Chapter 6

Temporal Logic

6.1 Where it fits

Temporal logics have been used to specify properties of concurrent systems since they were first proposed for this purpose around 1976, by Amir Pnueli. These logics differ from all of the specification methods we have seen so far in that they consider an entire execution at once, as a distinct object. The other methods consider individual steps that can be taken under various conditions, and infer what the executions can be from descriptions of the allowed and required steps. Here, the situation is reversed. The entire execution is the object under consideration, and what individual steps can or must be taken is derived from the temporal logic specification.

Temporal logic is a particular variant of *modal* logic, where new logical operators are introduced to quantify over a family of objects in the system. In the case of temporal logic in the simplest form, the object in question is a sequence. We interpret this sequence as possible states of the system being specified, as changed over time by atomic actions. This is the only thing "temporal" about the logic. Time does not appear explicitly in it, and in fact the operators could be applied to any sequence.

Temporal logics have been used to specify entire systems, to augment other specification methods that can only express safety properties, and as input to (semi-) automatic verification systems for finite

state programs. This application will be considered in more detail in Section 6.8.

Several variants of temporal logic have been developed. The basic operators of linear temporal logic will be presented first, and then some possible extensions will be introduced as the difficulties in expressive power are analyzed.

6.2 Linear temporal logic

In this form of temporal logic, an individual execution sequence is considered. Each element of this sequence is a global state in the system being specified, and the action that transformed the system from one state to the next is inferred implicitly. A logic over such sequences allows expressing assertions about the sequences of states. As in any logic, one way to interpret a temporal logic assertion is as defining the *characteristic set* of all the sequences for which the assertion is *true*.

Since the temporal operators we will see can be nested in a temporal logic assertion, they must be defined for every state, and not just at the beginning of the sequence. Thus we need to define, for each legal assertion P, precisely when it must be true. In other words, the formal definition requires that each operator be defined for each position in the (finite or infinite) sequence.

Given a sequence of states, σ, and an index i denoting some position in the sequence, and an assertion in the logic P this can be written as

$$(\sigma, i) \models P$$

One simple possibility for the logic is to assert something about a particular state with no reference to states before or after it in the sequence. If we are considering a specific state, such an assertion P is interpreted as claiming that the values in the state satisfy the assertion. Thus for a P with no temporal operators,

$$(\sigma, i) \models P \quad \textit{iff} \quad P(\sigma(i)).$$

The earlier methods, such as first-order logic, or predicates in a Z schema, can be used for such assertions. We shall assume that whatever

language is used for asserting about an individual state, it will be sufficiently expressive, and allow defining arbitrary predicates and functions of the state. Among the predicates useful for distributed specifications are those that relate to the control locations of the processes, and to the transition just executed, from the previous state of the sequence. Thus a predicate $at(L)$ is true when control is just before a command labelled by L (or a section of code named L), $in(L)$ means that some of the instructions associated with L can now execute, $terminated(P)$ means that a process P has completed executing (has no more enabled instructions, and will not execute anything more in the remainder of the execution).

Now the operators that deal with assertions about true sequences can be defined. There are two fundamental operators that appear in every version of temporal logic: $\Box$ and $\Diamond$.

An expression $\Box P$ is read *box P* or *from now on, P*. Given a sequence σ, and a position in the sequence i, then

$$(\sigma, i) \models \Box P \quad \textit{iff} \quad \forall j.i \leq j \leq |\sigma|.\ (\sigma, j) \models P$$

This means that for every state in σ after position i, the predicate P is true (if σ is infinite, the upper bound on i is ω). If the initial position of the sequence is considered (so $i = 0$), then $\Box P$ means that P is true throughout σ, and thus is an invariant, at least for the execution σ.

The assertion $\Box(x \geq 0)$ in state i means that $x \geq 0$ holds in every state from i onwards. As we will see later, when only the $\Box$ operator is used, along with simple predicates about states (and no negation), safety properties will be specified. Intuitively, we are asserting about all acceptable steps that can be taken.

The second basic temporal operator is written $\Diamond P$, read *diamond P*, or *eventually P*. With assumptions as above, this is formally defined by

$$(\sigma, i) \models \Diamond P \quad \textit{iff} \quad \exists j.i \leq j \leq |\sigma|.(\sigma, j) \models P.$$

So this is the existential operator for sequences, saying that there exists a state in which P holds, in some position from position i. This allows specifying liveness properties. For example,

$$\Diamond(x \geq 0)$$

asserts that in some state ($x \geq 0$) is true. Thus either it is true initially, or some step will occur that will cause this to be true.

Negation and disjunction can be added in a way that corresponds to our intuition. $\neg P$ holds in a state, if P does not hold in the state, and the assertion $P \vee Q$ is true if either P or Q is true of the state being considered. Then $\Box$ and $\Diamond$ are duals, with each of the following equalities:

$$\begin{aligned} \Diamond P &\equiv \neg\Box\neg P \\ \Box P &\equiv \neg\Diamond\neg P \\ \neg\Diamond P &\equiv \Box\neg P \end{aligned}$$

Happily, the equalities above can be shown by logical manipulation of the precise definitions, and they also conform to our intuitive understanding of 'eventually' and 'from-now-on'. For example, the second equality could be expressed in words as: " 'P is always true' is the same as saying that 'it is not true that, eventually, P is not true.' "

Above we have seen the simplest so-called *future fragment* of linear temporal logic, as used for the specification of reactive systems. (Historically, sometimes the letter G is used instead of $\Box$, and F instead of $\Diamond$.) The acceptable behaviors of a system are described as temporal logic assertions about the possible execution sequences of the system. The claim of linear temporal logic is that we can always make assertions about any "typical" execution sequence, and that we will understand this as applying to any execution that can possibly occur in the system. That is, the characteristic set of all sequences satisfying the assertion includes those sequences that are possible executions of the system being considered. For this temporal logic, it is a clear assumption that we are relating to all sequences of the system, by describing properties they all have to satisfy.

Although we defined the assertions in terms of suffixes of sequences, an assertion is associated with an entire execution sequence of states, rather than with a specific position within a sequence. There are actually two reasonable ways to interpret the intended meaning of a temporal logic assertion for an execution sequence. One is that the predicate should be true for the first state in the sequence (i.e., for $(\sigma, 0)$), and the other is that it should hold for all states in the sequence (i.e., for

every (σ, i)). The former is called the *anchored* version, while the latter is known as the *unanchored*. For the purposes of specification and naturalness of expression, the anchored version seems to have a slight advantage. The disadvantage is that the proof theory, e.g., axioms and deduction rules to show whether a formula is true, is somewhat more complex. Since we are mainly interested in specification, we will use the anchored view, and interpret a top-level assertion as relating to the first element in the sequence. Of course, for any assertion p, by simply writing $\Box p$, we specify that p must be true in every state of the sequence.

The $\Box$ and $\Diamond$ operators are surprisingly expressive for many common properties that arise in specifications. Termination can be expressed as a simple eventuality, $\Diamond at(Halt)$. This assertion uses a state predicate at expressing that control has reached some given statement (in this case, the Halt statement). A partial correctness assertion $\{P\}\ S\ \{Q\}$ as defined in the input/output assertions of Chapter 2, can be expressed in linear temporal logic as

$$P \ \rightarrow \ \Box(at(Halt) \rightarrow Q)$$

Since we assume the anchored interpretation, this means that if P holds in the initial state, then in every subsequent state, if the *Halt* location is reached in that state, then Q also holds in that state. Note that this means only that the implication is always true, but says nothing about whether $at(Halt)$ is ever true.

However, the real strength of temporal logic is for infinite execution sequences. In such systems temporal logic allows expressing global properties of the execution, independently of any specific locations. For example,

$$\Diamond\Box P$$

means that eventually $\Box P$ is true for some state, i.e., P becomes always true from that point onwards. This can be appropriate for specifying some stable state described by P. The specification means that P may not hold initially, and that it may 'flicker' on and off for some prefix of the execution sequence, but eventually it will be true and remain true in all subsequent states of the execution.

Another example that looks similar but differs significantly in meaning is

$$\Box\Diamond P$$

Again, a moment's consideration, and translating this to the English phrases given earlier, shows that it means "in every state from now on, P will eventually be true again." If we consider any state from the point where the assertion is made, it must satisfy $\Diamond P$, that is, be followed by a state satisfying P. But once that state is reached, it too must eventually be followed by another state satisfying P, and so forth. Thus this assertion can be true for an infinite execution sequence only if P is true in infinitely many states (Otherwise, the assertion $\Diamond P$ would not hold for any state after the "last" state satisfying P). Such a specification can be appropriate for situations where some resource is repeatedly available (or repeatedly granted), without having to note in advance how many steps must occur before the availability.

In a similar vein, the assertion

$$\Box(P \rightarrow \Diamond Q)$$

means that from now on, in every state, if P is true, then eventually Q is true. If P represents "a request has been made" then Q might represent "the request is answered", and such a specification corresponds to the assertion that every request is eventually answered.

Thus many natural properties of reactive systems can be specified using what we have seen so far. Unfortunately, there are properties that cannot be expressed. One common type of assertion is that P will be true until Q occurs. It can be proven that $\Box$ and $\Diamond$ are insufficient for expressing this assertion. A *binary* operator is needed. This operator can be written as $P\mathcal{U}Q$ and read P *until* Q. The formal definition is

$$(\sigma, i) \models P\mathcal{U}Q \text{ iff } \exists k.i \leq k \leq |\sigma|.((\sigma, k) \models Q \wedge \forall j.i \leq j < k.(\sigma, j) \models P)$$

This is sometimes called "strong until" because it requires that a state with Q does occur after the i-th state of the sequence. A weaker version, in which Q is not required to ever be true, will be written as $P\mathcal{U}_w Q$ and is sometimes read as P *unless* Q. If Q does not occur, P must be true in every state beginning from i. Thus

$$P\mathcal{U}_w Q \text{ iff } \Box P \vee P\mathcal{U}Q.$$

One type of specification using these operators might be

$$\Box(request\_made \rightarrow (request\_registered \ \mathcal{U} \ request\_answered))$$

That is, once a request is made, it will remain registered at least until the request is answered. Another states that once a process requests a resource it will be inactive until it receives a *goahead* message. Assuming predicates $sent(L)$ to indicate that a message named L has been sent, and an analogous $received(L)$, a specification might include

$$\Box(sent(request) \rightarrow (inactive(P) \ \mathcal{U} \ received(goahead))$$

Note that temporal logic complements a state machine specification that can easily express which transitions are possible, but cannot easily relate states that are separated by a subsequence of transitions. Thus one common use of temporal logic is to augment a state machine description. That is, additional properties not necessarily true of the state machine description of a data structure and its operations are added by temporal assertions. For example, we might decide to assert that every element in a set will eventually be removed and no longer be a member of the set. This is clearly not always true of a state machine description of a set, and is easily expressed as

$$\Box((x \in s) \rightarrow \Diamond(x \notin s))$$

Temporal logic assertions also are used to describe requirements that are already true of a state machine, but cannot be expressed in those notations. This is especially true for safety properties, such as invariants.

6.3 Fairness properties

Temporal logic is often used to express what are known as *fairness* properties of systems. These are assumptions about the scheduling of operations in a distributed system. The idea is that not all computations that involve a series of locally enabled transitions need be considered legal in a system. There are those that are excluded because a global fairness property relevant only for unbounded computations

is violated. The advantage of defining fairness properties satisfied by a system is that the remaining computations may satisfy properties not satisfied by the system without a fairness assumption. There are numerous possibilities for defining such fairness assumptions.

One of the most intuitive involves whether every process will have an opportunity to execute enabled actions. If each process is implemented on a separate physical processor, this is nothing more than the reasonable assumption that each processor will execute instructions if there are instructions available for execution. It directly generalizes the basic liveness assumption of any system: that some enabled action will eventually be executed somewhere in the system.

In temporal logic we could assert:

$$\Diamond\Box p \rightarrow \Box\Diamond q$$

If p represents "an operation is enabled" (can be chosen for execution), and q represents, "the operation was just executed", then this means that if the operation becomes continuously enabled from some point on, then eventually it will be executed. Such an assertion is known as *weak fairness*. The form of this assertion is disturbing to some because it seems to be self-contradictory: only if an operation is continuously enabled from some point on, is it guaranteed to be taken. Yet the very act of taking the action might make it no longer be enabled, and thus make the left-hand-side of the implication *false*. This implication of course can equivalently be viewed as the assertion

$$(\neg\Diamond\Box p) \vee (\Box\Diamond q)$$

Now the fairness condition is that either p is not eventually always true or q is repeatedly true.

What seems to be a closer translation of what is intuitively needed by weak fairness might be that from every state, either p is eventually not true, or eventually q is true. However, that assertion

$$\Box((\neg\Diamond p) \vee \Diamond q)$$

is actually equivalent to the one above. This surprising fact (to be proven in the Exercises) holds because of the special nature of eventualities in infinite computations: if they are true in the future for every

state, they are true repeatedly, as seen in the interpretation of $\Box\Diamond p$ as "infinitely often, p".

A stronger fairness assumption could be that if a process is repeatedly enabled, it will be chosen repeatedly for execution. In linear temporal logic, this could be expressed as:

$$\Box\Diamond p \rightarrow \Box\Diamond q$$

This property, known as *strong fairness*, can be difficult to guarantee in a distributed context, because the process could have enabled operations only at points when the scheduler is considering other options, and not have any operations enabled when the scheduler attempts to choose an action from that process.

Many other definitions of fairness are possible, and may be reasonable depending on the execution environment or the abstraction of the system being considered. They can involve the operations of the system, the communication channels, or even arbitrary groups of actions. The expressive power of linear temporal logic allows treating almost all fairness notions that have been investigated in the literature. Fairness assumptions can be exploited in showing properties of systems that include them, and act as part of the specification of the requirements from a scheduler in a system implementation.

6.4 The next question

Exactly which temporal operators are most appropriate in addition to $\Box$ and $\Diamond$ has aroused considerable controversy. One important question is whether a *next* operator is desirable. Such an operator, written $\bigcirc P$ (or XP in the versions where letters are used), is true in the i-th state if P is true in the $(i+1)$-th state. It does add expressive power to the notation, but its critics claim that the power is unneeded, and in fact disruptive. Since we are usually specifying concurrent systems, the fact that a particular state occurs immediately following another state is merely our abstraction of the concurrency into the interleaving model, as described in the introduction. It seems more robust and less subject to random timing considerations to concentrate on key states

that must occur eventually, no matter how atomic steps are interleaved from independent processes.

The next state also poses difficulties for refinements of high level specifications into lower level ones. What seems to be an atomic step on one level, so that it is reasonable to say that the "next" state satisfies a postcondition, may be implemented by a sequence of steps on a lower level, so that the desired postcondition only holds "eventually." If care is not taken in defining when a refinement is a proper implementation of upper-level operations, the temporal logic properties of the upper level that involve the *next* operator will not hold in the lower-level implementations. Thus some researchers consider the next operator undesirable. As a semantic expression of this feeling, Lamport has advocated only using operators that are *insensitive to stuttering*. In other words, the temporal modalities should not allow distinguishing between states that occur once and several contiguous repetitions of this same state. This automatically disqualifies the *next* operator, since $p = \bigcirc p$ is true for any state assertion p when the state repeats, and not true for some p when the next state differs from the present one.

Some of the properties that seem easier to express using the $\bigcirc$ operator are actually instances of realtime properties that are better treated in other ways, as will be seen in Chapter 9.

6.5 The past

Another key question is whether we need to introduce what are known as *past operators*. Most uses of temporal logic for specification have used only the operators we have seen so far, sometimes called the *future fragment* of temporal logic. However, for some years the advisability of including complementary past operators has been suggested. For a given state, the operator

$\boxminus$ is read "so-far" or "always-in-the-past." It means that p is true in every state up to and including the present state.

$\diamondminus$ is read "once" p, and means that p is true in some state up to and possibly including the present.

$p\mathcal{S}q$ is read "p since q" and is the past version of the until operator U. Such an assertion means that p is true in all states from the present back to the latest state in which q is true before the present, and there is such a state.

The formal definitions are similar to those seen for the corresponding future operators, with the inequalities of the indices reversed. For example,

$$(\sigma, i) \models P\mathcal{S}Q \quad iff \quad \exists k.i \geq k \geq 0.((\sigma, k) \models Q \wedge \forall j.i \geq j > k.(\sigma, j) \models P)$$

Of course, similar considerations to those for the future operators hold, so that equalities such as

$$\neg\diamondminus\neg p \quad iff \quad \boxminus p$$

can be shown to hold, and a weak version of the since operator, written $\mathcal{S}_w$, can be defined in which q does not have to be true in a past state, when p is true in all past states.

As before, we can also define a *previous* operator, $\ominus$, analogous to the next operator, which is true in the present state if p was true in the immediately preceding state. However, there is one difference between the past and the future: at least as used in specifying systems by considering possible execution sequences, the past is assumed to be finite, while the future may not be. That is, usually it is natural to assume some initial "first" state, while it is not necessary to assume a last state. This leads to the question of how the past operators should be defined in this first state. For all except the previous operator, there is no difficulty. Since the present is part of the past (just as it part of the future), the assertion is true if it is true in the first state. But how should we interpret $\ominus p$ in the first state? Here we shall define it as identically *false* for any p, since there is no previous state in which p can be evaluated. Then the first state can be identified as the unique state where $\ominus true$ is $false$, since for all other states, this assertion is true. Thus when the initial state needs to be related to other states, the predicate $first$ defined as $\neg\ominus true$ can be used.

Using past operators can help in expressing requirements naturally. For example,

$$\Box(q \rightarrow \diamondminus p)$$

means that in every state, if q is true, it is preceded by a state in which p is true. If q represents "a response was made" and p "a request was made" then this specifies that every response is preceded by a request. In other words, the above assertion means that there will be no spontaneous responses, and is entirely separate from the requirement we saw previously that every request is followed by a response.

Intuitively and formally, the past and the future are connected. For example we have:

$$(\Box p \vee \Box q) \equiv \Box(\boxminus p \vee \boxminus q)$$

$$\Diamond p \wedge \Diamond q) \equiv \Diamond(\diamondminus p \wedge \diamondminus q)$$

The first of these means that either always p or always q is equivalent to from every state being able to look back and always see p in the past, or to look back and to always see q. The only difference is in the point of view: looking forward from the first state, or looking backwards from later states. In fact, by changing the point of view, for an assertion about an entire sequence, with an appropriately rich collection of future operators, an assertion involving past operators can always be expressed without any past operators. (A few examples are seen in the Exercises.)

The question thus is raised again: why use the past operators at all, if they are really not needed for expressibility. There are three answers. The first is that it simply is easier to express many claims that arise naturally in specifications. The second is that using past operators may be better for increasing the compositionality of specifications and modules in a system. This application of the past is seen below. The third reason for using past operators is to allow convenient classification of classes of properties, and is treated in the following section.

6.6 Using the past for compositionality

Compositionality means that component subsystems can be combined into a system in a way that avoids having to reprove properties of the whole system from the beginning. That is, there is a way to combine the specifications of the subsystems so as to obtain the specification of the combined system. Using past operators seems to help avoid having to relate directly to internal variables or control locations of

subsystems when combinations are made. Below we consider two concurrent processes operating asynchronously in parallel, and show that they maintain mutual exclusion between two program sections CRITICAL0 and CRITICAL1, in process 0 and 1, respectively. Recall that the requirement can be expressed in temporal logic as

$$\Box(\neg(in(CRITICAL0) \wedge in(CRITICAL1)))$$

One widespread way to show this property is to use a global invariant. This is an assertion always true in the computation, that relates to the control locations of both processes, including those not in the critical sections. Consider an implemented system with the code:

$$P0 :: repeat\{st_0 : a(); CRITICAL0; en_0 : b()\}$$

$$P1 :: repeat\{st_1 : a(); en_1 : b(); CRITICAL1\}$$

In this program communication is assumed to be synchronous through the synchronization operations a or b. That is, both processes atomically execute a (together) only when they both are ready to do so (i.e., have reached the location where the command appears), and otherwise wait for the other partner to be ready for its part in the communication. The control predicates at, and in will be used, with meanings similar to the ones given in the state transition methods, especially that of Lamport. The invariant then could be:

$$\begin{aligned} at(st_0) &= (at(st_1) \vee in(CRITICAL1)) \wedge \\ at(en_1) &= (in(CRITICAL0) \vee at(en_0)) \end{aligned}$$

That is, control can be at location st_0 in $P0$ if and only if it is either at st_1 or in $CRITICAL1$ in $P1$, and analogously for en_1. This is shown inductively, as explained in Chapter 2. The mutual exclusion then follows immediately by using
$in(CRITICAL1) \rightarrow at(st_0)$ —from the invariant
along with
$at(st_0) \rightarrow \neg in(CRITICAL0)$ —from the definitions of at and in

The only problem with the reasoning above is that the invariant has to relate to the inner workings of both processes, and is global in nature.

It is awkward to reason about each process separately. However, if some past reasoning is added, along with a local predicate a expressing that "an $a()$ operation has just occurred" and another b that "a $b()$ operation has just occurred," then local reasoning can be used compositionally to give the desired result.

For process S_0 the assertion

$$\Box(in(CRITICAL0) \rightarrow (\neg b \mathcal{S}(a \wedge \neg b))$$

can be used. The reasoning used to show this invariant is entirely local to P_0. It is also obviously true from an inspection of the code: the assertion is merely that if control is inside the critical section, then no $b()$ operation has occurred since the last $a()$ operation. The similar assertion for P_1 is

$$\Box(in(CRITICAL1) \rightarrow (\neg a \mathcal{S}(b \wedge \neg a))$$

Again the reasoning is entirely local.

In this example the composition of the two processes means that they are executed in parallel. In this case, an invariant of the composition is obtained simply from the conjunction of the two invariants for the processes: since the proof and reasoning are local, each invariant remains true no matter what occurs in the other process.

The mutual exclusion can be shown by contradiction. Assume that

$$in(CRITICAL0) \wedge in(CRITICAL1)$$

is true in some state. Then from the invariants, we must have both $\neg b \mathcal{S}(a \wedge \neg b)$ and $\neg a \mathcal{S}(b \wedge \neg a)$. That is, no $b()$ operation has occurred since the last $a()$ operation, and no $a()$ operation has occurred since the last $b()$. Since the operations cannot occur at the same time, and both $a(0)$ and $b()$ operations have occurred, this assertion is impossible (consider that a was most recently true later than b, and get a contradiction, and then do the same for b).

The advantage of such modular reasoning is clear: even if one of the processes is changed, only the local reasoning and assertions are affected and all assertions about the unchanged process remain as they were. Whether a modular proof is always possible and whether past modalities are really essential for such reasoning remain open questions.

6.7 Characterizing classes of properties

Another application of temporal logic that uses the past operators is to characterize classes of properties. The goal of such classes is to identify similar properties that have a uniform proof technique. By identifying the kinds of properties used in a specification, it becomes easier to determine that the specification is sufficiently complete. Using past operators allows providing a syntactic characterization of the formulas of each class. For each, all formulas equivalent to those with a particular form are defined to be in the class. In [18], the syntactic classes are shown to correspond to classes of predicate automata, to language theoretic classes, and to complexity classes. Showing that various formulas are equivalent is also a productive source of problems and exercises in temporal logic, that should help familiarity with the notation.

In characterizing the formulas, a generic *past formula* is used. This is any temporal logic formula p that does not include future operators (and thus either relates only to the present state, or has past operators).

Now the categories can be identified, where below p is any past formula:

safety Any formula logically equivalent to one of the form $\Box p$ is a safety property. We have already seen several such properties. For example, mutual exclusion was written as

$$\Box(\neg(in(crit_1) \wedge in(crit_2)))$$

We now can see that a property of the form

$$\Box(q \rightarrow \diamondsuit p)$$

meaning "q is preceded by p" is a safety property. In the Exercises, the reader is asked to show that partial correctness is also a safety property.

guarantee Any property logically equivalent to $\diamond p$ is a guarantee property. This is appropriate for describing a 'one-time' event, such as termination or deadlock. Since total correctness (partial correctness plus termination) can be written as an eventuality of a past formula it too is a guarantee property.

obligation Any property logically equivalent to a conjunction of simple obligations of the form $\Box p \vee \Diamond q$ (where both p and q are any past formula) is an obligation property. Safety and guarantee properties are included in this category, which also has assertions of the form

$$\Diamond p \rightarrow \Diamond q$$

that express for example that if an exception occurs, there will also be an error announcement (not necessarily in that order, for clever implementations).

recurrence Any formula logically equivalent to one of the form $\Box\Diamond p$ is a recurrence property. Recall that this means that p will repeatedly be true in the computation. Since yet another way to write the weak fairness property seen earlier is simply as

$$\Box\Diamond(\neg p \vee q)$$

where p represents that a process has enabled actions, and q represents that one of the enabled actions is executed, this is a recurrence property.

persist Any formula logically equivalent to one of the form $\Diamond\Box p$ is a persistence property. This property means that p is true from some point on in the computation, i.e., that the computation will stabilize to states in which p holds.

reactivity Any formula logically equivalent to a conjunction of simple reactivity of the form $\Box\Diamond p \vee \Diamond\Box q$ is a reactivity property. This class of properties includes expression of strong fairness, since the formula $\neg\Box\Diamond q$ is equivalent to $\Diamond\Box\neg q$, which allows rewriting the above disjunction (after replacing $\neg q$ by r) as

$$\Box\Diamond r \;\rightarrow\; \Box\Diamond p.$$

The classification of properties is important because each class has proof techniques that are appropriate for it. Once a property is recognized as belonging to one of the classes, the way in which it is best verified becomes clearer. For example, safety properties are generally

proven by induction on the possible actions, as we have seen for invariants. One useful technique for identifying safety properties is that they always have what is known as a *finite refutation*. That is, if a sequence does *not* satisfy a safety property p, then there is a finite prefix of the sequence in which p is violated. Thus for mutual exclusion, if it is violated, there is a finite prefix with a state in which more than one process is in its critical section.

For the other classes of properties, there may be no finite refutation. For example, a computation that violates an assertion of termination does not have a finite prefix where termination is violated. The other properties besides safety need to use techniques of well-founded sets that consider the possible values assumed by the state components.

6.8 Branching temporal logic

In the linear approach we have seen so far, the assertions in the logic implicitly always relate to all of the possible execution sequences of a system. The specification is always about a single computation, and we understand this to mean that whatever computation actually occurs in a particular execution, must satisfy the specification. It is thus most natural to consider the set of possible execution sequences, and the assertions about each member of this set.

Those who advocate using a branching temporal logic claim that linear temporal logic, with all its variants, can never be powerful enough to describe all of the interesting properties we might like to specify about a system. They claim that it is also reasonable and possible to relate to the family of computations as such. In other words, perhaps we would like to express that there are computations among the set of executions possible in the system, that have certain properties. A classic example is a random-number generator. Clearly, part of the specification is that a value within some given range is obtained each time the generator is activated. But would we be satisfied with an implementation that always returns the value 53? Somehow, we would like to be able to express that for every value i, there exists some execution sequence that will produce the value i, even if that sequence did not occur this time. Such a property cannot be expressed using any

kind of linear temporal logic.

It should be noted, on the other hand, that the advocates of a linear temporal logic would claim that this is virtually the only common example, and that in general we indeed do want to specify properties true of every execution.

In the context of branching temporal logic, it is most natural to consider the collection of possible executions as being organized into a tree or an acyclic directed graph composed of states as nodes, and atomic events as edges connecting the nodes. From each state the outgoing edges represent the possible steps that can occur from that state, and the states at the other end represent possible next states, in some execution. This semantic view is appropriate because the possibility of selecting which continuation is of interest is recursively definable, and thus corresponds to specifying a path in the graph of possible executions. For this reason, one of the most common branching temporal logics is known as *CTL*, for Computation Tree Logic. In this logic, G is used instead of $\Box$, and F is used instead of $\Diamond$, simply for historical reasons. We also have the modalities A to indicate "for every path in the continuation" and E to indicate "there exists a path in the continuation."

An assertion relating to a node in the execution tree allows grouping together all computations with a common history: the path from the root to that node. Then assertions can be made about which continuations are possible from that point. The assertion

$$AGp$$

is exactly the same as the linear temporal assertion $\Box p$, since the implicit assumption of linear time is that the assertion is true for every computation. On the other hand,

$$EGp$$

means that for some computation, p is true for every state in the computation, but for other computations, that may not be true.

The assertion

$$EFp$$

means that for some computation, there is a state in which p holds. This will be true if somewhere in the computation tree, there is a node that satisfies p. Thus

$$EFAGq$$

means that for some computation, there is a state for which q holds in that state and in all descendants of that state (i.e., there is a subtree in which all nodes satisfy q).

The natural requirement from a random number generator that returns a value in variable x, from a range *Range* would be both the usual requirement that the value is from the required range:

$$AF(terminated \wedge x \in Range)$$

and the added requirement mentioned earlier that every value is a possible outcome:

$$\forall v \in Range.EF(terminated \wedge x = v).$$

To express that from every point there is a continuation in which a key event, such as an interrupt, can occur, we could use the assertion

$$AGEFkey\_event$$

Note that this does not mean that such an event ever *will* occur, only that it always potentially could.

One of the main applications for branching temporal specifications is in verification of finite-state programs, and especially hardware designs. Such programs can be described by a so-called 'execution graph' of states and possible transitions among the states. Efficient algorithms have been developed to determine whether paths through the graph satisfy properties described in branching temporal logic. This problem is known as *model checking*, and it is one of the primary motivations for writing temporal logic specifications. The need to reduce the complexity of model-checking algorithms has led to restrictions on the expressive power of the logics. For example, in CTL only pairs of the form AG, AF, EG, or AF are allowed, because efficient checking algorithms are possible for the properties that can be expressed in this way.

The branching logic in which arbitrary combinations of A and E with the modal operators is known as CTL^*. In this logic, the assertion

$$EGFp$$

means that for some computation, p holds infinitely often along that computation. More complex expressions such as $AGFEFGp$ are also possible. On the other hand, efficient model-checking algorithms do not exist when such full expressiveness is allowed. Therefore linear temporal logic and CTL are incomparable, since each can express properties that the other cannot.

One of the main limitations of CTL is that it cannot express fairness requirements directly within the formalism, since these are implications (or disjunctions) of CTL requirements, that are not themselves in CTL. (In the exercises, the reader is requested to demonstrate that reasonable attempts in CTL do not provide the needed property.) This drawback is considered so serious that some model checking systems provide a mechanism for expressing fairness requirements outside the regular logic. In the model checker SMV, for example, the specification is given as a transition system, plus a collections of objects that are executed fairly (either with weak or strong fairness), along with CTL requirements.

6.8.1 Partial order and interleaving sets

In classic temporal logic, all of the possible computations of the system are considered at once. In the linear version, a single 'typical' execution sequence is specified, and understood as asserting that every execution sequence of the system should satisfy the temporal predicate. In branching time temporal logic, the collection of execution sequences is seen as organized into a tree or directed acyclic graph that satisfies predicates of the logic.

However, in distributed systems it is clear that the collection of operations in a single execution are only partially ordered: independent operations in different processes can be seen as unordered. In the chapter on process algebras, the questions that this partial order raise are reconsidered. Approaches that treat partial order may not form a

global state at all, and may not consider a full interleaving of operations in the system.

However, a temporal logic treatment is possible in which global states and execution sequences are maintained. This approach groups together computations into related subsets. All computations that differ only in that independent operations are done in a different order will be considered equivalent, and will be grouped together into one set. In this view, information is needed on which operations can be considered to be independent of each other. This information can be part of the semantics of the programming language in which the system is to be written. For example, in a distributed system, assignments of constants to local variables of different processes are independent, and the order in which they occur is irrelevant. The semantics of a program then becomes a collection of sets of equivalent computations.

One instance of this approach, the temporal logic $ISTL$, uses the syntax of branching temporal logic with a different meaning, based on the interpretation above. A formula with the branching operators is interpreted as being *true* if it holds for each set of equivalent computations. That is, there is an implicit quantification over all of the equivalence classes (just as there is an implicit quantification over all of the computations in linear temporal logic). This meaning allows specifying some properties that cannot be easily expressed in other logics.

One typical example is in expressing the *serializability* of a database. In distributed databases series of primitive actions are called *transactions*, and there is an unbounded stream of requests to perform transactions such as reading one or more value or writing a new value. A scheduler, called the concurrency control algorithm, allocates which operations should be done at which site of the database in which order. The almost universal requirement in a specification of such a transaction system is known as *serializability*. This specification means simply that even though operations from different transactions can be mixed together, the operations have to be arranged so that the order in which they occur is equivalent to an ordering in which all operations associated with one transaction are done first, followed by all operations from another transaction, etc. Such a requirement is easy to express in $ISTL$, but not in other languages that do not divide the computations into equivalence classes. If the executions where transactions are

executed one-by-one are denoted by a predicate *Serial* true for those executions, then the requirement in $ISTL$ is simply $E\ Serial$. Recall that this means that *every* equivalence class of computations has one that satisfies *Serial*.

Such a logic is also appropriate for specifying cache-consistency algorithms and out-of-order execution in modern computer architectures. In both cases, the key property is that every execution sequence is equivalent to a 'simpler' one that is unoptimized. Distributed communication protocols that can have conflicting and overlapping interchanges among processes are also specified and verified using similar reasoning.

6.9 Bibliographic remarks

Temporal logics have a long history in Mathematics and Philosophy, as one family of modal logics. Work in this area can be seen in the influential book by Prior [23] and later in the work of Kamp [10]. The usefulness of (linear) temporal logic for specifying global properties of concurrent systems was first demonstrated by Amir Pnueli in [22], for the future fragment. The advisability of using past modalities is introduced in [17]. Amir Pnueli and Zohar Manna have written a comprehensive presentation of linear temporal logic, both for specification [18] and for verification [19]. A software verification tool called STeP (developed by Manna and his group at Stanford) is based on the deductive proof system and the specifications from those books. Other early work on versions of temporal logic are by Kroger [13] and Moszkowski [20].

Branching time temporal logic was presented in a propositional version by Clarke, Emerson, and Sistla [5, 4] in CTL, and as a first-order logic in [2].

A detailed survey of the various types of temporal logic may be found in [7].

The division of temporal properties into safety and liveness is due to Lamport [14], who has strongly advocated the use of temporal logics [15], and has developed TLA, the temporal logic of actions [16]. The identification of classes of properties by their syntactic temporal form is from [18], while another syntactic division into safety and liveness

properties is due to [1].

Model checking has been a primary application of temporal logic specifications, in conjunction with a state machine presentation of finite state algorithms. The SMV system of [3] uses CTL, while the Spin system of Bell Labs [9] uses linear temporal logic. The recursive μ - calculus [6] is a version of propositional temporal logic that is easier to manipulate mathematically for model checking, but is less readable as a specification method.

Interleaving sets for partial-order specification and reasoning are defined in [11, 12]. Related ideas of exploiting the partial order among operations are used in model checking in [21, 24] and are surveyed in [8].

Bibliography

[1] B. Alpern and F. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[2] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.

[3] J.R. Burch, E.M. Clarke, K.L. McMillan, D. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

[4] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, volume 131 of *LNCS*. Springer-Verlag, 1981.

[5] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[6] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the 1st Conference on Logic in Computer Science*. IEEE Comp. Soc. Press, June 1986.

[7] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 4, pages 997–1072. Elsevier Science Publishers, 1990.

[8] P. Godefroid. On the costs and benefits of using partial-order methods for the verification of concurrent systems. In *Partial Order Methods in Verification (D. Peled, V.R. Pratt, G.J. Holzmann, eds.), DIMACS Series, vol. 29*, 1997.

[9] G. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall International, 1991.

[10] H. Kamp. *Tense Logic and the Theory of Linear Order.* Ph.D. thesis, Michigan State University, 1968.

[11] S. Katz and D. Peled. Interleaving set temporal logic. *Theoretical Computer Science*, 75:263–287, 1990. Preliminary version appeared in the 6th ACM-PODC, 1987.

[12] S. Katz and D. Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101:337–359, 1992.

[13] F. Kroger. LAR: a logic of algorithmic reasoning. *Acta Informatica*, 8:243–246, 1977.

[14] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. on Software Eng.*, 3(2):125–143, March 1977.

[15] L. Lamport. What good is temporal logic. In *9th World Congress*, pages 657–668. IFIP, 1983.

[16] L. Lamport. The temporal logic of actions. *ACM Trans. Prog. Lang. Syst.*, 16(3):872–923, May 1994.

[17] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Conference on Logics of Programs, LNCS 193*, pages 196–218, 1985.

[18] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.

[19] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems - Safety.* Springer-Verlag, 1995.

[20] B. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, 1983.

[21] D. Peled. Combining partial order reductions with on-the-fly model checking. *Journal of Formal Methods in System Design*, 8:39–64, 1996.

[22] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[23] A. Prior. *Past, Present, and Future.* Clarendon Press, Oxford, 1967.

[24] P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In *Proceedings of CONCUR'93 (Eike Best, ed.), LNCS 715*, 1993.