Model Checking

Doron Peled

Department of Computer Science
Bar Ilan University
Ramat Gan 52900, Israel

Abstract. Model checking is an effective way of comparing a system description against its formal specification and search systematically for errors. The method is gaining a lot of success by being integrated in the hardware design process, and in development of complicated protocols and software. Being an intractable problem, model checking is the ground for very active research that is inspired by many different areas in computer science.

1 Introduction

Model checking, suggested in the early 80s [4, 7, 15], is the automatic verification of a (model of a) system against its formal specification. Model checking can thus be seen as a more exhaustive validation attempt than testing, and, being automatic, as a more practical method than theorem proving. Being a computationally intricate task, model checking promotes the study of different methods, competing on efficiency in terms of time and memory use.

The goal of model checking is mainly to find errors during the process of system design and development. Although an elevated assurance of correctness is gained when errors are not found by model checking, this does not exclude the existence of errors. Model checking can miss inconsistencies between the checked model and the specification, or report spurious errors, called *false positives*. Errors reported by model checking thus need to be further checked against the actual implementation.

Model checking has some strict limitations. As one verifies a *model* of the system rather than the system *itself*, discrepancies between the two may affect the verification results. Similarly, the properties against which the system is to be checked, written in some mathematical formalism (logic, automata), may not precisely fit the intention of the less formal system requirements, given originally in some natural language. Furthermore, the verification tool itself can be bogus. Nevertheless, one should realize that the goal of model checking is to improve the quality of the system and to search for errors, rather than stamping the system with a seal of absolute guarantee against errors.

Model checking suffers from being an intractable problem. Although there are many cases of applying model checking successfully, there are also many difficult instances where one or more of the existing methods fails to work efficiently. Several techniques for model checking compete on the size of systems that can be verified given the available memory and reasonable execution time. *Explicit state space* methods [19] examine the reachable states using search techniques; the search space represents both the states of the model and the checked property. *Symbolic model checking* [3] uses an efficient data structure called BDD to store sets of states, and progresses from one set of states to another rather than state by state. *Bounded Model Checking* [2] encodes a counterexample as a Boolean formula and uses the power of modern SAT solvers. The latter approach conquers new ground by using the impressive capabilities of SAT solvers enriched with decidable theories in order to verify systems with an infinite state space.

2 Modeling

In order to validate a system using model checking tools, the system has first to be modeled into the internal representation of that tool. (For a direct verification of finite state systems, see [12].) This can be done either manually, or automatically by a compiler. In many cases, the internal representation used by the tool is different from the original description of the system. The reason for this is that internal formalisms of tools often reflect some constraints such as dealing with finite states, and is also related to optimizations available by such tools. Even when the same formalism is both used for defining the system and for verification, the internal representation of the tool may differ from the way the system is operating. Such subtleties must be considered by tool designers as well as users; understanding their effect is important in analyzing the verification results.

There are many ways to represent finite state systems. The differences are being manifested in the kind of observations that are made about the possible executions and the way the executions are related to one another. The modeling also affects the complexity of the verification; it is tempting to use a modeling formalism that is very expressive, however, there is a clear tradeoff between expressiveness and complexity.

In order to model systems, we will assume some first order language $\mathcal{F}$. A *transition system* $\langle V, \mathcal{S}, I, T \rangle$ has the following components:

V A finite set of *variables*. The variables describe storage elements, including communication media (message queues) and program counters pointing to the current location in the code of each process. Each variable has some given finite state domain.

$\mathcal{S}$ A set of *states*. Each state is an assignment from variables to their respective domain. We will denote the fact that a state satisfies a first order formula φ from $\mathcal{F}$ by $s \models \varphi$.

I is a unquantified first order formula of $\mathcal{F}$ called the *initial condition*. A state $s \in S$ is initial if $s \models I$.

T is a finite set of *transitions*. Each transition $\tau = en_\tau \rightarrow f_\tau$ in T consists of (1) an *enabling condition* en_τ, which is an unqauntified first order formula of $\mathcal{F}$, and (2) a multiple assignment f_τ of the form $(v_1, v_2, \ldots v_n) := (e_1, e_2, \ldots e_n)$, where v_i are variables from V, and e_i are expressions of $\mathcal{F}$.

An *execution* of a transition system is a *maximal* finite or infinite sequence of states $s_0, s_1, \ldots$ satisfying the following conditions:

- $s_0 \models I$.
- For each $i \geq 0$, there is some transition $\tau \in T$ such that $s_i \models en_\tau$, and s_{i+1} is obtained from s_i by applying the multiple assignment $f_\tau = (v_1, v_2, \ldots v_n) := (e_1, e_2, \ldots e_n)$. That is, one first calculates *all* the expressions, then one performs the assignments (the order is now unimportant). One often denotes this by the notation $s_{i+1} = s_i[x_1/e_1, x_2/e_2, \ldots, v_n/x_n]$.
- Since an execution is maximal, it can be finite only if in the last state no transition is enabled. It is sometimes convenient to work only with infinite executions; then the last state in a finite execution is repeated indefinitely.

The use of the multiple assignments may seem questionable at first, as transitions often represent small (atomic) actions of the system. However, one should realize that even in a simple machine language instruction of a computer, it is often the case that two objects are changing their values: usually at least one memory place or a register or a flag, and in addition the program counter.

Example Let I be $c = a \wedge d = b \wedge e = 0$ and $T = \{\tau_1 : c > 0 \rightarrow (c, e) := (c-1, e+1), \tau_2 : d > 0 \rightarrow (d, e) := (d-1, e+1)\}$.

This transition system adds to e the value of $a + b$ (the variables a and b do not change during the execution). The execution only terminates when both c and d (with initial values as a and b, respectively) both reach 0.

Assume that initially $a = 2$ and $b = 1$. There are multiple executions (in this case, exactly three executions). At the initial state, both τ_1 or τ_2 are enabled. If τ_1 is picked up, then again there is a choice between τ_1 and τ_2. Such a choice of several transitions from the same state is called a *nondeterministic choice*. One of the executions, where τ_1 is chosen first, then τ_2 and then τ_1 again is as follows:

$$\begin{aligned}
s_0 &: \langle a = 2, b = 1, c = 2, d = 1, e = 0\rangle \\
s_1 &: \langle a = 2, b = 1, c = 1, d = 1, e = 1\rangle \\
s_2 &: \langle a = 2, b = 1, c = 1, d = 0, e = 2\rangle \\
s_3 &: \langle a = 2, b = 1, c = 1, d = 0, e = 3\rangle
\end{aligned}$$

This execution model, called the *interleaving model*, can be used to represent sequential, as well as concurrent executions. For concurrent systems, the transitions of all the processes are put together in the same set of transitions T. When several transitions can be executed concurrently (i.e., independently), they are enabled from the same state and by nondeterministic choice they will appear as if executed one after the other in any order (being independent of each other, the executing of one would not disable the other(s)). Nondeterminism can represent in the interleaving model two different things: (1) the selection by some process between different transitions that are enabled at the same time, and (2) the different relative orders between the execution of independent transitions of different processes.

For an intuitive explanation of the interleaving view, assume that the order of occurrences of transitions represent the moment when their effect is taking place. (If there is no single instance where this happens, then the transition need to be further partitioned, as discussed below.) If simultaneously terminating events is possible, their execution order is indeed not important and their relative effect commutes.

There are other models for concurrency that represent concurrent occurrence in a more intuitive manner: the *partial order semantics* models concurrent occurrences of transitions as unordered, while sequentially occurrences are ordered with respect to each other. It still allows nondeterministic choice, but distinguish it from the choice of order between events. The use of this model comes at a high complexity price for model checking [1, 20]. For most purposes, the interleaving model is sufficient and acceptable for modeling concurrent systems.

Example. Consider a partial solution to the mutual exclusion problem as follows:

```
P0:: L0:while true do          P1:: L1:while true do
     NC0:wait (Turn=0)              NC1:wait (Turn=1)
     CR0:Turn=1 od                  CR1:Turn=0 od
```

This little concurrent program, with two processes, can be modeled using the following six transitions:

τ_0: $pc_0 = L_0 \rightarrow pc_0 := NC_0$

τ_1: $(pc_0 = NC_0 \wedge Turn = 0) \rightarrow pc_0 := CR_0$

τ_2: $pc_0 = CR_0 \rightarrow (pc_0, Turn) := (L_0, 1)$

τ_3: $pc_1 = L_1 \rightarrow pc_1 := NC_1$

τ_4: $(pc_1 = NC_1 \wedge Turn = 1) \rightarrow pc_1 := CR_1$

τ_5: $pc_1 = CR_1 \rightarrow (pc_1, Turn) := (L_1, 0)$

The initial condition I is$Turn = 0 \vee Turn = 1$

Given a transition system, one can define its *state graph*, which is a graph of states reachable from any of its initial state. Each directed edge can be annotated by the transition that is executed to transform its incoming state into its outgoing state. The state graph for the above transition system is given in Figure 1. Initial states are marked with transitions with (nonedges) inwards pointing arrows. The executions of the transition system are then the maximal sequences that start with the initial states.

The modeling of the system is open for many modeling decisions. By making the wrong decisions, the verification process is prone to overlooking errors or to reporting spurious false positives. Even with the simple example of mutual exclusion above, one already has a nontrivial modeling choice: the *wait* statement, as modeled, allows progress exactly when the value of the Turn variable is set by the other process (or initially) to the id of the current process. This models a situation where a process can be preempted until some condition occurs. The alternative is to model the wait statement as a *busy waiting*, looping until the waited condition holds; we then need to add two transitions:

$\tau_1{}'$: $(pc_0 = NC_0 \wedge Turn = 1) \rightarrow pc_0 := NC_0$

$\tau_4{}'$: $(pc_0 = NC_1 \wedge Turn = 0) \rightarrow pc_1 := NC_1$

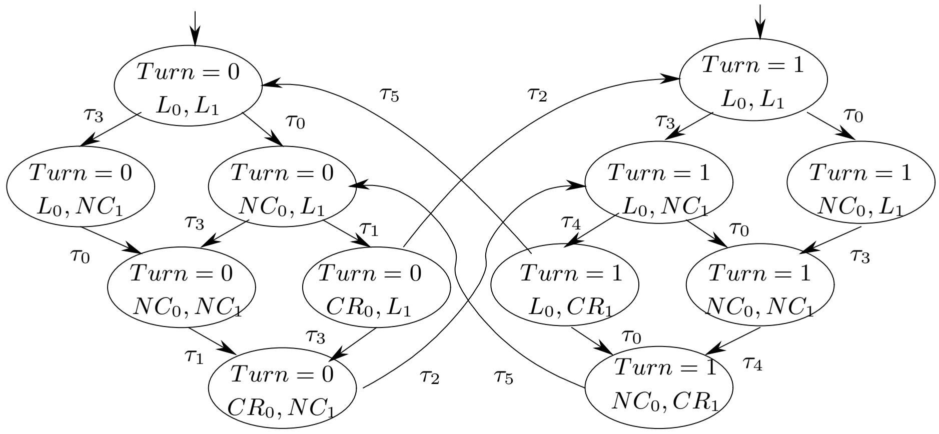

Fig. 1. State graph for mutual exclusion example

The state graph is now changed accordingly, with a self loop around the states $\langle Turn = 0, L_0, NC_1\rangle$ and $\langle Turn = 1, NC_0, L_1\rangle$. This change affects the properties of the model. Under the original model it holds that for every execution, both processes alternatively enter their critical section infinitely often in alternation (this only works because both are always interested in entering their critical section; this specific mutual exclusion solution has the defficiency that if a process does not want to enter its critical section, the other process will also be eventually blocked). With the busy waiting model, entering the critical section is not guaranteed even once. Now, one may consider an execution where a process is waiting for its turn forever, although enabled, as “unfair”. Indeed, a notion of fairness, which will be described later, can be added to the model, ruling out some executions. With such a fairness assumption, an infinite busy waiting loop may be ruled out, maintaining again the property that the processes enter their critical section infinitely often.

An important modeling decision is on the level of *atomicity* of the transitions. According to the interleaving model, at each state, an enabled transition is selected for execution and is performed entirely (atomically), without being able to observe intermediate changes. If the level of atomicity is too fine, we may needlessly increase the size of the state space and the memory required for performing the verification. If the atomicity is too coarse, there may be observable state changes that are not captured by the transition system, including additional interleaving that may be missed. Again, it is the job of the person performing the modeling to make the right choice.

Example. Two processors want both to increment a shared variable x by 1. Modeling this increment as a single atomic transition may reflect the behavior of the actual system if, e.g., the variable x is implemented as a register (this is possible, e.g., in the C language). On the other hand, it is also possible that x stored in some physical memory location is being first copied into an internal register (in each process, separately), which is incremented before the new value is being copied back to the memory location that holds x. In this case, if the initial value is 0, both processes will read the value 0, store it, increment it, and store 1 back to x; a loss of one increment.

Generating the state graph from the transition system can be done with a search method, e.g., Depth First Search (DFS) or Breadth First Search (BFS), starting at initial states and moving from one the to another by applying the enabled transitions. The state graph can be used already to check for simple properties. For example, one can check for deadlocks (states where no transition is enabled and that are not *intended* to be terminating), whether some bad states are reachable, or for dead code (code that is never executed).

The explicit state graph can become exponentially larger than the transition system. Consider n processes, each with two states such that local variable x_i is either 0 or 1; this gives us 2^n states. Deadlock detection in concurrent systems is in PSPACE complete. The lower bound can be achieved through a binary search for deadlock states from initial states. In this way, the state graph is not explicitly constructed. The time complexity of this state efficient algorithm is, unfortunately, even worst then performing the explicit search. Different model checking techniques are used to try and avoid the full construction of the explicit state space.

3 Specification

System specification can be seen as a contract between the consumer and the system developer. Natural language is commonly used to describe requirements from the system. It is susceptible to ambiguity. The use of formal specification allows a unique interpretation, as well as developing model checking algorithms. As in modeling, the use of different notations (natural language and a specification formalism) may cause a potential discrepancy.

Although model checking was suggested already in the 80s, there is still no consensus about the choice of specification formalism. One of the reasons for this situation is the clear tradeoff between expressiveness of the formalism and the complexity of the analysis. The natural tendency to allow a very expressive formalism is countered by the need to provide efficient algorithms for checking properties of the system.

One misleading measurement of complexity of the analysis is with respect to the size of the specification. Indeed, this measure is meaningful within a fixed specification formalism. However, it is often the case that when one formalism allows writing some more compact specification for the same property, the complexity of model checking properties in that latter formalism is correspondingly higher. Thus, it is often a mistake to select a formalism because it allows a much more compact representation of properties, or to select another formalism since the model checking problem is of relatively lower complexity.

Within the interleaving semantics there is an important choice: according to the *linear* view, we are interested in the properties that are common to all the system executions (but we may restrict the executions using some semantic constraints such as fairness assumptions, to be discussed later); according to the *branching* view, we put all the interleaving in a branching tree that is obtained by unfolding the state graph from the initial state(s) (a state can repeat infinitely many times in the tree, even at the same level). This allows us to observe and reason about the branching points. The difference between these two dichotomies, the linear and the branching view, has resulted a lot of interesting research [6]. Time is usually abstracted away, and the progress from one state to another is not done within equal time intervals.

Two ways of specifying properties according to the linear view is *linear temporal logic* and *Büchi automata*. The use of logic has a declarative nature, while the use automata has a more operational flavor. Engineers use both of these formalisms, although many of them still insist to write the specification in natural language.

3.1 Linear Temporal Logic

Linear temporal logic (LTL) [14] describes the progress of a system along an execution from one "world" (representing a state, when modeling a system) to another. The word "linear" means that the worlds are arranged in a total order. One of these worlds is minimal, while all the states are well founded (appear after a finite number of successors) from that state. For simplicity, we will assume that all executions are infinite, extending finite executions by repeating the last state forever.

We first define linear temporal logic. A linear structure $\langle W, R, w_0, P, L\rangle$ consists of the following components:

W An infinite set of worlds.
$R \subseteq W \times W$ is a well founded total order between the worlds.
$w_0 \in W$ is the minimal world according to the R order.
P is the finite set of *labels* for worlds. These can be just program states, where each world is mapped to exactly one state. More conveniently, instead of mapping the worlds to states directly, we can map them to Boolean propositions representing properties of the state (p can represent e.g., the property $x > y$). We will assume that each world is labeled with some subset of propositions from P.
$L : S \mapsto 2^P$ is the mapping from worlds to sets of labels.

A linear structure can then seen as a sequence $w_0 w_1 w_2 \ldots$. In our context, it represents the states in an execution of the modeled system, according to the linear interleaving semantics interpretation.

The syntax of temporal logic adds, on top of the propositional operators $\neg$, $\wedge$, $\vee$, some *modal* operators that describe how the behavior changes with time: $\bigcirc$ - nexttime, $\Diamond$-eventually, $\Box$-always, $\mathcal{U}$-until. The syntax of the logic is defined as follows, where $p \in P$ is a proposition:

$$\varphi ::= p \,|\, \neg\varphi \,|\, (\varphi \vee \varphi) \,|\, (\varphi \wedge \varphi) \,|\, \bigcirc \varphi \,|\, \Diamond\varphi \,|\, \Box\varphi \,|\, (\varphi \mathcal{U} \varphi)$$

The semantics is defined recursively on suffixes of the model (sequence) σ, where σ^i represents the suffix starting at the ith world (with $\sigma = \sigma^0$). The first state of a suffix σ^i will be denoted w_i.

- For $p \in P$, $\sigma^i \models p$ iff $p \in L(w_i)$.
- $\sigma^i \models (\varphi \vee \psi)$ iff $\sigma^i \models \varphi$ or $\sigma^i \models \psi$.
- $\sigma^i \models (\varphi \wedge \psi)$ iff $\sigma^i \models \varphi$ and $\sigma^i \models \psi$.
- $\sigma^i \models \neg\varphi$ iff $\sigma^i \not\models \varphi$.
- $\sigma^i \models \Diamond\varphi$ iff there exists j such that $j \geq i$ and $\sigma^j \models \varphi$.
- $\sigma^i \models \Box\varphi$ iff for all j such that $j \geq i$ it holds that $\sigma^j \models \varphi$.
- $\sigma^i \models (\varphi \mathcal{U} \psi)$ iff there exists j such that $j \geq i$ and $\sigma^j \models \psi$ and for all k such that $i \leq k < j$ it holds that $\sigma^j \models \varphi$.

Finally, we denote $\sigma \models \varphi$ iff $\sigma^0 \models \varphi$.

Thus, $\sigma \models \Diamond\varphi$ when φ holds for *some* suffix of σ, while dually, $\sigma \models \Box\varphi$ when φ holds for *every* suffix of σ. Let *true* be a shorthand for $(p \vee \neg p)$ for some arbitrary $p \in P$. The operators $\Diamond$ and $\Box$ are not necessary: $\Diamond\varphi$ can be expressed as $true\mathcal{U}\varphi$, and $\Box\varphi$ can be expressed as $\neg(true\mathcal{U}\neg\varphi)$. Sometimes a dual for $\mathcal{U}$, denoted $\mathcal{R}$ (for "release") is added, such that $\varphi R \psi = \neg(\neg\varphi\mathcal{U}\neg\psi)$. Defining termpoal formulas over the suffixes of a sequence σ allows combining the temporal operators. A useful combination is $\Box\Diamond\varphi$, which holds when φ holds for infinitely many suffixes. Dually, $\Diamond\Box\varphi$ means that φ holds for all but a finite number of suffixes. The operator $\bigcirc$ is sometimes removed; this has the effect of not being able to distinguish (by means of LTL properties) between *stuttering* executions, i.e., executions that differ only by the number of consecutive times that the same labeling appear [10, 13].

A system $\mathcal{P}$ satisfies an LTL property φ iff each execution σ of $\mathcal{P}$ satisfies φ. We then write $\mathcal{P} \models \varphi$. It can be the case that neither $\mathcal{P} \models \varphi$ nor $\mathcal{P} \models \neg\varphi$.

3.2 Büchi Automata

In order to specify properties of the system using automata, where the accepted words represent the allowed executions, we cannot use simple finite automata: finite automata define languages over *finite* words, while our systems allow infinite executions. A *Büchi* automaton $\mathcal{A} = \langle S, S_0, \Sigma, \delta, F \rangle$ is a finite automaton over *infinite words.* It contains the following components:

S is a finite set of *states.*
$S_0 \subseteq S$ is the set of *initial states.*
Σ is the finite *alphabet* of the automaton.
$\delta \subseteq S \times \Sigma \times S$ is a nondeterministic transition relation.
$F \subseteq S$ is the set of *accepting states.*

A *run* of an automaton $\mathcal{A}$ is an infinite alternating sequence $s_0\rho_0 s_1\rho_1 \ldots$ such that for each $i \geq 0$, $(s_i, \rho_i, s_{i+1}) \in \delta$. A run is *accepting* if at least one of the states appearing infinitely many times on it is in the set of accepting states F. A Büchi automaton *accepts* (or recognizes) infinite words $\rho_0\rho_1 \ldots$ from Σ^ω such that there exists an accepting run $s_0\rho_0 s_1\rho_1 s_2 \ldots$. The *language* $\mathcal{L}(\mathcal{A})$ of an automaton $\mathcal{A}$ consists of all the words from Σ^ω accepted by $\mathcal{A}$. To reason about the executions of systems, we let $\Sigma = 2^P$.

Deterministic Büchi automata, with a transition function $\delta : S \times \Sigma \mapsto S$, are strictly less expressive than nondeterministic Büchi automata. To see this, consider the language over $\Sigma = \{a, b\}$ with finitely many as. The Büchi automaton in Figure 2 accepts this language, while there cannot be a deterministic automaton for the same language. Note that for a deterministic Büchi automaton there exists exactly one path per each input. Assume there exists such a deterministic automaton. Consider a path containing only bs, denoted b^ω. This path reaches an accepting state from F, since this word is in the language of the automaton. Consider the prefix of this path that ends at the first time that an accept state occurs. Then we append an a to that path and subsequently infinitely many bs. Again this sequence is accepted and hence reach an accepting state after the a, and we can cut it after when reaching that state. By repeating this, we obtain a path with infinitely many as that accepts infinitely often; a contradiction.

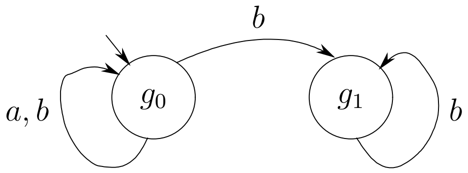

Fig. 2. A nondeterministic automaton accepting words with finitely many as

4 Model Checking

Explicit state model checking is based on a graph theoretic search performed over the combined state space of the system and the checked property.

An LTL formula φ can be translated into a corresponding Büchi automaton such that $\mathcal{L}(\varphi) = \mathcal{L}(\mathcal{A})$. We will later describe such a translation. Each state graph can also be represented as a Büchi automaton $\mathcal{B}$ as well. The set of states $S^{\mathcal{B}}$ are the reachable states $\mathcal{S}$ of the system. For the convenience of dealing only with infinite sequences, if the state graph includes states without successor (deadlocks or termination), then we can add a self loop to that state. It is convenient to add a new initial state ι with edges to all the initial states of the state graph (satisfying the initial condition I). Each edge from a state s to a state s' is labeled $L(s')$, i.e., according to the latter state s'. Now, the accepting states of $\mathcal{B}$ are all the states in $\mathcal{S}$ (namely, each infinite sequence is accepted. This makes the accepting runs of the constructed Büchi automaton $\mathcal{B}$ correspond to the executions of the system.

Now, let $\mathcal{A}_\varphi$ is an automaton with the same language as the LTL property φ, i.e., $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$. Let $\mathcal{B}$ represent the executions of the system. Then for the system to satisfy φ, it is required that

$$\mathcal{L}(\mathcal{B}) \subseteq \mathcal{L}(\mathcal{A}_\varphi) \tag{1}$$

This is because all the sequences of the system (those accepted by $\mathcal{B}$) must satisfy φ, and hence need to be in $\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$. Language containment is hard to perform. It is easier to perform automata intersection. Then instead of translating φ to $\mathcal{A}_\varphi$ and checking for containment, one can translate $\neg\varphi$ to $\mathcal{A}_{\neg\varphi}$ and check whether

$$\mathcal{L}(\mathcal{B}) \cap \mathcal{L}(\mathcal{A}_{\neg\varphi}) = \emptyset \tag{2}$$

holds. Now (1) and (2) are equivalent. To see this, note that $\mathcal{L}(\mathcal{A}_{\neg\varphi}) = \overline{\mathcal{L}(\mathcal{A}_\varphi)}$. In order to check condition (2), we need to know how to intersect two Büchi automata, and how to check for emptiness of a Büchi automaton. Observe that if the intersection is nonempty, it means that there is an execution of the system (represented by the automaton $\mathcal{B}$) that does not satisfy the property φ (is accepted by the automaton $\mathcal{A}_{\neg\varphi}$). Such a *counterexample* can be reported as a result of the model checking.

The intersection, often called the *product*, of two Büchi automata $\mathcal{A}$ and $\mathcal{B}$ is denoted by $\mathcal{A} \times \mathcal{B}$. It satisfies $\mathcal{L}(\mathcal{A} \times \mathcal{B}) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$. Consider first the simple case of intersecting two Büchi automata where $\mathcal{A} = \langle S^\mathcal{A}, S_0^\mathcal{A}, \Sigma, \delta^\mathcal{A}, F^\mathcal{A}\rangle$ and $\mathcal{B} = \langle S^\mathcal{B}, S_0^\mathcal{B}, \Sigma, \delta^\mathcal{B}, S^\mathcal{B}\rangle$. This is a special case of the intersection where *all* the states of $\mathcal{B}$ are accepting, as in the above model checking case. The product simulates the two automata running synchronously, where each input letter from Σ causes a transition of both components. In this case, the acceptance is decided by the first component of each pair:

- $S^{\mathcal{A}\times\mathcal{B}} = S^\mathcal{A} \times S^\mathcal{B}$.
- $S_0^{\mathcal{A}\times\mathcal{B}} = S_0^\mathcal{A} \times S_0^\mathcal{B}$.
- $\delta^{\mathcal{A}\times\mathcal{B}} = \{((s,r),\alpha,(s',r')) | (s,\alpha,s') \in \delta^\mathcal{A} \wedge (r,\alpha,r') \in \delta^\mathcal{B}\}$.
- $F^{\mathcal{A}\times\mathcal{B}} = F^\mathcal{A} \times S^\mathcal{B}$.

The more general case of intersection, of $\mathcal{A} = \langle S^\mathcal{A}, S_0^\mathcal{A}, \Sigma, \delta^\mathcal{A}, F^\mathcal{A}\rangle$ and $\mathcal{B} = \langle S^\mathcal{B}, S_0^\mathcal{B}, \Sigma, \delta^\mathcal{B}, F^\mathcal{B}\rangle$, is a bit more complicated. Consider two such automata in Figure 3 over $\Sigma = \{a, b, c\}$. The language of the left automaton $\mathcal{A}$ consists of words with infinitely many a's. The language of the right automaton $\mathcal{B}$ consists of words where the occurrences of b and a alternate, starting with a b, and any number of cs can appear between each b and its subsequent a, or after the last b. Clearly, the intersection has infinitely many bs and as alternating, with any number of cs between a b and the subsequent a. The product of the states and transition relation, (ignoring for the moment the acceptance issue, appears in Figure 4. In the figure, only the states reachable from the initial state appear; thus the states (g_0, q_1) and (g_1, q_0) were not included.

A naive attempt to define the accepting states, as those that have *both* an accepting component from $\mathcal{A}$ and an accepting component from $\mathcal{B}$, is incorrect. It requires that acceptance by two automata, running synchronously, is done each time simultaneously. This never happens in our example; thus both (g_0, q_0) and (g_1, q_1) would be nonaccepting (since q_0 and g_1 are nonaccepting components), resulting in, erroneously, the empty intersection. Another naive solution is to allow acceptance when *one* of the components is accepting. This will allow incorrectly accepting sequences where one of the automata accepts only finitely many times if the other still accepts infinitely often. In our example, this will make both states in Figure 4 accepting and thus will accept erroneously also sequences where the number of as and bs is finite; in fact, the language of the incorrectly constructed automaton is equivalent to the language of $\mathcal{B}$.

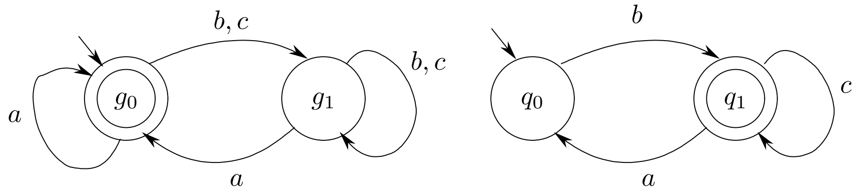

Fig. 3. Two automata to be intersected

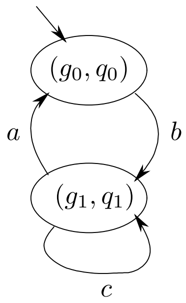

Fig. 4. Product of the states in Figure 3

The solution for defining the acceptance states in the general case of a product of Büchi automata is a bit more involved. In order to guarantee that both components of the product will meet their accepting states infinitely many times, one can observe the acceptance states of the components in alternation; wait for the acceptance of one component, then the other component and so forth. There is no worry about the situation where one component will accept several times before the other one does: if there are infinitely many times each component accepts, then we may decide to follow only a subset of these occurrences.

The construction takes two copies of the product without acceptance, as in Figure 4. The initial states consist of the pairs of initial states of both components of the first copy. When an accepting state of the *first* component in the first copy is reached, the edges out of this state will direct the execution to the corresponding successor state in the *second* copy. Similarly, when an accepting state of the *second* component in the second copy is reached, the edges will direct the execution to the corresponding successor of the *first* copy. Hence, each time acceptance is reached in one of the copies, execution continues with the other copy. Now acceptance is defined by using the accepting states of the first component in the first copy (or, alternatively, the accepting states of the second component in the second copy); it is sufficient to see these states infinitely many times, since this also implies that the other accepting states where found infinitely often (otherwise, we would have gotten stuck in the other copy).

For our example, we make two copies of the automaton in Figure 4 as in Figure 5. The initial state is that of the first (left) copy. Since the state (g_0, q_0) has the accepting component $g_0 \in F^{\mathcal{A}}$, its outgoing edge of b is directed towards the corresponding state $(g_1, q_1)'$ in the second copy rather than towards (g_1, q_1) in the first copy. Similarly, in the second copy we reach $(g_1, q_1)'$, where $q_1 \in F^{\mathcal{B}}$, and thus direct the edges labeled with a and with c towards the corresponding states in the first copy. Note that the state $(g_0, q_0)'$ of the second copy becomes unreachable and can be removed.

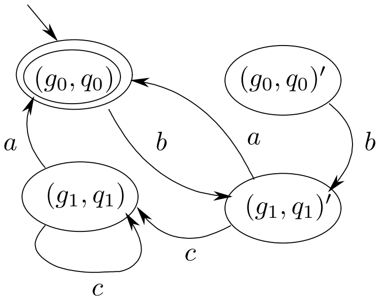

Fig. 5. Intersection of the automata in Figure 3

Formally, given two automata $\mathcal{A}$ and $\mathcal{B}$ as above, both with nontrivial accepting states, the product automaton is the following:

- $S^{\mathcal{A}\times\mathcal{B}} = S^{\mathcal{A}} \times S^{\mathcal{B}} \times \{0, 1\}$.
- $S_0^{\mathcal{A}\times\mathcal{B}} = S_0^{\mathcal{A}} \times S_0^{\mathcal{B}} \times \{0\}$.
- $((g, q, i), \alpha, (g', q', j)) \in \delta^{\mathcal{A}\times\mathcal{B}}$ iff $(g, \alpha, g') \in \delta^{\mathcal{A}}$ and $(q, \alpha, q') \in \delta^{\mathcal{B}}$ and exactly one of the following cases holds:

1. $i = 0$, $g \in F^{\mathcal{A}}$ and $j = 1$ [move from 1st to 2nd copy],
2. $i = 1$, $q \in F^{\mathcal{B}}$ and $j = 0$ [move from 2nd to 1st copy],
3. both case 1 and case 2 do not hold and $i = j$.

- $F^{\mathcal{A}\times\mathcal{B}} = F^{\mathcal{A}} \times S^{\mathcal{B}} \times \{0\}$.

It is easy to extend this construction to the product of n automata.

The next problem is to check the emptiness of a Büchi automaton. Observe that each infinite sequence containing only finitely many states has a suffix that includes only states that appear infinitely many times. This means that in the state graph of the automaton, these states are reachable from each other, i.e., they appear in a *strongly connected component* of such a graph, possibly part of a bigger strongly connected component. It is sufficient then to construct the a maximal strongly connected components, as in Tarjan's algorithm [17], and check weather there is such a component, reachable from some initial state, and which includes some accepting state. This is a necessary and sufficient condition for having an accepting sequence of the Büchi automaton: if such a reachable component exists, one can easily construct a path from some initial state to an accepting state within the component, and then a cycle from that state to itself. This also means that if there is an accepting execution of the Büchi automaton, then there is one that consists of a finite prefix and a recurring cycle of states. Such a sequence is called *ultimately periodic* or *lasso shaped.* This means that in case model checking of LTL or Büchi properties returns with finding a discrepancy between the model of a system and the specification, the counterexample can always be given as an ultimately periodic sequence. This can be presented in a finitary way, containing two finite components: a prefix and a periodic part.

An alternative algorithm for checking emptiness of a Büchi automaton is based on *double DFS* [5, 9]. Instead of looking for strongly connected components, one searches directly for an ultimately periodic sequence consisting of a prefix followed by a cycle. The first DFS discovers the states of the automaton. Whenever it *backtracks* to an accepting state (this means that this is the *last* time that this node is kept in the search stack of the first DFS), then the second DFS is called. The second DFS looks for a path that terminates in a state that is already in the search stack of the first DFS. If such a path is found, then a counterexample can be reported: the first part is the prefix of that search stack until that state; the periodic part is the content of the stack of the first DFS from that state, concatenated with the content of the stack of the second DFS. The first and second DFS work as coroutines: if the second DFS fails to find the required path, the first DFS continues its search from the point it has stopped. When the second DFS is called again, it maintains its hash table from the previous call, hence if a state that was accessed by it in some previous call appears again, there is backtracking, rather than re-exploring states from there again.

The search for counterexamples using double DFS is often called *on the fly* model checking. In many cases, when there is an error, it appears in many executions and the search is found much earlier before completing the search of the entire combined graph of the system and property. Another advantage of this algorithm is that it is relatively efficient on memory use: one only needs to keep the two DFS stacks and a hash table, which records whether a state has already participated in the first or second (or both) DFS. In particular, there is no need to store the edges.

Model checking for LTL properties can be done in space polynomial in the size of both the property and of the transition system [19, 16]. To achieve this, one does not construct first the Büchi automata for the state graph of the system and for the translation of the property. Rather, one performs a binary search through their combined state space for an ultimately periodic sequence. To perform that, one needs to keep some states of the combined state space on a stack, and to be able to check whether one such state is a successor of another.

Bounded model checking [2] avoids performing the search directly on the search space. Instead, it encodes the existence of an ultimately periodic cycle as a Boolean formula. Then SAT solving is used to check satisfiability of this formula. Since SAT solvers require a fixed formula, it is important to establish a good estimate (an overapproximation) on the size of the combined state space.

5 Translating from LTL to Büchi automata

The translation that will be described here allows converting any LTL formula into a Büchi automaton such that both define the same language of executions [8]. This immediately means that Büchi automata are at least as expressive as LTL. The converse does not hold: there are languages expressed using Büchi automata that do not have an LTL property with the same language. A classical example [21] is the language where p holds in all the even states (in the odd states, p may hold or not; if p must further not hold in the odd states, we can express this in LTL). The class of LTL formulas is in fact equivalent to a subset of the Büchi automata that are called *noncounting*. Another characterization of such languages is using star-free regular expressions that include concatenation, complementation and union but not the recurrence operators. A third characterization for the languages of LTL formulas is as the languages expressed using first order monadic logic over linear structures; the languages of Büchi automata are as expressive as second order monadic logic. For a survey on these formalisms, see [18].

Before making the translation, we first put the translated LTL formula in a normal form. To do that, we first get rid of all the modal operators except $\mathcal{U}$ and $\mathcal{R}$. We then recursively push negation inwards based on the following logical equalities:

- $\neg(\varphi \vee \psi) = (\neg\varphi \wedge \neg\psi)$
- $\neg(\varphi \wedge \psi) = (\neg\varphi \vee \neg\psi)$
- $\neg(\varphi \mathcal{U} \psi) = (\neg\varphi \mathcal{R} \neg\psi)$
- $\neg(\varphi \mathcal{R} \psi) = (\neg\varphi \mathcal{U} \neg\psi)$
- $\neg\neg\varphi = \varphi$.

After this transformation, negations can appear only next to propositional variables. This has the advantage that there is no need to complement automata during the transformation, eliminating an operation that is very expensive.

The conversion will first construct a *generalized Büchi automaton* rather than a Büchi automaton. Such an automaton has several acceptance sets $F_1,\ F_2,\ \ldots, F_n$. In order to accept a run, at least one state out of *each* acceptance set must occur infinitely many times. This is an intermediate representation that is convenient for our translation algorithm. However, it does not provide any new expressive power: a generalized Büchi automaton can be easily translated into a simple Büchi automaton (with one set of accepting states). The construction is, in fact, identical to the construction of a product of n Büchi automata with the same set of states, but each with a different accepting set F_i.

A data structure called *node*, which is gradually transformed into an automaton state, is used by this algorithm. Its structure appears in Figure 6. Such a node contains a set of *incoming* edges. Special value called the *initial marker* is also allowed in that field (visually, this can be denoted as an edge with no predecessor). There are three further fields, each for a set of subformulas of the translated formula:

New Subformulas that need to be satisfied from the current node and have not yet been processed.

Old Subformulas that need to be satisfied from the current node and were already processed.

Next Subformulas that need to be satisfied from the successor nodes to the current one.

We start the algorithm with a node that contains the formula φ to be translated in *New*, and empty fields *Old* and *Next*. The *incoming* field includes exactly the initial marker. Now, a formula is removed from the field *New* and is moved to the field *Old*. Then the current processed node is subject to two kinds of transformations depending on that subformula: (1) a *splitting* transformation, for subformulas of an *or* characteristics, where two copies of the node are generated, with some further subformulas added to the fields *New* or *Next*, or (2) an *evolving* transformation, for the other subformulas, where again the fields *New* or *Next* can gain new subformulas.

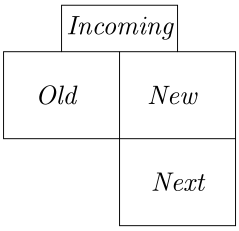

Fig. 6. The data structure for a *node*

The subformulas with *or* characteristics are the following:

$(\varphi \vee \psi)$ Then φ is added to the *New* field in one of the splited copies, and ψ is added to that field in the other copy.

$(\varphi \mathcal{U} \psi)$ Based on the following equivalence

$$(\varphi \mathcal{U} \psi) = (\psi \vee (\varphi \wedge \bigcirc(\varphi \mathcal{U} \psi))$$

we add in one of the copies ψ to *New*, and in the other copy both φ in *New* and $(\varphi \mathcal{U} \psi)$ to *Next*.

$\varphi \mathcal{R} \psi$ Based on the following equivalence

$$(\varphi \mathcal{R} \psi) = (\psi \wedge (\varphi \vee \bigcirc(\varphi \mathcal{R} \psi)))$$

and distributing the $\vee$ over $\wedge$, we add in one copy the subformulas ψ and φ to the *new* field, and in the other copy, we add ψ to *New* and $(\varphi \mathcal{R} \psi)$ to *Next*.

The first two splits in the translation of $(aU(bUc))$ appear in Figure 7.

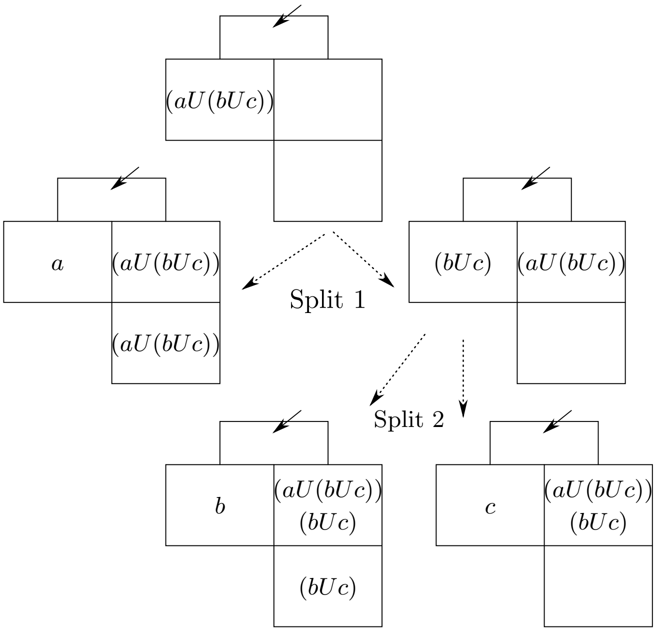

Fig. 7. Two first splits in translating $(aU(bUc))$

The other subformulas are the following:

$(\varphi \wedge \psi)$ Then both φ and ψ are added to the *New* field.

$\bigcirc\varphi$ Then, φ is added to *New*.

p, or $\neg p$ where p is a proposition Then, there is no subformula added to *New* or *Next*.

When the field *New* of a node N becomes empty, we check whether a node of the same values in the fields *Old* and *Next* already exists as an entry in a list of *States*. If it does exist, we just add to the *Incoming* field of the version in *States* the *Incoming* edges of N. Then we may still have some other nodes that are not fully constructed (waiting in the recursion of this algorithm) to which we need to return. If such an entry does not exist in *States*, then we add N to *States*. In this case we also create a new node with the following values:

- The *Incoming* field contains a single edge, from N.
- The *New* field is the copy of the *Next* field of N.
- The *Old* and *Next* fields are empty.

The above description provides the graph structure for the automaton. The elements in *States* are the states, and the *Incoming* edges mark the predecessor relation between the edges. This is still not enough: the splitting due to *until* subformulas takes care that the righthand side of such a subformula can be satisfied, but does not guarantee that this is not delayed indefinitely. In order to guarantee that for an *until* subformulas $(\rho\mathcal{U}\mu)$ the righthand side μ will eventually happen, a generalized Büchi condition is used. We do not need to worry about satisfying $(\rho\mathcal{U}\mu)$ when (1) μ holds from the current state, in this case $(\rho\mathcal{U}\mu)$ is already satisfied, or (2) $(\rho\mathcal{U}\mu)$ is not required. Correspondingly, for each $(\rho\mathcal{U}\mu)$ formula, we construct a generalized Büchi condition consisting of the states where either μ is in *Old* or $(\rho\mathcal{U}\mu)$ is not in *Old*.

We now add a single *initial* state, and connect it to all the states with the initial marker. We also label each edge from s to s' with the propositions and negated propositions that appear in the *Old* field of the target node s'. This is a compact representation of the automaton with edges labeled with both nonnegated and negated propositions, $Q_1 \subseteq P$ and $Q_2 \subseteq P$, respectively. In fact, under this compact representation each edge represents all the edges marked with Q such that $Q_1 \subseteq Q \subseteq (P \setminus Q_2)$. This captures all the cases where propositions in $P \setminus (Q_1 \cup Q_2)$ are either nonnegated or negated.

6 Fairness

Already with the simple mutual exclusion example we saw that if waiting for the $Turn$ variable to be set either to 0 or to 1 by the other process is modeled by busy waiting, progress is not guaranteed. In fact, the other process may be willing to make this change, but is blocked from commencing, since the busy waiting process is exclusively and endlessly being selected to check for the waited condition. This argument demonstrates a problem in the above definition of the interleaving semantics: merely selecting at each state which transition to execute may not always reflect well the behavior of the system. It is possible that additional *fairness* constraints need to be imposed on the executions to reflect the physical behavior in the real world. Although it is usually not desired to include (the usually unavailable) timing information, fairness constraints rule out some situations where some concurrent parts of the system wait indefinitely for others.

There are many commonly used fairness constraints. Four such constraints that are in common use and also demonstrate some available choices as follows:

Weak transition fairness It is impossible for a transition to be enabled forever from some point on without being subsequently executed.

Weak process fairness It is impossible for at least one transition from some process to be enabled forever from some point on forever without at least one of them being subsequently executed. Note that this includes the situation that these may be *different* transitions that are being enabled at different times. This can happen if our model has shared variables on which the execution of these transitions depend; the variables can be changed by other processes.

Strong transition fairness If a transition is enabled infinitely often from some point, then it must be consequently executed (infinitely often).

Strong process fairness If transitions of the same process (not necessarily each time the same one) are infinitely often enabled, then at least one of them will be executed eventually.

The situation in selecting the appropriate fairness constraint is similar to other modeling issues, where an incorrect selection can miss some actual execution or allow spurious ones.

One fairness constraint is *stronger* than another if each execution allowed by the former constraint is also an execution allowed by the latter constraint. In this case, under the stronger fairness, there are less (or the same) executions, which means that more properties can hold. For example, strong transition fairness is stronger than weak transition fairness (hence the words “strong” and “weak”). This is because strong transition fairness rules out more executions than weak transition fairness; it disallows not only situations when a transition is persistently enabled from some point but is never executed, but also the case where a transition is enabled infinitely often, and not continuously, yet is never executed.

Model checking under fairness depends on the fairness constraint assumed. In general, it is easier to perform model checking for weak fairness than for strong fairness [11]. For weak transition fairness, one has to find a reachable strongly connected component with an accepting state such that for each transition, if it is enabled in all the states in the component, then it is executed somewhere inside the component. Similarly, for weak process fairness, for each process, if at least one of its transitions is enabled in each state of the component, then it must be executed inside the component. For strong transition and process fairness the algorithm is more complicated and involves a repeated transformation of the strongly connected components (in quadratic time [11]).

Alternatively, one can express a fairness assumption f as a property ψ_f, then instead of model checking φ, check $\psi_f \rightarrow \varphi$. However, ψ_f here is a very large formula. Moreover, it needs to assert not only on states, but also on transitions, which further reduce the efficiency of model checking.

7 Branching Specification

An alternative logic to LTL is a branching temporal logic called CTL (for Computational Tree Logic) [4]. This logic is defined over trees, representing the branching structure corresponding to the checked system and obtained by unfolding the state graph from an initial state. The syntax of this logic is the following, where φ are *state* properties and ψ are *path properties*:

$$\rho ::= p \,|\, \neg\rho \,|\, (\rho \vee \rho) \,|\, (\rho \wedge \rho) \,|\, E\eta \,|\, A\eta$$

$$\eta ::= F\rho \,|\, G\rho \,|(\rho \mathcal{U} \rho)$$

The semantics is defined over a tree structure $\mathcal{T}$. For state properties, this is defined as follows, where s is some state in the tree, defining a subtree of $\mathcal{T}$ with s being its root:

- For $p \in P$, $s \models p$ iff $p \in L(s)$.
- $s \models \neg\varphi$ iff $s \not\models \varphi$.
- $s \models (\varphi \vee \psi)$ iff $s \models \varphi$ or $s \models \psi$.
- $s \models (\varphi \wedge \psi)$ iff $s \models \varphi$ and $s \models \psi$.
- $s \models E\psi$ iff there exists a path σ in the tree, starting with the state s such that $\sigma \models \psi$.
- $s \models A\psi$ iff for all the paths σ in the tree starting with the state s it holds that $\sigma \models \psi$.

For path formulas, let σ be a path in the tree $\mathcal{T}$:

- $\sigma \models F\varphi$ iff there exists some state s on σ such that $s \models \varphi$.
- $\sigma \models G\varphi$ iff for all the states s on σ it holds that $s \models \varphi$.
- $\sigma \models (\varphi \mathcal{U} \psi)$ iff there exists a state s on σ such that $s \models \psi$, and for all the states r that precede s on σ, $r \models \varphi$.

The state modalities E and A quantify, respectively, about the existence of a path from the current state, or about all the paths from this state. The path modalities G and F correspond, respectively, to the LTL $\Box$ and $\Diamond$. However, with CTL, one must alternate between state modalities and path modalities. Thus, one cannot express properties equivalent to the juxtaposition of two modalities, such as the LTL property $\Box\Diamond\varphi$. An extension of CTL, called CTL* allows that. Now we can express properties not only about *all* the executions but also about *some* of the executions, or combine these two possibilities. Like in LTL, one can define the *release* operator $\mathcal{R}$ as the dual of the *until* operator $\mathcal{U}$, and one can eliminate the operators G and F by the use of *until* (or *release*).

An important property that we may want to express in this way is that whatever choices are made in the execution, there is still a choice to go to a state satisfying p. This is written as $AGEFp$. The predicate p may characterize (hold in exactly) a "home" states, where one can restart the execution of the system after some malfunction has happened. An example for such a home state is in the design of spacecrafts. As a measure of regaining control from unwanted situation, a spacecraft is often designed to always be able to turn the side with light cells towards the sun and be receptive for updates transmitted from earth.

Model checking CTL has the nice property that it can be done in a compositional way directly on the state graph. This algorithm is performed in time linear in the size of the checked CTL property (and also linear in the size of the state space of the system). The states are already marked with the propositional letters. Then the states are marked with a subformula or with its negation depending on whether its subformulas are already marked. Accordingly, we mark the states with $(\varphi \wedge \psi)$ if we already mark them with φ *and* with ψ. Similarly, we mark a node with $(\varphi \vee \psi)$ if it is either marked with φ *or* is marked with ψ. We get negation for free: after taking care of the phase in the algorithm where φ is marked on all the states that satisfy it, the remaining states can be marked with $\neg\varphi$. The rest of the operators can be taken as pairs: EF, AF, EU, AU, ER and AR.

Once we eliminate the G and the F operators, we only need to show how to deal with the *until* and *release* operators. Thus, we will show how to do the model checking for formulas of the form $E\varphi U\psi$ and $E\varphi R\psi$. In the following algorithm, let S_φ be the set of states already marked as satisfying φ. Instead of using a set of transitions, we use a relation $R(s, s')$ between the values of the variables at the current state, and the value of the variables at the next state. For expressing this relation, we need two copies of the variables, e.g., for each variable x in the current state, there will be another variable x' representing its value in the next state. We can translate first each transition

$$\tau = en_\tau \rightarrow (v_1, v_2, \ldots, v_n) := (e_1, e_2, \ldots, e_n)$$

into a relation

$$R_\tau(s, s') := en_\tau \wedge {v_1}' = e_1 \wedge {v_2}' = e_2 \wedge \ldots \wedge {v_n}' = e_n$$

Then,

$$R(s, s') := \bigvee_{\tau \in T} R_\tau(s, s')$$

```
procedure CheckEU($S_\varphi, S_\psi$)
$Q := \emptyset$; $Q' = S_\psi$;
while $Q \neq Q'$ do
        $Q := Q'$;
        $Q' := Q \cup \{s | s \in S_\varphi \wedge \exists s'(R(s, s') \wedge Q(s'))\}$
end while;
return($Q$);
```

```
procedure CheckER($S_\varphi, S_\psi$)
$Q := \mathcal{S}$; $Q' = S_\varphi$;
while $Q \neq Q'$ do
        $Q := Q'$;
        $Q' := Q \cap \{s | s \in S_\psi \vee (\exists s'(R(s, s') \wedge Q(s')))\}$
end while;
return($Q$);
```

8 Symbolic Model Checking and BDD

The use of an explicit graph representation for model checking has the deficiency that nodes in the search are individually stored and analyzed. Symbolic model checking based on the BDD data structure [3] allows storing sets of states in an efficient manner as a directed acyclic graph. This data structure allows performing logical operations efficiently on sets of states, rather than state by state, including the calculation of the predecessors to a set of states according to the transition relation of the system.

An ordered Binary Decision Diagram (OBDD, but the "O" is often omitted) is a directed acyclic graph with a single root, representing a Boolean expression. Each nonleaf node has exactly two successors and is labeled with a Boolean variable. There is a total order between the variables so that they appear along each path from the root to the leafs in the same order; however, not all variables must appear in every such path. The leafs represent the Boolean values 0 (false) and 1 (true). Each nonleaf node has exactly two ordered edges: with the left edge marked 0 and the right marked 1.

A BDD represents a Boolean function over the variables appearing in its nonleaf nodes. To find out the Boolean value under some particular assignment, one has to follow a path in the graph from the root to a leaf according to the assignment: from every node x with value x that appears in the path, one must go to the left if x is 0 and to the right if x is 1. If x does not appear in the path, then the returned value under that path is independent on whether x is 0 or 1.

To take advantage of the BDD representation, one needs to minimize it to a compact form. This is done by repeatedly using the following rules:

- If multiple edges point to isomorphic subgraphs, then one need to keep only one copy of this subtree, redirecting these edges towards a single copy.
- If the left *and* the right outgoing edges of a node point to isomophic subgraphs, one can remove that node and direct its incoming edges *directly* towards a single copy.

The left part of Figure 8 shows a BDD that is a full binary tree. The right part of Figure 8 shows the result of applying the reduction to it.

One can now define Boolean operators on BDDs. After each such operation, the BDD is reduced as explained above.

- $f[0/x]$ and $f[1/x]$, restricting the BDD to x having the value 0 or 1, respectively. For the former operation, incoming edges to each node marked

x are redirected to the left outgoing edge of x and x is removed. For the latter operation, the symmetric transformation is performed with the right outgoing edge.
- $\exists x\varphi$ is calculated as $\varphi[0/x] \vee \varphi[1/x]$, i.e., two applications of the above operation.
- Applying some Boolean operator $f\#g$ between two BDDs f and g. This uses Shannon expansion:

$$f\#g = ((\neg x \wedge (f[0/x]\#g[0/x])) \vee (x \wedge (f[1/x]\#g[1/x]))).$$

Effectively this means that the calculation can be done recursively on the two BDD graphs for f and g. At the level of the leafs, one applies directly the Boolean function $\#$ to calculate the value. At a level of variable x, this results in a left edge to the result of the recursive application on the left subgraph, and a right edge to the result of the recursive application on the right subgraph. It may happen that (due to reduction) a subgraph rooted with a node x exists in one of the BDDs, either f or g, but not in the other BDD. This means that this subgraph does not depend on the value of x, and the next available node can be used instead. Since during this construction the same subgraph can be constructed again and again, one uses dynamic programming *memoizing* to prevent an explosion of the size of the intermediate (before reduction) graph representation.

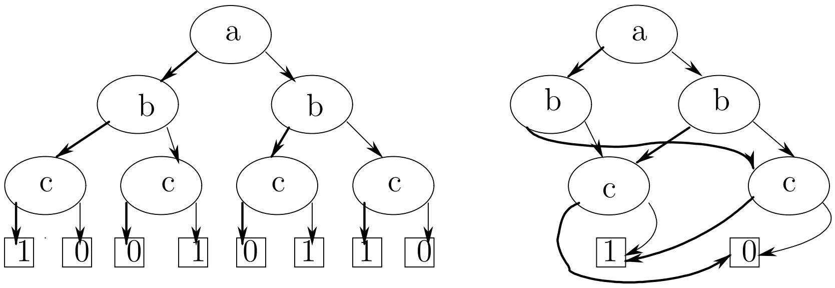

Fig. 8. A BDD unreduced (left) and reduced (right)

Now the entire CTL model checking can be done with BDDs. The above two procedures *CheckEU* and *CheckER* can be rewritten such that each set (in particular, the set variables Q and Q') is represented as a BDD. The transition relation $R(s, s')$ is also represented as a BDD between a copy of the variables (say x, y and z) at the current state, and a tagged copy of the variables (say x', y' and z') at the next state. Then, all the operations are done between BDDs.

Acknowledgment. I would like to thank Angelika Mader and Gal Katz for reading the manuscript and providing many insightful remarks.

References

1. R. Alur, K. McMillan, D. Peled, Deciding global partial order properties, ICALP 1998, Aalborg, Denmark, Lecture Notes in computer science 1443, 41-52.
2. A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu, TACAS 1999, Amsterdam, The Netherlands, Lecture Notes in Computer Science 1579, Springer, 193-207.
3. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Huang, Symbolic model checking: 10^{20} states and beyond, LICS 1990, Philadelphia, PA, USA. 428-439.
4. E. M. Clarke, E. A. Emerson, Design and synthesis of synchronization skeletons using branching-time temporal logic, Logic of programs, 1981, Lecture Notes in Temporal Logic 131, Springer, Yorktown Heights, NY, USA, 52-71.
5. C, Courcoubetis, M. Y. Vardi, P. Wolper, M. Yannakakis, Memory efficient algorithms for the verification of temporal properties, CAV 1990, New Brunswick, NJ, USA, Lecture Notes in Computer Science 531, Springer, 233-242.
6. E. A. Emerson, C.-L. Lei, Modalities for model checking: branching time logic strikes back. Science of Computer Programming 8(3), 1987, 275-306.
7. E. A. Emerson, E. M. Clarke, Characterizing correctness properties of prallel programs using fixpoints, ICALP 1980, Noorweijkerhout, The Netherlands, Lecture Notes in Computer Science 85, Springer, 169-181.
8. R. Gerth, D. Peled, M. Y. Vardi, P. Wolper, Simple on-the-fly automatic verification of linear temporal logic, PSTV 1995, Warsaw, Poland, 3-18.
9. G. J. Holzmann, D. Peled, M. Yannakakis, On nested depth first search, 2nd workshop on the SPIN verification system 1996, Dimacs series in discrete mathematics, volume 32, 1997, 23-32.
10. L. Lamport, What good is temporal logic, Information processing 83, IFIP 9th world congress, Paris, France, 657-668.
11. O. Lichtenstein, A. Pnutli, Checking that finite state concurrent programs satisfy their linear specification, POPL 1985, New Orleans, LO, USA, 97-107.
12. D. Peled, M. Y. Vardi, M. Yannakakis, Black box checking, FORTE 1999, Beijing, China, 225-240.
13. D. Peled, T. Wilke, Stutter-invariant temporal properties are expressive without the next-time operator, Information Processing Letters 63(5), 1997, 243-246.
14. A. Pnueli, The temporal logic of programs, FOCS 1977, Providence, RI, USA, 46-57.
15. J.-P. Quille, J. Sifakis, Specification and verification of concurrent systems in CESAR, Symposium on Programming 1982, Torino, Italy, Lecture Notes in Computer Science 137, Springer, 337-351.
16. A. P. Sistla, E. M. Clarke, The complexity of propositional linear temporal logic, STOC 1982, San Francisco, CA, USA, 159-168.
17. R. E. Tarjan, Depth-first search and linear graph algorithms, FOCS 1971, East lansing, MI, USA, 114-121.
18. W. Thomas, Automata on infinite objects, Handbook on theoretical computer science, 1991, Volume B, 133-191.
19. M. Vardi, P. Wolper, Automata theoretic techniques for modal logic of programs, STOC 1984, Washington, DC, USA, 446-456.
20. I. Walukiewicz, Difficult configurations - on the complexity of LTrL, Lecture Notes in Computer Science 1443, ICALP 1998, 140-151.
21. P. Wolper, Temporal logic can be more expressive, FOCS 1981, Nashville, TN, USA, 340-348.