Behavior models and verification

Lecture 7

Jan Kofroň, František Plášil
Today

- Symbolic model checking CTL using OBDD
  - and lattices
  - and fixpoints

- Partial Order Reduction
  - Optimization
Today: Symbolic CTL + POR

Model

Property specification

AG(start → AF heat)

Model checker

Property satisfied

Property violated

Error report

Jan Kofroň, František Plášil, Lecture 7
Recall: Lattice

- **Lattice** is a structure consisting of a partially ordered set $S$ of elements where every two elements have a unique supremum (least upper bound or join) and a unique infimum (greatest lower bound or meet).
- The set $P(S)$ of all subsets of $S$ forms a **complete lattice**.
- Each element $S'$ of the lattice can also be thought as a **predicate on $S$**.
- The least element is $\emptyset$ (false), the greatest element is $S$ (true).
- A function that maps $P(S)$ to $P(S)$ is called a **predicate transformer**.
Example: Subset lattice of \{1, 2, 3, 4\}
Recall: Fixpoints

- Let $\tau: P(S) \to P(S)$ be a predicate transformer.

- $\tau$ is **monotonic** provided that $Q \subseteq R$ implies $\tau(Q) \subseteq \tau(R)$.

- $Q$ is a **fixpoint** of $\tau$ iff $\tau(Q) = Q$.
function Lfp(tau : PredicateTransformer): Predicate
    Q := false;
    Q' = tau(Q);
    while (Q <> Q') do
        Q := Q';
        Q' := tau(Q);
    end while;
    return Q;
end function

function Gfp(tau : PredicateTransformer): Predicate
    Q := true;
    Q' = tau(Q);
    while (Q <> Q') do
        Q := Q';
        Q' := tau(Q);
    end while;
    return Q;
end function
Consider predicate transformer $P(S) = S \cup \{1\}$
Consider predicate transformer $P(S) = S \cup \{1\}$.
Consider predicate transformer $P(S) = S \cup \{1\}$

![Subset lattice of \{1, 2, 3, 4\}](image)
CTL operators as fixpoints

- We identify a CTL formula $f$ with the set/predicate $\{s|M, s \models f\}$ in $P(S)$
- $\text{EG}$, $\text{EU}$ may be characterized as least or greatest fixpoints of an appropriate predicate transformer:
  - $\text{EG} \ q = \nu Z (q \land EX \ Z)$
  - $\text{EU} \ [p \ U \ q] = \mu Z (q \lor (p \land EX \ Z))$

- The same holds for $\text{EF}$, $\text{AG}$, $\text{AF}$, $\text{AU}$, however those operators can be expressed using $\text{EG}$, $\text{EU}$
- Intuitively:
  - least fixpoints correspond to eventualities
  - greatest fixpoints correspond to properties that should hold forever
EG as fixpoint

Kripke structure $M$

$\tau^0(\text{true})$

$\tau^1(\text{true})$

$M, s_0 \models EG \, q$

$EG \, q = \forall Z. (q \land EX \, Z)$

$\tau(Z) = \{s: s \models q \land (\exists t: \, t \rightarrow s \land t \in Z)\}$
EU as fixpoint

Kripke structure $M$

$M, s_0 \models E[p U q]$

$E [p U q] = \mu Z. (q \lor (p \land EX Z))$

$\tau(Z) = \{s: s \models q\} \lor \{s: s \models p \land (\exists t: s \rightarrow t \land t \in Z)\}$
Symbolic model checking for CTL

- Explicit state model checking (presented earlier) is linear in size of Kripke structure and length of formula
  - State explosion problem

- Symbolic model checking algorithm operates on Kripke structures represented using OBDDs
Symbolic model checking for CTL

- Quantified Boolean formulae
  - Instead of common Boolean operators, we have (for a variable $x$ and a formula $f$)
    - $\exists x \ f$
    - $\forall x \ f$
  - The same expressive power as ordinary propositional formulae
- Using OBDDs, the quantification operators can be implemented as
  - $\exists x : f = f \mid_{x \leftarrow 0} \lor f \mid_{x \leftarrow 1}$
  - $\forall x : f = f \mid_{x \leftarrow 0} \land f \mid_{x \leftarrow 1}$
Symbolic model checking for CTL

• On top level the same approach as in explicit model checking algorithm
  ▪ Decomposing formula into sub-formulae and checking them in bottom-up manner

• Different handling of particular sub-formulae
  ▪ Based on Check*() procedures
Symbolic model checking for CTL

- **Check(CTLFormula f)**
  - f is an atomic proposition \( p \) \( \rightarrow \) return the OBDD for \( p \)
  - \( f = \neg f_1 \) or \( f = f_1 \land f_2 \) or \( f = f_1 \lor f_2 \) \( \rightarrow \) use the function Apply (*) and return the resulting OBDD

- **Formulae of the form EX f** \( \rightarrow \) return `CheckEX(Check(f))`
  - `CheckEX(OBDD o)`
  - o represents the formula f (set of states satisfying f)
  - `CheckEX(o(<v>)) = \exists<v'>[o(<v'>) \land R(<v>, <v'>)]`
  - R is the OBDD representing the transition relation

- **Formulae of the form E[f U g]** \( \rightarrow \) `CheckEU(Check(f), Check(g))`
  - Based on the fixpoint characterization of EU
    \( E \ [f \ U \ g] = \mu Z.(g \lor (f \land EX Z)) \)
  - Uses the Lfp procedure

- **Formulae of the form EG f** \( \rightarrow \) `CheckEG(Check(f))`
  - Based on the fixpoint characterization of EG
    \( EG \ f = \nu Z.(f \land EX Z) \)
  - Uses the Gfp procedure
Example of symbolic CTL model checking

\[ AF \, x = \neg EG (\neg x) \]
Example of symbolic CTL model checking

- $EG \neg x = \forall Z. (\neg x \land EX Z)$

- $\tau(Z) = \{s: s \models \neg x \land (\exists t: s \to t \land t \in Z)\}$

- We start with $Z$ as the set of all states (true):

- In each iteration, we conjunct predecessors of $Z$ with the set of states satisfying $\neg x$
Example of symbolic CTL model checking

\[ \neg x \land (\exists x0', x1': Z' \land TR) \]
Example of symbolic CTL model checking

\[ \neg x \land (\exists x_0', x_1': Z' \land TR) \]

Jan Kofroň, František Plášil, Lecture 7
Example of symbolic CTL model checking

\[ \neg x \land Z \land 0 \rightarrow 1 \]
Example of symbolic CTL model checking

Fixpoint reached $\rightarrow$ proceed upwards to:

$\neg E G (\neg x)$:

```
\begin{array}{c}
x_0 \\
0 \quad 1
\end{array} \quad \rightarrow \\
\begin{array}{c}
x_0 \\
1 \quad 0
\end{array}
```
Partial Order Reduction
Parallel composition of processes

- 3 processes
  - 1\textsuperscript{st} process – action $\alpha_1$
  - 2\textsuperscript{nd} process – action $\alpha_2$
  - 3\textsuperscript{rd} process – action $\alpha_3$

- State explosion

Jan Kofroň, František Plášil, Lecture 7
Parallel composition of processes

- Assume the actions $\alpha_1, \alpha_2, \alpha_3$ are independent
  - e.g., updates of different variables
- Assume the “intermediate states” are unimportant with respect to the property being checked
  - i.e., all the states on the picture are labeled by the same sets of atomic propositions
Partial Order Reduction (for model checking)

- Is it really necessary to search all those paths, differing only in the order of the actions \( \alpha_1, \alpha_2, \alpha_3 \)? **It is not!**
Partial Order Reduction

Markov chains
Timed automata
Labelled transition system
Kripke structure

Model

Property specification

$AG(\text{start} \rightarrow \text{AF heat})$

Model checker

Property satisfied

Error report

Property violated
Partial Order Reduction

• Idea:
  - Before the model checking is done, **reduced state graph** is constructed
    • The full state graph is never constructed
  - The method exploits the commutativity of concurrently executed transitions, which result in the same state when executed in different orders
  - Formulated by **Doron Peled** in 1993

• The name – Partial Order Reduction
  - Early versions of the algorithms were based on the partial order model of the program execution
  - Better name: **model checking using representatives**
Transitions

- It is necessary to distinguish among different transitions
  - Dependency relation among transitions
  - Invisibility of the transitions
- ➔ we modify the definition of the Kripke structure, obtaining a state transition system
State Transition System

- A Kripke structure
  - $M = (S, I, R, L)$
    - $S$ ... the set of states
    - $I \subseteq S$ ... the set of initial states
    - $R \subseteq S \times S$ ... the transition relation
    - $L: S \rightarrow 2^{AP}$ ... the labeling function

- A state transition system
  - $N = (S, T, S_0, L)$
    - $S, S_0 = I, L$ ... the same as for Kripke structures
    - $T$ ... the set of transitions
      - for each $\alpha \in T$, it is $\alpha \subseteq S \times S$

- A Kripke structure can be obtained from a state transition system
  - $R(s, s') \leftrightarrow \exists \alpha \in T: \alpha(s, s')$
State Transition System

- A transition $\alpha \in T$ is **enabled** in a state $s$
  - If there exists a state $s'$ s.t. $\alpha(s, s')$
  - Otherwise, $\alpha$ is **disabled**

- enabled($s$)
  - The set of all the transitions enabled in the state $s$

- A transition $\alpha \in T$ is **deterministic**
  - If for every state $s$, there is at most one state $s'$ s.t. $\alpha(s, s')$
  - In this case, we write $s' = \alpha(s)$
  - **We will only consider deterministic transitions**

- A **path** from a state $s$ in a state transition system
  - A finite or infinite sequence $\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} ...$ s.t. $s = s_0$ and for each $i$, $\alpha_i(s_i, s_{i+1})$ holds
expand_state(s_0)

procedure expand_state(s)
    work_set := ample(s);
    while work_set is not empty do
        choose α ∈ work_set;
        work_set := work_set \ {α}
        s' := α(s);
        if new(s') then
            expand_state(s');
        end if;
        create_edge(s, α, s');
    end while;
end procedure
The function Ample – Goals

- We must find a systematic way of calculating \textbf{ample}(s) for any given state \( s \)
- The calculation of \textbf{ample}(s) needs to satisfy three goals:
  1. Sufficiently many behaviors must be present in the reduced state graph
    - so that the model checking algorithm gives correct results
  2. The reduced state graph should be significantly smaller than the full one
  3. The overhead of computing ample must be reasonably small
Independence

- An **independence relation** $I \subseteq T \times T$ is a symmetric, antireflexive relation, satisfying the following 2 conditions:
  - (Enabledness)
    \[ \alpha, \beta \in \text{enabled}(s) \rightarrow \alpha \in \text{enabled}(\beta(s)) \]
  - (Commutativity)
    \[ \alpha, \beta \in \text{enabled}(s) \rightarrow \alpha(\beta(s)) = \beta(\alpha(s)) \]

- **Notes**
  - Note that the definition makes use of the fact that $I$ is symmetric
  - The enabledness condition states that a pair of independent transitions do not disable each other
    - However, one can enable another
  - The commutativity condition
    - is well defined due to the enabledness condition
    - executing independent transitions from a state $s$ in either order results in the same state $s'$

- The dependency relation $D$ is the complement of $I$
  \[ D = (T \times T) \setminus I \]
Independence

- The relation of independence has to be passed explicitly as a parameter of the model checking procedure
  - The information is obtained from
    - The model of the computation
    - Knowledge of the modeled system
- When it is hard to check whether $\alpha$, $\beta$ are independent
  - ... assuming they are dependent always preserves the correctness
- Even actions which cannot be executed in parallel can be considered to be independent
  - e.g. different processes incrementing a shared variable
Independence example: SPIN

- Let $\alpha$, $\beta$ be transitions performed by different processes; in the following cases, $\alpha$, $\beta$ are independent:
  - $\alpha$ accesses a local variable of its process, $\beta$ is an arbitrary transition
  - $\alpha$, $\beta$ access two different global variables or channels
    - This case includes sending and receiving messages on different channels, as well as testing length of different channels
  - $\alpha$, $\beta$ read the same global variable (or test length of the same channel)
  - $\alpha$ is a send operation on a channel $\text{chan}$, $\beta$ is a receive operation on $\text{chan}$, provided that
    - $\text{chan}$ is asynchronous
    - default behavior of send is used (i.e. send on a full channel is blocked)

- Note that the independence relation is symmetric
Is independence enough? – NO

- **Problem 1**: The checked property might be sensitive to the choice between the states $s_1, s_2$, not only the states $s, r$
- **Problem 2**: The states $s_1, s_2$ may have other successors in addition to $r$, which may not be explored if either one is eliminated

- **Addressing problems 1 and 2**:
  - Definition of *transition invisibility* and *stuttering equivalence*
  - Definition of conditions (C0 – C3) to be satisfied by function $ample(s)$
  - Partial order reduction for LTL $\neg x$
Transition invisibility

- \( L: S \rightarrow 2^{AP} \) ... labeling function
- A transition \( \alpha \in T \) is **invisible with respect to a set** \( AP' \subseteq AP \) if
  - for each pair of states \( s, s' \in S \) s.t. \( s' = \alpha(s) \) it holds that \( L(s) \cap AP' = L(s') \cap AP' \)
- A transition \( \alpha \in T \) is **invisible** if it is invisible with respect to \( AP \)
- A transition is **visible** if it is not invisible
- The relation of invisibility has to be passed explicitly as a parameter of the model checking procedure
  - The information is obtained from
    - The model of the computation
    - Knowledge of the modeled system
Two infinite paths

\[ \sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} ... \]
\[ \rho = r_0 \xrightarrow{\beta_0} r_1 \xrightarrow{\beta_1} ... \]

are stuttering equivalent, denoted \( \sigma \sim_{st} \rho \), if there are two infinite sequences of positive integers

\[ 0 = i(0) < i(1) < i(2) < ... \]
\[ 0 = j(0) < j(1) < j(2) < ... \]

s.t. for every \( k \geq 0 \),

\[ L(s_{i(k)}) = L(s_{i(k)+1}) = ... = L(s_{i(k+1)-1}) = \]
\[ = L(r_{j(k)}) = L(r_{j(k)+1}) = ... = L(r_{j(k+1)-1}) \]
A finite sequence of identically labeled states is called a **block**.

Intuitively, two paths are stuttering equivalent when they can be partitioned into infinitely many blocks, s.t. the states in the \(k\)-th block of one are labeled the same as the states in the \(k\)-th block of the other:

- Corresponding blocks can have different lengths.
Stuttering equivalence can be defined in a similar way for finite paths using finite sequences of indexes:

- \( 0 = i(0) < i(1) < i(2) < ... i(n) \)
- \( 0 = j(0) < j(1) < j(2) < ... j(n) \)

Stuttering is a particularly important concept for asynchronous systems:

- there is no correlation between the time separating two events and the number of transitions occurring between them

Two structures \( M, M' \) (Kripke structures or state transition systems) are stuttering equivalent iff:

- \( M \) and \( M' \) have the same set of initial states
- for each path \( \sigma \) of \( M \) that starts from an initial state \( s \) of \( M \) there exists a path \( \sigma' \) of \( M' \) from the same initial state \( s \) such that \( \sigma \sim_{st} \sigma' \)
- for each path \( \sigma' \) of \( M' \) that starts from an initial state \( s \) of \( M' \) there exists a path \( \sigma \) of \( M \) from the same initial state \( s \) such that \( \sigma' \sim_{st} \sigma \)
Recall: LTL

- An LTL formula has one of the following forms (\( \varphi \) and \( \psi \) are path formulae):
  - 0, 1, \( p \), \( \neg \varphi \), \( \varphi \land \psi \), \( \varphi \lor \psi \), \( \varphi \Rightarrow \psi \)
  - (for any variable \( p \in AP \))
  - \( X \varphi \)
  - \( G \varphi \)
  - \( F \varphi \)
  - \( \varphi U \psi \)
An LTL formula $f$ is **invariant under stuttering** if and only if for each pair of paths $\pi$ and $\pi'$ such that $\pi \sim_{st} \pi'$

- $\pi \models f$ if and only if $\pi' \models f$

We denote the subset of the logic LTL without the next operator by $\text{LTL}_{-X}$

**Theorem:** Any $\text{LTL}_{-X}$ property is invariant under stuttering

**Theorem:** Every LTL property that is invariant under stuttering can be expressed in $\text{LTL}_{-X}$
Stuttering equivalence in LTL

- **Theorem:** Let $M, M'$ be two stuttering equivalent structures. Then, for every LTL-$\chi$ property $f$, and every initial state $s \in S_0$, $M, s \models f$ if and only if $M', s \models f$

- **Exploitation:** partial order reduction generates a structure that is stuttering equivalent to the full graph

- **Example:** Suppose that $\alpha$ is invisible
  - $L(s) = L(s_1)$ and $L(s_2) = L(r)$
  - Consequently, $s \sim_{st} s_1 r \sim_{st} s_2 r$
Partial order reduction for $\text{LTL}_{\rightarrow}$

- The specification is invariant under stuttering
- A systematic way of generating ample sets
- The ample sets are used by the DFS algorithm to construct a reduced state graph which is stuttering equivalent to the full state graph (never constructed)
Partial order reduction for LTL\(_{-X}\)

- \( s \in S \)
- A state \( s \) is **fully expanded** when
  \[
  \text{ample}(s) = \text{enabled}(s)
  \]
- 4 conditions (C0 – C3) for selecting
  \[
  \text{ample}(s) \subseteq \text{enabled}(s)
  \]
- The reduction depends on the set of propositions \( AP' \) that appear in the LTL\(_{-X}\) formula
Partial order reduction for LTL_{\neg X}

\textbf{C0:} \textit{ample}(s) = \emptyset \iff \textit{enabled}(s) = \emptyset

\textbf{C1:} Along every path in the \textbf{full state graph} that starts at s, it holds that a transition that is dependent on a transition in \textit{ample}(s) cannot be executed without a transition in \textit{ample}(s) occurring first.

\textbf{C2:} If s is not fully expanded, then every \( \alpha \in \textit{ample}(s) \) is invisible.

\textbf{C3:} A cycle is not allowed if it contains a state in which some transition \( \alpha \) is enabled, but is never included in \textit{ample}(s) for any state s of the cycle.
Partial order reduction for LTL\(_{-X}\)

- Addressing our two problems
  - Assume that \(\text{ample}(s) = \{\beta\}\)
  - \(s_1\) not included in the reduced graph

- **Problem 1:** The checked property might be sensitive to the choice between the states \(s_1, s_2\), not only the states \(s, r\)
  - By C2, \(\beta\) is invisible \(\rightarrow\) stuttering equivalence
- Addressing our two problems
  - Assume that \( \text{ample}(s) = \{\beta\} \)
  - \( s_1 \) not included in the reduced graph

- **Problem 2:** The states \( s_1, s_2 \) may have other successors in addition to \( r \), which may not be explored if either is eliminated
  - \( \chi \) cannot be dependent on \( \beta \), otherwise \( (\alpha, \chi) \) violates C1
  - \( \chi \) is enabled in \( r \)
  - \( \beta \) is invisible (C2)
  - \( (s, s_1, s_1'), (s, s_2, r, r') \) are stuttering equivalent
Partial order reduction for LTL\(_{-X}\)

- Justification of C3:
  - \(\beta\) independent of the transitions \(\alpha_1, \alpha_2, \alpha_3\)
  - \(\alpha_1, \alpha_2, \alpha_3\) are interdependent
  - There is a proposition \(p\), which is changed from true to false by \(\beta\)
    - \(\Rightarrow \beta\) is visible
  - \(\alpha_1, \alpha_2, \alpha_3\) are invisible

\(\beta\)
A problem (cont.)

- \( \text{ample}(s_1) = \{\alpha_1\} \)
- \( \text{ample}(s_2) = \{\alpha_2\} \)
- \( \text{ample}(s_3) = \{\alpha_3\} \)
- The conditions \( C_0, C_1, C_2 \) are satisfied
- Transition \( \beta \) is ignored – we need \( C_3 \) as well
Java PathFinder

• Explicit code model checker for Java
  • Symbolic execution in latest version

• Special virtual machine performing the verification
  • All executions with respect to thread interleaving and non-deterministic choices (random())
Implementation in JPF

- Too many possible interleavings
  - Exponential in number of threads

- Idea: switch context only when it makes sense
  - Sequential update of local variables does not affect other threads’ state
  - JPF executes instructions of current thread until the next instruction is either
    - “Scheduling relevant” instruction or
    - “Nondeterministic” instruction (random() )
Implementation in JPF

- **Scheduling relevant instructions:**
  - Only about 10% of instructions
  - Synchronization (monitorEnter, monitorExit, invokeX on synchronized methods)
  - Field access (putX, getX)
  - Array element access (Xaload, Xastore)
  - Thread instructions (start, sleep, yield, join)
  - Object methods (wait, notify)

- Rescheduling only if there is another runnable thread