Origins of Model Checking

Joy Okonkwo (joy@aims.ac.za)
African Institute for Mathematical Sciences (AIMS)

Supervised by Dr. Jaco Geldenhuys and Dr. Cornelia Inggs
University of Stellenbosch, South Africa

June 8, 2007

Abstract

Computer systems are frequently used in our day to day activities. The failure of these systems in some application areas could result in loss of lives and resources, thus their correctness is of paramount importance. In this essay, we discuss the concept of model checking – an automatic verification technique for software and hardware systems to establish their correctness with respect to some desirable properties. We describe and compare two model checking algorithms that have been developed, use an example to describe how both model checking algorithms can be used to verify that a system satisfies its correctness properties, and lastly enumerate the strengths and limitations of these algorithms.

Contents

List of Figures

1. Introduction

The use of computers has intensely pervaded our day to day activities. In some application areas, failures could adversely affect lives and resources. Examples of such applications include the control of industrial machinery (e.g., nuclear power plants), transportation equipment (e.g., aircraft), and medical equipment. These systems are said to be reactive, meaning that they continuously interact with their environment by constantly producing output in response to continuous input from the users. As such, they are expected to produce the correct result, and on time as well. Since the failures of these systems cannot be tolerated, their correctness with respect to some desirable properties, becomes an extremely important issue.

However, verification of such systems to ensure that they function correctly is a complex problem because these systems are often concurrent[1] in nature. Several methods have been proposed to accomplish this task, but one that has obtained significant success is model checking.

The origins of model checking date back to the early 1980s, when Clarke and Emerson [CES83] as well as Queille and Sifakis [QS82], independently introduced a new algorithmic approach for the verification of computer systems. The concept of their approach is to check that a logical specification of a system (i.e., its correctness properties) is satisfied by a model of the system's behaviour represented by a state transition system. We will explain in detail what a state transition system is in section 2.2.

A model checker automatically verifies that a formal specification of a system has certain desirable properties. The verification process establishes the correctness of the system by comparing a specification of the intended behaviour of the system, which is a set of correctness properties (usually expressed in temporal logic), with a specification of the actual behaviour of the system which can be represented as a state transition system also known as a Kripke structure, or as an interpreted Petri net (details given in Chapter 2). This is done by using the model checking algorithm to search the state space of the state transition system to determine that all its reachable states satisfy each of the correctness properties expected of the system. A model checker can be used in different ways. A model can be developed first, and an implementation derived from it after verification, or a model can be derived from an existing implementation and then verified. In the two model checking algorithms discussed in this essay, the model is developed and verified before implementation is done.

The goal of this essay is to explore the concept of model checking, describe and compare two model checking approaches that have been developed, and describe how they can be used to verify that a system satisfies its correctness properties. The two model checking approaches are:

1. E. M. Clarke and E. A. Emerson's approach, where the state graph of the concurrent system is viewed as a finite Kripke structure/state transition system and its correctness properties are expressed in computation tree logic (CTL).

2. J. -P. Queille and J. Sifakis's approach, where an algorithmic description of the system written in a high-level language is automatically translated into an interpreted petri net

[1] Their processes are executed simultaneously.

(IPN) and its correctness properties are expressed in a branching time logic.

1.1 Essay Outline

Chapter 2 gives an overview of model checking; the syntax and semantics of temporal logic are explained, the theory of finite state transition systems is explained, the theory of Petri nets is explained, and a brief description of the exhaustive search method is given. Chapter 3 describes and compares two approaches of model checking as developed by E. M. Clarke and E. A. Emerson, and J. -P. Queille and J. Sifakis. Chapter 4 compares the two approaches using a simple example, and enumerates their limitations and strengths. Chapter 5 gives concluding remarks to the topic and possible related areas for further research.

2. Overview of Model Checking

Model checking is an automatic technique for verifying finite-state concurrent systems. Examples include circuit designs for hardware systems and communication protocols (which are standard rules that govern the interaction between systems in a network). The correctness properties of these systems are expressed in propositional temporal logic while the system itself is specified/-modeled as a state transition system. An efficient search procedure is used to automatically determine if the correctness properties are satisfied by the state transition system [EMCP00, CGL94]. Model checking was originally developed in 1981 by Clarke and Emerson [CES83]. Queille and Sifakis [QS82] independently discovered a similar verification technique shortly thereafter. These two approaches will be clearly discussed in the next chapter. Over the past few years, model checking has become a successful tool in detecting subtle errors in the design phase of safety-critical systems that might never have been detected using conventional simulation and testing techniques [Gei00].

One of the reasons for its success is that a model checker confirms whether a model satisfies its correctness properties or not. Furthermore, if the correctness properties are not satisfied, it gives a counterexample execution that shows why the specified correctness properties are not satisfied [EMCP00]. The counterexamples are thus very important in finding subtle errors in a complex reactive system, because they clearly show the designers or the programmers where the errors in the system are. This interesting aspect of model checking makes it very useful, because beyond its application for the verification of systems, it can also be used to find bugs in systems [CDW04].

Another reason for its success is that it is automatic and thus, has several important advantages over mechanical theorem provers or proof checkers[1] [CGL94]. Clarke and Emerson pointed out in their work that the task of proof construction is quite tedious [CES83]. They also state that mechanical theorem provers fail to be of much help due to the complexity of even the simplest logics, and can thus be replaced by model checking. Another advantage of model checking is that the procedure is fast, producing results in a matter of minutes or even seconds. Partial specifications can also be checked. So in the case of circuit designs for example, it is actually unnecessary to specify the circuit completely before information can be obtained regarding its correctness [CB98].

The main disadvantage of model checking is the state space explosion problem. This problem occurs in systems with many components that can interact with each other or with data structures that can assume many different values. In such cases, the number of global states can be exponential in the number of components and data structures, thereby making it impossible to represent the system by using state transition systems. Researchers and practitioners have made considerable progress on this problem over the last twenty years by using techniques such as "on-the-fly verification". One of the most significant breakthroughs was made in the fall of 1987 by McMillan [CB98]. He realised that using an explicit representation for transition relations severely limited the size of the circuits and protocols that could be verified. He argued that larger systems could be handled if transition relations were represented implicitly with ordered binary

[1] another technique for the verification of concurrent systems

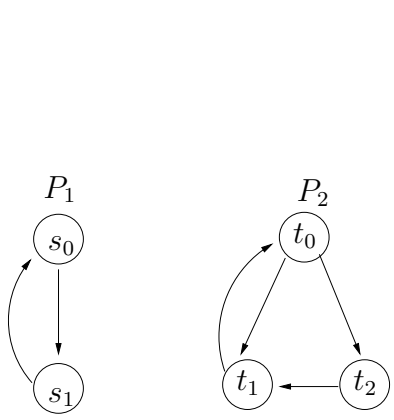
decision diagrams (OBDDS) [Bry86]. By using the original model checking algorithm developed with the new representation for transition relations, he was able to verify systems with more than 10^{20} states. Other successful techniques for the state explosion problem include partial order reductions [P.G94, Pel93, Val92], symmetry reduction [CFJ93, ES93, ID93], and bitstate hashing [Hol95].

In the rest of this chapter, we will discuss some of the concepts mentioned already in more detail to give a clearer understanding of model checking.

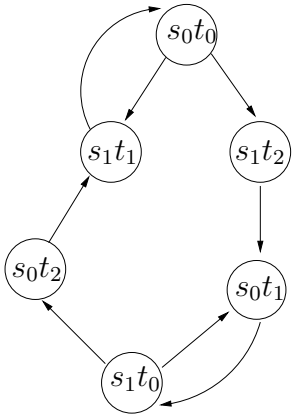
2.1 Finite State Transition Systems

Computer systems typically have more than one process which they execute concurrently. A process in this context is the execution of a procedure or function, or program which forms part of the system. The state of a process at any point in time consists of the values of explicit variables declared by the programmer and implicit variables such as the program counter and contents of data/address registers [MK99]. As a process executes, it changes its state by executing statements. Each statement consists of a sequence of one or more actions that bring about these state changes. The order in which these actions are allowed to occur, which indicates the possible execution paths of the program, can be represented by a state transition system.

For the modeling of concurrent systems using a state transition system, actions from the different processes that make up the system are arbitrarily interleaved to represent the possible execution paths of these systems. Figure 2.1 depicts the state transition system of two concurrent processes denoted as P_1 and P_2, as well as the state transition system of each individual process.



(a) State transition system for processes P_1 and P_2



(b) The resultant state transition system for the two processes

Figure 2.1: State transition system for a system with two processes P_1 and P_2

A simple illustration of the state transition system of a traffic light, a single process system, is shown in Figure 2.2

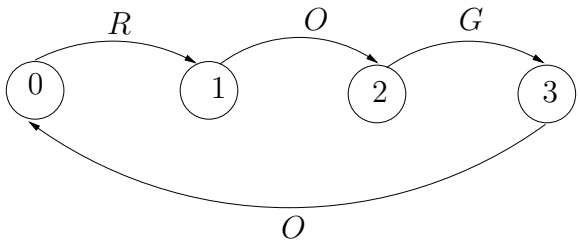


Figure 2.2: State transition system for a traffic light system

In Figure 2.2 R represents a red light, O represents an orange light, and G represents a green light. We can see that the system transits from one state to another depending on its current state. The following sequence of actions occur at the execution of a traffic light:

red $\longrightarrow$ orange $\longrightarrow$ green$\longrightarrow$ orange$\longrightarrow$ red $\longrightarrow$ orange$\longrightarrow$ green.....

This clearly shows that although the representation of the system is finite, the behaviour described need not be finite. The sequence of actions produced by an execution of a process is known as a path (or trace).

A process can also have more than one execution path as illustrated in Figure 2.3, which depicts the state transition system of a drink dispensing machine that dispenses coffee when its red button is pressed and iced tea when its blue button is pressed. This is also a program with a single process.

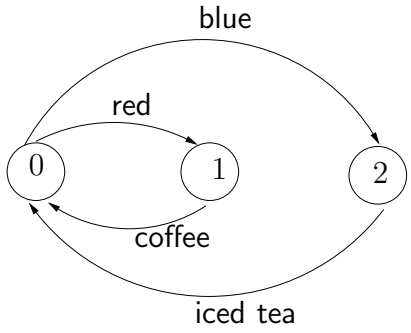


Figure 2.3: State transition system of a drink dispensing machine

The system has an infinite number of different infinite executions. If R means red, and C means coffee, and B means blue, and I means iced tea, the following are some of these executions:

- RCBIBIBIBIBIBIBIBI...
- RCRCBIBIBIBIBIBIBI...
- RCRCRCBIBIBIBIBIBI...
- RCRCRCRCBIBIBIBIBI...

So, the system can start with a finite number of RC and then execute BI forever.

Finite state transition systems as formalism for describing the behaviour of computer programs and systems, is a suitable input for a model checker [CE82]. Informally, it is basically a graph with the reachable states of the system as nodes and state transitions of the system as edges. Each state of the system is labelled with the properties that hold in it (known as atomic propositions AP).

Formally, a state transition system also known as a Kripke structure is defined as a 4-tuple $K = (S, s_0, R, L)$, where:

1. S is a finite set of states.
2. $s_0 \in S$ is an initial state.
3. $R \subseteq S \times S$ is a transition relation, for which it holds that $\forall s \in S : \exists s' \in S : (s, s') \in R$.
4. $L : S \rightarrow 2^{AP}$ is a function which labels each state with the atomic propositions which hold in that state.

2.2 Petri Nets

A Petri net is a graphical modeling tool, used for describing and studying the behaviour of systems that are concurrent in nature [GR82]. It consists of places, transitions, and directed arcs. The directed arcs connect places and transitions only. An arc that leads from a place to a transition is called an *input arc*; and the place called an *input place* of the transition, while an arc leading from a transition to a place is called an *output arc*; and the place called an *output place* of the transition.

The execution of a Petri net involves the movement of markers called tokens through the net [Pet81]. Tokens reside in places and they are represented diagrammatically with black dots. They are used in Petri nets to simulate the dynamic and concurrent activity of systems. A distribution of tokens over the places of the Petri net is called a marking. Thus, a Petri net containing tokens is called a marked Petri net. A marking is a vector $M = (m_1, m_2, ...m_n)$, where n is the number of places in a Petri net. Each component m_i of the vector M is a non-negative integer which represents the number of tokens in place p_i, where $i = 1, 2, ...n$. The initial distribution of tokens is known as the initial marking of a Petri net (which represents the initial state).

A change of state of the system is brought about by the *firing* of a Petri net transition (i.e., the action of a transition on input tokens of the Petri net), thus changing the initial marking of the Petri net. A transition can only fire if it is enabled, and it is enabled if every input place of the transition has at least one token. The firing of a transition involves the removal of a token from each of the input places of the transition, and the placement of tokens into each of its output places. The firing process of a transition changes the state of the system. This state change is defined by the next state function δ. It produces a new marking from a given marking and has the form $M' = \delta(M, t_j)$, where M is the marking before the transition, and M' is the new marking (i.e., the new state) after transition t_j has fired.

Figure 2.4 shows a Petri net with an initial marking $M = (2, 1, 0, 2)$. At the firing of the first transition, the system changes to another state with marking $M' = (1, 2, 0, 2)$. The movements of these tokens from the input places to the output places as a result of the firing of the transitions, represent the dynamic properties of the Petri net which in turn simulates the dynamic behaviour of the system under discussion.

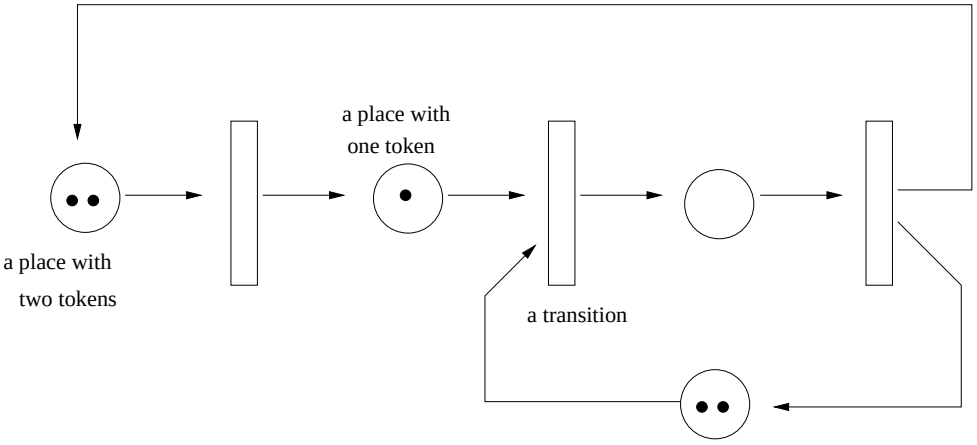


Figure 2.4: A Petri net

A Petri net is formally defined as a 5-tuple (S, T, F, M_0, W), where

- S is a set of places.
- T is a set of transitions.
- F is a set of arcs known as a flow relation. The set F is subject to the constraint that no arc may connect two places or two transitions. This is formally stated as $F \subseteq (S \times T) \cup (T \times S)$.
- $M_0 : S \to \mathbb{N}$ is an initial marking, where for each place $s \in S$, there are $n \in \mathbb{N}$ tokens.
- $W : F \to \mathbb{N}^+$ is a set of arc weights, which assigns to each arc $f \in F$ some $n \in \mathbb{N}^+$ denoting how many tokens are consumed from a place by a transition, or how many tokens are produced by a transition and put into each place.

An interpreted petri net (IPN) is a Petri net with the following additional attributes:

- A vector of variables X,
- A mapping which associates each transition of the Petri net with a guarded command (condition), e.g. $c_i \to a_i$, where c_i is a condition on X and a_i is a vectorial assignment.

Also an it functions the same way as Petri nets, but with the following additional rules:

- A transition can fire only when its associated condition is true

- When a transition fires, its associated action is executed

It is important to note that a Petri net can easily be translated to a finite state transition system and vice versa.

2.3 Temporal Logic

Logical formalisms for reasoning about time and how events evolve with time appear in several fields: Physics, Philosophy, Linguistics, Computer Science, etc. [Sch02]. In Computer science, temporal logics are used in automated reasoning, in planning, in semantics of programming languages, in artificial intelligence, etc. [Lut06, Woo90]. An area where temporal logic has been most successful is in the specification and verification of programs and systems. Temporal logics are used to reason about the behaviour of these programs and systems as they evolve over time. It is a convenient formal language for expressing and reasoning about the behavioural properties of concurrent programs [Pnu97, Pnu79].

Temporal logic formulae usually express safety properties, liveness properties, and fairness properties of these systems [Woo90]. These properties will be explained in detail later in this section. Temporal logics can be divided into two basic types: linear time temporal logic e.g., linear temporal logic (LTL) and branching time temporal logic e.g., computation tree logic (CTL) [CES83, Pnu97]. In linear time logics, all states are linearly ordered from past to future and there is only one possible future. In other words, the future is determined. In contrast, the future is not determined in branching time logics, because any given state may have several distinct immediate successor states. The syntax and semantics of these two temporal logics are given in the preceding subsections.

2.3.1 Linear Temporal Logic: LTL

Linear time temporal logic is a formalism to express properties of (linear) infinite executions of a system [Gei00]. LTL formulae can be used to express atomic propositions[2], and larger formulae can be built on these atomic propositions relating them over time. For example, given some properties p and q, we can express things like: 'every state in which p is true is eventually followed by a state in which q is true' or 'p is true in every state until q is true'.

There are two kinds of operators in temporal logic with which these properties are expressed: Logical operators and Modal operators [Pnu97]. Logical operators are usually truth–functional operators like: **not** $(\neg)$, **disjunction** or **or** $(\vee)$, **conjunction** or **and** $(\wedge)$, **implies** $(\rightarrow)$, and **equivalence** $(\equiv)$, while modal operators are usually non truth–functional operators like: X (for next), G (for globally), F (for finally), and U (for until). The syntax and semantics of LTL as adopted from [Gei00, Lut06, Pnu79] is given as:

[2] A term given to the boolean properties that can be observed in every state of the system.

Syntax and Semantics of LTL

The formal syntax for LTL is given below:

1. each atomic proposition $p \in AP$ is an LTL formula
2. if f_1 and f_2 are LTL formulae, then so are $\neg f_1$, $f_1 \wedge f_2$, $X f_1$, and $f_1 \cup f_2$.

A proposition p holds for the state sequence $\sigma = \sigma_0, \sigma_1, \ldots$ (which is an infinite execution of the system or an execution path of the system), if it holds in the first state σ_0. $X f_1$ means that the formula f_1 holds in the state sequence starting from the next state of σ, i.e., σ_1. The formula $f_1 \cup f_2$ expresses the fact that formula f_1 holds in every state up to the point where f_2 holds, i.e., there must be some point in time where f_2 holds and f_1 must hold at every state upto that point.

The notion that an LTL formula f holds for a state sequence or execution path σ (i.e. σ is a model of f) is denoted as $\sigma \models f$. The semantics of LTL formulae are defined formally on the relation $\models$ as:

Given an execution path σ and LTL formulae f_1 and f_2,

$\sigma \models p$	iff	$p \in L(\sigma_0)$
$\sigma \models \neg f$	iff	not $\sigma \models f$
$\sigma \models f_1 \wedge f_2$	iff	$\sigma \models f_1$ and $\sigma \models f_2$
$\sigma \models X f_1$	iff	$\sigma_1 \models f_1$
$\sigma \models G f_1$	iff	for all i, $\sigma_i \models f_1$
$\sigma \models F f_1$	iff	there exists an i such that $\sigma_i \models f_1$
$\sigma \models f_1 \cup f_2$	iff	there exists an i such that $\sigma_i \models f_2$ and for all $j < i, \sigma_j \models f_1$

Specifying Temporal Properties in LTL

Given a Kripke structure K representing a reactive system and an LTL formula f representing a temporal property, K satisfies f if $M, 0 \models f$ for all paths M of K. This can be written as $K \models f$.

Typical properties of reactive systems that need to be checked during verification are safety properties, liveness properties, and fairness properties [Lut06].

1. Safety property: A safety property asserts that 'nothing bad happens' in the system. In other words, the system never reaches a bad state. Two important safety properties are mutual exclusion (no two processes use the same resource at the same time) and freedom from deadlock (there is always atleast one process that is enabled). The latter can be expressed in LTL as: $G(enabled_1 \vee ... \vee enabled_k)$, where $enabled_i$ is true if process i has an action that can be executed for $1 \leqslant i \leqslant k$.

2. Liveness property: This property asserts that 'something good will eventually happen'. In other words there is progress in the system. This includes guaranteed accessibility (once the system is in a current state, it will eventually go to the next state without any obstruction), and responsiveness (if a request is issued, it will eventually be granted). This last property can be formally represented in LTL as $G(req \rightarrow Fgrant)$.

3. Fairness property: This includes unconditional fairness (every process is executed infinitely often), and strong fairness (every process enabled infinitely often is executed infinitely often).

Some typical LTL formulae are:

- $G(enabled_1 \vee ... \vee enabled_k)$: This means that at every state, at least one process is enabled.
- $G(req \rightarrow Fgrant)$: This means that on every execution path, once a request is issued, it will eventually be granted.
- $G(FRestart)$: This means that from every state it is possible to get to a state where Restart is true.

2.3.2 Computation Tree Logic: CTL

This is a branching time temporal logic in the sense that its model of time is a tree-like structure. In CTL, the future is not deterministic because there are different possible paths in the future, and any of these paths might be the actual path realised.

Syntax and Semantics of CTL

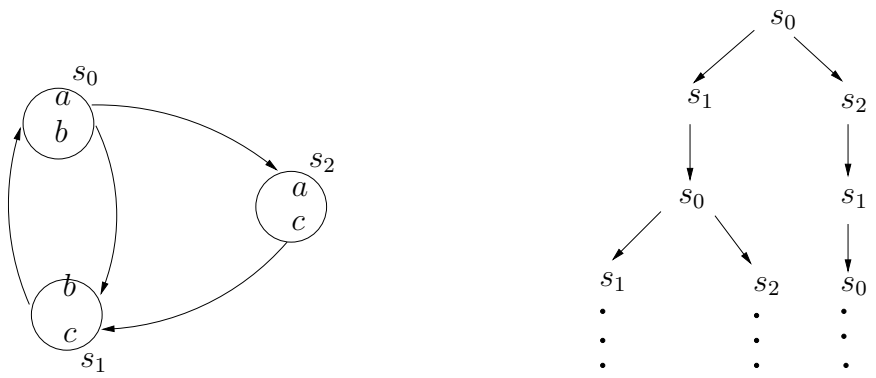
The syntax of CTL is based on the propositional operators $\wedge$ ("and"), $\vee$ ("or"), $\neg$ ("not") and $\rightarrow$ ("implies"), along with branching time operators. A branching time operator consists of a path quantifier A ("for all computational paths"), or E ("for some computational paths"), followed by one of the following temporal operators: G ("globally"), F ("finally"), X ("next") or U ("until"). The combination of the path quantifiers and the temporal operators gives rise to the following eight basic CTL operators: AX, EX, AG, EG, AF, EF, AU, EU.

The formal syntax of CTL as stated by Emerson [CES86], is given as follows:

1. every atomic proposition $p \in AP$ is a CTL formula.
2. if f_1 and f_2 are CTL formulae, then so are $\neg f_1$, $f_1 \wedge f_2$, $AX f_1$, $EX f_1$, $A[f_1 \cup f_2]$, $E[f_1 \cup f_2]$.

The semantics of CTL is defined with respect to a Kripke structure, $K = (S, s_0, R, P)$. For any structure $K = (S, s_0, R, P)$, and state $s_0 \in S$, there is an infinite computation tree with root labeled s_0 such that $s \rightarrow t$ is an arc in the tree iff $(s, t) \in R$. It is important to note that

an infinite sequence of states $(s_0, s_1, s_2, ...)$ such that $\forall i[(s_i, s_{i+1}) \in R]$ is called a *path*. The execution paths of a Kripke structure is in the form of a tree structure called a computation tree, as seen in Figure 2.5, showing a Kripke structure and its corresponding computation tree.



(a) Kripke structure (b) Computation tree

Figure 2.5: A Kripke structure and its corresponding computation tree

The semantics of CTL on the relation $\models$ is defined below; where the notation $K, s_0 \models f$ means that formula f holds at state s_0 in structure K.

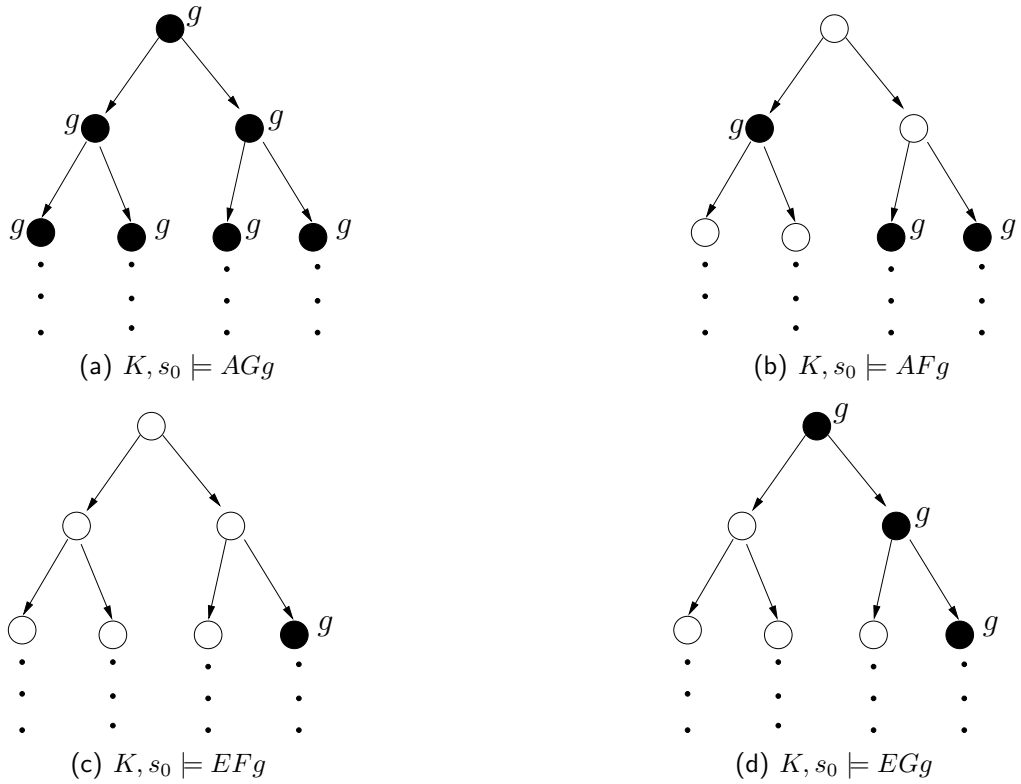
$s_0 \models p$	iff	$p \in P(s_0)$
$s_0 \models \neg f$	iff	not$(s_0 \models f)$
$s_0 \models f_1 \wedge f_2$	iff	$s_0 \models f_1$ and $s_0 \models f_2$
$s_0 \models AXf_1$	iff	for all states t such that $(s_0, t) \in R, t \models f_1$
$s_0 \models EXf_1$	iff	for some state t such that $(s_0, t) \in R, t \models f_1$
$s_0 \models A[f_1 \cup f_2]$	iff	for all paths $(s_0, s_1, ...), \exists i[i \geq 0 \wedge s_i \models f_2 \wedge \forall j[0 \leq j < i \rightarrow s_j \models f_1]]$
$s_0 \models E[f_1 \cup f_2]$	iff	for some path $(s_0, s_1, ...), \exists i[i \geq 0 \wedge s_i \models f_2 \wedge \forall j[0 \leq j < i \rightarrow s_j \models f_1]]$

Each of the eight basic CTL operators can be expressed in terms of AX EX, AU, EU. For example,

- $AGf = A[f \cup false]$
- $AXf = \neg EX(\neg f)$
- $EFf = E[true \cup f]$
- $AFf = \neg EG(\neg f)$
- $A[f_1 \cup f_2] = \neg E[\neg f_2 \cup \neg f_1 \wedge \neg f_2] \wedge \neg EG\neg f_2)$

Figure 2.6 shows computation trees illustrating the four most widely used CTL operators with s_0 as the root node.

Some typical CTL Formulae are:



(a) $K, s_0 \models AGg$ (b) $K, s_0 \models AFg$

(c) $K, s_0 \models EFg$ (d) $K, s_0 \models EGg$

Figure 2.6: Computation tree of 4 most widely used CTL operators

- $EF(Started \wedge \neg Ready)$: This means that it is possible to get to a state where Started holds but Ready does not hold.
- $AG(Req \Rightarrow AFAck)$: This means that in all paths at every state, once a request occurs, then it will eventually be acknowledged.
- $AG(AFDeviceEnabled)$: This means that DeviceEnabled holds infinitely often on every computation path.
- $AG(EFRestart)$: This means that from every state it is possible to get to a state where Restart is true.

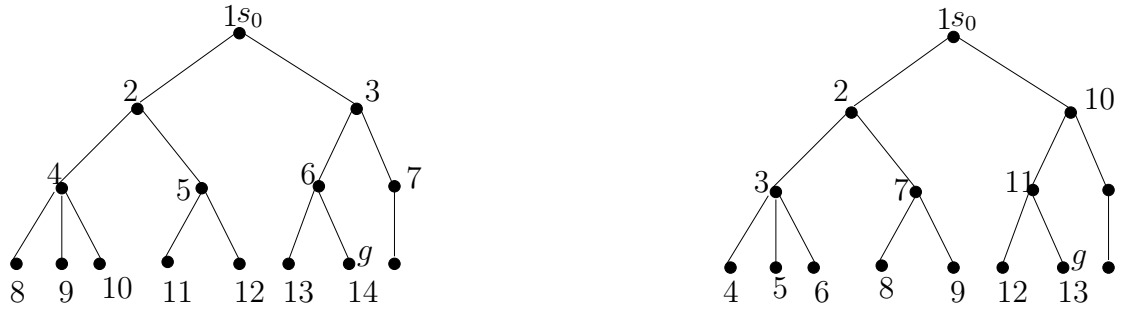
2.4 Exhaustive Search Methods

Exhaustive search methods are algorithms used to search the entire reachable state space of a system starting from the initial state. The state explosion problem explained earlier in this chapter, makes the exhaustive search methods very hard in practice with limited memory and time resources. This makes it only practical in small systems. If there are limited memory space and time resources available, another alternative search method known as non-exhaustive search method is sometimes employed. In this method, the entire reachable state space of the system is searched as far as the available resources allow, giving an answer if it finds one. But there

might never be an answer at all because the entire reachable state space of the system might not be completely searched before the available resources are used up. As such, exhaustive search methods become practically more useful if the state explosion problem is handled.

There are two basic types of exhaustive search methods: the breadth-first search and the depth-first search.

In **breadth-first search**, the search proceeds by generating and testing each node (state) that is reachable from a parent node before it expands and searches the children. The search is terminated once the target node is found. If no target is specified it will stop once all the nodes (states) have been generated (visited at least once). This is illustrated in Figure $2.7(a)$ with a tree whose root node is labeled s_0 to indicate the starting state of the search. The target node is labeled g while the search order is illustrated using numbers $1, 2, 3 \ldots$ in that order until the target node is found. This search method is said to be exhaustive because there is a guarantee



(a) Search order for breadth-first search

(b) Search order for depth-first search

Figure 2.7: The two basic exhaustive search methods

that all reachable states are generated by the search before it terminates with failure if the target is not found.

In **depth-first search**, the search proceeds by generating and testing one child node and then expanding and searching that child until the maximum depth is reached. When the maximum depth is reached and the target node is not found, the search backtracks to the previous level and searches any remaining paths from this level. This carries on until the target node is found. The backtracking procedure guarantees that it will exhaustively search all the reachable nodes of the tree if no target is specified before terminating. This is illustrated in Figure $2.7(b)$.

Model checking algorithms typically employ either breadth-first search or depth-first search. The concepts and terminology discussed in this chapter will be used through out the remainder of this essay.

3. Two Approaches to Model Checking

The goal of this chapter is to describe the two model checking algorithms that have been developed by Emerson and Clarke, and Queille and Sifakis. The concepts or requirements of model checking discussed in the previous chapter are used to explain how the algorithms can be used to verify that a given correctness property of a system expressed in temporal logic holds in the system, modeled as a state transition system or an interpreted Petri net (IPN).

3.1 The Model Checker by Emerson and Clarke

in the Emerson and Clarke model checker, a branching time temporal logic known as Computation Tree Logic (CTL) is used to express the correctness properties of the system being verified, while the system is modeled as a finite state transition system. The algorithm searches the entire state space of the state transition system to determine if it is a model of a given correctness property of the system expressed as a formula in computation tree logic.

To verify a given formula f holds in a finite state transition system (Kripke structure) $K = (S, s_0, R, L)$, the algorithm operates in stages. At each stage, all subformulae of f of length equal to the stage number are processed and each reachable state of the state transition system K, is labeled with the set of the subformulae of f that are true in that state. The length of a formula is determined by the number of its operators and operands. For instance, a formula $f = f_1 \wedge f_2$ with subformulae ($f_1 \wedge f_2$, f_1, and f_2) is of length 3, and its subformulae are of length 3, 1, and 1, respectively. The algorithm begins by processing the subformulae with the lowest length and labels each state with the subformulae that holds in that state. Using our earlier example, at stage 1, all subformulae of f that is of length 1 (i.e., f_1 and f_2) are processed and the states in which they hold are labeled with them. In stage 2, subformulae of f of length 2 are processed and so on. At the end of the ith stage, each state will be labeled with the set of subformulae of length less than or equal to i that are true in it. If the expression label(s) is used to denote the set of subformulae true for state s, then at the termination of the algorithm, $K, s \models f$ holds true for all states s of K, if and only if $f \in$ label(s); hence we say that the formula f holds true in structure K or K is a model of formula f.

With the assumption that the subformulae of formula f have been correctly labeled in the states of K where they hold, we now explain how the algorithm is used to verify that the formula f holds in all the states of K. We start with the state labeling algorithm called **procedure** label-graph(f), which is used to label the states where f holds, with f. It is designed to handle seven different forms of CTL: (i.e., f being an atomic proposition, or having one of the following forms: $\neg f$, $f_1 \wedge f_2$, AXf_1, EXf_1, $A[f_1 \cup f_2]$, $E[f_1 \cup f_2]$. Now, lets consider the case when we have a formula of the form $f = A[f_1 \cup f_2]$. In this case, the algorithm uses a depth-first search to explore the state space of the state transition system representing the system being verified. The procedure that does the search in this case is called **procedure** $au(f, s, b)$. It performs a search on the entire state space of K starting from a state s. When this procedure terminates, the boolean result parameter b will be set to true if f holds in s, otherwise it will be set to

false. The states that have been searched are marked using a bit array in the algorithm termed marked$[1 : nstates]$ which indicates that they have been visited. For example, marked(s):= true means that state s has been searched, but if false, it means that state s has not been searched). Once a state has been searched, but was not labeled, it is kept in a stack variable ST to indicate that it has been searched but the truth or falsity of the formula f has not yet been determined in it and thus needs additional processing. Another boolean procedure termed stacked(s) is used to determine whether a state s is currently on the stack ST.

The notations explained below are used to denote how to manipulate any given formula, how formulae are assigned to labels and how labels associated with the states of the system are accessed.

- Notations arg$1(f)$ and arg$2(f)$ are used to denote the first and second arguments of a formula f with two arguments; for example if $f = A[f_1 \cup f_2]$, then arg1 $= f_1$ and arg2 $= f_2$.
- labeled(s,f) returns true if a state s is labeled with formula f (i.e. if f holds in state s), and returns false if state s is not labeled with f (i.e. if f does not hold in state s).
- addlabel(s,f) is used to add formula f to the current label of state s if f holds in state s.

The algorithm for the case considered, as adopted from [CES83], is given below:

Procedure label-graph(f):

begin

ST := empty-stack; (the stack is currently empty)

for all $s \in S$ **do** marked(s) := false; (this indicates that the states have not be searched)

L: for all $s \in S$ **do** (A loop that ensures that the entire state space is searched)

If ¬marked(s) then $au(f, s, b)$ (searches a state that has not been marked)

end.

The **procedure** label-graph(f) calls the **procedure** $au(f, s, b)$ if a state is not marked (i.e. a state has not been searched). Once called, the **procedure** $au(f, s, b)$ searches the entire state space of K starting from state s to determine if formula f holds in them. If a state s is marked and stacked, it returns false (i.e. b := false), implying that state s has been searched but was not labeled and is in stack ST, and thus requires further processing before the truth or falsity of the formula f is determined in it. However, if a state s is already labeled with f, then the procedure returns true (i.e. b:= true), because the formula f holds in state s. Furthermore, if a state s is marked but neither stacked nor labeled, then it returns false, because the formula f does not hold in state s, and need not be put in the stack ST for further processing. This is illustrated in lines $2 - 16$ of the code for **procedure** $au(f, s, b)$, shown in Appendix A

If a state s is searched and the formula f holds in it, state s is labeled with f and the procedure returns true. For instance, for $f = A[f_1 \cup f_2]$: if f_2 is true at state s, then formula f is true at

state s. The algorithm labels state s with f and returns true. If f_1 and f_2 are false at state s, then formula f is false at state s, as such state s will not be labeled with f, and the algorithm returns false. This is illustrated in lines $17 - 28$ of the code for **procedure** $au(f, s, b)$.

However, if f_1 is true at state s and f_2 is not, then state s requires further processing to determine if formula f is true in it or not, and it is therefore kept in the stack ST. The further processing on state s requires checking to see if formula f is true at all successor states of state s. If there is some successor state s_1 at which formula f is false, then formula f is also false at state s. State s is removed from the stack ST and the algorithm returns false. But if formula f is true for all successor states of state s, then it is also true at state s; s is then removed from the stack ST and labeled with f. The algorithm then returns true.This is illustrated in lines $29 - 43$ of the code for **procedure** $au(f, s, b)$.

For the case of the formula of the form $f = E[f_1 \cup f_2]$, the algorithm works as explained above, except for the fact that all states labeled with f_2 are found first, then all the predecessor states of these states are checked to verify if f_1 holds in them. If that is the case, then f holds in those states. The model is said to satisfy f if there exits a path where f holds.

The correctness of this algorithm is established if the condition:

$$\forall s[labeled(s, f) \leftrightarrow s \models f]$$

holds when it terminates. The proof of this condition is not covered in this essay; interested readers are referred to [CES83].

3.1.1 Extension of the Algorithm to handle Fairness in Concurrent Systems

When verifying concurrent systems, correctness properties of the system along fair execution sequences are considered. For example, verifying the correctness property of a system that comprises many processes entails considering only those computation sequences in which each process of the system is executed infinitely often. In other words, a fairness condition in a system asserts that requests for service from each of the processes are granted sufficiently often [CES83]. In order to extend the algorithm to handle the verification of such systems along fair computation paths, it was observed that CTL could not express the correctness of fair executions in such systems, and its semantics was modified to obtain a new logic called CTL^F [CES83, CES86]. CTL^F has the same syntax as CTL. But whereas CTL is interpreted over Kripke structures as defined in section $2.3.2$, CTL^F formulae are interpreted over Kripke structures that are extended with an extra component F, which is a collection of subsets of S (i.e., $F \subseteq 2^S$). A path p is said to be F-fair if and only if the following condition holds:

each $c \in F$, appears infinitely many times on p.

CTL^F has the same semantics as CTL except that its path quantifiers (A and E) range over fair paths.

In order to extend the model checking algorithm to CTL^F, an additional proposition Q is introduced, which is said to be true in a state s if and only if there is a fair path starting from s. This is implemented by calculating the strongly connected components of the state space. A strongly connected component is said to be fair if it contains at least one state from each c_i in F. A state is then labeled with Q if and only if there is a path from it to some state of a fair strongly connected component.

An illustration of how fair paths of a system are verified with respect to a given formula f using this extended algorithm is given below as adopted from [CES83]; cases of the form where either $f = E[f_1 \cup f_2]$ or $f = A[f_1 \cup f_2]$ are considered with an assumption that the states of the state transition system have already been labelled with the immediate subformulae of f that are true in them.

1. $f = E[f_1 \cup f_2]$: f is true in a state if and only if the CTL formula $E[f_1 \cup (f_2 \wedge Q]$ is true in that state. This is determined using the earlier described CTL model checking algorithm. A state s is thus labelled with f if and only if f is true in it.

2. $f = A[f_1 \cup f_2]$. This can also be written as $A[f_1 \cup f_2] = \neg(E[\neg f_2 \cup (\neg f_1 \wedge \neg f_2)] \vee EG(\neg f_2))$. A state s can be checked if $s \models E[\neg f_2 \cup (\neg f_1 \wedge \neg f_2)]$ using the previous algorithm described earlier. But to check if $s \models EG(\neg f_2)$, an additional procedure is required which is: If K_T is taken to be a graph corresponding to the above formula structure; all states v such that $f_2 \in$ label(v) are eliminated from K_T and the resultant K'_T is obtained. All the strongly connected components of K'_T are found with those that are fair marked. If state s is in K'_T and there is a path from state s to a fair strongly component of K'_T then $s \models EG(\neg f_2)$, otherwise $s \models \neg EG(\neg f_2)$. As in 1 above, state s is labelled with f if formula f is true in s.

3.2 The Model Checker by Queille and Sifakis

The model checker developed by Queille and Sifakis is known as CESAR. Given an algorithmic description of a system using a program in a high-level language, the program is automatically translated by a translator to obtain a model of the system which represents some aspects of its described behaviour. This model which is an interpreted Petri Net (IPN) is then evaluated by an analyzer to verify if the described system satisfies a given specification of the correctness property of the system expressed in a branching time logic [QS82]. Figure 3.1 illustrates the CESAR model checker.

3.2.1 The Description Language

In this approach, the system being verified is described as a set of communicating sequential processes. The processes communicate by exchanging variables. Exchange is brought about by an agreement between two processes, in which one executes an output operation and the other executes an input operation [QS82]. The basic statement of the description language is the

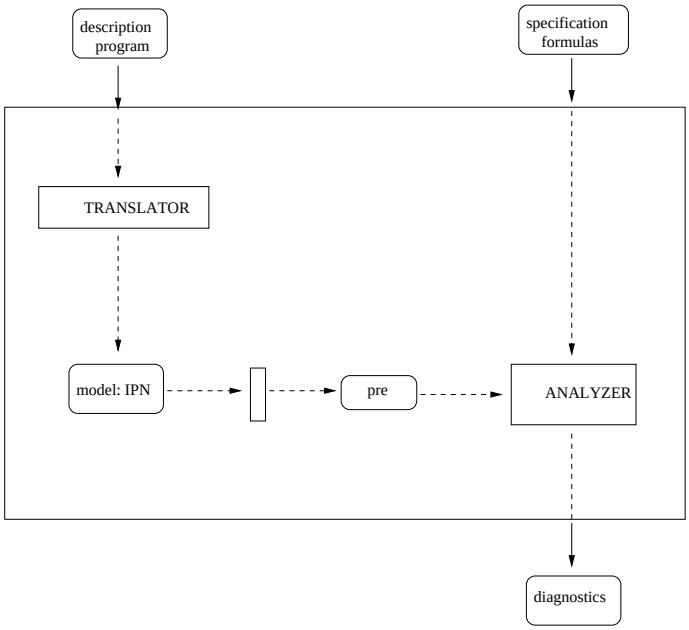


Figure 3.1: Illustration of the model checker CESAR

vectorial assignment, and it enables a simultaneous execution of an input and output operation. Two non-deterministic composed statements of the description language adopted from [QS82] are:

if $b_1 \rightarrow s_1$

$b_2 \rightarrow s_2$

fi

do $b_1 \rightarrow s_1$

$b_2 \rightarrow s_2$

od

where the $b_i's$ are boolean conditions known as guards and the $s_i's$ are sequences of statements. The if and do constructions have the following meanings:

IF: If any of the conditions is true, the corresponding sequence of statements are executed.

DO: This allows the repetition of an IF statement until the EXIT statement is encountered during the execution of some sequence of statements.

The choice about which sequence of statements to execute becomes non-deterministic if more than one condition is true.

3.2.2 Translation of Description Programs into IPN

Given a program which describes the system in the input language of the system CESAR, the translator generates an interpreted Petri Net (IPN) which represents the main aspects of its behaviour. This is done by composing the IPN that represents the sequential processes of the program.

3.2.3 The Specification Language

The specification language used by the model checker CESAR is a branching time logic L. It is constructed from a set of propositional variables F and the constants true and false (denoted as tt and ff respectively), by using the logical connectives $\neg$, $\wedge$, $\vee$, and $\rightarrow$, explained in Chapter 2, and the unary temporal operators POT (for potentially) and $INEV$ (for inevitable). The formulae of L is used to express the correctness properties of a system. To express these properties, a transition system is used as a model for L, with its semantics defined over it. Such a transition system is defined as a doublet $S = (Q, \rightarrow)$, where Q is a set of states and $\rightarrow$ is a binary relation on $Q(\rightarrow \subseteq Q \times Q)$, which represents the actions or transitions of the system. The semantics of L with respect to a formula F over a transition system S is given below as adopted from [QS82]. The set of all execution sequence from a state q is denoted as $Execq$ and the k-th element of a sequence s is denoted as $s(k)$.

$\forall f \in F \quad |f| \in [Q \rightarrow tt, ff]$, where $[Q \rightarrow tt, ff]$ is the set of the predicates on Q

$\forall q \in Q \quad |true|(q) = tt$

$\forall f \in L \quad |\neg f|(q) = tt \quad$ iff $\quad |f|(q) = ff$

$\forall f_1, f_2 \in L \quad |f_1 \wedge f_2|(q) = tt \quad$ iff $\quad |f_1|(q) = tt$ and $|f_2|(q) = tt$

$\forall f \in L \quad |POT(f)|(q) \equiv \exists\ s \in Execq\ \exists k \in \mathbf{N} \quad [q \rightarrow s(k)$ and $|f|(s(k))]$. This is equivalent to $AF(f)$ in CTL.

$\forall f \in L \quad |INEV(f)|(q) \equiv \forall\ s \in Execq\ \exists k \in \mathbf{N} \quad [q \rightarrow s(k)$ and $|f|(s(k))]$. This is equivalent to $EF(f)$ in CTL.

$|POT(f)|$ represents the set of states q of S such that there exist an execution sequence s starting from q in which $|f|$ holds in one of the states, and $|INEV(f)|$ represents the set of states q of S such that for all execution sequences s starting from q, there exists a state in which $|f|$ holds.

The dual operators of L, ALL and $SOME$ are interpreted as:

$|ALL(f)|(q) \equiv \forall s \in Execq\ \forall k \in \mathbf{N} \quad [q \rightarrow s(k)$ implies $|f|(s(k))]$. This is equivalent to $AG(f)$ in CTL.

$|SOME(f)|(q) \equiv \exists s \in Execq\ \forall k \in \mathbf{N} \quad [q \rightarrow s(k)$ implies $|f|(s(k))]$. This is equivalent to $EG(f)$ in CTL.

It follows that if a state q of a transition system satisfies $|ALL(f)|$, then all the states of all the

execution sequences from q satisfy $|f|$. Similarly, if a state q satisfies $|SOME(f)|$, then there exists some execution sequence from q such that all its states satisfy $|f|$.

Some examples illustrating the use of the specification language for expressing system properties are:

- **init** $\rightarrow ALL(p)$: This means that the proposition p is always true. This is an example of an invariant property (a property expected to hold true and never changes throughout the execution of a program).
- **init** $\rightarrow ALL\ POT$ **enable** a: This means that from every state q, which is a successor of a state satisfying **init**, there exists an execution sequence $s \in Execq$ containing a state in which action a is enabled. This is an example of a liveness property (a property that asserts that something good will eventually happen in the system).

Note that the keywords **init** and **enable** are propositional variables, where **init** is true in a state if that state is an initial state and **enable** is true in a state if an action can be executed from that state.

3.2.4 The Verification Method

The verification method consists of the iterative computation of fixed points of a function called a predicate transformer by the analyzer. As a result, it interprets the temporal operators used in constructing the formula, which expresses the correctness properties of the system being verified. A predicate transformer is a function that maps a set of states unto a set of states (i.e.,pre: $pred(s) \rightarrow pred(s)$

The concept of iteratively computing fixed points using the predicate transformer is explained as follows: The predicate transformer on a set of states Z adds unto the set, the set of states that make the formula true. The algorithm starts with Z as an empty set. Then iteratively, the predicate transformer is applied on the set of states in Z. After each iteration, the set of states in which the formula holds is added to the set Z. Then the predicate transformer is again applied on the resulting set of states. After a number of iterations, a fixed point is reached, in which all the reachable states where formula f holds are assigned to Z, so that further iterations do not result in the addition of states. If, at the fixed point, the initial state s_0 is included in Z, then we conclude that formula f, holds in the model.

Now lets illustrate the above concept with an example, using the formula $f = POT(f_1)$. We are going to consider a transition system as a model of this formula, since, as we noted before, an interpreted Petri net can also be explained in this way. To do this, we express the formula in its CTL equivalence as: $EF(f_1)$. Let Z represent the set of states in which the formula holds. The predicate transformer pre on f is then given by: pre$(Z) = f_1 \vee EX\ Z$. This means, the set of states where either f_1 holds or there exists a successor state which is an element of Z. Using the transition system in Figure 3.2, we start with $Z = false$, indicating that no state is in Z. Thus, in the first iteration, we search for the states in which f_1 holds, and add them to Z. This

updates Z with s_1. At the second iteration, we compute our predicate transformer again, but with Z being a non-empty set, because it now contains s_1. After the second iteration, s_0 is added to Z because there exists a successor state s_1, which is in Z. This results in $Z = \{s_1 \cup s_0\}$. We continue in the same manner so that after the fourth iteration, $Z = \{s_1 \cup s_0 \cup s_2 \cup s_3\}$. Now notice that at the fifth iteration, no further state is added to Z, since it now contains all the reachable states. This is known as the fixed point of the predicate transformer, hence pre$(Z) = Z$. Since the initial state s_0 is in Z, we conclude that our model satisfies f.

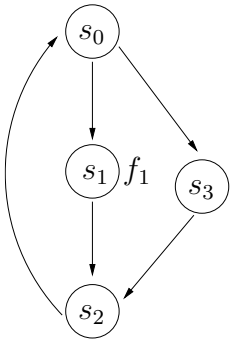


Figure 3.2: Transition system for formula $f = EF(f_1)$

Having explained how the fixed point of predicate transformer is computed iteratively, we now explain the procedures carried out by the analyzer to verify the properties of a system as follows: Given a formula f to be verified on a description program PROG and N the IPN obtained by the automatic translation of program PROG, $F = \{f_1, ...f_n\}$ is used to denote the set of the propositional variables of f. The following procedures as adopted from [QS82] are carried out by the analyzer to verify whether the formula f holds in the program.

1. Each place of N is associated with a boolean variable.
2. Each of the propositional variables of f is associated with a predicate which is the set of states/places of N that makes it true.
3. **init** is expressed as a predicate representing the set of all possible initial states.
4. N is reduced to obtain an IPN of less complexity using transformation rules which do not affect the places that make up the predicates.
5. Each temporal operator of f, is then characterised with a predicate transformer.
6. After the above 5 preliminary procedures, the algorithm is then used to verify the formula on the model by iteratively computing the predicate transformer on the predicates to obtain the set of states that make the formula true. The algorithm is allowed to run within an acceptable number of iterations, in order to avoid explosion while trying to compute the fixed point. In the course of the computation, if some iterative computation yields no result (i.e. no fixed point is reached) within the acceptable number of iterations, then the analyzer will not give an answer. Otherwise, it evaluates $|f|$, and if $|f|(q) = tt$ for every state q, then we say that the property of the system described by f is verified.

4. Comparison of the two Approaches

1. The Emerson and Clarke approach does not explicitly explain how the behaviour of the system is first specified using a programming language. Rather, it states that the system can be modeled directly using a state transition system and explicitly explain how the model checking algorithm is used to verified that a system satisfies its correctness properties. On the other hand, the Queille and Sifakis approach discussed all the aspects of their system which comprises the description language, the specification, the result used by the analyzer and the procedures taken to verify that a system satisfies its correctness properties.

2. The algorithms work in different ways; the Emerson and Clarke algorithm verifies a given formula by first breaking it into its subformulae. It checks that each subformulae is satisfied by the model before it ascertains that the formula is satisfied by the model. The Queille and Sifakis algorithm interprets the temporal operators of a formula and checks if it is satisfied by the model. This is done by iteratively computing the fixed points of its predicate transformer. Once a fixed point is reached, the formula is ascertained to be satisfied by the model.

3. The behaviour of the system being verified is modeled as a state transition system in the Emerson and Clarke approach, but as an Interpreted Petri net (IPN) generated from the program description in the Queille and Sifakis approach.

4. They both use a branching time logic, CTL and L respectively, to represent the correctness property of the system being verified. The temporal operators of these two logics have the same meaning but differ in syntax as shown in the examples below:

- $INEV(f)$ in L has the same meaning as $AF(f)$ in CTL
- $POT(f)$ in L has the same meaning as $EF(f)$ in CTL
- $ALL(f)$ in L has the same meaning as $AG(f)$ in CTL
- $SOME(f)$ in L has the same meaning as $EG(f)$ in CTL

4.1 How the Algorithms can be used to Verify The Alternating Bit Protocol

The Alternating Bit Protocol is a data link layer protocol that sees that lost and corrupted messages transmitted via a one-directional transmission line are retransmitted. In this protocol, a single control bit is used as a control measure for each transmitted message or acknowledgement in order to detect lost messages or acknowledgements and also help to recover them. It also makes sure that corrupted or unwanted messages are detected during transmission.

The protocol consists of two processes: a sender process and a receiver process, thus a good example for verifying concurrent programs. It functions as follows: the sender sends messages to the receiver, while the receiver in turn replies by sending an acknowledgement. Each message is encoded to avoid unwanted messages from intruders, and associated with a control bit which changes from one message to another. Once a message is sent with a specific control bit, the sender does not change the control bit in order to send another message until the acknowledgement to the message with its corresponding control bit is received. Thus for each message sent with a control bit, its corresponding acknowledgement is received with the same bit by the sender before another message is sent with another control bit. An arbitrary clock is used to measure a certain time period in which the sender can wait for an acknowledgement to any message sent should the message be lost during transmission. Once the time period elapses and no acknowledgement is received, the sender resends the message with its control bit. With this, lost messages can be recovered and there is a guarantee that every message sent must be received and acknowledged with several repetitions. The receiver functions in like manner by receiving a message and sending an acknowledgement.

4.1.1 Emerson and Clarke's Approach in Verifying The Alternating Bit Protocol

In their approach, the ABP program is specified using a programming language for communicating sequential processes. The two processes considered are the sender and receiver processes. They alternately exchange messages as seen in their corresponding CSP programs in Appendix B, adopted from [CES83]. Note that in the program, dm denotes a data message from the sender while am denotes an acknowledgement message from the receiver.

From the sender process in Appendix B, the sender generates and sends a message to the receiver, and then awaits an acknowledgement from the receiver. The acknowledgement sent can either be the correct one being expected or a corrupted one. If the latter is received, it is reported back as an error message. Similarly, the receiver receives a message from the sender and replies with an acknowledgement. If the message received is the expected message, its corresponding acknowledgement is sent, otherwise it is reported as an error message.

From the state transition system of the individual processes, a global state transition system of the ABP is generated by considering the possible ways in which the transitions of each process from one state to another may be interleaved, as seen in Figure 4.1. The model checker is then used to determine whether the ABP program satisfies its specification: that every data message generated and sent by the sender process is eventually accepted by the receiver process. This is written formally as:

$AG[gen\text{-}dm0 \rightarrow AX[A[\neg(gen\text{-}dm0 \vee gen\text{-}dm1) \cup acc\text{-}dm0]] \wedge$

$AG[gen\text{-}dm1 \rightarrow AX[A[\neg(gen\text{-}dm0 \vee gen\text{-}dm1) \cup acc\text{-}dm1]]$

From Figure 4.1, we can see that if only fair computation paths are considered (i.e. ignoring the infinite paths in which messages are lost after being retransmitted), then the extended model checker that can handle fairness discussed earlier in this chapter, can be used to correctly verify

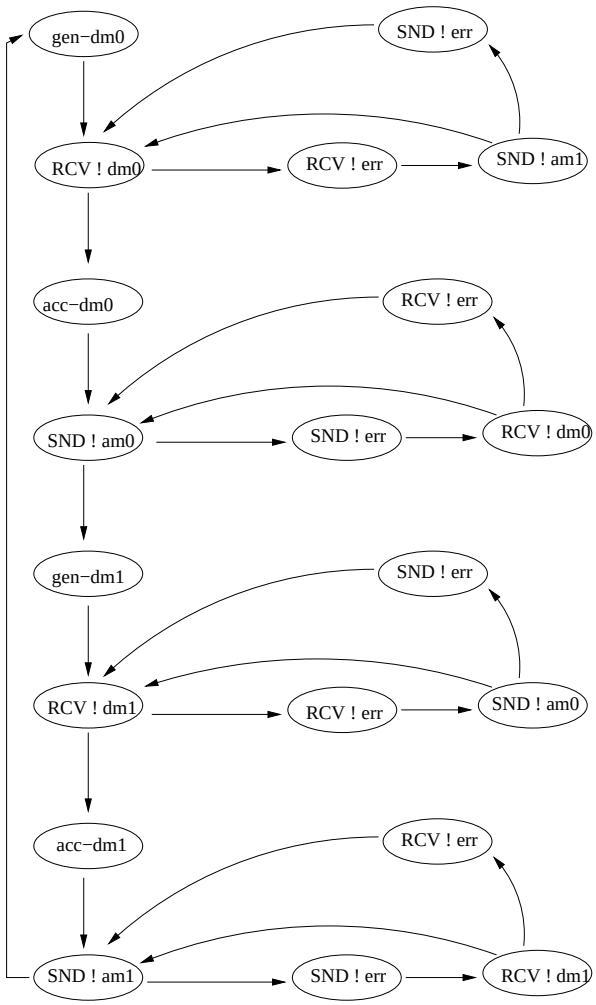


Figure 4.1: The global state transition system of the ABP

that the global state transition system satisfies its specifications.

For the sake of this essay, we are going to illustrate how the algorithm can be used to verify the formula: $f = A[\neg(gen\text{-}dm0 \vee gen\text{-}dm1) \cup acc\text{-}dm0]$ in the ABP model in Figure 4.1. Formula f means that neither the first nor the second message is generated and sent until an acknowledgement of the first message is received. To do this, we ignore the path in which messages can be garbled or lost and consider the path:
$(gen\text{-}dm0, RCV!dm0, acc\text{-}dm0, SND!am0, gen\text{-}dm1, RCV!dm1, acc\text{-}dm1, SND!am1)$. We denote the first element in this path as first state, the second element as second state... in that order, so that the last element is the eighth state. Next, we break the formula f into its subformulae as:

1. $A[\neg(gen\text{-}dm0 \vee gen\text{-}dm1) \cup acc\text{-}dm0]$
2. $\neg(gen\text{-}dm0 \vee gen\text{-}dm1)$
3. $gen\text{-}dm0 \vee gen\text{-}dm1$

4. $gen\text{-}dm0$
5. $gen\text{-}dm1$
6. $acc\text{-}dm0$

Now, we begin the verification task with subformula 6, and label the states in the path we are considering where it holds. This labels the third state with subformula 6. Next we check for subformula 5, and label the fifth state with 5. Similarly, we label the first state with 4, and the first and fifth states with 3. Now we check for 2, and label all the states where 3 is false with 2. The diagram in Figure 4.2 illustrates this.

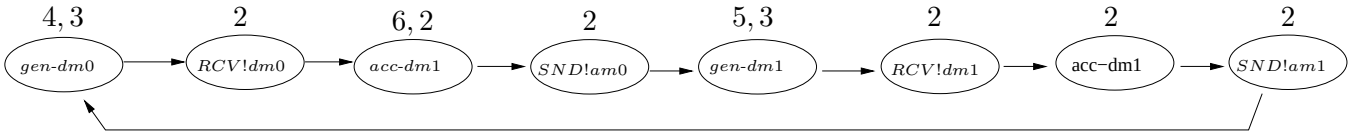


Figure 4.2: Fair computation path of ABP

Lastly, we use the concept of the procedure in Appendix A to verify if 1, which is formula f holds in our model. We expect f to hold if either 6 holds in the first state, or 2 holds and in its successor states, 1 holds. Since neither 2 nor 6 holds in the first state, we conclude that formula f does not hold in the model. Formula f is false in the model because the first message must be generated and sent before its acknowledgement is received. Hence the correct formula that is satisfied in the model is $AG[gen\text{-}dm0 \rightarrow AX[A[\neg(gen\text{-}dm0 \vee gen\text{-}dm1) \cup acc\text{-}dm0]]$, which means that globally along every path, once the first message is generated and sent, then at the next successor states, the same message or another message is not generated until the acknowledgement of the first message is received.

4.1.2 Queille and Sifakis's Approach in Verifying The Alternating Bit Protocol

In this approach, the ABP program is described by 4 processes: the sender process, the receiver process, and the transmission line which in turn is described by two processes; the sendtoreceive and receivetosend processes. The sender process and the receiver process function symmetrically. The sender sends a message and expects an acknowledgement to indicate that the receiver gets the message correctly. Once the acknowledgement is received, its further checked to see if it is actually the expected one by comparing its control bit with the control bit of the message that was sent. Repeated messages and acknowledgements with the same control bit are discarded except the first one received. Similarly, the receiver gets a message from the sender, checks if it is the expected message after which he replies with an acknowledgement, and then awaits the second message. If a message comes in with the previous control bit or an unwanted message, it is discarded and the previous acknowledgement sent, is repeated.

The sendtoreceive transmission line process transmits messages from sender to receiver. There are two situations that can occur, either the message is properly transmitted or it is lost or garbled

during transmission. If the latter happens, the sender is expected to resend the message without changing the control bit. The receivetosend process functions symmetrically by transmitting acknowledgement. The corresponding IPN for the ABP program illustrating the above explanation is shown in Figure 4.3. Some properties of the ABP which can be verified in the resulted IPN are given below:

- Init $\rightarrow$ ALL (**after** (send,repeat) $\rightarrow$ POT **enable** receive): This means that every message from the sender is eventually received by the receiver
- Init $\rightarrow$ ALL (**after** (sendack,repeatack) $\rightarrow$ POT **enable** receiveack): This means that every acknowledgement from the receiver is eventually received by the sender.

For the sake of this essay, we are only going to illustrate how the algorithm is used to verify the formula $f = POT$ **enable** (sender) , using the corresponding IPN of the ABP in Figure 4.3. Formula f means it is possible that at a point in time the sender can send a message. We begin by associating the propositional variable **enable** (sender) with a predicate, which is the set of states/places on the IPN that makes it true. Lets denote this set of states with P. Thus $P = \{s1m1 \cup s2m1 \cup s2a2 \cup s3\}$. Next the predicate transformer is applied on this set of states and adds unto P the set of states in which either **enable** (sender) holds or their exists a successor state of theirs contained in P. Lets denote this set of states as P'. Thus $P' = \{s3 \cup r2 \cup r3 \cup a2 \cup m2\}$. Hence, after the first iteration, $P = \{P \cup P'\}$. During the second iteration, $r1$ is added to the set, because it now has a successor state in P. After the third iteration, a fixed point is reached because no further set of states is added to the set P since all the reachable states are now contained in the set of states P. Hence formula f is verified in the IPN model.

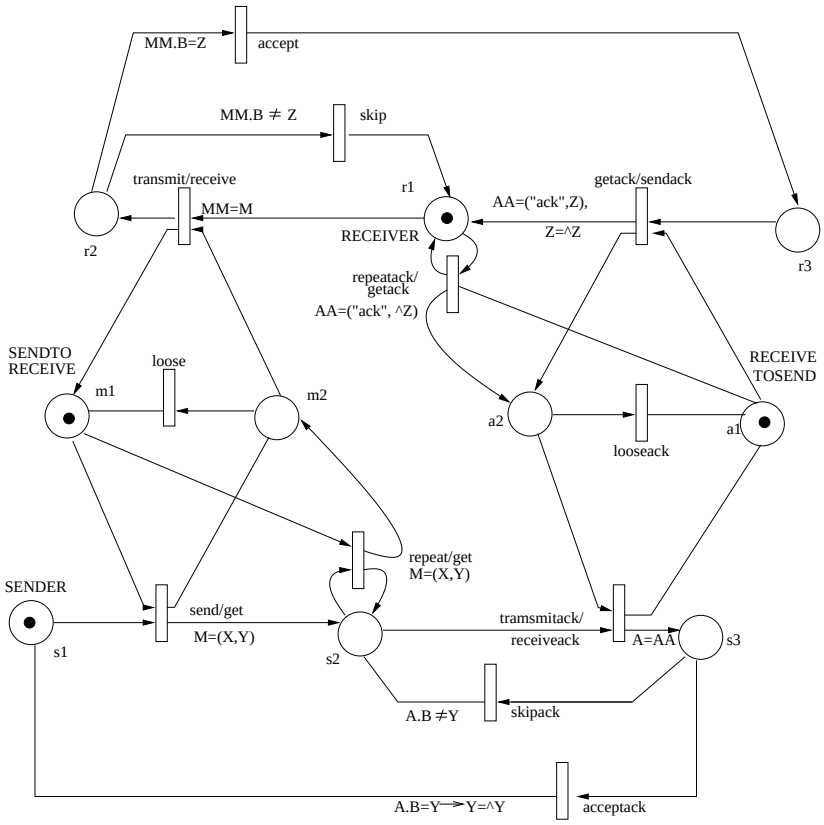


Figure 4.3: The corresponding IPN of the ABP program

4.2 Strengths and Limitations of the two Model Checking Algorithms

In this section, the strengths and limitations of the two model checking algorithms described earlier in this chapter are enumerated.

4.2.1 Strengths of Emerson and Clarke's Model Checking Algorithm

1. The non complexity of this model checking algorithm makes it a widely referenced approach in the verification of systems by researchers and practitioners.

2. The exhaustive search method employed while checking that the global state transition system of a given system satisfies its correctness property, guarantees that the entire state space of the system would be checked to ascertain its correctness with respect to those properties.

3. The use of a finite state transition system to represent the behaviour of the system abstracts from details of the system not relevant in the verification process.

4. The use of temporal logic makes it possible to express a wide range of correctness properties.

5. The extension of the model checking algorithm to handle fairness conditions makes it possible to verify systems along fair computation paths.

4.2.2 Limitations of Emerson and Clarke's Model Checking Algorithm

1. The major limitation of this approach is that the algorithm requires the entire state transition system to be stored in memory. The state explosion problem and limitation of memory make it difficult to represent large and complex systems using state transition system. Thus limiting this approach to the verification of small systems.

2. Another major limitation is time resources due to the recursive method used in the algorithm. The time spent in labeling the states with each subformula recursive call, is proportional to the number of states and number of edges of the state transition system representing the system, thus limiting this approach to the verification of small systems.

4.2.3 Strengths of Queille and Sifakis's Model Checking Algorithm

1. The concept of this model checking algorithm, which is based on the translation of the description of a system given in a high level language into a model which can be verified, makes it possible to abstract from details that are not relevant to the verification of the behaviour of the system, thus reducing the complexity of the system's verification.

2. Translating the description program into a Petri net makes it possible to name the control points of the system and thus makes the expression of the properties of the system being verified a alot easier with respect to these names.

3. The logic L, enables the expression of some fundamental properties of a system (e.g safety properties, liveness properties, and properties of response to an action), and makes the classification and comparison of these properties possible.

4.2.4 Limitations of Queille and Sifakis's Model Checking Algorithm

1. Since fixed point requires the entire reachable states of the system to be in the set of states where the formula holds, these states then must be stored in memory. The state explosion problem and limitation of memory limits this approach to the verification of small states.

2. Due to the iterative computation of fixed point of predicate transformer on every propositional variables and temporal operators used in expressing the system's properties, time resources pose as a limitation to this approach.

5. Conclusion

Every software or hardware system has specific correctness criteria. In the case of concurrent systems, making sure that these criteria are met, becomes a difficult task for developers. Model checking has made this task easy because it is automatic. The success of model checking in the verification of systems is attributed to its ability to verify that a system satisfies its correctness criteria for all possible executions.

The two model checkers discussed in this essay, were the first two that were developed. Ever since their proposal, many other techniques have been developed. In most of them, the concept of the Emerson and Clarke approach was referenced more than the Queille and Sifakis approach. This is due to the explicit way in which the Emerson and Clarke algorithm was explained, thus making it easy to understand. Also the branching time logic CTL introduced in their approach is the most widely used due to its expressive nature in handling systems' properties along non-deterministic computation paths.

In the course of this essay, it was observed that the major problem of these algorithms is that they can handle only small systems due to the state explosion problem. However, many techniques and algorithms have been proposed to handle this problem, thus making it possible to use model checking to verify properties of industrial size systems, but memory and time resources still limit the size of systems that can be verified using model checking. Hence research work on techniques to handle this problem still continues.

Appendix A. The procedure au

Listing A.1: Search procedure

```
1  procedure au(f , s, b)
2  begin
3        if marked(s) then
4                 begin
5                       if stacked(s) then
6                              begin
7                                     b:= false;
8                                     return
9                              end;
10                      if labeled(s,f ) then
11                              begin
12                                    b:= true;
13                                    return
14                              end
15                      b:= false;
16                      return
17                end;
18      marked(s):= true;
19      if labeled(s,arg2(f )) then
20              begin
21                   addlabel(s,f );
22                   b:= true;
23                   return
24              end
25      else if \neg labeled(s,arg1(f )) then
26              begin
27                   b:= false;
28                   return
29              end;
30      push(s,ST);
31      for all s1 \in successors(s) do
32              begin
33                   au(f , s1 , b1 );
34                   if   b 1  then
35                                 begin
36                                      pop(ST);
37                                      b:= false;
38                                      return
39                                 end
40              end;
41      pop(ST);
42      addlabel(s,f );
43      b:= true;
44      return
45 end of procedure au
```

Appendix B. The Alternating Bit Protocol program description

The Sender Process (SND)

[gen-$dm0$;
RCV ! $dm0$;
[RCV ? $am0$ $\rightarrow$ exit;
RCV ? $am1$ $\rightarrow$ RCV ! $dm0$;
RCV ? err $\rightarrow$ RCV ! $dm0$];
gen-dm1;
RCV ! dm1;
[RCV ? $am1$ $\rightarrow$ exit;
RCV ? $am0$ $\rightarrow$ RCV ! $dm1$;
RCV ? err $\rightarrow$ RCV ! $dm1$]];

The Receiver Process (RCV)

[[SND ? dmo $\rightarrow$ exit;
SND ? $dm1$ $\rightarrow$ SND ! $am1$;
SND ? err $\rightarrow$ SND ! $am1$];
acc-$dm0$;
SND ! $am0$;
[SND ! $dm1$ $\rightarrow$ exit;
SND ? $dm0$ $\rightarrow$ SND ! $am0$;
SND ? err $\rightarrow$ SND ! $am0$];
acc-$dm1$;
SND ! $dm1$;
]

Acknowledgement

The success of this work is attributed first to God Almighty for His infinite mercies, love, wisdom and knowledge granted onto me. All glory, honor and praise to His mighty name. I'm very much indebted to my supervisors: Dr. Jaco Geldenhuys and Dr. Corne Inggs for imparting the knowledge of what this work entails in me and seeing that this work was a success. Thanks for all your time and commitment throughout the duration of this work.

My profound gratitude to Prof. Neil Turok through whose initiative was AIMS instituted, to Prof. Fritz Hahne the director of AIMS, and all the staff of AIMS (academic and non academic), who have made my study at AIMS one that would never be forgotten. My special thanks to all my tutors: Henry Amuasi, Laure Gouba, Anahita New, Sam Webster, Paul Razafimandimby, Ambrose Chongo and my essay tutor Christian Rivasseau, thanks for all your efforts to see that this work was a success. I will always remain grateful to you all.

The tremendeous help and prayers from my parents and siblings cannot be quantified. You all are the reason why I'm where I am today. The support from some one dear to me, Mr. Isaac Olaoye will always be appreciated. Thanks for always being there for me. I won't fail to thank the entire student body of AIMS 2006 batch especially my Nigerian colleagues: Margaret, Emmanuel, Lois, Ndubuisi, Saheed, Wole and Victoria. You all contributed in making my stay at AIMS memorable.

Merci beaucoup mes amis !!!!!.

Bibliography

[Bar91] D. C. Barnard, *Reducing the state explosion problem during model checking*, Masters thesis, University of Stellenbosch, 1991.

[Bry86] Randal E. Bryant, *Graph-based algorithms for boolean function manipulation*, IEEE Transactions on Computers **35** (1986), no. 8, 677–691.

[Bul04] J. J. D. Bull, *A comparison of two different model checking techniques*, Masters thesis, University of Stellenbosch, 2004.

[CB98] Edmund M. Clarke and Sergey Berezin, *Model checking: Historical perspective and example (extended abstract)*, TABLEAUX '98: Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (London, UK), Springer-Verlag, 1998, pp. 18–24.

[CDW04] Hao Chen, Drew Dean, and David Wagner, *Model checking one million lines of C code*, Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS) (San Diego, CA), February 4–6, 2004, pp. 171–185.

[CE82] Edmund M. Clarke and E. Allen Emerson, *Design and synthesis of synchronization skeletons using branching-time temporal logic*, Logic of Programs, Workshop (London, UK), Springer-Verlag, 1982, pp. 52–71.

[CES83] E. M. Clarke, E. A. Emerson, and A. P. Sistla, *Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach*, POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (New York, NY, USA), ACM Press, 1983, pp. 117–126.

[CES86] ———, *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM Trans. Program. Lang. Syst. **8** (1986), no. 2, 244–263.

[CFJ93] E. M. Clarke, T. Filkorn, and S. Jha, *Exploiting symmetry in temporal logic model checking*, Computer Aided Verification: Proc. of the 5th International Conference CAV'93 (C. Courcoubetis, ed.), Springer, Berlin, Heidelberg, 1993, pp. 450–462.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long, *Model checking and abstraction*, ACM Transactions on Programming Languages and Systems **16** (1994), no. 5, 1512–1542.

[CTM+99] Sérgio Campos, Márcio Teixeira, Marius Minea, Edmund Clarke, and Andreas Kuehlmann, *Model checking semi-continuous time models using BDDs*, Elsevier science B.V (1999).

[EMCP00] Orna Grumberg Edmund M. Clarke and Doron A. Peled, *Model checking*, The MIT press, 2000.

[ES93] A. E. Emerson and A. P. Sistla, *Symmetry and model checking*, Computer Aided Verification: Proc. of the 5th International Conference CAV'93 (C. Courcoubetis, ed.), Springer, Berlin, Heidelberg, 1993, pp. 463–478.

[Gei00] Marc Geilen, *Model-checking in simulations of distributed systems*, Proceedings of 12th European Simulation Symposium ESS (Hamburg, Germany), September 28–30, 2000, pp. 28–30.

[GR82] Claude Girault and Wolfgang Reisig (eds.), *Application and theory of petri nets 1982, 16th international conference, turin, italy, june 26-30, 1982, proceedings*, Lecture Notes in Computer Science, vol. 935, Springer, 1982.

[Hol95] G. J. Holzmann, *An analysis of bitstate hashing*, Proc. 15th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP (Warsaw, Poland), Chapman and Hall, 1995, pp. 301–314.

[ID93] C.N. Ip and D.L. Dill, *Better verification through symmetry*, Computer Hardware Description Languages and their Applications (Ottawa, Canada) (D. Agnew, L. Claesen, and R. Camposano, eds.), Elsevier Science Publishers B.V., Amsterdam, Netherland, 1993, pp. 87–100.

[Lut06] Carsten Lutz, *Temporal logic*, Summer semester lecture notes in Computer Science (2006).

[Mer00] Stephan Merz, *Model checking: A tutorial overview*, MOVEP '00: Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes (London, UK), Springer-Verlag, 2000, pp. 3–38.

[MK99] Jeff Magee and Jeff Kramer, *Concurrency: state models & java programs*, John Wiley & Sons, Inc., New York, NY, USA, 1999.

[Pel93] D. Peled, *All from one, one for all: On model checking using representatives*, Computer Aided Verification: Proc. of the 5th International Conference CAV'93 (C. Courcoubetis, ed.), Springer, Berlin, Heidelberg, 1993, pp. 409–423.

[Pet81] James Lyle Peterson, *Petri net theory and the modeling of systems*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[P.G94] P.Godefroid, *Partial-order methods for the verification of concurrent systems, a n approach to the state-explosion problem*, Ph.D. thesis, University of Liège, December 1994, Also appeared as LNCS #1032, Springer-Verlag, 1996.

[Pnu79] Amir Pnueli, *The temporal semantics of concurrent programs*, Proceedings of the International Sympoisum on Semantics of Concurrent Computation (London, UK), Springer-Verlag, 1979, pp. 1–20.

[Pnu97] _____, *The temporal of programs*, In Proc. 18th IEEE Symp. Foundations of Computer Science (London, UK), Springer-Verlag, 1997, pp. pages 46–57.

[QS82] J-P. Queille and J. Sifakis, *Specification and verification of concurrent systems in CESAR*, International Symposium on Programming, LNCS 137, Springer Verlag, 1982, pp. 337 – 351.

[Sch02] P. Schnoebelen, *The complexity of temporal logic model checking*, (invited lecture). In Advances in Modal Logic, papers from 4th Int. Workshop on Advances in Modal Logic (AiML'2002), Toulouse, France World Scientific. A preliminary version is available at http://www.lsv.ens-cachan.fr/Publis/ (2002).

[Val92] Antti Valmari, *A stubborn attack on state explosion*, Formal Methods System Description **1** (1992), no. 4, 297–322.

[Woo90] William G. Wood, *Temporal logic case study*, Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems (London, UK), Springer-Verlag, 1990, pp. 257–263.