TEMPORAL LOGIC

Concurrent systems are notoriously hard to design and debug. Part of the problem is that concurrent systems exhibit a surprising variety of behaviors, and some bugs lead to failure only under pathological scenarios. The difficulty of catching such errors through conventional software engineering methods (such as testing) creates a need for more formal, systematic approaches to the design and analysis of such systems.

Temporal logic provides one such approach. The adjective *temporal* refers to the introduction of special logical modalities that allow the specification of *when* a property is expected to hold. For example, with temporal logic, one can state that if a process waits forever, it will eventually be serviced; this statement might be formalized as

$$(\Box\,\text{wait}) \Rightarrow (\Diamond\,\text{service})$$

where $\Box$ means *always* and $\Diamond$ means *eventually.* Although such analyses can be carried out within classical mathematics (by treating the system state as an explicit function of time), the encapsulation of time within temporal modalities makes the analyses easier to understand and more amenable to automation. Temporal logics are most often applied to systems that evolve through a sequence of discrete state transitions, but there are also logics designed for systems exhibiting both discrete and continuous behavior (discussed later).

An Example

As a running example, we consider Peterson's protocol for mutual exclusion (1). This protocol allows two processes (labeled P and Q) to share access to a resource, while making sure that the processes do not access the resource at the same time (this is the mutual exclusion property) and without requiring special hardware support (beyond atomic access to shared variables). While the Peterson protocol is less complex than most industrial examples (by orders of magnitude), it is still far from trivial.

For now we present the protocol with pseudocode; later, we model the protocol more precisely.

$P: try_p := true;$	$Q: try_q := true;$
$t := 1;$	$t := 0;$
$wait(\neg try_q \vee t = 0);$	$wait(\neg try_p \vee t = 1);$
access resource;	*access resource;*
$try_p := false;$	$try_q := false;$

The system starts with $try_p = try_q = false;$ the code shown for P is executed every time P wants to obtain access the resource (and similarly for Q).

There are several questions one might ask about this protocol:

- Does the protocol indeed prevent P and Q from accessing the resource simultaneously?
- If a P starts to execute the protocol, is it guaranteed to get access to the resource? If not, is it guaranteed that at least one of the processes will get access? If not, is it at least guaranteed that the system will not reach a deadlock state where neither component can do anything?
- On what process scheduling assumptions do these properties depend? How are these properties affected if the protocol is modified slightly (e.g., if a process is allowed to fail)?

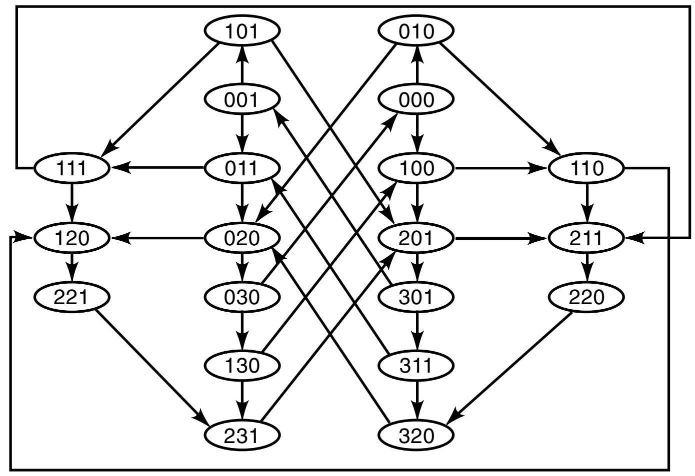

Figure 1. Peterson's protocol.

Preview

These questions are nontrivial, even for this (rather simple) protocol. One can ask similar questions about much more complex systems (e.g., microprocessors, distributed memory systems, and communication protocols). Temporal logic tools have been successfully applied to a number of such systems.

To analyze the Peterson protocol, we model it formally as a *transition system.* We then show how properties of its executions can be formulated in *linear-time logic* and proved using ordinary mathematics along with some special rules for handling temporal operators. Later we show how somewhat different properties can be formulated in *branching-time logic* and verified automatically using a special program called a *model checker.* We then survey some additional logics that can be used to reason about systems with timing constraints and systems that can undergo continuous state evolution.

TRANSITION SYSTEMS

To simplify the treatment of the Peterson protocol, we add to each process an explicit program counter, and eliminate the *try* variables; this transformation does not change the behavior of the protocol:

$P: p := 1;$	$Q: q := 1;$
$p, t := 2, 1;$	$q, t := 2, 0;$
$wait(q = 0 \vee t = 0);$	$wait(p = 0 \vee t = 1);$
$p := 3;$	$q := 3;$
access the resource;	*access the resource;*
$p := 0;$	$q := 0;$

Here, p, q, and t are counters (modulo 4, 4, and 2 respectively); initially, $p = q = 0$.

A *state* of the system is given by an assignment of values to its state variables (here p, q, and t). We notate states of the Peterson protocol by listing p, q, and t in order (e.g., the possible starting states of the protocol are 000 and 001). The behavior of a system can be specified by writing down a state graph showing how the state can change; the state graph of the Peterson protocol is shown in Fig. 1. (The figure does not include states, such as 321, which are not reachable from the starting states.) However, most systems of interest are too large to be specified in this way; it is usually more practical to describe this graph with formulas, as follows.

A *transition formula* is a Boolean formula built up from primed and unprimed state variables. If f is a transition formula and $s1$ and $s2$ are states, $s1 \xrightarrow{f} s2$ is the formula obtained from f by replacing unprimed variables with their values in $s1$, and replacing primed variables with their values in $s2$. For example, if f is the transition formula $p = 3 \wedge p' = 0$, then $300 \xrightarrow{f} 001$ simplifies to *true,* but $300 \xrightarrow{f} 301$ simplifies to *false.* Note that a transition formula does not restrict how unmentioned state variables can change at the same time.

A *state formula* is a transition formula without primed variables. If f is a state formula and s is a state, $f(s)$ abbreviates $s \xrightarrow{f}$ s, which is equivalent to f with variables replaced by their values in s. If f is a state formula, f' denotes the transition formula obtained from f by priming all state variables. As a convenience, we sometimes use states as state formulas (e.g., 210 is shorthand for the state formula $p = 2 \wedge q = 1 \wedge t = 0$).

A *transition system* T is given by a state formula $T.init$ (specifying the possible starting states of the system) along with a transition formula $T.\ trans$ (specifying how the state of the system can change from one moment to the next). We omit T when its value is clear from the context. A *path* e of T is an infinite sequence of states $(e_0, e_1, \ldots)$ in which consecutive pairs of states are related by the transition relation:

$$(\forall n : 0 \leq n \Rightarrow e_n \xrightarrow{T.trans} e_{n+1})$$

If, in addition, $init(e_0)$, then e is an *initial path* of T. The initial paths of a transition system capture its possible executions; for example, the initial paths of the Peterson protocol include the path

$$(000, 100, 110, 211, 220, 320, 020, 030, 130, \ldots)$$

A systematic way to translate an ordinary concurrent program into a transition system is to introduce explicit program counters (as above), to write a transition for each atomic step of each process, and to take the disjunction of all these transitions, along with a special transition *skip* in which all of the state variables remain fixed.

For example, the transitions of the process P of the Peterson protocol can be read as follows:

$$\begin{aligned} p = 0 \wedge p' = 1 \wedge t' = t \wedge q' = q \\ p = 1 \wedge p' = 2 \wedge t' = 1 \wedge q' = q \\ (q = 0 \vee t = 0) \wedge p = 2 \wedge p' = 3 \wedge t' = t \wedge q' = q \\ p = 3 \wedge p' = 0 \wedge t' = t \wedge q' = q \end{aligned}$$

The disjunction of these transitions can be written more compactly as the logically equivalent formula

$$\begin{aligned} &(p' = p + 1 \bmod 4) \\ &\wedge (q' = q) \\ &\wedge (p = 2 \Rightarrow (q = 0 \vee t = 0)) \\ &\wedge (p = 1 \Rightarrow t' = 1) \\ &\wedge (p \neq 1 \Rightarrow t' = t) \end{aligned}$$

Finally, a transition of the whole system is either a transition of P, a transition of Q, or a "stuttering" step where none of the variables changes:

$$\begin{aligned} init &\equiv p = q = 0 \\ trans &\equiv P \vee Q \vee skip \\ P &\equiv p' = (p + 1 \bmod 4) \wedge q' = q \\ &\quad \wedge (p = 2 \Rightarrow (q = 0 \vee t = 0)) \\ &\quad \wedge (p = 1 \Rightarrow t' = 1) \wedge (p \neq 1 \Rightarrow t' = t) \\ Q &\equiv q' = (q + 1 \bmod 4) \wedge p' = p \\ &\quad \wedge (q = 2 \Rightarrow (p = 0 \vee t = 1)) \wedge (q = 1 \Rightarrow t' = 0) \\ &\quad \wedge (q \neq 1 \Rightarrow t' = t) \\ skip &\equiv p = p' \wedge q = q' \wedge t = t' \end{aligned}$$

This system is analyzed in subsequent sections.

In passing, we note that there are alternative notations available for describing transition systems. The state-chart notation (2) provides a number of tools for making state graphs (Fig. 1) practical for somewhat larger systems. It is also possible to work directly with sequential programs (3) or communicating state machines (4).

LINEAR-TIME TEMPORAL LOGIC

In analyzing a transition system, we are primarily interested in proving that all of its paths satisfy some property. *Linear-time logics* provide languages for stating and proving properties of an individual path. There are many such logics; we illustrate some of their principles with a particularly simple logic, which we refer to as LTL. (This logic is closely related to the logic simple TLA of Ref. 5).

LTL formulas are defined as follows. Every transition formula is an LTL formula, and if f and g are LTL formulas, so are $\neg f$ ("not f"), $f \vee g$ ("f or g"), and $\Box f$ ("always f"). The semantics of LTL is given by the following rules, which define what it means for a formula f to hold for a path e (written $e \vDash f$) (the nth suffix of e, e^n, is defined as the path $e_i^n = e_{n+i}$):

$$\begin{aligned} e \vDash f &\triangleq e_0 \xrightarrow{f} e_1 \text{ for transition formula } f \\ e \vDash \neg f &\triangleq \neg(e \vDash f) \\ e \vDash f \vee g &\triangleq (e \vDash f) \vee (e \vDash g) \\ e \vDash \Box f &\triangleq (\forall n : 0 \leq n : e^n \vDash f) \end{aligned}$$

These definitions can be understood as follows. A transition formula holds for a path if and only if it relates the first two states of the path. (As a special case, a state formula holds for a path if and only if it holds for the first state of the path.) The negation of a formula holds for a path if and only if the formula does not hold for the path; the disjunction of formulas holds for a path if and only if either disjunct holds for the path. (The logical operators $\wedge$, $\Rightarrow$, and $\equiv$ can be defined from $\vee$ and $\neg$ in the usual way.) $\Box f$ holds for a path if f holds for every suffix of the path.

We define

$$\Diamond f \triangleq \neg \Box \neg f$$

$\Diamond f$ ("sometime f") holds for a path if and only if f holds for some suffix of the path. The operators $\Box$ and $\Diamond$ can be used to define a number of interesting properties:

- $\Box\Diamond f$ says that f holds infinitely often
- $\Diamond\Box f$ says that f holds almost everywhere
- $\Box(f \Rightarrow \Box f)$ says that f, once true, remains true
- $\Box(f \Rightarrow \Diamond g)$ says that every f state is followed by a g state

For any transition system T, the initial paths of T are precisely those paths satisfying the formula $T.\,init \wedge \Box T.\,trans$. This means that we can prove properties of a transition system by reasoning purely in terms of LTL formulas. It is possible to give a complete proof system for LTL, but we will instead concentrate on rules used for practical reasoning about transition systems.

Two classes of properties are of particular interest: *safety properties* ("nothing bad ever happens," e.g., the system never reaches a state where both processes are accessing the resource) and *progress properties* ("something good happens," e.g., a process trying to access the resource will eventually get in). These two classes are treated in the following sections.

Reasoning About Safety

Formulas of the form $\Box f$, where f is a transition formula, are typically proved with the following three rules:

- Propositional equivalences can be used to rewrite formulas to equivalent ones. For example, since the formulas X and $\neg\neg X$ are equivalent for any Boolean X, we can rewrite $\Box(p = 3)$ to $\Box\neg\neg(p = 3)$. We call this *the tautology rule.*
- For formulas f and g,

$$\Box(f \wedge g) \equiv \Box f \wedge \Box g$$

  (the *conjunction rule*). Note that the tautology and conjunction rules imply that $\Box$ is monotonic: if $f \Rightarrow g$ follows from ordinary propositional reasoning, then $\Box f \Rightarrow \Box g$.
- For state formula f,

$$\Box f \equiv f \wedge \Box(f \Rightarrow f')$$

  In terms of transition systems, if *T.init* satisfies f and *T.trans* preserves f, then f always holds throughout every initial path; such an f is called an *invariant* of the transition system. This rule says that an invariant is always true.

To prove $\Box f$, where f is a state formula, it is usually necessary to strengthen f to an invariant g. For example, the mutual exclusion condition $\neg(p = q = 3)$ holds in every reachable state of the Peterson protocol, but it is not an invariant, since *trans* does not preserve it (for example, $321 \overset{trans}{\Rightarrow} 331$). $\Box\neg(p = q = 0)$ can be proved as follows:

1. $(p = 3 \Rightarrow (q \leq 1 \vee t = 0))$ is an invariant of the Peterson protocol, so $\Box(p = 3 \Rightarrow (q \leq 1 \vee t = 0))$ by the invariance rule
2. Similarly, $\Box(q = 3 \Rightarrow (p \leq 1 \vee t = 1))$
3. $\Box((p = 3 \Rightarrow (q \leq 1 \vee t = 0)) \wedge (q = 3 \Rightarrow (p \leq 1 \vee t = 1)))$ from (1), (2), and the conjunction rule
4. $\Box\neg(p = q = 3)$ from (3) and the monotonicity of $\Box$

For interesting systems, the required invariants are often much more complicated than the properties being proved; this phenomenon is the primary source of complexity in most state-based program reasoning.

Reasoning About Progress

Recall that the Peterson protocol, as defined previously, has the stuttering step *skip* as one of its possible actions. Thus one possible behavior of the protocol is to remain in the same state forever; to prove that the system ever does anything, we need to add additional assumptions. We first consider how to specify these assumptions, and then show how to use them to prove more general types of progress properties.

Fairness. *Fairness conditions* provide a formal way to capture the assumption that certain things that can happen eventually do happen. For example, they can be used to guarantee that some process eventually takes a step, or that a process eventually stops accessing the resource. If f is a transition formula, f is *enabled* in those states where it is possible to execute the transition $f \wedge$ *trans;* formally,

$$enabled.f \equiv (\exists v' : f \wedge T.trans)$$

(where v' is the vector of all primed variables). For example, if *enterP* is the transition formula $p = 2 \wedge p' = 3$, *enabled.enterP* is the formula

$$(\exists p', q', t' : T \wedge p = 2 \wedge p' = 3)$$

If T is the Peterson protocol, this simplifies (using ordinary logical reasoning) to the state formula $p = 2 \wedge (q = 0 \vee t = 0)$.

There are several ways to specify that a transition is not unreasonably ignored.

Unconditional Fairness. The formula $\Box\Diamond f$ says that f is executed infinitely often. Note that this may have undesirable side effects; for example, unconditional fairness for *enterP* forces P to access the resource infinitely often.

Strong Fairness. The formula $\Box\Diamond enabled.f \Rightarrow \Box\Diamond f$ says that if f is enabled infinitely often, it must be executed infinitely often. For example, strong fairness for *enterP* says that if P infinitely often has the opportunity to access the resource, it will do so infinitely often.

Weak Fairness. The formula $\Diamond\Box enabled.f \Rightarrow \Box\Diamond f$ says that if f is almost always enabled, it must be happen infinitely often. For example, weak fairness for *enterP* says that if P is permanently able to enter, it will eventually do so. Note that weak fairness of f is equivalent to unconditional fairness for $enabled.f \Rightarrow f$.

In LTL, fairness conditions can be added directly as additional hypotheses to the formula being checked. For example, when we say that a formula f holds assuming unconditional fairness for g, we mean that $(\Box\Diamond g) \Rightarrow f$ holds.

Exploiting Fairness Hypotheses. The usual way to make use of weak or unconditional fairness is to use the following rule, which generates a progress property from an unconditional fairness property:

$$\Box(f \Rightarrow f' \vee g \vee g') \wedge \Box\Diamond h \Rightarrow \Box(f \Rightarrow \Diamond((f \wedge h) \vee g))$$

The first hypothesis says that whenever f holds, it remains true up until the first point that g holds. For example, in the Peterson protocol, if f is the formula $p = 2 \wedge t = 0$, then, assuming weak fairness of *enterP*,

$\Box(f \Rightarrow f' \vee p' = 3)$	from $\Box$*trans* and logical reasoning
$\Box\Diamond(f \Rightarrow p = 3)$	from weak fairness of *enterP*
$\Box(f \Rightarrow \Diamond(p' = 3))$	from the last two conclusions and the progress rule

Using similar reasoning, we can obtain the following progress properties for the Peterson protocol from the corresponding weak fairness properties:

1. $\Box(p = 1 \Rightarrow \Diamond(p = 2))$	from fairness of $p = 1 \wedge p' = 2$
2. $\Box(p = 2 \wedge t = 0 \Rightarrow \Diamond(p = 3))$	from fairness of $p = 2 \wedge p' = 3$
3. $\Box(211 \Rightarrow \Diamond(p = 2 \wedge t = 0))$	from fairness of $q = 1 \wedge q' = 2$
4. $\Box(201 \Rightarrow \Diamond(p = 3 \vee 211))$	from fairness of $p = 2 \wedge p' = 3$
5. $\Box(221 \Rightarrow \Diamond 231)$	from fairness of $q = 2 \wedge q' = 3$
6. $\Box(231 \Rightarrow \Diamond 201)$	from fairness of $q = 3 \wedge q' = 0$

These basic progress properties are then combined with the following rules, that say that progress is idempotent, transitive, and disjunctive:

$$\Box(p \Rightarrow \Diamond p)$$
$$\Box(p \Rightarrow \Diamond q) \wedge \Box(q \Rightarrow \Diamond r) \Rightarrow \Box(p \Rightarrow \Diamond r)$$
$$\Box(p \Rightarrow \Diamond r) \wedge \Box(q \Rightarrow \Diamond s) \Rightarrow \Box(p \vee q \Rightarrow \Diamond(r \vee s))$$

For example, from the progress properties proved above, we can prove

$$\Box(p \geq 1 \Rightarrow \Diamond p = 3)$$

which says that if P is trying to access the resource, it will eventually obtain access:

7. $\Box(211 \Rightarrow \Diamond p = 3)$	by (3), (2), and transitivity
8. $\Box(201 \Rightarrow \Diamond p = 3)$	by (4), (7), disjunctivity and transitivity
9. $\Box(231 \Rightarrow \Diamond p = 3)$	by (6), (8), and transitivity
10. $\Box(221 \Rightarrow \Diamond p = 3)$	by (5), (9), and transitivity
11. $\Box(p = 2 \wedge t = 1 \Rightarrow \Diamond p = 3)$	by (7), (8), (9), (10), and disjunction
12. $\Box(p = 2 \Rightarrow \Diamond p = 3)$	by (11), (2), and disjunctivity
13. $\Box(p = 1 \Rightarrow \Diamond p = 3)$	by (1), (12), and transitivity
14. $\Box(p > 0 \Rightarrow \Diamond p = 3)$	by (12), (13), idempotence and disjunctivity

Strong fairness properties are exploited in the same way; the only difference from unconditional fairness is the additional disjunct $\Diamond\Box\neg$*enabled.f*, which is just treated as a separate case.

Decision Procedures for LTL

The problem of checking if an LTL formula is true is PSPACE-complete, which means that in practice the time to perform the check is exponential in the length of the formula. One way to perform this check is to treat the formula as an omega-regular language (encoding individual states as ordered lists of variable-value pairs), which reduces the validity problem to the well-known problem of checking equivalence of omega-regular languages (6).

BRANCHING-TIME LOGICS

In linear-time logics, formulas specify properties that hold for all paths. Branching-time logics provide additional flexibility by allowing one to specify that a property must hold for some path. Although sound engineering demands that a system should work for every possible execution, there are several reasons that branching-time logics are useful:

- Branching-time formulas can guarantee that the system does not unrealistically constrain the environment in which is embedded. For example, for the Peterson protocol, one might specify that in every state, it is possible that $q > 0$ in the following state; this effectively says that the process Q is free to enter the protocol at any time.
- Branching-time formulas can specify that the system cannot reach a state in which the operations are forever stuck waiting for each other to release resources (for example, by requiring that it is always possible to reach a state where $p = q = 0$).
- Possibility can sometimes be used as a substitute for guaranteed progress under fairness hypotheses, which can make model checking much easier to carry out.

Our example of a branching-time logic is computation tree logic (CTL) (7), which is the logic used by most current model checkers. As opposed to linear-time logics, which specify properties of arbitrary paths, CTL formulas are always interpreted in the context of a transition system, and formulas hold or fail to hold for a particular state, rather than for a path.

CTL formulas are defined as follows. A *path quantifier* is either **A** (*necessarily*) or **E** (*possibly*). Every state formula is a CTL formula; if **Q** is a path quantifier and f and g are CTL formulas, then $\neg f$, $f \vee g$, $\mathbf{QX}f$, and $\mathbf{Q}f\mathbf{U}g$ are CTL formulas. Formulas are interpreted as follows:

$$s \vDash f \triangleq f(s) \text{ for state formula } f$$
$$s \vDash f \vee g \triangleq (s \vDash f) \vee (s \vDash g)$$
$$s \vDash \neg f \triangleq \neg(s \vDash f)$$
$$e \vDash \mathbf{X}f \triangleq e_1 \vDash f$$
$$e \vDash f\mathbf{U}g \triangleq (\exists n : e_n \vDash g \wedge (\forall m : m < n \Rightarrow e_m \vDash f))$$
$$s \vDash \mathbf{A}f \triangleq (\forall e : e \text{ a path of } T \wedge e_0 = s \Rightarrow e \vDash f)$$
$$s \vDash \mathbf{E}f \triangleq (\exists e : e \text{ a path of } T \wedge e_0 = s \wedge e \vDash f)$$

These definitions can be read as follows. A state formula holds in a state if it holds in the sense the section entitled "Transition Systems." The disjunction of two formulas holds in a state if and only if either disjunct holds; the negation of a formula holds if and only if the formula fails to hold. **X***f* ("next time *f*") holds for a path if and only if *f* holds for the second state of the path; *f***U***g* ("*f* until *g*") holds for a path if and only if *g* holds for some state of the path and *f* holds for every state up to the first state in which *g* holds. **A***f* ("necessarily *f*") holds in a state if and only if *f* holds for every path starting at that state; dually, **E***f* ("possibly *f*") holds for a state if and only if *f* holds for some path starting at that state.

The $\Box$ and $\Diamond$ operators can be defined from **U**, that is, $\Diamond f \equiv (true\mathbf{U}f)$, and $\Box f \equiv \neg\Diamond\neg f$. The path quantifiers **A** and **E** allow one to speak about the possible futures of the system. For example,

- $\mathbf{A}\Box f$ says that f always holds
- $\mathbf{E}\Box f$ says that it is possible for f to always hold
- $\mathbf{A}\Diamond f$ says that f is guaranteed to hold eventually
- $\mathbf{E}\Diamond f$ says that it is possible for f to eventually hold
- $\mathbf{A}\Box(f \Rightarrow \mathbf{E}\Diamond g)$ says that, from every f state it is possible to reach a g state

In the case of Peterson's protocol, one might wish to prove properties like

$$\mathbf{A}\Box(q = 0 \Rightarrow \mathbf{E}(q = 0\mathbf{U}p = 3))$$

which says that at any time at which Q is not trying to access the resource, it is possible for P to gain access without Q ever entering the protocol (i.e., P can gain access without any cooperation from Q).

Some LTL formulas can be translated to CTL equivalents by prefixing every temporal operator with **A** (e.g., the LTL formula $\Box\Diamond p = 0$ translates to the CTL formula $\mathbf{A}\Box\mathbf{A}\Diamond p = 0$). However, even without considering primed variables, there are LTL formulas that cannot be written as CTL formulas. For example, the formula $\Diamond(p = 1) \vee \Box(p = 0)$ has no CTL equivalent; it is not equivalent to $\mathbf{A}\Diamond(p = 1) \vee \mathbf{A}\Box(p = 0)$ (the first holds in the Peterson protocol, while the second does not), and $\mathbf{A}(\Diamond(p = 1) \vee \Box(p = 0))$ is not a CTL formula. There are branching-time logics that generalize both LTL and CTL, such as CTL* (8), but model checking procedures for such languages are at least exponential in the size of the formula being checked. The preferred solution is to work fairness assumptions into the system model and model checking procedures.

CTL Model Checking

We now describe a simple way to check CTL formulas in a transition system, by showing how to reduce each CTL formula f to an equivalent state formula, based on the transition relation *trans*. f then holds if and only if $init \Rightarrow f$ [because this is a state formula, it can be checked using ordinary (nontemporal) logic]. To reduce $\mathbf{QX}f$ or $\mathbf{Q}f\mathbf{U}g$ to a state formula, we first reduce f and g to state formulas. The next step depends on the formula being reduced:

- The state formula for $\mathbf{EX}f$ is $(\exists\ v'\colon (trans \wedge f'))$, where v' is the vector of all primed state variables. For example, in the Peterson protocol, if f is the formula $p = 3$, then $\mathbf{EX}f$ is

  $$(\exists\, p', q', t' : (trans \wedge p' = 3))$$

  which simplifies to the state formula

  $$p = 3 \vee (p = 2 \wedge (t = 0 \vee q = 0))$$

  Similarly, the state formula for $\mathbf{AX}f$ is $(\forall\ v'\colon (T.trans \wedge f'))$.
- The state formula of $\mathbf{E}f\mathbf{U}g$ is the strongest formula x satisfying the equation $x = ((f \wedge \mathbf{EX}x) \vee g)$; that is, $\mathbf{E}f\mathbf{U}g$ is the set of all states that can reach a g state via a sequence of f transitions. x can be calculated by starting with $x = g$, and repeatedly performing the assignment $x := x \vee (f \wedge \mathbf{EX}p)$ until a fixed point is reached (i.e., until the assignment yields a logically equivalent state formula).

  The state formula of $\mathbf{A}f\mathbf{U}g$ is calculated using the same procedure, but with all **A**'s above changed to **E**'s.

For example, to check that $\mathbf{E}((q = 0)\mathbf{U}(p = 3))$, we first calculate the state formula $\mathbf{E}((q = 0)\mathbf{U}(p = 3))$ as above; the successive values of x are

$$\begin{aligned}
&p = 3\\
&p = 3 \vee (q = 0 \wedge p = 2)\\
&p = 3 \vee (q = 0 \wedge (p = 1 \vee p = 2))\\
&p = 3 \vee q = 0
\end{aligned}$$

which is a fixed point. Since $init \Rightarrow (p = 3 \vee q = 0)$, $\mathbf{E}((q = 0)\mathbf{U}(p = 3))$ checks successfully.

Each of these operations can be performed in time linear in the number of states in the transition system, so the time to check a system for any CTL property grows at worst as the product of the size of the transition system and the size of the formula being checked. In contrast, the model-checking problem for LTL is PSPACE-complete, which means in practice that it is exponential in the size of the formula. Thus, CTL model checking is much more efficient.

One drawback of CTL is that, unlike LTL, fairness properties cannot be expressed within the logic. However, the model-checking procedure can be extended to work with fairness conditions; the penalty is an extra factor that is polynomial in the number of fairness assumptions (9). Most modern model checkers allow fairness assumptions as part of the system model.

The above computations can be performed symbolically (as indicated above); a model checker that works in this way is called a *symbolic model checker* (10). The main technology that makes this practical is the use of *ordered binary decision diagrams* (11) to represent state formulas in a way that makes their logical manipulation (in particular, testing whether two formulas are equivalent) very efficient (at least for large classes of formulas). An important research problem is the investigation of alternative ways to represent formulas that allow this efficient manipulation.

Explicit State Search. Most model checkers do not work explicitly with formulas, but instead work one state at a time. For example, to test $\mathbf{A}\Box f$, the checker can just enumerate the state graph of Fig. 1, checking that each state satisfies f. This approach, called *explicit state search,* is sometimes more efficient, particularly for system with many state variables but relatively few reachable states.

In practice, the coverage of explicit state search is limited by the space needed to keep track of the set of states that have been explored. Some special techniques have been developed to overcome this problem. One way to do this efficiently is to represent a set of states using a hash table of bits; a state is in the set if the table indices it hashes to (using several independent hash functions) all have their bits set. This representation is not perfect, because multiple states might hash to the same index, but it allows large sets to be represented very efficiently (using a few bits per state) (12).

A fair body of research has been devoted to techniques that avoid exploring redundant paths to the same state ("partial order techniques") and, more generally, to states that are equivalent under some symmetry relation (for systems composed of a number of identical processes).

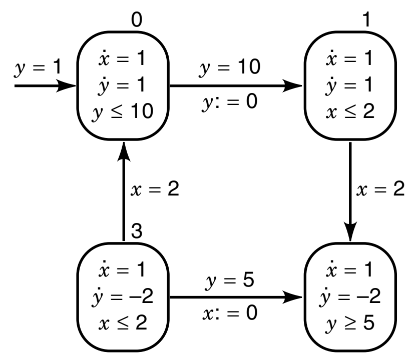

Figure 2. Water-level monitor.

REASONING ABOUT REAL-TIME SYSTEMS

Some systems depend on timing constraints for correctness. For example, many message transmission protocols depend on a sender's timeout being long enough to guarantee that a message was lost if no reply is received during the timeout interval. Systems that depend on such explicit timing assumptions are generically called *real-time* systems.

One way to reason about real-time systems is to represent the time with an explicit state variable t, along with axioms that say that time never moves backward and that that the time is guaranteed to move beyond any fixed boundary. Rules of inference are used to derive real-time formulas from other real-time formulas. See, for example, (13).

For automatic verification, it is necessary to introduce timing into the automata model. A *timed automaton* is like a regular finite-state transition system, except that it also has timers that can be set, and perform a specific action when they expire. Like ordinary transition systems, CTL properties of timed automata can be checked automatically (14).

A *hybrid system* is a system that can undergo both discrete transitions and continuous evolution (e.g., where the changes of real-valued variables are governed by differential equations). Hybrid systems typically arise in control applications; for example, in air-traffic control, the movement of planes is continuous, while the protocols governing communication are discrete. We conclude with a brief description of two formalisms for reasoning about hybrid systems.

Hybrid Automata

A hybrid system's behavior over time can be modeled as a sequence of phases; during each phase the system state is governed by a set of differential equations. At a phase boundary a discrete event occurs and the system state is governed by a new set of differential equations. In *hybrid automata* (15), phases are modeled by *modes* and discrete events by *control switches*. Associated with the hybrid automaton is a set of real-valued variables $\boldsymbol{x}$ denoting the "physical state" of the system. Associated with each mode is a set of differential inequalities governing $\boldsymbol{x}$ as well as an *invariant* condition upon $\boldsymbol{x}$. Whenever this invariant is falsified, the hybrid automaton jumps out of its current mode into an adjacent mode.

An invariant is *linear* if it is a disjunction of inequalities of the form $A \cdot \boldsymbol{x} \sim \boldsymbol{c}$ where A is a constant matrix, $\boldsymbol{c}$ is a constant vector, and $\sim$ is either $\leq$ or $\geq$. A hybrid automaton is linear if its invariants are linear and its differential inequalities are of the form $A \cdot \dot{\boldsymbol{x}} \sim \boldsymbol{c}$ where $\dot{\boldsymbol{x}}$ is the vector of first derivatives of the variables $\boldsymbol{x}$.

Figure 2 shows a linear hybrid model of a control system for a water tank. The variable y represents the level of the water in the tank; the system is designed to keep this level between 1 and 12 in. It does this by turning a pump on or off; when the pump is on (states 0 and 1), the water level rises at 1 in./sec., and when the pump is off (states 2 and 3), the water level falls at 2 in./sec. The diagram can be read as follows. The system starts with the water level at 1 in. and the pump on (state 0). When the water level reaches 10 in., the timer x is reset (state 1). When the timer hits 2 sec., the pump turns off (state 2). When the water level falls to 5 in., the timer is reset (state 3); 2 sec. later, the pump turns on again (state 0).

Certain temporal properties of linear hybrid automata can be checked automatically using methods from the logical theory of linear arithmetic (15).

Duration Calculus

The duration calculus (16) is a calculus for specifying and reasoning about the duration over which formulas hold. If S is a Boolean function of time, $\int S$ denotes a function called the duration of S. The value of $\int S$ for a real-valued interval of time denotes the total duration for which S holds in that interval. Atomic formulas are predicates upon durations. Formulas are constructed from other formulas using logical connectives, and a special connective called the ; operator ("chop"). The formula $A;B$ is true for an interval $[b, e]$ when this interval can be divided into an initial subinterval $[b, m]$ for which A is true and a final subinterval $[m, e]$ for which B holds. The requirement that a formula S holds for every subinterval of an interval can be expressed as $\neg(true;\neg S;true)$, abbreviated $\Box S$. This states that it is not possible to find a subinterval in the interval for which S is false.

New duration calculus formulas can be derived using inference rules; an example of such a rule is

$$(\textstyle\int S = x); (\textstyle\int S = y) \Rightarrow (\textstyle\int S = x + y)$$

which says that the duration of S for a sequential combination of intervals is equal to the sum of its durations for the two intervals.

For example, in the Peterson protocol, the requirement that P not wait for the resource for more than 4 time units could be written as

$$\int (p = 1 \vee p = 2) \leq 4$$

The requirement S that in a given interval of length at most 30, P not wait for more than 4 time units is expressed as

$$\int true \leq 30 \Rightarrow \int (p = 1 \vee p = 2) \leq 4$$

Note that this requirement would be trivially true for an interval larger than 30 s even when the interval contains a subinterval less than or equal to 30 s over which P waits for more than 4 s. The requirement that P waits for no more than 4 s out of any 30 s interval can be written as $\Box S$.

TOOLS

A good survey of applications of formal methods can be found in Ref. 17. Up-to-date listings of tools can be obtained from formal methods pages on the World-Wide Web. Some of the more popular notations and tools include the following:

COSPAN. The COSPAN system (6), sold commercially as FormalCheck, is an automata-based tool designed for verifying digital hardware systems designs written in the VHDL and Verilog.

HyTech. HyTech (15) is an implementation of linear hybrid automata.

SPIN. The SPIN system (12) is an explicit state-space search system for verifying systems written as synchronously communicating finite-state processes.

SMV. SMV (18) is a symbolic model checker for CTL (with fairness conditions). It has been used to find subtle published errors in a standardized cache coherence protocol.

TLA. TLA (5) (the Temporal Logic of Actions) is a linear-time logic similar to LTL, except that all formulas are stuttering-invariant, and temporal quantification is allowed. It has been applied to several large-scale specification problems.

UNITY. UNITY (19) is a linear-time programming notation and proof system. It has been applied to a large number of interesting concurrent and distributed programming problems.

FURTHER READING

The survey article, Ref. 20, provides an excellent overview of the theory of temporal logic. Ref. 21 provides a very thorough treatment of temporal logic as a specification tool, particularly for structured programs. Ref. 6, surveys automata-theoretic techniques for temporal verification. Reference 22 surveys logics for real-time systems; the book, Ref. 23, surveys logics for real-time and hybrid systems.

For other approaches to the analysis of concurrent systems, see the articles on PROCESS ALGEBRA and PROGRAMMING THEORY.

BIBLIOGRAPHY

1. G. L. Peterson, Myths about the mutual exclusion problem, *Inf. Process. Lett.,* **12** (3): 1981.
2. D. Harel, Statecharts: A visual formalism for complex systems, *Sci. Comput. Prog.,* **8**: 231–274, 1987.
3. Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems, Vol. 1: Specification,* New York: Springer, 1992.
4. N. Lynch and M. Tuttle, Hierarchical correctness proofs for distributed algorithms, *ACM Symp. Prin. Distrib. Comput.,* 1987, pp. 137–151.
5. L. Lamport, The temporal logic of actions, ACM TOPLAS, **16** (3): 872–923, May 1994.
6. R. Kurshan, *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach,* Princeton, NJ: Princeton University Press, 1994.
7. E. A. Emerson, E. M. Clarke, and A. P. Sistla, Automatic verification of finite state concurrent systems using temporal logic, *ACM TOPLAS,* **8** (2): 244–263, 1986.
8. E. Emerson and J. Halpern, "Sometimes" and "not never" revisited: On branching versus linear time, *J. ACM,* **33** (1): 1986.
9. E. A. Emerson and C. L. Lei, Modalities for model checking: Branching-time logic strikes back, *Sci. Comput. Prog.,* **8**: 275–307, 1987.
10. K. McMillan, *Symbolic model checking,* Ph.D. thesis, Carnegie-Mellon University, 1993.
11. R. E. Bryant, Symbolic boolean manipulation with ordered binary-decision diagrams, *ACM Comput. Surv.,* **24** (3): 293–318, 1992.
12. G. Holzmann, *Design and Validation of Computer Protocols,* Englewood Cliffs, NJ: Prentice Hall, 1991.
13. A. Pnueli, T. Henzinger, and Z. Manna, Temporal proof methodologies for real-time systems, *Proc. 18th ACM Symp. Prin. Prog. Lang.,* 1991, pp. 353–366.
14. R. Alur and D. Dill, A theory of timed automata, *Theor. Comput. Sci.,* **126**: 183–235, 1994.
15. P.-H. Ho, T. Henzinger, and H. Wong-Toi, Hytech: a model checker for hybrid systems, *Software Tools Technol. Transfer,* **1** (1), 1997.
16. A. Ravn, Z. Chaochen, and C. A. R. Hoare, A calculus of durations, *Inf. Process. Lett.,* **40** (5): 269–276, 1991.
17. E. Clarke et al., Formal methods: state of the art and future directions, *ACM Comput. Surv.,* **28** (4), 1996.
18. K. McMillan, *Symbolic Model Checking: An Approach to the State-Explosion Problem,* City: Kluwer, 1993.
19. K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation,* Reading, MA: Addison-Wesley, 1988.
20. A. E. Emerson, *Temporal Logic,* Amsterdam: North-Holland, 1989, pp. 997–1073.
21. A. Pnueli and Z. Manna, *The Temporal Verification of Reactive Systems, Vol. 1: Specification,* New York: Springer, 1992.
22. J. S. Ostroff, Formal methods for specification and design of safety-critical systems, *J. Syst. Software,* **33** (60), 1992.
23. C. Heitmeyer and D. Mandrioli (eds.), *Formal Methods for Real-Time Computing,* New York: Wiley, 1996.

ERNIE COHEN
SANJAI NARAIN
Bellcore

TERMINALS, TELECOMMUNICATION. See TELECOMMUNICATION TERMINALS.

TERMINATED CIRCULATORS. See MICROWAVE ISOLATORS.