Chapter 7
Leaping reachability analysis for LTL model-checking

Throughout the preceding chapters we have concentrated on relief strategies for verifying logical correctness properties of protocols, viz. indefinite progress, freedom of non-executable transitions, freedom of unspecified receptions and freedom of buffer overflows. These are general properties that are pertinent to basically all protocols, independent of their intended functionality. They are therefore often regarded also as “syntactic” correctness requirements. The focus of this chapter is instead on proving “semantic”, or functional correctness requirements of protocols and, moreover, of concurrent systems in general. Specifically, we consider the verification of properties that are expressible as formulas in linear-time temporal logic. Linear-time temporal logic (LTL) is a popular formalism for reasoning about the semantic correctness of concurrent systems. It is well suited for specifying temporal properties over infinite executions of a system, including arbitrary safety and liveness properties. Given a concurrent system and a temporal formula (in LTL), verifying that every execution of the system satisfies the formula is known as (LTL) model-checking.

For many concurrent systems, model-checking also suffers severely from the state explosion problem. An eminent and quite general approach to relieve the state explosion problem for model checking is the partial-order approach, which actually captures a group of cognate state exploration techniques called partial-order reduction methods. These methods have been developed in recent years by different researchers [God90, Val90, HGP92, KP92a, Val92, Val93, GW93, GW94, HP95, Pel96], largely independent of the specific model used for specifying concurrent systems. They have proved effective for verifying local and termination properties of concurrent systems, as well as for LTL model-checking. Experiments have shown that these methods can substantially reduce the space and time needed for LTL model-checking.

Notwithstanding these results, in this chapter we propose an enhancement of partial-order reduction methods, in particular of the most recent and advanced one described in [HP95, Pel96].
Following a prelude to concurrent systems, temporal logic and LTL model-checking, we describe the partial-order approach to LTL model-checking. We then show how the concepts underlying LRA in Chapter 5 can be combined with this approach to further relieve the state explosion problem for LTL model-checking. This is done first for concurrent systems in general, and subsequently for protocols defined in the CFsM model. Lastly, we provide an empirical comparison between the partial-order reduction method in [HP95, Pel96] and its proposed enhancement on the basis of experiments performed with the research tool package RELIEF discussed in Chapter 6. The main contributions of this chapter appeared in [Sch97, SU98b]

### 7.1 Preliminaries

#### 7.1.1 Representing concurrent systems

Thus far we have confined ourselves to the CFsM model for the specification and verification of communication protocols. In Chapter 2, a protocol was defined explicitly as a set of processes (or finite state machines) which communicate asynchronously by exchanging messages over FIFO queues. In this chapter we consider more generally any (concurrent) system that can be formalized as a finite labeled transition system (LTS). An LTS is defined in the customary way as a quadruple \((Q, q_0, \Sigma, D)\), where

- \(Q\) is a finite set of states,
- \(q_0 \in Q\) is the initial state of the LTS,
- \(\Sigma\) is a finite set of labels (the alphabet of the LTS), and
- \(D \subseteq Q \times \Sigma \times Q\) is a transition relation.

An LTS can be used most elementary to formalize the behavior of a single sequential process. It can also formalize the joint behavior of a finite number of interacting and concurrently executing sequential processes. Each transition of the LTS then corresponds to the execution of a specific atomic operation or statement within one (or some, in case of synchronization) of the processes, in accordance with a standard interleaving semantics of concurrency. An LTS is sufficiently abstract to model virtually any finite-state system. In particular, every (bounded) protocol specified in the CFsM model can be represented by an LTS: the behavior of such a protocol \(P = \{P_i \mid i \in I\}, L\) is formalized by the tuple \((Q, q^0, \Sigma, \Delta)\), with \(Q = R_P, q^0 = G^0, \Delta = \bigcup_{i \in I} \Delta_i\), and \(\Delta = \{G, t, H \mid G \in R_P \subseteq G, t \in H\}\) (cf. Chapter 2). In other words, all send and receive transitions in the protocol are viewed as atomic operations that act as global state transformers. Henceforth, we designate the term concurrent system to mean any system that can be formalized as a finite LTS, and we assume
implicitly that the system is composed of a finite number of distinguishable sequential processes \( P_1, P_2, \ldots, P_n \). The labels of the transitions of a concurrent system are referred to as operations. Definition 7.1 captures the basic semantics of concurrent systems.

**Definition 7.1**

Let \( S = (Q, q^0, \emptyset, \Box) \) be a concurrent system. An operation \( a \Box\emptyset \) is executable at a state \( q \Box Q \) if \( (q, a, q) \Box \emptyset \). The set of all operations (of a sequential process \( P_i \) in \( S \)) that are executable at \( q \) is denoted by \( X(q) (X_i(q)) \). A computation of \( S \) over a sequence \( \overline{a} = a_1a_2\ldots \) of operations from \( \emptyset \) is a finite or infinite sequence \( \overline{q} = q_0q_1q_2\ldots \) of states from \( Q \), where (i) \( q_0 = q^0 \), i.e. the sequence starts at the initial state of \( S \), (ii) \( q_i \overset{a_i}{\rightarrow} q_{i+1} \) for all \( i \geq 1 \) (and \( i \emptyset \Box \) when \( \Box \) is finite), and (iii) the sequence is maximal, i.e. \( \Box \) is either infinite or its last state \( q \Box \) satisfies \( X(q) = \emptyset \). A state \( q \Box Q \) is a reachable state if it occurs in some computation of \( S \).

Like in the previous chapters, we use \( q \overset{a}{\rightarrow} q \) to denote that the operation \( a \) leads from the state \( q \) to the state \( q \Box \) i.e. \( (q, a, q) \Box \emptyset \), and \( q \overset{a_1, a_2, \ldots, a_n}{\rightarrow} q \) to denote that the finite sequence of operations \( \overline{a} \) leads from \( q \) to \( q \Box \). Naturally, a computation of a concurrent system, or any segment thereof, may be seen either as a sequence of reachable states, or as the corresponding sequence of executed operations.

For ease of presentation, we will denote a computation interchangeably by a sequence of operations or by a sequence of states. As for protocols specified in the CFSM model, the complete reachable state space of a concurrent system \( S \), including all its computations, can be represented by a labeled directed graph. Every (finite or infinite) path through this reachability graph that starts with the node corresponding to the initial state of \( S \) resembles the effects of a computation of \( S \). There is one such path for each possible way in which the execution of operations can be interleaved in time.

### 7.1.2 Expressing properties of concurrent systems in temporal logic

Among the non-classical logics used in computer science, temporal logic has probably been the most successful. It is an extension of traditional logic (i.e. Boolean algebra and simple predicate calculus), and was first suggested by Pnueli [Pnu77] as a tool for the specification and verification of concurrent systems (also called concurrent programs). Temporal logic provides a sound logical basis for reasoning about the time varying behavior of concurrent systems without introducing an explicit notion of time. That is, it has been designed to reason about the order in which system events occur, as opposed to the actual times at which they occur. Expressions in temporal logic typically assert properties of sequences of states, whereas expressions in traditional logic assert properties of individual states.

Two variants of temporal logic are commonly distinguished, based on different conceptions of the nature of time: linear or branching. In linear temporal logic, time is viewed as linear, meaning
that each instant in time has a unique possible future (i.e. next instant). The structures over which linear temporal logic is interpreted are thus linear sequences. In branching temporal logic, each time instant may split into several possible futures, for instance those resulting from nondeterminism. All these possible futures are then considered to be equally real, while in linear temporal logic only one of them is regarded as the future that will actually occur. The structures over which branching temporal logic is interpreted can be viewed as infinite trees. Both the linear and branching variants of temporal logic are well-established. Excellent surveys are given in [Lam80, Wol89, Eme90]. Lamport [Lam80] has argued that the logic of linear time is better suited for reasoning about concurrent systems. Wolper [Wol89] has found that the linear variant is natural when the properties of a system can be expressed in terms of its computations, and that the branching variant is well adapted when the properties are thought of in terms of the structure of the system. In general, it appears largely a matter of debate as to which variant is to be preferred [Eme90]. For the purpose of this chapter we consider linear-time temporal logic (LTL) only.

Formulas in LTL assert properties of infinite sequences of states. An LTL formula is built from Boolean propositions, the Boolean connectives ‘¬’, ‘[]’, …, and the temporal operators ‘○’ (read as “next-time”), ‘□’ (read as “henceforth” or “always”), ‘◊’ (read as “eventually”), and ‘U’ (read as “until”). Precisely, let AP denote a finite set of atomic propositions, and \( p \Box AP \), then the syntax of LTL is as follows:

\[
f ::\ = \ p \mid \neg f \mid f_1 f_2 \mid ○ f \mid □ f \mid ◊ f \mid f_1 U f_2 .
\]

The atomic propositions in the set AP are assumed to refer to the states of the concurrent system for which an LTL formula asserts a property. Indeed, since all states of a concurrent system \( S \) are in essence just different combinations of values assigned to the variables that constitute \( S \), each state is uniquely characterized by the subset of atomic propositions which hold true in that state. Every computation of \( S \) can therefore be interpreted as a propositional sequence over the set \( 2^\text{AP} \). Although LTL allows assertions only over infinite sequences of states, in order to take into account also terminating computations of \( S \) it is common practice to transform these finite computations into infinite ones simply by repeating the last state forever [Val92, Pel96]. Let \( \square = q_0 q_1 q_2 \ldots \) be an infinite sequence of states interpreted as a propositional sequence, and denote by \( \square^i \) the suffix of \( \square \) starting from its \( i \)-th state. For an LTL formula \( f \), one usually writes \( \square \models f \) to mean that \( \square \) satisfies \( f \) (or \( f \) holds true for \( \square \)), in accordance with the following semantics:

- \( \square \models p \) iff \( p \Box AP \) holds true in \( q_0 \),
- \( \square \models \neg f \) iff not \( \square \models f \),
- \( \square \models f_1 \Box f_2 \) iff \( \square \models f_1 \) and \( \square \models f_2 \),
- \( \square \models ○ f \) iff \( \square^1 \models f \),
\[ \square \lvert= \square f \iff \square i \geq 0: \square i \lvert= f, \]
\[ \square \lvert= \Diamond f \iff \square i \geq 0: \square i \lvert= f, \]
\[ \square \lvert= f_1 U f_2 \iff \square i \geq 0: \square i \lvert= f_2 \text{ and } \square 0 \square j < i: \square j \lvert= f_1. \]

Note that the “eventually” operator is the dual of the “henceforth” operator, viz. \( \Diamond f = \neg \square \neg f \). Also note that \( \Diamond f \) and \( \square f \) can be expressed in terms of the “until” operator as \( \text{true} U f \) and \( f U \text{false} \), respectively. Informally, where the Boolean connectives (including ‘\( \neg \)’ and ‘\( \square \)’ defined in terms of ‘\( \neg \)’ and ‘\( \square \)’) have their usual interpretations, the temporal operators have the following meaning:

- \( \Diamond f \) holds in the current state if \( f \) holds in the next state,
- \( \square f \) holds in the current state if \( f \) holds in the current state and in all subsequent states (in the linear sequence on which the formula is interpreted),
- \( \Diamond f \) holds in the current state if \( f \) holds in the current state or in some subsequent state, and
- \( f_1 U f_2 \) holds in the current state if \( \Diamond f_2 \) holds in the current state, and if \( f_1 \) holds in the current state and in all subsequent states preceding the state in which \( f_2 \) holds.

Unlike the “henceforth”, “eventually” and “until” operators, the “next-time” operator is often omitted by researchers in reasoning about temporal properties of systems. This has been instigated mostly by Lamport, who strongly objects to the use of the “next-time” operator, claiming that it introduces a notion of time which is too discrete (namely between two immediately following time instants) to fit the level of abstraction appropriate for a specification formalism [Lam83]. Formally, he found that every nexttime-free LTL formula is closed under stuttering, meaning that the formula cannot distinguish between two stuttering equivalent sequences. Two sequences of states, or the corresponding propositional sequences, are stuttering equivalent if they can be made identical by replacing in both sequences every finite adjacent number of occurrences of the same state with a single occurrence [Lam83]. Since a temporal formula containing the “next-time” operator is not necessarily closed under stuttering, Lamport argued that this operator enables the expression of distinctions between systems that should be considered equivalent.

Many interesting properties of concurrent systems can be formalized in LTL, even without the “next-time” operator. For instance, that some property \( P \) is invariant throughout system execution is expressed simply as \( \square P \). In order to state that a property \( P \) always causes a property \( Q \) to hold subsequently, one writes \( \square (P \square \Diamond Q) \). This combination of operators is often used to specify the eventual response (i.e. \( Q \)) to some given request (i.e. \( P \)). Asserting that a property \( P \) is satisfied infinitely often is done by writing \( \square \Diamond P \). This means that for each state along a computation there is a future state in which \( P \) will be true. Lastly, an expression of the form \( P \square (P U Q) \) asserts that if
$P$ is true in the current state, then it will remain true at least until $Q$ becomes true.

In general, (nexttime-free) LTL is sufficiently expressive to capture all safety and liveness properties of concurrent systems. These properties have been regarded by many researchers as the two most fundamental types of properties one would want to prove of a concurrent system (see e.g. [Lam77, Lam80, Lam83, WVS83, AS87, MP92]). Intuitively, a safety property asserts that something bad never happens, whereas a liveness property asserts that something good must eventually happen [Lam80]. “Something bad” thereby refers to the system entering an unacceptable state, and “something good” to the system entering a desirable state. Well-known safety properties are partial correctness (a program never halts with the wrong answer), mutual exclusion (two processes are never in their critical sections at the same time), and indefinite progress or deadlock-freedom (a system never enters a state in which no further progress is possible). The absence of unspecified receptions and buffer overflows considered earlier for protocols in the CFSM model also classify as safety properties. Note that all these examples of safety properties are (global or local) state invariances. Safety properties can also stipulate some precedence relation between states or events. For systems that implement FIFO buffers, an example is the assertion that messages are taken from a buffer in the same order as they are put into the buffer (this property was assumed to hold for protocols in the CFSM model). Familiar liveness properties are termination (a program eventually halts) and freedom from starvation or livelocks (each process makes progress infinitely often). Other kinds of liveness properties that are important for concurrent systems include such assertions as “a request for service will eventually be granted”, “a process will eventually enter its critical section”, or “a message will eventually reach its destination”.

A formalization of safety and liveness properties appeared in [AS87], where the two kinds of properties are characterized from a language-theoretic point of view. It was also shown in [AS87] that every property which classifies neither as a safety property nor as a liveness property is in fact the conjunction of a safety and a liveness property. The safety/liveness classification is further discussed in view of a more refined hierarchy of temporal properties in [MP92], which includes a syntactic characterization of safety and liveness properties in terms of the temporal formulas that specify them. Basically, safety properties can be expressed using only the concept of “henceforth” (typically in the form $\Box P$ or $P \quad \Box Q$), whereas one needs the additional concept of “eventually” to express liveness properties (usually in the form $\Box \Diamond P$ or $(P \quad \Diamond Q)$).

Another useful concept in the application of temporal logic to concurrent systems is fairness. Considering fairness means taking into account certain assumptions about the context in which processes of a concurrent system are executed. For instance, if concurrent processes are executed on different processors it is customary to assume that, if a process has an operation that remains executable, it will eventually execute it (this assumption is often called weak fairness). Various
notions of fairness have been studied in [Fra86, MP92]. The purpose of these notions is to exclude from analysis the computations of a concurrent system that would not be permitted by the specific type of process scheduler that is assumed. The fairness assumptions then act as filters, removing certain classes of infinite computations that conflict with the assumptions made about the process scheduler. Like safety and liveness properties, fairness assumptions can be expressed in (nexttime-free) LTL [LP85]. If a fairness assumption is formalized by an LTL formula $f_1$, one can use a logical implication $f_1 \implies f_2$ to assert that the property expressed by $f_2$ holds true under this fairness assumption.

### 7.1.3 Model-checking

Currently the most advocated method for verifying temporal properties of (finite-state) concurrent systems is model-checking. In the context of LTL, model-checking refers to a fully automatic procedure for checking that a given concurrent system satisfies, or is a model of, some property formalized as an LTL formula [LP85, VW86, Wol89]. A concurrent system is thereby defined to satisfy an LTL formula if all the computations of the system satisfy the formula. At first, (LTL) model-checking was proposed as an off-line procedure. This means that the actual algorithms for verifying the satisfiability of LTL formulas are applied to a concurrent system after constructing the reachable state space of the system. These algorithms do not work directly on the state space, but rather construct from it a graph which contains in each node information to derive the formulas that hold true in the state represented by the node, based on fixpoint characterizations of the temporal operators (for instance, the fixpoint characterization of $\Box P$ is $P \land \Box \Diamond P$) [MW84, LP85]. It was recognized later that model-checking can also be performed on-the-fly [VW86], in which case the verification algorithms start to examine a given concurrent system during the construction of its reachable state space, not waiting for this construction to be completed. The main advantage of on-the-fly model-checking is that, if the checked formula does not hold true for the system, a counter example may be encountered before completing the construction of its state space. It is well argued in [Val93] that this advantage appears exactly when needed the most: the state spaces of incorrect systems tend to be “extra” large due typically to their faulty behavior. Another advantage of on-the-fly model-checking is that, in some cases, certain parts of the state space that are not important to the verification of the checked formula may be omitted, even when the formula happens to be satisfied by the system.

The ability to verify temporal properties on-the-fly has actually emerged from the so-called automata-theoretic approach to model-checking [WVS83, VW86]. This approach is based on the fact that for each LTL formula it is possible to construct a non-deterministic Büchi automaton [Büc62] that accepts exactly the (infinite) sequences of states satisfying the formula. Formally, a
Büchi automaton is a quintuple $A = (Q, q^0, \square, \blacksquare, F)$, where

- $Q$ is a set of states,
- $q^0 \square Q$ is the initial state,
- $\square$ is an alphabet,
- $\square \square Q \square \square Q$ is a transition relation, and
- $F \square Q$ is a set of acceptance states.

Büchi automata are a theoretical means to define languages of infinite strings: a string is accepted by a Büchi automaton if the automaton enters one of its acceptance states infinitely many times while reading the string. Notice that a Büchi automaton can be seen as an LTS with various states predefined as acceptance states. One defines a computation of a Büchi automaton $A$ over an infinite sequence of symbols $a_1a_2\ldots$ from $\square$ as an infinite sequence of states $q_0q_1q_2\ldots$ starting at the initial state $q^0$ of $A$, with $(q_{i-1}, a_i, q_i) \square \square$ for all $i \geq 1$ (cf. Definition 7.1). $A$ is then said to accept the computation (or the computation is accepting) iff for some acceptance state $q \square F$ there are infinitely many states $q_i$ such that $q_i = q$.

For an LTL formula $f$, the transitions in the corresponding Büchi automaton $A_f$ carry predicate labels from the alphabet $2^{AP}$, each of which represents a Boolean proposition [WVS83]. Recall that the atomic propositions of $AP$ in $f$ are supposed to refer only to the states of the concurrent system $S$ for which the formula formalizes a property, and hence these Boolean propositions are in fact propositions on the states of $S$ (i.e. on the values of the variables that constitute $S$). The Büchi automaton $A_f$ then accepts an infinite computation $\square$ of $S$ (a sequence of system states) iff there exists an accepting computation of $A_f$ (a sequence of automaton states) over $\square$.

In other words, if $\square$ satisfies the formula $f$ if there is a “path” $p$ in $A_f$ starting from the initial state of $A_f$, such that the label of the $i$-th edge in $p$ holds true in the $i$-th state of $\square$ for all $i \geq 1$, and some acceptance state of $F_f$ appears infinitely often in $p$. Further recall that the finite computations of $S$ are taken into account by first transforming them into infinite ones, through infinite repetition of their last states. An algorithmic construction of a Büchi automaton $A_f$ from an LTL formula $f$ can be found in for instance [Wol89]. This construction is exponential in the length of the formula, defined as the number of symbols (propositions and connectives) it contains. However, the exponential blow-up of the number of states in $A_f$ is usually not a concern since most formulas checked in practice are quite short, and since the construction algorithm often behaves much better than its upper bound [VW86, Wol89].

Example 7.2
Figure 7.1 depicts a Büchi automaton which accepts exactly all the infinite computations satisfying
the LTL formula $\neg \Box (P \bigcirc Q) = \Diamond (P \bigcirc \neg Q)$. That is, every sequence of states containing a state in which $P \bigcirc \neg Q$ holds true, and from which $Q$ never holds true in any state in the remainder of the sequence, is accepting. The initial state of the automaton is indicated by the symbol ‘ $'$ ‘, and its only acceptance state by a double circle. This formula can be used, for example, to express the negation of a precedence property of a concurrent system, stipulating that it is always the case that the execution of some operation $a$ (for instance, a send operation) is eventually followed by the execution of an operation $b$ (for instance, the matching receive operation). The LTL formula then accepts all computations that violate this property, i.e. all computations in which an occurrence of $a$ is never followed by an occurrence of $b$. The formula can indeed be described equivalently by a Büchi automaton over the alphabet of operations of the system, rather than by an automaton over state predicates as in Figure 7.1. Such an automaton over system operations is obtained from the automaton in Figure 7.1 by replacing $true$ with $\square$, $P \bigcirc \neg Q$ with $a$, and $\neg Q$ with $\square \backslash \{b\}$.

The automata-theoretic approach to LTL model-checking now proceeds as follows [WVS83, VW86]. Given a concurrent system $S$ and an LTL formula $f$ to be checked for $S$, one first builds a Büchi automaton $A_{\neg f}$ for the negation of $f$. It accepts all and only sequences of states that satisfy $\neg f$, i.e. that violate $f$. Secondly, one computes the so-called synchronous product of (the reachable state space of) $S$ and $A_{\neg f}$, a Büchi automaton which accepts exactly those computations of $S$ that violate $f$. This automaton is then checked for emptiness: either it does not accept any computation, implying that all computations of $S$ do in fact satisfy $f$, or it accepts at least one computation of $S$ which is a counter example to $f$.

The synchronous product of $S$ and $A_{\neg f}$ in the above three-step procedure is actually defined as the product of the two Büchi automata $A_S$ and $A_{\neg f}$, where $A_S$ is obtained from $S$ by designating all states of $S$ as acceptance states. Precisely, if $S = (Q_S, q^0_S, \Box_S, \square_S)$ then $A_S = (Q_S, q^0_S, \Box_S, \square_S, F_S)$, with $F_S = Q_S$, and the product of $A_S$ and $A_{\neg f}$ is the Büchi automaton $A_S \square A_{\neg f} = (Q, q^0, \Box, \square, F)$ defined by

- $Q = Q_S \square Q_{\neg f}$,
- $q^0 = (q^0_S, q^0_{\neg f})$,
- $\square = \Box_S \square \Box_{\neg f}$,
• $Q \subset Q$ such that $((x, y), (a, P), (x\bigwedge y) \qquad \text{iff} \quad (x, a, x \bigwedge y) \in S$, $(y, P, y \bigwedge y^\text{-f}$, and the Boolean proposition $P$ holds true in state $x \bigwedge Q_S$.

• $F = F_S \boxdot F_{\text{-f}}$.

For further referencing we call the Büchi automaton $A_S$ the full automaton for $S$. Notice that the transitions of $A_{\text{-f}}$ are essentially used to test the values of the variables of the concurrent system $S$ whenever the system is ready to execute an operation, as explained before. Operations of $S$ that can affect the truth value of any proposition in the LTL formula $f$ are also said to be visible. That is, an operation $a$ of $S$ is visible if there exists a transition label in $A_{\text{-f}}$ for which the corresponding proposition has a truth value in a system state $q$ that is different from its truth value in a state $q'$ with $(q, a, q \bigwedge q') \in S$; otherwise it is invisible. The set of all visible operations of $S$ with respect to the formula $f$ is denoted by $\text{vis}_f(S)$. As the exact set of visible operations is generally too hard to determine, in practice one would have to compute some upper approximation of $\text{vis}_f(S)$, which can be done by a syntactic analysis of (the operations of) $S$ [Val92].

An alternative definition for the product automaton $A_S \boxdot A_{\text{-f}}$ would apply if, as indicated in Example 7.2, $A_{\text{-f}}$ were taken as an automaton over the alphabet $S$ of system operations. In that case, the transitions of the product automaton would also carry labels from $S$, and the relation $Q \subset Q$ would be such that $((x, y), a, (x\bigwedge y) \qquad \text{iff} \quad (x, a, x \bigwedge y \in S$ and $(y, a, y \bigwedge y^\text{-f}$. In this framework, the transitions in $A_S$ and in $A_{\text{-f}}$ are thus synchronized on operations (i.e. on the transition labels), and the operations of $S$ that can, through synchronization, alter the state of $A_{\text{-f}}$ are the visible transitions. In the first definition of the product automaton one can see the transitions as being synchronized on states [God96]. Both frameworks are used in like manner for model-checking.

As it appears, the product automaton $A_S \boxdot A_{\text{-f}}$ can be computed without ever building the full automaton $A_S$. In other words, the reachable state space of $S$ need not be constructed explicitly. The product automaton can at the same time also be checked for emptiness. This is precisely what is meant by on-the-fly model-checking. First, the inspection of the state space of $S$ is guided by the checked formula, which acts as a constraint on the system’s behavior through the required coincidence of proposition labels (or the required synchronization of operations). In some cases the automaton $A_S \boxdot A_{\text{-f}}$ may therefore be smaller than $A_S$ itself. Second, the product automaton may be found non-empty before completing its construction. It is well-argued in [CV+92, Val93] that this advantage appears exactly when needed the most: state spaces of incorrect systems tend to be “extra” large due typically to their erroneous behavior. Deciding emptiness thereby amounts to checking whether there exists a cycle in $A_S \boxdot A_{\text{-f}}$ (when viewed as a graph) that is reachable from the initial state $(q^0_S, q^0_{\text{-f}})$ and that contains an acceptance state (which is hence repeated infinitely often). A particular memory-efficient algorithm for on-the-fly detection of such acceptance cycles is
given in [CV+92]. It requires space linear in the number of states of $A_S \Box A_{\neg f}$ and implements a so-called nested depth-first search: a first search to find a reachable acceptance state, followed by a second search to determine whether this acceptance state can be reached from itself. Checking whether the product of two Büchi automata is empty is known to be much easier than checking whether the language generated by one of the automata is included in the other (a PSPACE-complete problem), which explains why one uses the Büchi automaton for the formula $\neg f$ instead of $f$ [VW86, Wol89]. An overview of various algorithms for checking the emptiness of Büchi automata is given in [GH93], all of which also run in time linear to the size of these automata.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example.png}
\caption{The product $A_S \Box A_{\neg f}$ for the concurrent system $S$ and an LTL formula $f$ in Example 7.3.}
\end{figure}
Example 7.3
Consider a concurrent system $S$ composed of two sequential processes $P_1$ and $P_2$, depicted in Figure 7.2, and assume for simplicity that these processes execute autonomously, i.e. without any interaction. Suppose we want to check the absence of computations of $S$ in which operation $w$ occurs and operation $d$ does not occur. This property can be expressed by an LTL formula $f$ of the form $\Diamond \left( 'P_2\text{ is at local state } 22' \right) \land \Diamond \left( 'P_1\text{ is at local state } 13' \right)$. The negation of the property is described in Figure 7.2 as a Büchi automaton $A_{\neg f}$ over the alphabet $¥$ of operations of $S$. It accepts all infinite sequences of states starting with its initial state 0 in which its (only) acceptance state 1 is repeated infinitely often, and thus all computations of $S$ along which $w$ occurs and $d$ does not occur. Note that the operations $w$ and $d$ are the visible operations of $S$, as they can change the state of $A_{\neg f}$. The product automaton $A_5 \parallel A_{\neg f}$ is illustrated at the bottom of Figure 7.2, where the three double-circled states are the acceptance states (i.e. $A_{\neg f}$ must be in state 1). It is non-empty as it accepts several computations of $S$, namely those in which $w$ is executed and in which $a$ and $b$ are executed infinitely many times thereafter. In practice, the on-the-fly construction of $A_5 \parallel A_{\neg f}$ need not be completed, but can be stopped as soon as one of the accepting computations is found. $\square$

7.2 The partial-order approach to LTL model-checking

Being the main practical limitation for all verification methods based on state exploration, the state explosion problem must be reckoned with also when using model-checking for verifying temporal properties of concurrent systems. Certainly, on-the-fly model-checking already has a head start over off-line model-checking, but the often excessive size of the full automaton $A_5$ of a concurrent system $S$ still renders the construction of the product automaton $A_5 \parallel A_{\neg f}$ impractical for most real systems. An eminent and quite general approach to tackle the state explosion problem in concurrent system verification is the partial-order approach. This actually refers to a collection of cognate state exploration techniques, called partial-order reduction methods, that have been developed in recent years by different researchers [God90, Val90, HGP92, KP92a, Val92, Val93, GW93, GW94, HP95, Pel96]. Partial-order reduction methods are largely independent of the specific model used for specifying concurrent systems, and they have proved adequate not only for verifying local and termination properties [God90, Val90, HGP92, KP92a, GW93], but also for LTL model-checking [Val92, Val93, GW94, HP95, Pel96]. Experiments have indicated that these methods can substantially reduce the space and time needed for model-checking. They have also been shown to combine well with on-the-fly model-checking [Val93, GW94, Pel96] and with model-checking under certain fairness assumptions [Pel93, Pel96].

Like most improved state exploration techniques, partial-order reduction methods are inspired by the observation that many properties of interest to concurrent systems are insensitive to the
execution order of concurrent, or independent, atomic operations (see Chapter 3). Common to all partial-order reduction methods is the use of an explicit dependency relation among the operations of a concurrent system, which induces an equivalence relation between computations of the system (cf. Section 4.1.1). The term “partial-order reduction” then stems from the fact that equivalence classes of computations are actually partial orders of operation occurrences.

Definition 7.4
Let $S = (Q, q^0, [], [])$ be a concurrent system. A dependency relation for $S$ is a reflexive and symmetric relation $D [] []$ such that for all $a, b [] [], (a, b) [] D$ implies that the following two properties hold for each $q [] Q$:

1) if $a [] X(q)$ and $q \neq q[]$ then $b [] X(q)$ if $b [] X(q[])$;
2) if $a, b [] X(q)$, then there exists a unique state $q[]$ such that $q \not\vDash q[]$ and $q \vDash q[]$

Two operations $a, b [] []$ are dependent iff $(a, b) [] D$; otherwise, they are independent.

The first requirement listed in Definition 7.4 states that independent operations can neither enable nor disable each other, and the second requirement states that executing independent operations is commutative. The definition itself may at first seem of no more than semantic use, since it is not practical to check these requirements for every pair of operations and for every state of a concurrent system. However, in practice it is possible indeed to give easily checkable syntactic conditions that are sufficient for operations to be independent [HP95, God96]. For instance, two operations from the same sequential process can generally not be independent. If the two operations are defined in sequence, executing the first one will enable the other. If they appear together is a single selection, executing either operation will disable the other. Operations from distinct sequential processes can be independent under certain conditions. Two operations from distinct sequential processes that access only local variables within each process will in general be independent. Two send or receive operations on distinct message queues are usually also independent, but two such operations on the same queue need not be. For two send operations on the same queue, the operation that is executed first may disable the second if it yields a full queue, or the execution order of the two operations may be distinguished by the order in which the messages sent appear in the destination queue, which violates the commutativity requirement. For two receive operations on the same queue, the operation that is executed first may disable the second if it yields a empty queue. For a send and receive operation on the same queue, the send operation may enable the receive operation if the queue is currently empty, or vice versa if the queue is full. Various ways to refine dependency relations, in order to increase the number of pairs of independent operations, can be found in [KP92b, Val92, GP93, God96]. A particularly evident way is to define them as being conditional
upon states: instead of defining a dependency relation that holds for all (reachable) states of a system, it is possible to define such relation for each state individually [KP92b]. Definition 7.4 then becomes as follows.

**Definition 7.4bis**

Let \( S = (Q, q^0, \square, \square) \) be a concurrent system. A relation \( D \quad \square \quad D \quad Q \) is a *conditional dependency relation* for \( S \) iff for all \( a, b \square \) and \( q \square Q \), \((a, b, q) \square D\) implies that \((b, a, q) \square D\), and that the following two properties hold in state \( q \):

i) if \( a \square X(q) \) and \( q \not\equiv_b q \square \) then \( b \square X(q) \) iff \( b \square X(q) \);

ii) if \( a, b \square X(q) \), then there exists a unique state \( q \square \) such that \( q \not\equiv_b^* q \square \) and \( q \not\equiv_b^* q \square \)

Two operations \( a, b \square \) are dependent in a state \( q \square Q \) iff \((a, b, q) \square D\); otherwise, they are independent in \( q \).

For ease of presentation, we will adhere to the use of a binary, unconditional dependency relation between operations as in Definition 7.4. Nevertheless, all what follows in the rest of this chapter is valid also with a conditional dependency relation, and can readily be interpreted in that context.

**Definition 7.5**

Let \( S = (Q, q^0, \square, \square) \) be a concurrent system. Two *finite* sequences of operations \( \square \), \( \square \square \) are equivalent with respect to (wrt) a dependency relation \( D \) for \( S \), denoted by \( \square \equiv_D \square \square \) iff there exist sequences \( \square_1, \square_2, \ldots, \square_k \) such that \( \square_1 = \square, \square_k = \square \) and for all \( 1 \leq i < k \), \( \square_i = \square ab \) and \( \square_i+1 = \square ba \), for some \( \square \) and \( \square \square \) and \( a, b \square \) with \( a \) and \( b \) independent wrt \( D \) (in the states where they are permuted, in case of a conditional dependency relation \( D \)).

For \( \square, \square \square \equiv^* \square, \square \) (i.e. \( \square \square \) and \( \square \) can be finite or infinite), define \( \square \leq_D \square \square \) iff for all \( \square \square \not\equiv_D \) there exist \( \square \square \not\equiv_D \square \square \) such that \( \square \equiv_D \square \square \equiv_D \square \square \) and \( \square \square \equiv_D \square \square \equiv_D \square \square \), where \( \equiv_D \) is the set of finite prefixes of a (finite or infinite) sequence of operations \( \square \). Two infinite sequences of operations \( \square, \square \square \) are equivalent wrt \( D \) iff \( \square \leq_D \square \square \) and \( \square \leq_D \square \).

Intuitively, two sequences of operations are equivalent (wrt a given dependency relation) if one sequence can be obtained from the other by repeatedly permuting adjacent independent operations [Maz86]. The extension of the equivalence relation ‘\( \equiv_D \)’ to infinite sequences of operations in the definition above is adopted from [Pel96]. Equivalence classes induced by ‘\( \equiv_D \)’ are also called *traces* [Maz86], and traces consisting of computations (i.e. sequences of operations that are maximal) of a concurrent system are sometimes referred to as *runs* of the system. For any finite sequence of operations \( \square \), it follows readily from Definition 7.4 that all sequences of operations equivalent to \( \square \) lead to the same state (cf. Proposition 4.3).
Since equivalent computations of a concurrent system differ only in the order of independent, commutative operations, it appears not necessary in general to examine all computations in order to verify the system against various desirable properties. It is instead sufficient for many properties to examine just one representative computation per equivalence class of computations. Accordingly, partial-order reduction methods attempt as much as possible to fix an order among independent operations, by executing at each state encountered during state exploration only a subset of the operations executable at that state, rather than all of them. State exploration is thereby performed usually via a depth-first search, or some variation of it. Some conditions apply to selecting a subset of the executable operations at a given state, which must guarantee that the reduced part of the reachable state space of a concurrent system that is explored by a partial-order reduction method preserves the property being checked. Devised by different researchers, such subsets adhere to different names: stubborn sets [Val90, Val92, Val93], persistent sets [God90, GW93, GW94], faithful decompositions [KP92] or ample sets [HP95, Pel96]. Although the definitions of these sets and the associated algorithms differ, they do have much in common [God96] and are therefore referred to collectively as partial-order reduction methods.

For the course of this chapter it would be futile to try to capture all the subtleties of the various suggested partial-order reduction methods, some of which take advantage of confining themselves to a restricted class of properties. We will describe, and subsequently enhance, the partial-order reduction method based on ample sets [HP95, Pel96]. This particular method has been proposed most recently, and it is fairly generic in the sense that it can be adapted without too much difficulty to resemble the other partial-order reduction methods. Furthermore, it is advocated as the most advanced partial-order reduction method in terms of the properties that can be checked, the way fairness is dealt with, and the low overhead and high overall performance of its implementation [HP95, Pel96]. The method has been implemented as an extension to SPIN, a verification tool which is increasingly being used for teaching and for industrial applications [Hol91, SPIN95, SPIN96, SPIN97]. For convenience, the partial-order reduction method based on ample sets will be referred to as POVAS (Partial-Order Verification with Ample Sets).

POVAS is intended as a relief strategy for verifying concurrent systems against properties formalized by nexttime-free LTL formulas, i.e. for nexttime-free LTL model-checking. It comes in four different “modes”, depending on whether model-checking is done off-line or on-the-fly, and with or without certain fairness assumptions. We focus primarily on the off-line and on-the-fly versions without fairness assumptions. Model-checking under fairness assumptions with POVAS is conceptually not much different, and will be addressed later in Section 7.3.
### 7.2.1 Off-line LTL model-checking with POVAS

POVAS implements a depth-first search (DFS) algorithm which, in contrast to a classical brute-force DFS, incorporates three conditions for selecting a *subset* of the operations that are to be executed at a given state encountered during state exploration. For the off-line version in particular, when a state \( q \) on the DFS stack is expanded and at least one operation is executable at \( q \), a non-empty subset of \( X(q) \) is used to generate successor states for \( q \) in accordance with the following definition [Pel96].

**Definition 7.6**

Let \( S = (Q, q^0, \square, \Diamond) \) be a concurrent system, \( D \) a dependency relation for \( S, f \) an LTL formula to be checked for \( S \), and \( q \square Q \) the current state to be expanded during the DFS. An *ample set* in \( q \) is a non-empty subset \( A \square X(q) \) of operations executable at \( q \) satisfying the following three conditions:

1. For every non-empty sequence \( q_1 \Diamond \Diamond q_2 \Diamond \Diamond \ldots \Diamond q_k \Diamond \Diamond q_{k+1} \) from \( q_1 = q \), with \( a_i \Diamond \Diamond \Diamond A \) for all \( 1 \leq i \leq k \), each operation \( a_i \) is independent wrt \( D \) (in \( q_1 \), in case of a conditional dependency relation \( D \)) of all operations in \( A \);
2. If \( A \square X(q) \), then no operation \( a \square A \) with \( q \trianglelefteq a \square q \) is such that \( q \square q \) is on the current DFS stack;
3. If \( A \square X(q) \), then \( A \square \text{vis}_f(S) = \emptyset \).

An ample set in \( q \) is denoted by \( \text{ample}(q) \).

Note that the set \( X(q) \) itself is trivially an ample set in state \( q \). In the sequel, we refer to the three conditions on ample sets in Definition 7.6 as conditions **C1**, **C2** and **C3**, respectively. The first condition **C1** is a consistency requirement. It guarantees that after state \( q \) is reached, no operation outside \( \text{ample}(q) \) that is dependent of an operation in \( \text{ample}(q) \) can be executed before an operation in \( \text{ample}(q) \) is executed. Equivalently, every single operation outside \( \text{ample}(q) \) is either independent (in \( q \) of all operations in \( \text{ample}(q) \)), or it is not executable at \( q \) and at every state that can be reached from \( q \) without executing an operation in \( \text{ample}(q) \). The execution at \( q \) of only the operations in \( \text{ample}(q) \) does therefore not affect “negatively” the executability of any operation outside \( \text{ample}(q) \), for operations outside \( \text{ample}(q) \) which are already executable at \( q \) remain executable, while those not executable at \( q \) can “only” become executable. Condition **C2** is enforced to avoid the ignoring problem, which may cause the execution of operations to be deferred indefinitely along a cycle. We have already addressed this problem in Chapter 5 in the context of LRA for verifying logical correctness properties of protocols in the CFSM model. It is of the same nature here: condition **C2** guarantees that a state \( q \) is fully expanded (i.e. \( \text{ample}(q) = X(q) \)) whenever one of the operations in \( \text{ample}(q) \) closes a cycle on the DFS stack, thereby providing an exit from this cycle if
one exists (cf. Section 5.5). Lastly, condition C3 is enforced in view of the fact that the checked LTL formula may very well be sensitive to the order of two visible operations of the concurrent system (i.e. operations that can affect the truth value of the formula), even when these operations are mutually independent. The effect of not allowing visible operations in ample(q), in case it is a proper subset of X(q), is that all the possible execution orders of all visible operations will be explored. Every two visible operations are then essentially treated as being always dependent. In this regard, condition C3 actually justifies why POVAS should be restricted to LTL formulas that are nexttime-free or, in general, to temporal properties that are stuttering closed (see Section 7.1.2). Although POVAS can in principle also handle LTL formulas that do contain a “next-time” operator, such formulas generally cause all the operations of a concurrent system to be visible, and hence they would all be considered as dependent. Each state encountered during state exploration by POVAS would then be fully expanded, exactly as in a classical DFS, which annihilates any benefit coming from the use of POVAS. In the remainder of this chapter, we implicitly mean nexttime-free LTL when we refer to LTL.

The algorithm presented in [HP95, Pel96] for calculating ample sets is as follows. Based on the fact that operations of a single sequential process can generally not be independent, it seeks some process in a concurrent system whose set of operations executable at the current state q satisfy conditions C1 to C3. As soon as such a process Pi is found, the set Xi(q) is returned as ample(q). If no such process exists, the algorithm returns the entire set X(q) of all operations executable at q. Condition C2 must be checked during state exploration by inspecting the current contents of the DFS stack. However, most of the information required for checking C1 and C3 is gathered efficiently by a static analysis of the concurrent system before state exploration [HP95]. That is, during system compilation each local state of each sequential process is analyzed and annotated with one of three types of labels: safe, conditionally safe upon some condition, or unsafe. These labels signify whether at run time (i.e. during state exploration), when a system state q is expanded and some process is at its local state l, the set of operations of this process that are defined at l and executable at q satisfies conditions C1 and C3. A local state l of a process is labeled “safe” if it is determined at compile time that the set of (executable) operations defined at l will qualify as an ample set. This would be the case if all these operations are invisible, and if they are independent of every operation belonging to another process. Recall from Section 7.1.3 that (an upper approximation of) the set visl(S) of visible operations of a concurrent system is itself computed statically [Val92]. Analogously, l is labeled “unsafe” if it is already decided at compile time that no ample set can be formed from the operations defined at l. The local state l is labeled “conditionally safe” for some precomputed condition C (which is one out of a small number of conditions [Pel96]) when the operations defined at l form an ample set only if C holds during run time. For example, if only send operations are defined at l, such a condition can be that none of the
corresponding queues are filled to their capacity.

Several other algorithms have been proposed earlier to compute sets of operations that satisfy in particular condition $C1^6$. An overview and a comparison of these algorithms can be found in [God96], all of which also infer such sets from the syntactic structure of a concurrent system, but not statically as in the algorithm described above. They further differ from the algorithm in [HP95, Pel96] in their aim to compute the smallest sets of operations satisfying $C1$. Typically, the more information (static or dynamic) used, the smaller these sets can be, but at the cost of an increased computational complexity. Moreover, exploring the smallest number of operations at each step during state exploration is only a heuristic: it does not necessarily lead to the generation, and hence the storage, of the smallest number of states. For these reasons, checking $C1$ in the ample set algorithm is based on a more delicate trade-off between the storage space and the overall execution time needed for model-checking [HP95, Pel96].

To sum up, off-line model-checking with POVAS proceeds as a “selective” DFS, using the above algorithm for calculating ample sets to determine for each state encountered during the DFS the subset of successor states that need be expanded next. As a result, it explores only a reduced part of the reachable state space of a concurrent system $S$. Precisely, instead of building the full automaton $A_S = (Q_S, q_S^0, \bar{Q}_S, \bar{q}_S, Q_S)$ for $S$, it builds a reduced automaton $A^C_S$ for $S$ defined by the tuple $(Q^C, q^C_S, \bar{Q}^C, \bar{q}^C_S, Q^C)$, where $Q^C \subseteq Q_S$ and $\bar{Q}^C \subseteq \bar{Q}_S$ such that $(q, a, q\bar{q}) \in \bar{Q}^C$ iff $a$ ample($q$) (as opposed to $a X(q)$) and $q \not\in Q^C$. This reduced automaton $A^C_S$ preserves all non-progress states of $S$ (states at which no operation is executable), and it reveals all reachable local (or process) states and thus all non-executable operations (operations that are not executable at any reachable state of $S$). Moreover, for every computation $\bar{s}$ of $S$ (or equivalently of $A_S$), there is at least one computation $\bar{q}\bar{q}$ of $A^C_S$ such that $\bar{s}$ and $\bar{q}\bar{q}$ are stuttering equivalent [HP95, Pel96]. Hence, when a temporal property is closed under stuttering, the property holds true for (all computations of) $S$ iff it holds true for all the computations of $A^C_S$. Since all nexttime-free LTL formulas are closed under stuttering, algorithms for off-line LTL model-checking [LP85] can be applied directly to $A^C_S$, rather than to the full automaton $A_S$, in order to verify these formulas.

### 7.2.2 On-the-fly LTL model-checking with POVAS

POVAS can readily be combined with on-the-fly LTL model-checking in order to gain from both. Verifying an LTL formula $f$ for a concurrent system $S$ then involves constructing the product of the reduced automaton $A^C_S$ for $S$ and the Büchi automaton $A_{\neg f}$ for the negation of $f$, and checking its emptiness. This can again be done by seeking acceptance cycles in $A^C_S \bigcap A_{\neg f}$, similar as explained

---

6 Condition $C1$ is in fact the basic consistency requirement that underlies also the other partial-order reduction methods based on stubborn sets, persistent sets, or faithful decompositions.
before in Section 7.1.3. POVAS utilizes the nested DFS algorithm given in [CV+92], or actually a slight modification of it to ensure compatibility with the selective DFS algorithm based on ample sets [HPY96] (the need for this modification was not yet recognized in [HP95, Pel96], but the authors proposed the correction in [HPY96]).

For the on-the-fly version of POVAS, the calculation of ample sets itself also undergoes a minor change so that it applies to composite states of the product automaton \( A_f \) instead of single system states. When a composite state \((q, r)\) of \( A_{f} \) on the DFS stack is expanded and at least one operation is executable at system state \( q \), a non-empty subset \( \text{ample}(q, r) \) of \( X(q) \) is employed to generate successor states for \((q, r)\), which satisfies the earlier conditions \( \text{C1} \) and \( \text{C3} \) in Definition 7.6 and the new condition \( \text{C2}[\text{Pel96}] \):

\[
\text{C2} \quad \text{if } \text{ample}(q, r) \subseteq X(q), \text{ then no operation } a \subseteq \text{ample}(q, r) \text{ with } q \not\preceq q \text{ is such that the composite state } (q[r]) \text{ is on the current DFS stack.}
\]

Conditions \( \text{C1} \) and \( \text{C3} \) remain unaffected because the dependency relation \( D \) and the set \( \text{vis}_{f}(S) \) of visible operations are irrespective of the state of the Büchi automaton \( A_{f} \). Condition \( \text{C2} \) entails inspecting the DFS stack and must be adapted in particular because each system state may yield several composite states that differ in the state of the Büchi automaton \( A_{f} \). That is, the on-the-fly construction of \( A_{f} \) operates on a different DFS stack and may postpone the closing of cycles compared to the off-line construction of \( A_{f} \) [Pel96]. The new condition \( \text{C2}[\text{Pel96}] \) appears sufficient to guarantee that the modified version [HPY96] of the nested DFS algorithm in [CV+92], with the calculation of ample sets to determine successor states, detects at least one acceptance cycle on-the-fly in \( A_{f} \) if one or more such cycles exist in the “full product” \( A_{f} \). Thus, model-checking the LTL formula \( f \) on-the-fly with POVAS will yield a correct and conclusive verdict: one either finds a counter example to \( f \) (i.e. an acceptance cycle), or else \( f \) is satisfied by concurrent system \( S \).

**Example 7.7**

Consider again the concurrent system \( S \) and the LTL formula \( f \) described in Example 7.3, and depicted in Figure 7.2. Recall that the two sequential processes \( P_1 \) and \( P_2 \) execute autonomously, meaning that each operation of \( P_1 \) is always independent of each operation of \( P_2 \). Also recall that the operations \( w \) and \( d \) are the only visible operations of \( S \). The product automaton \( A_{f} \) obtained with POVAS, using the algorithm suggested in [HP95, Pel96] for calculating ample sets, is given in Figure 7.3. It is non-empty, like the “full product” \( A_{f} \) shown in Figure 7.2, but it contains less states and transitions. For instance, at the initial composite state \((10, 20, 0)\) only operation \( a \) of \( P_1 \) is executed since \( X_1((10, 20)) = \{a\} \) is an ample set in this state, i.e. \( \{a\} \) satisfies conditions \( \text{C1}, \text{C2} \) and \( \text{C3} \). This is not the case for \( X_1((11, 20)) = \{b, c\} \) in the subsequent
composite state \((11, 20, 0)\), which violates \(C_2\) since the execution of operation \(b\) leads back to the initial state \((10, 20, 0)\) that is on the current DFS stack. However, \(X_2((11, 20)) = \{v\}\) is now an ample set in \((11, 20, 0)\). Note further that, amongst others, the state \((12, 21, 0)\) is fully expanded as both \(w\) and \(d\) are visible operations and thus neither \(\{w\}\) nor \(\{d\}\) is an ample set in this state.

\[
\begin{array}{c}
(10, 20, 0) \\
\downarrow a \\
(11, 20, 0) \\
\downarrow v \\
(11, 21, 0) \\
\downarrow a \\
(10, 21, 0) \\
\downarrow w \\
(10, 22, 1) \\
\downarrow b \\
(11, 22, 1) \\
\downarrow c \\
(12, 22, 1) \\
\downarrow d \\
(12, 22, 1) \\
\end{array}
\]

\[
\begin{array}{c}
(12, 21, 0) \\
\downarrow d \\
(12, 22, 1) \\
\end{array}
\]

\[
\begin{array}{c}
(13, 21, 2) \\
\downarrow a \\
(12, 21, 0) \\
\downarrow w \\
(12, 22, 1) \\
\end{array}
\]

\[
\begin{array}{c}
(13, 22, 2) \\
\downarrow d \\
(12, 22, 1) \\
\end{array}
\]

\[\]

\[\]

\[
\]

\[\]

\[
\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]

\[\]
contain visible operations. The operations are dependent if an operation in $A$ is disjoint, this yields the contradiction that $A$ is mutually dependent. Proposition 7.8 proves these claims.

Proof: For the first claim, suppose there exist operations $a_1$ and $a_2$ such that $a_1$ and $a_2$ are mutually dependent (in $q$, in case of a conditional dependency relation). Since $a_1$ and $a_2$ are disjoint, this yields the contradiction that $A$ do not satisfy condition $C1$: $a_2$ is disjoint to $a_1$ does not contain visible operations.

7.3.1 Proper leap sets

The key to enhancing POVAS lies in a rather simple observation: states of a concurrent system may have multiple disjoint ample sets. For instance, a state $q$ has multiple disjoint ample sets particularly when there is more than one sequential process $P_i$ whose set $X(q)$ is non-empty and satisfies the conditions $C1$, $C2$, and $C3$ (which is the case in the above example). Two disjoint ample sets in the same state have the nice characteristic that no operation in either ample set can be dependent of an operation in the other ample set. Furthermore, two disjoint ample sets in the same state cannot contain visible operations. Proposition 7.8 proves these claims.

Proposition 7.8

Let $q$ be a state of a concurrent system $S$, with $X(q) \neq \emptyset$, and let $A_1$ and $A_2$ be sets of operations of $S$ such that $\emptyset \mathcal{A} A_1, A_2 \mathcal{\mid X(q)}$ and $A_1 \mathcal{\mid A_2} = \emptyset$. If $A_1$ and $A_2$ satisfy condition $C1$, then all operations in $A_1$ are independent (in $q$) of all operations in $A_2$. If $A_1$ and $A_2$ satisfy condition $C3$, then all operations in $A_1$ and $A_2$ are invisible.

Proof: For the first claim, suppose there exist operations $a_1 \mathcal{\mid A_1}$ and $a_2 \mathcal{\mid A_2}$ such that $a_1$ and $a_2$ are mutually dependent (in $q$, in case of a conditional dependency relation). Since $a_1$ and $a_2$ are disjoint, this yields the contradiction that $A_1$ and $A_2$ do not satisfy condition $C1$: $a_2 \mathcal{\mid A_1}$ (or $a_1 \mathcal{\mid A_2}$) is dependent of an operation in $A_1$ (or $A_2$) but can be executed at $q$ before an operation from $A_1$ (or $A_2$) is executed (cf. Definition 7.6). For the second claim, both $A_1$ and $A_2$ must be (non-empty) proper subsets of $X(q)$ since otherwise they cannot be disjoint. Thus, by condition $C3$, $A_1$ and $A_2$ do not contain visible operations.
Formally, we employ the following definition.

To exploit the possible existence of multiple pairwise disjoint ample sets in a system state \( q \), or actually the existence of disjoint subsets of \( X(q) \) that satisfy \( C1 \) and \( C3 \), we adopt the algorithm in Figure 7.4 for calculating such sets. This algorithm differs from the one given in [HP95, Pel96] in two ways. First, it enforces only conditions \( C1 \) and \( C3 \) on ample sets. The reason for omitting \( C2 \) will become clear in the next subsection. Secondly, it returns the set of all sets \( X_i(q) \) satisfying \( C1 \) and \( C3 \), or the empty set if no such \( X_i(q) \) exists. Thus, we calculate only nontrivial ample sets with respect to \( C1 \) and \( C3 \). Our algorithm is a straightforward adaptation of the algorithm in [HP95, Pel96] for finding just one ample set, and it does not introduce significant extra overhead. This is true especially since \( C1 \) and \( C3 \) are already checked for the most part statically by a prescan of the sequential processes during system compilation, as described in Section 7.2.1. Both algorithms then have a time complexity linear in the number of processes, since the algorithm in [HP95, Pel96] must also scan all processes in the worst case.

Let \( k \) be the number of ample sets, with respect to \( C1 \) and \( C3 \), returned by the algorithm in Figure 7.4 (i.e. \( k \) is the cardinality of the returned set \( AS \) in Figure 7.4 and \( 0 \sqcup k \sqcup 0 \) the number of sequential processes). When \( k > 0 \), each such ample set is a subset of \( X(q) \) satisfying \( C1 \) and \( C3 \) and thus, by Proposition 7.8, all its operations are invisible and independent (in \( q \)) of all operations in the other ample sets. This then motivates that any collection of executable operations forming an element of the Cartesian product of the \( k \) ample sets can be executed concurrently at state \( q \). When \( k = 0 \), no appropriate proper subset of \( X(q) \) has been found and all operations executable at \( q \) are to be executed separately. That is, the state \( q \) is then fully expanded as is the case in [HP95, Pel96]. Formally, we employ the following definition.

\[
AS = \emptyset
\]

for all processes \( P_i \) do

if \( X_i(q) \neq \emptyset \) then {

\( C3 = (X_i(q) \cap vis_i(S) = \emptyset) \)

if \( C3 \) then {

\( C1 = check_C1(X_i(q)) \)

/* check_C1(X_i(q)) returns true if \( X_i(q) \) satisfies \( C1 \), */

/* and false otherwise (see Section 7.2.1). */

if \( C1 \) then add the set \( X_i(q) \) to \( AS \)

}

return \( AS \)

Figure 7.4 Finding multiple disjoint ample sets (wrt \( C1 \) and \( C3 \)).
Definition 7.9
Let $S = (Q, q^0, \square, \lozenge)$ be a concurrent system, and for some $q \square Q$ let $ample_1(q), ample_2(q), \ldots, \ ample_k(q)$ denote the $k$ disjoint subsets of $X(q)$ satisfying conditions $C1$ and $C3$ that are returned by the algorithm in Figure 7.4. The set $pleap(q)$ of proper leap sets in $q$ is defined as follows:

$$pleap(q) = \square^k j \in ample_j(q) \quad \text{if } k > 0$$

$$pleap(q) = \{ \{a\} \mid a \square X(q) \} \quad \text{if } k = 0$$

When $k = 0$, all operations executable at $q$ are thus considered individually by including them in $pleap(q)$ in the form of singleton sets, like in Chapter 5.

We have designated the term “proper leap set” and the corresponding set denotation $pleap(q)$ to comply with the terminology in Chapter 5. In further analogy with Chapter 5, a permutation of the operations in a proper leap set $T$ is called a linearization of $T$, and the set of all linearizations of $T$ is denoted by $lin(T)$. For a finite or infinite sequence of proper leap sets $\square = T_1 T_2 \ldots$ we have $lin(\square) = \{ \square_1 \square_2 \ldots \mid \square \square lin(T_i) \}$ for all $i \geq 1$. Since all operations in a proper leap set are mutually independent, it follows from Definition 7.4 that all its linearizations are equivalent and lead to the same state. Hence, we write $q \square_0^0 q \square_0$ to mean that there is a set $T \square pleap(q)$ with $\square \square lin(T)$ such that $q \square_0^0 q \square_0$ and $q \square_0^0 q \square_0$. This so-called leap automaton for $S$ is adequate for verifying indefinite progress, but it does not yet lend itself for LTL model-checking. We will come back to this shortly. Let us first show that exploring $A^c_\ell$ indeed reveals all non-progress states of $S$.

Theorem 7.10 below proves that for every terminating computation $\square$ of a concurrent system $S$ there is at least one sequence of proper leap sets in $A^c_\ell$, starting from the initial state of $S$, whose linearizations are equivalent (wrt any dependency relation $D$) to $\square$. This directly implies that every non-progress state of $S$ is a state in the leap automaton $A^c_\ell$.

Theorem 7.10
Let $S = (Q, q^0, \square, \lozenge)$ be a concurrent system and $D$ a dependency relation for $S$. For every finite computation $\square \square \square$ of $S$, there exists a sequence of proper leap sets $\square$ in $A^c_\ell$ from the initial state $q^0$ of $S$, with $\square \square lin(\square)$, such that $\square \equiv_D \square$. 

---

7 This theorem should be compared with Lemma 5.10 and Theorem 5.11 in Chapter 5.
Proof: Let \( q^0 \models \Box_0^* q \), then \( X(q) = \emptyset \) by Definition 7.1 (i.e. computations of \( S \) are defined to be maximal). We first prove that there exists a proper leap set \( T_1 \) in the initial state \( q^0 \) of \( S \), with \( \Box_1 \models \text{lin}(T_1) \), such that \( \Box_1 \preceq_D \Box_2 \) (see Definition 7.5). If \( \text{pleap}(q^0) = \{ \{ a \} | a \Box X(q^0) \} \) then \( T_1 \) exists trivially, namely \( T_1 \) is the singleton set containing the first operation of \( \Box \). If \( \text{pleap}(q^0) = \bigcup_{j=1}^\infty \text{ample}_j(q^0) \), then by condition C1 and the fact that \( \Box \) is a terminating computation, it follows that for each process \( P_i \) with \( X_i(q) = \text{ample}_j(q) \) there exists an operation from \( \text{ample}_j(q) \) in \( \Box \) (since otherwise \( X(q) \neq \emptyset \)). Let \( a_j \) denote the first operation in \( \Box \) from \( \text{ample}_j(q) \), for each \( 1 \leq j \leq k \). Clearly, \( \{ a_1, a_2, \ldots, a_k \} \models \text{pleap}(q^0) \). By C1, each \( a_j \) is independent of all operations that occur in \( \Box \) before the occurrence of \( a_j \) itself, and hence the operations \( a_1, a_2, \ldots, a_k \) can all be permuted to the front of \( \Box \). Consequently, in this case let \( T_1 = \{ a_1, a_2, \ldots, a_k \} \) with \( \Box_1 \models \text{lin}(T_1) \), then for some \( \Box \) we have \( \Box = D \boxdot \Box_1 \) and thus \( \Box_1 \preceq_D \Box \). Let then \( q^0 \models \Box_{\Box_0^*} q^1 \models \Box_{\Box_0^*} q \). The proof of the theorem is now straightforward by finite repetition of the above reasoning, continuing with \( q^1 \models \Box_{\Box_0^*} q \) (cf. the proof of Theorem 5.11).

One should realize that the detection of non-progress states does not require the specification of a temporal formula, meaning that condition C3 on ample sets is actually void. Indeed, this condition is not used in the proof of Theorem 7.10. It is clear that enforcing just condition C1 generally aids the calculation of larger proper leap sets, which may result in a further reduction of the number of states and transitions explored.

7.3.2 Off-line LTL model-checking with (proper) leap sets

As just mentioned, a DFS that governs the execution of proper leap sets is not yet fit for LTL model-checking. It should be no surprise that the reason for this is that we have thus far omitted condition C2 on ample sets. Recall from Section 7.2.1 that this condition is enforced to avoid the ignoring problem, which may cause the execution of operations to be deferred indefinitely along a cycle. Condition C2 prohibits any operation in \( \text{ample}(q) \) from closing a cycle on the DFS stack in case \( \text{ample}(q) \) is a proper subset of \( X(q) \). However, incorporating C2 directly in the formulation of proper leap sets does not solve the ignoring problem for a DFS that governs the execution of these sets, because it employs a different DFS stack (see also Section 7.2.2). We enforce the following condition on proper leap sets instead. Denote by \( \text{op}(\text{pleap}(q)) \) the set of all operations in all proper leap sets in \( q \), i.e. \( \text{op}(\text{pleap}(q)) = \{ a \Box T | T \Box \text{pleap}(q) \} \). If \( \text{op}(\text{pleap}(q)) \) is a proper subset of \( X(q) \), then for no proper leap set \( T \Box \text{pleap}(q) \) it should hold that the execution of \( T \) at \( q \) leads to a state that is already on the DFS stack. Otherwise, \( \text{pleap}(q) \) is extended by adding to it all sets \( T \Box \{ a \} \), where \( T \) ranges over the sets in \( \text{pleap}(q) \) that do lead to a state on the DFS stack, and \( a \) ranges over the operations in \( X(q) \) that are not in \( \text{op}(\text{pleap}(q)) \).
Definition 7.11
Let $S = (Q, q^0, \emptyset, \emptyset)$ be a concurrent system, and let $q \in Q$ be the current state to be expanded during the DFS. The set $\text{pleap}(q)$ is defined as follows:

$$\text{xpleap}(q) = \text{pleap}(q) \cup \{ T \mid \{a \mid a \in X(q) \setminus \text{op}(\text{pleap}(q)) \} \text{ and } T \text{\pleap}(q) : q \xrightarrow{a} q\text{ is on the DFS stack} \}$$

Observe that each operation (if any) in $X(q) \setminus \text{op}(\text{pleap}(q))$ belongs to some sequential process $P_i$ whose set $X_i(q)$ is non-empty but does not satisfy $\text{C1}$ or $\text{C3}$. Thus, these operations can easily be determined also with the algorithm in Figure 7.4 (only a few straightforward statements need be added to the algorithm), i.e. together with the search for multiple disjoint ample sets that satisfy $\text{C1}$ and $\text{C3}$. It is further important to stress that we extend $\text{pleap}(q)$ rather than returning all singleton sets of executable operations, which would reflect the calculation of ample sets in [HP95, Pel96]. In this way the calculation of $\text{pleap}(q)$ is not wasted when it turns out that the execution of some proper leap set leads back to a state on the DFS stack. Lastly, observe that for each set $T \text{\pleap}(q)$ all operations in $T$ are mutually independent and at most one of these operations is visible, by the construction and condition $\text{C1}$.

Analogous to the leap automaton $A_S^c$ for a concurrent system $S$, an extended leap automaton $A_S^c$ for $S$ can be defined on the basis of the execution of elements from the extended set $\text{xpleap}(q)$.

Since $\text{xpleap}(q) \supseteq \text{pleap}(q)$ for any state $q$, $A_S^c$ strictly extends $A_S^c$. That is, each state of $A_S^c$ is a state of $A_S^c$ and each transition of $A_S^c$ is a transition of $A_S^c$. This extension solves the ignoring problem as it causes every cycle in $A_S^c$ (when viewed as a graph) to contain at least one state $q$ for which $\text{op}(\text{xpleap}(q)) = X(q)$, which is not necessarily the case for $\text{pleap}(q)$ (cf. Section 5.5 in Chapter 5).

Henceforth, the term “leap set” refers to an element of $\text{xpleap}(q)$ (in state $q$). As before, $\text{lin}()$ denotes the set of all linearizations of a (finite or infinite) sequence $\emptyset$ of leap sets. In addition, $q! \text{\pleap} q\emptyset$ denotes that (any linearization of) the leap set $T \text{\pleap}(q)$ leads from state $q$ to state $q\emptyset$ and likewise $q \xrightarrow{a} q\emptyset$ in case of a sequence of leap sets $\emptyset$. Remark that all linearizations of a sequence of leap sets from the initial state of $S$ are computations of $S$. Hence, like for computations of $S$, an infinite sequence of leap sets $\emptyset$ from the initial state of $S$ is said to satisfy an LTL formula $f$ iff the Büchi automaton $A_f$ accepts the computation of $A_S^c$ over $\emptyset$ (i.e. the sequence of states corresponding to $\emptyset$, see Section 7.1.3). Recall thereby once more that finite sequences of leap sets are taken into account by repeating forever the last states generated by these sequences. Since any leap set contains at most one operation from $\text{vis}_f(S)$, all visible operations in $\emptyset$ appear in the same order in each linearization of $\emptyset$, while the invisible operations correspond to stuttering steps. Therefore, either all or none of the linearizations of $\emptyset$ satisfy $f$ and, moreover, $\emptyset$ itself satisfies $f$ iff all linearizations of $\emptyset$ satisfy $f$. 
Let $D \equiv D \cap (vis_f(S) \cap vis_f(S))$ be the dependency relation $D$ of a concurrent system $S$ augmented with dependencies between all the visible operations for the checked formula $f$. This makes $f$ equivalence robust [Pel93, Pel96]: all sequences equivalent wrt $D$ contain the same visible operations and in the same order, and thus $f$ has the same truth value for each of them (i.e. either all or none of these sequences satisfy $f$). Surely, all linearizations of a sequence of leap sets are equivalent wrt $D$. The dependency relation $D \equiv$ is used specifically to show that a DFS governing the execution of leap sets retains the order of visible operations in the computations of a concurrent system (Lemma 7.13 and Lemma 7.14 below). This in turn is the key to proving the main result of the chapter: LTL model-checking can be conducted faithfully with the extended leap automaton $A^c_S$ for $S$ (Theorem 7.15 below). The associated proofs are somewhat similar to the proofs of the corresponding claims in [Pel96]. For convenience, the next definition is therefore taken directly from [Pel96].

**Definition 7.12**

Let $S = (Q, q^0, [\ ], [])$ be a concurrent system and $D$ a dependency relation for $S$. For a (finite or infinite) sequence $[\ ] [\ ] [\ ]$ of operations of $S$, denote by $op([\ ])$ the set of operations occurring in $[\ ]$ by $(i)$ the $i$-th operation in $[\ ]$, and by $(i+1\ldots)$ all but the first $i$ operations in $[\ ]$. A selection function for $[\ ]$ is a function $c : \{1,\ldots, [\ ]\} \mapsto \{true, false\}$, mapping each operation in $[\ ]$ to either true or false. Denote by $[\ ]_c (i)$ the sequence remaining from $[\ ]$ after the removal of all operations $[\ ](i)$ with $c(i) = false$ ($c(i) = true$). Also, denote by $c \rightarrow r$ the selection function $c$ shifted to the left $r$ places, i.e. $c \rightarrow r(i) = c(i+r)$. Define $[\ ] \preceq^D [\ ]$ iff there exists a selection function $c$ for $[\ ]$ such that (i) $[\ ] = [\ ]_c (i)$, (ii) $op([\ ]_c (i)) \cap A = [\ ]$, and (iii) for all $1 \leq i \leq [\ ]$, if $c(i) = false$ then each operation in $op([\ ]_c (i+1\ldots))$ is independent wrt $D$ of $[\ ]_c (i)$. 

Informally, $[\ ] \preceq^D [\ ]$ iff a sequence equivalent (wrt $D$) to $[\ ]$ can be obtained from $[\ ]$ by removing from $[\ ]$ some operations in $A$ that are independent of all the non-removed operations of $[\ ]$ that appear after them [Pel96]. For example, let $[\ ] = \{a, b, v, w\}$, $D = \{(a, a), (b, b), (v, v), (w, w)\}$, $[\ ] = abvw(ab)$, $[\ ] = (bwav)\|$ and $A = \{v, w\}$, then $[\ ] \preceq^D [\ ]$. To see this, choose a selection function $c$ for $[\ ]$ such that $c(i) = true$ if $i$ is odd or less than 5; $c(i) = false$ otherwise. One obtains $[\ ] = bwav(ba)\|$, i.e. all occurrences of $v$ and $w$ in $[\ ]$ except the first ones are removed. The equivalence $[\ ] \preceq^D [\ ]$ is then immediate. Notice that every removed operation $v$ or $w$ is independent of all the operations occurring to its right in $[\ ]$ that are not removed (these are the occurrences of $a$ and $b$ except the first $a$ and $b$).

**Lemma 7.13**

For a concurrent system $S = (Q, q^0, [\ ], [])$, let $q \cap Q$ be a state that is removed from the DFS stack during the construction of the extended leap automaton $A^c_S$ for $S$, and let $[\ ]_a [\ ]_b [\ ]_v [\ ]_w$ be a
computation of $S$, with $q^0 \overset{\alpha_0}{\rightarrow}^* q$. Then, there exist a sequence $q \overset{\alpha_0}{\rightarrow} q_1 \overset{\alpha_1}{\rightarrow} \ldots \overset{\alpha_{m-1}}{\rightarrow} q_m \overset{\alpha_m}{\rightarrow} q$ $(m \geq 1)$ in $A^c_S$, and a selection function $c$ for $\square \text{lin}(T_1 T_2 \ldots T_{m-1}(T_m \setminus \{a\}))$, such that

1. $a \not\square T_m$,
2. no operation in $T_1, T_2, \ldots, T_{m-1}, T_m \setminus \{a\}$ is visible,
3. no operation in $T_1, T_2, \ldots, T_{m-1}, T_m \setminus \{a\}$ is dependent wrt $D$ of $a$,
4. $\gamma_\alpha =_{D} \gamma_{\square_\alpha}^\alpha$,
5. $\square_\alpha$: $\square_\alpha =_{D} \square_\alpha$ (i.e. $\square \leq_{\square_\alpha} \square_\alpha$), and
6. the operations in $\square_\alpha$ are independent wrt $D$ of the operations in $\square$.

**Proof:** If $\text{xpleap}(q) = \{ \{ b \} | b \not\square X(q) \}$, then a sequence with the required properties exists trivially, namely $q \overset{\alpha_0}{\rightarrow} q$. Alternatively, if $\text{xpleap}(q) \supseteq \text{ample}(q) = \square_\alpha$ by construction of $\text{ample}(q)$, then for each $\text{ample}(q)$, no operation in $\text{ample}(q)$ is visible (by C3), and no operation outside $\text{ample}(q)$ that is dependent wrt $D$ of an operation in $\text{ample}(q)$ can occur in $a\square$ before the occurrence in $a\square$ of some operation in $\text{ample}(q)$ itself (by C1). The holds then true also wrt $D = D \square (\text{vis}(S) \square \text{vis}(S))$. It follows that for each process $P_i$ such that $X_i(q) = \text{ample}(q)$, either $\text{ample}(q)$ contains the first operation of $P_i$ in $a\square$, or all operations in $a \square$ are independent wrt $D$ of all operations in $\text{ample}(q)$ (i.e. $P_i$ has no operations occurring in $a \square$). Note that the latter cannot be the case if $\square$ is finite (which was in fact the key to proving Theorem 7.10). Thus, there exists $T = \{ a_1, a_2, \ldots, a_k \}$ $\not\square \text{xpleap}(q) \square$ such that $a_j$ is the first operation from $\text{ample}(q)$ occurring in $a \square$ if there is such operation, or else $a_j$ is any operation from $\text{ample}(q)$, for each $1 \not\square j \not\square k$, and all the $a_j$’s are invisible and mutually independent wrt $D$. The proof now continues by induction on the order in which states are removed from the DFS stack during the construction of the extended leap automaton $A_S^c$ for $S$. When removing the state $q$ from the DFS stack, two cases can be distinguished:

- $a \not\square T$ or $T$ leads to a state on the DFS stack

Remark that this case covers in particular the induction basis where $q$ is the first state removed from the DFS stack: each proper leap set in $\text{xpleap}(q)$ executed in $q$ leads to a state that is already on the DFS stack since any other state would have been removed before $q$ (a characteristic of a depth-first search). Let $T_1 = T \square \{ a \}$ if $a \not\square T$ (i.e. $a \not\square X(q) \not\square \text{op}(\text{pleap}(q))$; $T_1 = T$ otherwise. By construction, $a \not\square T_1$ and $T_1 \not\square \text{xpleap}(q)$, and all operations in $T_1 \setminus \{ a \}$ are invisible and independent wrt $D$ of $a$, viz. $q \overset{\alpha_0}{\rightarrow} q$ is a sequence satisfying properties (i), (ii) and (iii). For any sequence of operations $\square$ define the required selection function $c$ such that $c(i) = \text{true}$ if $\square(i) \not\square \text{op}()$; $c(i) = \text{false}$ otherwise. Here, $\square \square \text{lin}(T_1 \setminus \{ a \})$ and properties (iv), (v) and (vi) follow readily, again by construction of $T_1$.

- $a \not\square T$ and $T$ does not lead to a state on the DFS stack
Let \( T = T_1 \) and \( q \), \( q_1 \). Thus, \( q_1 \) is not on the DFS stack and when added it will be removed before \( q \) itself is removed (a characteristic of a depth-first search). But this means that the induction hypothesis can be applied to \( q_1 \), viz. there exists a sequence of leap sets \( T_2 \ldots T_{m-1} T_m \) from \( q_1 \) with respective selection function \( c \) for \( \text{lin}(T_2 \ldots T_{m-1}(T_m \setminus \{a\})) \) (defined as above) satisfying properties (i) to (vi). Now, since \( a \) \( T_1 \) and each operation in \( T_1 \) is invisible and independent wrt \( D \) of all operations in \( a \) before its own occurrence (if any) in \( \square \) (because \( T_1 \) \( j=1^k \) ample \( \mathcal{A} \)) it is immediate that \( T_1 T_2 \ldots T_{m-1} T_m \) is a sequence of leap sets from \( q \) which satisfies properties (i), (ii) and (iii). In order to prove property (iv), let \( \square \text{lin}(T_1) \) and \( \square \text{lin}(T_2 \ldots T_{m-1}(T_m \setminus \{a\})) = \square \text{lin} \). We derive \( \square a = \square \square \text{lin} = D \square \text{lin} \) using the induction hypothesis and the fact that the operations in a leap set are mutually independent wrt \( D \). Since all the operations in \( T_1 \) are independent wrt \( D \) of \( a \) and all the operations in \( \square \text{lin} \) are independent wrt \( D \) of all operations in \( \square \) and hence in \( \square \), it follows that \( \square \square \text{lin} = D \square \text{lin} \). Properties (v) and (vi) are proved similarly via the induction hypothesis.

Lemma 7.13 implies that for each operation \( a \) that becomes executable along some computation of a concurrent system \( S \), the extended leap automaton \( A^e_S \) for \( S \) also contains a sequence along which \( a \) becomes executable. Thus, aside from our objective, \( A^e_S \) serves to detect all (non-)executable operations of \( S \). Condition C3 on ample sets can thereby be void, as was the case for detecting non-progression states, which similarly favors the size of the extended leap automaton.

**Lemma 7.14**

Let \( S = (Q, q^0, \square, \square) \) be a concurrent system and \( f \) an LTL formula to be checked for \( S \). For every computation \( \square \square \square^{\square^{\square^0}} \square \square \square \square \) of \( S \) there exists a sequence of leap sets \( \square \) in \( A^e_S \) from the initial state \( q^0 \) of \( S \), with \( \square \text{lin}(\square) \), such that \( \square \text{lin}(\square) \).

**Proof:** For any (finite or infinite) computation \( \square \) of \( S \), while reading \( \square \), we describe a traversal of \( A^e_S \) starting from the initial state \( q^0 \) that yields a sequence \( \square \) of leap sets, with \( \square \text{lin}(\square) \), such that \( \square \text{lin}(\square) \). The following variables are used in the process:

- \( \square \) the sequence of operations read so far from \( \square \);
- \( \square \) the sequence of leap sets in \( A^e_S \) traversed so far, with \( \square \text{lin}(\square) \);
- \( \square \) a linearization of \( \square \) projected on the set of operations that have not yet been read from \( \square \) (i.e. removed from \( \square \text{lin}(\square) \) are the operations already read from \( \square \));
- \( q \) the current state of \( A^e_S \).

The variables \( \square \) and \( \square \) are initialized to the empty sequence, and \( q \) is initialized to \( q^0 \). Whenever the next operation \( a \) is read from \( \square \), the following updates are made:

1. \( \square = \square a \).
2. if $q = q_0 = a$ for some $a$, if $\forall a \not\in S$ such that all operations in $\square$ are independent wrt $D\square$ of $a$, then
   $\square := \square^a$.

3. else choose a sequence of leap sets $T_1T_2\ldots T_{m-1}T_m$ from the state $q$, leading to a state $q’$ such that $a \not\in T_m$ and $\square T_m a = \square T_m a \not\in \square T_m a$, with $\square\square lin(T_1T_2\ldots T_{m-1}(T_m \setminus \{a\}))$ and the selection function $c$, for any sequence of operations $\square$, such that $c(i) = true$ if $\square(i) \not\in ap(\square); c(i) = false$ otherwise. Make the following updates:
   
   - $\square := \square T_1T_2\ldots T_{m-1}T_m$;
   - $q := q'$

The following properties are now inductively proved to be invariant while reading $\square$:

i) $\square = D(\square)$

ii) $\square \not\in D(\square)$

iii) if the condition of Step 2 does not hold when checking it, then all the operations occurring in $\square$ are independent wrt $D\square$ of $a$, and

iv) the choice of the sequence $T_1T_2\ldots T_{m-1}T_m$ required by Step 3 can always be made when taking Step 3.

Initially, the properties (i) to (iv) trivially hold since $\square = \square$ and $\square$ are empty, and the algorithm is just before Step 1. At each step of the update procedure, $\square T_m a = \square T_m a \not\in \square T_m a$ since $a$ is the next operation from $\square$ read after $\square$ and $\square \not\in D(\square)$ by the induction hypotheses (i) and (ii). This together implies that $\square$ cannot contain operations that are dependent wrt $D\square$ of $a$ before the occurrence of $a$ (if any) in $\square$, which proves (iii). When $a$ does not occur in $\square$ and thus not in any leap set in $\square$, Step 3 is taken and the existence of the sequence of leap sets $T_1T_2\ldots T_{m-1}T_m$ from state $q$ required by Step 3 is guaranteed by Lemma 7.13, proving (iv). It is then easy to check that both (i) and (ii) are preserved by taking either Step 2 or Step 3 of the update procedure.

Let $\square$ denote the entire (finite or infinite) sequence of leap sets collected into $\square$ along a traversal of $A^c$ upon reading $\square$, with $\square lin(\square)$. From (i), for each $\square \square$ pref($\square$), $\square \not\in D(\square)$ and hence $\square \not\in D(\square)$. Also, from (ii), for each $\square \square$ pref($\square$), we have $\square \not\in D(\square)$. Thus, $\square = D(\square)$ by Definition 7.5, and $\square = D(\square)$ by definition of $\cdot \not\in D(\square)$ (in particular because all operations in $\square$ are invisible and independent wrt $D\square$ of the operations in $\square$).

\begin{theorem}

Let $S = (Q, q^0, \square, \square)$ be a concurrent system and $f$ an LTL formula to be checked for $S$. For every computation $\square \square \square \square \square$ of $S$ there exists a sequence of leap sets $\square$ in $A^c$ from the initial state $q^0$ of $S$, such that $\square$ satisfies $f$ iff $\square$ satisfies $f$.

\end{theorem}
Proof:!!By Lemma 7.14, there exists a sequence of leap sets \( \Box \) in the extended leap automaton \( A_\Box^c \) for \( S \), with \( \Box \Box \text{lin}(\Box) \), such that \( \Box \preceq \Box \Box \text{vis}_f(S) \Box \). To prove that \( \Box \) satisfies \( f \) iff \( \Box \) satisfies \( f \), it is sufficient to show that \( \Box \) satisfies \( f \) iff \( \Box \) satisfies \( f \). This follows from the fact that \( \Box \) captures the same visible operations and in the same order as they occur in \( \Box \) (any leap set contains at most one visible operation). That is, a computation of the extended leap automaton \( A_\Box^c \) over \( \Box \) is stuttering equivalent wrt \( f \) to a computation of the full automaton \( A_S \) over \( \Box \), and thus \( f \) cannot distinguish between these two computations (recall that \( f \) is implicitly assumed to be a nexttime-free LTL formula, i.e. \( f \) is stuttering closed). It remains to be shown that \( \Box \) and \( \Box \) yield stuttering equivalent sequences. Since \( \Box \preceq \Box \Box \text{vis}_f(S) \Box \), this is immediate by the definition of \( \preceq \Box \Box \text{vis}_f(S) \Box \) and the inclusion \( \text{vis}_f(S) \Box \Box \text{vis}_f(S) \Box \preceq D \Box \).

In conclusion, Theorem 7.15 warrants the application of off-line LTL model-checking algorithms [LP85] to the extended leap automaton of a concurrent system, as opposed to its full automaton or the reduced automaton resulting from POVAS.

### 7.3.3 On-the-fly LTL model-checking with (proper) leap sets

We now turn to the aptness of the extended leap automaton \( A_\Box^c \) for on-the-fly LTL model-checking. Recalling the preliminaries in Section 7.1.3, when the Büchi automaton \( A_{n-f} \) for the negation of the checked LTL formula \( f \) is defined over state predicates, each transition of the product automaton \( A_\Box^c \Box A_{n-f} \) is of the form \( (q, r) \xrightarrow{\alpha} (q \Box r) \), with \( q \xrightarrow{\alpha} q \Box \) a transition of \( A_\Box^c \) and \( r \xrightarrow{\alpha} r \Box \) a transition of \( A_{n-f} \) such that proposition \( P \) is true in system state \( q \). Alternatively, when \( A_{n-f} \) is defined over the alphabet of operations of the concurrent system \( S \), each transition of \( A_\Box^c \Box A_{n-f} \) is of the form \( (q, r) \xrightarrow{\alpha} (q \Box r) \), with \( q \xrightarrow{\alpha} q \Box \) a transition of \( A_\Box^c \), \( \Box \Box \text{lin}(T) \) and \( r \xrightarrow{\alpha} \Box \) a sequence in \( A_{n-f} \). Since there is at most one visible operation in any leap set, the product automaton is well-defined in either case. For the latter case in particular, at most one operation in \( T \) can actually cause a state transformation of \( A_{n-f} \), and \( r \Box \) follows therefore uniquely from \( r \) and (any linearization of) \( T \). The next theorem proves that it is sufficient to check the emptiness of \( A_\Box^c \Box A_{n-f} \) in order to verify \( S \) against the formula \( f \).

**Theorem 7.16**

Let \( S = (Q, q_0, \Box, [\Box]) \) be a concurrent system and \( f \) an LTL formula to be checked for \( S \). The product automaton \( A_S \Box A_{n-f} \) is empty iff the product automaton \( A_\Box^c \Box A_{n-f} \) is empty.

**Proof:**!!By Theorem 7.15, for every computation of \( S \) there exists a sequence of leap sets in the extended leap automaton \( A_\Box^c \) for \( S \), such that either both sequences satisfy \( f \) or both do not satisfy
$f$. Thus, $A'_S \Box A_{-f}$ is non-empty iff there exists some computation $[]$ of $S$ satisfying $\neg f$ iff there exists some sequence of leap sets $[]$ in $A'_S$ satisfying $\neg f$ iff $A'_S \Box A_{-f}$ is non-empty. ⌞

As for the on-the-fly construction of $A'_S \Box A_{-f}$ with POVAS (see Section 7.2.2), the on-the-fly construction of the product automaton $A'_S \Box A_{-f}$ can likewise postpone the closing of cycles on the DFS stack because it operates on a DFS stack (made up of composite states) that is different from the DFS stack used by the off-line construction of $A'_S$ (made up of single states of $S$). Since this affects solely condition $C2$ and since the definition of $pleap(q)$ does not rely on $C2$ (the ample sets used to construct $pleap(q)$ obey only $C1$ and $C3$), we need to modify only the definition of $xpleap(q)$ in order to make the execution of leap sets compatible with the algorithms for on-the-fly LTL model-checking. Similar to the modified condition $C2$ on ample sets, composite states of the product automaton $A'_S \Box A_{-f}$ are accounted for in leap sets as follows.

**Definition 7.17**

Let $S = (Q, q^0, [], []$) be a concurrent system, $f$ an LTL formula to be checked for $S$, and $(q, r)$ the current composite state of $A'_S \Box A_{-f}$ to be expanded during the DFS. Define

$$xpleap(q, r) = pleap(q) \Box \{T \Box \{a\} \mid a \Box X(q) \setminus op(pleap(q)) \text{ and } T \Box pleap(q): q \not\in_p q \Box (q \Box r) \text{ is on the DFS stack }\}$$

The adaptation of the algorithm for on-the-fly detection of acceptance cycles in the context of POVAS [HP95, Pel96, HPY96] now simply consists in using $xpleap(q, r)$ instead of $ample(q, r)$ to determine the subset of (not necessarily immediate) successor states of $q$ that need be explored next. Suffice it to say that this adaptation is indeed adequate for effectively combining the execution of leap sets with on-the-fly LTL model-checking. That is, the adapted algorithm returns true if the given concurrent system does not satisfy the checked LTL formula, and false otherwise. The correctness proof is virtually identical to the one given in [Pel96] for the on-the-fly construction algorithm of POVAS, considering also the modification suggested in [HPY96] (see Section 7.2.2) of the nested DFS algorithm in [CV+92] (see Section 7.1.3). It entails a reduction from the on-the-fly algorithm to a non-deterministic variant of the off-line algorithm, which preserves the validity of Lemma 7.13 and hence of Lemma 7.14 and Theorem 7.15. We refer the interested reader to [CV+92, Pel96, HPY96] for precise details.

It is appropriate here to point out that an alternative definition can be given for $xpleap(q, r)$, and likewise for $xpleap(q)$, that does not require inspection of the DFS stack. By always adding the sets $T \Box \{a\}$ to $pleap(q)$ for each $a \Box X(q) \setminus op(pleap(q)$ and for each $T \Box pleap(q)$, i.e. even if $T$ does not lead to a state on the DFS stack, all the previous results still hold (in particular, the sequence of leap sets sought in Lemma 7.13 is then guaranteed to be of length one). Although this generally
increases the size of the extended leap automaton $A^e_S$ for a concurrent system $S$, and of the corresponding product automaton $A^e_S \boxdot A_{\neg f}$, it may reduce the time for constructing these automata, especially when it turns out that at many expansion steps during state exploration the number of proper leap sets (i.e. the leap sets in $pleap(x)$) and the current DFS stack are large. In addition, for on-the-fly cycle detection one can directly use the nested DFS algorithm of [CV+92] without any modification.

**Example 7.18**
Consider once more the concurrent system $S$ and the LTL formula $f$ described in Example 7.3, and depicted in Figure 7.2. The product automaton $A^e_S \boxdot A_{\neg f}$ is shown in Figure 7.5. It is again non-empty, like the “full product” $A_S \boxdot A_{\neg f}$ in Figure 7.2, and smaller than the product $A^e_S \boxdot A_{\neg f}$ in Figure 7.3 obtained with POVAS. For instance, at the initial state $(10, 20, 0)$ both $X_1((10, 20)) = \{a\}$ and $X_2((10, 20)) = \{v\}$ satisfy conditions $C_1$ and $C_3$, yielding $pleap((10, 20, 0)) = \{a, v\}$. Furthermore, since this proper leap set does not lead to a composite state already on the DFS stack, $xpleap((10, 20, 0)) = pleap((10, 20, 0))$. Notice that at the composite state $(10, 21, 0)$ the subset $X_1((10, 21)) = \{a\}$ satisfies $C_1$ and $C_3$, but $X_2((10, 21)) = \{w\}$ does not because operation $w$ is visible. Thus, $pleap((10, 21, 0)) = \{a\}$. This proper leap set leads back to a state on the DFS stack, however, and hence $xpleap((10, 21, 0)) = \{a\}, \{a, w\}$. For this small but illustrative example, the extra reduction achieved amounts to just one state and one transition. Later we will see that the overall gain over POVAS is generally more significant for “larger” concurrent systems. 

![Figure 7.5](image-url) The product $A^e_S \boxdot A_{\neg f}$ for the concurrent system $S$ and the LTL formula $f$ in Example 7.3.
7.3.4  LTL model-checking under fairness assumptions

The presentation of POVAS and its proposed enhancement so far has been confined to LTL model-checking without fairness. As explained at the end of the preliminary subsection 7.1.2, when the interleaving semantics of a concurrent system involves fairness, all computations of the system that violate the assumed fairness assumptions are no longer considered. Since fairness assumptions can also be expressed in nexttime-free LTL [LP85], LTL model-checking under fairness assumptions can be done simply by checking formulas of the form \( f_1 \bigsquare f_2 \), where \( f_1 \) formalizes a conjunction of fairness assumptions and \( f_2 \) a desirable property. Unfortunately, adding \( f_1 \) as part of the formula often introduces many additional dependencies among operations [GW94, Pel96], since \( f_1 \) causes more (usually all) operations to be visible and condition C3 on ample sets must be applied also to \( f_1 \). A DFS based on the execution of ample sets or leap sets will then yield little or no gain at all.

In order to exploit a restricted class of fairness assumptions more efficiently and, in general, to introduce dependencies among visible operations more carefully, it was shown in [Pel93] that a temporal formula \( f \) which is not equivalence robust (see Section 7.3.2) can sometimes be made equivalence robust by rewriting \( f \) as a Boolean combination of sub-formulas \( f_i \) and treating each \( f_i \) individually when adding dependencies among visible operations. This is based on the simple fact that when two formulas \( f_1 \) and \( f_2 \) are equivalence robust, then so are \( f_1 \bigsquare f_2, f_1 \land f_2 \) and \( \neg f_1 \). Precisely, instead of augmenting the dependency relation \( D \) of a concurrent system \( S \) with all pairs of visible operations, which is the effect of imposing condition C3 on ample sets and thereby on (proper) leap sets, it appears sufficient to augment \( D \) with the pairs in \( \bigcup \{ \text{vis}_{f_1}(S) \bigsquare \text{vis}_{f_2}(S) \} \). This union is a subset of \( \text{vis}_{f}(S) \bigsquare \text{vis}_{f}(S) \) and can yield much fewer dependencies in several cases. For example, if \( f = \Diamond(P \bigcirc Q) \) then \( \text{vis}_{f}(S) \) includes all the operations of \( S \) whose execution can change the truth value of the Boolean propositions \( P \) or \( Q \). However, this formula is logically equivalent to \( f_1 \land f_2 = \Diamond P \land Q \), where \( \text{vis}_{f_1}(S) \) includes the operations that can change \( P \) and \( \text{vis}_{f_2}(S) \) includes those that can change \( Q \). Thus, any two operations such that one can change only \( P \) but not \( Q \), and the other can change \( Q \) but not \( P \), are dependent wrt to \( D \bigsquare \text{vis}_{f_1}(S) \bigsquare \text{vis}_{f_2}(S) \) but not necessarily wrt \( D \bigsquare \bigcup \{ \text{vis}_{f_1}(S) \bigsquare \text{vis}_{f_2}(S) \} \). Other logical equivalences among temporal formulas that can be used profitably as rewriting rules are: \( \Box(f_1 \bigsquare f_2) = \Box f_1 \bigsquare \Box f_2, \Box \Diamond(f_1 \land f_2) = \Box \Diamond f_1 \land \Box \Diamond f_2, \Box \Diamond(f_1 \bigsquare f_2) = \Diamond \Box f_1 \bigsquare \Diamond \Box f_2, \) as well as \( (f_1 \bigsquare f_2) U f_3 = (f_1 U f_3) \bigsquare (f_2 U f_3) \) and \( f_3 U (f_1 \land f_2) = (f_3 U f_1) \land (f_3 U f_2) \). Although rewriting can increase the length of a formula exponentially, it is argued in [Pel93, Pel96] that the checked formulas are generally quite short and, moreover, that they need not be rewritten completely. That is, the rewriting rules can be used to separate Boolean components of a formula one at a time without explicitly generating the rewritten formula. This is done with a recursive algorithm in time linear in the length of the formula [Pel93]. Following this algorithm, the original formula can still be used for actual model-checking.
In summary, POVAS employs the dependency relation $D \square \bigcup_i (\text{vis}_{f_i}(S) \square \text{vis}_{f_i}(S))$ instead of $D \square \text{vis}_{f}(S) \square \text{vis}_{f}(S)$ (the latter implicitly through condition C3) for LTL model-checking with a certain class of fairness assumptions [Pel96], including such assumptions as weak fairness, process fairness and process justice [Fra86, MP92]. This is accomplished by dropping condition C3 and enforcing condition C1 with respect to $D \square \bigcup_i (\text{vis}_{f_i}(S) \square \text{vis}_{f_i}(S))$ as opposed to just $D$. In effect, the fairness assumptions then act as “low-cost” filters on the computations of a concurrent system, allowing the calculation of ample sets with respect to a subset of these computations. This may decrease the size of ample sets and thus result in the exploration of a yet smaller number of states and transitions. It is evident that the same advantage applies also to the proposed enhancement of POVAS. Indeed, we can equally use $D \square \bigcup_i (\text{vis}_{f_i}(S) \square \text{vis}_{f_i}(S))$ in place of $D \square \text{vis}_{f}(S) \square \text{vis}_{f}(S)$, and construct leap sets as before from ample sets that respect this refined dependency relation.

A final note concerns yet another, very recent improvement of POVAS for on-the-fly LTL model-checking, where the number of visible operations with respect to a formula $f$ may diminish during model-checking [KPV97]. Roughly speaking, it is shown that the set $\text{vis}_{f}(S)$ itself can in many cases be reduced dynamically, yielding even fewer dependencies, by exploiting information of the current state of the Büchi automaton $A_{\text{vis}}$. This is translated into a visibility condition that relaxes condition C3. We refer to [KPV97] for details. Because this improvement operates “only” at the level of (the Büchi automaton $A_{\text{vis}}$ for) the formula $f$, it can also be applied directly to our enhancement of POVAS.

## 7.4 LTL model-checking in the CFSM model

Having addressed the topic of LTL model-checking in general for finite-state concurrent systems formalized as LTSs, for the remainder of the chapter we turn the focus back to protocols defined as networks of CFSMs. As noted earlier, every bounded protocol $\square$ in the CFSM model qualifies as a finite-state concurrent system: its behavior is defined by the LTS $(R_j, G^0, \bigcup_i [i], \{(G, t, H) \mid G \square R_j \square G \models H\})$. POVAS and its proposed enhancement based on leap sets are thus suited for LTL model-checking in the CFSM model. We now show how to realize these two relief strategies specifically for the CFSM model, by harmonizing their formulation with the formulation of LRA in Chapter 5. This facilitates the integration of POVAS and its enhancement in the research tool package RELIEF discussed in Chapter 6, which can then be used to perform an experimental comparison of the performance of both techniques.

POVAS and its proposed enhancement make use of a dependency relation among transitions (or operations) to tackle the wasteful exploration of many equivalent interleavings of concurrent transitions. As discussed, it is thereby important that it can be easily checked in practice whether two transitions are (in)dependent. This is certainly the case for protocols in the CFSM model: a
syntactic condition that is sufficient for two transitions \( t \) and \( t' \) to be independent is that they are not from the same process and they do not involve the same simplex channel. It is not difficult to see that the dependency relation induced by this condition is a valid one according to Definition 7.4. Nevertheless, for the CFSM model we can readily establish a weaker condition by considering a conditional dependence among transitions, as in Definition 7.4bis. For each individual global state \( G \), two transitions are independent in \( G \) if they are not from the same process and if neither of them is enabled at \( G \) by the other. Proposition 7.19 proves that this condition is sufficient to meet the requirements listed in Definition 7.4bis. Recall from Section 4.1 that a transition \( t \) defined at \( G \) is enabled at \( G \) by a transition \( t' \) if \( t \) is potentially executable at \( G \) and the execution of \( t' \) at \( G \) causes \( t \) to become executable immediately thereafter. A send (receive) transition is potentially executable at \( G \) if it involves a channel that is full (empty) in \( G \).

**Proposition 7.19**

Let \( G \) be a global state of a protocol \( \square \). Two transitions \( t \) and \( t' \) are independent in \( G \) if it holds that 

(i) \( \text{act}(t) \neq \text{act}(t') \), and

(ii) \( t \) is not enabled at \( G \) by \( t' \) and \( t' \) is not enabled at \( G \) by \( t \).

**Proof:** Since transitions \( t \) and \( t' \) are not from the same sequential process (i.e. \( \text{act}(t) \neq \text{act}(t') \)), and since neither transition is enabled at \( G \) by the other transition, it follows immediately that the two requirements in Definition 7.4bis are satisfied. That is, if \( t \) (\( t' \)) is executable at \( G \), leading to some global state \( H \), then \( t' \) (\( t \)) is executable at \( G \) iff it is executable at \( H \), and if both \( t \) and \( t' \) are executable at \( G \), then executing the sequence \( tt' \) from \( G \) yields the same global state as executing the sequence \( t' t \) from \( G \). Transitions \( t \) and \( t' \) are thus independent in \( G \).

The above translated requirement on a conditional dependency relation for protocols in the CFSM model renders in turn a translation of condition \( C1 \) on ample sets (see Definition 7.6) for these protocols, as stated by the next proposition.

**Proposition 7.20**

Let \( G \) be a global state of a protocol \( \square \), and let \( A \subseteq X(G) \) be a (non-empty) subset of transitions executable at \( G \). \( A \) satisfies condition \( C1 \) if for each \( i \in \text{act}(A) \) it holds that:

i) \( X_i(G) \subseteq A \), and

ii) if \( A \subseteq X(G) \), then \( P_i(G) = \emptyset \).

**Proof:** By Definition 7.6, condition \( C1 \) stipulates that for each non-empty sequence \( \square = G^1 \circ \ldots \circ G^m \circ G^{m+1} \) from \( G^1 = G \), with \( t_j \subseteq \bigcup_{i \in A} t_i \setminus A \) for all \( 1 \leq j \leq m \), each transition \( t_j \) is independent in \( G^j \) of all transitions in \( A \). This holds trivially if \( A = X(G) \), satisfying properties (i) and (ii), since in that case no such non-empty sequence \( \square \) from \( G \) exists. Alternatively, if \( A \subseteq X(G) \)
then by properties (i) and (ii), \( X_i(G) \sqcap A \) and \( P_i(G) = \emptyset \) for all \( i \sqcap \text{act}(A) \). We prove that this implies that for all \( 1 \leq j \leq m \): (1) \( \text{act}(t_j) \sqcap \text{act}(A) = \emptyset \), and (2) if \( i \sqcap \text{act}(A) \), then \( P_i(G^i) = \emptyset \).

To show (1), suppose that \( \text{act}(t_j) = \{i\} \sqcap \text{act}(A) \) for some \( j \), then \( X_i(G) \sqcap A \) and \( P_i(G) = \emptyset \). Hence, \( t_j \sqcap X_i(G) \sqcap P_i(G) \). However, since no transition in \( A \), and thus no transition in \( X_i(G) \), is executed along \( \sqcap \), it follows that \( t_j \) cannot become executable at \( G^j \) – a contradiction. To show (2), suppose that \( i \sqcap \text{act}(A) \), then \( X_i(G) \sqcap A \) and \( P_i(G) = \emptyset \). From the proof of (1), no transition of process \( P_i \) is executed along \( \sqcap \), implying that \( P_i(G^i) = \emptyset \) for all \( 1 \leq j \leq m \).

In conclusion, from (1) it follows that for each transition \( t \sqcap A \) we have \( \text{act}(t_j) \neq \text{act}(t) \), while from (2) it follows that for each \( i \sqcap \text{act}(A) \) we have \( E_i(G^i) \sqcap P_i(G^i) = \emptyset \) (i.e. no process with transitions in \( A \) has enabled transitions at \( G^i \)). Thus, by Proposition 7.19, transition \( t_j \sqcap X(G^i) \) is independent in \( G^i \) of all transitions in \( A \). As this holds for all \( 1 \leq j \leq m \), \( A \) satisfies condition \( \textbf{C1} \). □

The first requirement in Proposition 7.20 stipulates that the subset \( A \) of \( X(G) \) contains for each process either all or none of this process’ executable transitions at \( G \), which is in fact necessary for condition \( \textbf{C1} \) to hold. To see this, suppose that \( X_i(G) \sqcap \) \( \emptyset \) for some \( i \sqcap \text{act}(A) \), then process \( P_i \) has two transitions \( t, t' \sqcap X_i(G) \) where \( t \sqcap A \) and \( t' \sqcap A \). These transitions are dependent in \( G \) since they are from the same process. But \( G \sqcap H \) is then a non-empty sequence from \( G \) of transitions outside \( A \) containing a transition that is dependent with a transition in \( A \), and thus \( \textbf{C1} \) is violated. The second requirement in Proposition 7.20 prohibits every process with transitions in \( A \) from having potentially executable transitions at \( G \) if \( A \) is a proper subset of \( X(G) \). To illustrate the importance of this condition for \( \textbf{C1} \), suppose that \( P_i(G) \neq \emptyset \) for some \( i \sqcap \text{act}(A) \), then process \( P_i \) has two transitions \( t \sqcap X_i(G) \) and \( t' \sqcap P_i(G) \) such that \( t \sqcap A, t' \sqcap A \), and \( t \) and \( t' \) are dependent in \( G \). If \( A \sqcap X(G) \), then since \( t' \) is potentially executable at \( G \) it may be possible that \( t \) becomes executable, and is executed, along a sequence from \( G \) of only transitions outside \( A \). Again, in that case \( \textbf{C1} \) would be violated. This scenario can actually be drawn for the initial state \( G^0 \) of the simple protocol depicted in Figure 5.1, by letting \( A \models X_2(G^0) = \{(20, -b, 21)\} \). We have \( X_2(G^0) \sqcap X(G^0) \) and \( P_2(G^0) = \{(20, +a, 22)\} \neq \emptyset \), and hence \( X_2(G^0) \) does not satisfy the second requirement in Proposition 7.20. Condition \( \textbf{C1} \) is violated here because \( G^0 \sqcap \) \( (20, 0, 20) \) \( G^1 \sqcap \) \( (20, +a, 22) \) \( G^2 \) is a sequence from \( G^0 \) of transitions outside \( X_2(G^0) \), while transition \( (20, +a, 22) \) is dependent in \( G^1 \) with \( (20, -b, 21) \). □

In Section 7.2 we described the algorithm used by POVAS for computing an ample set in a global state \( G \). It aims at finding some process \( P_i \) whose set of executable transitions \( X_i(G) \) is non-empty and satisfies the three conditions \( \textbf{C1}, \textbf{C2} \) (or \( \textbf{C2} \) for on-the-fly model-checking) and \( \textbf{C3} \). For the enhancement of POVAS in Section 7.3 we adopted a similar algorithm to find \textit{all} processes \( P_i \) for which \( X_i(G) \) satisfies just \( \textbf{C1} \) and \( \textbf{C3} \). Concerning the implementation of these algorithms in the CFSM model, it is now immediate from Proposition 7.20 that one can check \( \textbf{C1} \) simply by
establishing whether process \( P_i \) has potentially executable transitions at \( G \). Checking \( \textbf{C2} \) and \( \textbf{C3} \) is of course done as before by examining the DFS stack and the visibility of the transitions in \( X_i(G) \) with respect to the given LTL formula. Precisely, when \( G \) is the current global state of a protocol \( \square = (\{P_i \mid i \in I\}, L) \) to be expanded during a DFS, and \( f \) is the LTL formula to be checked for \( \square \), then for each \( i \in I \) we have that:

- \( X_i(G) \) satisfies \( \textbf{C1} \) if \( P_i(G) = \emptyset \);
- \( X_i(G) \) satisfies \( \textbf{C2} \) if no \( x \in X_i(G) \) with \( G \vdash_{\square} H \) is such that \( H \) is on the DFS stack;
- \( X_i(G) \) satisfies \( \textbf{C3} \) if \( X_i(G) \not\subseteq \text{vis}_f(\square) = \emptyset \).

In terms of the algorithm in Figure 7.4 for finding multiple disjoint ample sets \( \text{wrt} \ \textbf{C1} \) and \( \textbf{C3} \), the function call \textit{check}_C1\( (X_i(G)) \) is thus replaced by the simple test \( P_i(G) = \emptyset \).

With the above translation condition \( \textbf{C1} \) for protocols in the CFSM model, the formulation of LRA for verifying logical correctness properties in Chapter 5 can now be adapted easily to incorporate also the proposed enhancement of POVAS for LTL model-checking. Specifically, the leap sets to be used for LTL model-checking in the CFSM model can be constructed on the basis of wait-sets, in accordance with the following two definitions (cf. definitions 5.30 and 5.46).

**Definition 7.21**

Let \( G \) be a global state of a protocol \( \square = (\{P_i \mid i \in I\}, L) \), and let \( J, K \not\subseteq L \) and \( V \not\subseteq \bigcup_{i \in I} i \). Define \( \text{wait}(G, J, K, V) = \{ i \in I \mid X_i(G) \not\subseteq \emptyset \substack{\text{wrt} \ \square, \ P_i(G) \not\subseteq \emptyset \substack{\text{wrt} \ J, i \substack{\text{wrt} \ K: X^+_j(G) \not\subseteq \emptyset \substack{\text{wrt} \ X_j(G) \not\subseteq \emptyset}}}} \}. \) Define

\[
\text{pleap}(G, J, K, V) = \{ T \mid T \text{pleap}(G) \not\subseteq \text{act}(T) = \{ i \in I \mid i \not\subseteq \text{wait}(G, J, K, V) \} \} \\
\quad \text{if } \text{wait}(G, J, K, V) \not\subseteq I
\]

\[
\text{pleap}(G, J, K, V) = \{ \{ t \} \mid t \text{X}(G) \}
\quad \text{otherwise.}
\]

**Definition 7.22**

Let \( G \) be a global state of a protocol \( \square \) to be expanded during the DFS. Define

\[
\text{xpleap}(G, J, K, V) = \text{pleap}(G, J, K, V) \not\subseteq
\{ T \bigbar \{ \{ t \} \mid t \text{X}(G) \} \text{ such that } \text{act}(t) \not\subseteq \text{wait}(G, J, K, V), \text{ and }
\quad T \text{pleap}(G, J, K, V) \not\subseteq \text{lin}(T) \text{ with } G \not\subseteq H \text{ and } H \text{ on the DFS stack} \}
\quad \text{if } \text{wait}(G, J, K, V) \not\subseteq I
\]

\[
\text{xpleap}(G, J, K, V) = \text{pleap}(G, J, K, V)
\quad \text{otherwise.}
\]
Observe that when \( V \) includes the set \( \text{vis}_f(\square) \) of visible transitions of a protocol \( \square \) wrt to some LTL formula \( f \), the wait-set \( \text{wait}(G, J, K, V) \) captures at least every process \( P_i \) whose set \( X_i(G) \) does not qualify as an ample set in \( G \) with respect to conditions \( C1 \) and \( C3 \), i.e. every process without executable transitions at \( G \) (violating the non-emptiness requirement on ample sets), or with potentially executable transitions at \( G \) (violating \( C1 \)), or with executable transitions at \( G \) that are visible wrt \( f \) (violating \( C3 \)). This attests that Definition 7.21 and Definition 7.22 indeed comply with Definition 7.9 and Definition 7.11 in Section 7.3, respectively. As a result, the reduced global state space obtained by the execution of the leap sets in \( \text{xleap}(G, J, K, V) \) in global states is adequate for deciding the absence of non-progress states, non-executable transitions, unspecified receptions wrt \( J \) and buffer overflows wrt \( K \), as derived before in Chapter 5, and moreover for deciding the satisfiability of any LTL formula \( f \) with \( \text{vis}_f(\square) \) \( \square \) \( V \). In order to verify a protocol \( \square \) against a given LTL formula \( f \), one would then typically set \( V = \text{vis}_f(\square) \) and \( J = K = \emptyset \). To sum up, by incorporating the proposed enhancement of POVAS into the formulation of LRA, we have established LRA as a uniform relief strategy for verifying both syntactic and semantic correctness properties of protocols in the CFSM model.

### 7.5 Experiments

The off-line versions of POVAS and its proposed enhancement have been implemented in the research tool package RELIEF, based on their formulation above for protocols in the CFSM model. Since the proposed enhancement of POVAS implements the execution of leap sets commensurate with LRA, in this section we will name it also LRA for short. Following the evaluation approach motivated in Chapter 6, the two model-checking techniques have been tested on the 400 sample protocols obtained with the automatic protocol synthesizer in RELIEF (see Section 6.3.1), and on the three real protocols taken from the literature: the X.21 call establishment/clear protocol [WZ78], the cache coherence protocol [Hol91] (see Appendix), and the alternating bit protocol with unreliable channels [Pac87] (see Section 6.3.2). The results of the experiments with the 400 synthesized protocols are given in Table 7.1, which compares the average percentages of reduction obtained with POVAS and LRA for off-line model-checking, per number of processes in a protocol and per concurrency level of a protocol. Recall that the latter was introduced in Chapter 6 as a conceivable measure for the degree of parallelism in a protocol. The first two rows of Table 7.1 show the reductions by POVAS and LRA over the conventional reachability analysis (CRA), respectively. The third row compares LRA directly to POVAS by normalizing the reductions by LRA with respect to those by POVAS. Table 7.2 gives the results of the experiments with the three real protocols. Overall, the numbers clearly indicate that using LRA instead of POVAS can further decrease both the memory and time resources needed for model-checking.
Chapter 7 Leaping reachability analysis for LTL model-checking

**Table 7.1** LRA compared to POVAS for off-line model-checking.

<table>
<thead>
<tr>
<th>Techniques</th>
<th>Average reductions (%) per concurrency level</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>number of processes</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>LRA vs. CRA</td>
<td>states</td>
</tr>
<tr>
<td></td>
<td>transitions</td>
</tr>
<tr>
<td>LRA vs. CRA</td>
<td>space</td>
</tr>
<tr>
<td></td>
<td>time</td>
</tr>
<tr>
<td>LRA vs. POVAS</td>
<td>states</td>
</tr>
<tr>
<td></td>
<td>transitions</td>
</tr>
<tr>
<td>LRA vs. POVAS</td>
<td>space</td>
</tr>
<tr>
<td></td>
<td>time</td>
</tr>
<tr>
<td>POVAS vs. CRA</td>
<td>states</td>
</tr>
<tr>
<td>LRA vs. POVAS</td>
<td>space</td>
</tr>
<tr>
<td></td>
<td>time</td>
</tr>
</tbody>
</table>

**Table 7.2** LRA and POVAS applied to three real protocols.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Technique</th>
<th>States</th>
<th>Transitions</th>
<th>Space (MB)</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X.21 call establishment/clear</td>
<td>CRA</td>
<td>29868</td>
<td>64903</td>
<td>1.42</td>
<td>10.58</td>
</tr>
<tr>
<td></td>
<td>POVAS</td>
<td>21805</td>
<td>32816</td>
<td>1.04</td>
<td>16.08</td>
</tr>
<tr>
<td></td>
<td>LRA</td>
<td>15500</td>
<td>26882</td>
<td>0.75</td>
<td>6.33</td>
</tr>
<tr>
<td>Cache coherence</td>
<td>CRA</td>
<td>37037</td>
<td>126152</td>
<td>1.90</td>
<td>32.52</td>
</tr>
<tr>
<td></td>
<td>POVAS</td>
<td>8760</td>
<td>11388</td>
<td>0.45</td>
<td>8.78</td>
</tr>
<tr>
<td></td>
<td>LRA</td>
<td>5572</td>
<td>7966</td>
<td>0.28</td>
<td>5.45</td>
</tr>
<tr>
<td>Alternating bit</td>
<td>CRA</td>
<td>135352</td>
<td>626608</td>
<td>6.84</td>
<td>77.05</td>
</tr>
<tr>
<td></td>
<td>POVAS</td>
<td>92414</td>
<td>266206</td>
<td>4.64</td>
<td>69.68</td>
</tr>
<tr>
<td></td>
<td>LRA</td>
<td>63876</td>
<td>198583</td>
<td>3.24</td>
<td>65.90</td>
</tr>
</tbody>
</table>

### 7.6 Summary

In this chapter we studied the verification of temporal properties of finite-state concurrent systems and protocols. In particular, we addressed the state explosion problem in the context of LTL model-checking. LTL (linear-time temporal logic) is a propositional logic well suited for reasoning about semantic correctness properties of concurrent systems, including arbitrary safety and liveness properties. LTL model-checking refers to a fully automatic procedure, based on state exploration, for checking whether a given system satisfies some temporal property that can be expressed as a formula in LTL. In order to relieve the state explosion problem for LTL model-checking, a series of
so-called partial-order methods have been developed in recent years. It has been demonstrated that these methods can in many cases substantially reduce the space and time needed for LTL model-checking.

In this chapter we have built on the concepts underlying partial-order methods to yield an approach that enables further reductions in space and time for LTL model-checking. Specifically, we have proposed an enhancement of the partial-order method based on ample sets as described in [HP95, Pel96]. This method, which we referred to as POVAS (Partial Order Verification with Ample Sets), was chosen because it is generic in the sense that it can be readily adapted to capture the other partial-order methods (those based on persistent sets or stubborn sets), and because it is the most advanced partial-order method in terms of the properties that can be checked, the way fairness is dealt with, and the low overhead and high overall performance of its implementation [HP95, Pel96]. The idea behind the proposed enhancement stems from the principles underlying LRA in Chapter 5: instead of exploring a fixed interleaving order among concurrent operations, as does POVAS through the execution of ample sets, we abstain from any order altogether by executing leap sets that mimic a truly concurrent execution of these operations. Although POVAS and its proposed enhancement cannot be strictly compared in the sense that one does not subsume the other (i.e. their respective sets of reachable global states are not comparable by means of set inclusion), the experiments performed with both techniques confirmed that our approach to LTL model-checking is indeed an enhancement of POVAS. That is, our approach generally results in better space and time reductions and therefore widens the applicability of LTL model-checking to more complex concurrent systems and protocols.