11 Temporal Logic

In classical logic, the predicate P in "if $P \wedge (P \Rightarrow Q)$ then Q" retains its truth value even after Q has been derived. In other words, in classical logic the truth of a formula is *static*. However, real-life implications are *causal* and *temporal*. To handle these, it must be possible to specify that "an event happened when P was true, moments after that Q became true, and now "P is not true and Q is true". This may also be stated using "states". We associate predicates with states such that a predicate is true in some state S and false in some other states. For example, the statements

P	the train is approaching the gate,
$P \Rightarrow Q$	if the train approaches the gate, the gate is lowered,
Q	the gate is lowered before the train is in the gate, and
R	the gate remains closed until the train crosses the gate

describe changes to the states of a control system that continuously maintains an ongoing interaction with the environmental objects train and gate. The program controlling the train and the gate, according to the specification stated above, is a reactive program. Reactive system refers to the reactive program together with its environment. Such systems are typically non-terminating, read input and produce output on a continuous basis at different system states, interact with other systems and devices in its environment, exhibit concurrent behavior and may have to obey strict timing constraints. For example, the statements P and R denote continuous activities, the statement Q involves the temporal ordering of the actions "gate closing" and "train arriving at the crossing", and the statements P and Q imply concurrency. Such statements cannot be formalized in classical logic. In order to formalize these statements in logic, we need to introduce constructs such as *next*, *always*, *after*, *since*, and *until* and provide semantics to terms involving them. With proper semantics, these terms can express ordinal, temporal, and causal relationships on events and state transitions, without reference to the actual times at which they happen. Temporal logic was developed by Pnueli [22] to describe such orderings. Many kinds of temporal logics exist today, some in which one can add timing constraint like "the gate must be lowered within 3 seconds after receiving the message that the train is approaching". The temporal logics introduced in the following sections are simple extensions of propositional and first order logic and do not involve real-time measurements.

V.S. Alagar, K. Periyasamy, *Specification of Software Systems*,
Texts in Computer Science,
DOI 10.1007/978-0-85729-277-3_11, © Springer-Verlag London Limited 2011

11.1 Temporal Logic for Specification and Verification

Temporal logics can support specification and reasoning at different levels of abstractions. They provide unified logical systems in which behavior specification, design specification, and implementation level details can be expressed and related in an intuitive manner.

1. *Requirements Description:* At the requirements level, propositions and predicates are determined to model the problem requirements, and temporal formulas are constructed to formalize the temporal properties of the requirements. In particular, functionalities that must be present "always", or "at some future instance", or "always from some future instance" can be stated as temporal logic formulas [15].
2. *Design Level Specification:* At the design level, the behavior of an object can be characterized by a sequence of states, and the events triggering the successive state transitions. Temporal logic formulas are used to assert properties that hold over (1) all sequences of states, (2) some sequences of states, and (3) some future state in some sequence. Temporal logic can be used to interpret sequential and concurrent actions [16, 17, 22].
3. *Program Specification:* An interpretation of a program using temporal logic becomes a temporal specification of the program [19, 22, 24]. Verification of program properties can be done by stating each property as a temporal formula and then showing that the specification satisfies the formula.
4. *Formal Verification:* The rules of temporal logic proof calculus are applied to show the correctness of a temporal logic specification with respect to more abstract system specifications [5–7, 16]. Even when the design specifications use a different formal notation, temporal logic may be brought in at formal verification stages, as done in model checking.

Many types of concurrent and reactive systems can be modeled in temporal logic and the critical properties of the system can be formally verified in the model. In particular, in the specification of concurrent or reactive systems, three important behavioral properties, termed *safety*, *liveness*, and *fairness* by Owiciki and Lamport [20], can be formally expressed.

- *Safety:* Informally, a safety property implies that *something bad will not happen*. Typical examples are:

  1. *the gate will remain closed while a train is crossing the gate*
  2. *water level in the boiler should be at least 3 meters, and the reactor temperature must be less than 3000 degrees*.

  Other examples of safety properties include

  3. *partial correctness*—the program does not produce stack overflow
  4. *mutual exclusion*—two processes are not simultaneously in the critical section
  5. *deadlock-freedom*—the program does not reach the state of deadlock.

- *Liveness:* Liveness property implies that *something good will eventually happen*. Typical examples are:

1. *whenever the gate is directed to raise, it will eventually do so*
2. *program terminates eventually.*

- *Fairness:* Fairness property implies that whenever an attempt is made to perform an action or request a service, it will eventually succeed. Typical examples are:

1. *starvation-freedom*, where a process does not wait forever to be serviced
2. *progress*, where every message sent in a channel is eventually received.

11.2 Concept of World and Notion of Time

The term "world" means a *frame* or a *state* and is characterized by a set of dimensions such as time, space, audience, and events. Natural language expressions are interpreted in *intentional logic* [4] by evaluating the expression over different *modes* that are worlds. In Example 1, taken from Wan [30], a pair (*month*, *location*) determines a world, which is a unit of time/space.

Example 1 An evaluation of the expression

E': the average temperature this month here is greater than 0°C

can be obtained by interpreting it along the dimensions *month* and *location*. The table below gives only a partial evaluation because there exists many locations not included in the table.

	Jan	*Feb*	*Mar*	*Apr*	*May*	*Jun*	*Jul*	*Aug*	*Sep*	*Oct*	*Nov*	*Dec*
Montreal	F	F	F	F	T	T	T	T	T	F	F	F
Ottawa	F	F	F	T	T	T	T	T	T	F	F	F
Toronto	F	F	T	T	T	T	T	T	T	T	F	F
Vancouver	F	T	T	T	T	T	T	T	T	T	T	T

□

In temporal logic, "world" refers only to time points. A world is like a model at a specific moment in time that assigns truth values to atomic propositions. To navigate between the different worlds, an *accessibility relation* needs to be defined. With this relation as we navigate through the worlds, we are moving *through time*. So, when we refer to "time" we are not referring to absolute time (or clock time) but we are emphasizing only the temporal relation defined by the accessibility relation.

11.2.1 Temporal Abstraction

The accessibility relation may be explained through temporal abstraction. We describe this notion following the discussion in [18]. As we move from requirements specification, which is abstract, to a design specification, which is concrete, more details are added. An abstract specification may specify a component behavior as a state machine, emphasizing only on the correct sequence of actions. The times at which these actions must occur might have been left out, and may be added during the design of the component. In some cases, time information might be stated at the abstract level only at some states which are considered *time-critical*. The details of time left unspecified at other intermediate states should not affect the overall behavior of the component, in particular during the design stage when more detailed time information may be added. Even at the design stage, the stated times may not be the *exact* time points, rather only the time points that are *relative* to the stages through which the timed behavior evolves. The abstract specification represents a *temporal abstraction* of the more detailed behavior given by the design. Continuing this discussion, it is understood that the design represents a temporal abstraction of the program. A correctness condition should establish a relationship between these different formal representations of time.

In many problems such as hardware specifications and reactive system specification, each unit of time in an abstract specification corresponds to an interval of time in its concrete counterpart. It may be that a particular moment at the abstract level corresponds to some moment within an interval or to a set of adjacent time moments in the concrete level. Consequently, a correctness condition formulated at the abstract level, which involves only "coarse-grained" time must be consistent with the "fine-grained" time scale in the concrete level. This correspondence between the time scales is described by a function f, that assigns for every t_a in the abstract level a time point $t_c = f(t_a)$ at the concrete level such that the order of time is preserved:

$$\vdash \forall t_a, t'_a.(t_a < t'_a) \Rightarrow (t_c < t'_c)$$

The properties of time that should hold in the design will determine the function f. The inverse of f need not be a function, because many possible designs exist for an abstract specification. If f denotes the mapping from the requirements to the design and g denotes the mapping from the design to the implementation then the function $f \circ g$ is the time map from the requirements to the implementation. The advantage of temporal abstractions include

- irrelevant details about intermediate states are suppressed in the abstract level, and
- time scale is not absolute, and consequently a specification can focus on relative time points without being specific about "which" time points are in fact of interest.

11.2.2 Discrete or Continuous

When certain computations or actions need to be described as continuously varying, a dense model of time is appropriate. In such a case, the topology of time is that of real numbers or a subset of real numbers. When a property is present over a sequence of intervals, and in each interval the property persists in every sub-interval, then an interval and continuous model of time are used. When dealing with properties that are present only at certain time instants, a discrete model of time is chosen. In this case, the model of time is isomorphic to a subset of natural numbers.

11.2.3 Linear and Branching Models of Time

One may want to postulate that for any state (moment in time) there is either exactly one next state or several possible next states. The former case is called the *linear model* of time, and the latter case is called the *branching model* of time. The branching model is useful to handle uncertainty, and hence possible alternative futures. Properties such as "p is true *throughout every* possible future", "p is true *at the next time in some possible* future", "p is true *eventually in every* possible future", and "p is true *eventually in some possible* future" can be stated and reasoned about in branching time temporal languages.

11.2.4 Further Specializations of Time

In distributed computing and communication over shared networks, there is a need to make a distinction between *global* and *local* times. The other possibilities include *bounded time* and *interval time*. A bounded time applies to dense (discrete) model when a finite subset of real numbers (natural numbers) is chosen to model time. An interval time model has a discrete time structure, so that the next and previous intervals can be referred to. Within each interval, a property may hold in every sub-interval or only at specific discrete points. In addition, time modeled as intervals may be bounded or unbounded. As a result of this classification, many different models of time can be realized. Some of these include "discrete linear global time", "discrete branching local time", and "continuous branching interval time".

11.3 Propositional Temporal Logic (PTL)

PTL is a *discrete linear-time* temporal logic introduced by Pnulei [22]. The primary features are that (1) time structure at the abstract level is the set of non-negative integers, (2) for every moment in time there is only one future moment in time, and (3) only propositions are allowed in formulas.

In PTL, the set of propositional logic operators is extended with the temporal operators $\Box$ (*always*), $\Diamond$ (*eventually*) and $\bigcirc$ (*next*). Intuitive meanings of temporal operators used in first order formulas φ and ψ are given in the table below.

Formula	Intuitive meaning
$\Box\varphi$	φ is true in *all* future moments
$\Diamond\varphi$	φ is true at *some* future moment in time
$\bigcirc\varphi$	φ is true in the *next* moment in time
$\varphi\,\mathcal{U}\,\psi$	φ is true up and until some future moment when ψ becomes true

With the above intuitive meanings, we can formalize a few requirements as in Example 2.

Example 2

- Always, if an email is sent through the network, then it will eventually be delivered.

$$\Box(send\_email \Rightarrow \Diamond delivered)$$

- If a car is parked and the meter has expired, then at the next moment it will be ticketed.

$$(parked \wedge meter\_expired) \Rightarrow \bigcirc\, ticket\_for\_violation$$

- The university library never closes.

$$\Box library\_open$$

- Always, after the machine gets a coin and the user press a button, it gives coffee or tea.

$$\Box(coin \wedge \bigcirc press\_Button \Rightarrow \Diamond(serve\_Coffee \vee serve\_Tea))$$

- Always, after pressing a button the machine will serve coffee and then tea immediately afterward.

$$\Box(press\_Button \Rightarrow (serve\_Coffee \wedge \bigcirc\, serve\_Tea))$$

- The gate remains closed until the train leaves the crossing.

$$gate\_closed\,\mathcal{U}\,train\_exits$$

$\Box$

11.3.1 Syntax

The vocabulary of temporal logic consists of a finite set *PROP* of propositional symbols. The logical connectives *true*, *false*, $\vee$, $\wedge$, and $\Rightarrow$ together with the temporal connectives $\bigcirc$, $\Diamond$, $\Box$, and $\mathcal{U}$ are added to the vocabulary.

The unary operators bind *stronger* than binary ones. Operators $\neg$, $\bigcirc$, $\Box$, and $\Diamond$ bind equally strong. As an example, the formula $\neg P\mathcal{U}Q$ is interpreted as $(\neg P)\mathcal{U}Q$. Temporal binary operators have stronger precedence over $\wedge$, $\vee$, and $\Rightarrow$. The formula $P \vee Q\mathcal{U}R$ is parsed as $P \vee (Q\mathcal{U}R)$. Parentheses should be inserted to override precedence.

The set of well-formed formulas of PTL, denoted wff_T, is inductively defined as the smallest set of formulas satisfying the following rules.

1. We add a special proposition symbol *start* which is true only at the *beginning of time*. It is not meaningful to use *start* at any other state.
2. Every proposition P in *PROP* is in wff_T.
3. The constants *true*, *false*, and *start* are in wff_T.
4. If φ is in wff_T, then (φ) and $\mathcal{T}(\varphi)$, where $\mathcal{T}$ is a temporal operator, are also in wff_T. The formulas $\neg(\varphi), \Box(\varphi), \Diamond(\varphi)$ and $\bigcirc(\varphi)$ are well-formed.
5. If φ and ψ are in wff_T, then so are $(\varphi \vee \psi)$, $(\varphi \wedge \psi)$, and $(\varphi \Rightarrow \psi)$.

As an example, the formulas $\varphi \Rightarrow \Box\Diamond(\psi \vee \chi)$, and $\varphi \wedge \Diamond(\psi \Rightarrow \varphi)$ are well-formed, whereas the formulas $\psi\Diamond\varphi$, and $\varphi \wedge \Box\chi \bigcirc \psi$ are not well-formed. Stating well-formed formulas in natural language may be quite cumbersome. For example, stating the formula $\Box((\varphi \Rightarrow \bigcirc\psi)$ in natural language is not that difficult: it states that at every moment in time the property "if φ is true then at the next moment ψ is true" holds. However, stating the formula $\Box(\varphi \wedge \Diamond\psi) \Rightarrow \Box\,(\varphi \Rightarrow \bigcirc\chi)$ in natural language is more difficult. A semantic model helps us to precisely interpret formulas and understand their meanings.

11.3.2 Model and Semantics

The time structure for PTL is $\mathbb{N}_0$, the set of non-negative integers. Each state $i \in \mathbb{N}_0$ is associated with a set of propositions that are *true* in that state. That is, the formal basis of the semantic interpretation of PTL is a *sequence* of propositional models. Formally, the models of PTL is defined by

$$\mathcal{M} = \langle \mathbb{N}_0, \pi \rangle,$$

where π maps state $i \in \mathbb{N}_0$ to $\pi(i) = s_i \subset PROP$. $\mathcal{M}$ is called model variety. An alternate representation of the model variety is

$$\mathcal{M} = \langle s_0, s_1, s_2, \ldots \rangle$$

A pictorial representation of $\mathcal{M}$ is given below.

Discrete Linear Model

$$\begin{tikzcd}
{} \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & {} \\
 & & & & & & \{P, Q\} \arrow[u] & \{Q\} \arrow[u] & \{P, R\} \arrow[u] & & & & &
\end{tikzcd}$$

11.3.3 Formal Semantics

The semantics of PTL formulas for a model $\mathcal{M}$ is an interpretation relation

$$\models (\mathcal{M} \times \mathbb{N}_0) \mapsto \mathbb{B}$$

where $\mathbb{B} = \{true, false\}$. For a formula φ

$$\langle \mathcal{M}, i \rangle \models \varphi$$

is true if φ is satisfied at state s_i within the model $\mathcal{M}$. This satisfaction relation is built inductively, defined first for propositions, next for formulas involving classical logic connectives, and finally for general temporal formulas.

1. *Semantics of Propositions:* A proposition $P \in PROP$ satisfies $\langle \mathcal{M}, i \rangle$ iff $P \in s_i$. Formally,

   $$\langle \mathcal{M}, i \rangle \models P \textit{ iff } P \in s_i$$

   The proposition *start* is true only at the beginning of time.

   $$\langle \mathcal{M}, i \rangle \models start \textit{ iff } (i = 0)$$

2. *Semantics of Standard Logic Formulas:* We consider formulas involving $\{\neg, \wedge, \vee, \Rightarrow\}$.

   $$\langle \mathcal{M}, i \rangle \models \neg\varphi \textit{ iff } \langle \mathcal{M}, i \rangle \models \varphi \text{ is not true.}$$

   $$\langle \mathcal{M}, i \rangle \models \varphi \wedge \psi \textit{ iff both } \langle \mathcal{M}, i \rangle \models \varphi \textit{ and } \langle \mathcal{M}, i \rangle \models \psi \text{ are true.}$$

   $$\langle \mathcal{M}, i \rangle \models \varphi \vee \psi \textit{ iff either } \langle \mathcal{M}, i \rangle \models \varphi \textit{ is true or } \langle \mathcal{M}, i \rangle \models \psi \text{ is true.}$$

   $$\langle \mathcal{M}, i \rangle \models \varphi \Rightarrow \psi \textit{ iff if } \langle \mathcal{M}, i \rangle \models \varphi \textit{ is true then} \langle \mathcal{M}, i \rangle \models \psi \text{ is true.}$$

3. Semantics of Temporal Logic Formulas: We consider formulas involving $\bigcirc, \Box, \Diamond, \mathcal{U}$.

   $$\langle \mathcal{M}, i \rangle \models \bigcirc\varphi \textit{ iff } \langle \mathcal{M}, i+1 \rangle \models \varphi$$

   $$\langle \mathcal{M}, i \rangle \models \Box\varphi \textit{ iff } \forall j (j \geq i) \langle \mathcal{M}, j \rangle \models \varphi \textit{ holds}$$

   $$\langle \mathcal{M}, i \rangle \models \Diamond\varphi \textit{ iff } \exists j (j \geq i) \langle \mathcal{M}, j \rangle \models \varphi \textit{ holds}$$

   $$\langle \mathcal{M}, i \rangle \models \varphi \mathcal{U} \psi \textit{ iff } \exists j (j \geq i) \textit{ and } \langle \mathcal{M}, j \rangle \models \psi, \textit{and}$$
   $$\forall k (i \leq k < j) \langle \mathcal{M}, k \rangle \models \varphi \textit{ holds}$$

11.3.4 More Temporal Operators

We describe three more temporal operators below. Manna and Pnueli [17] includes an exhaustive list of temporal operators.

The *Unless* Operator $\mathcal{W}$

The formula $\varphi\ \mathcal{W}\ \psi$ (read φ unless ψ) states that φ is true unless ψ becomes true. The difference between *Until* and *Unless* operators is that in the former case ψ is *guaranteed* to be true at some future state, whereas in the latter case ψ may never become true. Consequently, $\varphi\ \mathcal{W}\ \psi$ specifies two scenarios: (1) ψ becomes true at some future state and the behavior is identical to the *until* operator, and (2) ψ never becomes true and consequently φ is true forever. Thus the formal semantics is

$$\langle \mathcal{M}, i\rangle \models \varphi \mathcal{W} \psi \textit{ iff either } \langle \mathcal{M}, i\rangle \models \varphi \mathcal{U} \psi \textit{ or } \langle \mathcal{M}, i\rangle \models \Box\varphi$$

The *Since* Operator $\mathcal{S}$

The formula $\varphi\ \mathcal{S}\ \psi$ (read φ since ψ) states that ψ has happened sometime in the past and φ has continuously held since then. Notice that in the model for PTL there is no moment in time that is less than 0. Thus the formal semantics is

$$\langle \mathcal{M}, i\rangle \models \varphi \mathcal{S} \psi \textit{ iff } \exists j (0 \le j < i) \langle \mathcal{M}, j\rangle \models \psi \textit{ and}$$
$$\forall k (k \ge j) \langle \mathcal{M}, k\rangle \models \Box\varphi$$

The *Release* Operator $\mathcal{R}$

The formula $\varphi\ \mathcal{R}\ \psi$ (read φ release ψ) states that ψ has to be true at all states until and including the state at which φ first becomes true. If φ never becomes true then ψ must remain true forever. Notice that if φ becomes true at state s_i then ψ is not true from state s_{i+1} (unless something else happens), but we cannot assert anything about ψ after state s_i. Recognize that the *Release* operator is *the inverse* of the *Unless* operator.

Example 3 The formulas below are interpreted according to the above semantics.

1. $\textit{water\_level\_high} \Rightarrow \bigcirc \textit{alarm\_rings}$: if the water level is high in state s_i, $i \ge 0$, then at state s_{i+1} alarm rings.
2. $\neg(\Box(\textit{get\_rank} \wedge \textit{excel\_in\_sports})$: getting a rank in studies and excelling in sports is not always possible.
3. $\textit{system\_down} \Rightarrow \Diamond \textit{system\_up}$: if the system is down at state s_i then at some future state s_j, $j \ge i$ the system will be come back up.
4. $\textit{file\_front} \wedge \textit{printer\_on} \Rightarrow (\Diamond \textit{start\_print} \wedge \textit{printing} \mathcal{U} \textit{file\_end})$: if the printer is on and the front of a file is received by it at state s_i then the printer will start printing the file at some future state s_j, $j \ge i$, and continue printing until the end of file is reached at some state k, $k > j$.
5. $\textit{execute\_program}\ \mathcal{W}\ \textit{stackoverflow}$: the program execution is continued unless there is stack overflow.
6. $\textit{cross\_gate}\ \mathcal{S}\ \textit{gate\_open}$: the gate remains open since the train crossed the gate.
7. $\textit{arrived\_at\_terminal}\ \mathcal{R}\ \textit{driver\_on\_duty}$: upon arrival of train at the terminal the driver on duty is released. $\Box$

11.3.5 Axioms

A sample set of axioms for PTL is given here. A set of axioms are grouped under a *law*. An axiom gives the equivalence between two PTL formulas. In general, many formulas equivalent to a given formula may exist. The axioms provide a basic set, using which the number of equivalent formulas may be reduced to those shown in the list below. For a complete list of axioms, principles of soundness, completeness, and axiomatization of systems consult [10, 17, 26, 29].

- *Absorption Axioms:* The absorption law removes operators whose effect are absorbed by other operators in a formula. If, always, it is true that φ is true infinitely often in a future time moment then it must be true at every future state starting at time 0. The converse is also true.

Absorption Law

$$\Diamond\Box\Diamond\varphi \equiv \Box\Diamond\varphi$$
$$\Box\Diamond\Box\varphi \equiv \Diamond\Box\varphi$$

- *Distributive Axioms:* The distributive law for $\Diamond$ distributes over *disjunction*, the distributive law for $\Box$ distributes over *conjunction*, and these laws are dual to each other. It should be noted that $\Diamond$ does not distributes over *conjunction*, and $\Box$ does not distribute over *disjunction*. Informally this can be reasoned about. For example, the formula $\Diamond\varphi \wedge \Diamond\psi$ asserts that eventually a state in which φ holds and a state in which ψ holds will be reached. But that does not mean that a state in which both φ and ψ are true will be reached. However, the operator $\bigcirc$ distributes over *conjunction* and $\mathcal{U}$ (until).

Distributive Law

$$\Diamond(\varphi \vee \psi) \equiv \Diamond\varphi \vee \Diamond\psi$$
$$\Box(\varphi \wedge \psi) \equiv \Box\varphi \wedge \Diamond\psi$$
$$\bigcirc(\varphi \wedge \psi) \equiv (\bigcirc\varphi) \wedge (\bigcirc\psi)$$
$$\bigcirc(\varphi\mathcal{U}\psi) \equiv (\bigcirc\varphi)\mathcal{U}(\bigcirc\psi)$$

- *Duality Axioms:* The duality rule for the next-step operator $\bigcirc$ says that it is dual to itself. Read it as "it is not the case that φ is true in the next step is equivalent to saying that it is the case that φ is not true in the next step". The duality rule for the future operator $\Diamond$ involves the always operator $\Box$, and conversely the duality rule for the always operator $\Box$ involves the future operator $\Diamond$. Read the duality rule for $\Box$ operator as "it is not the case that φ is always true is equivalent to saying that at some future time point φ is not true". Interchange the operators $\Box$ and $\Diamond$ in a rule to get (read) the other rule.

Duality Law

$$\neg\Box\varphi \equiv \Diamond\neg\varphi$$
$$\neg\Diamond\varphi \equiv \Box\neg\varphi$$
$$\neg\bigcirc\varphi \equiv \bigcirc\neg\varphi$$

- *Expansion Axioms:* Expansion axioms provide an *inductive* definition of temporal operators. As an example, if $\Diamond\varphi$ holds in the beginning, then either φ is true or $\bigcirc\Diamond\varphi$ is true in the beginning. In general, each law will assert the truth of a formula which does not involve a temporal operator in the current state, and it asserts the truth of the formula in the next state using the next state operator $\bigcirc$.

Expansion Law

$$\Box\varphi \equiv \varphi \wedge \bigcirc\Box\varphi$$
$$\Diamond\varphi \equiv \varphi \vee \bigcirc\Diamond\varphi$$
$$\varphi\,\mathcal{U}\,\psi \equiv \psi \vee (\varphi \wedge \bigcirc(\varphi\,\mathcal{U}\,\psi))$$

- *Idempotent Axioms:* The idempotent axioms eliminate the superfluous operator from a formula. For example, it is not necessary to say that "*always* a formula is always true".

Idempotent Law

$$\Box\Box\varphi \equiv \Box\varphi$$
$$\Diamond\Diamond\varphi \equiv \Diamond\varphi$$
$$\varphi\,\mathcal{U}\,(\varphi\,\mathcal{U}\,\psi) \equiv \varphi\,\mathcal{U}\,\psi$$
$$(\varphi\,\mathcal{U}\,\psi)\,\mathcal{U}\,\psi \equiv \varphi\,\mathcal{U}\,\psi$$

- *Induction Axiom:* The axiom is similar to expansion law, applied to "implication". It asserts that temporal logic provides an inductive rule for proving assertions.

Induction Law

$$\vdash \Box(\varphi \Rightarrow \bigcirc\varphi) \Rightarrow (\varphi \Rightarrow \Box\varphi)$$

It is instructive to verify the equivalence of two formulas by proving that the semantic models of the formulas are identical. See exercises.

11.3.6 Formalizing Properties in PTL

We give examples showing propositional temporal logic characterizations for safety, liveness and fairness properties.

Example 4 Consider the *mutual exclusion problem* for two processes P_1 and P_2. A process has (1) a critical section, (2) a non-critical section, and (3) a section where it waits before entering the critical section. Let $wait_i$ denote that process P_i is in its waiting phase, and $criti_i$ denote that P_i is in its critical section. The *safety* property stating that P_1 and P_2 never simultaneously have access to their critical sections is expressed by the formula

$$\Box(\neg criti_1 \vee \neg criti_2)$$

The *liveness* property stating that each process P_i is infinitely often in its critical section is expressed by the formula

$$(\Box\Diamond criti_1) \wedge (\Box\Diamond criti_2)$$

A fairness property is that every waiting process will eventually enter its critical section. This property is also called *starvation freedom*. It can be expressed by the formula

$$(\Box\Diamond wait_1 \Rightarrow \Box\Diamond criti_1) \wedge (\Box\Diamond wait_2 \Rightarrow \Box\Diamond criti_2) \qquad \Box$$

Example 5 This example, called *Dining Philosophers*, is due to Dijkstra [8]. Five philosophers are sitting at a round table with a bowl of rice in the middle. The philosophers think, eat, and may have to wait before they eat. In between two neighboring philosophers there is only one chopstick. To take some rice out of the bowl and eat, a philosopher needs two chopsticks (eating by hand is forbidden!). So, a philosopher must take the chopsticks from the left and right side in order to eat. But if one philosopher keeps both chopsticks, the two neighbors will starve. The problem is to design a protocol for the dining so that the complete system is *deadlock-free*. That is, at least one philosopher can eat and think infinitely often. Additionally, some fairness is needed—every philosopher should be able to eat and think infinitely often. To specify deadlock-freedom, let us denote the philosophers by $0, 1, 2, 3, 4$. Let $wait_i$ be the proposition meaning that philosopher i is waiting. Let $occupied_i$ be the proposition that chopstick i is in use. The formula that expresses deadlock-freedom is

$$\Box\neg\left(\bigwedge_{0\le i<5} wait_i \wedge \bigwedge_{0\le i<5} occupied_i\right) \qquad \Box$$

Example 6 This example describes the temporal properties of a traffic light with phases R (red), G (green), and Y (yellow). We can consider the phases as "instances" (atomic). That is, a traffic light in phase R will go to the *next* phase G. That is, there is "nothing" in between R and G. We are not interested at "how long" the light will be red before turning yellow. With this assumption, the formula that expresses the "transition from *state* (phase) R to *state* (phase) G is

$$\Box(R \Rightarrow \bigcirc G)$$

However, if you want to consider *durations*, then "the traffic light remains red for sometime before turning green". Since we do not know for how long it will remain red, we can only state that eventually the light will become green.

$$\Box(R \Rightarrow \Diamond G)$$

If we want to be more descriptive of the behavior, we may say "once green, the light eventually becomes red after being yellow for sometime".

$$\Box(G \Rightarrow \bigcirc(G\mathcal{U}(Y \wedge \bigcirc(Y\mathcal{U}R))))$$

The traffic light is green infinitely often.

$$\Box\Diamond G$$

The traffic light is red only in a finite number of states.

$$\Diamond\Box\neg R$$ □

11.3.7 Specifications

In this section, we discuss methods for specifying communication and concurrency, and illustrate them in specifying reactive systems.

11.3.7.1 Communication and Concurrency

We consider a system involving several objects (components or agents) who may interact among themselves in order to achieve their goals. Essentially, each object is independent of the other, except when it needs a resource or information from another object in order to continue its activity. In a discrete time model, all components have a common global clock and have the same definition of next moment. In the discussion below, we let $Spec_A$ and $Spec_B$, respectively, denote the specifications of components A and B, $\langle \mathbb{N}_0, \pi_1 \rangle$ denote the model of $Spec_A$ and $\langle \mathbb{N}_0, \pi_2 \rangle$ denote the model of $Spec_B$.

True Concurrency The execution of components progress independently and simultaneously through the same sequence of states. This type of concurrency is called *synchronous*. There is no interaction between the propositions/formulas in $Spec_A$ and the propositions/formulas in $Spec_B$. The specification of the combined system that execute concurrently and synchronously is $Spec_A \wedge Spec_B$, whose model is $\langle \mathbb{N}_0, \pi \rangle$, where $\pi(i) = \pi_1(i) \wedge \pi_2(i)$, for $i \in \mathbb{N}_0$.

Example 7

$$Spec_A : \Box \begin{bmatrix} start \Rightarrow P \wedge \\ P \Rightarrow \bigcirc \bigcirc Q \wedge \\ Q \Rightarrow \bigcirc P \end{bmatrix}$$

$$Spec_B : \Box \begin{bmatrix} start \Rightarrow A \wedge \\ A \Rightarrow \bigcirc B \wedge \\ B \Rightarrow \bigcirc \bigcirc A \end{bmatrix}$$

The model for $Spec_A$ is $\langle \mathbb{N}_0, \pi_1 \rangle$, where

$$\pi_1(i) = \begin{cases} P & \text{if } i = 3k, k \geq 0 \\ Q & \text{if } i = 3k+2, k \geq 0 \end{cases}$$

The model for $Spec_B$ is $\langle \mathbb{N}_0, \pi_2 \rangle$, where

$$\pi_2(i) = \begin{cases} A & \text{if } i = 3k, k \geq 0 \\ B & \text{if } i = 3k+1, k \geq 0 \end{cases}$$

The model for $Spec_A \wedge Spec_B$ is $\langle \mathbb{N}_0, \pi \rangle$, where

$$\pi(i) = \begin{cases} A \wedge P & \text{if } i = 3k, k \geq 0 \\ B & \text{if } i = 3k+1, k \geq 0 \\ Q & \text{if } i = 3k+2, k \geq 0 \end{cases}$$

□

Interleaving Executions Two components execute independently; however, only one of the components can execute during a certain period of time. The model $\langle \mathbb{N}_0, \pi \rangle$ of this behavior is given by

$$\pi(i) = \begin{cases} \pi_1(i) & \text{if } A \text{ is executing} \\ \pi_2(i) & \text{if } B \text{ is executing} \end{cases}$$

Executions Triggered by Messages Message passing can be specified in one of three possible ways.

- *Peer-to-peer communication:* An object A sends a message *directly* to another object B.
- *Broadcast Communication:* An object sends a message to several objects; however, it does not know the recipients of the message.
- *Multicast Communication:* This is a restricted form of broadcast, which specifies a constraint based on which the set of recipients are characterized. The sender does not know the actual recipients of the message.

Messages are sent through *channels*, which have the following characteristics:

- channel is *unidirectional*,
- channel does not lose messages, and
- messages are delivered in the order they are sent.

To specify interaction between components A and B through message passing, it is necessary to have two channels, one for component A to send messages to component B, and another for component A to receive messages from component B. We view the set of propositions in each specification to be partitioned into three sets *input*, *output*, and *internal*. A proposition $P \in output_A$ and a proposition $\overline{P} \in input_B$ are related by the connective $\Rightarrow$ to show the direction of flow of the message from component A to component B. Similarly, a proposition $Q \in output_B$ and a proposition $\overline{Q} \in input_A$ are related by the connective $\Rightarrow$ to show the direction of flow of the message from component B to component A. Depending upon the type of communication, quality of service criteria (QoS) and the medium of communication, the message is delivered in one of the following ways:

- the message is delivered *instantaneously*: $\Box[P \Rightarrow \bar{P}]$
- the message is delivered at the next moment: $\Box[P \Rightarrow \bigcirc\bar{P}]$
- the message is delivered at some future moment $\Box[P \Rightarrow \Diamond\bar{P}]$.

There exist other possibilities such as "the message is delivered at a *specific* time", or "the message is delivered within a certain interval of time" which we do not consider.

In Example 8, *send_msg* $\in output_A$ and *rcv_msg* $\in input_B$, and the communication specification states that message *send_msg* is sent by component A to component B. When it is received at the next moment in time, the proposition *rcv_msg* becomes true.

Example 8

$$Spec_A : \Box \begin{bmatrix} start \Rightarrow P \wedge \\ P \Rightarrow \bigcirc \bigcirc Q \wedge \\ Q \Rightarrow \bigcirc P \wedge \\ Q \Rightarrow send\_msg \end{bmatrix}$$

$$Spec_B : \Box \begin{bmatrix} rcv\_msg \Rightarrow A \wedge \\ A \Rightarrow \bigcirc B \wedge \\ B \Rightarrow \bigcirc \bigcirc A \end{bmatrix}$$

The communication specification is
$Spec_{AtoB}$:

$$\Box[send\_msg \Rightarrow \bigcirc rcv\_msg]$$

The specification of the message passing system is

$$Spec_A \wedge Spec_B \wedge Spec_{AtoB}$$

The behavior of the specification is best described by looking at its model. Component A is executed starting in its initial state. In the initial state P is true. The second line of $Spec_A$ ensures that Q becomes true at time point 2. The third and fourth lines of $Spec_A$ assert that the proposition *send_msg* is made true at time point 2, and P is made true at time point 3. From this moment on, P is true at time points $6, 9, \ldots$, Q is true at time points $5, 8, \ldots$, and *send_msg* is true whenever Q is true. The interaction specification

$send\_msg \Rightarrow \bigcirc rcv\_msg$ ensures that the proposition rcv_msg becomes true the moment after $send\_ msg$ is true. Since the proposition $send\_msg$ is true at time points $2, 5, \ldots$, the proposition rcv_msg becomes true at time points $3, 6, \ldots$. The first line of $Spec_B$ ensures that proposition A is true whenever proposition rcv_msg is true. The second line of $Spec_B$ affirms that proposition B is true the moment after proposition A is true. The last line of $Spec_B$ affirms that the behavior described thus far is repeated. Consequently, proposition A is true at time pints $3, 6, \ldots$, and proposition B is true at time points $4, 7, \ldots$. Notice that in the combined specification, the propositions $send\_msg$ and rcv_msg do not arise. That is, propositions in the sets $output_A$ and $input_B$ "cancel out", leaving the combined behavior to be described only by the propositions in $inner_A$ and $inner_B$. The models of specifications $Spec_A$, $Spec_B$, and $Spec_A \wedge Spec_B \wedge Spec_{AtoB}$ are given below.

Model of $Spec_A$

$$\pi_A(i) = \begin{cases} P & \text{if } i = 3k, k \geq 0 \\ Q & \text{if } i = 3k+2, k \geq 0 \\ send\_msg & \text{if } i = 3k+2, k \geq 0 \end{cases}$$

Model of $Spec_B$

$$\pi_B(i) = \begin{cases} A & \text{if } i = 3k, k \geq 1 \\ B & \text{if } i = 3k+1, k \geq 1 \\ rcv\_msg & \text{if } i = 3k+1, k \geq 1 \end{cases}$$

Model of $(Spec_A \wedge Spec_B \wedge Spec_{AtoB})$

$$\pi(i) = \begin{cases} P & \text{if } i = 0 \\ P \wedge A & \text{if } i = 3k, k \geq 1 \\ Q & \text{if } i = 3k-1, k \geq 1 \\ B & \text{if } i = 3k+1, k \geq 1 \end{cases}$$

□

11.3.7.2 Reactive System Specification: Rail Road Crossing Problem

We consider a *rail road crossing* instance of a reactive system in which event orderings are specified using temporal operators. In a general version of the problem, multiple trains are allowed to run in parallel while crossing gates and the communication between trains and controllers of the gates at the crossings have strict real-time constraints. But we consider a simple version in which time is abstracted by the "time moments (states)". Because in PTL only individual objects can be specified, we assume that there is one train which wishes to cross one gate which is monitored by one controller. When the train is approaching the gate, it informs the controller, which in turn instructs the gate at the crossing to close. The gate, upon receiving the instruction from the controller, closes before the train is in the crossing. When leaving the crossing, the train informs the controller, which instructs the gate to open. The gate, upon receiving the instruction from the controller, raises until it is fully open. The specification of this system should ensure the safety property "the gate remains closed while the train is in the crossing". The significant aspects of this problem are

- *reactivity*, whereby the controller interacts with the environmental objects train and controller,
- *concurrency*, whereby train actions and controller actions may overlap, and
- *asymmetry*, in the interaction pattern

Many specifications that satisfy the safety property can be given for this problem. Below one such specification is given, and an informal proof of the safety property is given. The events in the system are modeled as propositions, shown in the table below.

Train	Gate	Controller
tr_1: approaching	g_1: lowered	ct_1: receives "approaching" message from the train
tr_2: in the crossing	g_2: closed	ct_2: sends message "lower" to the gate
tr_3: is crossing	g_3: raised	ct_3: receives "exit" message from the train
tr_4: crossed the gate	g_4: open	ct_4: sends "raise" message to the gate

We need to specify the relationships among the events modeled as propositions. There exist many possible orderings of events which can characterize a safe system. Let us postulate the following behavior. We use the notation $\bigcirc^k$ to denote the kth next.

Behavior of Train

$$Spec_{train} : \Box \begin{bmatrix} tr_1 \Rightarrow \bigcirc^3 tr_2 \wedge \\ tr_2 \Rightarrow tr_3 \,\mathcal{U}\, tr_4 \wedge \\ tr_1 \Rightarrow \bigcirc^6 \Box tr_4 \end{bmatrix}$$

Behavior of Controller

$$Spec_{controller} : \Box \begin{bmatrix} ct_1 \Rightarrow \bigcirc ct_2 \wedge \\ ct_3 \Rightarrow \bigcirc ct_4 \end{bmatrix}$$

Behavior of Gate

$$Spec_{gate} : \Box \begin{bmatrix} g_1 \Rightarrow \bigcirc (g_2 \,\mathcal{U}\, g_3) \wedge \\ g_3 \Rightarrow \bigcirc g_4 \end{bmatrix}$$

Two types of interactions exist in the system: interaction between train and controller and interaction between controller and gate. We need to specify these interactions. The interaction specification $Spec_{train\text{-}controller}$ models the communication that takes "one unit of time", whereas the interaction specification $Spec_{controller\text{-}gate}$ models the communication that is "instantaneous". The rationale is that the gate object is "tightly coupled" to the controller and hence the message sent by the controller reaches the gate instantaneously, whereas the train object is quite independent from the controller object and hence the message sent by the train needs some time (>0) to reach the controller.

Behavior of Train–Controller Interaction The train informs the controller that it is approaching and this information is received at the “next moment” in time by the controller. The train also informs the controller that it has crossed the gate and this information is received at the “next moment” by the controller. This interaction specification is specified below.

$$Spec_{train\text{-}controller} : \Box \begin{bmatrix} tr_1 \Rightarrow \bigcirc ct_1 \\ tr_4 \Rightarrow \bigcirc ct_3 \end{bmatrix}$$

Behavior of Controller–Gate Interaction The controller informs the gate to close and this information is received instantaneously by the gate. The controller informs the gate to raise and this information is also received instantaneously by the gate. This interaction specification is specified below.

$$Spec_{controller\text{-}gate} : \Box \begin{bmatrix} ct_2 \Rightarrow g_1 \\ ct_4 \Rightarrow g_3 \end{bmatrix}$$

Initially *start* is true. At sometime in future the train is approaching. So the initial state specification is

$$Spec_{init} : start \Rightarrow \Diamond tr_1$$

Thus the full specification of the rail road crossing problem is obtained by “combining” the above specification units, which we write as

$$Spec_{railroad} \equiv Spec_{train} \wedge Spec_{controller} \wedge Spec_{gate} \wedge Spec_{train\text{-}controller} \wedge$$
$$Spec_{controller\text{-}gate} \wedge Spec_{init}$$

We need to verify in this specification the safety property “whenever the train is in the crossing the gate remains closed”. From the semantics of $start \Diamond tr_1$, we infer that there exists $i \geq 0$ such that

$$\langle \mathcal{M}, i \rangle \models tr_1.$$

Applying the formal semantics to specifications, as shown in the box below, we get a formal semantic model of the railroad specification. In the box below we see that whenever tr_3 is true, g_2 is true. This proves the safety property for the given specification.

Model for rail road specification

$$\langle M, i \rangle \models tr_1$$
$$\langle M, i+1 \rangle \models cr_1$$
$$\langle M, i+2 \rangle \models ct_2 \wedge g_1$$
$$\langle M, i+3 \rangle \models tr_2 \wedge tr_3 \wedge g_2$$
$$\langle M, j \rangle \models tr_3 \wedge g_2, \ j = i+5, i+6, i+6, i+7$$
$$\langle M, i+8 \rangle \models tr_4 \wedge ct_3 \wedge g_2$$
$$\langle M, i+9 \rangle \models tr_4 \wedge ct_4 \wedge g_3$$
$$\langle M, k \rangle \models tr_4 \wedge g_4, k > i+10$$

11.3.7.3 Refinement

A specification Spec'_A of component A is a *refinement* of Spec_A of component A if every model of Spec'_A is a *subset* of a model of Spec_A. Informally, a refinement reduces the number of possible models that satisfy a specification. Hence, successive refinements will lead to a fewer models and hence to a fewer implementations that are possible to verify. In PTL specification, any communication that involves $\diamond$ can be replaced by $\bigcirc$ operator to get a refinement. In particular, $\mathit{Spec}'_A : P \Rightarrow \bigcirc^4 Q$, $\mathit{Spec}''_A : P \Rightarrow \bigcirc^2 Q$, and $\mathit{Spec}'''_A : P \Rightarrow \bigcirc Q$ are some of the refinements of the specification $\mathit{Spec}_A : P \Rightarrow \diamond Q$. Depending upon the design goal, a suitable refinement must be chosen.

11.4 First Order Temporal Logic (FOTL)

The alphabet of PTL is extended to include a set of *predicate variables*, a set of function symbols, and a set of constants. In addition to the connectives in PTL, we add the universal quantifier $\forall$, and the existential quantifier $\exists$. We consider only a *partial* FOTL. A full account of FOTL can be found in [11, 17, 26].

The well-formed formulas in FOTL are defined as in Sect. 11.3.1, by letting predicates of arity ≥ 0 in wff_T and inductively defining the formulas in wff_T. Operator precedence remains the same as in PTL. Quantification over temporal formulas involves only one variable. As an example, we consider formulas such as $\forall x \bullet p(x) \Rightarrow \bigcirc q(x)$, and $\forall x \bullet (\mathit{head}(x) \geq 6) \wedge (\mathit{tail}(x) \neq \mathit{null})$. Such formulas are called *monadic*. In full FOTL, a formula can have quantification over more than one variable, even allowing quantification over arbitrary structures such as sets and trees.

The semantics of FOTL formulas involve "interpretation" as in predicate logic and temporal semantics as in PTL. The time structure is still $\mathbb{N}_0$ and we consider only global time. Thus a model for FOTL formulas is

$$\langle \mathbb{N}_0, \pi, I \rangle$$

where I is an interpretation which assigns for every formula element a value in a certain domain. Thus, for each time moment $i \in \mathbb{N}_0$, $\pi(i) = s_i$ is the state in which the formulas are to be evaluated using the interpretation I. Denoting the value of a variable x (expression e, predicate p, formula f) in a state s by $s[x]$ ($s[e], s[p], s[f]$), the evaluation steps for formulas, not involving temporal operators, in an interpretation I are described below:

1. Step 1—Evaluating Expressions
   (a) An expression e is evaluated in a state s by assigning values to all free variables and associating meaning to basic constructs. The value of the expression $e = 2x - 3y$ in a state s is $s[e] = 2s[x] - 3s[y]$.
2. Step 2—Evaluating Predicates

   (a) For the predicate $p(t_1,\ldots,t_n)$, where $t_1,\ldots,t_n$ are terms, define $s[p(t_1,\ldots,t_n)] = p(s[t_1],\ldots,s[t_n])$.
   (b) Predicate formulas
      (i) $s[\neg p] = \neg s[p]$
      (ii) $s[p \vee q] = s[p] \vee s[q]$
      (iii) $s[p \wedge q] = s[p] \wedge s[q]$
      (iv) $s[p \Rightarrow q] = s[p] \Rightarrow s[q]$
      (v) $s[p \Leftrightarrow q] = s[p] \Leftrightarrow s[q]$
3. Step 3—Evaluating Quantified Formulas
   (a) $s[\forall x \bullet p] = \forall x \bullet s[p]$.
   (b) $s[\exists x \bullet p] = \exists x \bullet s[p]$.

For example, consider the interpretation $I : \langle x = -1, y = 3, z = 1\rangle$ in state s for the formula $(x + y > z) \Rightarrow \bigcirc(y \leq 2 * z)$. Since $s[(x + y > z)$ is true in state s, in the next state it is asserted that $(y \leq 2 * z)$ must become true. That is, in the next state there must exist an interpretation that makes the formula $(y \leq 2 * z)$ true.

11.4.1 Formalizing Properties in FOTL

In Sect. 11.3.6, we discussed formalization of safety, liveness, and fairness properties in PTL. In this section, we illustrate through examples formalization of some properties using quantified FOTL formulas. Example 9 formalizes some properties of a bi-directional communication channel. Example 10 is a formal specification of safety and liveness requirements of a queue module that may be shared between different processors.

Example 9 Consider the problem of sending and receiving messages over a communication channel. Let $\{a, b\}$ denote the set of end-points of a channel, and $e \in \{a, b\}$. Let

$$\overline{e} = \begin{cases} a & \text{if } e = b \\ b & \text{if } e = a \end{cases}$$

Let M denote the set of messages and $m \in M$ be any arbitrary message transmitted over the channel $\langle a, b\rangle$. The temporal logic formulas, given below, involve atomic formulas such as $accept(m, e)$ and $deliver(m, e)$ and temporal operators. They are quantified over the set of messages $M' = M \setminus \{\text{DISCONNECT}\}$, considered as a set of constants.

1. To state that a channel is operational at all time, it is sufficient to state that both end-points are accepting messages all the time.

$$\exists m, m' \bullet \Box\, (channel\_on(a, b) \Leftrightarrow accept(m, a) \wedge accept(m', b)).$$

2. A channel cannot copy messages; if a message was delivered at some time, then it cannot be redelivered unless it was accepted again.

$$\forall m \bullet (deliver(m, \overline{e}) \Rightarrow \bigcirc\Box(\neg deliver(m, \overline{e})\, \mathcal{W}\, accept(m, e))).$$

3. A channel cannot accept two different messages at the same end-point at the same time:

$$\forall m, m' \bullet \Box\, (accept(m,e) \wedge accept(m',e) \Rightarrow m = m')$$

4. A message accepted at e will be delivered at $\overline{e}$, unless $\overline{e}$ has accepted a disconnect message at a preceding time.

$$\forall m \bullet \Box\, (accept(m,e) \Rightarrow$$
$$\Diamond(\neg((\neg accept(\text{DISCONNECT}, \overline{e})\, \mathcal{U}\, (deliver(m,\overline{e}))) \wedge$$
$$((\neg(deliver(m,\overline{e}))\, \mathcal{S}\, accept(\text{DISCONNECT}, \overline{e})))))$$

5. This axiom asserts the safety property that there can be no loss of messages in an active channel and that all messages accepted are eventually delivered. The messages are delivered at the end-point $\overline{e}$ in the same order in which they were accepted at the end-point e:

$$\forall m \bullet \Box\, (accept(m,e) \wedge \bigcirc\Diamond accept(m',e) \wedge$$
$$\neg(accept(\text{DISCONNECT}, \overline{e})\, \mathcal{U}\, deliver(m', \overline{e}))$$
$$\Rightarrow \Diamond((deliver(m,\overline{e}) \wedge \bigcirc\Diamond deliver(m',\overline{e})))$$

6. When a disconnect message is either accepted or delivered at one end, the channel stops functioning at that end.

$$((accept(\text{DISCONNECT}, e) \vee deliver(\text{DISCONNECT}, e)) \Rightarrow$$
$$\neg\exists m \bullet (\bigcirc\Box(\neg accept(m,e) \wedge \neg deliver(m,e)))) \qquad \Box$$

Example 10 is from Lamport [16]. It illustrates FOTL formalization of safety and liveness properties of a finite queue based on its states. The queue is a shared data type and hence PUT and GET operations may be initiated concurrently by more than one process. With the atomicity assumption and modeling concurrency as the interleaving of atomic operations, only one operation can occur on the queue at any specific time. That is, given the current contents of the queue, only one process can perform the PUT or GET operation on this state. A process uses GET to fetch a value at an instant; when the queue is empty, the process waits until another process puts a value in the queue.

Example 10 The capacity of the queue is max. The functions characterizing the queue states are defined below and the variables in the post-state are distinguished by suffixing them with a prime.

cur_queue:	the current state of the queue
putval:	argument to PUT. The precondition for PUT is $putval \neq$ nil and the post-condition is $putval' =$ nil.
getval:	argument to GET. The precondition for GET is $getval =$ nil and the post-condition is $getval' \neq$ nil.

enter(PUT), *enter*(GET), *exit*(GET) and *exit*(PUT) are boolean-valued functions signaling the initiation and termination of the operations.

Liveness Properties

1. The liveness property for the PUT operation is that it terminates. The element *putval* is inserted only if it does not cause an overflow; the symbol '$*$' denotes insertion at the rear.

   $enter(\text{PUT}) \wedge (length(cur\_queue) < \text{max}) \Rightarrow$

   $\Diamond(exit(\text{PUT}) \wedge (cur\_queue' = cur\_queue * putval))$

   $enter(\text{PUT}) \wedge (length(cur\_queue) = \text{max}) \Rightarrow$

   $\Diamond(exit(\text{PUT}) \wedge (cur\_queue' = cur\_queue))$

2. The liveness property for the GET operation is that it terminates only when a value is fetched from the queue. That is, if the queue is empty the operation waits until a value is put in the queue and then fetches the value.

   $enter(\text{GET}) \wedge \neg empty(cur\_queue) \Rightarrow$

   $\Diamond(exit(\text{GET}) \wedge (getval * cur\_queue' = cur\_queue))$

   $enter(\text{GET}) \wedge empty(cur\_queue) \Rightarrow enter(\text{GET})\, \mathcal{U}\, exit(\text{PUT})$

3. When the queue is empty, some process will eventually put a value in the queue.

   $empty(cur\_queue) \Rightarrow \Diamond enter(\text{PUT})$

Safety Properties
Safety properties assert what may or may not happen to the queue due to the actions PUT and GET. The state of the queue changes under the following situations:

1. $putval \neq nil$, and PUT is invoked by some process on a queue that is not full;
2. GET is invoked by some process, and *cur_queue* is not empty.

Situation 1. Let $(enter(\text{PUT}) \wedge putval \neq \text{nil})$ hold for *cur_queue*. The next state is cur_queue' which is the same as *cur_queue* when the queue is full or $cur\_queue' = cur\_queue * putval$. So, the temporal logic formula is

$$\Box((enter(\text{PUT}) \wedge putval \neq \text{nil}) \Rightarrow$$

$$(((length(cur\_queue) = \text{max}) \wedge (cur\_queue' = cur\_queue))$$

$$\vee((length(cur\_queue) < \text{max}) \wedge (cur\_queue' = cur\_queue * putval)))).$$

Situation 2. Let $(enter(\text{GET}) \wedge getval = \text{nil})$ hold for *cur_queue*. The next state is cur_queue' which is the same as *cur_queue* when the queue is empty, or $cur\_queue = getval * cur\_queue'$. So, the temporal logic formula is

$$\Box((enter(\text{GET}) \wedge getval = \text{nil}) \Rightarrow ((empty(cur\_queue) \wedge empty(cur\_queue'))$$

$\vee(\neg empty(cur\_queue) \wedge (cur\_queue = getval * cur\_queue')))).$ □

11.4.2 Temporal Logic Semantics of Sequential Programs

In this section, we consider the temporal logic semantics for simple sequential programs whose elements are the following:

- *Assignment Statement:* $x := e$
- *Composition of Statements:* $S_1; S_2$
- *Conditional Selection: if e then* S_1 *else* S_2
- *Repetition: while e do S*

Their formal meanings under Hoare logic have been discussed in Chap. 10. Below we give FOTL semantics to these constructs. The function [[——]] assigns for each program element a well-formed formula in FOTL.

Semantics of Assignment Statement Informally, this asserts that

- before the assignment is executed there exists some state s_i where an interpretation I_{s_i} exists,
- expression e is evaluated in the interpretation I_{s_i}, as $s_i[e]$ explained earlier, and
- the result of this evaluation is the result of interpretation $I_{s_{i+1}}$ applied to x, as $s_{i+1}[x]$ in state s_{i+1}.

The states s_i and s_{i+1} need not be shown explicitly; instead the $\bigcirc$ operator may be used.

Semantics—assignment statement

$$[[x := e]] = \exists I_{s_{i+1}} \bullet s_{i+1}[x] = s_i[e]$$

or

$$[[x := e]] = \bigcirc(x = e)$$

Semantics of Composition Informally it asserts that at current moment the first statement is evaluated and the moment next to the termination of the first statement, the second statement is evaluated.

Semantics—sequential composition of statements

$$[[S_1; S_2]] = [[S_1]] \wedge \bigcirc[[S_2]]$$

Semantics of Conditional Statement Let $I(e)$ denote the evaluation of test expression at state s_i, and assume that the evaluation does not take any time. If $I(s_i[e])$ is true then the expression $[[S_1; S_3]]$ is assigned a meaning in state s_i. If $I(s_i[e])$ is false then the expression $[[S_2; S_3]]$ is assigned a meaning in state s_i.

Semantics—conditional statement

$$[[(\textit{if } e \textit{ then } S_1 \textit{ else } S_2); S_3]] = (I(e) \Rightarrow [[S_1; S_3]]) \wedge (\neg I(e) \Rightarrow [[S_2; S_3]])$$

Example 11 We derive a temporal formula for capturing the semantics of the following program:

```
begin
x:=2;
if (x>2) then x:=x-2 else x:=x+1;
y:=1/x;
end
```

Let *Prog* denote the program. Then

$$[[Prog]] = \bigcirc((x = 2) \wedge [[if \ldots]])$$

$$[[Prog]] = \bigcirc((x = 2) \wedge ((x > 2) \Rightarrow [[x := x - 2]]) \wedge ((x \leq 2) \Rightarrow [[x := x + 1 \ldots]]))$$

$$[[Prog]] = \bigcirc((x = 2) \wedge [[(x := x + 1) \ldots]])$$

$$[[Prog]] = \bigcirc((x = 2) \wedge \bigcirc((x = 3) \wedge [[y := 1/x \ldots]]))$$

$$[[Prog]] = \bigcirc((x = 2) \wedge \bigcirc((x = 3) \wedge \bigcirc((y = 1/3) \wedge [[end]])))$$

$$[[Prog]] = \bigcirc((x = 2) \wedge \bigcirc((x = 3) \wedge \bigcirc((y = 1/3) \wedge true)))$$

$$[[Prog]] = \bigcirc((x = 2) \wedge \bigcirc((x = 3) \wedge \bigcirc((y = 1/3))))$$

$$[[Prog]] = \bigcirc(x = 2) \wedge \bigcirc\bigcirc(x = 3) \wedge \bigcirc\bigcirc\bigcirc(y = 1/3) \qquad \square$$

Semantics of Repetition Statement The semantics is given by recursively using the conditional statement axiom.

Semantic—while statement

$$[[(while\ e\ do\ S_1); S]] = (I(e) \Rightarrow [[(S_1; (while\ e\ do\ S_1))]]) \wedge (\neg I(e) \Rightarrow [[S]])$$

Example 12 We derive a temporal formula that gives the semantics of

$$x := 1;\ while\ (x < 3)\ do\ x := x + 1;\ end$$

$$[[x := 1;\ while\ (x < 3)\ do\ x := x + 1;\ end]]$$

$$= \bigcirc((x = 1) \wedge [[while\ \ldots]])$$

$$= \bigcirc((x = 1) \wedge ((x < 3) \Rightarrow [[x := x + 1;\ while\ \ldots]]) \wedge ((x \geq 3) \rightarrow [[end]]))$$

$$= \bigcirc((x = 1) \wedge [[x := x + 1; while\ \ldots]])$$

$$= \bigcirc((x = 1) \wedge \bigcirc((x = 2) \wedge [[while\ \ldots]]))$$

$$= \bigcirc((x = 1) \wedge \bigcirc((x = 2) \wedge ((x < 3) \Rightarrow$$
$$[[x := x + 1; while\ \ldots]]) \wedge ((x \geq 3) \rightarrow [[end]])))$$

$$= \bigcirc((x = 1) \wedge \bigcirc((x = 2) \wedge \bigcirc((x = 3) \wedge (x \geq 3) \rightarrow [[end]])))$$

$$= \bigcirc((x = 1) \wedge \bigcirc((x = 2) \wedge \bigcirc((x = 3) \wedge [[end]])))$$

$$= \bigcirc((x = 1) \wedge \bigcirc((x = 2) \wedge \bigcirc((x = 3) \wedge true)))$$

$$= \bigcirc(x = 1) \wedge \bigcirc\bigcirc(x = 2) \wedge \bigcirc\bigcirc\bigcirc(x = 3) \qquad \square$$

11.4.3 Temporal Logic Semantics of Concurrent Systems with Shared Variables

We add to the communication primitives discussed in Sect. 11.4.2 a new one, called *communication through shared variables*. We are still restricting to global discrete linear time model. When two components A and B share some variables, we must specify how such variables are to be accessed and modified by each component. It is necessary that each component sees exactly the *same* values for the shared variables at every state; otherwise there will be anomalies in the computation. We should allow components to see modifications to the variables at any state, and read from and write to these variables, subject to the following restrictions:

1. Read and write operations are *atomic*, in the sense that these operations take unit amount of time and cannot be interrupted.
2. Only one component can write at any moment in time.
3. When a component is reading from a variable (writing on it) the other component cannot write on it (read from it).

11.4.3.1 Component Specification

We restrict to a single variable x shared between components A and B and use the notations x_A and x_B to denote the same variable x in respective components. With this convention, the communication specification for the shared variable x between components A and B will include $\square(x_A \Leftrightarrow x_B)$. We specify restriction (3) as

$$\square\neg(\mathit{writing\_to}(A, x_A) \wedge \mathit{writing\_to}(B, x_B))$$

and include it in the communication specification. The program region where the shared variable is accessed is the critical region of the program. Hence, restriction (3) is also equivalent to $\square(\neg(\mathit{criti}_1 \wedge \mathit{criti}_2))$ (see Example 4). Thus, the communication specification for a single shared variable x is

$$\square(x_A \Leftrightarrow x_B) \wedge \square\neg(\mathit{writing\_to}(A, x_A) \wedge \mathit{writing\_to}(B, x_B))$$

The concurrent specification of components A and B interacting through a shared variable x, denoted $\mathit{Spec}_{A\|_x B}$, is

$$\begin{aligned} \mathit{Spec}_A \wedge \mathit{Spec}_B \wedge (\square(y_A \Leftrightarrow y_B) \\ \wedge \square\neg(\mathit{writing\_to}(A, y_A) \wedge \mathit{writing\_to}(B, y_B))) \end{aligned}$$

We illustrate a mixture of communication mechanisms in the following two examples.

Example 13 Consider components A and B which share a variable y. Their specifications are given below:

$$Spec_A : \Box \begin{bmatrix} start \Rightarrow (x = 5 \wedge y_A = 14) \wedge \\ even(y_A) \Rightarrow \bigcirc(y_A = (y_A - x) \wedge \\ writing\_to(A, y_A)) \end{bmatrix}$$

$$Spec_B : \Box \begin{bmatrix} start \Rightarrow (w = 9) \wedge \\ odd(y_B) \Rightarrow \bigcirc(y_B = y_B + w) \wedge \\ writing\_to(B, y_B)) \end{bmatrix}$$

Let us denote the concurrent specification of components A and B as $Spec_{A\|_y B}$. To understand the behavior of $Spec_{A\|_y B}$ we need to calculate the predicates that are true at time points starting from 0. From the specifications $Spec_A$ and $Spec_B$ and the shared variable principle, the predicate $(x = 5) \wedge (y_A = 14) \wedge (y_B = 14) \wedge (w = 9)$ is true at time 0. Hence, the predicate $even(y_A)$ is true at time 0. Consequently, the second statement in $Spec_A$ is to be executed. This makes the predicate $(x = 5) \wedge (y_A = 9) \wedge (y_B = 9) \wedge (w = 9)$ true in state 1. Notice that the values of x and w remain unchanged, although this is not explicitly stated in the specification. In state 1, the predicate $odd(y_B)$ is true and consequently the second statement of specification $Spec_B$ is executed. This makes the predicate $(x = 5) \wedge (y_A = 18) \wedge (y_B = 18) \wedge (w = 9)$ true in state 2. Continuing the analysis for successive states, we notice that in even numbered states the predicate $even(y_A)$ is true, and the second statement of $Spec_A$ is executed. As a consequence, the value of y_A is decreased by 5, an odd amount, which makes y_A (and y_B) odd at the next state. In odd numbered states the predicate $odd(y_B)$ is true, and the second statement of $Spec_B$ is executed. As a consequence, the value of y_B is increased by 9, an odd amount, which makes y_B (and (y_A) even at the next state. Consequently the behavior of $Spec_{A\|_y B}$ is infinite, only one of the components is active at any one time, and the values of x and w do not change. After some calculations, the model $\langle \mathbb{N}_0, \pi \rangle$ of the concurrent specification $Spec_{A\|_y B}$ can be succinctly determined as shown below.

$$\pi(i) = \begin{cases} (x = 5) \wedge w = 9 \wedge y = 14 + 4k) & \text{if } i = 2k, k \geq 0 \\ (x = 5) \wedge w = 9 \wedge y = 5 + 4k) & \text{if } i = 2k - 1, k \geq 1 \end{cases} \qquad \Box$$

Example 14 In this example, we simulate the behavior of a game played by two persons *Alice* and *Bob* with a slot machine M. Both *Alice* and *Bob* interact with a slot machine M; however, they do not interact between themselves. The time points of their interactions overlap; however, the slot machine responds to them only after receiving input from both of them. The slot machine responds to both of them simultaneously, according to the following rules.

- The slot machine does not accept any amount less than \$5.
- Both *Alice* and *Bob* specify amounts (> \$5) they want to bet.
- *Alice* bets at every moment in time.
- *Bob* bets at the beginning and at every second moment in time.

- The slot machine M waits until both bets are received, and then does the following:

  It determines the winner at the next moment (the moment after both bets are received), and sends an amount x subject to the constraint ($max - min \leq x \leq max$), where max and min, respectively, denote the maximum and minimum of the two amounts bet by *Alice* and *Bob*.

11.4.3.2 Specifications

$$Spec_{Alice} : \Box \left[\begin{array}{l} start \Rightarrow req\_game(Alice, n) \wedge \\ req\_game(Alice, n) \Rightarrow \bigcirc req\_game(Alice, n) \wedge \\ \Diamond got\_result(Alice, n') \end{array} \right]$$

$$Spec_{Bob} : \Box \left[\begin{array}{l} start \Rightarrow req\_game(Bob, m) \wedge \\ req\_game(Bob, m) \Rightarrow wait \wedge \\ wait \Rightarrow \bigcirc req\_game(Bob, m) \wedge \\ \Diamond got\_result(Bob, m') \end{array} \right]$$

$$Spec_{M} : \Box \left[\begin{array}{l} [rcv\_bet(Alice, \bar{n}) \wedge rcv\_bet(Bob, \bar{m})] \Rightarrow \\ \bigcirc [choose(x \in \{max(\bar{n}, \bar{m}) - min(\bar{n}, \bar{m}), \ldots, max(\bar{n}, \bar{m})\}) \Rightarrow \\ \bigcirc ((give\_result(Alice, x) \wedge give\_result(Bob, 0)) \vee \\ (give\_result(Bob, x) \wedge give\_result(Alice, 0)))] \end{array} \right]$$

The predicate $choose(x \in S)$, where S is a set, asserts that a value x is chosen from the set, but does not specify exactly which element of the set is chosen and how it is chosen. Such a predicate introduces *nondeterminism* in the design. In $Spec_M$ the second line uses the predicate *choose* to specify that a value in the range $[max - min, max]$ is determined nondeterministically. The third line in $Spec_M$ states that once x is determined then in the next moment there is a *choice* of announcing the winner. The choice operator $\vee$ is introduced in the right hand side of $\Rightarrow$. The communication specifications are given below.

Communication Specifications

$$Comms(Alice, M) : \Box(req\_game(Alice, n) \Rightarrow \Diamond rcv\_bet(Alice, \bar{n})) \wedge \Box(\bar{n} \Leftrightarrow n)$$
$$Comms(Bob, M) : \Box(req\_game(Bob, m) \Rightarrow \Diamond rcv\_bet(Bob, \bar{m})) \wedge \Box(\bar{m} \Leftrightarrow m)$$
$$Comms(M, Alice) : \Box(give\_result(Alice, \bar{m}) \Rightarrow \Diamond got\_result(Alice, n')) \wedge \Box(\bar{m} \Leftrightarrow n')$$
$$Comms(M, Bob) : \Box(give\_result(Bob, \bar{n}) \Rightarrow \Diamond got\_result(Bob, m')) \wedge \Box(\bar{n} \Leftrightarrow m')$$

The full specification is

$$Spec_{Alice} \wedge Spec_{Bob} \wedge Spec_{M} \wedge Comms(Alice, M) \wedge Comms(Bob, M) \wedge$$
$$Comms(M, Alice) \wedge Comms(M, Bob)$$ □

In Examples 13 and 14, the communication specification fixes the interaction point of the components. The programs corresponding to such components, assuming that they start at the same time and execute at the same speed, are expected to reach the specified interaction point at the same instant for "hand shaking". When programs that run at different speeds share variables and the interaction specification states only mutual exclusion principle, a more thorough analysis of the shared variables becomes necessary. That is, different interleaving executions must be analyzed. For a full account of the temporal semantics of concurrent programs consult Pnueli [23].

Example 15 Programs $Prog_1$ and $Prog_3$ share the variable x, programs $Prog_2$ and $Prog_3$ share the variable y, and programs $Prog_1$ and $Prog_2$ have no shared variable. Assume that the programs are started simultaneously at time 0. In these programs, the shared variables are accessed only at the statements P_1, P_2, and P_3 as shown below.

$$Prog_1 : \left[\begin{array}{c} \vdots \\ P_1 :: x := x + 2; \\ \vdots \end{array}\right]$$

$$Prog_2 : \left[\begin{array}{c} \vdots \\ P_2 :: y := y + 2; \\ \vdots \end{array}\right]$$

$$Prog_3 : \left[\begin{array}{c} \vdots \\ P_3 :: \begin{cases} x := x - y + 1; \\ x := x + y + 1; \end{cases} \\ \vdots \end{array}\right]$$

Using the temporal semantics (Sect. 11.4.2) for each program $Prog_i$, we create the temporal logic formulas $T(Prog_i)$. We may regard $T(Prog_i)$ as temporal specifications of the component whose implementation is $Prog_i$. In such a specification, we can ignore variables that are not accessed in the critical sections and identify $T(Prog_i)$ with $T(P_i)$. We can use the semantics discussed above for the specifications $T(P_1)$, $T(P_2)$, and $T(P_3)$ to provide a model of the concurrent execution of the program statements P_1, P_2, P_3. The program $P_1 \parallel P_2$ exhibits pure concurrency, for they have no shared variables. Hence the model of $P_1 \parallel P_2$ is the model of $T(P_1) \wedge T(P_2)$. The model for the concurrent program $P_1 \parallel_x P_3$ is calculated below from the model $T(P_1) \wedge T(P_3)$, subject to the interaction points. The temporal specifications of P_1 and P_3 are

$$T(P_1) = \exists u \in Val_1(i). \bigcirc (x = u + 2)$$

$$T(P_3) = \exists a, b \in Val_3(j). \bigcirc ((x' = a - b + 1) \wedge \bigcirc (x'' = x' + b + 1)),$$

where $Val_1(i)$ and $Val_3(j)$, respectively, denote the set of values of state variables of program statements P_1 and P_3 at times i and j. We need to consider two situations: $i + 1 < j$, which states that P_1 writes on variable x before P_3 commences execution, and $i > j + 1$, which states that P_3 completes writing on variable x before P_1 starts execution. In the former case the behavior is

state $i+1$	$(x=u+2)$
state j	$(x=u+2)$
state $j+1$	$(a=u+2) \wedge (x=u-b+3)$
state $j+2$	$(a=u+2) \wedge (x=u+4)$

In the latter case the behavior is

state j	$(x=a) \wedge (x'=a-b+1)$
state $j+1$	$(x'=a-b+1) \wedge (x''=a+2)$
state $i-1$	$(x=a+2)$
state i	$(x=a+4)$

□

11.5 Formal Verification

Formal verification for temporal logic is of two kinds, known as *axiomatic* and *algorithmic*. In an axiomatic approach, a set of general axioms and a proof calculus [17] are formulated. They form the kernel to which more domain specific axioms and inference rules may be added. These axioms and inference rules are then applied to show the correctness of temporal logic specification with respect to more abstract system specification. In an algorithmic approach, a model of the system is constructed and an algorithm is developed to prove that the model satisfies the property, stated as a temporal logic formula. These two approaches are separate systems of studies in itself, which we pursue only in a small measure. The goal in this section is to outline the basic characteristics of these methods for FOTL and PTL and illustrate with simple examples.

In Sect. 11.5.1, we illustrate a version of axiomatic verification, that closely follows Hoare style axiomatization, applied to a simple planning system specified in FOTL. In Sect. 11.5.2, we introduce a model of the system specified by PTL formulas and give the algorithmic approach for verifying properties in the model.

11.5.1 Verification of Simple FOTL Specifications

We restrict to a simple subsystem of FOTL which consists of quantified temporal formulas that involve only the temporal operators $\Box$, $\Diamond$, and $\bigcirc$. Since FOTL specifications are interpreted over linear states, in order to prove a property in the system it is sufficient to verify that the property holds at every system state. Let us write $\varphi(i)$ to mean that φ is true at the ith state of $\mathcal{M}$, the semantic model of the specification. If we can show

- $\varphi(0)$ is true, and

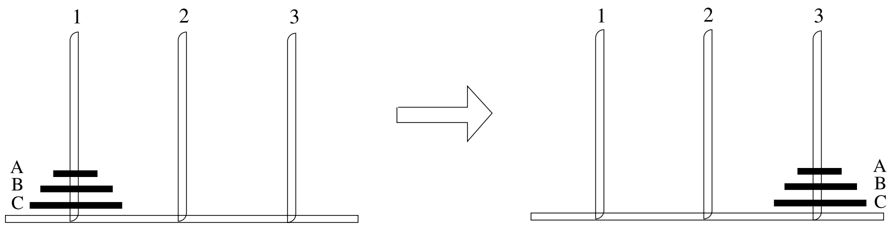


Fig. 11.1 Towers of Hanoi

- for any i establish $\varphi(i+1)$, knowing that $\varphi(i)$ is true,

then we know by induction Axiom (see Sect. 11.3.5) that $\varphi(i)$ is true for all $i \in \mathbb{N}_0$. It is this principle that we will use for formally verifying invariant properties, such as safety. We can recast the induction result using pre- and postconditions for actions at states. Let $a_1, a_2, \ldots$ be a sequence of actions performed at states $s_1, s_2, \ldots$. If P_i is a precondition and Q_i is the postcondition for action a_i at state s_i, and $\mathcal{T}(a_i)$ is the temporal logic specification of the system at state s_i, it is necessary to verify that $P_i \wedge \mathcal{T}(a_i) \Rightarrow Q_i$, and $Q_i \Rightarrow P_{i+1}$, where P_{i+1} is the precondition for action a_{i+1} at state s_{i+1}. This formulation is similar to the Hoare style axioms for verifying correctness of sequential programs discussed in Chap. 10.

Example 16 Planning problems require the construction of a series of actions such that the goal can be achieved by executing those actions in a certain order. Temporal logic can be used to specify actions and their ordering in planning problems. We consider the Tower of Hanoi problem to illustrate the methodology.

> Three pegs p_1, p_2, and p_3 and three disks A, B and C are available. Initially the three disks are stacked on peg p_1, with the smallest disk A on top and largest disk C at the bottom. It is required to plan a move of the disks from peg p_1 to peg p_3, using peg p_2 as a temporary storage area, such that after every single move, the disks are ordered with the largest one at the bottom.

A configuration refers to the collection of pegs and the ordering of disks on them. The initial and final configurations are shown in Fig. 11.1. We assume that each peg p has a "largest" disk D attached to its bottom. This disk cannot be moved. Denoting the size of a disk x by $size(x)$ we have the constraint

$$size(A) < size(B) < size(C) < size(D_p) = \infty$$

We abstract each peg with disks stacked in it in increasing order of size from its top to its bottom as a *stack* of linearly ordered items. All actions on a peg, according to the problem requirement, should satisfy this property. That is, "items from top to bottom in a stack are linearly ordered, in decreasing order" is an *invariant* property of the Hanoi system. Once this is recognized, we want to provide a temporal logic specification of a stack that satisfies this invariant property, and then use the stack operations to define the move actions for solving the Hanoi problem.

Stack Specification Let $(\mathcal{D}, \prec)$ be a linearly ordered domain. That is, for any two values v_1, v_2 in $\mathcal{D}$, either $v_1 \prec v_2$ or $v_2 \prec v_1$ can hold. A stack contains elements from $(\mathcal{D}, \prec)$, subject to the invariant property stated above. Assume that $\mathcal{D}$ has an infimum $-\infty$, and a supremum ∞. That is, $\forall v, v \in \mathcal{D}$ the property $-\infty < v < \infty$ holds. Stack operations should satisfy the invariant property. In the stack specification shown below, *curr_stack* denotes the current stack configuration.

Operation	*Specification*
TOP	argument is *curr_stack* it returns a value $v \in \mathcal{D}$. precondition is *true* let φ denote the predicate $v = \infty$ if *curr_stact* is empty $v = TOP(curr\_stack)$ if *curr_stack* is not empty in the postcondition $\varphi \wedge (curr\_stack' = curr\_stack)$
ADD	adds an element to the current stack. arguments to ADD are $addval \in \mathcal{D}$, and *curr_stack*. precondition is $addval \prec TOP(curr\_stack)$ postcondition is $TOP(curr\_stack') = addval \wedge$ $(curr\_stack' = addval * curr\_stack)$
REMOVE	removes the element from top of the stack argument is *curr_stack*. precondition is *curr_stack* is not empty. postcondition is $(TOP(curr\_stack) * curr\_stack' = curr\_stack)$

Proving Invariant Property Let $I(s)$ denote the invariant property of the stack: "items from top to bottom in a stack s are linearly ordered, in decreasing order". $I(empty)$ is trivially true. We need two basic axioms on ordered sequences to prove that stack operations preserve the invariant property $I(s)$. Let $\sigma = \langle v_1, v_2, \ldots, v_k \rangle$, $v_i \in \mathcal{D}$ be a strictly monotonic increasing sequence. That is

$$v_1 \prec v_2 \prec \cdots \prec v_{k-1} \prec v_k$$

If $v \prec v_1$ then the sequence $v * \sigma$, where $*$ denotes insertion at the front of the sequence, is also strictly monotonically increasing. This follows from the transitive property of $\prec$. If the element v_1 (in general, any v_i) is removed from the sequence, the rest of the sequence remains ordered. These properties are stated below for insertion and deletion of items in a stack of ordered items.

Ordered Stack—Insertion Axiom

$$\frac{I(s), x \prec TOP(s)}{I(x * s)}$$

Ordered Stack—Deletion Axiom

$$\frac{I(s), s = x * s'}{I(s')}$$

From the postcondition of *TOP* operation, infer $curr\_stack' = curr\_stack$. Hence, it follows that *TOP* operation preserves the invariant property:

$$I(curr\_stack') = I(curr\_stack)$$

When the precondition of the *ADD* operation

$$I(curr\_stack) \wedge (addval \prec TOP(curr\_stack))$$

is satisfied from the insertion axiom, it follows that $I(curr\_stack')$ holds after termination of *ADD* operation.

The precondition of *REMOVE* specification is *curr_stack* is not empty. So, we infer

$$curr\_stack = x * curr\_stack',$$

where $x = TOP(curr\_stack)$. Combining $I(curr\_stack)$ with this result, we derive

$$(curr\_stack = x * curr\_stack') \wedge I(curr\_stack),$$

and from the deletion axiom we derive $I(curr\_stack')$.

Hanoi Specification We give a temporal specification of actions. The basic action is a move, which is described using stack operations. Each move is atomic, in the sense that its execution cannot be interrupted and it takes one unit of time to complete it. The move action has parameters x and y which represent, respectively, the configurations of the peg from which the top disk is moved and the configuration of the destination peg in which the disk is placed. It is advantageous to have the notation $x = TOP(x) * rest\_of(x)$ for a nonempty configuration. If the configuration x is empty then $x = TOP(x) = D_x$.

Move Action:: move(x,y)

$$\langle v = TOP(x); \quad ADD(v, y); \quad REMOVE(x)\rangle$$

Using the stack specification, the meaning of action $move(x, y)$ can be formally written as

$$move(x, y) \Rightarrow \bigcirc((TOP(y') = TOP(x)) \wedge (rest\_of(y') = y) \wedge (x' = rest\_of(x))),$$

where x' and y' denote the post states of peg configurations x and y. For the sake of simplicity, we use a "horizontal notation" to display the status of a (stack) peg x, as in $u * w * D_x$, for which $TOP(x) = u$ as and $rest\_of(x) = w * D_x$.

1. *Initial Configuration:*
   Initially, disks A, B and C are in peg p_1, with A at the top, on top of disk B, disk B on top of disk C which is on top of the bottom disk D_{p_1}. The pegs p_2 and p_3 are empty.

Initial Configuration

$p_1 : A * B * C * D_{p_1}$ $\quad$ $I(p_1) \wedge I(p_2) \wedge I(p_3)$

$p_2 : D_{p_2} \quad p_3 : D_{p_3}$

2. *Goal Specification:*
The goal is to have disks A, B and C stacked on peg p_3 in the same order as in the initial state. The goal configuration is specified as follows:

Goal Configuration

$p_1 : D_{p_1} \quad p_2 : D_{p_2}$ $\quad$ $I(p_1) \wedge I(p_2) \wedge I(p_3)$

$p_3 : A * B * C * D_{p_3}$

3. *Planning:*
Starting with the initial configuration, the goal configuration is achieved by a sequence of moves as specified below.

$$move(p_1, p_3) \wedge \bigcirc move(p_1, p_2) \wedge \bigcirc^2 move(p_3, p_2) \wedge$$
$$\bigcirc^3 move(p_1, p_3) \wedge \bigcirc^4 move(p_2, p_1) \wedge \bigcirc^5 move(p_2, p_3) \wedge$$
$$\bigcirc^6 move(p_1, p_3)$$

4. *Verification:*
For each move operation, we have to (1) verify that the precondition is true, (2) calculate the postcondition, and (3) prove that the invariant is preserved. We should also prove that when all the moves are completed, the goal configuration is reached.

Verification for the first move: $move(p_1, p_3)$.

We substitute $[p_1/x, p_3/y]$ in $move(x, y)$ definition.

First Move Action:: $move(p_1, p_3)$

$\langle v = TOP(p_1); \quad ADD(v, p_3); \quad REMOVE(p_1)\rangle$

Initially, p_1 is not empty. From the postcondition of *TOP* operation, infer $v = TOP(p_1) = A$. Since $TOP(p_3) = D_{p_3}$, and $size(A) < size(D_{p_3})$ is true, the precondition of the operation $ADD(v, p_3)$ is true. From the postcondition of $ADD(v, p_3)$ we infer $TOP(p_3') = v \wedge p_3' = v * p_3$. By atomicity assumption, we have the result

$$\bigcirc((TOP(p_3') = A) \wedge (rest\_of(p_3') = p_3) \wedge (p_1' = rest\_of(p_1)))$$

Notice that we could have obtained the above result directly from the temporal specification of $move(x, y)$. Applying deletion axiom for $REMOVE(p_1)$ we get $I(p_1')$, and

Table 11.1 Hanoi Configurations

Time	Action	Configuration	Time	Action	Configuration
0	$move(p_1, p_3)$	$p_1 : A * B * C * D_{p_1}$	4	$move(p_2, p_1)$	$p_1 : D_{p_1}$
		$p_2 : D_{p_2}$			$p_2 : A * B * D_{p_2}$
		$p_3 : D_{p_3}$			$p_3 : C * D_{p_3}$
1	$move(p_1, p_2)$	$p_1 : B * C * D_{p_1}$	5	$move(p_2, p_3)$	$p_1 : A * D_{p_1}$
		$p_2 : D_{p_2}$			$p_2 : B * D_{p_2}$
		$p_3 : A * D_{p_3}$			$p_3 : C * D_{p_3}$
2	$move(p_3, p_2)$	$p_1 : C * D_{p_1}$	6	$move(p_1, p_3)$	$p_1 : A * D_{p_1}$
		$p_2 : B * D_{p_2}$			$p_2 : D_{p_2}$
		$p_3 : A * D_{p_3}$			$p_3 : B * C * D_{p_3}$
3	$move(p_1, p_3)$	$p_1 : C * D_{p_1}$	7		$p_1 : D_{p_1}$
		$p_2 : A * B * D_{p_2}$			$p_2 : D_{p_2}$
		$p_3 : D_{p_3}$			$p_3 : A * B * C * D_{p_3}$

applying insertion axiom for $ADD(v, p_3)$ we get $I(p_3')$. Since the move operation does not change the state of p_2, $I(p_2') = I(p_2)$. Thus, we have calculated the configuration at time moment 1, and proved that the first move operation preserves the invariance property $I(p_1) \wedge I(p_2) \wedge I(p_3)$.

Starting at the configuration at time 1 and following the above steps for the operation $move(p_1, p_2)$, we will arrive at the second configuration and will prove the invariant property $I(p_1) \wedge I(p_2) \wedge I(p_3)$. Calculations at successive time points are shown in Table 11.1. □

11.5.2 Model Checking

Model checking is an *algorithmic* technique for verifying finite state concurrent systems. Model checking method has successfully been applied to verify hardware designs, communication protocols, and reactive system properties. It is a rich field of study in which theory blends with practical techniques to verify safety and liveness properties of complex systems. Because model checking can be *automated*, it may be preferred to axiomatic verification method. However, model checking has some limitations as well. In this section, we discuss some basic principles of model checking with PTL and FOTL formulas.

The first step toward model checking is to create a formal model of the system and specify the property to be verified in the model. The formal model that is used for reactive systems is a state transition graph, called *Kripke structure* (KS). A KS resembles a state machine that we have seen in Chaps. 6 and 7, yet they are more general. Essentially, a KS has a set of states, a set of initial states, a set of transitions between states, and for each state a set of properties that are true in that state. Every state must have a transition. There is no accepting state. The language used to describe the properties in each state and label the transitions go toward defining the richness of the KS. In reactive and concurrent system models, input may be received at more than one state and the program need not terminate. This is why in KS there is no accepting state and more than one input state is allowed. In our study, we will restrict to PTL and simple FOTL formulas for stating the properties and labeling the transitions. Example 17 informally explains the construction of KS for a set of FOTL formulas. We use the notation x' to denote the variable x in a post state.

Example 17 We want to formally represent the computation of the program P

$$\langle x' = (x + y)\ mod\ 3;\ y' = (y + 1)\ mod\ 3\rangle$$

over the domain $V = \{0, 1, 2\}$ as a Kripke structure. The two program statements are to be executed as an atomic unit. A state represents "variable-value" pairs. That is, $\langle x = v_1, y = v_2\rangle$, where $v_1, v_2 \in V$ is a state, which is written (v_1, v_2). We associate the formula $x = v_1 \wedge y = v_2$ with the state (v_1, v_2). Thus the set of all states for the given variables over the domain V is

$$V \times V = \{(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)\}.$$

Let $(x = 0 \wedge y = 2)$ be the initial state. Program P that defines the relationship between (x, y) and (x', y') is the *transition* relation for the KS. That is, executing the program at the initial state gives the next state $x' = 2 \wedge y' = 0$. So, in KS, there is a transition from state $(x = 0 \wedge y = 2)$ to $(x = 2 \wedge y = 0)$. The transition relation for the KS is the set of ordered pairs of vertices related by the execution of P. So, the KS that models the computation P is

- *set of states:* $S = V \times V$
- *set of initial states:* $S_0 = \{(0, 2)\}$
- *transition relation:*

$$R = \{((0, 2), (2, 0)), ((2, 0), (2, 1)), ((2, 1)(0, 2)), ((0, 0), (0, 1)), ((0, 1), (1, 2)),$$
$$((1, 2), (0, 0)), ((1, 1), (2, 2)), ((2, 2), (0, 0)), ((0, 0), (1, 1))\}$$

- *properties:*

$$\begin{array}{l|l|}
L((0, 0)) = \{x = 0 \wedge y = 0\} & L((0, 1)) = \{x = 0 \wedge y = 1\} \\
L((1, 0)) = \{x = 1 \wedge y = 0\} & L((1, 1)) = \{x = 1 \wedge y = 1\} \\
L((2, 0)) = \{x = 2 \wedge y = 0\} & L((2, 1)) = \{x = 2 \wedge y = 1\}
\end{array}$$

$$\begin{array}{l}
L((0, 2)) = \{x = 0 \wedge y = 2\} \\
L((1, 2)) = \{x = 1 \wedge y = 2\} \\
L((2, 2)) = \{x = 2 \wedge y = 2\}
\end{array}$$

□

Formally a Kripke structure over a set of atomic propositions AP is a four tuple $KS = (S, S_0, R, L)$ where

1. S is a *finite* set of states.
2. $S_0 \subset S$ is the set of initial states.
3. R is a *total* transition relation on S, that is, for every $s \in S$ there is an s' such that $R(s, s')$.
4. L associates with each state s, a set $L(s) \subset AP$ of propositions that are true in that state.

A path in the structure KS from state s is an infinite sequence of states $\langle s_0, s_1, s_2, \ldots \rangle$, where $s = s_0$, and $R(s_i, s_{i+1})$ holds for all $i \geq 0$. Every state in KS is reachable from the initial state. In Example 17, the only path that starts in the initial state $(0, 2)$ is $(0, 2)(2, 0)(2, 1)$.

11.5.3 Program Graphs, Transition Systems, and Kripke Structures

In computing literature, especially in the field of formal modeling and verification, many different, but related, models are constructed. Program graph (PG) is a directed graph model constructed from a program description [1]. The vertices are program states, and a directed edge formalizes state change under a constraint. The semantics of state transitions is the "guard/action" paradigm that was introduced for EFSMs in Chap. 7. So, for all practical purposes, a PG is a EFSM in which typed variables appear as guards for transitions. In PG, the program statements are the interpretation functions which define evaluation of variables in a state. The semantics of the transition $s \xrightarrow{[g/\alpha]} s'$ is that a nondeterministic choice is made at state s to select a transition that satisfies g, the action α is executed according to the evaluation defined by I, and the transition results in state s'. Program graphs can be combined exactly the same way that EFSMs are combined, except that for concurrent programs the proper semantics of concurrency must be followed to combine their respective graphs. Program graphs of concurrent programs with shared variables must be combined using the *interleaving* semantics. That is,

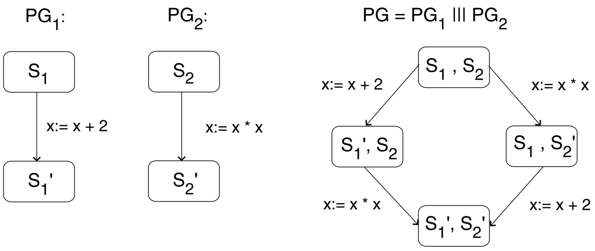
$$\frac{s_1 \xrightarrow{[g/\alpha]} s_1'}{(s_1, s_2) \xrightarrow{[g/\alpha]} (s_1', s_2)}$$

$$\frac{s_2 \xrightarrow{[g/\alpha]} s_2'}{(s_1, s_2) \xrightarrow{[g/\alpha]} (s_1, s_2')}$$

and the interpretation combines the interpretations of the two programs. We denote the interleaving program graph of PG_1 and PG_2 by $PG_1|||PG_2$. Example 18 illustrates this principle.

Example 18 The program graphs PG_1 and PG_2 corresponding to the program statements $P_1 : x := x + 2$, and $P_2 : x := x * x$ are shown in Fig. 11.2. The programs P_1 and P_2 have the shared variable x. Interleaving semantics apply for shared variables. Applying the semantics for interleaving we get the graph $PG : PG_1|||PG_2$ shown in Fig. 11.2. □

Fig. 11.2 Program graphs and their product: Example 18



A transition system (TS) over a set of propositions *AP* is almost identical to the Kripke structure (KS) over *AP*, the only exception is that a TS may be *infinite*. The states of TS are called *locations*, and the state transitions are caused by atomic actions. In an infinite TS, the set of locations S, the set of propositions *AP*, and the set of actions are infinite. The semantics of a transition from location s to location s', written $s \xrightarrow{\alpha} s'$, is that a transition from state s is selected *nondeterministically*, the action α is performed, and the location s' is reached. The outcome of this selection process cannot be predicted a priori. Similarly, if the TS has more than one initial state, the start state is selected nondeterministically.

Because a finite TS is a KS, every location of TS is a tuple containing evaluations of variables. In general, a location of TS is of the form $\langle s, p_1, p_2, \ldots, p_k, \rangle$, where p_i are propositions (or simple predicates of the form $b = 1$) that are true in that location, and s is a local name for the location. Comparing the semantics of PG and TS, it is evident that the state transition semantics can be given to transitions in PG and get the transition system TS_{PG} of a program graph PG, as described below:

- For each location s of PG create the locations $\{\langle s, (v_i, a_i)\rangle \mid v_i \in V\}$ of TS_{PG}, where a_i is the evaluation of a variable v_i in location s.
- Corresponding to a transition $s \xrightarrow{[g/a]} s'$ in PG, g must satisfy the evaluation of variables at s and a must satisfy the evaluation of variables at s'. Let $p_1, p_2, \ldots, p_k$ denote the evaluations that satisfy g at s, and $q_1, q_2, \ldots, q_k$ denote the result of applying the action α on $p_1, p_2, \ldots, p_k$, the transition rule becomes

$$\frac{s \xrightarrow{[g/\alpha]} s'}{\langle s, p_1, p_2, \ldots, p_k\rangle \xrightarrow{\alpha} \langle s_1', q_1, q_2, \ldots, q_k\rangle}$$

The transition systems TS_{PG_1} TS_{PG_2}, and $TS_{PG_1|||PG_2}$ of the program graphs constructed in Example 18 are shown in Fig. 11.3.

Notice that

$$TS_{PG_1}|||TS_{PG_1} \neq TS_{PG_1|||PG_2}$$

The reason is that the interleaving product of transition systems assume that the programs are truly concurrent, and share no variable. Therefore, in model checking systems with shared variables, we should compute the interleaving program graphs from the programs

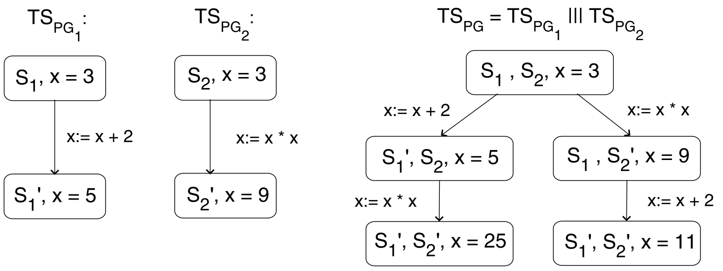
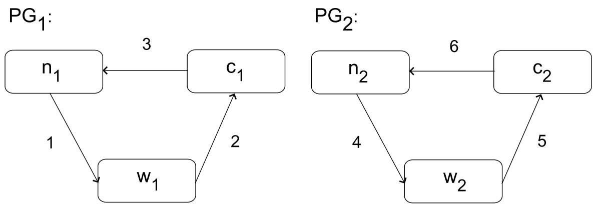


Fig. 11.3 Transition graphs corresponding to the program graphs in Fig. 11.2

Fig. 11.4 Program graphs for the mutual exclusion programs in Example 19



and then create its transition system. With this background we now look at the *mutual exclusion* algorithm of Peterson [21].

In Example 19, processes P_1 and P_2 share the three variables x of type $\{1,2\}$, and b_1 and b_2 of type Boolean. In order to enter the critical section, a process waits, gets the token, enters the critical section and performs its action, releases the token and exits into noncritical section. The main interest is on the shared variables in order to prove the safety property, as stated in Example 4, Sect. 11.3.6. So we abstract away from the computations in noncritical sections and focus only on the conditions for performing the action in the critical section. The situations are abstractly denoted with states containing the propositions c_i, n_i, w_i, as explained below:

c_i : process P_i is in the critical section
n_i : process P_i is not in the critical section
w_i : process P_i is waiting to enter the critical section

Thus, for the program graph PG_i of process P_i, the locations are c_i, n_i, w_i, and the set of variables is $V_i = \{x, b_1, b_2\}$, $i = 1, 2$. The variables b_i indicate the current location of process P_i. That is, $b_i = w_i \vee c_i$, and is set when P_i wants to wait. Initially we assume $b_1 = b_2 = \mathit{false}$. If $x = i$ then process P_i may enter its critical section. The program-graphs generated from the program descriptions of P_1 and P_2 in Example 19 are shown in Fig. 11.4.

Example 19

```
P_1:
loop forever
(*noncritical actions*)
⋮
(*request to enter critical section*)
⟨b_1 := true; x := 2⟩
(*waiting*)
wait until (x = 1 ∨ ¬b_2)
(*(action in critical section*)
 do critical section od 
(*release*)
b_1 := false;
(*noncritical section*)
⋮
end loop
```

```
P_2:
loop forever
(*noncritical actions*)
⋮
(*request to enter critical section*)
⟨b_2 := true; x := 1⟩
(*waiting*)
wait until (x = 2 ∨ ¬b_1)
(*(action in critical section*)
 do critical section od 
(*release*)
b_2 := false;
(*noncritical section*)
⋮
end loop
```

The specification of the program graphs PG_1 and PG_2, in the notation of EFSM, are given below.

$$PG_i = \{Q_i, \Sigma_i, V_i, \Lambda_i\}, i = 1, 2$$

where

$Q_i = \{c_i, n_i, w_i\}$
$\Sigma_i = \{\}$
$V_i = \{x, b_1, b_2\}, x \in \{1, 2\}, b_1, b_2 \in \{true, false\}$
Λ_1 : *Transition Specifications for* PG_1

1. $n_1 \xrightarrow{[\text{true}/(b_1=\text{true}\wedge x=2)]} w_1$
2. $w_1 \xrightarrow{[(\neg b_2 \vee x=1)/...]} c_1$
3. $c_1 \xrightarrow{[\text{true}/(b_1=\text{false})]} n_1$

Λ_2 : *Transition Specifications for* PG_2

4. $n_2 \xrightarrow{[\text{true}/(b_2=\text{true}\wedge x=1)]} w_2$
5. $w_2 \xrightarrow{[(\neg b_1 \vee x=2)/...]} c_2$
6. $c_2 \xrightarrow{[\text{true}/(b_2=\text{false})]} n_2$

Figure 11.5 shows the program graph PG which is the product of the program graphs PG_1 and PG_2 given in Fig. 11.4. The graph PG is constructed using the interleaving semantics on PG_1 and PG_2. In this graph, we find that the only ways to reach the state $\langle c_1, c_2\rangle$ are to try transitions from states $\langle w_1, c_2\rangle$ and $\langle c_1, w_2\rangle$. However, in state $\langle c_1, w_2\rangle$, $x = 1 \wedge b_2 = true$ is true, whereas the guard condition for transition (labeled 5) at that state is $\neg b_1 \wedge x = 2$, which is false. Similarly, in state $\langle c_2, w_1\rangle$, $x = 2 \wedge b_1 = true$ is true, whereas the guard condition for transition (labeled 2) at that state is $\neg b_2 \wedge x = 1$, which is false.

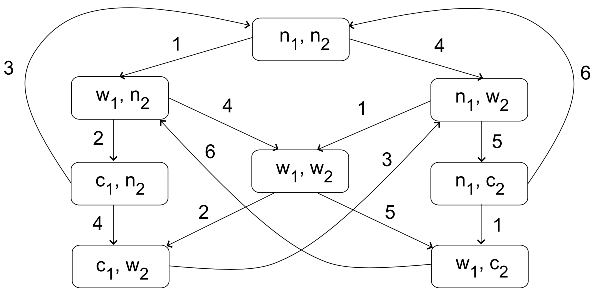


Fig. 11.5 Product of program graphs for the mutual exclusion programs in Example 19

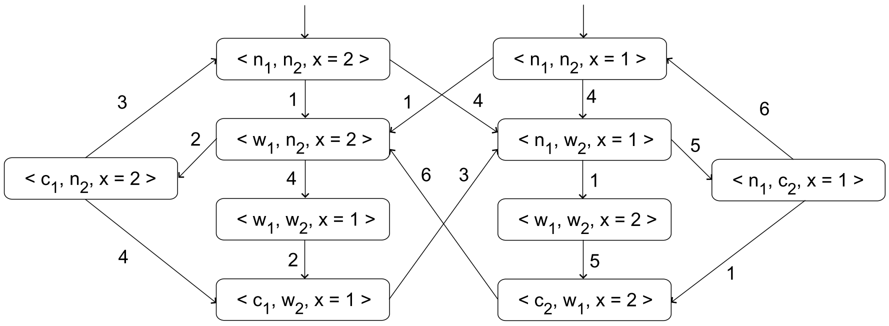


Fig. 11.6 Transition System for the mutual exclusion programs in Example 19

Consequently the state $\langle c_1, c_2\rangle$ is not reachable. That is, in all states of the program graph *PG* the property $\neg(c_1 \wedge c_2)$ is true. This proves the safety property of Peterson's mutual exclusion algorithm. □

The transition system for the program graph in Fig. 11.5 is shown in Fig. 11.6. Notice that the state (n_1, n_2) of the program graph corresponds to the two states $\langle n_1, n_2, x = 1\rangle$ and $\langle n_1, n_2, x = 2\rangle$ of the transition system. This is due to the reason that in the noncritical section n_1, the variable x is set to 2 and in the noncritical section n_2, the variable x is set to 1, and consequently in the combined state (n_1, n_2), the variable x can take either value. The transition specifications for both figures are identical.

In summary, a KS is a TS, a PG can be transformed to a TS, and a EFSM is a TS. That is, a TS, also called *labeled transition systems* (LTS), is the most general formal model for describing concurrent and reactive systems. When we start with a program, we construct its PG and then derive its TS. When we start with a model, say EFSM, then we may regard it as a PG and then derive the corresponding TS. Since we are dealing only with finite state concurrent and reactive systems, we will be dealing only with finite transition systems, which are Kripke structures.

A model checking algorithm is a *decision procedure*, which for a given model and a property should output "YES", if the model satisfies the property or output "NO" if the model fails to satisfy the property. When a Kripke structure is the formal model, we need to include "accept" states in it in order that the algorithm may recognize the acceptance

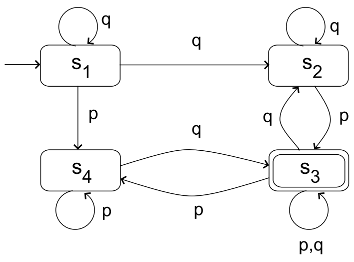


Fig. 11.7 Büchi automaton: Example 20

of the stated property. Moreover, the property itself may be stated as "something never happens", which requires a verification for infinite number of time points. The structure, known as Büchi Automaton (BA) [3], has the following attractive properties necessary for model checking.

- A Kripke structure can be transformed into a BA by simply making every state in KS an accepting state, and adding a new initial state.
- A BA is a finite state automaton extended to accept/reject infinite strings. Consequently, PTL formulas which are infinite strings can be recognized by a BA.
- PTL properties can be turned into Büchi automata.
- It is *easier* (linear time) to check if a BA accepts any string at all.
- It is known that Büchi automata are closed under union, intersection and complementation. This means that there exists an automaton that recognizes exactly the complement of a given language, and an automaton that recognizes the intersection of two automata. An equivalent statement is that the class of Büchi recognizable languages is closed under boolean operations.

11.5.4 Model Checking using Büchi Automata

A Büchi automaton is a finite state machine $\mathcal{A} = (Q, \Sigma, q_0, F, \delta)$ which accepts *infinite* strings. The notation $(ab)^\omega$ denotes the infinite repetition of the finite string ab, and the notation $(ab)^*$ denotes any arbitrary, but finite, repetitions of string ab. A *run* σ is an infinite sequence of states $s_0, s_1, \dots, s_n, \dots$, where s_0 is an initial state, and (s_i, s_{i+1}) is a transition in the BA. For a run σ, let $\mathit{Inf}(\sigma)$ denote the set of states that occur infinitely often in σ. The run σ is *accepted* by the BA if $\mathit{Inf}(\sigma) \cap F \neq \emptyset$, where F is the set of accept states of the BA. That is, some accept state must be *visited* infinitely often in the run σ.

Example 20 The automaton in Fig. 11.7 is a BA which has one accept state s_3. Example runs are s_1^ω, $s_1^* s_2 s_3^\omega$, and $s_1^* s_4^* s_3^\omega$. For the run $\sigma_1 = s_1^* s_2 s_3^\omega$, $\mathit{Inf}(\sigma_1) = \{s_3\}$ and hence the run σ_1 is an accepting run. For $\sigma_2 = s_1^* s_4^* s_3^* s_2^\omega$, $\mathit{Inf}(\sigma_2) = \{s_2\}$, and $\mathit{Inf}(\sigma_2) \cap \{s_3\} = \emptyset$. Hence the run σ_2 is not an accepting run. The string that corresponds to the accepting run $s_1^* s_2 s_3^\omega$ is $q^* q p^\omega$. It is easy to verify that the automaton accepts the strings in which both p and q appear infinitely often. □

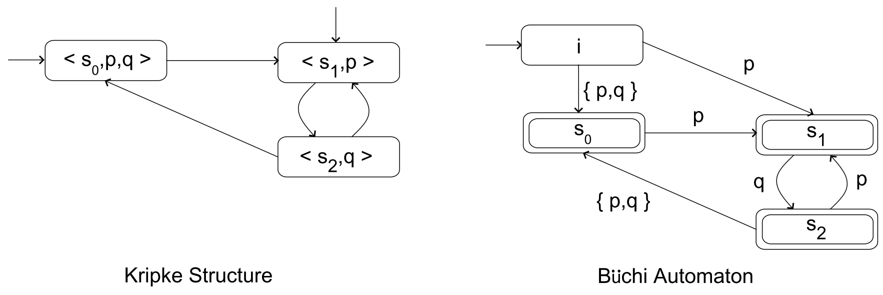


Fig. 11.8 Büchi automaton corresponding to a Kripke structure

The language $\mathcal{L}(\mathcal{A})$ is the set of strings accepted by $\mathcal{A}$. Every PTL formula is accepted by a Büchi Automaton. Consequently, a PTL formula can be expressed as a Büchi automaton. However, the converse is not true, because Büchi automaton can capture properties not expressible by PTL.

A Kripke structure (S, R, S_0, L), where $L : S \to 2^{AP}$, can be transformed to a Büchi automaton $(Q, \Sigma, q_0, F, \delta)$ as follows:

states:	$Q = S \cup \{i\}$, (all states of the Kripke structure plus one new state are the states of the automaton)
alphabet:	$\Sigma = 2^{AP}$ (alphabet is the set of propositions used in the labelings of the transition system)
initial states:	$q_0 = \{i\}$ (initial state is the new state created in Q)
accept states:	$F = S$ (all states of the Kripke structure become final states of the automaton)
transitions:	$\delta(s, \alpha) = s'$, $s, s' \in Q$ if and only if (s, s') is a transition in R and $L(s') = \alpha$

Figure 11.8 gives a Kripke structure and its corresponding Büchi automaton. The property to be verified can also be given as Büchi automata. Figure 11.9 shows the basic PTL formulas and their corresponding BAs. To reduce the number of transitions in a BA representation, we annotate the transitions as in Fig. 11.8. We can use boolean expressions, rather than a subset of propositions for annotation. Each transition annotated in this manner actually represents several transitions, where each transition corresponds to a truth assignment for AP that satisfies the boolean expression. As an example, when $AP = \{p, q, r\}$, a transition labeled $p \wedge \neg r$ matches labeled with $\{p, q\}$ and $\neg r$.

Model Checking Procedure Model checking is an algorithmic way of determining whether or not a system satisfies a stated property. There are several model checking procedures [6]. One of them, called *automata theoretic method*, is based on the following observation. A system modeled as Büchi automata $\mathcal{S}$ satisfies the property specified as another Büchi automata $\mathcal{P}$ when

$$\mathcal{L}(\mathcal{S}) \subset \mathcal{L}(\mathcal{P}) \tag{11.1}$$

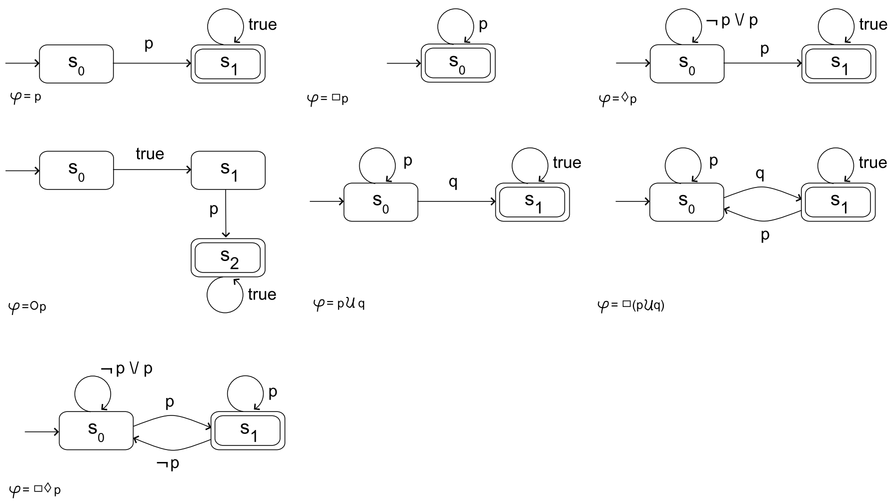


Fig. 11.9 Büchi automata corresponding to PTL formulas

That is, the behavior of the modeled system is contained within the behaviors allowed by the property. The relation (11.1) can be rewritten as

$$\mathcal{L}(\mathcal{S}) \cap \overline{\mathcal{L}(\mathcal{P})} = \emptyset \tag{11.2}$$

This means that any behavior disallowed by the property is not contained within the behaviors of the system. If the intersection in (11.2) is not empty, any behavior in it is a counterexample to the claim.

Computing Product of Büchi Automata The method for computing the intersection of finite automata does not work for computing the product of Büchi automata. Finite automata accept only finite words, whereas Büchi automata accept infinite words. An infinite word that is accepted by the product of two Büchi automata should visit the accept states of each automaton infinitely often. It is therefore necessary to record in every state of the product machine a *tag* indicating whether the product automaton is checking for an accept state of the first automaton or the second automaton. The product automaton accepts the string only if it switches the focus (as indicated by the tags) from the second to the first (or equivalently, from the first to second) infinitely often. That is, the accept states of the product automaton are precisely those where "switching back and forth" happens. The formal construction is given below.

The product of two Büchi automata $\mathcal{A}_i = (Q_i, \Sigma, q_{0i}, F_i, \delta_i)$, $i = 1, 2$, is $\mathcal{A} = (Q, \Sigma, q_0, F, \delta)$, where

- *states:* $Q = Q_1 \times Q_2 \times \{1, 2\}$
- *initial states:* $q_0 = \{(s_1, s_2, 1) \mid s_1 \in q_{01}, s_2 \in q_{02}\}$

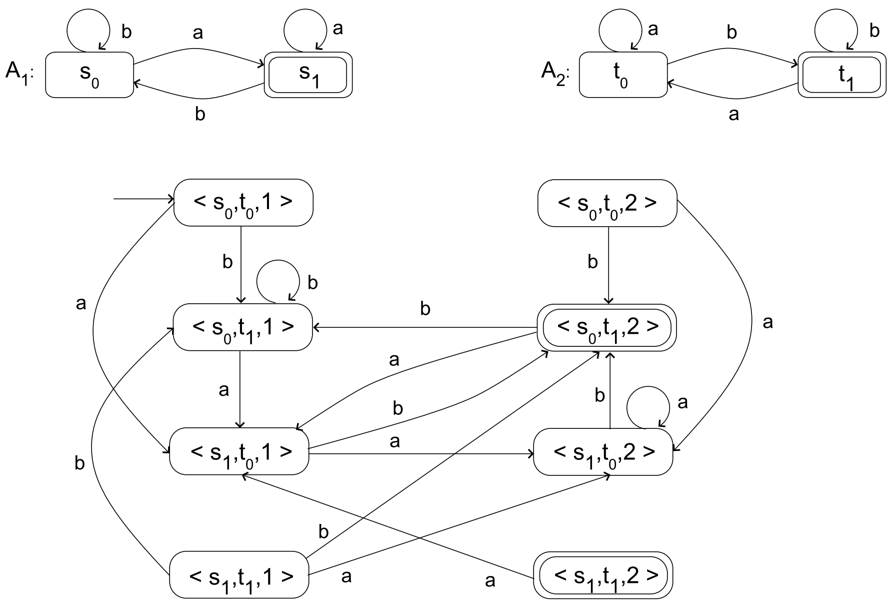


Fig. 11.10 Product of two Büchi automata

- *transitions:* For $a \in \Sigma$,

$$(s_1, s_2, 1) \xrightarrow{a} (s_1', s_2', 1) \text{ if } s_1 \xrightarrow{a} s_1', s_2 \xrightarrow{a} s_2', \text{and } s_1 \notin F_1.$$
$$(s_1, s_2, 1) \xrightarrow{a} (s_1', s_2', 2) \text{ if } s_1 \xrightarrow{a} s_1', s_2 \xrightarrow{a} s_2', \text{and } s_1 \in F_1.$$
$$(s_1, s_2, 2) \xrightarrow{a} (s_1', s_2', 2) \text{ if } s_1 \xrightarrow{a} s_1', s_2 \xrightarrow{a} s_2', \text{and } s_2 \notin F_2.$$
$$(s_1, s_2, 2) \xrightarrow{a} (s_1', s_2', 1) \text{ if } s_1 \xrightarrow{a} s_1', s_2 \xrightarrow{a} s_2', \text{and } s_2 \in F_2.$$

- *accept states:* $F = Q_1 \times F_2 \times \{2\}$ (or $F_1 \times Q_2 \times \{1\}$)

We write $A_1 \bigotimes A_2$ to denote the product automaton of the Büchi automata A_1 and A_2. Figure 11.10 shows two Büchi automata and their product.

When we transform a Kripke structure to a Büchi automaton, we make every state of the Büchi automaton an accepting state. Therefore, in this approach we ignore the tags. In general, tags are necessary for computing the product of any two Büchi automata. When we use boolean expressions to label the transitions of the automata, the transition relation is to be modified to reflect the equivalence of boolean expression for simultaneous transitions. That is, for the transitions $s_1 \xrightarrow{\alpha_1} s_1'$, and $s_2 \xrightarrow{\alpha_2} s_2'$, the simultaneous transition $(s_1, s_2) \xrightarrow{\alpha} (s_1', s_2')$ exists in the product automata if and only if $\alpha_1 \equiv \alpha_2 \equiv \alpha$.

Based on our discussion so far, the model checking procedure using Büchi automata is as follows:

1. Construct the Büchi automaton $\mathcal{A}_{sys}$ for the system.
2. Construct the Büchi automaton $\mathcal{A}_{\neg\varphi}$ for negation of the property φ to be verified.

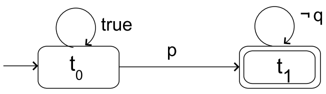


Fig. 11.11 Büchi automata corresponding to the formula $\varphi = \Box(p \Rightarrow \bigcirc\Diamond q)$

3. Compute the product $\mathcal{A}_{sys} \bigotimes \mathcal{A}_{\neg\varphi}$.
4. If there is no accepting run (a cycle through an accepting state) in the product automata, declare that the property φ is true in the system *sys*. This is justified by (11.2). Otherwise, the property φ is not true in the system *sys*, and every accepting run in the product automata is a counterexample to the claim.

Example 21 illustrates the model checking steps using the emptiness criteria in the product of two Büchi automata.

Example 21 We want to verify the property $\varphi = \Box(p \Rightarrow \bigcirc\Diamond q)$ in the Büchi automaton shown in Fig. 11.8. The steps are shown below:

1. Let B_{sys} be the Büchi automaton in Fig. 11.8.
2. We need to transform $\neg\varphi$ into a *normal form* in which all negations are pushed inside the temporal operators. We use the duality axioms (see Sect. 11.3.5) successively:

$$\begin{aligned}
\varphi &= \Box(p \Rightarrow \bigcirc\Diamond q)\\
\neg\varphi &= \neg\Box(p \Rightarrow \bigcirc\Diamond q)\\
&= \Diamond\neg(p \Rightarrow \bigcirc\Diamond q)\\
&= \Diamond(p \wedge \neg \bigcirc \Diamond q)\\
&= \Diamond(p \wedge \bigcirc\neg\Diamond q)\\
&= \Diamond(p \wedge \bigcirc\Box\neg q)
\end{aligned}$$

   The Büchi automaton $B_{\Diamond(p\wedge\bigcirc\Box\neg q)}$ is shown in Fig. 11.11.
3. We construct the product automaton. In the state t_2 of the automaton $B_{\neg\varphi}$, the only transition is labeled by $\neg q$, which is not a label of any transition in the automaton B_{sys}. Hence, in the product automaton, there is no transition from any state $(*, t_1)$, where $*$ is a state of B_{sys}. The transition $t_0 \xrightarrow{\text{true}} t_0$ matches with any transition in B_{sys}, and the transition $t_0 \xrightarrow{\text{p}} t_1$ labeled p can be taken simultaneously with the transitions in B_{sys} that have label p. The product automata $B_{sys} \bigotimes B_{\neg\varphi}$, as shown in Fig. 11.12, can be constructed in this manner. Since there is no cycle through the accepting states in the product automaton, we conclude that the property φ is true in the Büchi automaton shown in Fig. 11.8. □

11.6 Exercises

1. Express $(\Diamond\varphi)$ using $\mathcal{U}$ operator. Express $\Box\varphi$ using $\Diamond$ operator.
2. Give natural language statements for the following formulas:

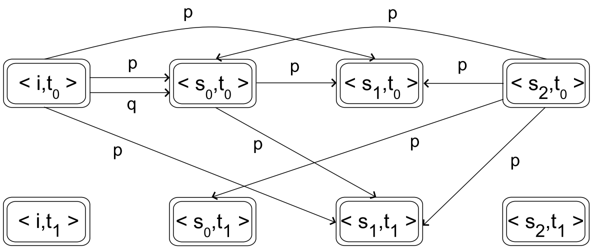


Fig. 11.12 Product of the Büchi automata in Figs. 11.8 and 11.11

- $\Diamond(\varphi_1 \mathcal{U} \varphi_2)$
- $\Diamond\varphi_1 \mathcal{U} \varphi_2$
- $\Box\varphi \Rightarrow \Diamond\psi$
- $\neg\Box(x > 1)$
- $(\neg q \mathcal{U} p) \vee \Box p$
- $\Diamond(\varphi \wedge \bigcirc\neg\varphi)$

3. Give temporal logic formulas for the following statements:
   (a) It is always true that at some future moment only one of φ or ψ becomes true.
   (b) Initially $x > 0$, $y > 0$, at some future moment $x > y$ is true and it stays true unless y becomes negative.
   (c) A printer completes the printing of a file within a finite amount of time.
   (d) Only one user can use the printer at a time.
   (e) The gate eventually opens after the train exits from the crossing, provided the gate does not receive close message from the controller within 3 time units from the instant the train exits.
   (f) It is not the case that the channel delivers two identical messages in successive steps.
   (g) Whenever the input number is odd, the execution never terminates.
   (h) Whenever the parity bit is 1 in a register, the register contents do not change for the next two steps.
   (i) Events p and q alternate infinitely often.
   (j) Event p should eventually followed by event q or event r but not by both.
4. Write a fairness condition for Dining Philosopher's problem (Example 5) in PTL.
5. Prove or disprove the following equivalences:

- $\Box\varphi \equiv \bigcirc(\Box\varphi)$
- $\Diamond(\varphi \wedge \psi) \equiv \Diamond\varphi \wedge \Diamond\psi$
- $\bigcirc\Diamond\varphi \equiv \Diamond\bigcirc\varphi$
- $\Box(\varphi \Rightarrow \psi) \equiv (\Box\varphi \Rightarrow \Box\Diamond\psi)$
- $(\bigcirc\varphi \Rightarrow \psi) \Rightarrow \bigcirc(\varphi \Rightarrow \psi)$

6. Prove the equivalences stated in Sect. 11.3.5.
7. Define the temporal operator $\mathcal{N}$ (**atnext** [14]), to be used as $\varphi\mathcal{N}\psi$, using the basic temporal operators. It has the meaning "φ holds at the nearest future time point where ψ holds". Either prove or give a counterexample for the following formulas:

- $\Box\varphi \Rightarrow \varphi\mathcal{N}\psi$

- $\Box(\varphi \Rightarrow \varphi \mathcal{N} \psi)$
- $\Box(\varphi \mathcal{N} \psi) \Rightarrow \Box\Diamond\psi$

8. Below is a temporal specification for two components A and B that communicate through messages.

$$Spec_A : \Box\begin{bmatrix} start \Rightarrow \varphi \\ \wedge\varphi \Rightarrow \bigcirc\psi \\ \wedge\psi \Rightarrow \bigcirc\varphi \\ \wedge\psi \Rightarrow \bigcirc\bigcirc send\_msg \end{bmatrix}$$

$$Spec_B : \Box\begin{bmatrix} rcv\_msg \Rightarrow \bigcirc\vartheta \\ \wedge\vartheta \Rightarrow \bigcirc\eta \\ \wedge\eta \Rightarrow \bigcirc\vartheta \end{bmatrix}$$

Give a model of the following concurrent specifications:

(1) $Spec_1 :: Spec_A \wedge Spec_B \wedge \Box[send\_msg \Rightarrow \bigcirc rcv\_msg]$

(2) $Spec_2 :: Spec_A \wedge Spec_B \wedge \Box[send\_msg \Rightarrow \Diamond rcv\_msg]$

Prove that $Spec_1$ is a refinement of $Spec_2$.

9. In the previous question suppose *send_msg* is changed to $send\_msg(t)$, where t is a token and it is either 0 or 1. Component B should send an acknowledgement (a simple message with no parameter) to component A if the value of token is 1. With this change, rewrite the concurrent specifications in Exercise 8.
10. With respect to the rail road crossing specification in Sect. 11.3.7.2 do the following:
    (a) Give formal semantics for the following formulas:
        (i) $g_4 \mathcal{W} tr_1$
        (ii) $g_4 \mathcal{S} tr_3$
        (iii) $tr_1 \mathcal{R} g_4$
    (b) Modify the railroad problem requirements: The system allows two trains, one controller, and one gate. When one train is in the crossing or busy crossing the gate, and the controller receives "approaching" message from the other train the controller should instruct the gate closed until both trains exit the crossing. However, both trains should not share the crossing at any moment. Give a specification for this problem.
11. Specify a *signal* object with the behavior that in the initial state it is *red*. When it receives a message at some time it changes to *yellow* at the next instant. It stays yellow until it receives a message to turn *green*. It stays *green* unless it receives a message that will turn it *red*. This behavior is repeated infinitely often. Modify the requirements of the rail road crossing problem in Sect. 11.3.7.2 as follows:
    (a) The controller informs the signal about the status of the gate through different messages.
    (b) If the signal is *red* and it receives the message from the controller that "the gate is open", it continues to stay in that state. When it receives the message that the "gate is lowered", it turns *yellow* in the next time moment. When the signal is *yellow*,

if it receives the message “gate is closed” from the controller, it turns *green* at the same time moment. When it receives the message “gate is raised” it changes to *yellow* at the next time moment. If it receives the message “gate is open” when it is *yellow*, it changes to *red* at the same moment.

(c) The train observes the signals and modifies its behavior. If it observes *red* signal it will *stop* before reaching the crossing. If it observes *yellow* it will *slow down* before reaching the gate. If it observes *green* it will either maintain its normal speed or *resume_normal* speed.

Give a formal specification for the reactive behavior of train, controller, signal, and gate components. Give a formal model of the system and informally infer from your model the satisfaction of the safety property.

12. This exercise is concerned with specifying a plain old telephone system: a system consisting of a finite number of telephones, where communication between two telephones is possible through a *channel*. All telephones are connected to one channel. Each telephone has a unique number. The state of a telephone is determined by the truth values of the following set of predicates: *on-hook(x), off-hook(x), dial-tone(x), ringing(x), busy-tone(x)*. A user can perform the following actions: *lift-handset(x), replace-handset(x), dial-number(x,y)*. When a number y is dialed in a telephone x, either there is a busy tone at the telephone y, or the telephone y rings. In the latter case, either the phone is eventually picked up or it is not picked up. If the telephone is picked up, the telephone is said to be connected, and if the telephone is not picked up there is no connection. In such an operation no other telephone is affected.
    (a) Give temporal logic formulas for a set of constraints on the telephone system. Example constraints are “a telephone is either on-hook or off-hook, but not both”, and “a phone cannot have a busy tone and not be connected to some number”.
    (b) Give a temporal logic specification for placing a call that results in a successful connection. Give a channel specification, as complete as possible, so that you can provide the full specification of processing a telephone call. Verify using the formal model of the system that every call made by a telephone is connected unless the telephone at the destination is busy.
13. Give temporal logic semantics for the program below:

```
begin
x:=0;
y:=3;
while (x <= y) do
{
 if even(x) then <y:=y+x; x;=x+1>
            else <y:=y-2;x:=y+x;>
}
end
```

14. Modify the requirements of Example 14 as follows: the slot machine sends a randomly chosen amount x, which is either $(max - 2)$ or $(min + 2)$, as long as the cash balance in the slot machine remains greater than $(max + min)$ (of the previous round). Once the balance falls below this limit it continues to send an amount 0, until its balance is

once again at least ($max + min$). This behavior is repeated forever. Give the complete specification of the slot machine interaction with its players.

15. *Craps* is a gambling game which eventually terminates. A player, say *Alice*, plays the game using a machine that generates *random numbers* to simulate the roll of a pair of dice. The rules of the game and declaring the result of the game rest with a referee (R), an automatic device, with whom *Alice* and her random number generating (*RAG*) device communicate. Each time *Alice* asks *RAG* for a number, *RAG* delivers the referee R an integer in the range $[1,7]$ and the name *Alice*. To simulate a throw on a pair of dice, *Alice* will ask *RAG* at two successive moments. The device *RAG* takes one moment of time to generate the number and one moment of time to deliver it to R. Two moments after the judge receives the necessary information, it will announce the result to *Alice*, and *Alice* will receive the result the moment after it is sent by R. The referee has set down the following rules for playing the game.

    - Let a and b denote the random numbers received after *Alice* requests for it.
    - Let $s = a + b$
    - If $s = 7$ or $s = 11$, R announces that *Alice* is the winner. The game is over.
    - If $s = 2$ or $s = 3$ or $s = 12$, R announces that *Alice* is the loser. The game is over.
    - In all other cases R requests *Alice* to play again.
    - *Alice* should request *RAG* at two successive moments to get new numbers.
    - Let a' and b' be the numbers received by R.
    - Let $s' = a' + b'$.
    - If $s' = 7$ the referee announces that *Alice* is the loser. The game is over.
    - If $s' = s$ the referee announces that *Alice* is the winner. The game is over.
    - The referee follows the rules from Step 5 until the game is decided.

    Give a temporal logic specification for playing this game. HINT: Define the operator *while* :: $\varphi \mathcal{W} \psi$ meaning "φ holds at least as long as ψ holds", and use it to specify the loop.

16. In software design, the *publisher–subscriber* design pattern is often used. This problem is on modeling the behavior of *publisher–subscriber*. Component C_p is *publisher* of information, and components $C_{s_i}, i \geq 0$ are subscribers to information published by C_p. Give a model of the following behavior.

    - *subscribe:* A component repeatedly sends message *enroll( )* to the publisher C_p until it receives an acknowledgment *enrolled(i)*. The publisher assigns the ID i to the component, and is named C_{s_i}.
    - *publish:* The publisher C_p sends out (broadcasts) information ϑ at *odd* time moments $t \geq 3$, to all components who are enrolled at time k, $k < t$.
    - *release:* A component C_{s_i} repeatedly sends message *release(i)* to the publisher C_p until it receives an acknowledgment *released(i)*. The publisher removes ID i.
    - *membership:* A component that received an acknowledgment *enrolled(i)* is a subscribed member from the moment of enrollment to the moment of release.

    Prove the following properties:

    - *safety property:* Only subscribers should receive the information published by the publisher.

- *liveness property:* All subscribers receive the same information.

17. Construct the program graph and the transition system corresponding to the program

```
begin
while (x > 1)
if (x > y) then x:=x-y;
if (x <= y) then y:=y-x;
}
end
```

18. Consider the two actions

$$P_1 :: \langle y := y - x; x := x + 2\rangle$$

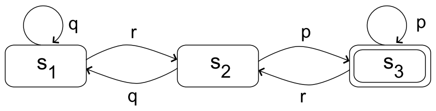
$$P_2 :: \langle x := y + x; x := x - 2\rangle$$

as programs that share the variables x and y. Construct

- the program graph PG_1 and PG_2 of the programs P_1 and P_2
- the program graph of $PG_1|||PG_2$
- the transition systems TS_{PG_1}, TS_{PG_2} and $TS_{PG_1|||PG_2}$ corresponding to program graphs PG_1, PG_2, and $PG_1|||PG_2$

Is $TS_{PG_1}|||TS_{PG_2} = TS_{PG_1|||PG_2}$

19. Model check the formula $\Diamond(r \Rightarrow p \vee \neg q)$ on the Büchi automaton model shown below.



20. Transform the Kripke structure (S, R, S_0, L) defined below to a Büchi automaton and model check the formula $\Diamond(p \vee t)$ in it.
$S = \{s_0, s_1, s_2, s_3, s_4\}$,
R: $s_0 \xrightarrow{\alpha_1} s_1$, $s_2 \xrightarrow{\alpha_2} s_1$, $s_2 \xrightarrow{\alpha_3} s_4$, $s_3 \xrightarrow{\alpha_2} s_0$, $s_3 \xrightarrow{\alpha_1} s_2$, $s_4 \xrightarrow{\alpha_3} s_0$
L: $L(s_0) = \{p, s\}$, $L(s_1) = \{r, s\}$, $L(s_2) = \{q, r\}$, $L(s_3) = \{p, q\}$,
$L(s_4) = \{p, t\}$, where p, q, r, s, t are propositions.
$S_0 := \{s_0, s_4\}$

11.7 Bibliographic Notes

Modal logics [4, 13] and temporal modalities [25] are the foundations on which the temporal logics are based. Linear temporal logic was introduced by Pnueli [22]. Since then a variety of temporal logics have been studied, most notably CTL (Clock Temporal Logic) [2, 9], CTL*, and Lamport's Temporal Logic of Actions [16]. Both CTL and CTL* are branching time logics. CTL* subsumes both CTL and LTL. There are logics based on them for industrial usage. These include IBM Sugar based on CTL, Intel FORSpec based on LTL, PSL (IEEE-1850 standard) that incorporates features from CTL*, and the TLA+2 tools. In PSL, it is possible to write both property-oriented and model-based specifications.

Temporal logic is most suited for specifying the behavior of reactive and concurrent systems. The behavior of such systems, considered as traces, are collections of infinite words which constitute an ω-language over an alphabet. This connection between LTL formulas and ω-languages was first studied by Sistala et al. in [27]. This seminal work lead to model checking using Büchi automata which recognizes ω-languages.

The term model checking refers to a collection of methods for the automatic analysis of properties, such as safety, liveness, and fairness in reactive and concurrent systems. The input to model checkers are always *abstract models* of the system. A positive verdict from a model checker is of "limited value" because of possible (and potential) *incompleteness* in the model, approximations that might have lead to contain the complexity of the model, and limitations imposed by available resources. The positive aspect of getting a negative verdict is that the counter-examples produced by the model checker are invaluable in debugging complex systems. Testing community use model checkers just for this feature. On the whole, model checking has the following strengths:

1. It is a systematic approach and is an automatic verification procedure.
2. Many tools are available; for example SPIN system developed by Holzmann [12] received an award for its outstanding performance. Consequently, model checking has become portable and scalable to industrial applications, requiring only a moderate level of expertise in the formal notations used.
3. Many systems, some fairly large, have been model checked. It is now done *routinely* on a widespread basis for verifying properties of large systems composed of both hardware and software.
4. There is a wide networking community of users, researchers, and developers. They continue to produce new techniques and tools.

However, there are still a few problems with this approach.

1. It supports only *partial* verification. That is, only one property can be checked at any one time. In order to check fairness and safety, two independent invocations to a model checker is necessary.
2. Positive verdict of one property cannot guarantee that the system is free of other errors.
3. It is restricted to verifying stated properties in finite state models, and not on actual systems.
4. It is not suitable for infinite state systems. Many real world systems may not be finite state systems. Many techniques, such as *predicate abstraction*, *partial order reduction*, and *symbolic representations* are known [1, 6] to reduce infinite state systems to finite state systems. Not all these techniques are fully automated.
5. State explosion is a common problem. A reactive system may not terminate and a concurrent system may have many asynchronous processes. The formulas required to rep-

resent the state spaces of such systems are usually quite complex, making the state space hard to contain within manageable size.

The result of Sistala and Clarke [28] on the complexity of LTL model checking sets limits on what is practically feasible. Model checking is done *only* on an abstract model which cannot be built automatically. Because of the disadvantages listed above, standard validation procedures such as testing are necessary to ensure that the implementation adequately reflects the properties verified by a model checker as well as properties that are not model checked.

References

1. Baier C, Katoen JP (2007) Principles of model checking. MIT Press, Cambridge
2. Ben-Ari M, Pnueli A, Manna Z (1983) The temporal logic of branching time. Acta Inform 20:207–226
3. Büchi JR (1960) On a decision method in restricted second order arithmetic. Z Math Log Grundl Math 6:66–92
4. Carnap R (1947) Meaning and necessity. Chicago University Press, Chicago. Enlarged Edition 1956
5. Clarke EM, Emerson EA, Sistala AP (1986) Automatic verification of finite state concurrent systems using temporal logic specifications. ACM Trans Program Lang Syst 8(2):244–263
6. Clarke EM, Grumberg O, Peled DA (1999) Model checking. MIT Press, Cambridge
7. Courcoubetis C, Vardi M, Wolper P, Yannakakis M (1992) Memory efficient algorithms for the verification of temporal logic properties. Form Methods Syst Des 1:275–288
8. Dijkstra E (1965) Solutions of a problem in concurrent programming control. Commun ACM 8(9):569
9. Emerson EA, Clarke EM (1980) Characterizing correctness properties of parallel programs using fixpoints. In: Automata, languages, and programming. Lecture notes in computer science, vol 85, pp 169–181
10. Gabbay D, Pnueli A, Stavi J (1980) The temporal analysis of fairness. In: Proceedings of the seventh ACM symposium on principles of programming languages, January 1980, pp 163–173
11. Gabbay D, Hodkinson I, Reynolds M (1994) Temporal logic: mathematical foundations and computational aspects. Oxford logic guides, vol 1. Clarendon, Oxford
12. Holzmann G (2003) The SPIN model checker: primer and reference manual. Addison-Wesley, Reading
13. Kripke SA (1963) Semantic considerations on modal logic. Acta Philos Fenn 16:83–94
14. Gröger F (1986) Temporal logic of programs. EATCS monographs on theoretical computer science. Springer, Berlin
15. Lamport L (1983) What good is temporal logic. In: Proceedings of IFIP'83 congress, information processing. North-Holland, Amsterdam, pp 657–668
16. Lamport L (1994) The temporal logic of actions. ACM Trans Program Lang Syst 16(3):872–923
17. Manna Z, Pnueli A (1992) The temporal logic of reactive and concurrent systems—specifications. Springer, New York
18. Melham TF (1989) Formalizing abstraction mechanisms for hardware verification in higher order logic. PhD thesis, University of Cambridge, August 1989
19. Moszkowski B (1986) Executing temporal logic programs. Cambridge University Press, Cambridge

20. Owiciki S, Lamport L (1982) Proving liveness properties of concurrent programs. ACM Trans Program Lang Syst 4(3):455–495
21. Peterson GL (1981) Myths about the mutual exclusion problem. Inf Process Lett 12(3):115–116
22. Pnueli A (1977) The temporal logic of programs. In: Proceedings of the eighteenth symposium on the foundations of computer science, Providence, USA
23. Pnueli A (1981) The temporal semantics of concurrent programs. Theor Comput Sci 13:45–60
24. Pnueli A (1986) Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. Lecture notes in computer science, vol 224
25. Prior A (1957) Time and modality. Oxford University Press, London
26. Rescher N, Urquhart A (1971) Temporal logic. Springer, Berlin
27. Sistala AP, Vardi M, Wolper P (1983) Reasoning about infinite computation paths. In: Proceedings of the 24th IEEE FOCS, pp 185–194
28. Sistala AP, Clarke EM (1985) The complexity of linear propositional temporal logic. J ACM 32(3):733–749
29. Stirling C (1992) Modal and temporal logics. In: Handbook of logic in computer science. Oxford University Press, London
30. Wan K (2006) Lucx: Lucid Enriched with context. PhD thesis, Concordia University, Montreal, Canada