Theoretical and practical model checking of automation systems

Igor Buzhinsky

igor.buzhinskii@aalto.fi

Summer school in Luleå, Sweden

June 18–19, 2018
Introduction

- Formal definition of LTL, CTL, and the model checking problem
- Model checking tools
- Modeling a PLC-like closed-loop system

- These slides are based on the contents of the course “Automation software synthesis and analysis” in Aalto University and my practical experience
Several tutorials based on these slides can be completed independently
I can help with doing them (after going through these slides)

- Temporal logics
- UPPAAL
- NuSMV
- LTL synthesis
- SPIN
- **System**: interface, internal structure, state, dynamics
- A formal model of the system can be described using the same elements
- We will only consider **discrete-state** models
- The simplest way to model a continuous-state system as a discrete-state model is to discretize time and continuous values
- Models of automation systems can be **open-loop** or **closed-loop**, depending on whether only the controller or both the controller and the plant are modeled
Closed-loop model: the most basic view

- **The controller** is the software or hardware whose correctness we wish to check.
- **The plant** comprises processes and devices with which the controller works.
- The controller is often deterministic but the plant is not.
- Human behavior may be modeled either separately or inside the plant model.
- If needed, the diagram above can be extended with “free” inputs, e.g. buttons may be treated as additional inputs to the controller or the plant (or pressing them can be modeled inside the plant model).
Model checking: introduction
State (reachability) graph of a system

- **Nodes**: all reachable states of the system
- If the system is modular, then the state of the system consists of the state of all its modules
- **Directed edges**: one-step evolutions of the state
- Multiple outgoing edges are possible from each state, i.e. **nondeterminism** is common
State graph: example

NCES module (actually, a Petri net)

State graph, state = $p_1p_2p_3$
Kripke structures

- Formalization of a state graph

- Let $\text{AP}$ be the a finite set of so-called atomic propositions

- Then $M = (S, I, T, L)$ is a Kripke structure, where:
  - $S$ is a finite set of states
  - $I \subset S$ is a set of initial states
  - $T \subset S \times S$ is a transition relation
  - $L : S \rightarrow 2^{\text{AP}}$ is a labeling function

- No deadlock assumption (for simplicity of definitions):
  $\forall s \in S \exists s' \in S : (s, s') \in T$
State graph interpreted as a Kripke structure

$\text{AP} = \{ "p_i = j" | i = 1..3, j = 0..2 \}$

$S$: nodes of this graph

$I \subset S = \{101\}$

Note: in UPPAAL and NCES models there is always one initial state!

$T \subset S \times S$: edges of the graph, e.g. $(002, 011), (011, 002), ...$

$L : S \rightarrow 2^{\text{AP}}$: token assignments (markings) in each state, e.g.

$L(020) = \{ "p_0 = 0", "p_1 = 2", "p_3 = 0" \}$

Specifications can be interpreted as predicates over Kripke structures
System behaviors are paths in Kripke structures

Infinite paths are common in formal verification. This is the reason why deadlocks are undesirable.

What happens in terms of the original system?
Model checking: LTL
Assume that now we have only two atomic propositions: $p$ and $q$. All possible behaviors are infinite sequences over $2\{p,q\}$. Example: $\{p, q\}, \{p\}, \{}, \text{cycle}(\{q\}, \{p, q\})$. Boolean logic is able to characterize single elements of such sequences. Can we somehow introduce predicates over infinite sequences of atomic propositions? For example, to formulate a specification: each $p$ is followed by $\neg p$ on the next step (which is false for the example).
Linear temporal logic (LTL)

- Formal language which extends the usual propositional Boolean logic
- Variables: atomic propositions, e.g. $p$ and $q$
- Usual Boolean operators are allowed, e.g. $p \rightarrow q$ (i.e. $\neg p \lor q$) is an LTL formula, but it refers to the first element of an infinite sequence

Temporal operators

- $G$: globally (always), e.g. $G(p \rightarrow q)$ means “in each element of the sequence, $p \rightarrow q$ holds”
- $F$: in the future, e.g. $F(p \rightarrow q)$ means “for some element of the sequence, $p \rightarrow q$ holds”
- $X$: on the next step, e.g. $X(p \rightarrow q)$ means “$p \rightarrow q$ holds for the second element of the sequence”
- $U$: until (binary operator), e.g. $p U q$ means “$q$ must happen at some step, and the sequence must satisfy $p$ until (non-inclusive) $q$ happens”
Examples of LTL formulas

- Path 1: \( \{p, q\}, \{p\}, \emptyset, \text{cycle}(\{q\}, \{p, q\}) \)
- Path 2: \( \text{cycle}(\{p, q\}) \)
- Path 3: \( \emptyset, \text{cycle}(\{p\}, \{p, q\}, \{q\}) \)

- \( f_1 = G p \) – path 2
- \( f_2 = F(\neg p \land \neg q) \) – paths 1, 3
- \( f_3 = p U(\neg p \land \neg q) \) – paths 1, 3

Temporal operators can be applied to arbitrary LTL formulas!

- \( f_4 = X X X X p \) (“on the fifth step”) – paths 1, 2, 3
- \( f_5 = F G(p \land q) \) (“globally from some point”) – path 2
- \( f_6 = G F(p \land q) \) (“infinitely often”) – paths 1, 2, 3
- \( f_7 = G(p \rightarrow X q) \) (“p is always followed by q”) – paths 2, 3
LTL: simplification and equivalence rules

- $G G f = G f$
- $F F f = F f$
- $G X f = X G f$
- $F X f = X F f$
- $\neg G(f) = F(\neg f)$
- $\neg F(f) = G(\neg f)$
LTL verification: definition

- Kripke structure $M$ satisfies LTL formula $f$ (written: $M \models f$), if all paths in $M$ which start in $M$'s initial states satisfy $f$

Quiz: which of these LTL formulas are satisfied by the KS on the right? Why?

- $f_1 = G\ p$
- $f_2 = F(\neg p \land \neg q)$
- $f_3 = p\ U(\neg p \land \neg q)$
- $f_4 = X\ X\ X\ X\ p$
- $f_5 = F\ G(p \land q)$
- $f_6 = G\ F(p \land q)$
- $f_7 = G(p \rightarrow X\ q)$
LTL verification: definition

- Kripke structure $M$ satisfies LTL formula $f$ (written: $M \models f$), if all paths in $M$ which start in $M$’s initial states satisfy $f$

Quiz: which of these LTL formulas are satisfied by the KS on the right? Why?

- $f_1 = \mathbf{G} p$
- $f_2 = \mathbf{F}(\neg p \land \neg q)$
- $f_3 = p \mathbf{U}(\neg p \land \neg q)$
- $f_4 = \mathbf{X} \mathbf{X} \mathbf{X} \mathbf{X} p$
- $f_5 = \mathbf{F} \mathbf{G}(p \land q)$
- $f_6 = \mathbf{G} \mathbf{F}(p \land q)$
- $f_7 = \mathbf{G}(p \rightarrow \mathbf{X} q)$

Answer: only $f_6$
We wish to check whether \( f \) holds for Kripke structure \( M \).

\( \neg f \) is converted to a so-called Büchi automaton, which is an acceptor over infinite words which satisfy \( \neg f \).

\( M \) is composed with this automaton.

If the composition accepts at least one infinite word, then this word satisfies \( \neg f \) and belongs to \( M \), so \( f \) is false, and the obtained word is a counterexample.

Otherwise, \( f \) is true.

We won’t go into details.
Two cylinders system

- The extension of each cylinder is discretized into four intervals.
- When both cylinders share interval 4, they collide.
- A workpiece can be placed into the shared interval.
- If a cylinder reaches interval 4 and there is a workpiece, it is pushed.
LTL specification for the two cylinders system: plant model

- Atomic propositions: $h_1, h_2, h_3, h_4$ (displacements of the horizontal cylinder), $v_1, v_2, v_3, v_4$ (displacements of the vertical cylinder), $w$ (workpiece is present)

- Cylinder has a position: $G(h_1 \lor h_2 \lor h_3 \lor h_4)$, $G(v_1 \lor v_2 \lor v_3 \lor v_4)$

- Cylinder can’t have more than one position:  
  $G(\neg(h_1 \land h_2) \land \neg(h_1 \land h_3) \land \neg(h_1 \land h_4) \land \neg(h_2 \land h_3) \land \neg(h_2 \land h_4) \land \neg(h_3 \land h_4))$,
  $G(\neg(v_1 \land v_2) \land \neg(v_1 \land v_3) \land \neg(v_1 \land v_4) \land \neg(v_2 \land v_3) \land \neg(v_2 \land v_4) \land \neg(v_3 \land v_4))$

- If a cylinder is fully extended, then there is no workpiece:  
  $G(h_4 \lor v_4 \rightarrow \neg w)$

- ...

- Such specifications can help “debug” the plant model
LTL specification for the two cylinders system: requirements for the controller

- We won’t use any additional atomic propositions: all we need can be specified in terms of the plant!

- Cylinders do not collide: \( G \neg (h_4 \land v_4) \)

- When a workpiece appears, it must be eventually pushed away: 
  \( G(w \rightarrow F \neg w) \)

- Cylinders iterate (each new workpiece is pushed by a different cylinder):
  \[
  G((h_4 \land (X \neg h_4) \land F w) \rightarrow X((\neg w \land \neg v_4 \land \neg h_4) U(w \land (w U(v_4 \land \neg h_4)))))
  \]
  and the same for the other cylinder
Model checking: CTL
In LTL, there is always an implicit quantification over all paths starting in initial states.

In **CTL**, all temporal operators are annotated with quantifiers.

CTL formulas characterize not infinite sequences, but rather states of the Kripke structure.

A Kripke structure satisfies a CTL formula, if **all its initial states** satisfy this formula.

Let $s$ be a state of the KS, then $s \models f$ means $s$ satisfies $f$. 
CTL: temporal operator $\text{EX}$

$s \models \textbf{EX}(f)$ ("exists next", not supported by UPPAAL): there is a successor of $s$ where $f$ holds
$s \models \mathbf{AX}(f)$ ("for all next", not supported by UPPAAL): in all successors of $s$, $f$ holds
$s \models EF(f)$ ("exists in the future", $E<>$ in UPPAAL): there exists a path starting in $s$ such that $f$ becomes valid at some point of this path
CTL: temporal operator $\text{AF}$

- $s \models \text{AF}(f)$ ("for all in the future", $\text{A}\langle\rangle$ in UPPAAL): for all possible paths starting in $s$, $f$ becomes true at some point
CTL: temporal operator $\text{EG}$

$s \models \text{EG}(f)$ ("exists globally", $E[]$ in UPPAAL): there exists a path starting in $s$ such that $f$ holds at every state along this path
CTL: temporal operator $\textbf{AG}$

- $s \models \textbf{AG}(f)$ ("for all globally", $\textbf{A}[]$ in UPPAAL): for all possible paths starting in $s$, $f$ is always true
CTL: temporal operators \textbf{EU} and \textbf{AU}

- $s \models f \textbf{EU} g$ ("exists until", not supported by UPPAAL): there exists a path starting in $s$ such that $f$ holds until (non-inclusive) $g$, and $g$ eventually happens
- $s \models f \textbf{AU} g$ ("for all until", not supported by UPPAAL): for all possible paths starting in $s$, $f$ holds until (non-inclusive) $g$, and $g$ eventually happens
CTL verification: example

- KS satisfies the CTL formula iff all its initial states satisfy it
- Which of these CTL formulas are satisfied by the KS on the right? Why?
  - $f_1 = \text{AG} p$
  - $f_2 = \text{AG}(p \lor q)$
  - $f_3 = \text{AF}(p \land q)$
  - $f_4 = \text{EF}(\neg p \land \neg q)$
  - $f_5 = \text{AX AX AX AX AX} p$
  - $f_6 = \text{EF EG}(p \land q)$
  - $f_7 = \text{EG EF}(p \land q)$
  - $f_8 = \text{AG}(p \rightarrow \text{AX} q)$

Answer:

- $f_2$, $f_3$, $f_6$, $f_7$
CTL verification: example

- KS satisfies the CTL formula iff all its initial states satisfy it
- Which of these CTL formulas are satisfied by the KS on the right? Why?
  - $f_1 = \mathbf{AG} p$
  - $f_2 = \mathbf{AG}(p \lor q)$
  - $f_3 = \mathbf{AF}(p \land q)$
  - $f_4 = \mathbf{EF}(\neg p \land \neg q)$
  - $f_5 = \mathbf{AX} \mathbf{AX} \mathbf{AX} \mathbf{AX} p$
  - $f_6 = \mathbf{EF} \mathbf{EG}(p \land q)$
  - $f_7 = \mathbf{EG} \mathbf{EF}(p \land q)$
  - $f_8 = \mathbf{AG}(p \rightarrow \mathbf{AX} q)$
- Answer: $f_2, f_3, f_6, f_7$
Graph theory approach

- If $f$ is just a Boolean formula, then it is trivial to check whether it holds in a desired state
- $\text{AX } f$ holds in $s$, if we previously found that $f$ holds for each in all its successors
- Similar (sometimes more complex) ideas for other operators
- We won’t go into details

Symbolic approach

- Implicit representation of states via Boolean formulas
- Mitigation of the state explosion problem
CTL verification: two cylinders

- Atomic propositions: $h_1..h_4, v_1..v_4, w$

**Quiz:** specify the following properties in CTL:

- If a cylinder is fully extended, then there is no workpiece
- Cylinders do not collide
- When a workpiece appears, it must be eventually pushed away
- Cylinders iterate
Quiz answers

- If a cylinder is fully extended, then there is no workpiece: \( \text{AG}(h_4 \lor v_4 \rightarrow \neg w) \)
- Cylinders do not collide: \( \text{AG} \neg (h_4 \land v_4) \)
- When a workpiece appears, it must be eventually pushed away: \( \text{AG}(w \rightarrow \text{AF} \neg w) \)
- Cylinders iterate: cannot be specified
### Common specification types (“patterns”)

<table>
<thead>
<tr>
<th>Name</th>
<th>LTL</th>
<th>CTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generality / Invariance</td>
<td>$G f$</td>
<td>$AG f$</td>
</tr>
<tr>
<td>Bounded response</td>
<td>$G(p \rightarrow X^n q)$</td>
<td>$AG(p \rightarrow (AX)^n q)$</td>
</tr>
<tr>
<td>Unbounded response</td>
<td>$G(p \rightarrow F q)$</td>
<td>$AG(p \rightarrow AF q)$</td>
</tr>
<tr>
<td>Infinitely often</td>
<td>$G F p$</td>
<td>$AG AF p$</td>
</tr>
</tbody>
</table>
There are properties which cannot be expressed in both LTL and CTL

- $G p / \ AG p$, $F p / \ AF p$,
  $G F p / \ AG AF p$ – both LTL and CTL
- $EF p$ – only CTL, but there is a workaround to check it in LTL!
- $F G p$ – only LTL
- $AG EF p$ – only CTL
- $CTL^*$ is a larger logic which allows combining quantified and unquantified temporal operators, but it cannot be easily verified
Model checking: safety and liveness LTL properties
An LTL formula $f$ is a **safety** formula, if all possible counterexamples to $f$ have a **finite prefix** such that every its infinite continuation is a counterexample.

Informally speaking, such properties state that “something bad” does not happen.

Examples: $f$, $Xf$, $Gf$, $G(f \rightarrow Xg)$

Each safety property can be converted to a (possibly nondeterministic) safety automaton.

Safety automaton rejects an input sequence if it can visit a rejecting state while reading it.
Assume that we have a Kripke structure...

\[ f = G(\neg p) \]

State machine to check \( f \)? With guards on transitions and a rejecting state
Examples of safety automata (2)

\[ f_1 = \mathbf{G}(\neg p) \]

\[ f_2 = x \land X y \]

\[ f_3 = \mathbf{G}(x \land X y) \]
Examples of safety automata (3)

\[ f_4 = G(x \rightarrow y \land X y) \]

\[ f_5 = F y \]

Not a safety property!
Liveness and other properties

- LTL property $f$ is a liveness property iff each possible finite trace of the model can be extended to a one which satisfies $f$
- Informally: something “good” must happen
- Examples: $F f$, $F G f$, $G F f$, $G (f \rightarrow F g)$

- Each other LTL property can be represented as a conjunction of a safety property and a liveness property (Alpern & Schneider, 1985)
- Example: $f U g$

- Safety property are often easier to model-check (and also to apply formal synthesis), use them where possible!
Model checking reachability
Reachability use cases

- State subset = any predicate over atomic propositions

1. Detect whether a certain dangerous state subset is reachable in the model
2. Check whether a certain state subset is reachable in order to check the validity of the model
3. Coverage test generation: force the model checker to generate counterexample which show reachability of certain state subsets, and then interpret these counterexamples as test cases
We check the reachability of Boolean formula $f$

$F f$ (LTL) – incorrect, since LTL checks for all paths!

$EF f$ (CTL) – checks that $f$ is reachable from all states (according to the definition of CTL)

$G \neg f$ (LTL), $AG \neg f$ – checks that $f$ is reachable from some state; reachability will result in a counterexample showing a path to $f$!

This is the recommended way of checking reachability
Testing by means of model checkers

- Basic idea: formulate **coverage goals** as Boolean formulas $f_1, \ldots, f_k$, run the model checker for $G \neg f_1, \ldots, G \neg f_k$, use the counterexamples as test cases
- Many approaches develop this idea
- There is a tool which implements testing by means of model checking for closed-loop systems: https://github.com/igor-buzhinsky/formal_testing_in_closed_loop (the corresponding paper is not yet published, but a draft can be sent privately)
Tools: UPPAAL model checker
About UPPAAL

- Graphical, platform-independent tool
- Models are specified visually as state machines
- A model may contain multiple state machines, which can interact using shared variables and channels
- By default, execution of different state machines is asynchronous (unless channels are used for synchronization)
- Timed automata are supported
- Partial support of CTL: can check $\text{AG} f$, $\text{AF} f$, $\text{EG} f$, $\text{EG} f$, $\text{AG}(f \rightarrow \text{F} g)$, where $f$ and $g$ are Boolean formulas
- Alternative notation for temporal operators: $\text{G} = []$, $\text{F} = <>$
- LTL safety properties can be checked by modeling them as safety automata and checking whether an “unsafe” states of these automata are reachable
UPPAAL: state machines

```
Name: OUTPUT_TRAY
Parameters: broadcast chan &exec

- ABSENT
  exec?
  pp0 = false
- TOP
  exec?
- BOTTOM
```
bool vcu = true, vcd, s0 = true, s1, s2, s3, pp0, pp1, pp2, pp3, vacuum;
bool VCGD, L1CGO, L1CGI, L2CGO, L2CGI, VENTURI;
bool carries_wp, h1retracted = true, h1extended, h2extended, h2retracted = true;
broadcast chan tray_exec, cyl_exec, controller_exec, monitor_exec;

void update_vars() {
    s0 = h1retracted && h2retracted;
    s1 = h1extended && h2retracted;
    s2 = h1retracted && h2extended;
    s3 = h1extended && h2extended;
carries_wp &= vacuum;
vacuum = VENTURI;
}
UPPAAL: simulator
UPPAAL: verifier

Overview

- `pp1 -> !pp1`
- `pp2 -> !pp2`
- `pp3 --> !pp3`

`(pp1 | pp2 | pp3) --> carries_wp`
`carries_wp --> pp0`
`(pp1 | pp2 | pp3) --> (!pp1 & !pp2 & !pp3)`
`(pp1 | pp2 | pp3) --> vacuum`

Query

- `pp3 --> !pp3`

Comment

- Status
- `pp3 --> !pp3`
- **Property is not satisfied.**
Tools: NuSMV and symbolic model checking
State spaces can be very large

If $n$ bits are used to store all the state variables of the model, the state space can have up to $2^n$ elements in the worst case

Idea of symbolic model checking: avoid explicit construction of the state graph
State subsets as Boolean constraints

- The set of reachable states as a Boolean formula: $p \lor q$

- The set of initial states:
  $p \oplus q = p \land \neg q \lor \neg p \land q$
Transition relation as Boolean constraints

- $p, q$: values on this step
- $p', q'$: values on the next step

Transition relation for the Kripke structure on the left as a Boolean formula:

- $(p \land \neg q \rightarrow q' \land \neg p') \land (q \land \neg p \rightarrow p' \land q') \land (p \land q \rightarrow p')$

Alternative way:

- $(p \land \neg q \land q' \land \neg p') \lor (q \land \neg p \land p' \land q') \lor (p \land q \land p')$
Assume that our Kripke structure has atomic propositions \( p_1, \ldots, p_n \)
Boolean constraints \( f_{\text{init}}[p_1, \ldots, p_n] \) and \( f_{\text{trans}}[p_1, \ldots, p_n, p'_1, \ldots, p'_n] \)
How to model-check \( g = \mathbf{AG} h \), where \( h \) is a Boolean formula?
Compute a sequence of formulas \( f_i \): the set of states reachable in \( i \) steps
\[
f_0 := f_{\text{init}}; \quad f_i := f_{i-1} \lor \text{remove_primes}(\exists p_1, \ldots, p_n : f_{i-1} \land f_{\text{trans}})
\]
If \( f_i \land \neg h \) is satisfiable, then \( g \) is false
If at some point \( f_i \) and \( f_{i-1} \) become equivalent, we can stop the procedure and conclude that \( g \) is true
How to perform all these symbolic operations efficiently? There are binary decision diagrams (BDDs), a reduced form of decision trees
Example of a BDD

- Solid arrows: variable is true
- Dashed arrows: variable is false
- If in the end we come to 1, then the formula is true for our assignment
- If we come to 0, it is false
- On this BDD: 
  \[(x_1 = y_1) \land (x_2 = y_2)\]
NuSMV

- Open-source symbolic model checker
- Supports LTL and CTL BDD-based model checking, bounded (SAT-based) LTL model checking
- Can be downloaded here: http://nusmv.fbk.eu/
- Command-line tool, models are specified in text files

nuXmv (https://nuxmv.fbk.eu/) is a derivative of NuSMV which offers better SAT-based model checking and can verify infinite-state systems

- Not covered in this slide set
MODULE main()
VAR
  p: boolean;
  q: boolean;
  c: 0..10;
INIT
  (c = 0)
  & (p)
TRANS
  (next(c) = (c + 1) mod 10)
  & (next(p) = !p)
CTLSPEC AG(c != 10)
LTLSPEC G(p -> X(!p))

• Integers are supported
• Are the specifications in the end satisfied? – Yes
• What about q? – It can be anything on any turn
MODULE main()
VAR
    p: boolean;
    q: boolean;
    c: 0..10;
ASSIGN
    init(c) := 0;
    init(p) := TRUE;
    next(c) := c + 1 mod 10;
    next(p) := !p;
DEFINE
    c_plus_1 := c + 1;
CTLSPEC AG(c != 10)
LTLSPEC G(p → X(!p))
MODULE CYLINDER(fwd, back)

VAR
    pos: 0..5;

ASSIGN
    init(pos) := 0;
    next(pos) := fwd ? next_pos : back ? prev_pos : pos;

DEFINE
    next_pos := pos < 5 ? (pos + 1) : pos;
    prev_pos := pos > 0 ? (pos - 1) : pos;
    home := pos = 0;
    end := pos = 5;

Modules can have inputs (in the declaration), and their variables and definitions can be interpreted as outputs

C-style choice operator ?:
NuSMV: controller

MODULE CONTROLLER(home, end)
VAR
    state: {moving_fwd, moving_back};
ASSIGN
    init(state) := moving_fwd;
    next(state) := case
        home: moving_fwd;
        end: moving_back;
    TRUE: state;
esac;
DEFINE
    fwd := state = moving_fwd;
    back := state = moving_back;

- Example of explicit state machine modeling
NuSMV: closed-loop composition

MODULE main()
VAR
   -- this is the way to write comments, by the way
   cyl: CYLINDER(ctr.fwd, ctr.back);
   ctr: CONTROLLER(cyl.home, cyl.end);

LTLSPEC G F cyl.end -- TRUE
LTLSPEC G F cyl.home -- TRUE

- **Synchronous**: all the modules make a step together!
- How to model asynchronous interaction? By specifying execution permissions: no permission = state is not changed
Often, if there is a deadlock in the model, NuSMV complains about the set of fair states being empty. This means that there is a deadlock in the model.

Unfortunately, there are more dangerous situations where there is no deadlock, but some behaviors of the model are disabled due to modeling errors.

Safety measures:
- Use ASSIGN instead of TRANS and INIT.
- Specify reachability requirements (e.g. $\text{G} \neg p$ to check that $p$ can be true) to ensure that important state space regions are reachable.

There is a model “nusmv-deadlocks” in the materials.
Checking equivalent LTL/CTL specifications in NuSMV

- If there is an LTL property $f$ and CTL property $g$ and they are equivalent (i.e. result in the same value for each model), then usually CTL model checking is faster (Clarke et al., 1994)

- Examples of equivalent properties: $\mathbf{G} p$ and $\mathbf{AG} p$, $\mathbf{GF} p$ and $\mathbf{AG AF} p$, $\mathbf{G}(p \rightarrow \mathbf{F} g)$ and $\mathbf{AG}(p \rightarrow \mathbf{AF} g)$

- For arbitrary properties, equivalence may be hard to prove (see the tutorial on temporal logics)

- Also remember about bounded model checking (BMC), a reduced form of model checking which is often faster

- If you need to check an invariant (the pattern $\mathbf{G} p / \mathbf{AG} p$), NuSMV has specific (and potentially faster) algorithms to check them; see INVARSPEC in the NuSMV manual
Path visualization tool

- Download from https://github.com/igor-buzhinsky/nusmv_counterexample_visualizer
- The tool supports interpreting counterexamples produced by NuSMV (will be covered later in the course), but can also be used to structurally explain LTL formula values on user-specified paths
- **Important atomic propositions** are highlighted

- Variable view: suppose that the state of the formal model is composed on a number of variables, either Boolean or integer
- Boolean variables can be interpreted as atomic propositions right away
- Statements over integer variables (e.g. comparisons like $x > 5$) can also be interpreted as atomic propositions
Tools: SPIN model checker
SPIN

- Open-source **explicit-state** model checker
- Supports LTL
- Can be downloaded here: http://spinroot.com/
- Can be run as a command-line tool, but also has GUI (iSpin)
int pos = 0;
bool home = true, end, fwd, back;

// to be executed in a loop:
#define next_pos (pos < 5 -> (pos + 1) : pos)
#define prev_pos (pos > 0 -> (pos - 1) : pos)
pos = (fwd -> next_pos : (back -> prev_pos : pos));
home = pos == 0;
end = pos == 5;

- C-like syntax, but the choice operator has a different syntax
- C macros and other preprocessor directives are supported
- Conditional and loop statements (not shown) are very different, see online manuals if interested
bool home = true, end, fwd, back;
mtype = { moving_fwd, moving_back };  
mtype state;

// to be executed in a loop:
state = (home -> moving_fwd :
    (end -> moving_back : state));
fwd = state == moving_fwd;
back = state == moving_back;

- mtype can be used for enumerations
int pos = 0;
bool home = true, end, fwd, back;
mtype = { moving_fwd, moving_back };
mtype state;

init { do :: atomic { // a loop of atomic steps
    // <plant loop code>
    // <controller loop code>
} od }

ltl visiting_end { [] <> end }; // G F end, true
ltl visiting_home { [] <> home }; // G F home, true

- Using this pattern, PLC-like applications can be modeled
Like UPPAAL, SPIN can verify asynchronous applications

- Multiple processes are supported
- `init` is executed in the beginning
- Other process types can be declared with the keyword `proctype`
- Their instances can be spawned with the keyword `run`
- Processes can execute asynchronously, unless explicitly constrained (e.g. by channels)
- **Partial order reduction** is used to reduce the state space in case of asynchrony
Explicit-state vs. symbolic model checking
Symbolic vs. explicit-state model checking

- Explicit-state model checking: the state space of the system is stored and analyzed explicitly, as a graph
- Time and memory complexity of explicit-state model checking is linear with respect to the size of the state space

- Symbolic model checking: operations with the state space are performed explicitly since its subsets can be represented as Boolean formulas
- Binary decision diagrams (BDDs) allow efficient operations with them
- The complexity of symbolic model checking does not explicitly depend on the number of states
Example: elevator model parameterized by the number of states $n$

- The state space grows exponentially with $n$
- The model was model-checked with different algorithms, execution times in seconds are given below ($\text{TL} = 12$ hours)

<table>
<thead>
<tr>
<th>$n$</th>
<th>BDD-based CTL</th>
<th>BDD-based LTL</th>
<th>SAT-based BMC</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>10</td>
<td>80</td>
<td>45</td>
</tr>
<tr>
<td>7</td>
<td>39</td>
<td>485</td>
<td>95</td>
</tr>
<tr>
<td>8</td>
<td>26</td>
<td>197</td>
<td>191</td>
</tr>
<tr>
<td>9</td>
<td>24</td>
<td>166</td>
<td>339</td>
</tr>
<tr>
<td>10</td>
<td>17509</td>
<td>3525</td>
<td>604</td>
</tr>
<tr>
<td>11</td>
<td>731</td>
<td>TL</td>
<td>1035</td>
</tr>
<tr>
<td>12</td>
<td>71</td>
<td>3512</td>
<td>2176</td>
</tr>
<tr>
<td>13</td>
<td>1355</td>
<td>19897</td>
<td>3022</td>
</tr>
<tr>
<td>14</td>
<td>8130</td>
<td>TL</td>
<td>3784</td>
</tr>
</tbody>
</table>
When explicit-state model checking is better?

- Suppose that we have a closed-loop system where the plant is modeled as a state machine with a reasonably small number of states (e.g. \( \leq 5000 \))
- Such plant models can be constructed automatically from traces (Buzhinsky et al., 2017)
- According to my practical experience, NuSMV processes large state machines poorly
- And the benefits of the small state space cannot be exploited

- In contrast, explicit-state model checking is fast when the state space of the plant is small (compared to the open-loop case)
Some practical results on model checking nuclear I&C systems (the NPP model was provided by Fortum)

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>S5</th>
<th>S6</th>
<th>S7</th>
<th>S8</th>
</tr>
</thead>
<tbody>
<tr>
<td># temporal specs</td>
<td>9</td>
<td>24</td>
<td>26</td>
<td>15</td>
<td>10</td>
<td>18</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>Open-loop</td>
<td>SPIN</td>
<td>54</td>
<td>TL</td>
<td>TL</td>
<td>TL</td>
<td>TL</td>
<td>TL</td>
<td>3</td>
</tr>
<tr>
<td>MC time (s)</td>
<td>NuSMV</td>
<td>5</td>
<td>1</td>
<td>11</td>
<td>11</td>
<td>1</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>Closed-loop</td>
<td>SPIN</td>
<td>3</td>
<td>44</td>
<td>277</td>
<td>98</td>
<td>256</td>
<td>148</td>
<td>3</td>
</tr>
<tr>
<td>MC time (s)</td>
<td>NuSMV</td>
<td>2611</td>
<td>137</td>
<td>769</td>
<td>TL</td>
<td>718</td>
<td>1104</td>
<td>268</td>
</tr>
</tbody>
</table>

- Time limit (TL) = 10 minutes × the number of temporal specifications
- Open-loop model checking is faster in NuSMV
- Closed-loop model checking is faster in SPIN
If the model is too complex, NuSMV will be trying to verify the first temporal property for too long.

This can be so even with bounded model checking.

In contrast, in SPIN the maximum search depth can be limited, leading to a possibility of performing a reduced, less reliable model checking.

Another technique with a similar effect is **bitstate hashing**, where the memory occupied by a single state is reduced.
LTL synthesis and related tools
In supervisory control synthesis, there was a plant and requirements to be satisfied under the control of a blocking supervisor.

In LTL synthesis, there is no explicit plant, but only an LTL formula to be satisfied.

A controller interacts with an environment.

- $x_1, \ldots, x_k$: input Boolean variables; the environment can assign any values to inputs on each step.
- $y_1, \ldots, y_m$: output Boolean variables; the controller can assign any values to outputs on each step.
- $f[x_1, \ldots, x_k, y_1, \ldots, y_m]$: LTL formula.

On each step, first the environment chooses the inputs, and then the controller chooses the outputs.

LTL synthesis problem: synthesize a controller such that for all possible behaviors of the environment $f$ is satisfied.
Is the LTL synthesis problem solvable for the following formulas? If yes, how does the controller behave? If no, how should the environment behave to defeat any possible controller?

- $f = G(x \leftrightarrow y)$ – yes; $y := x$
- $f = G(x \rightarrow X y)$ – yes; $y := 1$
- $f = G((X x) \rightarrow y)$ – yes; $y := 1$
- $f = G((X x) \leftrightarrow y)$ – no; the environment can choose the next $x$ different from the previous $y$
- $f = F(x \land y)$ – no; the environment can always set $x := 0$
- $f = F x \rightarrow F(x \land y)$ – yes; $y := 1$
If the plant is finite-state, it is (usually) possible to encode it as an LTL formula

If $f_p$ describes the plant and $f_c$ are the requirements for the controller assuming that the plant submits to $f_p$, then it is sufficient to synthesize a controller for $f = f_p \rightarrow f_c$

The environment still can assign any possible values for inputs, but if it violates $f_p$, then the controller wins
G4LTL-ST

- Graphical, platform-independent tool
- Download from https://sourceforge.net/projects/g4ltl/
- Supports Boolean variables and timers
- Generates PLC code from LTL specification
BoSy

- Command-line tool, download from https://www.react.uni-saarland.de/tools/bosy/
- “BoSy is a synthesis tool based on a various bounded synthesis encodings”
EFSM-Tools

- Command-line toolset, download from https://github.com/ulyantsev/EFSM-tools/tree/devel
- Supports synthesis of controllers from LTL properties and behavior traces
- Can construct minimum finite-state machine which comply with the given specification
- Can also construct nondeterministic plant models based on the same data
Tidy modeling of PLC-like behavior in closed loop
Assumption: the software/hardware we wish to model-check is cycle-based

We will use a PLC program as an example

PLC initialization: (nothing) $\rightarrow$ initial PLC state (deterministic)

PLC loop: PLC state, plant sensor measurements $\rightarrow$ new PLC state, actuator signals (deterministic)

Plant initialization: (nothing) $\rightarrow$ initial plant state, initial sensor measurements (nondeterministic)

Plant evolution: plant state, actuator signals $\rightarrow$ new plant state, plant sensor measurements (nondeterministic)
PLC-like behavior (2)

- State of the closed-loop system: PLC state, actuator signals, plant state, plant sensor measurements
- Let’s consider a behavior trace of a discrete-state model
- What happens on its first step?
  - Plant state and sensor measurements: based on plant initialization
  - PLC state: based on PLC initialization

PLC actuator signals: ???
PLC actuator signals also have default values, but sending them to the plant may be unreasonable.
Instead, it’s better to run the PLC loop once in order for the PLC to know the sensor measurements of the plant.
State of the closed-loop system: PLC state, actuator signals, plant state, plant sensor measurements

Let’s consider a behavior trace of a discrete-state model

What happens on its first step?

- Plant state and sensor measurements: based on plant initialization
- PLC state: based on PLC initialization
- PLC actuator signals: ???

PLC actuator signals also have default values, but sending them to the plant may be unreasonable

Instead, it’s better to run the PLC loop once in order for the PLC to know the sensor measurements of the plant
PLC-like behavior (2)

- State of the closed-loop system: PLC state, actuator signals, plant state, plant sensor measurements
- Let's consider a behavior trace of a discrete-state model
- What happens on its first step?

- Plant state and sensor measurements: based on plant initialization
- PLC state: based on PLC initialization
- PLC actuator signals: ???
- PLC actuator signals also have default values, but sending them to the plant may be unreasonable
- Instead, it’s better to run the PLC loop once in order for the PLC to know the sensor measurements of the plant
Thus, on the first step of the model:

1. The plant selects its initial state and sensor measurements
2. The controller (deterministically) selects its initial state, reads the plant sensor measurements and makes its first cycle

On the second step: the plant makes an evolution based on the PLC outputs, and then the PLC makes a new cycle

Thus, on each step, first the plant acts, and then the controller does

Other ways of modeling are also possible, but I personally believe that this one is more tidy
Modeling PLC-like behavior in SPIN: pattern

<Plant state variables and sensor signals>
<Controller state variables and their default values>
<Controller actuator signals>

bool initialization = true;

init { do :: atomic {
  if
    :: initialization -> <initial plant state selection>;
  :: else -> <normal plant state update>;
  fi
  initialization = false;
  <set plant sensor signals based on the state>
  d_step { <PLC cycle> }
} od }

ltl example { X([] !bad_state) }
Similar principles can be used in UPPAAL models

init is the main (in our case, the only one) process in SPIN

do is an infinite loop

atomic makes intermediate steps invisible in model checking

d_step merges a sequence of deterministic operations into a single state change

In SPIN, the initial step of the model is unique, but we need multiple initial states → let’s wrap each LTL property into X to ignore the invalid initial state
MODULE PLANT(...inputs...)  
...  
MODULE CONTROLLER(...inputs...)  
...  
MODULE main  
VAR  
  p: PLANT(p.sensors);  
  c: CONTROLLER(c.actuators);  

LTLSPEC X(G !bad_state)  
CTLSPEC AX(AG !bad_state)

- Recall that in NuSMV all modules make their steps synchronously!  
- But it is still possible to base the action of the controller on the action of the plant! How?
• Plant: \( \text{next(state)} := f(\text{state, actuator\_signals}) \)

• Controller: \( \text{next(state)} := g(\text{state, next(sensor\_signals)}) \)

• Configure \textit{init} to produce the same unique dummy initial state as in the SPIN model

• This allows specifying PLC cycle / plant evolution only in \textit{trans}

• Thus, temporal properties are again wrapped into the \( \textbf{X} \) operator
The end


Thank you for your attention!

Questions?

Igor Buzhinsky
igor.buzhinskii@aalto.fi
http://rain.ifmo.ru/~buzhinsky/