Chapter 3

An introduction to model checking

3.1. Introduction

In formal logic, *model checking* designates the problem of determining whether a formula φ evaluates to true or false in an interpretation $\mathcal{K}$, written $\mathcal{K} \models \varphi$. This problem finds applications in computer science: for example, $\mathcal{K}$ might represent a knowledge base and φ could be a query of which we wish to determine if it is implied by the knowledge in the base. We are then interested in finding efficient algorithms for determining whether $\mathcal{K} \models \varphi$ holds. In this chapter, we are interested in applications where $\mathcal{K}$ represents a system and φ a formula that represents a correctness property of this system. Typically, the systems we are interested in are *reactive*, that is, they interact repeatedly with their environment. They are often more concerned with control than with data and are usually composed of several components operating in parallel. Starting from a simple lift control application, we present basic ideas and concepts of verification algorithms in this context. The first publications about model checking appeared in 1981 by Clarke and Emerson [CLA 81] and by Queille and Sifakis [QUE 81]. Since then much progress has been made, and model checking has left the academic domain to enter mainstream development, notably of embedded systems and of communication protocols. Advances in the theory and application of model checking are reported in several important international conferences (including CAV, CHARME, and TACAS).

The inputs of the model checker are a description of the system to be analyzed and the property to verify. The tool either confirms that the property is true in the model or informs the user that it does not hold. In that case, the model checker will

Chapter written by Stephan MERZ.

also provide a counter-example: a run of the system that violates the property. This answer helps find the reason for the failure and has significantly contributed to the success of model checking in practice. Unfortunately, in practice model checking does not always yield such clear-cut results because the resource requirements (in terms of execution time and memory needed) can prohibit verifying more than an approximate model of the system. The positive outcome of model checking then no longer guaranteed the correctness of the system and, reciprocally, an error found by the model checker may be due to an inaccurate abstraction of the system. Model checking is therefore not a substitute for standard procedures to ensure system quality, but it is an additional technique that can help discover design problems at early stages of system development.

This chapter is intended as an introduction to the fundamental concepts and techniques of algorithmic verification. It reflects a necessarily subjective reading of the (abundant) studies. We try to give many references to original work so that the chapter can be read as an annotated bibliography. More extensive presentations of the subject can be found in books and more detailed articles, including [BÉR 01, CLA 99, CLA 00, HUT 04]. The structure of the chapter is as follows: section 3.2 presents an example of verifying a lift controller using the model checker SPIN. Basic definitions of transition systems, as well as an algorithm for verifying system invariants, are given in section 3.3. Section 3.4 is devoted to introducing temporal logics and ω-automata, which serve as the bases for the model checking algorithms presented in section 3.5. Finally, some topics of research are presented in section 3.6, which concludes the chapter. We restrict ourselves here to the case of discrete, untimed systems. Model checking techniques for timed systems are presented in detail in Chapter 4.

3.2. Example: control of an elevator

We will consider a simple model for the control of an elevator serving the floors of a building. This model serves merely to demonstrate the basic ideas, it is not intended as a realistic model of elevator control. It is generally advisable to start model checking for rough abstractions of the intended system in order to manage the complexity of the model and to get the first results quickly. Additional features required for a more realistic model can then be added one after another.

The model of figure 3.1 is written in the language PROMELA [HOL 03]. In this language, a system description is given by several parallel processes that execute asynchronously. Our model consists of a process `lift` that represents the lift (controller) and as many additional processes `button` as there are floors in the building: that number is specified by the constant `FLOORS`. All processes are started by the process `init`. The externally observable interface of the elevator control is made up of the global

```
#define FLOORS  4       /* number of floors */
int floor = 0;          /* current floor */
bool request[FLOORS];   /* outstanding requests */

proctype lift() {
  bool moving = false; bool going_up = true; int j;

  do
  :: moving && going_up -> atomic { floor++;
       if :: request[floor] -> moving = false; request[floor] = false
          :: else -> skip
       fi     }
  :: moving && !going_up -> atomic { floor--;
       if :: request[floor] -> moving = false; request[floor] = false
          :: else -> skip
       fi     }
  :: !moving && going_up -> atomic { j = floor+1;
       do :: j == FLOORS -> going_up = false; break
          :: j < FLOORS && request[j] -> moving = true; break
          :: else -> j++
       od     }
  :: !moving && !going_up -> atomic { j = floor-1;
       do :: j < 0 -> going_up = true; break
          :: j >= 0 && request[j] -> moving = true; break
          :: else -> j--
       od     }
  :: assert (0 <= floor && floor < FLOORS)
  :: assert (!moving || !request[floor])
  od
}

proctype button(int myfloor) {
  do :: atomic { myfloor != floor -> request[myfloor] = true }
     :: true -> skip
  od
}

init {
  int i=0;
  run lift();
  do :: i < FLOORS -> run button(i); i++ od
}
```

Figure 3.1. *Model of an elevator in* PROMELA

	Assertions				Formula (3.1)			
FLOORS	States	Transitions	Time	Memory	States	Transitions	Time	Memory
4	6,151	35,297	0.38	2.9	8,769	164,763	1.10	6.1
5	35,721	239,788	1.11	8.1	52,052	1.38e06	9.59	9.4
6	194,556	1.48e06	9.79	21.1	288,395	1.01e07	76.64	58.5

Figure 3.2. *Model size and resource consumption by* SPIN

variables `floor` whose value gives the floor where the lift is currently situated, and `request`, a Boolean array that indicates the requests to be served by the lift.

Each process `button(i)` represents the behavior of passengers who may call the lift to serve floor `i`. The process contains a non-terminating loop (`do` ... `od`) in the body of which the lift can be requested at that floor provided the lift is currently located at a different floor. In this model we do not distinguish between calls from within the lift cabin and from the floors.

The process `lift` declares the local Boolean variables `moving` and `going_up` that indicate if the lift is currently moving and, if so, in which direction. The integer variable `j` is used as an auxiliary scratch variable. The body of that process again consists of an infinite loop that models the elevator control. There are four branches (introduced by `::`) according to the different states the lift can be in. If it is currently moving, it will arrive at the next-higher or next-lower floor; it will stop there in case there is a pending request and then reset the request. If the lift is stopped at a floor, the controller checks first if there is another request in the current direction or, if that is not the case, in the opposite direction. In the case of a pending request, the lift will start moving in the required direction, otherwise it will remain stopped at the current floor.

The two final branches of the loop contain assertions that are used to verify elementary correctness properties of the model. The first assertion states that the lift never moves out of the range of legal floor values – this is not completely obvious from the way the model is written. The second assertion verifies that the lift does not pass a floor for which there is a pending request. Assertions are a particular kind of *safety properties*, which state that "nothing bad ever happens".

Beyond these assertions, an important correctness property of the elevator is that every request will eventually be served. This is an example of a *liveness property* which state that "something good happens eventually". For the first floor, this property is expressed in temporal logic (see section 3.4) by the formula

$$\mathrm{G}(\texttt{request[1]} \Rightarrow \mathrm{F}(\texttt{floor} = 1)). \tag{3.1}$$

The model checker SPIN can be used to verify assertions as well as temporal logic formulas over PROMELA models. For the verification of the liveness property (3.1), we have to specify the option *with weak fairness* in order to ensure that the process `lift` will not remain inactive forever. Figure 3.2 shows the size of the models (in terms of numbers of states and transitions), as well as the memory consumption (in MB) and the time (in seconds) required by SPIN to verify the indicated properties over the model of Figure 3.1. We can observe that the overall cost of verification increases by about an order of magnitude per additional floor, and that the verification of (3.1) is significantly more expensive than just assertion checking.

3.3. Transition systems and invariant checking

The semantic framework for system models used for model checking is provided by the concepts of *transition systems* and *Kripke structures*. We now formally define these notions and present an algorithm for verifying invariant properties.

3.3.1. *Transition systems and their runs*

Transition systems describes the states, the initial states and the possible state transitions of systems. They provide a general framework for describing the (operational) semantics of reactive systems, independently of concrete formalisms used for their specification.

DEFINITION 3.1.– *A* labeled transition system $\mathcal{T} = (Q, I, E, \delta)$ *is given by:*

– *a set* Q *of* states*;*

– *a subset* $I \subseteq Q$ *of* initial states*;*

– *a set* E *of (action)* labels*;*

– *and a transition relation* $\delta \subseteq Q \times E \times Q$*.*

We require that δ *is a total relation: for every* $q \in Q$ *there exist* $e \in E$ *and* $q' \in Q$ *such that* $(q, e, q') \in \delta$*. An action (label)* $e \in E$ *is called* enabled *at a state* $q \in Q$ *if* $(q, e, q') \in \delta$ *holds for some* $q' \in Q$*.*

A run *of* $\mathcal{T}$ *is an* ω*-sequence* $\rho = q_0 \stackrel{e_0}{\Rightarrow} q_1 \stackrel{e_1}{\Rightarrow} \ldots$ *of states* $q_i \in Q$ *and labels* $e_i \in E$ *such that* $q_0 \in I$ *is an initial state and* $(q_i, e_i, q_{i+1}) \in \delta$ *is a transition for all* $i \in \mathbb{N}$*. A state* $q \in Q$ *is called* reachable *in* $\mathcal{T}$ *if there exists some run* $q_0 \stackrel{e_0}{\Rightarrow} q_1 \stackrel{e_1}{\Rightarrow} \ldots$ *of* $\mathcal{T}$ *such that* $q_n = q$ *for some* $n \in \mathbb{N}$*.*

Definition 3.1 is generic in the sets Q and E: it does not specify the structure of the sets of states and labels of a transition system and can be specialized for different settings. Often, states will be given as variable assignments, and labels represent individual system actions. The latter are particularly useful for specifying fairness

constraints. Another example can be found in the definition of a timed transition system in section 4.2. A transition system is *finite* if its set of states is finite. Throughout this chapter, we will consider verification techniques for finite transition systems.

We have assumed that the transition relation δ is total in order to simplify some of the technical development. In particular, we need only consider infinite system executions rather than distinguish between finite and infinite ones. This hypothesis is easily satisfied by assuming a "stuttering" action $\tau \in E$ with $(q, \tau, q') \in \delta$ if and only if $q = q'$, for all $q, q' \in Q$. In this case, the deadlock states of a transition system are those that only enable the τ transition.

We emphasize that transition systems are a semantic concept for the description of system behavior. In practice, verification models are usually described in modeling languages, including (pseudo) programming languages such as PROMELA, process algebras or Petri nets (see Chapter 1). In general, the size of the transition system corresponding to a system description in some such language will be exponential in the size of its description. Different model checkers are optimized for certain classes of systems such as shared-variable or message passing programs.

Verification algorithms determine whether a transition system satisfies a given property. As we have seen in the elevator example of section 3.2, properties are built from elementary propositions that can be true or false in a system state. The following definition formalizes this idea with the concept of a Kripke structure that extends a transition system with an interpretation of atomic propositions in states.

DEFINITION 3.2.– *Let $\mathcal{V}$ be a set. A Kripke structure $\mathcal{K} = (Q, I, E, \delta, \lambda)$ extends a transition system by a mapping $\lambda : Q \to 2^{\mathcal{V}}$ that associates with every state $q \in Q$ the set of propositions true at state q. The runs of a Kripke structure are just the runs of its underlying transition system.*

The labeling λ of the states of a Kripke structure with sets of atomic propositions allows us to evaluate formulae of propositional logic built from the propositions in $\mathcal{V}$. We write $q \models P$ if state q satisfies propositional formula P. Let us note in passing that in principle, propositional logic is expressive enough for describing state properties of finite transition systems.

3.3.2. *Verification of invariants*

A *system invariant* is a state property P such that $q \models P$ holds for all reachable system states q. Invariants are elementary safety properties. In the particular case of an *inductive invariant*, P is supposed to hold true for every initial state and to be preserved by all transitions, i.e. $q \models P$ and $(q, E, q') \in \delta$ implies that $q \models P$. Of course, every inductive invariant is a system invariant, but the converse is not necessarily true. In particular, a system invariant need not be preserved by transitions from

```
boolean verify_inv(KripkeStructure ks, Predicate inv) {
  Set seen = new Set();
  Set todo = new Set();
  foreach (State i in ks.getInitials()) {
    if (!seen.contains(i)) {
      todo.add(i);
      while (!todo.isEmpty()) {
        State s = todo.getElement();
        todo.remove(s); seen.add(s);
        if (!s.satisfies(inv)) { return false; }
        foreach (State s' in ks.successors(s)) {
          if (!seen.contains(s')) {
            todo.add(s')
  } } } } }
  return true;
}
```

Figure 3.3. *Checking invariants by state enumeration*

non-reachable states. Inductive invariants are the basis for deductive system verification, but less important for model checking.

Verifying that a finite Kripke structure $\mathcal{K}$ satisfies an invariant property P is a conceptually simple problem, and it illustrates well the basic ideas of algorithms for model checking. We can simply enumerate the reachable system states and verify that P is satisfied by every one of them. Termination is guaranteed by the finiteness of $\mathcal{K}$. A basic algorithm for invariant checking is presented in Figure 3.3: the variable `seen` contains the set of states that have already been visited, whereas `todo` represents a set of states that need to be explored. For each state s in `todo`, the algorithm checks whether the predicate holds at s and otherwise aborts the search, returning false. It then adds to `todo` all successors s' of s (with respect to arbitrary labels) that have not yet been seen. The algorithm returns `true` when there are no more states to explore.

In an implementation of this algorithm, the set `seen` will be represented by a hash table so that membership of an element in that set can be decided in quasi-constant time. The set `todo` will typically be represented by a stack or a queue, corresponding to depth-first or breadth-first search. If `todo` is a stack, it is easy to produce a counter-example in case the invariant does not hold: the algorithm can be organized such that when a state s violating the predicate is found, the stack contains a path from an initial state to s, which can be displayed to the user. If `todo` is a queue, we are certain to find invariant violations at a minimum depth in the state-space graph, and this makes it easier for the user to understand why the invariant fails to hold. However, the generation of counter-examples in a queue-based implementation requires an auxiliary data structure for retrieving the predecessor of a state.

SPIN uses the algorithm of Figure 3.3 for the verification of assertions. The user can choose whether depth-first or breadth-first search should be used. There are specialized tools for invariant checking such as Murϕ [DIL 92], and the main challenge in implementing such a tool is to be able to search large state spaces, beyond 10^6–10^7 states. For such system sizes, the set `seen` can no longer be stored in the main memory. Although the disk can be used in principle, suitable access strategies must be found to simulate random access, and verification will be slowed down considerably. Combinations of the three following principles are useful for the analysis of large systems in practice (see also section 3.6 for an additional discussion of these techniques):

– *Compression techniques* rely on compact representations of data structures such as states and sets of states in memory. For example, the implementation may decide to store state signatures (hash codes) instead of proper states in the set `seen`. A collision between the signatures of two distinct states could then lead to cutting off parts of the search space, hence possibly missing invariant violations. However, any error reported by the algorithm would still correspond to a valid counter-example. By estimating the probability of collisions and using different hash functions during several runs over the same model, we can estimate the coverage of verification and improve the reliability of the result.

– *Reduction-based techniques* attempt to determine a subset of the runs whose exploration guarantees the correctness of the system invariant over the overall system. In particular, independence of different transitions enabled in a given state or symmetry relations among data or system parameters can significantly reduce the number of runs that need to be explored by the algorithm.

– Finally, *abstraction* can help to construct a significantly smaller model that can be verified exhaustively and whose correctness guarantees the correctness of the original model. Whereas abstractions are usually constructed during system modeling in an ad-hoc manner, this relation can be formalized and the construction of abstractions can be done automatically. In general, failure of verification over an abstract model does not imply that the invariant does not hold over the original model because the abstraction could have identified reachable and unreachable states of the original model. However, a spurious counter-example produced over an abstract model often helps to suggest an improvement of the abstraction.

3.4. Temporal logic

Given a Kripke structure $\mathcal{K}$, we are interested in their properties, such as:

– Does (the reachable part of) $\mathcal{K}$ contain "bad" states, such as deadlock states where only the τ action is enabled, or states that do not satisfy an invariant?

– Are there executions of $\mathcal{K}$ such that, after some time, a "good" state is never reached or a certain action never executed? Such executions correspond to *livelocks* where some process does not make progress yet the entire system is not blocked.

– Can the system be re-initialized? In other words, is it always possible to reach an initial system state?

Temporal logic provides a language in which such properties can be formulated. Different temporal logics can be distinguished according to their syntactic features, or to the semantic structures over which their formulae are evaluated. Linear time temporal logic and branching time temporal logic are the two main kinds of temporal logic, and they will be introduced in the following. We will then present some principles of the theory of ω-automata and its connection with linear time temporal logic. This correspondence will be useful for the presentation of a model checking algorithm in section 3.5.

3.4.1. *Linear time temporal logic*

Linear time temporal logic extends classical logic by temporal modalities to refer to different (future or past) time points. Its formulae are interpreted over infinite sequences of states, such as the runs of Kripke structures from which the labels have been omitted. We will consider here a propositional version PTL of this logic. As before, we assume given a set $\mathcal{V}$ of atomic propositions. The interpretation of PTL is based on a function $\lambda : Q \to 2^{\mathcal{V}}$ that evaluates the atomic propositions over states, just as in the definition 3.2 of a Kripke structure. For a sequence $\sigma = q_0 q_1 \ldots$ of states, we denote by σ_i the state q_i, and by $\sigma|_i$ the suffix $q_i q_{i+1} \ldots$ of σ.

DEFINITION 3.3.– *Formulae of the logic* PTL *and their semantics over sequences $\sigma = q_0 q_1 \ldots$ of states (with respect to a function $\lambda : Q \to 2^{\mathcal{V}}$) are inductively defined as follows:*

– an atomic proposition $v \in \mathcal{V}$ is a formula and $\sigma \models v$ iff $v \in \lambda(\sigma_0)$,

– propositional combinations of formulae (by means of the operators $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$) are formulae and their semantics are the usual ones,

– if φ is a formula then so is $\mathrm{X}\,\varphi$ ("next φ") and $\sigma \models \mathrm{X}\,\varphi$ if and only if $\sigma|_1 \models \varphi$,

– if φ and ψ are formulae then $\varphi\ \mathrm{U}\ \psi$ ("φ until ψ") is a formula and $\sigma \models \varphi\ \mathrm{U}\ \psi$ if and only if there exists some $k \in \mathbb{N}$ such that $\sigma|_k \models \psi$ and $\sigma|_i \models \varphi$ for all i with $0 \leqslant i < k$.

The set $Mod(\varphi)$ of the models *of a formula φ of* PTL *is the set of those state sequences σ such that $\sigma \models \varphi$.*

Formula φ is valid *if $\sigma \models \varphi$ holds for all σ. It is* satisfiable *if $\sigma \models \varphi$ holds for some σ. Formula φ is* valid in a Kripke structure *$\mathcal{K}$, written $\mathcal{K} \models \varphi$, if $\sigma \models \varphi$ holds for all runs σ of $\mathcal{K}$.*

The formula $\varphi\ \mathrm{U}\ \psi$ requires that ψ eventually becomes true and that φ holds at least until that happens. Other useful formulae are defined as abbreviations. Thus, $\mathrm{F}\,\varphi$

("eventually φ") is defined as $\mathrm{true}\ \mathrm{U}\ \varphi$, and holds true of σ if φ is true of some suffix of σ. The dual formula $\mathrm{G}\,\varphi$ ("always φ") is defined as $\neg\,\mathrm{F}\,\neg\varphi$ and requires that φ holds true of all suffixes of σ. Finally, the formula $\varphi\ \mathrm{W}\ \psi$ ("φ unless ψ") abbreviates $(\varphi\ \mathrm{U}\ \psi) \vee \mathrm{G}\,\varphi$. It states that φ stays true while ψ is false: if ψ remains false forever then φ must hold true of all suffixes.

The PTL formula $\mathrm{G}\,\mathrm{F}\,\varphi$ asserts that for every suffix $\sigma|_i$ there exists a suffix $\sigma|_j$, where $j \geqslant i$, that satisfies φ. In other words, φ has to be true infinitely often. Dually, the formula $\mathrm{F}\,\mathrm{G}\,\varphi$ asserts that φ will eventually stay true.

The notions of (general) validity and satisfiability of PTL are standard. More important for the purposes of model checking is the notion of *system validity*: a property expressed by a PTL formula holds for a system if it is true for all of its runs. For example, the following formulae express typical correctness properties of a system for managing a resource shared between two processes. In writing these properties, we assume that the propositions req_i and own_i (for $i = 1, 2$) represent the states in which process i has requested (respectively, obtained) the resource.

$\mathbf{G}\,\neg(own_1 \wedge own_2)$. This formula describes mutual exclusion for access to the resource: at no moment do both processes own the resource. More generally, formulae of the form $\mathrm{G}\,P$, for a non-temporal formula P, express invariants.

$\mathbf{G}(req_1 \Rightarrow \mathbf{F}\ own_1)$. Every request of access to the resource by process 1 will eventually be honored in the sense that process 1 will be granted access to the resource. Formulae of the form $\mathrm{G}(P \Rightarrow \mathrm{F}\,Q)$, for non-temporal formulae P and Q, are often called *response properties* [MAN 90]. We have encountered a formula of this form in the lift example; see Formula (3.1).

$\mathbf{G}\,\mathbf{F}(req_1 \wedge \neg(own_1 \vee own_2)) \Rightarrow \mathbf{G}\,\mathbf{F}\ own_1$. This formula is weaker than the preceding one as it requires process 1 to obtain the resource infinitely often provided that it is requested infinitely often at a time when the resource is free. For example, the previous property is impossible to satisfy (together with the basic requirement of mutual exclusion) if the second process never releases the resource after it obtained it, whereas the present property states no obligation in such a case. Formulae of the shape $\mathrm{G}\,\mathrm{F}\,P \Rightarrow \mathrm{G}\,\mathrm{F}\,Q$ can also be used to express assumptions of (strong) fairness.

$\mathbf{G}(req_1 \wedge req_2 \Rightarrow (\neg owns_2\ \mathbf{W}\ (owns_2\ \mathbf{W}\ (\neg owns_2\ \mathbf{W}\ owns_1))))$. When both processes request the shared resource, process 2 will obtain the resource at most once before process 2 gets access. This property, called *one-bounded overtaking*, is an example of a precedence property, as it expresses requirements on the relative ordering of events. Intuitively, the formula asserts the existence of four (possibly empty or infinite) intervals during which the different propositions hold.

EXTENSIONS OF PTL.– The logic PTL is generally recognized as a useful language for formulating correctness properties of systems. Nevertheless, its expressive power is limited, and several authors have proposed extensions of PTL by additional operators. In particular, the modalities of PTL are all directed towards the future, and one can obviously define symmetric operators such as $\mathrm{P}\,\varphi$, which asserts that φ was true at some preceding instant of time. Kamp [KAM 68] has shown that this extension does not increase the expressive power of the logic (see also [GAB 94] for generalizations of this result) as far as system validity is concerned: every formula containing past time modalities can be rewritten into a corresponding "future-only" formula that describes the same property. For example, the formula $\mathrm{G}(own_1 \Rightarrow \mathrm{P}\,req_1)$ which asserts that the resource is obtained only in response to a preceding request is equivalent with respect to system validity to the formula $\neg own_1\ \mathrm{W}\ req_1$. This observation has been used to justify the elimination of past time operators from model checking tools. Nevertheless, past time modalities can increase the readability of formulae, and they can be (exponentially) more succinct [LAR 02].

PTL can also be extended by operators corresponding to regular expressions [WOL 83] or, equivalently, by operators defined as smallest or largest fixed points or by quantification over atomic propositions. We will observe a close relationship between PTL and (ω-)regular languages in sections 3.4.3 and 3.4.4, which explains that "regular" operators can be added to PTL without significant complications in model checking algorithms. The language PSL [PSL 04] (*property specification language*), which has been recognized as an IEEE standard in 2005, is based on an extension of PTL by past time and regular modalities.

3.4.2. *Branching time temporal logic*

Whereas formulae of linear time temporal logic are evaluated over the runs of a system, branching time temporal logic makes assertion about the system as a whole. In particular, besides expressing properties that should be true for all system executions it is also possible to assert the existence of runs satisfying certain conditions. For example, the property of reinitializability states that for every reachable state there exists some path leading back to an initial state. The following definition introduces the branching time logic CTL (*computation tree logic*), which has been very popular for model checking.

DEFINITION 3.4.– *The formulae of the logic* CTL *and their semantics, with respect to a state q of a Kripke structure $\mathcal{K}$, are defined as follows:*

– an atomic proposition $v \in \mathcal{V}$ is a formula and $\mathcal{K}, q \models v$ if $v \in \lambda(q)$,

– propositional combinations of formulae (by means of the operators $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$) are formulae and their semantics are the usual ones,

– if φ is a formula then so is $\mathrm{EX}\,\varphi$ and $\mathcal{K}, q \models \mathrm{EX}\,\varphi$ if and only if there exist e and q' with $(q, e, q') \in \delta$ and $\mathcal{K}, q' \models \varphi$,

– if φ is a formula then so is EG φ *and* $\mathcal{K}, q \models$ EG φ *if and only if there exists some path* $q = q_0 \stackrel{e_0}{\Rightarrow} q_1 \ldots$ *such that* $\mathcal{K}, q_i \models \varphi$ *for all* $i \in \mathbb{N}$*,*

– if φ and ψ are formulae then φ EU ψ *is a formula and* $\mathcal{K}, q \models \varphi$ EU ψ *if and only if there exists some path* $q = q_0 \stackrel{e_0}{\Rightarrow} q_1 \ldots$ *and some* $k \in \mathbb{N}$ *such that* $\mathcal{K}, q_k \models \psi$ *and* $\mathcal{K}, q_i \models \varphi$ *for all* i *with* $0 \leqslant i < k$*.*

The satisfaction set $[\![\varphi]\!]_{\mathcal{K}}$ *of a formula φ in a Kripke structure $\mathcal{K}$ is the set of all states $q \in Q$ such that $\mathcal{K}, q \models \varphi$. Formula φ is* valid in a Kripke structure $\mathcal{K}$*, written* $\mathcal{K} \models \varphi$*, if $\mathcal{K}, q \models \varphi$ for all initial states $q \in I$, that is, if $I \subseteq [\![\varphi]\!]_{\mathcal{K}}$.*

Formula φ is valid *(*satisfiable*) if it is valid in every (some) Kripke structure $\mathcal{K}$.*

The modal operators of CTL combine temporal references with quantification over paths. Further connectives can be defined as abbreviations. Thus, EF φ abbreviates true EU φ and asserts that φ will become true along some path starting at the state of evaluation. The formulae AX φ and AG φ are defined as $\neg$ EX $\neg\varphi$ and $\neg$ EF $\neg\varphi$, and state that φ holds at all immediate successors, respectively at all states reachable from the state of evaluation. Similar definitions can be given to introduce the operators AF, AU, and AW. Operators of the shape A _ express *universal properties* that have to hold along all possible paths starting at the current state, whereas the operators E _ express *existential properties*. In particular, AG P asserts that the non-temporal formula P is a system invariant. The formula AG$(req_1 \Rightarrow$ EF $own_1)$ requires that every request for the resource by process 1 may be followed by getting access to the resource, although there may also be some executions along which the process is never granted access. Similarly, system validity of the formula AG EF $init$ (for a suitable proposition $init$) means that the system can be reinitialized from every reachable state.

Although they are interpreted over structures of different shape, the expressiveness of PTL and CTL can be compared based on the notion of system validity: a PTL formula φ and a CTL formula ψ correspond to each other if they are valid in the same Kripke structures. Obviously, PTL cannot define existential properties such as AG EF $init$. Perhaps more surprisingly, there also exist PTL formulae for which there is no corresponding CTL formula, and therefore the expressive power of PTL and CTL is incomparable [LAM 80]. Thus, reactivity properties G F $P \Rightarrow$ G F Q do not have a counterpart in CTL. A simpler example is provided by the Kripke structure $\mathcal{K}$ shown in Figure 3.4 (transition labels have been omitted). It is easy to see that $\mathcal{K} \models$ F G v: that property is true for the run of $\mathcal{K}$ that loops at q_0, but also for the executions that eventually move to state q_2. On the other hand, the CTL formula[1] AF AG v is not valid in the Kripke structure $\mathcal{K}$. To see this, it is enough to observe that the formula AG v

1. Obviously, this observation does not prove that there is no other CTL formula that corresponds to F G v. The proof of this assertion requires a more detailed argument [LAM 80].

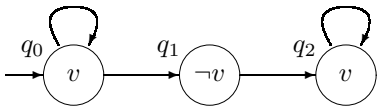


Figure 3.4. *A Kripke structure satisfying* F G v *but not* AF AG v

is never satisfied along the run of $\mathcal{K}$ that loops at q_0 because from q_0 we can always move to state q_1, which does not satisfy v.

OTHER BRANCHING TIME LOGICS.– This relative lack of expressiveness of CTL is due to the strict alternation between path quantifiers (A, E) and temporal operators (X, G, F, U, W). The logic CTL* relaxes this constraint and (strictly) subsumes the logics PTL and CTL. For example, the CTL* formula A F G v corresponds to the PTL formula F G v. See [EME 90, EME 86a, VAR 01] for more in-depth comparisons between linear time and branching time temporal logics.

The *modal μ-calculus* [KOZ 83, STI 01] is based on defining temporal connectives by their characteristic recursive equivalences. For example, let us consider

$$\mathrm{EG}\,\varphi \;\Leftrightarrow\; \varphi \wedge \mathrm{EX}\,\mathrm{EG}\,\varphi \qquad \text{and} \qquad \mathrm{EG}_2\,\varphi \;\Leftrightarrow\; \varphi \wedge \mathrm{EX}\,\mathrm{EX}\,\mathrm{EG}_2\,\varphi.$$

The left-hand equivalence is valid for the connective EG of CTL, the one on the right-hand side characterizes an operator EG_2 that requires the existence of a path such that φ is true at all states with an even distance from the original point of evaluation. (Formula φ can be true or false at the other states.) It can be shown that such an operator is not definable in CTL or CTL*. In the μ-calculus it can be defined by the formula $\nu X : \varphi \wedge \mathrm{EX}\,\mathrm{EX}\,X$.

The temporal logic ATL (*alternating time temporal logic*) [ALU 97] refines the quantification over the paths in a Kripke structure by refering to the labels of transitions, interpreted as identifying the processes of a system. For example, ATL formulae can be written to express that the controller by itself can guarantee the mutual exclusion of access to the resource, or that the controller and the first process can conspire to exclude the second process from accessing it. The logic ATL is therefore particularly useful for the specification and analysis of open systems formed as the composition of independent components.

3.4.3. *ω-automata*

The algorithm for the verification of invariants presented in section 3.3.2 is conceptually simple, but it is not immediately clear how to generalize it for the verification of arbitrary temporal properties. Transition systems, even if they are finite,

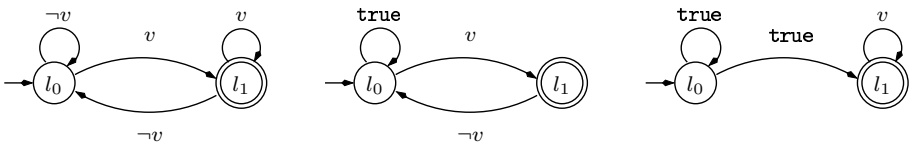


Figure 3.5. *Three Büchi automata*

usually generate an infinite set of runs, each of which is an infinite sequence of states. Certain verification algorithms are grounded in a close correspondence between temporal logics and finite automata operating on infinite objects (sequences or trees). In contrast to the more "declarative" presentation of properties by formulae, automata provide a more "operational" description, and are amenable to decision procedures. We are now going to study the principles of this theory, which goes back to the work of Büchi [BÜC 62], Muller [MUL 63], and Rabin [RAB 69], but limit the discussion mainly to Büchi automata. Much more detailed information can be found in the excellent articles by Thomas [THO 97] or Vardi *et al.* [VAR 95, KUP 94]. As before, we assume given a set $\mathcal{V}$ of atomic propositions, which defines the alphabet $2^{\mathcal{V}}$ of our automata.

DEFINITION 3.5.– *A* Büchi automaton $\mathcal{B} = (L, L_0, \delta, F)$ *is given by a finite set* L *of locations, a set* $L_0 \subseteq L$ *of initial locations, a relation* $\delta \subseteq L \times 2^{\mathcal{V}} \times L$ *and a set* $F \subseteq L$ *of accepting locations.*

A run *of* $\mathcal{B}$ *over a sequence* $\sigma = s_0 s_1 \ldots$ *where* $s_i \subseteq 2^{\mathcal{V}}$ *is a sequence* $\rho = l_0 l_1 \ldots$ *of locations* $l_i \in L$ *such that* $l_0 \in L_0$ *is an initial location and* $(l_i, s_i, l_{i+1}) \in \delta$ *holds for all* $i \in \mathbb{N}$*. The run* ρ *is* accepting*if it contains an infinite number of locations* $l_k \in F$*.*

The language $\mathcal{L}(\mathcal{B})$ *of* $\mathcal{B}$ *is the set of sequences* σ *for which there exists an accepting run of* $\mathcal{B}$*.*

Observe that the structure of a Büchi automaton is just that of a non-deterministic finite automaton (NFA). The only difference is in the definition of the acceptance condition which requires that the run passes infinitely often through an accepting location. An ω-language L (i.e., a set of ω-sequences σ) is called *ω-regular* if it is generated by a Büchi automaton, i.e. if $L = \mathcal{L}(\mathcal{B})$ for some Büchi automaton $\mathcal{B}$.

Büchi automata, as we have defined them, operate on ω-sequences of subsets of $\mathcal{V}$, and this is in close correspondence with the interpretation of (linear time) temporal logic over Kripke structures. Indeed, any run $q_0 \stackrel{e_0}{\Rightarrow} q_1 \stackrel{e_1}{\Rightarrow} \ldots$ of a Kripke structure $\mathcal{K}$ can be identified with the corresponding sequence $\lambda(q_0)\lambda(q_1)\ldots$ where λ is the propositional valuation of states of $\mathcal{K}$. In this sense, PTL formulae and Büchi automata operate over the same class of structures.

Figure 3.5 shows three Büchi automata, each of which contains two locations l_0 and l_1, of which l_0 is initial and l_1 is accepting. The transitions are labeled with

propositional formulae in an obvious manner: for example, the transition label $\neg v$ represents those subsets of $\mathcal{V}$ that do not contain v. The left-hand automaton is deterministic because for any propositional interpretation $s \subseteq \mathcal{V}$, the successor of both locations is uniquely determined. It accepts precisely those sequences where v is true infinitely often and, intuitively, corresponds to the PTL formula $\mathrm{G\,F}\, v$. The middle automaton is non-deterministic because there is a choice between staying at l_0 or moving to l_1 when reading an interpretation satisfying v while in location l_0. Any sequence accepted by this automaton must contain an infinite number of interpretations satisfying v, followed by an interpretation satisfying $\neg v$. The language of this automaton is therefore characterized by the PTL formula[2] $\mathrm{G\,F}(v \wedge \mathrm{X}\,\neg v)$. It is not difficult to find a deterministic Büchi automaton that accepts the same language. The right-hand automaton is also non-deterministic, and any sequence accepted by it must terminate in a sequence of states satisfying v. This language is also described by the PTL formula $\mathrm{F\,G}\, v$. It can be shown [THO 97] that there is no deterministic Büchi automaton defining this language. Intuitively, the reason is that an infinite "prophecy" is required when choosing to move to location l_1.

In particular, and unlike standard non-deterministic finite automata over finite words, non-deterministic Büchi automata are strictly more expressive than deterministic ones. Apart from this difference, the theory of ω-regular languages closely parallels that of ordinary regular languages. Specifically, we will make use of the decidability of the emptiness problem.

THEOREM 3.1.– *For a Büchi automaton $\mathcal{B} = (L, L_0, \delta, F)$, the emptiness problem $\mathcal{L}(\mathcal{B}) = \emptyset$ can be decided in time linear in the size of the automaton.*

PROOF.– Since L is a finite set, $\mathcal{L}(\mathcal{B}) = \emptyset$ if and only if there exists locations $l_0 \in L_0$ and $l_f \in F$ such that l_f is reachable from l_0 and (non-trivially) from itself, i.e. there exist finite words x and $y \neq \varepsilon$ over $2^{\mathcal{V}}$ such that $l_0 \stackrel{x}{\Longrightarrow} l_f$ and $l_f \stackrel{y}{\Longrightarrow} l_f$. The existence of such a cycle in the graph of $\mathcal{B}$ can be decided in linear time, for example by using the algorithm of Tarjan and Paige [TAR 72] that enumerates the strongly connected components of $\mathcal{B}$ and checking that some SCC contains an accepting location. QED

It follows from the proof of Theorem 3.1 that every non-empty ω-regular language contains a word of the form xy^ω where x and y are finite words and y^ω denotes infinite repetition of the word y.

Further important results about ω-regular languages show closure under Boolean operation and projection. Closure under union and projection are easy to prove, using essentially the same automaton constructions as for the corresponding results for NFA

2. Let us note that an equivalent formula is $\mathrm{G\,F}\, v \wedge \mathrm{G\,F}\, \neg v$.

over finite words. Closure under intersection is essentially proved by constructing the product automata for the two original languages, although some care must be taken to define the acceptance condition. However, proving closure under complementation is difficult. The standard proof known from NFA first constructs a deterministic finite automaton, and this fails for Büchi automata because they cannot be determinized. The original proof by Büchi was non-constructive and combinatorial in nature, and a series of papers over the following 25 years established explicit constructions, culminating in optimal constructions of complexity $O(2^{n \log n})$ by Safra [SAF 88] and by Kupferman and Vardi [KUP 97b].

OTHER TYPES OF ω-AUTOMATA.– There exist many other types of non-deterministic ω-automata that differ essentially in the definition of the acceptance condition. In particular, generalized Büchi automata are of the shape $\mathcal{B} = (L, L_0, \delta, \mathcal{F})$, with an acceptance condition $\mathcal{F} = \{F_1, \ldots, F_n\}$ of sets F_i of locations. A run is accepted if every set F_i is visited infinitely often. Using a counter modulo n, it is not hard to simulate a generalized Büchi automaton by a standard one [VAR 94]. For Muller automata, the acceptance condition is also a set $\mathcal{F} \subseteq 2^L$, and a run is accepted if the set of all locations that appear infinitely often is an element of $\mathcal{F}$. Muller automata again define the class of ω-regular languages. Rabin and Streett automata are special cases of Muller automata. Beyond independent interest in these classes of automata, they are used in Safra's complementation proof. The more recent complementation by Kupferman and Vardi extends rather smoothly to different kinds of non-deterministic automata [KUP 05].

Alternating automata [MUL 88, VAR 95, KUP 97b] differ from Büchi automata in that several locations can be simultaneously active during a run. The transition relation of an alternating automaton can be specified using propositional formulae built from the propositions in $\mathcal{V}$ and the locations, where the latter are restricted to occur positively. For example,

$$\delta(q_1) \;=\; \big(v \wedge w \wedge q_1 \wedge (q_3 \vee q_4)\big) \;\vee\; \big(\neg w \wedge (q_1 \vee q_2)\big),$$

specifies that if the location q_1 is currently active and the current interpretation satisfies $v \wedge w$, then q_1 and q_3 or q_1 and q_4 will be activated after the transition. If the current interpretation satisfies $\neg w$, then q_1 or q_2 will be active. Otherwise, the automaton blocks. Alternating automata thus combine the non-determinism of Büchi automata and parallelism, and their runs are infinite trees labeled with locations (or directed acyclic graphs, *dags*, if multiple copies of the same location are merged). With suitable acceptance conditions, these automata again define ω-regular languages but they can be exponentially more succinct than Büchi automata. On the other hand, the emptiness problem becomes of exponential complexity, and this trade-off can be useful in certain model checking applications [HAM 05]. Alternating automata are closely related to certain logical games, which have also received much attention during recent years.

3.4.4. *Automata and* PTL

We have already pointed out some (informal) correspondences between automata and PTL formulae, and indeed formula φ can be considered as defining the ω-language $Mod(\varphi)$. It is therefore quite natural to compare the expressive power of formulae and automata, and we will now sketch the construction of a generalized Büchi automaton $\mathcal{B}_\varphi$ that specifically accepts the models of the PTL formula φ.

The construction avoids the difficult complementation of Büchi automata by using a "global" algorithm that considers all subformulas of φ simultaneously. More precisely, let $\mathcal{C}(\varphi)$ denote the set of subformulae ψ of φ and their complements $\overline{\psi}$, identifying $\neg\neg\psi$ and ψ. It is easy to see that the size of set $\mathcal{C}(\varphi)$ is linear in the length of formula φ.

The locations of $\mathcal{B}_\varphi$ correspond to subsets of $\mathcal{C}(\varphi)$, with the intuitive meaning that whenever $\rho = l_0 l_1 \ldots$ is an accepting run of $\mathcal{B}_\varphi$ over σ then $\sigma|_i$ satisfies all formulae in l_i, for all $i \in \mathbb{N}$. Formally, the set L of locations of $\mathcal{B}_\varphi$ consists of all sets $L \subseteq \mathcal{C}(\varphi)$ that satisfy the following "healthiness conditions":

– for all formulae $\psi \in \mathcal{C}(\varphi)$, either $\psi \in L$ or $\overline{\psi} \in L$ but not both;

– if $\psi_1 \vee \psi_2 \in \mathcal{C}(\varphi)$, then $\psi_1 \vee \psi_2 \in l$ if and only if $\psi_1 \in l$ or $\psi_2 \in l$;

– similar conditions hold for the other propositional connectives;

– if $\psi_1 \mathrm{U}\ \psi_2 \in l$, then $\psi_1 \in l$ or $\psi_2 \in l$;

– if $\neg(\psi_1 \mathrm{U}\ \psi_2) \in l$, then $\overline{\psi_2} \in l$.

The initial locations of $\mathcal{B}_\varphi$ are those locations containing the formula φ. The transition relation δ of $\mathcal{B}_\varphi$ consists of the triples (l, s, l') that satisfy the following conditions:

– $s = l \cap \mathcal{V}$: the state s must satisfy precisely those atomic propositions "promised" by the source location l;

– if $\mathrm{X}\,\psi \in l$, then $\psi \in l'$ and if $\neg\,\mathrm{X}\,\psi \in l$, then $\overline{\psi} \in l'$;

– if $\psi_1 \mathrm{U}\ \psi_2 \in l$ and $\overline{\psi_2} \in l$, then $\psi_1 \mathrm{U}\ \psi_2 \in l'$;

– if $\neg(\psi_1 \mathrm{U}\ \psi_2) \in l$ and $\psi_1 \in l$, then $\neg(\psi_1 \mathrm{U}\ \psi_2) \in l'$.

The last two conditions can be explained by the "recursion law" for the U operator:

$$\psi_1 \mathrm{U}\ \psi_2 \ \Leftrightarrow\ \psi_2 \vee (\psi_1 \wedge \mathrm{X}(\psi_1 \mathrm{U}\ \psi_2)). \tag{3.2}$$

The conditions above define the initial locations and the transition relation, it remains to define the acceptance condition of the generalized Büchi automaton $\mathcal{B}_\varphi$. The intuitive idea is to make sure that whenever a location that contains formula $\psi_1 \mathrm{U}\ \psi_2$ appears in an accepting run of $\mathcal{B}_\varphi$ it will be followed by a location containing ψ_2. Let

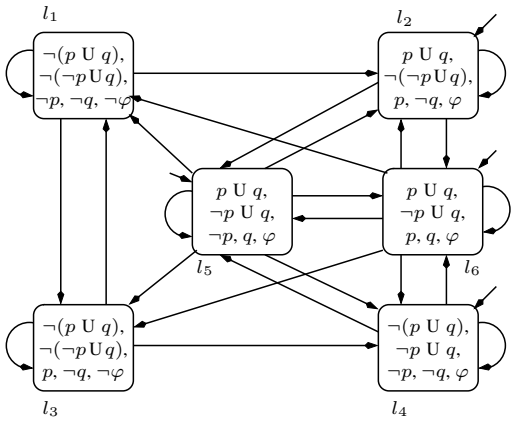


Figure 3.6. *Büchi automaton $\mathcal{B}_\varphi$ for $\varphi \equiv (p \text{ U } q) \vee (\neg p \text{ U } q)$*

therefore $\psi_1^1 \text{ U } \psi_2^1, \ldots, \psi_1^k \text{ U } \psi_2^k$ be all formulae of this shape in $\mathcal{C}(\varphi)$. The acceptance condition $\mathcal{F} = \{F_1, \ldots F_k\}$ of automaton $\mathcal{B}_\varphi$ contains a set of locations F_i for each formula $\psi_1^i \text{ U } \psi_2^i$, where $l \in F_i$ if and only if $\psi_2^i \in l$ or $\psi_1^i \text{ U } \psi_2^i \notin l$.

For example, the automaton shown in Figure 3.6 results from an application of the above construction to the formula $\varphi \equiv (p \text{ U } q) \vee (\neg p \text{ U } q)$. We have omitted the transition labels, which are just given by the sets of atomic propositions that appear in the source states. The acceptance condition consists of sets $F_1 = \{l_1, l_3, l_4, l_5, l_6\}$ and $F_2 = \{l_1, l_2, l_3, l_5, l_6\}$, corresponding to the two subformulae $p \text{ U } q$ and $\neg p \text{ U } q$. For example, set F_1 serves to exclude runs that terminate in the loop at location l_2 because these runs "promise" $p \text{ U } q$ without ever satisfying q.

The above construction of the automaton $\mathcal{B}_\varphi$ is similar to that of a tableau for the logic PTL [WOL 83]. The following correctness theorem is due to Vardi *et al.*, see for example [VAR 94, GER 95].

PROPOSITION 3.2.– *For any* PTL *formula φ of length n there is a Büchi automaton $\mathcal{B}_\varphi$ with $2^{O(n)}$ locations such that $\mathcal{L}(\mathcal{B}_\varphi) = Mod(\varphi)$.*

Together, Theorem 3.1 and Proposition 3.2 imply that the satisfiability and validity problems of the logic PTL are decidable in exponential time: formula φ is satisfiable if and only if $\mathcal{L}(\mathcal{B}_\varphi) \neq \emptyset$, and it is valid if and only if $\mathcal{L}(\mathcal{B}_{\neg\varphi}) = \emptyset$. On the other hand, Sistla and Clarke [SIS 87] have shown that these problems are PSPACE-complete, one can therefore not hope for a significantly more efficient algorithm in the general case. However, the simple construction of $\mathcal{B}_\varphi$ described above systematically generates an automaton of exponential size. Constructions used in practice [DAN 99, GAS 01] attempt to avoid this blowup whenever possible. Let us finally note that the construction of an alternating automaton for a PTL formula is of linear complexity [VAR 95].

Proposition 3.2 naturally leads to the reciprocal question whether every ω-regular language can be defined by a PTL formula. The answer was already given by Kamp [KAM 68] in 1968: he showed that PTL corresponds to the monadic first-order theory of linear orders. This logical language only contains unary predicate (and no function) symbols, the equality predicate $=$, the relation $<$, interpreted over the natural numbers; see for example [GAB 94, THO 97]. However, Büchi [BÜC 62] proved that the class of ω-regular languages correspond to the analogous fragment of second-order logic, and this language is strictly more expressive. For example, the linear-time counterpart G_2 to the connective EG_2 considered at the end of section 3.4.2 where $G_2\,\varphi$ is true for σ if φ holds for all suffixes $\sigma|_{2n}$ with even offset is definable by a Büchi automaton but not by a PTL formula [WOL 83]. Starting from this observation, extensions of PTL by "grammar operators" that directly correspond to ω-regular expressions [WOL 83], by fixed-point definitions similar to the modal μ-calculus [STI 01], or by quantification over atomic propositions, have been proposed.

Automata corresponding to branching time temporal logics can also be defined, and they operate over infinite trees [KUP 94, THO 97]. We do not present the details here because they are not necessary for the model checking algorithms for branching-time logics that we will describe in section 3.5.2.

3.5. Model checking algorithms

Given a Kripke structure $\mathcal{K}$ and a formula φ of temporal logic, the model checking problem is to determine whether φ is valid in $\mathcal{K}$, written $\mathcal{K} \models \varphi$ (see Definitions 3.3 and 3.4). Beyond a yes/no answer to this question, model checking tools generally attempt to explain their verdict. For example, a PTL model checker will produce a counter-example, i.e. a run of $\mathcal{K}$ that does not satisfy φ if $\mathcal{K} \not\models \varphi$. The model checking problems for the logics that we have considered so far are decidable when $\mathcal{K}$ is a finite-state system, and we will explain in this section the principles of model checking algorithms for PTL and for CTL.

Observe that the model checking problem has two parameters, $\mathcal{K}$ and φ. We can therefore imagine two basic strategies for its solution: *global* algorithms recurse on the syntax of formula φ and evaluate its subformulae at the states of $\mathcal{K}$ in order to determine the satisfaction of φ. *Local* algorithms recurse on the structure of $\mathcal{K}$: they explore the parts of $\mathcal{K}$ that contribute to evaluating φ, much like how the algorithm of section 3.3.2 uses graph search to determine satisfaction of a proposed invariant. Global algorithms are traditionally used for CTL model checking, while PTL model checking is mostly based on local algorithms because system validity for PTL does not decompose along the formula structure. For example, $\mathcal{K} \models \varphi \vee \psi$ does not require either $\mathcal{K} \models \varphi$ or $\mathcal{K} \models \psi$.

3.5.1. *Local* PTL *model checking*

The translation from PTL formulae to Büchi automata introduced in section 3.4.4 provides the basis for a PTL model checking algorithm, refining the procedure for satisfiability checking described there. Indeed, $\mathcal{K} \not\models \varphi$ if and only if there exists a run of $\mathcal{K}$ that does not satisfy φ. If we consider $\mathcal{K}$ as a Büchi automaton (with trivial acceptance condition) that defines the set $\mathcal{L}(\mathcal{K})$ of runs of $\mathcal{K}$, we arrive at the following chain of equivalences:

$$\mathcal{K} \not\models \varphi \quad \Longleftrightarrow \quad \mathcal{L}(\mathcal{K}) \cap \mathit{Mod}(\neg\varphi) \neq \emptyset \quad \Longleftrightarrow \quad \mathcal{L}(\mathcal{K}) \cap \mathcal{L}(\mathcal{B}_{\neg\varphi}) \neq \emptyset,$$

where the second equivalence is justified by Proposition 3.2. The last problem in this chain can be solved by determining language emptiness for the product of $\mathcal{K}$ and $\mathcal{B}_{\neg\varphi}$.

More formally, assume that $\mathcal{K} = (Q, I, E, \delta_{\mathcal{K}}, \lambda)$ is a Kripke structure and that $\mathcal{B}_{\neg\varphi} = (L, L_0, \delta_{\mathcal{B}}, F)$ is the Büchi automaton[3] corresponding to the negation of φ. The model checking algorithm operates on pairs (q, l) of states q of $\mathcal{K}$ and locations l of $\mathcal{B}_{\neg\varphi}$. The initial pairs consist of initial states and locations of the two components. The successors of a pair (q, l) are all pairs (q', l') such that

– q' is a successor state of q in $\mathcal{K}$, i.e. $(q, e, q') \in \delta_{\mathcal{K}}$ for some $e \in E$, and

– l' is a possible successor of l under the interpretation of the atomic propositions determined by q, i.e. $(l, \lambda(q), l') \in \delta_{\mathcal{B}}$.

The two automata therefore take simultaneous transitions. The acceptance condition of the product is determined by that of the Büchi automaton: a pair (q, l) is accepting if $l \in F$ is an accepting state of $\mathcal{B}_{\neg\varphi}$.

The Büchi automaton $\mathcal{A}$ thus defined recognizes the language $\mathcal{L}(\mathcal{K}) \cap \mathcal{L}(\mathcal{B}_{\neg\varphi})$, and in particular $\mathcal{K} \models \varphi$ if and only if $\mathcal{L}(\mathcal{A}) = \emptyset$. By Theorem 3.1, this condition can be verified in linear time in the size of $\mathcal{A}$ by searching for an accepting cycle. Because the size of $\mathcal{A}$ is proportional to the product of the sizes of $\mathcal{K}$ and of $\mathcal{B}_{\neg\varphi}$, the explicit construction of $\mathcal{A}$ is prohibitive in practice. Courcoubetis *et al.* [COU 92] invented an algorithm that constructs $\mathcal{A}$ "on the fly", that is, during the search for an acceptance cycle. This algorithm appears in Figure 3.7. The current state of the search is represented by a stack of pairs. The algorithm combines two depth-first search algorithms: starting from an initial pair, the procedure `dfs` explores a path of the product automaton. When this search completes, it backtracks to the last accepting pair q and then initializes a second search (indicated by the parameter `search_cycle` set to true) of a path that leads back to q. Courcoubetis *et al.* proved that this algorithm

3. Here we assume that $\mathcal{B}_{\neg\varphi}$ is an ordinary (not generalized) Büchi automata. As noted in section 3.4.3, the translation from generalized Büchi automata to ordinary Büchi automata is of polynomial complexity.

```
void mc_ptl(Kripke ks, Buchi aut) {
   void dfs(boolean search_cycle) {
     Pair p = stack.top();
     foreach (Pair q in p.successors()) {
        if (search_cycle && (q == seed))
        { report acceptance cycle and exit }
        if (!seen.contains(q, search_cycle)) {
           stack.push(q); seen.add(q, search_cycle);
           dfs(search_cycle);
           if (!search_cycle && q.isAccepting()) {
              seed = q; dfs(true);
     }  }  }
     stack.pop();
   }
   // initialization
   Stack stack = new Stack(); Set seen = new Set(); seed = null;
   foreach (Pair p in makeInitialPairs(ks, aut)) {
      stack.push(p); seen.add(p, false);
      dfs(false);
}  }
```

Figure 3.7. *On-the-fly* PTL *model checking algorithm*

will report an acceptance cycle if the product automaton contains one, although it may not find all existing cycles (even if the search were to continue after the first reported cycle).

The algorithm avoids the construction of the product automaton: the stack only contains the prefix of the currently explored path, and the set `seen` stores the pairs that have already been visited during the search. A possible optimization would store only a subset of these pairs by trading some redundant search for less memory consumption. Just as for the algorithm for invariant checking of Figure 3.3, when the procedure `dfs` finds an acceptance cycle, it is represented in the search stack and can be displayed as a counter-example.

The complexity of the algorithm in Figure 3.7 is still linear in the size of the product of $\mathcal{K}$ and $\mathcal{B}_{\neg\varphi}$; by Theorem 3.2 it is therefore linear in the size of $\mathcal{K}$ and exponential in the size of φ. In practice, the linear factor is often more problematic because correctness properties tend to be short formulae.

For systems where the size of the product automaton exceeds a few million pairs, the set `seen` no longer fits into the main memory. In order to reduce memory consumption, it is possible to store signatures of states as computed by a hash function, rather than the states themselves [HOL 98]. When doing so, different pairs may generate the same signature, leading to a premature termination of the algorithm. This risk can be reduced by repeating the verification, using different hash functions.

Different algorithms for the model checking of PTL formulae, still based on ω-automata, have been proposed by Couvreur [COU 99] and by Geldenhuys and Valmari [GEL 04]; Schwoon and Esparza [SCH 05] discuss the pros and cons of these algorithms in more detail. The on-the-fly algorithm based on alternating automata [HAM 05] is intended for the verification of large formulae.

3.5.2. *Global* CTL *model checking*

Global model checking algorithms recurse on the syntax of the formula to be verified. We consider here an algorithm for the branching-time logic CTL that calculates the satisfaction sets $[\![\psi]\!]_{\mathcal{K}}$ (see Definition 3.4) for the subformulae of the formula φ of interest. Recall that in CTL, φ is system valid if $I \subseteq [\![\varphi]\!]_{\mathcal{K}}$, where I is the set of initial states of $\mathcal{K}$. The first clauses for the recursive definition of $[\![\varphi]\!]_{\mathcal{K}}$ are quite obvious:

$$\begin{array}{rcl} [\![v]\!]_{\mathcal{K}} & = & \{q \in Q : v \in \lambda(q)\} \quad \text{for } v \in \mathcal{V} \\ [\![\neg\psi]\!]_{\mathcal{K}} & = & Q \setminus [\![\psi]\!]_{\mathcal{K}} \\ [\![\psi_1 \vee \psi_2]\!]_{\mathcal{K}} & = & [\![\psi_1]\!]_{\mathcal{K}} \cup [\![\psi_2]\!]_{\mathcal{K}} \\ [\![\psi_1 \wedge \psi_2]\!]_{\mathcal{K}} & = & [\![\psi_1]\!]_{\mathcal{K}} \cap [\![\psi_2]\!]_{\mathcal{K}} \\ [\![\mathrm{EX}\,\psi]\!]_{\mathcal{K}} & = & \delta^{-1}([\![\psi]\!]_{\mathcal{K}}) \\ & = & \{q \in Q : \text{there are } e, q' \text{ such that } (q, e, q') \in \delta \text{ and } q' \in [\![\psi]\!]_{\mathcal{K}}\} \end{array}$$

It remains to find appropriate definitions for the connectives EG and EU, and we will make use of their recursive characterizations:

$$\begin{array}{rcl} \mathrm{EG}\,\psi & \Leftrightarrow & \psi \wedge \mathrm{EX}\,\mathrm{EG}\,\psi \\ \psi_1\ \mathrm{EU}\ \psi_2 & \Leftrightarrow & \psi_2 \vee (\psi_1 \wedge \mathrm{EX}(\psi_1\ \mathrm{EU}\ \psi_2)). \end{array}$$

Using the above definitions of $[\![\_]\!]_{\mathcal{K}}$, these laws can be rewritten as

$$\begin{array}{rcl} [\![\mathrm{EG}\,\psi]\!]_{\mathcal{K}} & = & [\![\psi]\!]_{\mathcal{K}} \cap \delta^{-1}([\![\mathrm{EG}\,\psi]\!]_{\mathcal{K}}) \\ [\![\psi_1\ \mathrm{EU}\ \psi_2]\!]_{\mathcal{K}} & = & [\![\psi_2]\!]_{\mathcal{K}} \cup ([\![\psi_1]\!]_{\mathcal{K}} \cap \delta^{-1}([\![\psi_1\ \mathrm{EU}\ \psi_2]\!]_{\mathcal{K}})), \end{array}$$

yielding implicit characterizations of these sets. Indeed, $[\![\mathrm{EG}\,\psi]\!]_{\mathcal{K}}$ and $[\![\psi_1\ \mathrm{EU}\ \psi_2]\!]_{\mathcal{K}}$ are respectively the greatest and the least fixed points of the following functions:

$$f_{\mathrm{EG}\,\psi} : \begin{cases} 2^Q \to 2^Q \\ S \mapsto [\![\psi]\!]_{\mathcal{K}} \cap \delta^{-1}(S) \end{cases} \qquad f_{\psi_1 \mathrm{EU} \psi_2} : \begin{cases} 2^Q \to 2^Q \\ S \mapsto [\![\psi_2]\!]_{\mathcal{K}} \cup ([\![\psi_1]\!]_{\mathcal{K}} \cap \delta^{-1}(S)). \end{cases}$$

Both functions $f_{\mathrm{EG}\,\psi}$ and $f_{\psi_1 \mathrm{EU} \psi_2}$ are monotonic and (by Tarski's fixed point theorem) therefore have least and greatest fixed points. Moreover, since Q and therefore 2^Q are finite sets, these functions are even continuous, and the fixed points can be effectively computed as the limits of the sequences

$$\begin{array}{l} Q \supseteq f_{\mathrm{EG}\,\psi}(Q) \supseteq f_{\mathrm{EG}\,\psi}(f_{\mathrm{EG}\,\psi}(Q)) \supseteq \ldots \\ \emptyset \subseteq f_{\psi_1 \mathrm{EU} \psi_2}(\emptyset) \subseteq f_{\psi_1 \mathrm{EU} \psi_2}(f_{\psi_1 \mathrm{EU} \psi_2}(\emptyset)) \subseteq \ldots \end{array}$$

Because at least one state is removed (respectively added) at each step as long as the fixed point has not been reached, the computation terminates after at most $|Q|$ iterations. The complexity of computing these functions at each iteration is linear in $|\delta|$, hence at worst quadratic in $|Q|$. The model checking algorithm requires computing the sets $[\![\psi]\!]_{\mathcal{K}}$ for all subformulae ψ of φ, whose number is linear in the length of φ. Thus, the overall complexity of this "naive"CTL model checking algorithm is linear in $|\varphi|$ and cubic in $|Q|$.

Clarke, Emerson, and Sistla [CLA 86] have defined a more efficient algorithm whose complexity is linear in the product of the size of the formula and the size of $|\mathcal{K}|$ (and therefore at worst quadratic in $|Q|$). For the computation of $[\![\psi_1 \text{ EU } \psi_2]\!]_{\mathcal{K}}$, the idea is to perform a backward search starting from the states in $[\![\psi_2]\!]_{\mathcal{K}}$. For $[\![\mathrm{EG}\,\psi]\!]_{\mathcal{K}}$, the graph of $\mathcal{K}$ is first restricted to those states satisfying ψ, and the algorithm enumerates the strongly connected components (SCCs) of this subgraph. The set $[\![\mathrm{EG}\,\psi]\!]_{\mathcal{K}}$ consists of all states from where such an SCC is reachable. These states are easily enumerated using a breadth-first search algorithm.

We have observed in section 3.4.2 that fairness conditions cannot be expressed in CTL. Verification of CTL properties under fairness hypotheses therefore requires adapting the model checking algorithm. (For PTL this is not necessary because one can verify formulae $\mathit{fair} \Rightarrow \varphi$ where fair is a PTL encoding of the fairness conditions.) McMillan [MCM 93] proposed to characterize "fair" states of a Kripke structure using CTL formulae. A run σ of $\mathcal{K}$ is fair if it contains infinitely many states satisfying these formulae. For example, weak fairness with respect to a certain action can be expressed in this scheme by identifying the states where the action is either disabled or has just been taken. We can then define variants $\mathrm{EG_f}$ and $\mathrm{EU_f}$ of the operators EG and EU whose semantics differ by asserting the existence of a *fair* path (with respect to all fairness constraints) satisfying the temporal conditions, rather than the existence of an arbitrary path. It is easy to see that the formula $\psi_1 \ \mathrm{EU_f} \ \psi_2$ is equivalent to the formula $\psi_1 \text{ EU } (\psi_2 \wedge \mathrm{EG_f}\,\mathrm{true})$, and it is therefore enough to find a model checking algorithm for formulae of the form $\mathrm{EG_f}\,\psi$. For the algorithm of Clarke, Emerson and Sistla, it is enough to consider the SCCs of the subgraph of $\mathcal{K}$ containing the states satisfying ψ that contain, for each fairness constraint, some state satisfying that constraint. This information can be obtained while the SCCs are searched, and the complexity of the algorithm remains linear in the size of the model and the formula.

Global model checking algorithms can also be defined for other branching-time logics, and in particular for the propositional μ-calculus. In that case, the complexity is of the order $|\varphi| \cdot |\mathcal{K}|^{qd(\varphi)}$, where $qd(\varphi)$ denotes the maximum nesting depth of fixed point operators in φ. Emerson and Lei [EME 86b] have observed that the computation of nested fixed points of the same type (least or greatest fixed point) can be carried out simultaneously. In this way, they obtained an algorithm of complexity $|\varphi| \cdot |\mathcal{K}|^{ad(\varphi)}$ where $ad(\varphi)$ denotes the maximum alternation depth of fixed

point operators. In particular, the alternation-free fragment of the μ-calculus has an algorithm of the same complexity as the logic CTL but offers a strictly greater expressiveness [CLE 93, EME 93].

3.5.3. *Symbolic model checking algorithms*

Model checking algorithms manipulate sets of states. Efficient data structures to represent such sets are therefore critical for obtaining practically useful implementations of these algorithms. Explicit enumeration of states is limited to a few million states. A popular alternative is to represent sets symbolically, since these representations are more sensitive to the structure of the set than its size, and they can help verify much larger systems. In particular, the use of binary decision diagrams (BDDs, more precisely reduced ordered BDDs) has been a technological breakthrough for the implementation of model checking algorithms. Main advantages of BDDs are, first, that they provide canonical representations of sets, and therefore set equality can be decided by simple pointer comparison. Second, Boolean operations can be performed in polynomial time.

The basic idea is to represent a set S by its characteristic predicate χ_S, for which $x \in S$ if and only if $\chi_S(x)$ is true. Without loss of generality, we assume that states of finite transition systems are represented by a finite set $\{b_1, \ldots, b_n\}$ of Boolean variables. The transition relation can then be represented as a predicate over the set of variables $\{b_1, \ldots, b_n, b'_1, \ldots, b'_n\}$ such that variables b_i represent the source state of the transition, and variables b'_i the target state.

A BDD can be understood as an efficient representation of a binary decision tree. For example, Figure 3.8(a) shows a decision tree that determines whether the addition of two two-bit numbers b_1b_0 and c_1c_0 will produce a carry bit. This decision tree represents the set

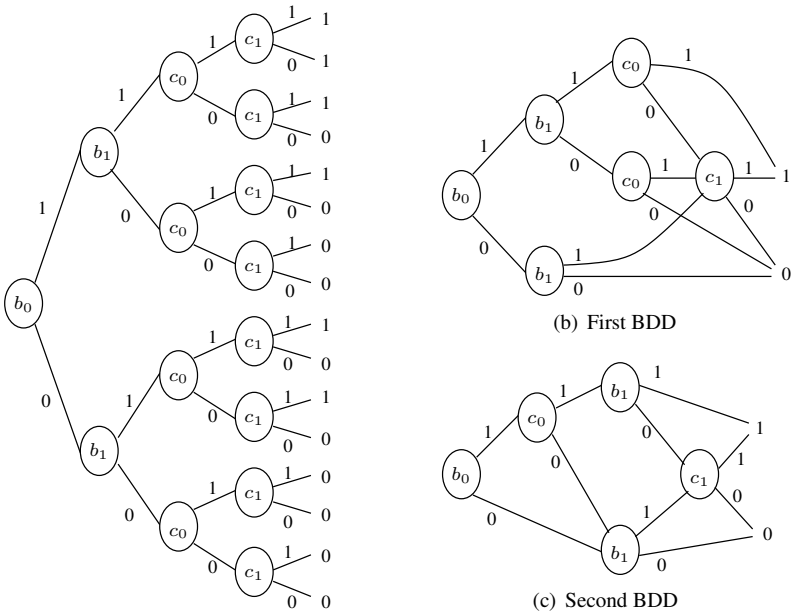
$$\{(11, 11), (11, 01), (11, 10), (01, 11), (10, 11), (10, 10)\}$$

of pairs of two-digit numbers; these are the labels of all paths leading to the result 1. (Note that the variables appear in the same order b_0, b_1, c_0, c_1 along any branch in the tree.) The same set is represented by the propositional formula

$$(b_1 \wedge c_1) \vee (b_0 \wedge c_0 \wedge (b_1 \vee c_1))$$

if we identify the values 0 and 1 with the truth values “false” and “true”.

The tree in Figure 3.8(a) contains many redundancies. For example, the values of c_0 and c_1 are of no importance if b_0 and b_1 are both 0. A more compact representation eliminates these redundancies. First, isomorphic subtrees can be identified, and this results in a dag representation. Second, nodes whose children along both branches are



(a) Decision tree

(b) First BDD

(c) Second BDD

Figure 3.8. *Decision tree and two BDDs*

identical can be eliminated. If we carry out these steps for our example, we obtain the BDD that appears in Figure 3.8(b). This representation is more compact because nodes are shared whenever possible. An implementation of a BDD package ensures maximum sharing by remembering each allocated BDD so that it can be reused instead of reallocated when it is needed a second time.

Every Boolean function is represented by a unique BDD with respect to a fixed variable order. However, the choice of this order can significantly influence the size of the BDD representing a given function. For example, the BDD shown in Figure 3.8(c) again represents our carry function, but with respect to the variable order b_0, c_0, b_1, c_1. When computing the carry for a growing number of bits, this second BDD grows linearly whereas the BDD for the original ordering grows exponentially. Choosing an optimal variable order for a given Boolean function is an NP-complete problem [BRY 92]. Standard implementations of BDD libraries use heuristics to determine the variable order [BER 95, FEL 93], but manual tuning is often necessary. In general, it is advisable that variables that are strongly inter-dependent are close to each other in the variable order [END 93, FUJ 93]. Unfortunately, there exist functions for which the size of BDD representations grow exponentially independently of the chosen variable order. Examples are multiplication of bit vectors or representations of FIFO queues.

Given two BDDs f and g (for a fixed variable order), their Boolean combinations can be computed recursively:

– if f or g are terminal BDD nodes (0 or 1) then the result is easily determined by the operation;

– otherwise, let b be the smallest variable according to the variable order at the roots of BDDs f and g, and let h_0 and h_1 be, recursively, the results of the operation applied to the sub-BDDs of f and g corresponding to b being 0 and 1. If $h_0 = h_1$, then the result does not depend on the value of b, so we return h_0, otherwise the result is the BDD with root b and successors h_0 and h_1.

The number of sub-problems to be computed is bounded by the number of pairs of nodes in the BDDs f and g. Assuming that the results of recursive computations are stored in a hash table with (nearly) constant access time, the cost of the operation is proportional to the product of the sizes of f and g.

Another useful BDD operation is *projection*, that is, existential quantification over a Boolean variable. Observing that

$$(\exists b : f) \;=\; f|_{b=0} \vee f|_{b=1},$$

the BDD representing this formula can be computed by disjunction and substitution of constants for variables. Quantification over several variables can be done simultaneously. Although the worst-case complexity of projection is exponential, this is rarely observed in practice.

A symbolic CTL model checking algorithm is easily obtained by implementing the naive recursive computation of the satisfaction sets $[\![\psi]\!]_{\mathcal{K}}$ mentioned in section 3.5.2 based on a BDD representation. For this purpose, we assume that the set of initial states of $\mathcal{K}$ and the transition relation are also represented in BDD form. The computation of pre-images $\delta^{-1}(s)$ translates into evaluating the expression

$$\exists \vec{b'} : \delta \wedge S'$$

where $\vec{b'}$ is the set of primed variables representing the states of $\mathcal{K}$, and where S' is a copy of the BDD for S where each variable b_i has been replaced by its primed version b'_i. The BDD machinery provides the necessary primitives for carrying out all required computations. In particular, termination of the fixed point computations is easily detected using pointer comparison.

It is interesting to compare the complexity of the BDD-based algorithm with that of the explicit-state algorithm. The representation of the sets manipulated by the algorithm can be exponentially more succinct. However, the number of iterations remains bounded by the size of the model and can therefore be exponential in the size of the BDD. Moreover, computing pre-images is based on quantification and therefore has

exponential worst-case complexity. Fortunately, in practice the necessary fixed point iterations tend to converge quickly, particularly for the verification of electronic circuits with short data paths. Indeed, symbolic model checking has permitted the verification of systems with 10^{100} states or more [CLA 93b]. The crucial point is usually to find a variable order that allows the model checker to represent the transition relation.

The symbolic algorithm described here can be used for the verification of PTL formulae, using a symbolic representation of the Büchi automaton, and therefore of the product automaton. This technique is described in detail by Clarke *et al.* [CLA 97] and by Schneider [SCH 99].

BOUNDED MODEL CHECKING.– Although BDD representations have long dominated as the data structure used for representing Boolean functions in the model checking area, other representations can be useful. In particular, SAT algorithms for deciding the satisfiability of formulae of (non-temporal) propositional logic have made significant progress since the late 1990s [MIT 05]. These algorithms usually assume their input to be given as a list of clauses, and are not based on canonical representations of Boolean functions, virtually ruling out the calculation of fixed points. Instead, bounded model checking algorithms attempt to find an execution of finite length that violates the property of interest. The maximum size of a potential counter-example can (at least in theory) be determined from the size of the model and the formula to be verified. For example, an invariant holds for a Kripke structure if it is true for all prefixes of runs whose length is at most that of the longest loop in the graph of $\mathcal{K}$ (called the diameter of $\mathcal{K}$).

Given a bound $k \in \mathbb{N}$, the existence of a finite run of size $\leqslant k$ and leading to a state that does not verify an invariant is easily coded as a satisfiability problem in propositional logic by providing k copies of the state variables and relating subsequent copies by a (non-temporal) formula encoding transition relation. If finite loops back into the execution prefix are also considered, this idea can be generalized for full temporal logic [BIE 03]. In practice, bounded model checking is very useful to quickly find relatively small counter-examples. It is less appropriate for full-fledged verification because the worst-case bound on the maximum path lengths that need to be considered is again exponential in the size of the formula.

3.6. Some research topics

At the end of this introductory discussion of model checking concepts and techniques, we list references to various current topics of research, without trying to be exhaustive concerning either the list of topics or the references. The general objective is to make model checking and verification techniques applicable to more classes of systems and to larger systems.

REDUCTION TECHNIQUES.– In order to ensure that a system satisfies a given property, it is often enough to explore a representative subset of system runs instead of all runs. Because model checking is usually applied to reactive systems with several parallel components, reduction techniques geared towards concurrent systems are particularly useful. Two actions e_1 and e_2 are called *independent* if at any state where e_1 and e_2 are both possible, the execution of e_1 leaves e_2 executable and vice versa, and if the combined effects of executing the two action sequences $e_1; e_2$ and $e_2; e_1$ are the same. For example, the actions that represent two different processes sending messages along two different channels are independent. If the property of interest is not sensitive to the order of execution of the two actions (for example, if the property does not mention the communication channels), the execution of action e_2 can be delayed. Depending on the degree of independence and concurrency of a system, the systematic application of this idea can lead to substantial reductions in the number of transitions to explore. Nevertheless, one should take care that delayed actions will be considered eventually, and in particular before closing a loop in the reduced state space. Different authors have proposed reduction algorithms along these lines [ESP 94, GOD 94, HOL 94, PEL 96, VAL 90]. The key to making this idea work in practice is to find a good compromise between the cost of determining that actions are independent, say, by static analysis of the model, and the savings obtained during verification.

A different form of reduction makes use of *symmetries* in the state spaces of systems, which are also a source of redundancy during verification. For example, communication protocols usually do not inspect the data values that are transmitted over channels, and we can identify two states if they only differ in the values that channels contain. Similarly, systems often consist of several copies of identical processes whose precise identity is not relevant for system execution. These forms of symmetries induce an equivalence relation on the state space of the system, and it is enough to verify the quotient of the original system state space with respect to this equivalence relation, as long as the property to be verified is also symmetric [CLA 93a, IP 93, STA 91].

COMPOSITIONAL VERIFICATION.– In order to alleviate the effects of combinatorial explosion, it can be useful to preserve the process structure of a system instead of representing it as a "flat" transition system. In particular, specifications are written for each system component whose combination entails the overall correctness. In order to establish overall correctness we have to

– ensure that each component satisfies its specification, and

– verify that the correctness of each component implies system correctness.

In general, an individual component cannot be expected to operate correctly in an arbitrary environment. Hypotheses about the functioning of the component's environment are therefore explicitly identified in the component specification, and must

be verified for the considered system. This becomes non-trivial in the case of mutual assumptions between components. Important approaches to compositional verification are presented in [ROE 98, ROE 01]. In the context of algorithmic verification, compositional verification is often referred to as the *module checking* problem, see [GRU 94, KUP 97a, MCM 97], among other works. An interesting extension of this problem is to synthesize components from temporal logic specifications [KUP 00, PIT 06, PNU 89] rather than verify the correctness of models a posteriori.

ABSTRACTION AND SOFTWARE MODEL CHECKING.– The models used for verification are generally abstract representations of real systems. Instead of manually providing these abstractions, tools can be built that try to compute sound abstractions from a more detailed description of a system. The construction of such abstractions has been widely studied, for some foundational articles see [CLA 94, DAM 94, LOI 95]. An intuitive and elegant presentation of abstractions consists of defining the state space of the abstract model by a set of predicates over the concrete state space. This representation, known as *predicate abstraction*, was originally considered for combinations of deductive and algorithmic verification tools, see for example [BJO 00, CAN 01, GRA 97, KES 00]. More recently, several tools combine this format with techniques of abstract interpretation, with the objective of model checking program code. The main challenge here is to obtain a meaningful finite-state abstraction of (usually infinite-state) software. A useful strategy is to start from the control-flow graph and first compute an over-approximation of the state space by abstracting the data values manipulated by the program. This abstraction serves as input for a conventional finite-state model checker, which will probably return a counter-example. Analyzing this abstract run over the concrete program, the verification tool determines if the counter-example represents a run of the concrete system (in which case it is reported to the user) or whether it is due to an imprecise abstraction. In the latter case, it will construct a more detailed abstract model, and the procedure is repeated, with the goal of eventually reaching a definitive verdict. This technique is known as *counter-example guided abstraction refinement* (CEGAR); see for example [BAL 02, HEN 04, SUW 05]. It is characterized by combining different approaches to verification, including abstract interpretation, automatic theorem proving and model checking.

INFINITE-STATE SYSTEMS.– In this chapter we have discussed model checking algorithms for finite-state systems. The extension of these techniques to infinite-state systems (directly or in combination with abstraction techniques) is an important research topic that has been studied at many different angles. Of course, real-time and hybrid systems are examples of infinite-state systems, and they are the main topic of this book. Probabilistic systems have also become of more and more interest, and again we refer to Chapters 8 and 9. More generally, verification problems can be tractable for system classes that generate sufficiently regular state spaces, such as certain properties of pushdown systems. See [ESP 01] for a classification of infinite-state systems and an annotated bibliography about this field.

3.7. Bibliography

[ALU 97] ALUR R., HENZINGER T. A., KUPFERMAN O., "Alternating-time temporal logic", *38th IEEE Symposium on Foundations of Computer Science*, IEEE Press, p. 100–109, 1997.

[BAL 02] BALL T., RAJAMANI S. K., "The SLAM project: debugging system software via static analysis", *Principles of Programming Languages (POPL 2002)*, p. 1–3, 2002.

[BER 95] BERN J., MEINEL C., SLOBODOVÁ A., "Global rebuilding of BDDs – avoiding the memory requirement maxima", WOLPER P., Ed., *7th Intl. Conf. Computer Aided Verification (CAV'95)*, vol. 939 of *Lect. Notes in Comp. Sci.*, Springer, p. 4–15, 1995.

[BÉR 01] BÉRARD B., BIDOIT M., FINKEL A., LAROUSSINIE F., PETIT A., PETRUCCI L., SCHNOEBELEN PH., *Systems and Software Verification. Model-Checking Techniques and Tools*, Springer, 2001,

[BIE 03] BIERE A., CIMATTI A., CLARKE E., STRICHMAN O., ZHU Y., "Bounded model checking", *Highly Dependable Software*, vol. 58 of *Advances in Computers*, Academic Press, 2003.

[BJO 00] BJORNER N., BROWNE A., COLON M., FINKBEINER B., MANNA Z., SIPMA H., URIBE T., "Verifying temporal properties of reactive systems: a STeP tutorial", *Formal Methods in System Design*, vol. 16, p. 227–270, 2000.

[BRY 92] BRYANT R. E., "Symbolic Boolean manipulations with ordered binary decision diagrams", *ACM Computing Surveys*, vol. 24, num. 3, p. 293–317, 1992.

[BÜC 62] BÜCHI J. R., "On a decision method in restricted second-order arithmetics", *Intl. Cong. Logic, Method and Philosophy of Science*, Stanford Univ. Press, p. 1–12, 1962.

[CAN 01] CANSELL D., MÉRY D., MERZ S., "Diagram refinements for the design of reactive systems", *Universal Computer Science*, vol. 7, num. 2, p. 159–174, 2001.

[CLA 81] CLARKE E. M., EMERSON E. A., "Synthesis of synchronization skeletons for branching time temporal logic", *Workshop on Logic of Programs*, vol. 131 of *Lect. Notes in Comp. Sci.*, Yorktown Heights, N.Y., Springer, 1981.

[CLA 86] CLARKE E., EMERSON E., SISTLA A., "Automatic verification of finite-state concurrent systems using temporal logic specifications", *ACM Trans. Prog. Lang. and Systems*, vol. 8, num. 2, p. 244–263, 1986.

[CLA 93a] CLARKE E. M., ENDERS R., FILKORN T., JHA S., "Exploiting symmetry in temporal logic model checking", *Formal Methods in System Design*, vol. 9, p. 77–104, 1993.

[CLA 93b] CLARKE E., GRUMBERG O., HIRAISHI H., JHA S., LONG D., MCMILLAN K., NESS L., "Verification of the Futurebus+ cache coherence protocol", AGNEW D., CLAESEN L., CAMPOSANO R., Eds., *IFIP Conf. Computer Hardware Description Lang. and Applications*, Ottawa, Canada, Elsevier, p. 5–20, 1993.

[CLA 94] CLARKE E. M., GRUMBERG O., LONG D. E., "Model checking and abstraction", *ACM Trans. Prog. Lang. and Systems*, vol. 16, num. 5, p. 1512–1542, 1994.

[CLA 97] CLARKE E. M., GRUMBERG O., HAMAGUCHI K., "Another look at LTL model checking", *Formal Methods in System Design*, vol. 10, p. 47–71, 1997.

[CLA 99] CLARKE E. M., GRUMBERG O., PELED D., *Model Checking*, MIT Press, Cambridge, Mass., 1999.

[CLA 00] CLARKE E. M., SCHLINGLOFF H., "Model Checking", ROBINSON A., VORONKOV A., Eds., *Handbook of Automated Deduction*, p. 1367–1522, Elsevier, 2000.

[CLE 93] CLEAVELAND R., STEFFEN B., "A linear-time model-checking algorithm for the alternation-free modal μ-calculus", *Formal Methods in System Design*, vol. 2, p. 121–147, 1993.

[COU 92] COURCOUBETIS C., VARDI M., WOLPER P., YANNAKAKIS M., "Memory-efficient algorithms for the verification of temporal properties", *Formal Methods in System Design*, vol. 1, p. 275–288, 1992.

[COU 99] COUVREUR J.-M., "On-the-fly verification of linear temporal logic", WING J., WOODCOCK J., DAVIES J., Eds., *Formal Methods (FM'99)*, vol. 1708 of *Lect. Notes in Comp. Sci.*, Toulouse, France, Springer, p. 253–271, 1999.

[DAM 94] DAMS D., GRUMBERG O., GERTH R., "Abstract interpretation of reactive systems: abstractions preserving $\forall$CTL*, $\exists$CTL* and CTL*", OLDEROG E.-R., Ed., *Prog. Concepts, Methods, and Calculi*, Amsterdam, Elsevier, p. 561–581, 1994.

[DAN 99] DANIELE M., GIUNCHIGLIA F., VARDI M., "Improved automata generation for linear temporal logic", *Computer Aided Verification (CAV'99)*, vol. 1633 of *Lect. Notes in Comp. Sci.*, Trento, Italy, Springer, p. 249–260, 1999.

[DIL 92] DILL D. L., DREXLER A. J., HU A. J., YANG C. H., "Protocol verification as a hardware design aid", *Intl. Conf. Computer Design: VLSI in Computers and Processors*, IEEE Comp. Soc., p. 522–525, 1992.

[EME 86a] EMERSON E. A., HALPERN J. Y., "'Sometimes' and 'not never' revisited: on branching time vs. linear time", *Journal of the ACM*, vol. 33, p. 151–178, 1986.

[EME 86b] EMERSON E. A., LEI C. L., "Efficient model checking in fragments of the propositional μ-calculus", *1st Symp. Logic in Comp. Sci. (LICS'86)*, Boston, Mass., IEEE Press, 1986.

[EME 90] EMERSON E. A., "Temporal and modal logic", VAN LEEUWEN J., Ed., *Handbook of Theoretical Computer Science*, vol. B, p. 997–1071, Elsevier, 1990.

[EME 93] EMERSON E. A., JUTLA C. S., SISTLA A. P., "On model checking for fragments of μ-calculus", COURCOUBETIS C., Ed., *Computer-Aided Verification (CAV'93)*, vol. 697 of *Lect. Notes in Comp. Sci.*, Springer, 1993.

[END 93] ENDERS R., FILKORN T., TAUBNER D., "Generating BDDs for symbolic model checking", *Distributed Computing*, vol. 6, p. 155–164, 1993.

[ESP 94] ESPARZA J., "Model checking using net unfoldings", *Science of Computer Programming*, vol. 23, p. 151–195, 1994.

[ESP 01] ESPARZA J., "Verification of systems with an infinite state space: an annotated bibliography", CASSEZ F., JARD C., ROZOY B., RYAN M. D., Eds., *Modeling and Verification of Parallel Processes*, vol. 2067 of *Lect. Notes in Comp. Sci.*, p. 183–186, Springer, 2001.

[FEL 93] FELT E., YORK G., BRAYTON R., VINCENTELLI A. S., "Dynamic variable reordering for BDD minimization", *Eur. Design Automation Conf.*, p. 130–135, 1993.

[FUJ 93] FUJI H., OOMOTO G., HORI C., "Interleaving based variable ordering methods for binary decision diagrams", *Intl. Conf. Computer Aided Design*, IEEE Press, 1993.

[GAB 94] GABBAY D., HODKINSON I., REYNOLDS M., *Temporal Logic: Mathematical Foundations and Computational Aspects*, vol. 1, Clarendon Press, Oxford, UK, 1994.

[GAS 01] GASTIN P., ODDOUX D., "Fast LTL to Büchi automata translation", BERRY G., COMON H., FINKEL A., Eds., *13th Intl. Conf. Computer Aided Verification (CAV'01)*, vol. 2102 of *Lect. Notes in Comp. Sci.*, Paris, France, Springer, p. 53–65, 2001.

[GEL 04] GELDENHUYS J., VALMARI A., "Tarjan's algorithm makes LTL verification more efficient", JENSEN K., PODELSKI A., Eds., *10th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, vol. 2988 of *Lect. Notes in Comp. Sci.*, Barcelona, Spain, Springer, p. 205–219, 2004.

[GER 95] GERTH R., PELED D., VARDI M., WOLPER P., "Simple on-the-fly automatic verification of linear temporal logic", *Protocol Specification, Testing, and Verification*, Warsaw, Poland, Chapman & Hall, p. 3–18, 1995.

[GOD 94] GODEFROID P., WOLPER P., "A partial approach to model checking", *Information and Computation*, vol. 110, num. 2, p. 305–326, 1994.

[GRA 97] GRAF S., SAIDI H., "Construction of abstract state graphs with PVS", GRUMBERG O., Ed., *9th Intl. Conf. Computer Aided Verification (CAV'97)*, vol. 1254 of *Lect. Notes in Comp. Sci.*, Springer Verlag, p. 72–83, 1997.

[GRU 94] GRUMBERG O., LONG D. E., "Model checking and modular verification", *ACM Trans. Prog. Lang. and Systems*, vol. 16, num. 3, p. 843–871, 1994.

[HAM 05] HAMMER M., KNAPP A., MERZ S., "Truly on-the-fly LTL model checking", HALBWACHS N., ZUCK L., Eds., *11th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, vol. 3440 of *Lect. Notes in Comp. Sci.*, Edinburgh, UK, Springer, p. 191–205, 2005.

[HEN 04] HENZINGER T. A., JHALA R., MAJUMDAR R., MCMILLAN K., "Abstractions from proofs", *31st Symp. Princ. of Prog. Lang. (POPL'04)*, ACM Press, p. 232–244, 2004.

[HOL 94] HOLZMANN G., PELED D., "An improvement in formal verification", *IFIP Conf. Formal Description Techniques*, Bern, Switzerland, Chapman & Hall, p. 197–214, 1994.

[HOL 98] HOLZMANN G., "An analysis of bitstate hashing", *Formal Methods in System Design*, vol. 13, num. 3, p. 289–307, 1998.

[HOL 03] HOLZMANN G., *The SPIN Model Checker*, Addison-Wesley, 2003.

[HUT 04] HUTH M., RYAN M. D., *Logic in Computer Science*, Cambridge University Press, Cambridge, U.K., 2nd edition, 2004.

[IP 93] IP C. N., DILL D., "Better verification through symmetry", *11th Intl. Symp. Comp. Hardware Description Languages and their Applications*, North Holland, p. 87–100, 1993.

[KAM 68] KAMP H. W., "Tense logic and the theory of linear order", PhD thesis, Univ. of California at Los Angeles, 1968.

[KES 00] KESTEN Y., PNUELI A., "Verification by augmented finitary abstraction", *Information and Computation*, vol. 163, num. 1, p. 203–243, 2000.

[KOZ 83] KOZEN D., "Results on the propositional μ-calculus", *Theor. Comp. Sci.*, vol. 27, p. 333–354, 1983.

[KUP 94] KUPFERMAN O., VARDI M., WOLPER P., "An automata-theoretic approach to branching-time model checking", *6th Intl. Conf. Computer-Aided Verification (CAV'94)*, *Lect. Notes in Comp. Sci.*, Springer, 1994.

[KUP 97a] KUPFERMAN O., VARDI M., "Module checking revisited", *9th Intl. Conf. Computer Aided Verification (CAV'97)*, vol. 1254 of *Lect. Notes in Comp. Sci.*, Springer, p. 36–47, 1997.

[KUP 97b] KUPFERMAN O., VARDI M. Y., "Weak alternating automata are not so weak", *5th Israeli Symp. Theory of Computing and Systems*, IEEE Press, p. 147–158, 1997.

[KUP 00] KUPFERMAN O., MADHUSUDAN P., THIAGARAJAN P., VARDI M., "Open systems in reactive environments: control and synthesis", PALAMIDESSI C., Ed., *Proc. 11th Int. Conf. on Concurrency Theory*, vol. 1877 of *Lecture Notes in Computer Science*, Springer Verlag, p. 92–107, 2000.

[KUP 05] KUPFERMAN O., VARDI M., "Complementation constructions for nondeterministic automata on infinite words", HALBWACHS N., ZUCK L., Eds., *11th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, vol. 3440 of *Lect. Notes in Comp. Sci.*, Edinburgh, UK, Springer, p. 206–221, 2005.

[LAM 80] LAMPORT L., "'Sometime' is sometimes 'not never"', *7th Symp. Princ. of Prog. Lang. (POPL'80)*, p. 174–185, 1980.

[LAR 02] LAROUSSINIE F., MARKEY N., SCHNOEBELEN PH., "Temporal logic with forgettable past", *17th IEEE Symp. Logic in Computer Science (LICS'02)*, Copenhagen, Denmark, IEEE Press, p. 383–392, 2002.

[LOI 95] LOISEAUX C., GRAF S., SIFAKIS J., BOUAJJANI A., BENSALEM S., "Property preserving abstractions for the verification of concurrent systems", *Formal Methods in System Design*, vol. 6, p. 11–44, 1995.

[MAN 90] MANNA Z., PNUELI A., "A hierarchy of temporal properties", *9th ACM Symp. Princ. Distributed Computing*, ACM, p. 377–408, 1990.

[MCM 93] MCMILLAN K., *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[MCM 97] MCMILLAN K. L., "A compositional rule for hardware design refinement", GRUMBERG O., Ed., *9th Intl. Conf. Computer Aided Verification (CAV'97)*, vol. 1254 of *Lect. Notes in Comp. Sci.*, Haifa, Israel, Springer, p. 24–35, 1997.

[MIT 05] MITCHELL D. G., "A SAT solver primer", *EATCS Bulletin*, vol. 85, p. 112–133, 2005.

[MUL 63] MULLER D. E., "Infinite sequences and finite machines", *Switching Circuit Theory and Logical Design*, New York, IEEE Press, p. 3–16, 1963.

[MUL 88] MULLER D., SAOUDI A., SCHUPP P., "Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time",

3rd IEEE Symp. Logic in Computer Science (LICS'88), IEEE Press, p. 422–427, 1988.

[PEL 96] PELED D., "Combining partial order reductions with on-the-fly model-checking", *Formal Methods in System Design*, vol. 8, num. 1, p. 39–64, 1996.

[PIT 06] PITERMAN N., PNUELI A., SA'AR Y., "Synthesis of reactive designs", EMERSON E. A., NAMJOSHI K. S., Eds., *Verification, Model Checking, and Abstract Interpretation (VMCAI 2006)*, vol. 3855 of *Lect. Notes in Comp. Sci.*, Charleston, S.C., Springer, p. 364–380, 2006.

[PNU 89] PNUELI A., ROSNER R., "On the synthesis of a reactive module.", *Principles of Programming Languages (POPL 1989)*, ACM Press, p. 179–190, 1989.

[PSL 04] PSL CONSORT., "Property Specification Language (version 1.1)", Technical report, Accellera, June 2004.

[QUE 81] QUEILLE J., SIFAKIS J., "Specification and verification of concurrent systems in Cesar", *5th Intl. Symp. Programming*, vol. 137 of *Lect. Notes in Comp. Sci.*, Springer, p. 337–351, 1981.

[RAB 69] RABIN M. O., "Decidability of second-order theories and automata on infinite trees", *Trans. American Math. Soc.*, vol. 141, p. 1–35, 1969.

[ROE 98] DE ROEVER W.-P., LANGMAACK H., PNUELI A., Eds., *Compositionality: the significant difference*, vol. 1536 of *Lect. Notes in Comp. Sci.*, Springer-Verlag, 1998.

[ROE 01] DE ROEVER W.-P., DE BOER F., HANNEMANN U., HOOMAN J., LAKHNECH Y., POEL M., ZWIERS J., *Concurrency Verification: Introduction to Compositional and Non-compositional Methods*, Cambridge University Press, 2001.

[SAF 88] SAFRA S., "On the complexity of ω-automata", *29th IEEE Symp. Found. Comp. Sci. (FOCS'88)*, IEEE Press, p. 319–327, 1988.

[SCH 99] SCHNEIDER K., "Yet another look at LTL model checking", *IFIP Conf. Correct Hardware Design and Verification Methods (CHARME'99)*, *Lect. Notes in Comp. Sci.*, Bad Herrenalb, Germany, Springer, 1999.

[SCH 05] SCHWOON S., ESPARZA J., "A note on on-the-fly verification algorithms", HALBWACHS N., ZUCK L., Eds., *11th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, vol. 3440 of *Lect. Notes in Comp. Sci.*, Edinburgh, UK, Springer, p. 174–190, 2005.

[SIS 87] SISTLA A., VARDI M., WOLPER P., "The complementation problem for Büchi automata with applications to temporal logic", *Theor. Comp. Sci.*, vol. 49, p. 217–237, 1987.

[STA 91] STARKE P. H., "Reachability analysis of Petri nets using symmetries", *Syst. Anal. Model. Simul.*, vol. 8, p. 293–303, 1991.

[STI 01] STIRLING C., *Modal and Temporal Properties of Processes*, Springer, Berlin, 2001.

[SUW 05] SUWIMONTEERABUTH D., SCHWOON S., ESPARZA J., "jMoped: a Java bytecode checker based on Moped", HALBWACHS N., ZUCK L., Eds., *11th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, vol. 3440 of *Lect. Notes in Comp. Sci.*, Edinburgh, UK, Springer, p. 541–545, 2005.

[TAR 72] TARJAN R. E., "Depth first search and linear graph algorithms", *SIAM Journal of Computing*, vol. 1, p. 146–160, 1972.

[THO 97] THOMAS W., "Languages, automata, and logic", ROZENBERG G., SALOMAA A., Eds., *Handbook of Formal Language Theory*, vol. III, p. 389–455, Springer, 1997.

[VAL 90] VALMARI A., "A stubborn attack on state explosion", *2nd Intl. Conf. Computer Aided Verification (CAV'90)*, vol. 531 of *Lect. Notes in Comp.Sci.*, Rutgers, N.J., Springer, p. 156–165, 1990.

[VAR 94] VARDI M., WOLPER P., "Reasoning about infinite computations", *Information and Computation*, vol. 115, num. 1, p. 1–37, 1994.

[VAR 95] VARDI M. Y., "Alternating automata and program verification", *Computer Science Today*, vol. 1000 of *Lect. Notes in Comp. Sci.*, p. 471–485, Springer, 1995.

[VAR 01] VARDI M., "Branching vs. linear time—final showdown", MARGARIA T., YI W., Eds., *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, vol. 2031 of *Lect. Notes in Comp. Sci.*, Genova, Italy, Springer, p. 1–22, 2001.

[WOL 83] WOLPER P., "Temporal logic can be more expressive", *Information and Control*, vol. 56, p. 72–93, 1983.