Model Checking: A Tutorial Overview

Stephan Merz

Institut für Informatik, Universität München
merz@informatik.uni-muenchen.de

Abstract. We survey principles of model checking techniques for the automatic analysis of reactive systems. The use of model checking is exemplified by an analysis of the Needham-Schroeder public key protocol. We then formally define transition systems, temporal logic, ω-automata, and their relationship. Basic model checking algorithms for linear- and branching-time temporal logics are defined, followed by an introduction to symbolic model checking and partial-order reduction techniques. The paper ends with a list of references to some more advanced topics.

1 Introduction

Computerized systems pervade more and more our everyday lives. We rely on digital controllers to supervise critical functions of cars, airplanes, and industrial plants. Digital switching technology has replaced analog components in the telecommunication industry, and security protocols enable e-commerce applications and privacy. Where important investments or even human lives are at risk, quality assurance for the underlying hardware and software components becomes paramount, and this requires formal models that describe the relevant part of the systems at an adequate level of abstraction. The systems we are focussing on are assumed to maintain an ongoing interaction with their environment (e.g., the controlled system or other components of a communication network) and are therefore called *reactive systems* [60,94]. Traditional models that describe computer programs as computing some result from given input values are inadequate for the description of reactive systems. Instead, the behavior of reactive systems is usually modelled by transition systems.

The term model checking designates a collection of techniques for the automatic analysis of reactive systems. Subtle errors in the design of safety-critical systems that often elude conventional simulation and testing techniques can be (and have been) found in this way. Because it has been proven cost-effective and integrates well with conventional design methods, model checking is being adopted as a standard procedure for the quality assurance of reactive systems.

The inputs to a model checker are a (usually finite-state) description of the system to be analysed and a number of properties, often expressed as formulas of temporal logic, that are expected to hold of the system. The model checker either confirms that the properties hold or reports that they are violated. In the latter case, it provides a counter-example: a run that violates the property. Such a run can provide valuable feedback and points to design errors. In practice, this view turns out to be somewhat idealized: quite frequently, available resources

F. Cassez et al. (Eds.): MOVEP 2000, LNCS 2067, pp. 3–38, 2001.
© Springer-Verlag Berlin Heidelberg 2001

only permit to analyse a rather coarse model of the system. A positive verdict from the model checker is then of limited value because bugs may well be hidden by the simplifications that had to be applied to the model. On the other hand, counter-examples may be due to modelling artefacts and no longer correspond to actual system runs. In any case, one should keep in mind that the object of analysis is always an *abstract model* of the system. Standard procedures such as code reviews are necessary to ensure that the abstract model adequately reflects the behavior of the concrete system in order for the properties of interest to be established or falsified. Model checkers can be of some help in this validation task because it is possible to perform "sanity checks", for example to ensure that certain runs are indeed possible or that the model is free of deadlocks.

This paper is intended as a tutorial overview of some of the fundamental principles of model checking, based on a necessarily subjective selection of the large body of model checking literature. We begin with a case study in section 2 where the application of model checking is considered from a user's point of view. Section 3 reviews transition systems, temporal logics, and automata-theoretic techniques that underly some approaches to model checking. Section 4 introduces basic model checking algorithms for linear-time and branching-time logics. Finally, section 5 collects some rather sketchy references to more advanced topics. Much more material can be found in other contributions to this volume and in the textbooks and survey papers [27,28,69,97,124] on the subject. The paper contains many references to the relevant literature, in the hope that this survey can also serve as an annotated bibliography.

2 Analysis of a Cryptographic Protocol

2.1 Description of the Protocol

Let us first consider, by way of example, the analysis of a public-key authentication protocol suggested by Needham and Schroeder [104] using the model checker Spin [65]. Two agents A(lice) and B(ob) try to establish a common secret over an insecure channel in such a way that both are convinced of each other's presence and no intruder can get hold of the secret without breaking the underlying encryption algorithm. This is one of the fundamental problems in cryptography: for example, a shared secret could be used to generate a session key for subsequent communication between the agents.

The protocol is pictorially represented in Fig. 1.[1] It requires the exchange of three messages between the participating agents. Notation such as $\langle M \rangle_C$ denotes that message M is encrypted using agent C's public key. Throughout, we assume the underlying encryption algorithm to be secure and the private keys of the honest agents to be uncompromised. Therefore, only agent C can decrypt $\langle M \rangle_C$ to learn M.

[1] The original protocol includes communication between the agents and a central key server to distribute the public keys of the agents. We concentrate on the core authentication protocol, assuming all public keys to be known to all agents.

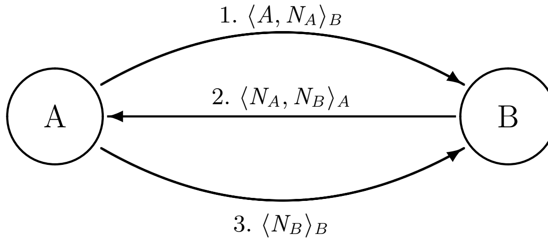

Fig. 1. Needham-Schroeder public-key protocol.

1. Alice initiates the protocol by generating a random number N_A and sending the message $\langle A, N_A \rangle_B$ to Bob (numbers such as N_A are called *nonces* in cryptographic jargon, indicating that they should be used only once by any honest agent). The first component of the message informs Bob of the identity of the initiator. The second component represents "one half" of the secret.
2. Bob similarly generates a nonce N_B and responds with the message $\langle N_A, N_B \rangle_A$. The presence of the nonce N_A generated in the first step, which only Bob could have decrypted, convinces Alice of the authenticity of the message. She therefore accepts the pair $\langle N_A, N_B \rangle$ as the common secret.
3. Finally, Alice responds with the message $\langle N_B \rangle_B$. By the same argument as above, Bob concludes that this message must originate with Alice, and therefore also accepts $\langle N_A, N_B \rangle$ as the common secret.

We assume all messages to be sent over an insecure medium. Attackers may intercept messages, store them, and perhaps replay them later. They may also participate in ordinary runs of the protocol, initiate runs or respond to runs initiated by honest agents, who need not be aware of their partners' true identity. However, even an attacker can only decrypt messages that were encrypted with his own public key.

The protocol contains a severe flaw, and the reader is invited to find it before continuing. The error was discovered some 17 years after the protocol was first published, using model checking technology [91].

2.2 A Promela Model

We represent the protocol in Promela ("protocol meta language"), the input language for the Spin model checker.[2] In order to make the analysis feasible, we make a number of simplifying assumptions:

– We consider a network of only three agents: A, B, and I(ntruder).

[2] The full code is available from the author.

- The honest agents A and B can only participate in one protocol run each. Agent A can only act as initiator, and agent B as responder. It follows that A and B need to generate at most one nonce.
- The memory of agent I is limited to a single message.

Although the protocol is very small, our simplifications are quite typical of the analysis of "real-world" systems via model checking: models are usually required to be finite-state, and the complexity of analysis typically depends exponentially on the size of those models. (Esparza's contribution to this volume surveys the state of the art concerning model checking techniques for infinite-state models.) Of course, our assumptions imply that certain errors such as "confusion" that could arise when multiple runs of the protocol interfere will go undetected in our model. This explains why model checking is considered a debugging rather than a verification technique. When no errors have been found on a small model, one can consider somewhat less stringent restrictions, as far as available resources permit. In any case, it is important to clearly identify the assumptions that underly the system model in order to assess the coverage of the analysis.

With these caveats, it is quite straightforward to write a model for the honest agents A and B from the informal description of section 2.1. PROMELA is a guarded-command language with C-like syntax; it provides primitives for message channels and operations for sending and receiving messages. We first declare an enumeration type that contains symbolic constants to make the model more readable. Because one nonce suffices for each agent, we simply assume that these have been precomputed and refer to them by symbolic names.

```
mtype = { ok, err, msg1, msg2, msg3, keyA, keyB, keyI,
          agentA, agentB, agentI, nonceA, nonceB, nonceI };
```

We represent encrypted messages as records that contain a `key` and two `data` entries. Decryption can then be modelled as pattern-matching on the `key` entry.

```
typedef Crypt { mtype key, data1, data2 };
```

The network is modelled as a single message channel shared by all three agents. For simplicity, we assume synchronous communication on the network, indicated by a buffer length of 0; this does not affect the possible communication patterns but helps to reduce the size of the model. A message on the network is modelled as a triple consisting of an identification tag (the message number), the intended receiver (which the intruder is free to ignore), and an "encrypted" message body.

```
chan network = [0] of { mtype,   /* msg# */
                        mtype,   /* receiver */
                        Crypt };
```

Figure 2 contains the PROMELA code[3] for agent A. Initially, a partner (either B or I) is chosen nondeterministically for the subsequent run (the token `::` introduces the different alternatives of nondeterministic selection), and its public

[3] In actual PROMELA, record formation is not available as a primitive operation, but must be simulated by a series of assignments.

```
mtype partnerA;
mtype statusA = err;

active proctype Alice() {
  mtype  pkey, pnonce;
  Crypt  data;

  if /* choose a partner for this run */
  :: partnerA = agentB; pkey = keyB;
  :: partnerA = agentI; pkey = keyI;
  fi;
  network ! (msg1, partnerA, Crypt{pkey, agentA, nonceA});

  network ? (msg2, agentA, data);
  (data.key == keyA) && (data.info1 == nonceA);
  pnonce = data.info2;

  network ! (msg3, partnerA, Crypt{pkey, pnonce, 0});
  statusA = ok;
}
```

Fig. 2. PROMELA code for agent A.

key is looked up. A message of type 1 is then sent to the chosen partner, after which agent A waits for a message of type 2 intended for her to arrive on the network. She verifies that the message body is encrypted with her key and that it contains the nonce sent in the first message. (PROMELA allows Boolean conditions to appear as statements; such a statement blocks if the condition is found to be false.) If so, she extracts the partner's nonce, responds with a message of type 3, and declares success. (The variable `statusA` will be used later to express correctness statements about the model.)

The code for agent B is similar, exchanging sending and reception of messages.

In contrast, the intruder cannot be modelled using a fixed protocol—the purpose of the analysis is to let SPIN find the attack if one exists at all. Instead, agent I is modelled highly nondeterministically: we describe the actions that are possible at any given state and let SPIN choose among them. The overall structure of the code shown in Fig. 3 is an infinite loop that offers a choice between receiving and sending of messages on the network.

The first alternative models the reception or interception of a message (the "don't care" variable "`_`" reflects the fact that the intruder need not respect the intended recipient of a message). The message body may be stored in the variable `intercepted`, even if it cannot be decrypted. If, moreover, the message has been encrypted for agent I, it can be analyzed to extract nonces; since the model is based on a fixed set of nonces, it is enough to set Boolean flags for nonces that the intruder has learnt so far.

```
bool  knows_nonceA, knows_nonceB;

active proctype Intruder() {
  mtype msg, recpt;
  Crypt data, intercepted;
  do
  :: network ? (msg, _, data) ->
     if /* perhaps store the message */
     :: intercepted = data;
     :: skip;
     fi;
     if /* record newly learnt nonces */
     :: (data.key == keyI) ->
        if
        :: (data.info1 == nonceA) || (data.info2 == nonceA)
           -> knows_nonceA = true;
        :: else -> skip;
        fi;
        /* similar for knows_nonceB */
     :: else -> skip;
     fi;
  :: /* Replay or send a message */
     if /* choose message type */
     :: msg = msg1;
     :: msg = msg2;
     :: msg = msg3;
     fi;
     if /* choose recipient */
     :: recpt = agentA;
     :: recpt = agentB;
     fi;
     if /* replay intercepted message or assemble it */
     :: data = intercepted;
     :: if
        :: data.info1 = agentA;
        :: data.info1 = agentB;
        :: data.info1 = agentI;
        :: knows_nonceA -> data.info1 = nonceA;
        :: knows_nonceB -> data.info1 = nonceB;
        :: data.info1 = nonceI;
        fi;
        /* similar for data.info2 and data.key */
     fi;
     network ! (msg, recpt, data);
  od;
}
```

Fig. 3. PROMELA code for agent I.

The second alternative represents agent I sending a message. There are two subcases: either replay a previously intercepted message or construct a new message from the information learnt so far. Note that we allow arbitrary ("type-correct") entries for the unencrypted fields of a message. Of course, most of the resulting combinations can be immediately recognized as inappropriate by the honest agents. Our model therefore contains many deadlocks, which we ignore during the following analysis.

2.3 Model Checking the Protocol

The purpose of the protocol is to ensure mutual authentication (of honest agents) while maintaining secrecy. In other words, whenever both A and B have successfully completed a run of the protocol, then A should believe her partner to be B if and only if B believes to talk to A. Moreover, if A successfully completes a run with B then the intruder should not have learnt A's nonce, and similarly for B. These properties are can be expressed in temporal logic (cf. section 3.2) as follows:

$$\begin{array}{l}
\mathbf{G}(statusA = ok \wedge statusB = ok \implies \\
\qquad (partnerA = agentB \Leftrightarrow partnerB = agentA)) \\
\mathbf{G}(statusA = ok \wedge partnerA = agentB \implies \neg knows_nonceA) \\
\mathbf{G}(statusB = ok \wedge partnerB = agentA \implies \neg knows_nonceB)
\end{array}$$

We present Spin with the model of the protocol and the first formula. In a fraction of a second, Spin declares the property violated and outputs a run that contains the attack. The run is visualized as a message sequence chart, shown in Fig. 4: Alice initiates a protocol run with Intruder who in turn (but masquerading as A) starts a run with Bob, using the nonce received in the first message. Bob replies with a message of type 2 that contains both A's and B's nonces, encrypted for A. Although agent I cannot decrypt that message itself, it forwards it to A. Unsuspecting, Alice finds her nonce, returns the second nonce to her partner I, and declares success. This time, agent I can decrypt the message, extracts B's nonce and sends it to B who is also satisfied. As a result, we have reached a state where A correctly believes to have completed a run with I, but B is fooled into believing to talk to A. The same counterexample will be produced when analysing the third formula, whereas the second formula is declared to hold of the model.

The counterexample produced by Spin makes it easy to trace the error in the protocol to a lack of explicitness in the second message: the presence of the expected nonce is not sufficient to prove the origin of the message. To avoid the attack, the second message should therefore be replaced with $\langle B, N_A, N_B \rangle$. After this modification, Spin confirms that all three formulas hold of the model—which of course does not prove the correctness of the protocol (see, e.g., [106] for work on the formal verification of cryptographic protocols using interactive theorem proving).

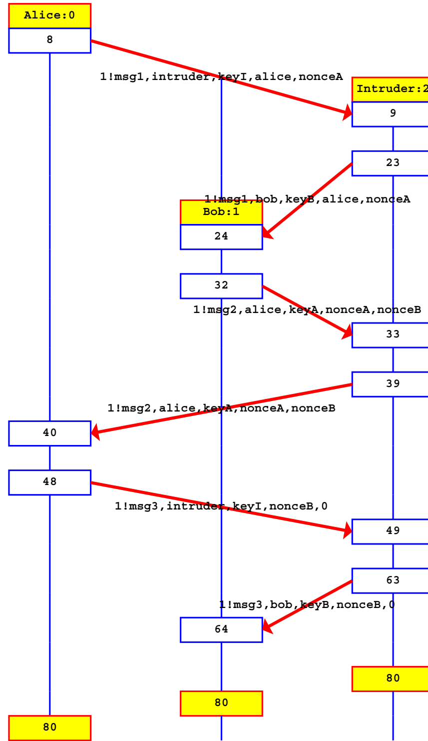

Fig. 4. Message sequence chart visualizing the attack.

3 Systems and Properties

Reactive systems can be broadly classified as *distributed* systems whose subcomponents are spatially separated and *concurrent* systems that share resources such as processors and memories. Distributed systems communicate by *message passing*, whereas concurrent systems may use *shared variables*. Concurrent processes may share a common clock and execute in lock-step (*time-synchronous* systems, typical for hardware verification problems) or operate asynchronously, sharing a common processor. In the latter case, one will typically assume *fairness conditions* that ensure processes that could execute are eventually scheduled for execution. A common framework for the representation of these different kinds of systems is provided by the concept of *transition systems*. Properties of (runs of) transition systems are conveniently expressed in temporal logic.

3.1 Transition Systems

Definition 1. *A* transition system $\mathcal{T} = (S, I, \mathcal{A}, \delta)$ *is given by a set* S *of* states*, a non-empty subset* $I \subseteq S$ *of* initial *states, a set* $\mathcal{A}$ *of* actions*, and a total* transition relation $\delta \subseteq S \times \mathcal{A} \times S$ *(that is, we require that for every state* $s \in S$ *there exist* $A \in \mathcal{A}$ *and* $t \in S$ *such that* $(s, A, t) \in \delta$*).*

An action $A \in \mathcal{A}$ *is called* enabled *at state* $s \in S$ *iff* $(s, A, t) \in \delta$ *holds for some* $t \in S$.

A run *of* $\mathcal{T}$ *is an infinite sequence* $\rho = s_0 s_1 \ldots$ *of states* $s_i \in S$ *such that* $s_0 \in I$ *and for all* $i \in \mathbb{N}$, $(s_i, A_i, s_{i+1}) \in \delta$ *holds for some* $A_i \in \mathcal{A}$.

A transition system specifies the allowed evolutions of the system: starting from some initial state, the system evolves by performing actions that take the system to a new state. Slightly different definitions of transition systems abound in the literature. For example, actions are sometimes not explicitly identified. We have assumed the transition relation to be total in order to simplify some of the definitions below. Totality can be ensured by including a *stuttering action* that does not change the state; only the stuttering action is enabled in deadlock or quiescent states. Definition 1 is often augmented by fairness conditions, see section 4.2. Some papers use the term *Kripke structure* instead of transition system, in honor of the logician Saul A. Kripke who used transition systems to define the semantics of modal logics [78].

In practice, reactive systems are described using modelling languages, including (pseudo) programming languages such as PROMELA, but also process algebras or Petri nets. The operational semantics of these formalisms is conveniently defined in terms of transition systems. However, the transition system that corresponds to such a description is typically of size exponential in the length of the description. For example, the state space of a shared-variable program is the product of the variable domains. Modelling languages and their associated model checkers are usually optimized for particular kinds of systems such as synchronous shared-variable programs or asynchronous communication protocols. In particular, for systems composed of several processes it is advantageous to exploit the process structure and avoid the explicit construction of a single transition system that represents the joint behavior of processes. This will be further explored in section 4.4.

3.2 Properties and Temporal Logic

Given a transition system $\mathcal{T}$, we can ask questions such as the following:

- Are any "undesired" states reachable in $\mathcal{T}$, such as states that represent a deadlock, a violation of mutual exclusion etc.?
- Are there runs of $\mathcal{T}$ such that, from some point onwards, some "desired" state is never reached or some action never executed? Such runs may represent livelocks where, for example, some process is prevented from entering its critical section, although other components of the system may still make progress.

– Is some initial system state of $\mathcal{T}$ reachable from every state? In other words, can the system be reset?

Temporal logic [45,79,94,95,117] is a convenient language to formally express such properties. Let us first consider temporal logic of linear time whose formulas express properties of runs of transition systems. Assume given a denumerable set $\mathcal{V}$ of atomic propositions, which represent properties of individual states.

Definition 2. *Formulas of propositional temporal logic* **PTL** *of linear time are inductively defined as follows:*

- *Every atomic proposition $v \in \mathcal{V}$ is a formula.*
- *Boolean combinations of formulas are formulas.*
- *If φ and ψ are formulas then so are $\mathbf{X}\,\varphi$ ("next φ") and $\varphi\,\mathbf{U}\,\psi$ ("φ until ψ").*

PTL formulas are interpreted over *behaviors*, that is, ω-sequences of states. We assume that atomic propositions $v \in \mathcal{V}$ can be evaluated at states $s \in S$ and write $s(\mathcal{V})$ to denote the set of propositions true at state s. For a behavior $\sigma = s_0 s_1 \ldots$, we let σ_i denote the state s_i and $\sigma|_i$ the suffix $s_i s_{i+1} \ldots$ of σ.

Definition 3. *The relation $\sigma \models \varphi$ ("φ holds of σ") is inductively defined as follows:*

- *$\sigma \models v$ (for $v \in \mathcal{V}$) iff $v \in \sigma_0(\mathcal{V})$.*
- *The semantics of boolean combinations is defined as usual.*
- *$\sigma \models \mathbf{X}\,\varphi$ iff $\sigma|_1 \models \varphi$.*
- *$\sigma \models \varphi\,\mathbf{U}\,\psi$ iff for some $k \geq 0$, $\sigma|_k \models \psi$ and $\sigma|_j \models \varphi$ holds for all $0 \leq j < k$.*

Other useful **PTL** formulas can be introduced as abbreviations: $\mathbf{F}\,\varphi$ ("*finally* φ", "*eventually* φ") is defined as **true** $\mathbf{U}\,\varphi$; it asserts that φ holds of some suffix. The dual formula $\mathbf{G}\,\varphi \equiv \neg\,\mathbf{F}\,\neg\varphi$ ("*globally* φ", "*always* φ") requires φ to hold of all suffixes. The formula $\varphi\,\mathbf{W}\,\psi$ ("*φ waits for ψ*", "*φ unless ψ*") is defined as $(\varphi\,\mathbf{U}\,\psi) \vee \mathbf{G}\,\varphi$ and requires φ to hold for as long as ψ does not hold; unlike $\varphi\,\mathbf{U}\,\psi$, it does not require ψ to become true eventually.

The following formulas are examples for typical correctness assertions about a two-process resource manager. We assume req_i and $owns_i$ to be atomic propositions true when process i has requested the resource or when it owns the resource.

$\mathbf{G}\,\neg(owns_1 \wedge owns_2)$: It is never the case that both processes own the resource. In general, properties of the form $\mathbf{G}\,p$, for non-temporal formulas p, express *system invariants.*

$\mathbf{G}(req_1 \implies \mathbf{F}\,owns_1)$: Whenever process 1 has requested the resource, it will eventually obtain it. Formulas of this form are often called *response properties* [93].

$\mathbf{G}\,\mathbf{F}(req_1 \wedge \neg(owns_1 \vee owns_2)) \implies \mathbf{G}\,\mathbf{F}\,owns_1$: If it is infinitely often the case that process 1 has requested the resource when the resource is free, then process 1 infinitely often owns the resource. This formula expresses a (strong) fairness condition for process 1.

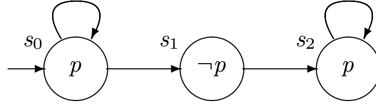

Fig. 5. A transition system $\mathcal{T}$ such that $\mathcal{T} \models \mathbf{F}\,\mathbf{G}\,p$ but $\mathcal{T} \not\models \mathbf{AF}\,\mathbf{AG}\,p$.

$\mathbf{G}(\mathit{req}_1 \wedge \mathit{req}_2 \implies (\neg \mathit{owns}_2 \;\mathbf{W}\; (\mathit{owns}_2 \;\mathbf{W}\; (\neg \mathit{owns}_2 \;\mathbf{W}\; \mathit{owns}_1))))$:
Whenever both processes compete for the resource, process 2 will be granted the resource at most once before it is granted to process 1. This property, known as "1-bounded overtaking", is an example for a *precedence property*. It is best understood as asserting the existence of four, possibly empty or right-open, intervals that satisfy the respective conditions.

PTL formulas assert properties of single behaviors, but we are interested in *system validity*: we say that formula φ holds of $\mathcal{T}$ (written $\mathcal{T} \models \varphi$) if φ holds of all runs of $\mathcal{T}$. In this sense, **PTL** formulas express *correctness properties* of a system. The existence of a run satisfying a certain property cannot be expressed in **PTL**. Such *possibility properties* are the domain of branching-time logics such as the logic **CTL** (*computation tree logic* [25]).

Definition 4. *Formulas of propositional* **CTL** *are inductively defined as follows:*

- *Every atomic proposition $v \in \mathcal{V}$ is a formula.*
- *Boolean combinations of formulas are formulas.*
- *If φ and ψ are formulas then $\mathbf{EX}\,\varphi$, $\mathbf{EG}\,\varphi$, and $\varphi\;\mathbf{EU}\;\psi$ are formulas.*

CTL formulas are interpreted at the states of a transition system. A *path* in $\mathcal{T}$ is an ω-sequence $\sigma = s_0 s_1 \ldots$ of states related by δ; it is an s-*path* if $s = s_0$.

Definition 5. *The relation $\mathcal{T}, s \models \varphi$ is inductively defined as follows:*

- *$\mathcal{T}, s \models v$ (for $v \in \mathcal{V}$) iff $v \in s(\mathcal{V})$.*
- *The semantics of boolean combinations is defined as usual.*
- *$\mathcal{T}, s \models \mathbf{EX}\,\varphi$ iff there exists an s-path $s_0 s_1 \ldots$ such that $\mathcal{T}, s_1 \models \varphi$.*
- *$\mathcal{T}, s \models \mathbf{EG}\,\varphi$ iff there is an s-path $s_0 s_1 \ldots$ such that $\mathcal{T}, s_i \models \varphi$ holds for all i.*
- *$\mathcal{T}, s \models \varphi\;\mathbf{EU}\;\psi$ iff there exist an s-path $s_0 s_1 \ldots$ and $k \geq 0$ such that $\mathcal{T}, s_k \models \psi$ and $\mathcal{T}, s_j \models \varphi$ holds for all $0 \leq j < k$.*

Derived **CTL**-formulas include $\mathbf{EF}\,\varphi \equiv \mathbf{true}\;\mathbf{EU}\;\varphi$, $\mathbf{AX}\,\varphi \equiv \neg\,\mathbf{EX}\,\neg\varphi$, and $\mathbf{AG}\,\varphi \equiv \neg\,\mathbf{EF}\,\neg\varphi$. For example, the formula $\mathbf{AG}\,\neg(\mathit{owns}_1 \wedge \mathit{owns}_2)$ expresses mutual exclusion for the two-process resource manager, whereas $\mathbf{AG}(\mathit{req}_1 \implies \mathbf{EF}\,\mathit{owns}_1)$ asserts that whenever process 1 requests the resource, it *can* eventually obtain the resource, although there may be executions that do not honor the request. The formula $\mathbf{AG}\,\mathbf{EF}\,\mathit{init}$ (for a suitable predicate *init*) asserts that the system is resettable.

System validity for **CTL**-formulas is defined by $\mathcal{T} \models \varphi$ if $\mathcal{T}, s \models \varphi$ holds for all initial states s of $\mathcal{T}$. The expressiveness of **PTL** and **CTL** can be compared by analyzing which properties of transition systems can be formulated. It turns out that neither logic subsumes the other one [84,41,43]: whereas **PTL** is clearly incapable of expressing possibility properties, fairness properties cannot be stated in **CTL**. More specifically, there is no **CTL** formula that is system valid iff the **PTL** formula $\mathbf{F}\,\mathbf{G}\,\varphi$ is. In particular, it does not correspond to $\mathbf{AF}\,\mathbf{AG}\,\varphi$, as shown in Fig. 5: every run of the transition system $\mathcal{T}$ satisfies $\mathbf{F}\,\mathbf{G}\,p$ (either it stays in state s_0 forever or it ends in state s_2), but $\mathcal{T}, s_0 \not\models \mathbf{AF}\,\mathbf{AG}\,p$ (for the run that stays in state s_0 there is always the possibility to move to state s_1).

Extensions and variations. The lack of expressiveness of **CTL** is due to the requirement that path quantifiers (**E**, **A**) and temporal operators (**X**, **G**, **U**) alternate. The logic **CTL*** [41,43] removes this restriction and (strictly) subsumes both **PTL** and **CTL**. For example, the **CTL*** formula $\mathbf{AFG}\,p$ is system valid iff the **PTL** formula $\mathbf{F}\,\mathbf{G}\,p$ is.

The *propositional μ-calculus* [77], also known as μ**TL**, allows properties to be defined as smallest or greatest fixed points, generalizing recursive characterizations of temporal operators such as

$$\mathbf{EG}\,\varphi \quad\equiv\quad \varphi \wedge \mathbf{EX}\,\mathbf{EG}\,\varphi$$

It strictly subsumes the logic **CTL***. For example, the formula $\nu X.\varphi \wedge \mathbf{AX}\,\mathbf{AX}\,X$ asserts that φ holds at every state with even distance from the current state.

Alternating-time temporal logic [6] refines the path quantifiers of branching time temporal logics by allowing references to different processes (or agents) of a reactive system. One can, for example, assert that the resource manager can ensure mutual exclusion between the clients, or that the manager and client 1 can cooperate to prevent client 2 to access the resource.

3.3 ω-Automata

We have seen how to interpret temporal logic formulas over transition systems. On the other hand, one can construct a finite automaton that represents the models of a given **PTL** formula. This close connection between temporal logic and automata is the basis for **PTL** decision procedures and model checking algorithms because many properties of finite automata are decidable, even when applied to ω-words. The theory of automata over infinite words and trees was initiated by Büchi [19], Muller [101], and Rabin [110]. We present some of its basic elements; for more comprehensive expositions see the excellent survey articles by Thomas [120,121].

Definition 6. *A* Büchi automaton $\mathcal{B} = (Q, I, \delta, F)$ *over an alphabet* Σ *is given by a finite set* Q *of* locations[4], *a non-empty set* $I \subseteq Q$ *of* initial locations, *a* transition relation $\delta \subseteq Q \times \Sigma \times Q$, *and a set* $F \subseteq Q$ *of* accepting locations.

[4] We use the term *locations* rather than the conventional *states* to avoid confusion with the states of transition systems and temporal logic.

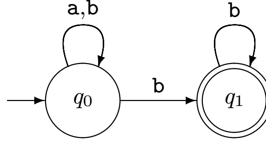

Fig. 6. A Büchi automaton.

A run *of $\mathcal{B}$ over an ω-word $w = a_0 a_1 \ldots \in \Sigma^\omega$ is an infinite sequence $\rho = q_0 q_1 \ldots$ of locations $q_i \in Q$ such that $q_0 \in I$ and $(q_i, a_i, q_{i+1}) \in \delta$ holds for all $i \in \mathbb{N}$. The run ρ is* accepting *iff there exists some $q \in F$ such that $q_i = q$ holds for infinitely many $i \in \mathbb{N}$.*

The language *$\mathcal{L}(\mathcal{B}) \subseteq \Sigma^\omega$ is the set of ω-words for which there exists some accepting run ρ of $\mathcal{B}$. A language $L \subseteq \Sigma^\omega$ is called ω-*regular *iff $L = \mathcal{L}(\mathcal{B})$ for some Büchi automaton $\mathcal{B}$.*

Büchi automata are presented just as ordinary (non-deterministic) finite automata over finite words [68]. The notion of "final locations", which obviously does not apply to ω-words, is replaced by the requirement that a run passes infinitely often through an accepting location. Figure 6 shows a two-location Büchi automaton with initial location q_0 and accepting location q_1 whose language is the set of ω-words over $\{\mathtt{a}, \mathtt{b}\}$ that contain only finitely many a's.

Many properties of classical finite automata carry over to Büchi automata. For example, the emptiness problem is decidable.

Theorem 1. *For a Büchi automaton $\mathcal{B}$ with n locations, it is decidable in time $O(n)$ whether $\mathcal{L}(\mathcal{B}) = \emptyset$.*

Proof. Because Q is finite, $\mathcal{L}(\mathcal{B}) \neq \emptyset$ iff there exist locations $q_0 \in I$, $q \in F$ and finite words $x \in \Sigma^*$ and $y \in \Sigma^+$ such that $q_0 \stackrel{x}{\Rightarrow} q$ and $q \stackrel{y}{\Rightarrow} q$ (where $q \stackrel{w}{\Rightarrow} q'$ means that there is a path in $\mathcal{B}$ from location q to q' labelled with w). The existence of such paths can be decided in linear time using the Tarjan-Paige algorithm [119] that enumerates the strongly connected components of $\mathcal{B}$ reachable from locations in I, and checking whether some SCC contains some accepting location. □

Observe that the construction used in the proof of theorem 1 implies that an ω-regular language is non-empty iff it contains some word of the form xy^ω where $x \in \Sigma^*$ and $y \in \Sigma^+$.

Unlike the case of standard finite automata, deterministic Büchi automata are strictly weaker than non-deterministic ones. For example, there is no deterministic Büchi automaton that accepts the same language as the automaton $\mathcal{B}$ of Fig. 6. Intuitively, the reason is that $\mathcal{B}$ uses unbounded non-determinism to "guess" when it has seen the last input a (for a rigorous proof see e.g. [120]). It is therefore impossible to prove closure of the class of ω-regular languages under complement in the standard way (first construct a deterministic Büchi

automaton equivalent to the initial one, then complement the set of accepting locations). Nevertheless, Büchi [19] has shown that the complement of an ω-regular language is again ω-regular. His proof relied on combinatorial arguments (Ramsey's theorem) and was non-constructive. A succession of papers has replaced this argument with explicit constructions, culminating in the following result due to Safra [111] of essentially optimal complexity; Thomas [121,122] explains different strategies for proving closure under complement.

Proposition 1. *For a Büchi automaton $\mathcal{B}$ with n locations over alphabet Σ there is a Büchi automaton $\overline{\mathcal{B}}$ with $2^{O(n \log n)}$ locations such that $\mathcal{L}(\overline{\mathcal{B}}) = \Sigma^\omega \setminus \mathcal{L}(\mathcal{B})$.*

Other types of ω-automata have also been considered. *Generalized Büchi automata* define the acceptance condition by a (finite) set $\mathcal{F} = \{F_1, \ldots, F_n\}$ of sets of locations [126]. A run is accepting if some location from every F_i is visited infinitely often. Using a counter modulo n, it is not difficult to simulate a generalized Büchi automaton by a standard one. The algorithm for checking nonemptiness can be adapted by searching some strongly connected component that contains some location from every F_i. *Muller automata* also specify the acceptance condition as a set $\mathcal{F}$ of set of locations; a run is accepting if the set of locations that appears infinitely often is an element of $\mathcal{F}$. Rabin and Streett automata use pairs of sets of locations to define even more elaborate acceptance conditions, such as requiring that if locations in a set $R \subseteq Q$ are visited infinitely often then there are also infinitely many visits to locations in another set $G \subseteq Q$. Streett automata can be exponentially more succinct than Büchi automata, and deterministic Rabin and Streett automata are at the heart of Safra's proof. It is also possible to place acceptance conditions on the transitions rather than the locations [7,36].

Alternating automata [102] present a more radical departure from the format of Büchi automata and have attracted considerable interest in recent years. The basic idea is to allow the automaton to make a transition from one location to several successor locations that are simultaneously active. One way to define such a relation is to let $\delta(q, a)$ be a positive Boolean formula with the locations as atomic propositions. For example,

$$\delta(q_1, a) \quad = \quad (q_2 \wedge q_3) \vee q_4$$

specifies that whenever location q_1 is active and input symbol $a \in \Sigma$ is read, the automaton moves to locations q_2 and q_3 in parallel, or to location q_4. Runs of alternating automata are no longer infinite sequences, but rather infinite trees or dags of locations. Although they also define the class of ω-regular languages, alternating automata can be exponentially more succinct than Büchi automata, due to their inherent parallelism. On the other hand, checking for nonemptiness is normally of exponential complexity.

3.4 Temporal Logic and Automata

We can consider a behavior as an ω-word over the alphabet $2^{\mathcal{V}}$, identifying a system state s and the set $s(\mathcal{V})$ of atomic propositions that s satisfies. From

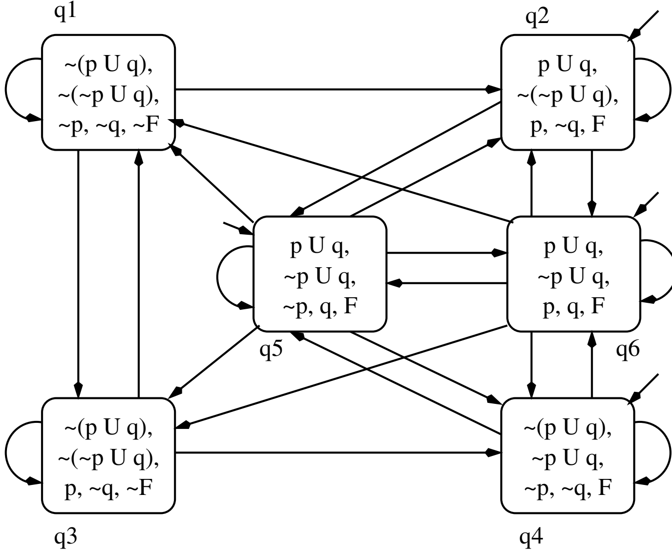

Fig. 7. Büchi automaton for $F \equiv (p \mathbf{U} q) \vee (\neg p \mathbf{U} q)$.

this perspective, **PTL** formulas and ω-automata are two different formalisms to describe ω-words, and it is interesting to compare their expressiveness. For example, the Büchi automaton of Fig. 6 can be identified with the **PTL** formula $\mathbf{F}\,\mathbf{G}\,\mathsf{b}$.

We outline a construction of a generalized Büchi automaton $\mathcal{B}_\varphi$ for a given **PTL** formula φ such that $\mathcal{B}_\varphi$ accepts precisely those runs over which φ holds. In view of the high complexity of complementation (cf. Prop. 1), the construction is not defined by induction on the structure of φ but is based on a "global" construction that considers all subformulas of φ simultaneously. The *Fischer-Ladner closure* $\mathcal{C}(\varphi)$ of formula φ is the set of subformulas of φ and their complements, identifying $\neg\neg\psi$ and ψ. The locations of $\mathcal{B}_\varphi$ are subsets of $\mathcal{C}(\varphi)$, with the intuition that an accepting run of $\mathcal{B}_\varphi$ from location q satisfies the formulas in q. More precisely, the locations q of $\mathcal{B}_\varphi$ are all subsets of $\mathcal{C}(\varphi)$ that satisfy the following *healthiness conditions*:

- For all $\psi \in \mathcal{C}(\varphi)$, either $\psi \in q$ or $\neg\psi \in q$, but not both.
- If $\psi_1 \vee \psi_2 \in \mathcal{C}(\varphi)$ then $\psi_1 \vee \psi_2 \in q$ iff $\psi_1 \in q$ or $\psi_2 \in q$.
- Conditions for other boolean combinations are similar.
- If $\psi_1 \mathbf{U} \psi_2 \in q$, then $\psi_2 \in q$ or $\psi_1 \in q$.
- If $\psi_1 \mathbf{U} \psi_2 \in \mathcal{C}(\varphi) \setminus q$, then $\psi_2 \notin q$.

The initial locations of $\mathcal{B}_\varphi$ are those locations containing φ. The transition relation δ of $\mathcal{B}_\varphi$ is defined such that $(q, s, q') \in \delta$ iff all of the following conditions hold:

- $s = q \cap \mathcal{V}$ is the set of atomic propositions that appear in $\mathcal{V}$; these must obviously be satisfied immediately by any run starting in q.

– q' contains ψ (resp., does not contain ψ) if $\mathbf{X}\,\psi \in q$ (resp., $\mathbf{X}\,\psi \in \mathcal{C}(\varphi) \setminus q$).
– If $\psi_1 \,\mathbf{U}\, \psi_2 \in q$ and $\psi_2 \notin q$ then $\psi_1 \,\mathbf{U}\, \psi_2 \in q'$.
– If $\psi_1 \,\mathbf{U}\, \psi_2 \in \mathcal{C}(\varphi) \setminus q$ and $\psi_1 \in q$ then $\psi_1 \,\mathbf{U}\, \psi_2 \notin q'$.

The healthiness and next-state conditions are justified by propositional consistency and by the "recursion law"

$$\psi_1 \,\mathbf{U}\, \psi_2 \quad \equiv \quad \psi_2 \vee (\psi_1 \wedge \mathbf{X}(\psi_1 \,\mathbf{U}\, \psi_2))$$

In particular, they ensure that whenever some location contains $\psi_1 \,\mathbf{U}\, \psi_2$, subsequent locations contain ψ_1 for as long as they do not contain ψ_2.

It remains to define the acceptance conditions of $\mathcal{B}_\varphi$, which must ensure that every location containing some formula $\psi_1 \,\mathbf{U}\, \psi_2$ will be followed by some location containing ψ_2. Let $\psi_1^1 \,\mathbf{U}\, \psi_2^1, \ldots, \psi_1^k \,\mathbf{U}\, \psi_2^k$ be all subformulas of this form in $\mathcal{C}(\varphi)$. Then $\mathcal{B}_\varphi$ has the acceptance condition $\mathcal{F} = \{F_1, \ldots, F_k\}$ where F_i is the set of locations that do not contain $\psi_1^i \,\mathbf{U}\, \psi_2^i$ or that contain ψ_2^i. As an example, Fig. 7 shows the automaton $\mathcal{B}_F$ for the formula $F \equiv (p \,\mathbf{U}\, q) \vee (\neg p \,\mathbf{U}\, q)$. For clarity, we have omitted the edge labels, which are simply the set of atomic propositions contained in the source location. The acceptance sets corresponding to the subformulas $p \,\mathbf{U}\, q$ and $\neg p \,\mathbf{U}\, q$ are $\{q_1, q_3, q_4, q_5, q_6\}$ and $\{q_1, q_2, q_3, q_5, q_6\}$. For example, they ensure that no accepting run remains forever in location q_2.

This construction, which is very similar to a tableau construction [128], implies the existence of a Büchi automaton that accepts precisely the models of any given **PTL** formula. The following proposition is due to [87,126].

Proposition 2. *For every* **PTL** *formula φ of length n there exists a Büchi automaton $\mathcal{B}_\varphi$ with $2^{O(n)}$ locations that accepts precisely the behaviors of which φ holds.*

Combining proposition 2 and theorem 1, it follows that the satisfiability problem for **PTL** is solvable in exponential time by checking whether $\mathcal{L}(\mathcal{B}_\varphi) = \emptyset$; in fact, Sistla and Clarke [114] have shown that the **PTL** satisfiability problem is PSPACE-complete. Note that the above construction invariably produces a Büchi automaton $\mathcal{B}_\varphi$ whose size is exponential in the length of the formula φ. Constructions that try to avoid this exponential blow-up [56,38,36] are the basis for actual implementations.

On the other hand, it is not the case that every ω-regular language can be defined by a **PTL** formula: Kamp [74] has shown that **PTL** formulas can define exactly the same behaviors as first-order logic formulas of the *monadic theory of linear orders*, that is, formulas built from $=$, $<$, and unary predicates $P_v(x)$, for $v \in \mathcal{V}$, interpreted over the natural numbers, see also [54]. This fragment of first-order logic is known to define the set of *star-free* ω-regular languages, a result due to McNaughton and Papert [98,121]. For example, the set of behaviors such that proposition p is true at the even positions (and may be true or false elsewhere) is not **PTL**-definable [128]. To attain the level of expressiveness of ω-regular languages (which, by Büchi's theorem, is that of the monadic second order theory of linear orders), **PTL** can be augmented by so-called "automaton operators" [128], by fixed-point formulas [117] or by quantification over atomic

propositions. Unfortunately, the satisfiability problem for some of these logics is of non-elementary complexity; moreover, few applications seem to require the added expressiveness. Nevertheless, such a decision procedure has been implemented in Mona [76] and performs surprisingly well on practical examples.

Automata for other temporal logics. Automata-theoretic characterizations of branching-time logics [80] are based on tree automata [120,121], which again define a notion of regular tree languages. Alternating automata allow for a rather uniform presentation of decision procedures for linear-time, branching-time, and alternating-time temporal logics [103,125,82], based on different restrictions on the automaton format. An essentially equivalent approach that does not mention automata can be formulated in terms of logical games [118]. In particular, winning strategies replace the traditional presentation of counter-examples; this can give better feedback to the user who can then explore different scenarios that violate a property. The model checkers Truth [85] and CWB-NC [31] are based on these concepts.

4 Algorithms for Model Checking

Given a transition system $\mathcal{T}$ and a formula φ, the model checking problem is to decide whether $\mathcal{T} \models \varphi$ holds or not. If not, the model checker should provide an explanation why, in the form of a counterexample (i.e., a run of $\mathcal{T}$ that violates φ). For this to be feasible, $\mathcal{T}$ is usually required to be finite-state.

In accordance with the two parameters of the model checking problem ($\mathcal{T}$ and φ), there are two basic strategies when designing a model checking algorithm: "global" algorithms recurse on the structure of φ and evaluate each of its subformulas over all of $\mathcal{T}$. "Local" algorithms, in contrast, explore only parts of the state space of $\mathcal{T}$, but check all subformulas of φ in the process. The choice between global and local model checking algorithms does not affect the worst-case complexity of model checking algorithms, but the average behavior on practical examples can differ greatly. Observe that local algorithms may even be able to find errors of infinite-state systems; this is also true for global algorithms that represent the state space of $\mathcal{T}$ in an implicit form, as considered in section 4.3. Traditionally, **PTL** model checking has been based on the local approach, while model checkers for **CTL** and other branching-time logics have used global algorithms.

4.1 Local PTL Model Checking

The model checking problem for **PTL** can be restated as follows: given $\mathcal{T}$ and φ, does there exist a run of $\mathcal{T}$ that does not satisfy φ? This is a refinement of the satisfiability problem considered in section 3.4: instead of asking whether $\mathcal{L}(\mathcal{B}_{\neg\varphi}) = \emptyset$, we now ask whether the language defined by the product of $\mathcal{T}$ and $\mathcal{B}_{\neg\varphi}$ is empty or not.

Formally, assume given a finite transition system $\mathcal{T} = (S, I, \mathcal{A}, \delta_{\mathcal{T}})$ and a Büchi automaton $\mathcal{B}_{\neg\varphi} = (Q, J, \delta_{\mathcal{B}}, F)$ that accepts precisely those behaviors

```
dfs(boolean search_cycle) {
   p = top(stack);
   foreach (q in successors(p)) {
      if (search_cycle and (q == seed))
         report acceptance cycle and exit;
      if ((q, search_cycle) not in visited) {
         push q onto stack;
         enter (q, search_cycle) into visited;
         dfs(search_cycle);
         if (not search_cycle and (q is accepting)) {
            seed = q; dfs(true);
   } } }
   pop(stack);
}
// initialization
stack = emptystack(); visited = emptyset(); seed = nil;
foreach initial pair p {
   push p onto stack;
   enter (p, false) into visited;
   dfs(false)
}
```

Fig. 8. On-the-fly **PTL** model checking algorithm.

that do not satisfy φ. The model checking algorithm operates on pairs (s, q) of system states and automaton locations. A pair (s_0, q_0) is *initial* if $s_0 \in I$ and $q_0 \in J$ are initial for $\mathcal{T}$ and $\mathcal{B}_{\neg\varphi}$, respectively. A pair (s', q') is a *successor* of (s, q) if both $(s, A, s') \in \delta_{\mathcal{T}}$ (for some $A \in \mathcal{A}$) and $(q, s(\mathcal{V}), q') \in \delta_{\mathcal{B}}$ hold: $\mathcal{T}$ and $\mathcal{B}_{\neg\varphi}$ make joint transitions, the input for $\mathcal{B}_{\neg\varphi}$ being determined by the values of the atomic propositions at the current system state. A pair (s, q) is *accepting* if $q \in F$ is an accepting automaton location; recall that $\mathcal{T}$ does not define an accepting condition. In particular, we assume any fairness conditions to be expressed as part of the formula φ.

As in the proof of theorem 1, $\mathcal{T}$ and $\mathcal{B}_{\neg\varphi}$ admit a joint execution iff there is some accepting pair that is reachable from some initial pair and from itself. The model checking algorithm shown in Fig. 8 is due to Courcoubetis et al [34]. It is called an “on-the-fly” algorithm because the exploration of reachable pairs is interleaved with the search for acceptance cycles. The algorithm maintains a stack of pairs whose successors need to be explored (resulting in a depth-first search) and a set of pairs that have already been visited. Starting from the initial pairs, the procedure `dfs` generates reachable pairs until some accepting pair is found. At this point, the search switches to cycle search mode (indicated by the boolean parameter `search_cycle`) and tries to find a path that leads back to the accepting pair. Pairs that have already been encountered in the current search mode are not explored further. Courcoubetis et al. [34] have shown that the algorithm will find some acceptance cycle if one exists, although it is not guaranteed to find all cycles (even if the search were continued instead of exiting).

When an acceptance cycle is found, the sequence of system states contained in the stack represents a run of $\mathcal{T}$ that violates formula φ and can be displayed to the user as a counter-example. Observe that the algorithm of Fig. 8 needs to store only the path back from the current pair back to the initial pair that it started from, and the set of visited pairs. In particular, it does not have to construct the entire product automaton. Of course, when no acceptance cycle is found (and the system is declared error-free), all reachable pairs will have to be explored eventually. However, state exploration stops as soon as an error has been detected. This can be an important practical advantage: the state space of a correct system is constrained by its invariants, which are usually broken when errors are introduced. It is therefore quite common for buggy systems to have many more reachable states, and resources could easily be exhausted if all of them had to be explored.

For large models, storing the set of visited pairs may become a problem. If one is willing to trade complete coverage for the ability to analyze systems that would otherwise be unmanageable, one can instead maintain a set of *hash codes* of visited pairs, possibly using several hashing functions [66].

The model checking algorithm of Fig. 8 has time complexity linear in the product of the sizes of $\mathcal{T}$ and of $\mathcal{B}_{\neg\varphi}$; by proposition 2 the latter can be exponential in the size of φ. However, correctness assertions are often rather short, and as we mentioned in section 3.1, the size of $\mathcal{T}$ can be exponential in the size of the description input to the model checker. Therefore, in practice the size of the transition system is the limiting factor. Given current technology, the analysis of systems on the order of 10^6–10^7 reachable states is feasible. Techniques that try to overcome this limit are described in section 4.4.

4.2 Global CTL Model Checking

Let us now consider global model checking algorithms for the logic **CTL**. By $[\![\psi]\!]_{\mathcal{T}}$ (for a **CTL** formula ψ) we denote the set of states s of $\mathcal{T}$ such that $\mathcal{T}, s \models \psi$. The model checking problem can then be rephrased as deciding whether $I \subseteq [\![\varphi]\!]_{\mathcal{T}}$ holds. The satisfaction sets $[\![\psi]\!]_{\mathcal{T}}$ can be computed by induction on the structure of ψ, as follows:

$$\begin{aligned}
[\![v]\!]_{\mathcal{T}} &= \{s : v \in s(\mathcal{V})\} \quad (\text{for } v \in \mathcal{V}) \\
[\![\neg\psi]\!]_{\mathcal{T}} &= S \setminus [\![\psi]\!]_{\mathcal{T}} \\
[\![\psi_1 \vee \psi_2]\!]_{\mathcal{T}} &= [\![\psi_1]\!]_{\mathcal{T}} \cup [\![\psi_2]\!]_{\mathcal{T}} \\
[\![\mathbf{EX}\,\psi]\!]_{\mathcal{T}} &= \delta^{-1}([\![\psi]\!]_{\mathcal{T}}) \quad = \quad \{s : t \in [\![\psi]\!]_{\mathcal{T}} \text{ for some } A, t \text{ s.t. } (s, A, t) \in \delta\} \\
[\![\mathbf{EG}\,\psi]\!]_{\mathcal{T}} &= \mathit{gfp}(\lambda X.[\![\psi]\!]_{\mathcal{T}} \cap \delta^{-1}(X)) \\
[\![\psi_1\ \mathbf{EU}\ \psi_2]\!]_{\mathcal{T}} &= \mathit{lfp}(\lambda X.[\![\psi_2]\!]_{\mathcal{T}} \cup ([\![\psi_1]\!]_{\mathcal{T}} \cap \delta^{-1}(X)))
\end{aligned}$$

where $\mathit{lfp}(f)$ and $\mathit{gfp}(f)$, for a function $f : 2^S \to 2^S$, denote the least and greatest fixed points of f. (These fixed points exist and can be computed effectively because S is finite.) The clauses for the **EG** and **EU** connectives are justified from the recursive characterizations

$$\begin{aligned}
\mathbf{EG}\,\psi &\equiv \psi \wedge \mathbf{EX}\,\mathbf{EG}\,\psi \\
\psi_1\ \mathbf{EU}\ \psi_2 &\equiv \psi_2 \vee (\psi_1 \wedge \mathbf{EX}(\psi_1\ \mathbf{EU}\ \psi_2))
\end{aligned}$$

The clause for **EU** calls for the computation of a least fixed point. Intuitively, this is because ψ_2 has to become true eventually, and thus the unfolding of the fixed point must eventually terminate. On the other hand, the greatest fixed point is required in the computation of $[\![\mathbf{EG}\,\psi]\!]$ because ψ has to hold arbitrarily far down the path. Observe that the least fixed point of the function corresponding to $\mathbf{EG}\,\psi$ is the empty set, whereas the greatest fixed point in the case of **EU** computes $[\![\psi_1\ \mathbf{EW}\ \psi_2]\!]$.

For an implementation, we need to be able to efficiently calculate the *inverse image* function δ^{-1}. Sets $[\![\psi]\!]_{\mathcal{T}}$ that have already been computed can be memorized in order to avoid recomputation of common subformulas. In order to assess the complexity of the algorithm, first note that computation of the fixed points is at most cubic in $|S|$ (if the computation has not stabilized, at least one state is added to or removed from the current approximation per iteration, and every iteration may need to search the entire set of transitions, which may be quadratic in $|S|$). Second, there are as many recursive calls as φ has subformulas, so the overall complexity is linear in the length of φ and cubic in $|S|$.

Clarke, Emerson, and Sistla [29] have proposed a less naive algorithm whose complexity is linear in the product of the sizes of the formula and the model. For formulas $\psi_1\ \mathbf{EU}\ \psi_2$, the idea is to apply backward breadth-first search. For $\mathbf{EG}\,\psi$, first the model is restricted to states satisfying ψ (which have already been computed recursively), and the strongly connected components of this restricted graph are enumerated. The set $[\![\mathbf{EG}\,\psi]\!]_{\mathcal{T}}$ consists of all states of the restricted model from which some SCC can be reached; these states are again found using breadth-first search.

Because fairness assumptions can not be formulated in **CTL**, they must be specified as part of the model, and the model checking algorithm needs to be adapted accordingly. For example, the SMV model checker [97] allows to specify fairness constraints via **CTL** formulas. We define fair variants $\mathbf{EG}_f$ and $\mathbf{EU}_f$ of the **CTL** operators whose semantics is as in definition 5, except that quantifiers are restricted to fair paths, i.e., paths that contain infinitely many states satisfying the constraints. Let us call a state s *fair* iff there is some fair s-path; this is the case iff $\mathcal{T}, s \models \mathbf{EG}_f\ \mathbf{true}$ holds. It is easy to see that $\psi_1\ \mathbf{EU}_f\ \psi_2$ is equivalent to $\psi_1\ \mathbf{EU}\ (\psi_2 \wedge \mathbf{EG}_f\ \mathbf{true})$, hence we need only define an algorithm to compute $[\![\mathbf{EG}_f\,\psi]\!]_{\mathcal{T}}$. The algorithm of Clarke, Emerson, and Sistla can be modified by restricting to those SCCs that for each fairness constraint ζ_i contain some state satisfying ζ_i. The complexity of fair **CTL** model checking is thus still linear in the sizes of the formula and the model. For more information on different kinds of fairness constraints and their associated model checking algorithms see [42,44,81].

A global model checking algorithm for the branching-time fixed point logic μ**TL** can be defined along the same lines. The complexity is then of the order $|\varphi|\cdot|S|^{qd(\varphi)}$ where $qd(\varphi)$ denotes the nesting depth of the fixed point operators in the formula φ. However, Emerson and Lei [44] observed that the computation of fixed points can be optimized for blocks of fixed point operators of the same type, resulting in a complexity of order $|\varphi|\cdot|S|^{ad(\varphi)}$ where $ad(\varphi)$ is the alternation depth of fixed point operators of different type in φ. In particular, the complexity of model checking *alternation-free* μ**TL** is the same as for **CTL** [42,32].

4.3 Symbolic Model Checking

The ability to analyze systems of relevant size using model checking requires efficient data structures to represent objects such as transition systems and sets of system states. Any finite-state system can be encoded using a set $\{b_1, \dots, b_n\}$ of binary variables, just as ordinary data types of programming languages are represented in binary form on a digital computer. Sets of states, for example the set of initial states, can then be represented as propositional formulas over $\{b_1, \dots, b_n\}$, and sets of pairs of states, such as the pairs (s, t) related by δ (for some action) can be represented as propositional formulas over $\{b_1, \dots, b_n, b'_1, \dots, b'_n\}$ where the unprimed variables represent the pre-state s and the primed variables represent the post-state t. The size of the representing formula depends on the structure of the represented set rather than on its size: for example, the empty set and the set of all states are represented by **false** and **true**, both of size 1. For this reason, such representations are often called *symbolic*, and model checking algorithms that work on symbolic representations are called *symbolic model checking* techniques [20,97].

Binary decision diagrams [16,18] (more precisely, reduced ordered BDDs) are a data structure for the symbolic representation of sets that have become very popular for model checking because they offer the following features:

- Every boolean function has a unique, canonical BDD representation. If sharing of BDD nodes is enforced, equality of two functions can be decided in constant time by checking for pointer equality.
- Boolean operations such as negation, conjunction, implication etc. can be implemented with complexity proportional to the product of the inputs.
- Projection (quantification over one or several boolean variables) is easily implemented; its complexity is exponential in the worst case but tends to be well behaved in practice.

BDDs can be understood as compact representations of ordered decision trees. For example, Fig. 9 shows a decision tree for the formula

$$(x_1 \wedge y_1) \vee ((x_1 \vee y_1) \wedge (x_0 \wedge y_0))$$

which is the characteristic function for the carry bit produced by an addition of the two-bit numbers x_1x_0 and y_1y_0. To find the result for a given input, follow the path labelled with the bit values for each of the inputs. The label of the leaf indicates the value of the function. The tree is ordered because the variables appear in the same order along every branch.

The decision tree of Fig. 9 contains many redundancies. For example, the values of y_0 and y_1 are irrelevant if x_0 and x_1 are both 0. Similarly, y_0 is irrelevant in case x_0 is 0 and x_1 is 1. The redundancies can be removed by combining isomorphic subtrees (producing a directed acyclic graph from the tree) and eliminating nodes with identical subtrees. In our example, we obtain the BDD shown on the left-hand side of Fig. 10, where the leaf labelled 0 and all edges leading into it have been deleted for clarity. In an actual implementation, all BDD nodes that have been allocated are kept in a hash table indexed by the top variable and the two sub-BDDs, in order to avoid identical BDDs to be created twice.

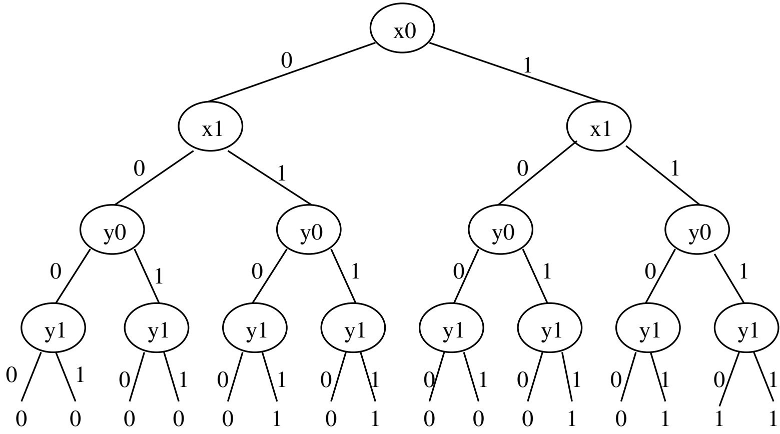

Fig. 9. Ordered decision tree for 2-bit carry.

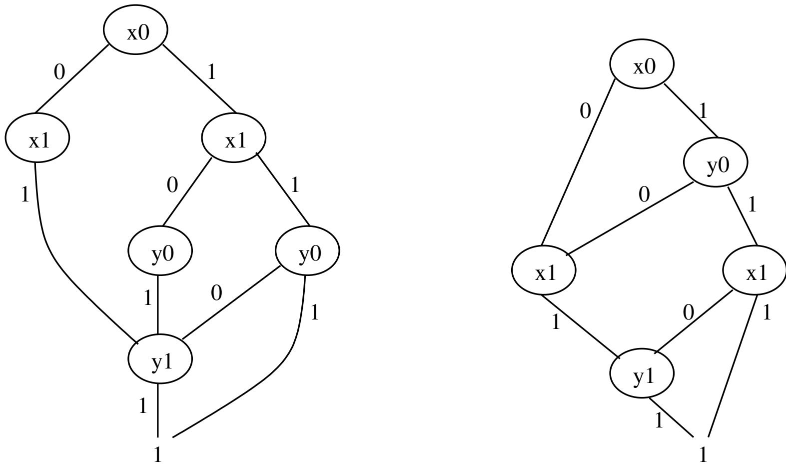

Fig. 10. BDDs for carry from 2-bit adder.

This ensures that two BDDs are functionally equivalent if and only if they are identical.

For a fixed variable ordering the BDD representing any given propositional formula is uniquely determined (and equivalent formulas are represented by the same BDD), but BDD sizes can vary greatly for different variable orderings. For example, the right-hand side of Fig. 10 shows a BDD for the same formula as before, but with the variable ordering x_0, y_0, x_1, y_1. When considering the carry for n-bit addition, the BDD sizes for the variable ordering $x_0, \ldots, x_{n-1}, y_0, \cdots, y_{n-1}$ grow exponentially with n, whereas they grow only linearly for the ordering $x_0, y_0, \ldots, x_{n-1}, y_{n-1}$. It is usually a good heuristic to group "dependent" variables closely together [53,47]. In general, however, the problem of finding an optimal variable ordering is NP-hard [17], and existing BDD libraries offer automatic reordering strategies based on steepest-ascent heuristics [51,10]. There are also functions (such as multiplication) for which no variable ordering can avoid exponential growth. This is also a problem when representing queues, frequently necessary for the analysis of communication protocols, and special-purpose data structures have been suggested [13,57].

Given two BDDs f and g (w.r.t. some fixed variable ordering) the BDD that corresponds to Boolean combinations such as $f \wedge g$, $f \vee g$ etc. can be constructed as follows:

- If f and g are both terminal BDDs (0 or 1), return the terminal BDD for the result of applying the operation.
- Otherwise, let v be the smaller of the variables at the root of f and g. Recursively apply the operation to the sub-BDDs that correspond to v being 0 and 1 (often called the "co-factors" of f and g for variable v). The results l and r correspond to the left- and right-hand branches of the result BDD. If $l = r$, return l, otherwise return a BDD with top variable v and children l and r.

When recursive calls to this "apply" function are memorized in a hash table, the number of subproblems to be solved is at most the number of pairs of nodes in f and g. Assuming perfect hashing, the complexity is therefore linear in the product of the sizes of f and g.

Observing that existential quantification over propositional variables can be computed as

$$(\exists v : f) \quad \equiv \quad f|_{v=0} \vee f|_{v=1}$$

the computation of a BDD corresponding to the quantified formula can be reduced to calculating co-factors and disjunction, and in fact quantification over a set of variables can be performed in a single pass over the BDD.

Symbolic **CTL** *model checking.* The naive **CTL** model checking algorithm of section 4.2 is straightforward to implement based on a BDD representation of the transition system $\mathcal{T}$. It computes BDDs for the sets $[\![\psi]\!]_{\mathcal{T}}$; in particular, the inverse image $\delta^{-1}(X)$ of a set X that is represented as a BDD is computed as the BDD

$$\exists b'_1, \ldots, b'_n : \delta \wedge X'$$

where X' is a copy of X in which all variables have been primed, and $b'_1, \ldots, b'_n$ are all the primed variables. Naive computation of fixed points is also very simple using BDDs because equality of BDDs can be decided in constant time.

It is interesting to compare the complexity of this BDD-based algorithm with that of explicit-state **CTL** model checking: Because the representation of the transition relation using BDDs can be exponentially more succinct than an explicit enumeration, the symbolic algorithm has exponential worst-case complexity in terms of the BDD sizes for the transition relation. First, the number of iterations required for the calculation of the fixed points may be exponential in the number of the input variables, and secondly, the computation of the inverse image may produce BDDs exponential in the size of their inputs. In practice, however, the number of iterations required for stabilization is often quite small, and the inverse image operation is well-behaved. This holds especially for hardware verification problems of “regular” structure and with short data paths. (A precise definition of “regular” is, however, very difficult.) For this class of problems, symbolic model checking has been successfully applied to the analysis of systems with 10^{100} states and more [30]. The main problem is then to find a variable ordering that yields a small representation of the transition system.

Symbolic model checking for other logics. The approach used for symbolic **CTL** model checking extends basically unchanged for propositional μ**TL**. An extension for the richer *relational μ-calculus* [105] has been described by Burch et al. [20] and implemented in the model checker μcke [12].

Symbolic model checking for **PTL** has been considered in [24,112]. The basic idea is to represent each formula in $\mathcal{C}(\varphi)$ by a boolean variable and to define the transition relation and acceptance condition of $\mathcal{B}_{\neg\varphi}$ in terms of these variables rather than constructing the automaton explicitly.

Bounded model checking. Although symbolic model checking has traditionally been associated with BDDs, other representations of boolean functions have also attracted interest. A recent example is the *bounded model checking* technique described in [11]. It relies on the observation that state sequences of fixed length, say k, can be represented using k copies of the variables used to represent a single state. The set of fixed-length sequences that represent terminating or looping runs of a given finite-state transition system $\mathcal{T}$ can therefore be encoded by formulas of (non-temporal) propositional logic, as well as the semantics of **PTL** formulas φ over such sequences. For any given length k, the existence of a state sequence of length k that represents a run of $\mathcal{T}$ satisfying φ can thus be reduced to the satisfiability of a certain propositional formula, which can be decided using efficient algorithms such as Stålmarck’s algorithm [115] or SATO [130]. On the other hand, the *small model property* of **PTL** (which follows from the tableau-based decision procedure discussed in section 3.4) implies that there is a run of $\mathcal{T}$ satisfying φ if and only if there is some such run that can be represented by a sequence of length at most $|S| \cdot 2^{|\varphi|}$. A model checking algorithm is therefore obtained by enumerating all finite executions up to this bound.

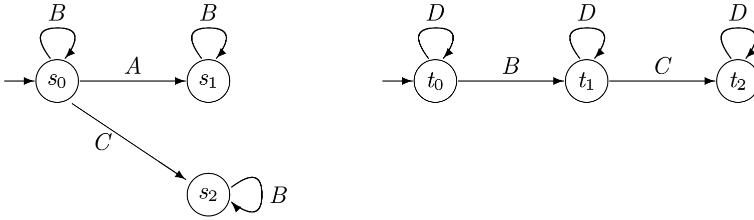

Fig. 11. Transition systems for two processes.

4.4 Partial-Order Reductions

Whereas symbolic model checking derives its power from efficient data structures for the representation and manipulation of large sets of sufficiently regular structure, algorithms based on explicit state enumeration can be improved if only a fraction of the reachable pairs need to be explored. This idea has been applied most successfully in the case of asynchronous systems that are composed of concurrent processes with relatively little interaction. The full transition system has as its runs all possible interleavings of the actions of the individual processes. For many properties, however, the relative order of concurrent actions is irrelevant, and it suffices to consider only a few sequentializations. More sophisticated models than simple interleaving-based representations have been considered in concurrency theory. In particular, *Mazurkiewicz traces* model runs as partial orders of events. Reduction techniques that take advantage of the commutativity of actions are therefore often called *partial-order reductions*, although the analogy to Mazurkiewicz traces is usually rather superficial.

The main problem in the design of a practical algorithm is to detect when two actions commute, given only the "local" knowledge available at a given system state. For example, consider the transition systems for two processes represented in Fig. 11. The left-hand process has a choice between executing actions A and C, whereas the right-hand process must perform action B before action C. Assuming that processes synchronize on common actions, action C is disabled at the global state (s_0, t_0), whereas A, B, and D could be performed. Moreover, all these actions commute at state (s_0, t_0). In particular, A and B can be executed in either order, resulting in the global state (s_1, t_1). However, it would be an error to conclude that only the successors of state (s_0, t_0) with respect to action A need be considered, because action C can then never be taken. The lesson is that actions that are currently disabled must nevertheless be taken into account when constructing a reduced state space.

There is also a danger of prematurely stopping the state exploration because actions are delayed forever along a loop. For an extreme example, consider again the transition systems of Fig. 11 at the global state (s_0, t_0). The local action D of the right-hand process is certainly independent of all other actions. The only successor with respect to that action is again state (s_0, t_0). A naive modification

of the model checking algorithm of Fig. 8 would stop generating further states at that point, which is obviously inadequate.

Partial-order reduction algorithms [123,58,67,48,108] differ in how these problems are dealt with in order to arrive at a reasonably efficient algorithm that is adequate for the given task. The general idea is to approximate the semantic notion of commutativity of actions using syntactic criteria. For example, for a language based on shared variables, two actions of different processes are certainly independent if they do not update the same variable. For message passing communication, send and receive operations over the same channel are independent at those states where the channel is neither empty nor full. Second, the formula φ being analysed must be taken into account: call an action A *visible* for φ if A may change the value of a variable that occurs in φ. Holzmann and Peled [67] define an action to be *safe* if it is not visible and if it is provably independent (with the help of syntactic criteria) of all actions of different processes, even if these actions are currently disabled. The depth-first search algorithm shown in figure 8 can then be modified so that only successor states are considered for some process that can only perform safe actions at the current state. Consideration of the actions of other processes is thus delayed. However, the delayed actions must be considered before a loop is completed. This rather simple heuristic can already lead to substantial savings and carries almost no overhead because the set of safe actions can be determined statically.

More elaborate reduction techniques are considered, for example, in [58,107, 124]. There is always a tradeoff between the potential effectiveness of a reduction method and the overhead involved in computing a sufficient set of actions that must be explored at a given state. Moreover, the effectiveness of partial-order reductions in general depends on the structure of the system: while they are useless for tightly synchronized systems, they may dramatically reduce the numbers of states and transitions explored during model checking for loosely coupled, asynchronous systems.

5 Further Topics

We conclude this survey with brief references to some more advanced topics in the context of model checking. Several of these issues are addressed in detail in other contributions to this volume.

Abstraction. Although techniques such as symbolic model checking and partial-order reduction attempt to battle the infamous state explosion problem, the size of systems that can be analysed using model checking remains relatively limited: even astronomical numbers such as 10^{100} states are generated by systems with a few hundred bits, which is a far cry from realistic hardware or software systems. Model checking must therefore be performed on rather abstract models. It is often advocated that model checking be applied to high-level designs during the early stages of system development because the payoff of finding bugs at that level is high whereas the costs are low. For example, Lilius and Paltor [88] describe a tool for model checking UML state machine diagrams [14],

and model checking of system specifications of similar degrees of abstraction has been considered in [5,52].

When the analysis of big models cannot be avoided, it is rarely necessary to consider them in full detail in order to verify or falsify some given property. This idea can be formalized as an abstraction function (or relation) that induces some abstract system model such that the property holds of the original, "concrete" model if it can be proven for the abstract model. (Dually, abstractions can be set up such that failure of the property in the abstract model implies failure in the concrete model.) In general, the appropriate abstraction relation depends on the application and has to be defined by the user. Abstraction-based approaches are therefore not entirely automatic "push-button" methods in the same way that standard model checking is. Given a concrete model and an abstraction relation, one can either attempt to construct the abstract model using techniques of abstract interpretation [35] or verify the correctness of a proposed abstract model using theorem proving. There is a large body of literature on abstraction techniques, including [26,37,89,90,99].

A particularly attractive way of presenting abstractions is in the form of *predicate abstractions* where predicates of interest at the concrete level are mapped to Boolean variables at the abstract level. The abstract models can then be presented as *verification diagrams*, which are intuitively meaningful to system designers and can be used to (interactively) verify systems of arbitrary complexity [39,92,113,75,22].

For restricted classes of systems, it may be possible to apply fixed abstraction mappings (an example is provided by parameterized systems with simple communication patterns [9]) and thus obtain completely automatic methods. Valmari, in his contribution to this volume, also considers a fixed notion of abstraction that is amenable to full automation.

Symmetry reductions. Informal correctness arguments are often simplified by appealing to some form of symmetry in the system. For examples, components may be replicated in a regular manner, or data may be processed such that permuting individual values does not affect the overall behavior. More formally, a transition system $\mathcal{T}$ is said to be invariant under a permutation π of its states and actions if $(s, A, t) \in \delta$ iff $(\pi(s), \pi(A), \pi(t)) \in \delta$ and $s \in I$ iff $\pi(s) \in I$ holds for all states s, t and all actions A. $\mathcal{T}$ is invariant under a group G of permutations if it is invariant under every permutation in the group. Such a group G induces an equivalence relation on the set of states defined by $s \sim t$ iff $t = \pi(s)$ for some $\pi \in G$. Provided the properties are also insensitive to the permutations in G, one can check the quotient of $\mathcal{T}$ under $\sim$ and obtain a system that can be much smaller [116,23,70,71].

Infinite-state systems. The extension of model checking techniques to infinite-state systems with sufficiently regular state spaces has been an area of active research in recent years [21,49,50,100]. See Esparza's contribution to this volume for more details.

Parameterized systems. One is often interested in the properties of a family of finite-state systems that differ in some parameter such as the number of pro-

cesses. Although individual members of the family can be analyzed using standard model checking techniques, the verification of the entire family requires additional considerations. A natural idea is to perform standard model checking for fixed parameter values and then establish correctness for arbitrary parameter values by induction. In some cases, even the induction step can be justified by model checking. For example, Browne et al. [15] suggest to model check a two-process system, and to establish a bisimulation relation between two-process and n-process systems, ensuring that formulas expressed in a suitable logic cannot distinguish between them. This approach has been extended in [83,127] by using a finite-state process I that acts as an invariant in that the composition of I with another process is again bisimilar to I. Because both I and the individual processes are finite-state, this can be accomplished using (a variation of) standard model checking. Related techniques are described in [46,55].

Compositional verification. The effects of state explosion can be mitigated when the overall verification effort can be subdivided by considering the components of a complex system one at a time. As in the case of abstraction, compositional reasoning normally requires additional input from the user who must specify appropriate properties to be verified of the individual components. The main problem is that components cannot necessarily be expected to function correctly in arbitrary environments, because their design relies on properties of the system the components are expected to be part of. Thus, corresponding assumptions have to be introduced in the statement of the components' correctness properties. Early work on compositional verification [8,109] required components to form a hierarchy with respect to their dependency. In general, however, every component is part of every other component's environment, and circular dependencies among components are to be expected. More recently, different formulations of assumption-commitment specifications have been studied [1,33,96] that can accomodate circular dependencies, based on a form of computational induction. A collection of papers on compositional methods for specification and verification is contained in [40]. Model checking algorithms for modular verification are described, among others, in [59,73,72].

Real-time systems. Whereas temporal logics such as **PTL** and **CTL** only formalize the relative ordering of states and events, many systems require assertions about quantitative aspects of time, and adequate formal models such as timed automata [2] or timed transition systems [62] and logics [4] have been proposed. Algorithms for the reachability and model checking problems for such models include [3,63,64]. In general, the complexity for the verification of real-time and hybrid systems is much higher than for untimed systems, and tools such as Kronos [129], Uppaal [86] or HyTech [61] are restricted to relatively small systems. See the contribution by Larsen and Pettersson to this volume for a more comprehensive presentation of the state of the art in model checking techniques for real-time systems.

References

1. Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.
2. R. Alur. Timed automata. In *Verification of Digital and Hybrid Systems*, NATO ASI Series. Springer-Verlag, 1998.
3. R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *5th Ann. IEEE Symp. on Logics in Computer Science*, pages 414–425. IEEE Press, 1990.
4. R. Alur and T. A. Henzinger. Logics and models of real time: a survey. In *Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 74–106. Springer-Verlag, 1992.
5. R. Alur, G. J. Holzmann, and D. Peled. An analyzer for message sequence charts. In B. Steffen and T. Margaria, editors, *Tools and Constructions for the Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 35–48, Passau, Germany, 1996. Springer-Verlag. See also http://cm.bell-labs.com/cm/cs/what/ubet/index.html.
6. Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. In *38th IEEE Symposium on Foundations of Computer Science*, pages 100–109. IEEE Press, October 1997.
7. A. Anuchitanukul. *Synthesis of Reactive Programs.* PhD thesis, Stanford University, 1995.
8. H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *16th ACM Symp. on Theory of Computing*, pages 51–63. ACM Press, 1984.
9. K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl. Abstracting WS1S systems to verify parameterized networks. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, volume 1785 of *Lecture Notes in Computer Science*, pages 188–203. Springer-Verlag, 2000.
10. J. Bern, C. Meinel, and A. Slobodová. Global rebuilding of BDDs – avoiding the memory requirement maxima. In P. Wolper, editor, *7th Workshop on Computer Aided Verification (CAV'95)*, volume 939 of *Lecture Notes in Computer Science*, pages 4–15. Springer-Verlag, 1995.
11. A. Biere, A. Cimatti, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *36th ACM/IEEE Design Automation Conference (DAC'99)*, 1999.
12. Armin Biere. *Effiziente Modellprüfung des μ-Kalküls mit binären Entscheidungsdiagrammen.* PhD thesis, Univ. Karlsruhe, Germany, 1997.
13. B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In R. Alur and T. Henzinger, editors, *8th Workshop on Computer-Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 1996.
14. G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modelling Language: User Guide.* Addison Wesley, 1999.
15. M. C. Browne, E. M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite-state processes. *Information and Computation*, 81:13–31, 1989.
16. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

17. R. E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Trans. on Computers*, 40(2):205–213, 1991.
18. R. E. Bryant. Symbolic boolean manipulations with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–317, 1992.
19. J. R. Büchi. On a decision method in restricted second-order arithmetics. In *International Congress on Logic, Method and Philosophy of Science*, pages 1–12. Stanford University Press, 1962.
20. J. R. Burch, E. M. Clarke, K. L. McMillan, D. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
21. O. Burkart and J. Esparza. More infinite results. *Electronic Notes in Theoretical Computer Science*, 6, 1997. http://www.elsevier.nl/locate/entcs/volume6.html.
22. Dominique Cansell, Dominique Méry, and Stephan Merz. Predicate diagrams for the verification of reactive systems. In *2nd Intl. Conf. on Integrated Formal Methods (IFM 2000)*, Lecture Notes in Computer Science, Dagstuhl, Germany, November 2000. Springer-Verlag. To appear.
23. E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In C. Courcoubetis, editor, *5th Workshop on Computer-Aided Verification (CAV'93)*, volume 697 of *Lecture Notes in Computer Science*, Elounda, Crete, 1993. Springer-Verlag.
24. E. M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10:47–71, 1997.
25. Edmund M. Clarke and E. Allen Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, N.Y., 1981. Springer-Verlag.
26. Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
27. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, Cambridge, MA, 1999.
28. Edmund M. Clarke and Holger Schlingloff. Model checking. In A. Voronkov, editor, *Handbook of Automated Deduction.* Elsevier, 2000. To appear.
29. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
30. E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the Futurebus+ cache coherence protocol. In D. Agnew, L. Claesen, and R. Camposano, editors, *IFIP Conference on Computer Hardware Description Languages and their Applications*, pages 5–20, Ottawa, Canada, 1993. Elsevier Science Publishers B.V.
31. R. Cleaveland and S. Sims. Generic tools for verifying concurrent systems. *Science of Computer Programming*, 2000. See also http://www.cs.sunysb.edu/~cwb/.
32. R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal μ-calculus. *Formal Methods in System Design*, 2:121–147, 1993.
33. P. Collette. An explanatory presentation of composition rules for assumption-commitment specifications. *Information Processing Letters*, 50(1):31–35, 1994.

34. C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal methods in system design*, 1:275–288, 1992.
35. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press.
36. J.-M. Couvreur. On-the-fly verification of linear temporal logic. In J.M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 – Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 253–271, Toulouse, France, 1999. Springer-Verlag.
37. Dennis Dams, Orna Grumberg, and Rob Gerth. Abstract interpretation of reactive systems: Abstractions preserving $\forall$CTL*, $\exists$CTL* and CTL*. In Ernst-Rüdiger Olderog, editor, *Programming Concepts, Methods, and Calculi (PROCOMET '94)*, pages 561–581, Amsterdam, 1994. North Holland/Elsevier.
38. M. Daniele, F. Giunchiglia, and M. Vardi. Improved automata generation for linear temporal logic. In *Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 249–260, Trento, Italy, 1999. Springer-Verlag.
39. Luca de Alfaro, Zohar Manna, Henny B. Sipma, and Tomás Uribe. Visual verification of reactive systems. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 334–350. Springer-Verlag, 1997.
40. W.-P. de Roever, H. Langmaack, and A. Pnueli, editors. *Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science.* Springer-Verlag, 1998.
41. E. A. Emerson and J. Y. Halpern. “sometimes” and “not never” revisited: on branching time vs. linear time. *Journal of the ACM*, 33:151–178, 1986.
42. E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model checking for fragments of μ-calculus. In C. Courcoubetis, editor, *5th Workshop on Computer-Aided Verification (CAV'93)*, volume 697 of *Lecture Notes in Computer Science.* Springer-Verlag, 1993.
43. E. A. Emerson and C. L. Lei. Modalities for model checking: Branching time strikes back. In *12th Symp. on Principles of Programming Languages (POPL'85)*, New Orleans, 1985. ACM Press.
44. E. A. Emerson and C. L. Lei. Efficient model checking in fragments of the propositional μ-calculus. In *1st Symp. on Logic in Computer Science*, Boston, Mass., 1986. IEEE Press.
45. E. Allen Emerson. *Handbook of theoretical computer science*, chapter Temporal and modal logic, pages 997–1071. Elsevier Science Publishers B.V., 1990.
46. E. Allen Emerson and Kedar S. Namjoshi. Automatic verification of parameterized synchronous systems. In R. Alur and T. Henzinger, editors, *8th International Conference on Computer Aided Verification (CAV'96)*, Lecture Notes in Computer Science. Springer-Verlag, 1996.
47. R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for symbolic model checking. *Distributed Computing*, 6:155–164, 1993.
48. J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23:151–195, 1994.
49. J. Esparza. Decidability of model-checking for infinite-state concurrent systems. *Acta Informatica*, 34:85–107, 1997.

50. J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *14th IEEE Symposium on Logic in Computer Science*, pages 352–359, Trento, Italy, 1999. IEEE Press.
51. E. Felt, G. York, R. Brayton, and A. S. Vincentelli. Dynamic variable reordering for BDD minimization. In *European Design Automation Conference*, pages 130–135, 1993.
52. T. Firley, U. Goltz, M. Huhn, K. Diethers, and T. Gehrke. Timed sequence diagrams and tool-based analysis – a case study. In R. France and B. Rumpe, editors, *2nd Intl. Conference on the Unified Modelling Language (UML'99)*, volume 1723 of *Lecture Notes in Computer Science*, pages 645–660. Springer-Verlag, 1999.
53. H. Fuji, G. Oomoto, and C. Hori. Interleaving based variable ordering methods for binary decision diagrams. In *Intl. Conf. on Computer Aided Design (ICCAD'93).* IEEE Press, 1993.
54. D. Gabbay, I. Hodkinson, and M. Reynolds. *Temporal Logic: Mathematical Foundations and Computational Aspects*, volume 1. Clarendon Press, Oxford, UK, 1994.
55. S. M. German and A. P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39:675–735, 1992.
56. R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification, Testing, and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.
57. P. Godefroid and D. E. Long. Symbolic protocol verification with queue BDDs. In *11th Ann. IEEE Symp. on Logic in Computer Science (LICS'96)*, New Brunswick, NJ, 1996. IEEE Press.
58. P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110(2):305–326, 1994.
59. Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.
60. David Harel and Amir Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume F13 of *NATO ASI Series*, pages 477–498. Springer-Verlag, 1985.
61. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.
62. T. A. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for timed transition systems. *Information and Computation*, 112:273–337, 1994.
63. Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. A space-efficient on-the-fly algorithm for real-time model checking. In *7th International Conference on Concurrency Theory (CONCUR 1996)*, volume 1119 of *Lecture Notes in Computer Science*, pages 514–529. Springer-Verlag, 1996.
64. Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovin. Symbolic model checking for real-time systems. *Information and Computation*, 111:193–244, 1994.
65. Gerard Holzmann. The Spin model checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, may 1997.
66. Gerard Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, November 1998.
67. Gerard Holzmann and Doron Peled. An improvement in formal verification. In *IFIP WG 6.1 Conference on Formal Description Techniques*, pages 197–214, Bern, Switzerland, 1994. Chapman & Hall.

68. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, Reading, Mass., 1979.
69. Michael Huth and Mark D. Ryan. *Logic in Computer Science.* Cambridge University Press, Cambridge, U.K., 2000.
70. C. N. Ip and D. Dill. Better verification through symmetry. In *11th Intl. Symp. on Computer Hardware Description Languages and their Applications*, pages 87–100. North Holland, 1993.
71. C. N. Ip and D. Dill. Verifying systems with replicated components in Murphi. In *Intl. Conference on Computer-Aided Verification (CAV'96)*, Lecture Notes in Computer Science. Springer-Verlag, 1996.
72. Bernhard Josko. Verifying the correctness of AADL modules using model checking. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 386–400. Springer-Verlag, Berlin, 1989.
73. Bernhard Josko. *Modular Specification and Verification of Reactive Systems.* PhD thesis, Univ. Oldenburg, Fachbereich Informatik, April 1993.
74. H. W. Kamp. *Tense Logic and the Theory of Linear Order.* PhD thesis, Univ. of California at Los Angeles, 1968.
75. Yonit Kesten and Amir Pnueli. Verifying liveness by augmented abstraction. In *Annual Conference of the European Association for Computer Science Logic (CSL'99)*, Lecture Notes in Computer Science, Madrid, 1999. Springer-Verlag.
76. Nils Klarlund. Mona & Fido: The logic-automaton connection in practice. In *Computer Science Logic, CSL '97*, volume 1414 of *LNCS*, pages 311–326, Aarhus, Denmark, 1998.
77. Dexter Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
78. Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.
79. Fred Kröger. *Temporal Logic of Programs*, volume 8 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, Berlin, 1987.
80. O. Kupferman, M. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In *6th Intl. Conf. on Computer-Aided Verification (CAV'94)*, Lecture Notes in Computer Science. Springer-Verlag, 1994. Full version (1999) available at http://www.cs.rice.edu/˜vardi/papers/.
81. O. Kupferman and M. Y. Vardi. Verification of fair transition systems. In R. Alur and T. Henzinger, editors, *8th Workshop on Computer-Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 372–382. Springer-Verlag, 1996.
82. Orna Kupferman and Moshe Y. Vardi. Weak alternating automata are not so weak. In *5th Israeli Symposium on Theory of Computing and Systems*, pages 147–158. IEEE Press, 1997.
83. R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. In *8th Ann. ACM Symp. on Principles of Distributed Computing.* ACM Press, 1989.
84. Leslie Lamport. 'sometime' is sometimes 'not never'. In *Proc. 7th Ann. Symp. on Princ. of Prog. Lang. (POPL'80)*, pages 174–185. ACM SIGACT-SIGPLAN, January 1980.

85. M. Lange, M. Leucker, T. Noll, and S. Tobies. Truth – a verification platform for concurrent systems. In *Tool Support for System Specification, Development, and Verification*, Advances in Computing Science. Springer-Verlag Wien New York, 1999.
86. K. Larsen, P. Petterson, and W. Yi. Uppaal in a nutshell. *Software Tools for Technology Transfer*, 1, 1997.
87. Orna Lichtenstein, Amir Pnueli, and Lenore Zuck. The glory of the past. In Rohit Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218, Berlin, June 1985. Springer-Verlag.
88. J. Lilius and I. P. Paltor. Formalising UML state machines for model checking. In R. France and B. Rumpe, editors, *UML'99 – Beyond the Standard*, volume 1723 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
89. Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:11–44, 1995. A preliminary version appeared as Spectre technical report RTC40, Grenoble, France, 1993.
90. D. E. Long. *Model checking, Abstraction and Compositional Verification*. PhD thesis, CMU School of Computer Science, 1993. CMU-CS-93-178.
91. Gavin Lowe. Breaking and fixing the Needham-Schroeder public key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996.
92. Z. Manna, A. Browne, H.B. Sipma, and T.E. Uribe. Visual abstractions for temporal verification. In A. Haeberer, editor, *AMAST'98*, volume 1548 of *Lecture Notes in Computer Science*, pages 28–41. Springer-Verlag, 1998.
93. Zohar Manna and Amir Pnueli. A hierarchy of temporal properties. In *9th. ACM Symposium on Principles of Distributed Computing*, pages 377–408. ACM, 1990.
94. Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems—Specification*. Springer-Verlag, New York, 1992.
95. Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems—Safety properties*. Springer-Verlag, New York, 1995.
96. Kenneth L. McMillan. A compositional rule for hardware design refinement. In O. Grumberg, editor, *9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 24–35, Haifa, Israel, 1997. Springer-Verlag.
97. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
98. R. McNaughton and S. Papert. *Counter-Free Automata*. MIT Press, Cambridge, Mass., 1971.
99. Stephan Merz. Rules for abstraction. In R. K. Shyamasundar and K. Ueda, editors, *Advances in Computing Science—ASIAN'97*, volume 1345 of *Lecture Notes in Computer Science*, pages 32–45, Kathmandu, Nepal, December 1997. Springer-Verlag.
100. Faron Moller. Infinite results. In U. Montanari and V. Sassone, editors, *7th International Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 195–216, Pisa, Italy, 1996. Springer-Verlag.
101. D. E. Muller. Infinite sequences and finite machines. In *Switching Circuit Theory and Logical Design: Fourth Annual Symposium*, pages 3–16, New York, 1963. IEEE Press.

102. D. E. Muller, A. Saoudi, and P. E. Schupp. Alternating automata, the weak monadic theory of the tree and its complexity. In *13th ICALP*, volume 226 of *Lecture Notes in Computer Science*, pages 275–283. Springer-Verlag, 1986.
103. D.E. Muller, A. Saoudi, and P.E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *3rd IEEE Symposium on Logic in Computer Science*, pages 422–427. IEEE Press, 1988.
104. Roger Needham and Michael Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.
105. D. M. Park. Finiteness is mu-ineffable. Theory of Computation Report 3, University of Warwick, 1974.
106. Lawrence C. Paulson. Proving security protocols correct. In *14th IEEE Symposium on Logic in Computer Science*, pages 370–383, Trento, Italy, 1999. IEEE Press.
107. D. Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8(1):39–64, 1996.
108. W. Penczek, R. Gerth, and R. Kuiper. Partial order reductions preserving simulations. Submitted for publication, 1999.
109. Amir Pnueli. In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume F 13 of *ASI*, pages 123–144. Springer-Verlag, Berlin, 1985.
110. M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, 1969.
111. Shmuel Safra. On the complexity of ω-automata. In *29th IEEE Symposium on Foundations of Computer Science*, pages 319–327. IEEE Press, 1988.
112. Klaus Schneider. Yet another look at LTL model checking. In *IFIP Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'99)*, Lecture Notes in Computer Science, Bad Herrenalb, Germany, 1999.
113. H.B. Sipma, T.E. Uribe, and Z. Manna. Deductive model checking. In *8th International Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 208–219, New Brunswick, N.J., 1996. Springer-Verlag.
114. A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *Journal of the ACM*, 32:733–749, 1985.
115. G. Stålmarck. A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula. Swedish Patent No. 467076 (1992), US Patent No. 5 276 897 (1994), European Patent No. 0404 454 (1995).
116. P. H. Starke. Reachability analysis of Petri nets using symmetries. *Syst. Anal. Model. Simul.*, 8:293–303, 1991.
117. Colin Stirling. *Handbook of Logic in Computer Science*, volume 2, chapter Modal and temporal logics, pages 477–563. Oxford Science Publications, Clarendon Press, Oxford, 1992.
118. Colin Stirling. Bisimulation, model checking, and other games. Mathfit instructional meeting on games and computation, 1997. Available at http://www.dcs.ed.ac.uk/home/cps/.
119. R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1:146–160, 1972.

120. Wolfgang Thomas. Automata on infinite objects. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, volume B: Formal Models and Semantics*, pages 133–194. Elsevier, Amsterdam, 1990.
121. Wolfgang Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Language Theory*, volume III, pages 389–455. Springer-Verlag, New York, 1997.
122. Wolfgang Thomas. Complementation of Büchi automata revisited. In J. Karhumäki, editor, *Jewels are Forever, Contributions on Theoretical Computer Science in Honor of Arto Salomaa*, pages 109–122. Springer-Verlag, 2000.
123. A. Valmari. A stubborn attack on state explosion. In *2nd International Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165, Rutgers, June 1990. Springer-Verlag.
124. A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.
125. Moshe Y. Vardi. Alternating automata and program verification. In *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 471–485. Springer-Verlag, 1995.
126. M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.
127. P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In J. Sifakis, editor, *Intl. Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.
128. Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–93, 1983.
129. S. Yovine. Kronos: A verification tool for real-time systems. *Software Tools for Technology Transfer*, 1, 1997.
130. H. Zhang. Sato: An efficient propositional prover. In *Intl. Conf. on Automated Deduction (CADE'97)*, number 1249 in Lecture Notes in Computer Science, pages 272–275. Springer-Verlag, 1997.