
10
      if (desired_climb > CLIMB_MAX) {
11
        desired_climb = CLIMB_MAX;
12
13
14
      return;
15
  }
```

The CFG for the snippet of code shown above is given in Figure 16.3. The system of flow constraints for this CFG according to the rules in Section 16.4.1 is as follows:

$$\begin{array}{rcl} x_1 & = & 1 \\ x_5 & = & 1 \\ x_1 & = & d_{12} + d_{13} \\ x_2 & = & d_{12} = d_{23} \\ x_3 & = & d_{13} + d_{23} = d_{34} + d_{35} \\ x_4 & = & d_{34} = d_{45} \\ x_5 & = & d_{35} + d_{45} \end{array}$$

A solution for the above system of equations is

$$x_1 = x_2 = x_3 = x_4 = x_5 = 1,$$

implying that each basic block gets executed exactly once, and that both conditionals evaluate to *true*. However, as we discussed in Example 16.7, it is impossible for both conditionals to evaluate to *true*. Since CLIMB\_MAX = 1.0, if desired\_climb is less than -1.0 in basic block 1, then at the start of basic block 3 it will be set to -1.0.

The following constraint rules out the infeasible path:

$$d_{12} + d_{34} \le 1 \tag{16.9}$$



Figure 16.3: Control-flow graph for Example 16.12.

This constraint specifies that both conditional statements cannot be *true* together. It is of course possible for both conditionals to be *false*. We can check that this constraint excludes the infeasible path when added to the original system.

More formally, for a program without loops, if a set of k edges

$$(i_1, j_1), (i_2, j_2), \dots, (i_k, j_k)$$

in the CFG cannot be taken together in a program execution, the following constraint is added to the optimization problem:

$$d_{i_1j_1} + d_{i_2j_2} + \ldots + d_{i_kj_k} \le k - 1 \tag{16.10}$$

For programs with loops, the constraint is more complicated since an edge can be traversed multiple times, so the value of a  $d_{ij}$  variable can exceed 1. We omit the details in this case; the reader can consult Li and Malik (1999) for a more elaborate discussion of this topic.

In general, the constraints added above to exclude infeasible combinations of edges can change the complexity of the optimization problem, since one must also add the following **integrality** constraints:

$$x_i \in \mathbb{N}, \text{ for all } i = 1, 2, \dots, n$$
 (16.11)

$$d_{ij} \in \mathbb{N}, \text{ for all } i, j = 1, 2, \dots, n$$
 (16.12)

In the absence of such integrality constraints, the optimization solver can return fractional values for the  $x_i$  and  $d_{ij}$  variables. However, adding these constraints results in an integer linear programming (ILP) problem. The ILP problem is known to be NP-hard (see Appendix B, Section B.4). Even so, in many practical instances, one can solve these ILP problems fairly efficiently (see for example Li and Malik (1999)).

### 16.4.3 Bounds for Basic Blocks

In order to complete the optimization problem for WCET analysis, we need to compute upper bounds on the execution times of basic blocks – the  $w_i$  coefficients in the cost function of Section 16.4.1. Execution time is typically measured in CPU cycles. Generating such bounds requires detailed microarchitectural modeling. We briefly outline some of the issues in this section.

A simplistic approach to this problem would be to generate conservative upper bounds on the execution time of each instruction in the basic block, and then add up these perinstruction bounds to obtain an upper bound on the execution time of the overall basic block.

The problem with this approach is that there can be very wide variation in the execution times for some instructions, resulting in very loose upper bounds on the execution time of a basic block. For instance, consider the latency of memory instructions (loads and stores) for a system with a data cache. The difference between the latency when there is a cache miss versus a hit can be a factor of 100 on some platforms. In these cases, if the analysis does not differentiate between cache hits and misses, it is possible for the computed bound to be a hundred times larger than the execution time actually exhibited.

Several techniques have been proposed to better use program context to predict execution time of instructions more precisely. These techniques involve detailed microarchitectural modeling. We mention two main approaches below:

- Integer linear programming (ILP) methods: In this approach, pioneered by Li and Malik (1999), one adds cache constraints to the ILP formulation of Section 16.4.1. Cache constraints are linear expressions used to bound the number of cache hits and misses within basic blocks. The approach tracks the memory locations that cause cache conflicts those that map onto the same cache set, but have different tags and adds linear constraints to record the impact of such conflicts on the number of cache hits and misses. Measurement through simulation or execution on the actual platform must be performed to obtain the cycle count for hits and misses. The cost constraint of the ILP is modified to compute the program path along which the overall number of cycles, including cache hits and misses, is the largest. Further details about this approach are available in Li and Malik (1999).
- Abstract interpretation methods: Abstract interpretation is a theory of approximation of mathematical structures, in particular those that arise in defining the semantic models of computer systems (Cousot and Cousot (1977)). In particular, in abstract interpretation, one performs sound approximation, where the set of behaviors of the system is a subset of that of the model generated by abstract interpretation. In the context of WCET analysis, abstract interpretation has been used to infer invariants at program points, in order to generate loop bounds, and constraints on the state of processor pipelines or caches at the entry and exit locations of basic blocks. For example, such a constraint could specify the conditions under which variables will be available in the data cache (and hence a cache hit will result). Once such constraints in order to generate execution time estimates. Further details about this approach can be found in Wilhelm (2005).

In addition to techniques such as those described above, accurate measurement of execution time is critical for finding tight WCET bounds. Some of the measurement techniques are as follows:

1. *Sampling CPU cycle counter:* Certain processors include a register that records the number of CPU cycles elapsed since reset. For example, the **time stamp counter register** on x86 architectures performs this function, and is accessible through a

rdtsc ("read time stamp counter") instruction. However, with the advent of multicore designs and power management features, care must be taken to use such CPU cycle counters to accurately measure timing. For example, it may be necessary to lock the process to a particular CPU.

- 2. Using a logic analyzer: A logic analyzer is an electronic instrument used to measure signals and track events in a digital system. In the current context, the events of interest are the entry and exit points of the code to be timed, definable, for example, as valuations of the program counter. Logic analyzers are less intrusive than using cycle counters, since they do not require instrumenting the code, and they can be more accurate. However, the measurement setup is more complicated.
- 3. *Using a cycle-accurate simulator:* In many cases, timing analysis must be performed when the actual hardware is not yet available. In this situation, a cycle-accurate simulator of the platform provides a good alternative.

# 16.5 Other Quantitative Analysis Problems

Although we have focused mainly on execution time in this chapter, several other quantitative analysis problems are relevant for embedded systems. We briefly describe two of these in this section.

## 16.5.1 Memory-bound Analysis

Embedded computing platforms have very limited memory as compared to general-purpose computers. For example, as mentioned in Chapter 9, the Luminary Micro LM3S8962 controller has only 64 KB of RAM. It is therefore essential to structure the program so that it uses memory efficiently. Tools that analyze memory consumption and compute bounds on memory usage can be very useful.

There are two kinds of memory bound analysis that are relevant for embedded systems. In **stack size analysis** (or simply **stack analysis**), one needs to compute an upper bound on the amount of stack-allocated memory used by a program. Recall from Section 9.3.2 that stack memory is allocated whenever a function is called or an interrupt is handled. If the program exceeds the memory allocated for the stack, a stack overflow is said to occur.

If the program does not contain recursive functions and runs uninterrupted, one can bound stack usage by traversing the **call graph** of the program – the graph that tracks which functions call which others. If the space for each stack frame is known, then one can track the sequence of calls and returns along paths in the call graph in order to compute the worst-case stack size.

Performing stack size analysis for interrupt-driven software is significantly more complicated. We point the interested reader to Brylow et al. (2001).

**Heap analysis** is the other memory bound analysis problem that is relevant for embedded systems. This problem is harder than stack bound analysis since the amount of heap space used by a function might depend on the values of input data and may not be known prior to run-time. Moreover, the exact amount of heap space used by a program can depend on the implementation of dynamic memory allocation and the garbage collector.

# 16.5.2 Power and Energy Analysis

Power and energy consumption are increasingly important factors in embedded system design. Many embedded systems are autonomous and limited by battery power, so a designer must ensure that the task can be completed within a limited energy budget. Also, the increasing ubiquity of embedded computing is increasing its energy footprint, which must be reduced for sustainable development.

To first order, the energy consumed by a program running on an embedded device depends on its execution time. However, estimating execution time alone is not sufficient. For example, energy consumption depends on circuit switching activity, which can depend more strongly on the data values with which instructions are executed.

For this reason, most techniques for energy and power estimation of embedded software focus on estimating the average-case consumption. The average case is typically estimated by profiling instructions for several different data values, guided by software benchmarks. For an introduction to this topic, see Tiwari et al. (1994).

# 16.6 Summary

Quantitative properties, involving physical parameters or specifying resource constraints, are central to embedded systems. This chapter gave an introduction to basic concepts in

quantitative analysis. First, we considered various types of quantitative analysis problems, including extreme-case analysis, average-case analysis, and verifying threshold properties. As a representative example, this chapter focused on execution time analysis. Several examples were presented to illustrate the main issues, including loop bounds, path feasibility, path explosion, and cache effects. An optimization formulation that forms the backbone of execution time analysis was presented. Finally, we briefly discussed two other quantitative analysis problems, including computing bounds on memory usage and on power or energy consumption.

Quantitative analysis remains an active field of research – exemplifying the challenges in bridging the cyber and physical aspects of embedded systems.

## **Tools for Execution-Time Analysis**

Current techniques for execution-time analysis are broadly classified into those primarily based on **static analysis** and those that are **measurement-based**.

Static tools rely on abstract interpretation and **dataflow analysis** to compute facts about the program at selected program locations. These facts are used to identify dependencies between code fragments, generate loop bounds, and identify facts about the platform state, such as the state of the cache. These facts are used to guide timing measurements of basic blocks and combined into an optimization problem as presented in this chapter. Static tools aim to find conservative bounds on extreme-case execution time; however, they are not easy to port to new platforms, often requiring several man-months of effort.

Measurement-based tools are primarily based on testing the program on multiple inputs and then estimating the quantity of interest (e.g., WCET) from those measurements. Static analysis is often employed in performing a guided exploration of the space of program paths and for test generation. Measurement-based tools are easy to port to new platforms and apply broadly to both extreme-case and average-case analysis; however, not all techniques provide guarantees for finding extreme-case execution times.

Further details about many of these tools are available in Wilhelm et al. (2008); Seshia and Rakhlin (2012). Here is a partial list of tools with links to papers and websites:

Name	Primary Type	Institution & Website/References
aiT	Static	AbsInt Angewandte Informatik GmbH (Wilhelm, 2005)
		http://www.absint.com/ait/
Bound-T	Static	Tidorum Ltd.
		http://www.bound-t.com/
Chronos	Static	National University of Singapore (Li et al., 2005)
		http://www.comp.nus.edu.sg/~rpembed/chronos/
Heptane	Static	IRISA Rennes
		http://www.irisa.fr/aces/work/heptane-demo/heptane.html
SWEET	Static	Mälardalen University
		http://www.mrtc.mdh.se/projects/wcet/
GameTime	Measurement	UC Berkeley
		Seshia and Rakhlin (2008)
RapiTime	Measurement	Rapita Systems Ltd.
		http://www.rapitasystems.com/
SymTA/P	Measurement	Technical University Braunschweig
		http://www.ida.ing.tu-bs.de/research/projects/symtap/
Vienna M./P.	Measurement	Technical University of Vienna
		http://www.wcet.at/

## **Exercises**

1. This problem studies execution time analysis. Consider the C program listed below:

```
int arr[100];
1
2
   int foo(int flag) {
3
      int i;
4
      int sum = 0;
5
6
      if (flag) {
7
        for(i=0;i<100;i++)</pre>
8
           arr[i] = i;
9
      }
10
11
12
      for(i=0;i<100;i++)</pre>
        sum += arr[i];
13
14
15
      return sum;
16
   }
```

Assume that this program is run on a processor with data cache of size big enough that the entire array arr can fit in the cache.

- (a) How many paths does the function foo of this program have? Describe what they are.
- (b) Let T denote the execution time of the second for loop in the program. How does executing the first for loop affect the value of T? Justify your answer.
- 2. Consider the program given below:

```
void testFn(int *x, int flag) {
1
    while (flag != 1) {
2
       flag = 1;
3
       *x = flag;
4
     }
5
    if (*x > 0)
6
      *x += 2;
7
 }
8
```

In answering the questions below, assume that x is not NULL.

(a) Draw the control-flow graph of this program. Identify the basic blocks with unique IDs starting with 1.

Note that we have added a dummy source node, numbered 0, to represent the entry to the function. For convenience, we have also introduced a dummy sink node, although this is not strictly required.

- (b) Is there a bound on the number of iterations of the while loop? Justify your answer.
- (c) How many total paths does this program have? How many of them are feasible, and why?
- (d) Write down the system of flow constraints, including any logical flow constraints, for the control-flow graph of this program.
- (e) Consider running this program uninterrupted on a platform with a data cache. Assume that the data pointed to by x is not present in the cache at the start of this function.

For each read/write access to \*x, argue whether it will be a cache hit or miss. Now, assume that \*x is present in the cache at the start of this function. Identify the basic blocks whose execution time will be impacted by this modified assumption.

3. Consider the function check\_password given below that takes two arguments: a user ID uid and candidate password pwd (both modeled as ints for simplicity). This function checks that password against a list of user IDs and passwords stored in an array, returning 1 if the password matches and 0 otherwise.

```
struct entry {
1
     int user;
2
     int pass;
3
  };
4
  typedef struct entry entry_t;
5
6
   entry_t all_pwds[1000];
7
8
9
   int check_password(int uid, int pwd) {
     int i = 0;
10
     int retval = 0;
11
12
     while(i < 1000) {
13
       if (all_pwds[i].user == uid && all_pwds[i].pass == pwd) {
14
          retval = 1;
15
         break;
16
       }
17
       i++;
18
     }
19
20
```

```
21 return retval;
22 }
```

(a) Draw the control-flow graph of the function check\_password. State the number of nodes (basic blocks) in the CFG. (Remember that each conditional statement is considered a single basic block by itself.)

Also state the number of paths from entry point to exit point (ignore path feasibility).

(b) Suppose the array all\_pwds is sorted based on passwords (either increasing or decreasing order). In this question, we explore if an external client that calls check\_password can *infer anything about the passwords* stored in all\_pwds by repeatedly calling it and *recording the execution time* of check\_password. Figuring out secret data from "physical" information, such as running time, is known as a *side-channel attack*.

In each of the following two cases, what, if anything, can the client infer about the passwords in all\_pwds?

- (i) The client has exactly one (uid, password) pair present in all\_pwds
- (ii) The client has NO (uid, password) pairs present in in all\_pwds

Assume that the client knows the program but not the contents of the array all\_pwds.

4. Consider the code below that implements the logic of a highly simplified vehicle automatic transmission system. The code aims to set the value of current\_gear based on a sensor input rpm. LO\_VAL and HI\_VAL are constants whose exact values are irrelevant to this problem (you can assume that LO\_VAL is strictly smaller than HI\_VAL).

```
volatile float rpm;
1
2
  int current_gear; // values range from 1 to 6
3
4
  void change_gear() {
5
     if (rpm < LO_VAL)</pre>
6
       set_gear(-1);
7
     else {
8
       if (rpm > HI_VAL)
9
          set_gear(1);
10
     }
11
12
13
     return;
```

```
}
14
15
   void set_gear(int update) {
16
      int new_gear = current_gear + update;
17
      if (new_gear > 6)
18
        new_gear = 6;
19
      if (new_gear < 1)</pre>
20
        new_gear = 1;
21
22
23
      current_gear = new_gear;
24
25
     return;
   }
26
```

This is a 6-speed automatic transmission system, and thus the value of current\_gear ranges between 1 and 6.

Answer the following questions based on the above code:

(a) Draw the control-flow graph (CFG) of the program starting from change\_gear, *without inlining* function set\_gear. In other words, you should draw the CFG using call and return edges.

For brevity, you need not write the code for the basic blocks inside the nodes of the CFG. Just indicate which statements go in which node by using the line numbers in the code listing above.

- (b) Count the number of execution paths from the entry point in set\_gear to its exit point (the return statement). Ignore feasibility issues for this question. Also count the number of paths from the entry point in change\_gear to its exit point (the return statement), including the paths through set\_gear. State the number of paths in each case.
- (c) Now consider path feasibility. Recalling that current\_gear ranges between 1 and 6, how many feasible paths does change\_gear have? Justify your answer.
- (d) Give an example of a feasible path and of an infeasible path through the function change\_gear. Describe each path as a sequence of line numbers, ignoring the line numbers corresponding to function definitions and return statements.