Computer-Aided Formal Verification

Sergey Ichtchenko

Mini-project, Michaelmas Term 2024

Question 1

(a) Consider each pair of temporal logic formulae below. Identify which logic each formula is written in and then show whether the pair are equivalent or not.

(i) $(\Box a \lor \Box b) \to \Diamond(a \land b)$ and $\Box(\neg a \lor \neg b) \to (\Diamond\neg a \land \Diamond\neg b)$

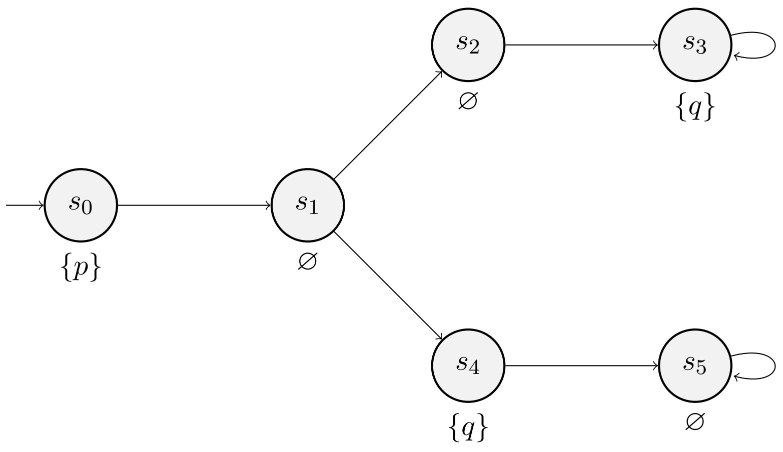
(ii) $\forall\Box(p \to \forall\Diamond\forall\bigcirc q)$ and $\forall\Box(p \to \Diamond\bigcirc q)$

(i) Both of the formulae are in **LTL**. They **are** equivalent. Proof:

$$
\begin{aligned}
&(\Box a \lor \Box b) \to \Diamond(a \land b) = \\
&= \neg(\Box a \lor \Box b) \lor \Diamond(a \land b) = \\
&= (\neg\Box a \land \neg\Box b) \lor \Diamond(a \land b) = \\
&= (\Diamond\neg a \land \Diamond\neg b) \lor \Diamond(a \land b) = \\
&= \Diamond(a \land b) \lor (\Diamond\neg a \land \Diamond\neg b) = \\
&= \Diamond\neg(\neg a \lor \neg b) \lor (\Diamond\neg a \land \Diamond\neg b) = \\
&= \neg\Box(\neg a \lor \neg b) \lor (\Diamond\neg a \land \Diamond\neg b) = \\
&= \Box(\neg a \lor \neg b) \to (\Diamond\neg a \land \Diamond\neg b)
\end{aligned}
$$

$\Box$

(ii) The first formula is in CTL, while the second is in CTL*. They are **not** equivalent. Proof: Consider the following LTS $\mathcal{M}$:



Now, observe that $\mathcal{M} \vDash \forall\Box(p \to \Diamond\bigcirc q)$ since there are two paths starting at s_0:

- $\pi_1 = s_0 s_1 s_2 s_3 s_3 s_3 ...$
- $\pi_2 = s_0 s_1 s_4 s_5 s_5 s_5 ...$
- $\text{trace}(\pi_1) = \{p\}\ \varnothing\ \varnothing\ \{q\}\ \{q\}\ \{q\}\ ...$
- $\text{trace}(\pi_2) = \{p\}\ \varnothing\ \{q\}\ \varnothing\ \varnothing\ \varnothing\ ...$

$\pi_1 \vDash p \to \Diamond\bigcirc q$ since for all $k \geq 1$ we have that $\pi_1[k...] \nvDash p$, and for $k = 0$ there exists $k' = 2$ such that $\pi_1[0...][k'...] \vDash \bigcirc q$ since $\pi_1[0...][2...][1...] \vDash q$. Similarly, $\pi_2 \vDash p \to \Diamond\bigcirc q$ since for all $k \geq 1$ we have that $\pi_2[k...] \nvDash p$, and for $k = 0$ there exists $k' = 1$ such that $\pi_1[0...][k'...] \vDash \bigcirc q$ since $\pi_1[0...][1...][1...] \vDash q$. Thus $\mathcal{M} \vDash \forall\Box(p \to \Diamond\bigcirc q)$.
However, $M \nvDash \forall\Box(p \to \forall\Diamond\forall\bigcirc q)$. Consider the path π_2. $\pi_2 \vDash p$, therefore for the statement to hold we also require that $\pi_2 \vDash \forall\Diamond\forall\bigcirc q$ which implies that $s_0 \vDash \forall\Diamond\forall\bigcirc q$. Again, consider the path π_2 starting from s_0. We need to find a $k \in \mathbb{N}$ such that $\pi_2[k...] \vDash \forall\bigcirc q$. The only way this could be possible is with $k = 1$, as for all other values of k, $\pi_2[k...] \nvDash \bigcirc q$. However, $k = 1$ also does not work, since $\pi_2[1...] \vDash \forall\bigcirc q$ implies that $\pi_2[1] = s_1 \vDash \forall\bigcirc q$ which is false since the path $s_1 s_2 s_3 s_3 s_3 ... \nvDash \bigcirc q$. Thus we have exhausted all possibilties and $M \nvDash \forall\Box(p \to \forall\Diamond\forall\bigcirc q)$. $\Box$

(b) Assuming that a, b, and c are atomic propositions, translate each of the following statements into the temporal logic LTL.

(i) At most two of a, b, and c are ever true simultaneously.

(ii) If a and b are ever true simultaneously then, from that point on, at least one of them is always true at any point.

(iii) c is true at exactly two distinct time steps and these are not consecutive.

(iv) Each of b and c are true infinitely often but the number of times that they are true simultaneously is finite.

(i) $\Box\neg(a \land b \land c)$

(ii) $\Box((a \land b) \to \Box(a \lor b))$

Note: "at least one of them is always true at any point" interpreted as "at any point, either a is true or b is true" and not "either a is true for all points or b is true for all points".

(iii) $(\neg c)\, \mathtt{U}\, (c \land \bigcirc(\neg c \land (\neg c)\, \mathtt{U}\, (c \land \bigcirc\Box\neg c)))$

(iv) $\Box\Diamond b \land \Box\Diamond c \land \Diamond\Box\neg(b \land c)$

(c) For each of the properties expressed in part (b), state whether or not it belongs to these linear-time property classes: invariant, safety, liveness.

Property	Invariant?	Safety?	Liveness?
i	Yes	Yes	No
ii	No	Yes	No
iii	No	No	No
iv	No	No	Yes

Explanation:

Firstly, recall from the lectures that all invariants are safety properties. In addition, safety and liveness properties are disjoint, since safety properties have bad prefixes and liveness properties have the property that any finite prefix can be extended to a word satisfying the liveness property.

Property (i) is an invariant since it checks whether a certain property is true in each individual state (and doesn't need information about other states to determine that). Since all invariants are also safety properties, it is also a safety property. Thus it is also not a liveness property.

Property (ii) is not an invariant as it talks about more than one state. It is a safety property as any violation has a bad prefix (a trace containing $\{a, b\}$ followed by $\varnothing$ at some point). Thus it is also not a liveness property.

Property (iii) is not an invariant since it talks about more than one state. It is not a safety property since the trace $\{c\}\ \varnothing\ \varnothing\ \varnothing$... violates it but does not have a bad prefix, since any of its prefixes can be extended to a satisfying trace by adding $\{c\}$ followed by an infinite sequence of $\varnothing$. It is also not a liveness property, since the prefix $\{c\}\ \{c\}\ \varnothing$ cannot be extended to a satisfying trace since it is a bad prefix containing two consecutive cs.

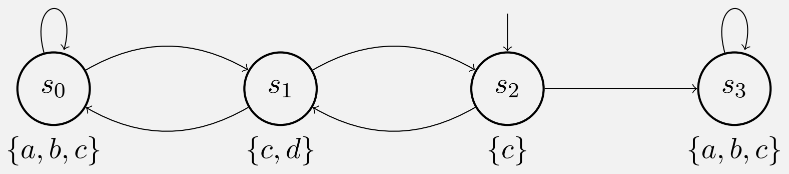
Property (iv) is a liveness property as any finite prefix can be extended to a satisfying word by adding $\{b\}$ and $\{c\}$ but not $\{b, c\}$ to the prefix infinitely many times. It is thus not an invariant or a safety property.

□

Question 2

(a) Illustrate the application of the CTL model checking algorithm to determine whether the LTS $\mathcal{M}$ below satisfies the formula:

$$\phi = \forall\Diamond\neg\exists[\neg d \,\mathtt{U}\, \forall\bigcirc a]$$



The LTS $\mathcal{M}$ does **not** satisfy ϕ.

Step 1: convert the formula to ENF

$$\begin{aligned}\phi = \forall\Diamond\neg\exists[\neg d \,\mathtt{U}\, \forall\bigcirc a] &= \\ = \neg\exists\Box\neg(\neg\exists[\neg d \,\mathtt{U}\, \forall\bigcirc a]) &= \\ = \neg\exists\Box\exists[\neg d \,\mathtt{U}\, \forall\bigcirc a] &= \\ = \neg\exists\Box\exists[\neg d \,\mathtt{U}\, \neg\exists\bigcirc \neg a]\end{aligned}$$

Step 2: recursively calculate the satisfying set of ϕ

First, note that our state space is $S = \{s_0, s_1, s_2, s_3\}$. We will calculate the satisfying set of ϕ by working through each part of the formula inside-out.

- $\text{Sat}(a) = \{s_0, s_3\}$
- $\text{Sat}(\neg a) = S \setminus \text{Sat}(a) = \{s_1, s_2\}$
- $\text{Sat}(d) = \{s_1\}$
- $\text{Sat}(\neg d) = S \setminus \text{Sat}(d) = \{s_0, s_2, s_3\}$
- $\text{Sat}(\exists\bigcirc \neg a) = \{s \in S | \text{Post}(s) \cap \{s_1, s_2\} \neq \varnothing\} = \{s_0, s_1, s_2\}$
- $\text{Sat}(\neg\exists\bigcirc \neg a) = S \setminus \text{Sat}(\exists\bigcirc \neg a) = \{s_3\}$
- $\text{Sat}(\exists[\neg d \,\mathtt{U}\, \neg\exists\bigcirc \neg a]) = \text{CheckExistsUntil}(\{s_0, s_2, s_3\}, \{s_3\})$

  To apply the CheckExistsUntil algorithm, we set $T_0 = \{s_3\}$ and repeatedly evaluate $T_i = T_{i-1} \cup \{s \in \{s_0, s_2, s_3\} | \text{Post}(s) \cap T_{i-1} \neq \varnothing\}$ until we find a fixed point.

  Doing this, we get $T_0 = \{s_3\}, T_1 = \{s_2, s_3\}, T_2 = \{s_2, s_3\}$. Thus $\text{Sat}(\exists[\neg d \,\mathtt{U}\, \neg\exists\bigcirc \neg a]) = \{s_2, s_3\}$.

- $\mathrm{Sat}(\exists\Box\exists[\neg d \,\mathtt{U}\, \neg\exists\bigcirc \neg a]) = \mathrm{CheckExistsAlways}(\{s_2, s_3\})$

  To apply the CheckExistsAlways algorithm, we set $T_0 = \{s_2, s_3\}$ and repeatedly evaluate $T_i = T_{i-1} \cap \{s \in \{s_2, s_3\} | \mathrm{Post}(s) \cap T_{i-1} \neq \varnothing\}$ until we find a fixed point.

  Doing this, we get $T_0 = \{s_2, s_3\}, T_1 = \{s_2, s_3\}$. Thus $\mathrm{Sat}(\exists\Box\exists[\neg d \,\mathtt{U}\, \neg\exists\bigcirc \neg a]) = \{s_2, s_3\}$.

- $\mathrm{Sat}(\phi) = \mathrm{Sat}(\neg\exists\Box\exists[\neg d \,\mathtt{U}\, \neg\exists\bigcirc \neg a]) = S \setminus \mathrm{Sat}(\exists\Box\exists[\neg d \,\mathtt{U}\, \neg\exists\bigcirc \neg a]) = \{s_0, s_1\}$

Step 3: conclusion

Observe now that $I = \{s_2\} \not\subseteq \{s_0, s_1\} = \mathrm{Sat}(\phi)$. Hence $\mathcal{M} \not\models \phi$. $\Box$

(b) Illustrate the application of the LTL model checking algorithm to determine whether the same LTS $\mathcal{M}$ above satisfies the formula:

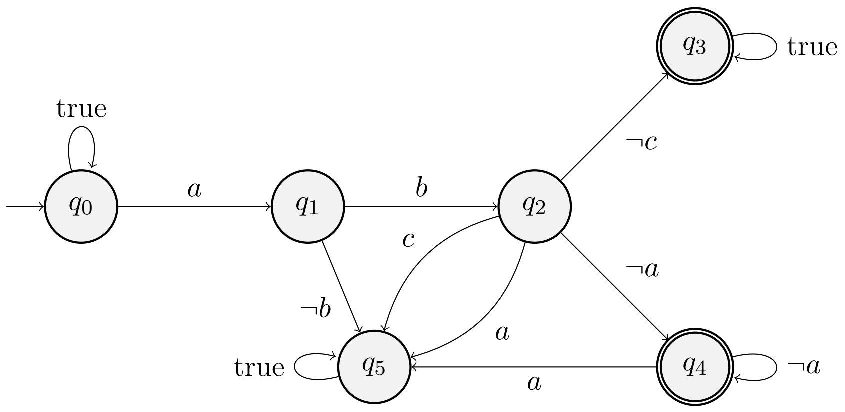
$$\psi = \Box(a \rightarrow \bigcirc(b \rightarrow \bigcirc(c \wedge \Diamond a)))$$

The LTS $\mathcal{M}$ does **not** satisfy the formula ψ.

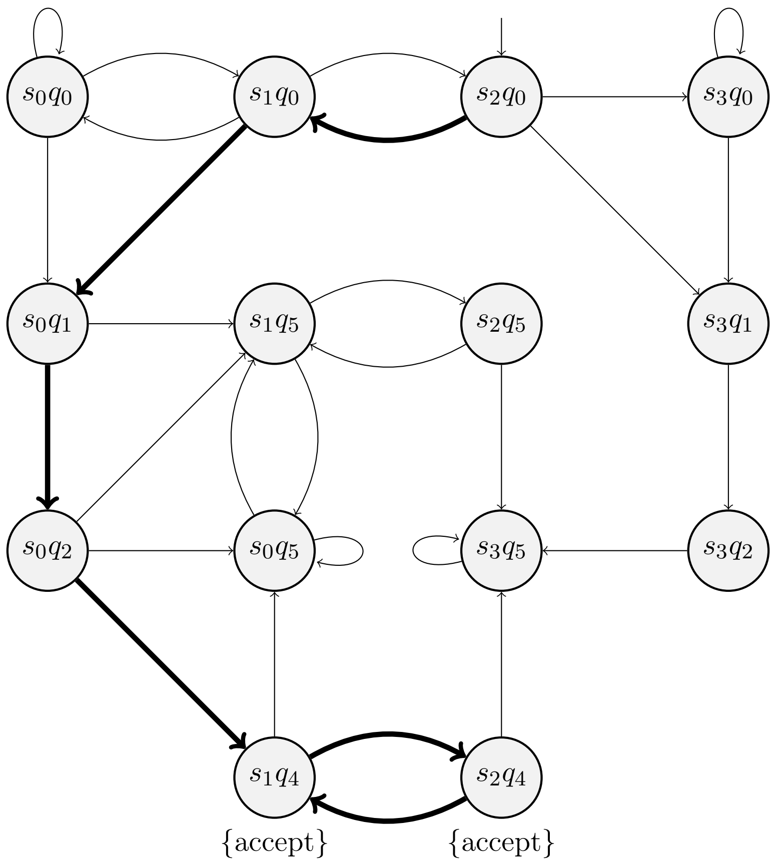
Step 1: negate the formula

$$\begin{aligned}
\psi &= \Box(a \rightarrow \bigcirc(b \rightarrow \bigcirc(c \wedge \Diamond a))) = \\
&= \Box(\neg a \vee \bigcirc(\neg b \vee \bigcirc(c \wedge \Diamond a))) \\
\neg\psi &= \neg\Box(\neg a \vee \bigcirc(\neg b \vee \bigcirc(c \wedge \Diamond a))) = \\
&= \Diamond\neg(\neg a \vee \bigcirc(\neg b \vee \bigcirc(c \wedge \Diamond a))) = \\
&= \Diamond(a \wedge \bigcirc\neg(\neg b \vee \bigcirc(c \wedge \Diamond a))) = \\
&= \Diamond(a \wedge \bigcirc(b \wedge \bigcirc\neg(c \wedge \Diamond a))) = \\
&= \Diamond(a \wedge \bigcirc(b \wedge \bigcirc(\neg c \vee \neg\Diamond a))) = \\
&= \Diamond(a \wedge \bigcirc(b \wedge \bigcirc(\neg c \vee \Box\neg a)))
\end{aligned}$$

Step 2: Construct NBA for $\neg\psi$



Step 3: Construct LTS-NBA product



Step 4: Finding an accepting cycle

We observe that we can find the following accepting cycle in the LTS-NBA product:

$$s_2q_0, s_1q_0, s_0q_1, s_0q_2, s_1q_4, s_2q_4, s_1q_4, s_2q_4, \ldots$$

where the states s_1q_4, s_2q_4 repeat forever. This cycle is highlighted in bold in the above figure.

Thus $\mathcal{M} \not\models \psi$. $\square$

(c) Consider the temporal operator $\Diamond^{[l,u]}$, which imposes lower and upper bounds on the occurrence of the event.

$$\pi \vDash \Diamond^{[l,u]}\phi \text{ iff } \exists k \in \mathbb{N} \text{ such that } k \geq l, k \leq u, \text{ and } \pi[k...] \vDash \phi$$

For each of the two logics, CTL and LTL: (i) explain whether adding $\Diamond^{[l,u]}$ increases expressivity; (ii) describe an appropriate way to extend the existing model checking algorithm to incorporate this operator and analyse the efficiency of the resulting algorithms.

LTL (i)

The operator does **not** increase expressivity in LTL.

For LTL, if u is an integer then we can rewrite the new operator as

$$\Diamond^{[l,u]}\phi = \underbrace{\bigcirc\bigcirc...\bigcirc}_{l}(\phi \vee \underbrace{\bigcirc(\phi \vee \bigcirc(...\phi)}_{u-l}))$$

If $u = \infty$ then we can also rewrite this as

$$\Diamond^{[l,\infty]}\phi = \underbrace{\bigcirc\bigcirc...\bigcirc}_{l}\Diamond\phi$$

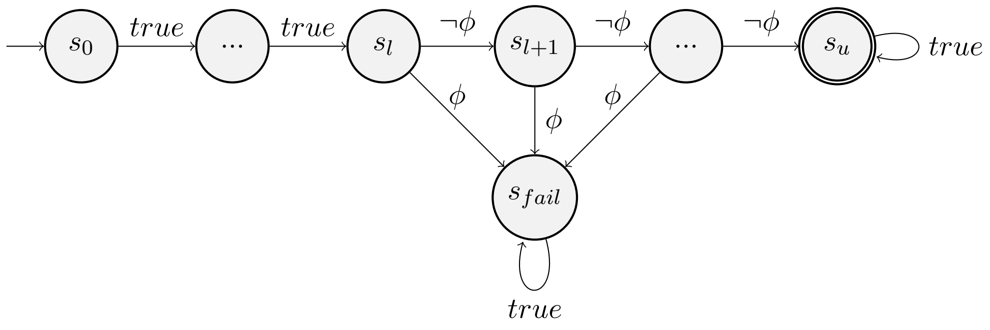
LTL (ii)

In this case, it suffices to convert the given extended LTL formula into the standard LTL using the above conversion and then running the LTL model checking algorithm. However, this may result in a huge number of states in the NBA generated during the model checking phase, as there are LTL formulas ϕ whose NBA is of size $O(2^{|\phi|})$.

An alternative way of constructing the NBA is to notice that

$$\neg\Diamond^{[l,u]}\phi = \underbrace{\bigcirc\bigcirc...\bigcirc}_{l}(\neg\phi \wedge \underbrace{\bigcirc(\neg\phi \wedge \bigcirc(...\phi)}_{u-l}))$$

For this expression alone, we can construct the following NBA:



However, if ϕ is not an atomic proposition but another LTL formula, the proper recognition for $\neg\phi$ and ϕ needs to be inserted where the current $\neg\phi$ and ϕ transitions are, respectively. Thus there are $l + (l - u) \cdot (|\mathcal{A}_\phi| + |\mathcal{A}_{\neg\phi}|)$ states inserted into the NBA, which is exponential in the worst case. Thus, with our construction, the time complexity of the model checking algorithm is exponential in $l - u$.

CTL (i)

The operator does **not** increase expressivity in CTL.

For CTL, if u is an integer then we can rewrite the new operator as

$$\forall\Diamond^{[l,u]}\phi = \underbrace{\forall\bigcirc\,\forall\bigcirc\,...\forall\bigcirc}_{l}(\phi \vee \underbrace{\forall\bigcirc\,(\phi \vee \forall\bigcirc\,(...\phi))}_{u-l})$$

$$\exists\Diamond^{[l,u]}\phi = \underbrace{\exists\bigcirc\,\exists\bigcirc\,...\exists\bigcirc}_{l}(\phi \vee \underbrace{\exists\bigcirc\,(\phi \vee \exists\bigcirc\,(...\phi))}_{u-l})$$

If $u = \infty$ then we can also rewrite this as

$$\forall\Diamond^{[l,\infty]}\phi = \underbrace{\forall\bigcirc\,\forall\bigcirc\,...\forall\bigcirc}_{l}\,\forall\Diamond\phi$$

$$\exists\Diamond^{[l,\infty]}\phi = \underbrace{\exists\bigcirc\,\exists\bigcirc\,...\exists\bigcirc}_{l}\,\exists\Diamond\phi$$

CTL (ii)

Again, we may convert any extended CTL formula to a standard CTL formula using the above rewriting and run the CTL model checking algorithm.

In this case, doing this is straightforward, and we increase the size of ϕ by $l + 3 \cdot (u - l) + 1$. Hence the complexity of the model checking algorithm is linear in u.

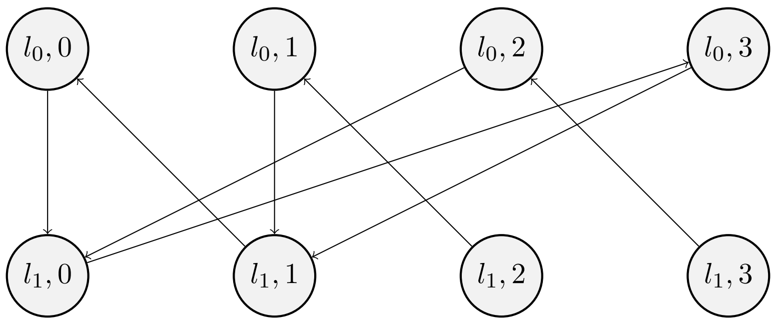
Question 3

Consider the following program.

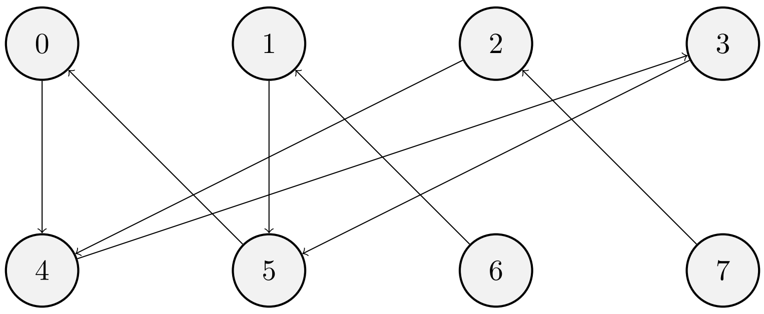
```
while (true){
    x := x mod n;
    x := x+3 mod 2n;
}
```

(a) Fixing $n = 2$, show in detail how the *transition relation* representing this program is represented using each approach as: (i) a Binary Decision Diagram (BDD); and (ii) a symbolic expression to be passed to a SAT or SMT solver, respectively.

Let us first construct the LTS for the program. We define the states of the LTS as pairs of variables. The first variable, l_0 or l_1, determines where in the program we are. l_0 means that the program is *about to* execute (but has not yet executed) the first line `x := x mod n;`, while l_1 indicates the same for the second line `x := x+3 mod 2n;`. The second variable encodes the value of the program variable x. As it is initially undefined, we let the value of x range from 0 to $2n-1$ in the states of the LTS. The LTS for the program is:



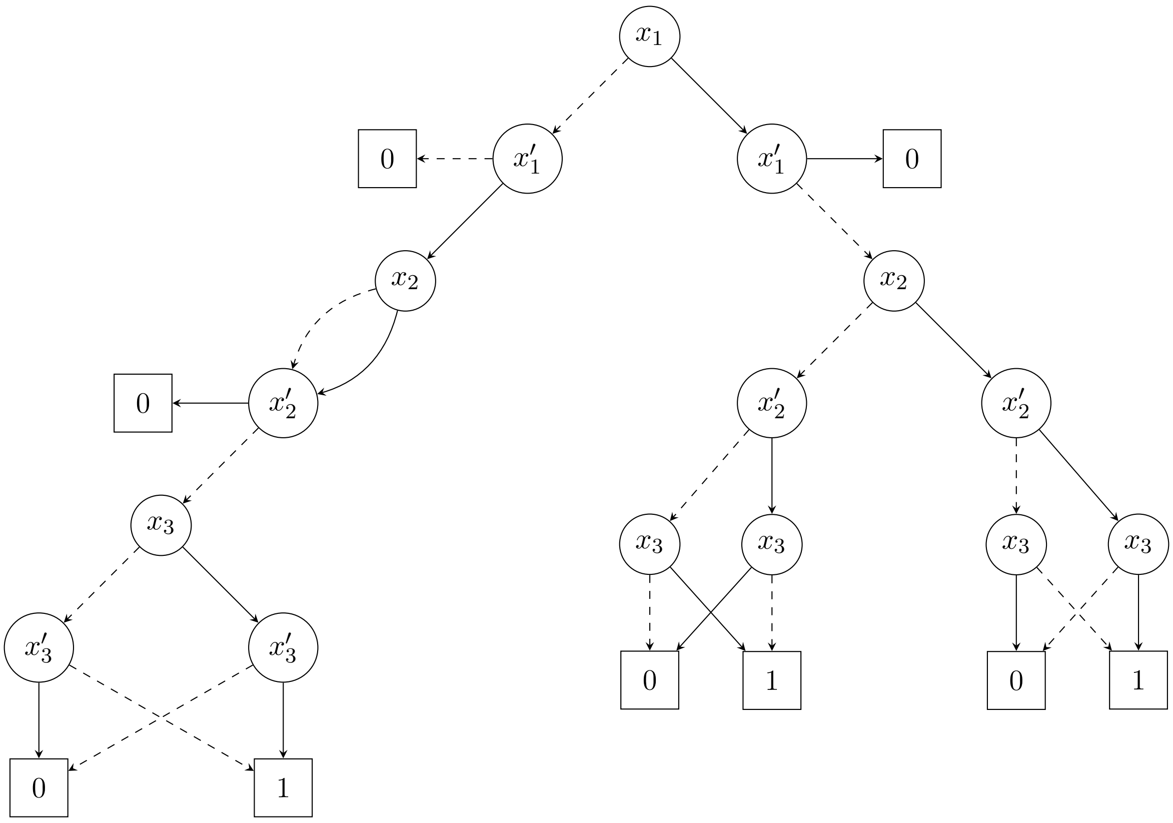
We now relabel the states systematically for our encoding:



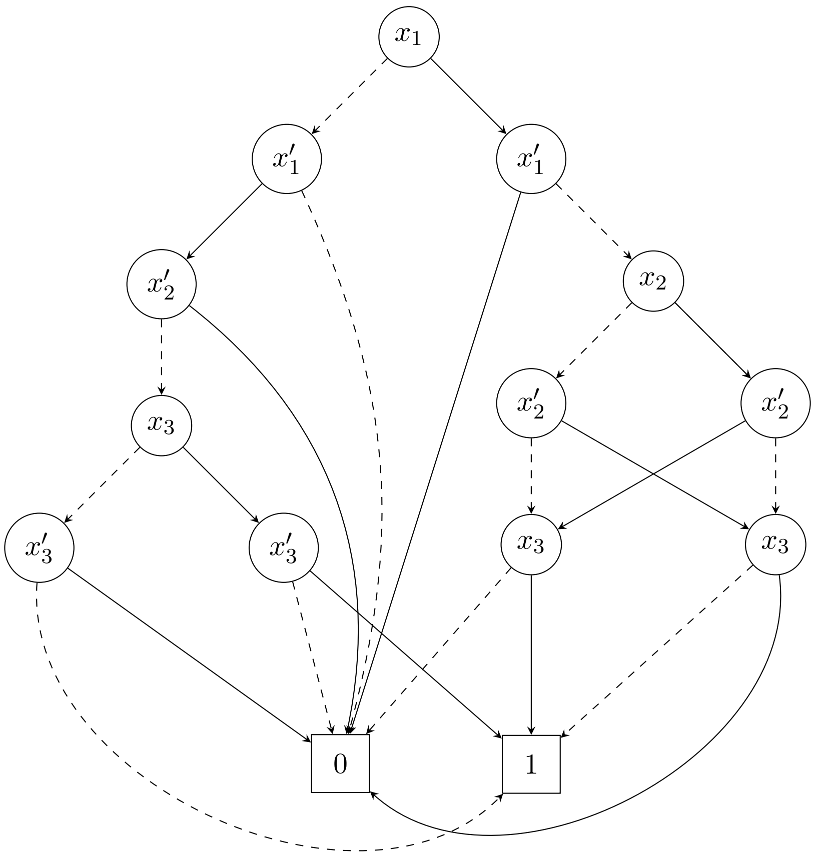
We now define the transitions of the LTS in a table as follows:

Transition	x_1	x_2	x_3	x'_1	x'_2	x'_3	Full expression
$(0,4)$	0	0	0	1	0	0	000100
$(1,5)$	0	0	1	1	0	1	001101
$(2,4)$	0	1	0	1	0	0	010100
$(3,5)$	0	1	1	1	0	1	011101
$(4,3)$	1	0	0	0	1	1	100011
$(5,0)$	1	0	1	0	0	0	101000
$(6,1)$	1	1	0	0	0	1	110001
$(7,2)$	1	1	1	0	1	0	111010

(i) Let us convert this transition relation into a BDD.



By further simplifying the BDD, we arrive at the following:



This is the BDD representing the transition relation of the program. $\square$

(ii) Let us encode the transition relation table as a symbolic expression. We can do this by rewriting the table as a logical proposition in the variables $x_1, x_2, x_3, x_1', x_2', x_3'$.

$$\begin{aligned}
f_\to = &\big(\neg x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_1' \wedge \neg x_2' \wedge \neg x_3'\big) \vee \\
&\vee \big(\neg x_1 \wedge \neg x_2 \wedge x_3 \wedge x_1' \wedge \neg x_2' \wedge x_3'\big) \vee \\
&\vee \big(\neg x_1 \wedge x_2 \wedge \neg x_3 \wedge x_1' \wedge \neg x_2' \wedge \neg x_3'\big) \vee \\
&\vee \big(\neg x_1 \wedge x_2 \wedge x_3 \wedge x_1' \wedge \neg x_2' \wedge x_3'\big) \vee \\
&\vee \big(x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge \neg x_1' \wedge x_2' \wedge x_3'\big) \vee \\
&\vee \big(x_1 \wedge \neg x_2 \wedge x_3 \wedge \neg x_1' \wedge \neg x_2' \wedge \neg x_3'\big) \vee \\
&\vee \big(x_1 \wedge x_2 \wedge \neg x_3 \wedge \neg x_1' \wedge \neg x_2' \wedge x_3'\big) \vee \\
&\vee \big(x_1 \wedge x_2 \wedge x_3 \wedge \neg x_1' \wedge x_2' \wedge \neg x_3'\big)
\end{aligned}$$

$\square$

(b) For each of the two approaches to model checking discussed above, give an example of a formally specified property of the program that could be checked using that method and which would be less suited to analysis with the other approach. Explain your reasoning in each case.

In this part, the value of n is fixed but not necessarily equal to 2.

(i) Property that can be checked with SMC but is less suited for analysis with BMC

The property $\exists\Diamond a$ where a is the statement "$x = 1$" can be easily checked (or disproved) by symbolic model checking but is less suited for analysis with bounded model checking.

Checking the property with SMC can be done efficiently by doing a symbolic fixed point computation of $f_{\mathrm{Sat}(\exists(\mathit{true}\,\mathtt{U}\,a))}$.

However, for BMC, we would need to construct the logical proposition for $f_{\rightarrow}$ and apply it up to the completeness threshold for all possible initial states. This takes exponential time, since we are not giving the algorithm one single initial state, but are instead forced to check all of the possibilities.

(ii) Property that can be checked with BMC but is less suited for analysis with SMC

The property $b \wedge \forall\Box a$ where a is the statement "$x < 2n - 1$" and b is the statement $x = 0$ can be easily checked (or disproved) by bounded model checking but is less suited for analysis with symbolic model checking.

To check the property, we need to construct the logical proposition for $f_{\rightarrow}$, which contains one clause for each transition in the LTS of the program. With a fixed n, there are $2n$ states and thus $2n$ transitions, which is easy to construct. To perform the BMC algorithm we start with the initial state $\mathit{init} = \neg x_0 \wedge ... \neg x_k$ where k is the number of bits needed to encode the states and apply $f_{\rightarrow}$ up to the completeness threshold.

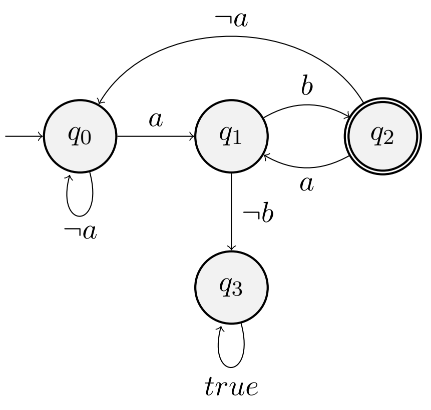
However, to perform the SMC algorithm we would need to construct a BDD for the transition relation and for a large expression, which suffers from state space explosion with large n.

Question 4

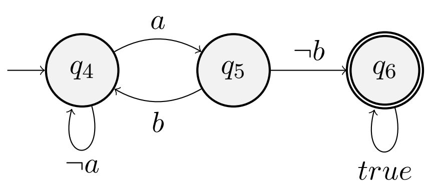
Runtime verification is used to check the correctness of individual system executions and to detect whether the violation or satisfaction of any LTL formula has happened as early as possible. This method of verification considers only finite traces, as opposed to standard LTL where traces satisfying a formula are always infinite.
For any LTL formula ϕ, the paper constructs a monitor FSM $\mathcal{M}^{\phi}$ which reads finite traces obtained from the execution of a program and outputs one of the following symbols:

- $\top$, which indicates that every continuation of the prefix satisfies ϕ
- $\bot$, which indicates that no continuation of the prefix satisfies ϕ
- ?, which indicates that some continuations satisfy ϕ and some do not.

Let us use the method described in the paper to construct such an FSM for the LTL formula $\phi = \Box(a \to \bigcirc b)$. First, we construct $\mathcal{A}^{\phi}$ and $\mathcal{A}^{\neg\phi}$, which are the NBAs accepting all words satisfying ϕ and all words falsifying ϕ, respectively:

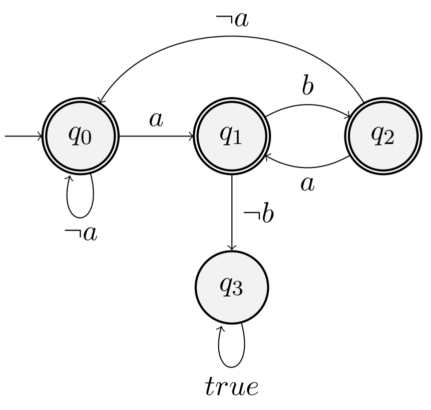


$\mathcal{A}^{\phi}$

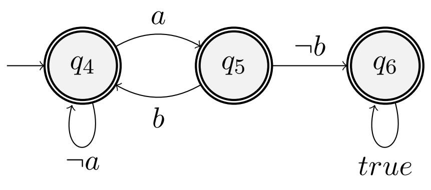


$\mathcal{A}^{\neg\phi}$

We then define $\mathcal{F}^{\phi}(q) = \top$ iff $\mathcal{L}(A^{\phi}(q)) \neq \varnothing$, and construct an NFA $\hat{\mathcal{A}}^{\phi}$ where a state q is accepting iff $\mathcal{F}^{\phi}(q) = \top$. We also do identically for $\neg\phi$ to arrive at the following NFAs:

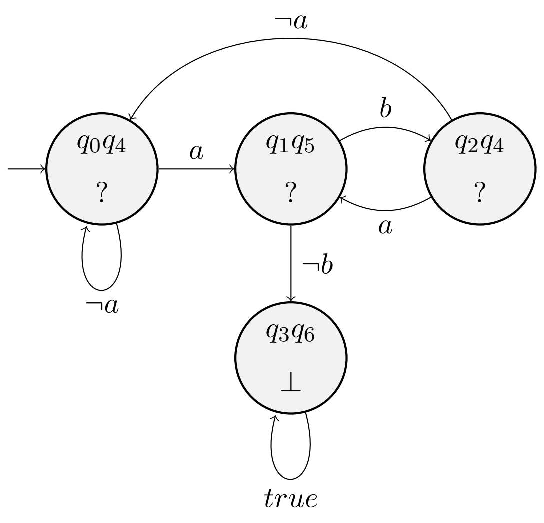


$\hat{\mathcal{A}}^{\phi}$

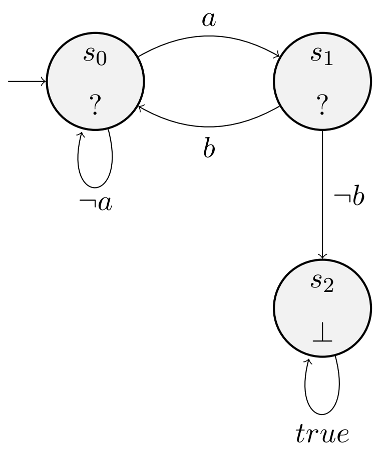


$\hat{\mathcal{A}}^{\neg\phi}$

In general, we would now have to convert these NFAs $\hat{\mathcal{A}}^{\phi}, \hat{\mathcal{A}}^{\neg\phi}$ into DFAs $\tilde{\mathcal{A}}^{\phi}, \tilde{\mathcal{A}}^{\neg\phi}$ with a worst-case exponential blow-up in states. However, in this example, we have already constructed our NBAs such that the resulting NFAs have deterministic transition rules, and hence no conversion is necessary. Thus $\tilde{\mathcal{A}}^{\phi} = \hat{\mathcal{A}}^{\phi}, \tilde{\mathcal{A}}^{\neg\phi} = \hat{\mathcal{A}}^{\neg\phi}$. Then, the minimised product of these two automata is our final monitor automaton $\mathcal{M}^{\phi}$:



Product automaton $\bar{A}^{\phi} = \tilde{\mathcal{A}}^{\phi} \times \tilde{\mathcal{A}}^{\neg\phi}$



Unique minimised product automaton $\mathcal{M}^{\phi}$

Using this automaton, we can input any finite-length trace u of a program execution and read the value of $[u \vDash \phi]$ from the resulting state, which is equal to one of $\{\top, \perp, ?\}$. Note that this gives us an efficient way of testing whether a prefix is good, bad, or neither, since according to Remark 3.2 from the paper a prefix is good iff $[u \vDash \phi] = \top$, bad iff $[u \vDash \phi] = \perp$, and neither otherwise. In particular, we can observe that for this LTL formula $\phi = \Box(a \to \bigcirc b)$ there do not exist any good prefixes, since none of the states output a $\top$ symbol. This aligns with our expectations, since any prefix can be extended to a word that does not satisfy ϕ by adding an a followed by a $\neg b$. Similarly, we are able to observe that a minimal bad prefix exists and is equal to $\{a\}\,\{\neg b\}$.

In Definition 3.4, the paper defines that a property is monitorable iff it has no ugly prefix; that is, a prefix that has no finite continuation that is either bad or good. It then shows that all safety and co-safety properties are monitorable. Let us discuss the other two classes we have encountered during the course: invariants and liveness properties. Since all invariants are safety properties, we conclude that invariants are also monitorable. On the other hand, liveness properties may or may not be monitorable. For example, the liveness property $\psi_1 = \Diamond a$ is monitorable, as any prefix can be extended to a good prefix by adding an a to the end, which implies that there are no ugly prefixes. However, the liveness property $\psi_2 = \Box\Diamond a$ is not monitorable, as there do not exist any good or bad prefixes for the language defined by ψ_2. Proof: there are no bad prefixes by definition of a liveness property, and there are no good prefixes since every prefix can be extended to an infinite word that does not satisfy ψ_2 by appending infinite $\varnothing$ to the end. Thus we can conclude that the empty prefix is an ugly prefix for ψ_2

and thus the property is not monitorable.

On a theoretical level, this method of model checking may seem inefficient due to its worst-case time complexity of $O(2^{2^{|\phi|}})$. Compare this to the other model checking algorithms we have seen during the course: CTL model checking has a complexity of $O(|M|\cdot|\phi|)$, which is linear in both LTS and formula size, and LTL model checking has a complexity of $O(|M|\cdot 2^{|\phi|})$. However, the paper shows that in practice, runtime verification has a smaller average complexity as the resultant monitor automaton can be efficiently reduced. In fact, they observe that constructing the monitor automaton is generally more efficient than trying to synchronise the two NFAs on-the-fly. In addition, doing runtime verification is more efficient as it does not require one to model the whole system as an LTS beforehand. The method only uses the LTL formula to construct its automaton and therefore can analyse programs of arbitrarily large size just as fast. In general, useful LTL formulae are relatively small, and an exponential or even double-exponential blow-up is acceptable. By contrast, traditional CTL or LTL model checking has a complexity that is linear in the model size, which may become too large to handle if the program suffers from state-space explosion, which is often the case with more complex systems.

However, the efficiency of runtime verification comes with the downside that it is not designed to guarantee the correctness of a program. Other methods that consider finite state spaces of a program, such as bounded model checking, guarantee that the execution of a program will be correct up to some depth, and with a suitably large depth defined by the completeness threshold, it is able to guarantee correctness of a whole program. However, runtime verification can only consider individual finite traces of a program and cannot therefore be complete.

Due to its live monitoring and relative simplicity, runtime verification is often useful in live systems that must be protected against undesirable conditions where model checking and the other methods studied on the course are not useful. For example, it is used by the ContractLarva tool to monitor whether a smart contract has been violated by a party and to trigger remedial behaviour, such as issuing fines, as a result [1]. The violation of a contract is considered to be an *accepted risk*, and therefore a model checker aiming to test the whole state space would always be able to find a violation of a desired condition in the system, which is not useful information. However, with runtime verification, we get insight on whether a violation *actually* happens during execution, and with the techniques outlined in the present paper, we can guarantee that the violation is detected as early as possible.

References

[1] Joshua Ellul and Gordon J. Pace. "Runtime Verification of Ethereum Smart Contracts". In: *2018 14th European Dependable Computing Conference (EDCC)*. 2018 14th European Dependable Computing Conference (EDCC). Sept. 2018, pp. 158–163. DOI: 10.1109/EDCC.2018.00036. URL: https://ieeexplore.ieee.org/document/8530777 (visited on 07/01/2025).