Logic Model Checking: What is it about?
- The Basic Method
- General Remarks
- Motivating Examples

Automata and Logic
- Finite State Automata
- Büchi Automata
- Something on Logic and Automata
- Implications in Model Checking
- Automata Products

Model Checking Algorithm
- Preliminaries
- The Algorithm

Final Remarks
- Something on Automata

Logic Model Checking (1)
- Model checking is a technique for verifying finite-state concurrent systems
- Theoretically speaking, model checking consists of the following tasks:
  - Modeling the system
    - It may require the use of abstraction
    - Often using some kind of automaton
  - Specifying the properties the design must satisfy
    - It is impossible to determine all the properties the systems should satisfy
    - Often using some kind of temporal logic
  - Verifying that the system satisfies its specification
    - In case of a negative result: error trace
    - An error trace may be product of a specification error (false negative)
Logic Model Checking (2)

The application of model checking in a design project typically consists of the following steps:

1. Choose the properties (correctness requirements) critical to the design
2. Build a verification model guided by the above correctness requirements
   - The model should be as smallest as possible
   - It should, however, capture everything which is relevant to the properties to be verified
3. Select the appropriate verification method based on the model and the properties
4. Refine the verification model and correctness requirements until all correctness concerns are adequately satisfied

The main causes of combinatorial complexity in SPIN/Promela are:

- The number of and size of buffered channels
- The number of asynchronous processes

---

The Basic Method

There are different model checking techniques. We will use SPIN which is based on the automata-theoretic approach:

- **System:** \( \mathcal{L}(S) \) (the set of possible behaviors of S)
- **Property:** \( \mathcal{L}(P) \) (the set of valid/desirable behaviors)
- Prove that \( \mathcal{L}(S) \subseteq \mathcal{L}(P) \) (everything possible is valid)
- **Method**
  - Let \( \overline{\mathcal{L}(P)} \) be the language \( \Sigma^\omega \ \setminus \ \mathcal{L}(P) \) of words not accepted by \( P \)
  - Prove \( \mathcal{L}(S) \cap \overline{\mathcal{L}(P)} = \emptyset \)
- There is no accepted word by \( S \) disallowed by \( P \)

This will be clear at the end of the talk, .... I hope

---

Scope of the Method

- Logic model checkers (LMC) are suitable for concurrent and multi-threading finite state systems
- Some of the errors LMC may catch:
  - Deadlocks (two or more competing processes are waiting for the other to finish, and thus neither ever does)
  - Livelocks (two or more processes continually change their state in response to changes in the other processes)
  - Starvation (a process is perpetually denied access to necessary resources)
  - Priority and locking problems
  - Race conditions (attempting to perform two or more operations at the same time, which must be done in the proper sequence in order to be done correctly)
  - Resource allocation problems
  - Dead code (unreachable code)
  - Violation of certain system bounds
  - Logic problems: e.g, temporal relations
A Bit of History

The following diagram shows the evolution of the theoretical foundations of LMC:

1936: First theory on computability, e.g., Turing machines
1940-50: The first computers are built
1955: Early work on tense logics (predecessors of LTL)
1955: Early work on software crisis and software engineering
1958: LTL (linear temporal logic) for system verification
1964: Edsger Dijkstra's paper on Guarded Command Languages
1968: Two terms introduced: software crisis and software engineering
1975: Amir Pnueli introduces linear temporal logic for system verification
1976: Pnueli's seminal paper on temporal logic for system verification
1976-1979: First experiments with reachability analyzers (e.g., Jan Hajek: 'Approver')
1978: Tony Hoare's paper on Communicating Sequential Processes
1980: Earliest predecessor of Spin: 'Pan' (Bell Labs)
1989: Spin version 0 verification of class of $\mathcal{A}$-regular properties
1993: BDDs and the SMV model checker (Ken McMillan, CMU)
1995: Partial order reduction in Spin. LTL conversion in Spin. (Doron Peled)
2001: Support for embedded C code in Spin version 4.0
2003:標量 first search mode added to Spin version 4.1

The two most popular logic model checking systems today:
- Spin: an explicit state LTL model checker based on automata theoretic verification method targeting software verification (asynchronous systems)
- SMV: a symbolic CTL model checker targeting hardware circuit verification (synchronous systems)

On Correctness (reminder)

- A system is correct if it meets its design requirements.
- There is no notion of “absolute” correctness: It is always w.r.t. a given specification
- Getting the properties (requirements) right is as important as getting the model of the system right
- Examples of correctness requirements
  - A system should not deadlock
  - No process should starve another
  - Fairness assumptions
    - E.g., an infinite often enabled process should be executed infinitely often
  - Causal relations
    - E.g., each time a request is send, and acknowledgment must be received

Building Verification Models

- Statements about system design and system requirement must be separated
  - One formalism for specifying behavior (system design)
  - Another formalism for specifying system requirements (correctness properties)
- The two types of statements define a verification model
- A model checker can now
  - Check that the behavior specification (the design) is logically consistent with the requirement specification (the desired properties)
Distributed Algorithms

Two asynchronous processes may be easily get blocked when competing for a shared resource in real-life conflicts ultimately get resolved by human judgment. Computers, though, must be able to resolve it with fixed algorithms.

Thread Interleaving

- The number of possible thread interleavings is...
  - 3! = 3 · 2 · 1 = 6
  - 3! = 3 · 2 · 1 = 6
  - 3! = 3 · 2 · 1 = 6
  - 3! = 3 · 2 · 1 = 6
  - Total: 3! · 3! · 3! = 1,680 possible executions.

- Are all these executions okay?
- Can we check them all? Should we check them all?
- In classic system testing, how many would normally be checked?

A Small Multi-threaded Program

```c
int x, y, z;
int *p, *q, *r;
int **a;

thread_1(void) /* initialize p, q, and r */
{
  p = &x;
  q = &y;
  z = &r;
}

thread_2(void) /* swap contents of x and y */
{
  r = *p;
  *p = *q;
  *q = r;
}

thread_3(void) /* access z via a and p */
{
  a = &p;
  *a = z;
  **a = 12;
}
```

A Simpler Example

- Consider two 2-state automata:
  - Representing two asynchronous processes
  - One can print an arbitrary number of '0' digits, or stop.
  - The other can print an arbitrary number of '1' digits, or stop.

- How many different combined executions are there? I.e., how many different binary numbers can be printed? How would one test that this system does what we think it does?
**Finite State Automata**

**Definition**

A finite state automaton is a tuple \( (S, s_0, L, F, T) \), where

- \( S \) is a finite set of states
- \( s_0 \in S \) is a distinguished initial state
- \( L \) is a finite set of labels (symbols)
- \( F \subseteq S \) is the set of final states
- \( T \subseteq S \times L \times S \) is the transition relation, connecting states in \( S \)

We will, in general, follow Holzmann’s notation: \( A.S \) denotes the state \( S \) of automaton \( A \), \( A.T \) denotes the transition relation \( T \) of \( A \), and so on.

If understood from the context, we will avoid the use of \( A._. \)

**Example**

The above automaton may be interpreted as a Process Scheduler:

- \( A.S: \{ s_0, s_1, s_2, s_3, s_4 \} \)
- \( A.L = \{ \alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5 \} \)
- \( A.F = \{ s_4 \} \)
- \( A.T = \{ (s_0, \alpha_0, s_1), (s_1, \alpha_1, s_2), \ldots \} \)

**Determinism vs non-determinism**

A finite state automaton \( A = (S, s_0, L, F, T) \) is deterministic iff

\[
\forall s \forall l, ((s, l, s') \in A.T \land (s, l, s'') \in A.T) \implies s' \equiv s''
\]

I.e., the destination state of a transition is uniquely determined by the source state and the transition label. An automaton is called non-deterministic if it does not have this property.

**Example:** The automaton corresponding to the process scheduler is deterministic.
Definition of a Run

A run of a finite state automaton $A = (S, s_0, L, F)$ is an ordered and possibly infinite set of transitions (a sequence) from $T$

$$\sigma = \{(s_0, l_0, s_1), (s_1, l_1, s_2), \ldots\}$$

such that

$$\forall i, i \geq 0 \cdot (s_i, l_i, s_{i+1}) \in T$$

Each run corresponds to a state sequence in $S$ and a word in $L$

Example:

A state sequence from a run: $\{idle, ready, \{execute, waiting\}^*\}$

The corresponding word in $L$: $\{start, run, \{block, unblock\}^*\}$

Remarks:
- A single state sequence may correspond to more than one word
- For non-deterministic automata, the same word may correspond to different state sequences

Language Accepted by an Automaton

The language $L(A)$ of automaton $A = (S, s_0, L, F, T)$ is the set of words in $A$ that correspond to the set of all the accepting runs of $A$

Notice that there can be infinitely many words in the language of even a small finite state automaton

Example:

A characterization of the complete language of automaton $A$ (an infinite set of words): $\{start, run, \{pre-empt, run\}^*, \{block, unblock\}^*, \text{stop}\}$

The shortest word in the language: $\{start, run, \text{stop}\}$
Reasoning about Runs

**sample property:**

“if first $p$ becomes true and then later $q$ becomes true, then $r$ can no longer become true”

**interpretation:**

$\overline{p} \cup \overline{q} \rightarrow \overline{r}$

**correctness claim:**

it is an error if in a run we see first $p$ then $q$ and then $r$

this property is easily expressed with the standard definition of acceptance

**reaching this state constitutes a complete match of the pattern that specifies the correctness violation**

Reasoning about *Infinite* Runs

*a classic liveness property:*

“if $p$ then eventually $q$”

**interpreted:**

$\forall i : s_i \rightarrow s_{i+1}$

**error**

**correctness claim:**

it is an error if in a run we see first $p$ then $q$ and then $r$

this property can only be violated by an infinite run…

the standard notion of acceptance applies only to finite runs…

We need, thus, to extend the notion of run, acceptance, …

Büchi Acceptance

- An infinite run is often called an $\omega$-run (“omega run”)
- An acceptance property for $\omega$-runs are called $\omega$-acceptance and can be defined in different ways
  - The so-called Büchi, Müller, Rabin, Streett, etc, acceptance conditions
  - We adopt here the introduced by Büchi (1960)

**Definition**

An accepting $\omega$-run of finite state automaton $A = (S, s_0, L, F, T)$ is an infinite run $\sigma$ such that

$$\exists i \geq 0, (s_{i-1}, l_{i-1}, s_i) \in \sigma \cdot s_i \in L \land s_i \in \sigma^\omega$$

i.e., at least one state in $A, F$ is visited infinitely often.

Automata with the above acceptance condition are called Büchi automata

Büchi Automata

**Example**

an accepting $\omega$-run for this automaton:

$\{ \text{idle, ready, \{execute, ready\}*} \}$

the corresponding $\omega$-word:

$\{ \text{start, run, \{pre-empt, run\}}* \}$

the $\omega$-language of an automaton is the set of all $\omega$-words accepted
Generalized Büchi Automata

Definition
A generalized Büchi automaton is an automaton \( A = (S, s_0, L, F, T) \), where \( F \subseteq \wp(S) \) (\( F = \{ f_1, \ldots, f_n \} \) and \( f_i \subseteq S \)). A run \( \sigma \) of \( A \) is accepting if

\[
\text{for each } f_i \in F, \text{inf}(\sigma) \cap f_i \neq \emptyset.
\]

- A generalized Büchi Automaton differs from a Büchi Automaton by allowing multiple accepting sets instead of only one
- Generalized Büchi automata are not more expressive than usual Büchi automata

The Stutter Extension Rule

Definition
The stutter extension of a finite run \( \sigma \) with final state \( s_n \), is the \( \omega \)-run

\[
\sigma (s_n, \varepsilon, s_n)^\omega
\]

Example
run: \{ idle, ready, execute, waiting, execute, [end,]* \}
word: \{ start, run, block, unblock, stop, \( \varepsilon \)* \}

LTL (reminder)
Semantics

Definition
An LTL formula \( \varphi \) holds for an \( \omega \)-run \( \sigma \), written \( \sigma \models \varphi \) if:

- \( \sigma \models \varphi \) iff \( \sigma_0 \models \varphi \) when \( \varphi \in \text{Prop} \) (Propositional formula)
- \( \sigma \models \neg \varphi \) iff \( \sigma \not\models \varphi \)
- \( \sigma \models \varphi \lor \psi \) iff \( \sigma \models \varphi \) or \( \sigma \models \psi \)
- \( \sigma \models [\varphi] \) iff \( \sigma[k] \models \varphi \) for all \( k \geq 0 \)
- \( \sigma \models [\varphi] \) iff \( \sigma[k] \models \varphi \) for some \( k \geq 0 \)
- \( \sigma \models [\varphi] \) iff \( \sigma[1] \models \varphi \)
- \( \sigma \models \varphi U \psi \) iff \( \sigma[k] \models \psi \) for some \( k \geq 0 \), and \( \sigma[i] \models \varphi \) for every \( i \) such that \( 0 \leq i < k \)
- \( \sigma \models \varphi R \psi \) iff for every \( j \geq 0 \)

- if \( \sigma[j] \not\models \varphi \) for every \( i < j \) then \( \sigma[j] \models \psi \)
From Kripke Structures to Büchi Automata

LTL formulas can be interpreted on sets of infinite runs of Kripke structures.

We recall the definition (slightly different from previous lecture)

**Definition**

A *Kripke structure* \( M \) is a four-tuple \( (W, R, W_0, V) \) where

- \( W \) is a finite non-empty set of states (worlds)
- \( R \subseteq W \times W \) is a total accessibility relation between states (transition relation)
- \( W_0 \subseteq W \) is the set of starting states
- \( V : W \rightarrow \wp(\mathcal{AP}) \) is a map labeling each state with a set of propositional variables

A *path in* \( M \) is an infinite sequence \( \sigma = w_0, w_1, w_2, \ldots \) of worlds such that for every \( i \geq 0 \), \( w_i R w_{i+1} \). One can think of a path as an infinite branch in a tree corresponding to the unwind of the Kripke structure.

---

From Kripke Structures to Büchi Automata

**Example**

A Kripke structure (whose only infinite run is a model to \( \Box q \) and \( \Box \Diamond p \), for instance):

\[
\begin{array}{c}
\rightarrow \\
\text{s}_0 \\
\{p, q\} \\
\rightarrow \\
\text{s}_1 \\
\{q\}
\end{array}
\]

The corresponding Büchi Automaton:

\[
\begin{array}{c}
\rightarrow \\
\text{i} \\
\leftarrow \\
\{p,q\} \\
\rightarrow \\
\text{s}_0 \\
\leftarrow \\
\{q\} \\
\rightarrow \\
\text{s}_1 \\
\leftarrow \\
\{p,q\}
\end{array}
\]

---

From Logic to Automata

For any LTL formula \( \psi \) there exists a Büchi automaton that accepts precisely those runs for which the formula \( \psi \) is satisfied.

**Example:** The formula \( \Diamond \Box p \) corresponds to the following nondeterministic Büchi automaton:

\[
\begin{array}{c}
\rightarrow \\
\text{s}_0 \\
\leftarrow \\
\text{true} \\
\rightarrow \\
\text{s}_1 \\
\leftarrow \\
\text{p}
\end{array}
\]

We will see the algorithm next lecture... For the moment, believe me that it is indeed the case.
Omega-regular Properties

- something not expressible in pure LTL:
  - (p) can hold after an even number of execution steps, but never holds after an odd number of steps
  - ❱ X (p) certainly does not capture it:
    \[ \text{(p) must hold after an even number of execution steps, but never after an odd number of steps) } \]
  - \[ p \land \Diamond \Box (p) \land \Box (\Box (p) \to \Box \Diamond !p) \land \Box (\Diamond !p \to \Box !p) \]
    does not capture it either (because now p must always hold after all even steps):
  - this formula expresses it correctly

Expressiveness

- modal \( \mu \)-calculus
- \( \omega \)-tree automata
- \( \omega \)-word automata
- Büchi automata
- CTL
- CTL*

Implications in Model Checking

- At the beginning we said that the automata-based model checking method was based on the following check:
  \[ \mathcal{L}(S) \cap \overline{\mathcal{L}(P)} = \emptyset \]
  
  where \( S \) is a model of the system and \( P \) of the property
  
  So, the following Büchi automata’s \textit{decidable} properties are important for model checking
    - Language emptiness: are there any accepting runs?
    - Language intersection: are there any run accepted by 2 or more automata?
    - Language complementation

Implements in Model Checking

In theory:
- The system is represented as a Büchi automaton \( A \)
- The automaton corresponds to the asynchronous product of automata \( A_1, \ldots, A_n \) (representing the asynchronous processes)
  \[ A = \bigotimes_{i=1}^{n} A_i \]
- The property is originally given as an LTL formula \( \psi \)
- The property \( \psi \) is translated into a Büchi automaton \( B \)
- We perform the following check:
  \[ \mathcal{L}(A) \cap \overline{\mathcal{L}(B)} = \emptyset \]

But... complementing a Büchi automaton is difficult!

\[ ^1 \text{Alternatively, the property can be given directly as a Büchi automaton} \]
Implications in Model Checking

In practice (e.g., in SPIN) we want to avoid automata complementation:

- Assume $A$ as before
- The negation of the property $\psi$ is automatically translated into a Büchi automaton $\overline{B}$ (since $L(\overline{B}) \equiv L(B)$)
- By making the synchronous product of $A$ and $\overline{B}$ ($B \otimes A$) we can check whether the system satisfies the property $L(A) \cap L(\overline{B}) = \emptyset$

  - If the intersection is empty, the property $\psi$ holds for $A$
  - Otherwise, use an accepted word of the nonempty intersection as a counterexample

Asynchronous Product

Example

- Assume two non-terminating asynchronous processes $A_1$ and $A_2$:
  - $A_1$ tests whether the value of a variable $x$ is even, in which case updates it to $3x + 1$
  - $A_2$ tests whether the value of a variable $x$ is odd, in which case updates it to $x/2$
- Let $\psi$ the following property: $\square \diamond (x \geq 4)$
  - The negation of the formula is: $\diamond \square (x < 4)$

Question: Given an initial value for $x$, does the property hold?

Asynchronous Product

Definition

The asynchronous product $\prod$ of a finite set of finite automata $A_1, \ldots, A_n$ is a new finite state automaton $A = (S, s_0, L, T, F)$ where:

- $A, S$ is the Cartesian product $A_1 \times A_2 \times \ldots \times A_n, S$
- $A, s_0$ is the $n$-tuple $(A_1, s_0, A_2, s_0, \ldots, A_n, s_0)$
- $A, L$ is the union set $A_1, L \cup A_2, L \cup \ldots \cup A_n, L$
- $A, T$ is the set of tuples $(((x_1, \ldots, x_n), l, (y_1, \ldots, y_n))$ such that $\exists i, 1 \leq i \leq n, (x_i, l, y_i) \in A_i, T$ and $\forall j, 1 \leq j \leq n, j \neq i \implies (x_j \equiv y_j)$
- $A, F$ contains those states from $A, S$ that satisfy $\forall (A_1, s, A_2, s, \ldots, A_n, s) \in A, F, \exists i, 1 \leq i \leq n, A_i, s \in A_i, F$

Asynchronous Product

Example

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$A_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>$s_0$</td>
</tr>
<tr>
<td>$(x \equiv 2)$</td>
<td>$x = 3x + 1$</td>
</tr>
<tr>
<td>$x = x/2$</td>
<td>$x &lt; 4$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
</tr>
<tr>
<td>$(x \equiv 2)$</td>
</tr>
<tr>
<td>$x = 3x + 1$</td>
</tr>
<tr>
<td>$x = x/2$</td>
</tr>
<tr>
<td>$x &lt; 4$</td>
</tr>
</tbody>
</table>

An unreachable state under Promela interpretation of statement (label) semantics.

We can also "expand" the automaton into a pure automaton, without variables.
Asynchronous Product Example

![Asynchronous Product Diagram](image)

Synchronous Product

Definition

The **synchronous product** $\otimes$ of a finite set of two finite automata $P$ and $B$ is a new finite state automaton $A = (S, s_0, L, T, F)$ where:

- $A.S$ is the Cartesian product $P'.S \times B.S$ where $P'$ is the stutter closure of $P$
  - A self-loop labeled with $\varepsilon$ is attached to every state in $P$ without outgoing transitions in $P.T$
- $A.L$ is the set of pairs $(l_1, l_2)$ such that $l_1 \in P'.L$ and $l_2 \in B.L$
- $A.T$ is the set of pairs $(t_1, t_2)$ such that $t_1 \in P'.T$ and $t_2 \in B.T$
- $A.F$ is the set of pairs $(s_1, s_2)$ such that $s_1 \in P'.F$ or $s_2 \in B.F$

Remarks

- Not all the states in $A.S$ are necessary reachable from $A.s_0$
  - Their reachability depends on the semantics given to the labels in $A.L$ (the interpretation of the labels depends on Promela semantics as we'll see in a future lecture)
- The transitions in the product automaton are the transitions from the component automata arranged such that only one of the components automata can execute at a time
  - This gives an interleaving semantics of the processes
- Promela has also rendez-vous synchronization (A special global variable has to be set)
  - Some transitions may synchronize by sending and receiving a message
- For hardware verification, the asynchronous product is defined differently: each of the components with enabled transitions is making a transition (simultaneously)

Synchronous Product Example

![Synchronous Product Diagram](image)
Synchronous Product

Remarks
- We require the stutter-closure of \( P \) since \( P \) is a finite state automaton (the asynchronous product of the processes automata) and \( B \) is a standard Büchi automaton obtained form a LTL formula.
- Not all the states in \( A.S \) or \( A.F \) are necessary reachable from \( A.s_0 \).
- The main difference between asynchronous and synchronous products are on the definitions of \( L \) and \( T \) – In a synchronous product:
  - The transitions correspond to joint transitions of the component automata.
  - The labels are pairs: the combination of the two labels of the original transitions in the component automata.
- In general \( P \otimes B \neq B \otimes P \), but given that in SPIN \( B \) is particular kind of automaton (labels are state properties, not actions), we have then \( P \otimes B \equiv B \otimes P \).

Strongly-connected Components

Example

- Strongly-connected subsets:
  \( S = \{s_0, s_1\}, \quad S' = \{s_1, s_3, s_4\}, \quad S'' = \{s_0, s_1, s_3, s_4\} \)
- Strongly-connected components: Only \( S'' = \{s_0, s_1, s_3, s_4\} \)

Strongly-connected Components

Definition
A subset \( S' \subseteq S \) in a directed graph is strongly-connected if there is a path between any pair of nodes in \( S' \), passing only through nodes in \( S' \).

A strongly-connected component (SCC) is a maximal set of such nodes, i.e. it is not possible to add any node to that set and still maintain strong connectivity.

Checking Emptiness

- Let \( \sigma \) be an accepting run of a Büchi automaton
  \( A = (S, s_0, L, T, \bar{F}) \)
  - Since \( S \) is finite, there is some suffix \( \sigma' \) of \( \sigma \) s.t. every state on \( \sigma' \) is reachable from any other state on \( \sigma' \).
  - I.e., the states on \( \sigma' \) are contained in a SCC of the graph of \( A \).
  - This component is reachable from an initial state and contains an accepting state.

- Thus, checking non-emptiness of \( L(A) \) is equivalent to finding a SCC in the graph of \( A \) that is reachable from an initial state and contains an accepting state.

- There are different algorithms for finding SCC. E.g.:
  - Tarjan’s version of the depth-first search (DFS) algorithm
  - SPIN nested depth-first search algorithm

- If the language \( L(A) \) is non-empty, then there is a counterexample which can be represented in a finite way.
  - It is ultimately periodic, i.e., it is of the form \( \sigma_1 \sigma_2^\omega \), where \( \sigma_1 \) and \( \sigma_2 \) are finite sequences.
Model Checking Algorithm

Let $A$ be the automaton specifying the system and $\overline{B}$ the automaton corresponding to the negation of the property $\psi$.

1. Construct the intersection automaton $C = A \cap \overline{B}$
2. Apply an algorithm to find SCCs reachable from the initial states of $C$
3. If none of the SCCs found contains an accepting state, the model $A$ satisfies the property/specification $\psi$
4. Otherwise,
   - Take one strongly-connected component $SC$ of $C$
   - Construct a path $\sigma_1$ from an initial state of $C$ to some accepting state $s$ of $SC$
   - Construct a cycle from $s$ and back to itself (such cycle exists since $SC$ is a strongly-connected component)
   - Let $\sigma_2$ be such cycle, excluding its first state $s$
   - Announce that $\sigma_1\sigma_2\omega$ is a counterexample that is accepted by $A$, but it is not allowed by the property/specification $\psi$

Kripke Structures and Büchi Automata

Observation

- In Peled’s book “Software Reliability Methods” the definition of a Büchi automaton is very similar to our Kripke structure, with the addition of acceptance states
- There is a labeling of the states associating to each state a set of subsets of propositions (instead of having the propositions as transition labels)
- We have chosen to define Büchi Automata in the way we did since this definition is compatible with the implementation of SPIN
- It was taken from Holzmann’s book “The SPIN Model Checker”

Further Reading

- The first two parts of this lecture were mainly based on Chap. 6 of Holzmann's book “The SPIN Model Checker”
  - Automata products: Appendix A
- The 3rd part was taken from Peled's book

For next lecture (29.03.2006): Read Chap. 6 of Peled's book, mainly section 6.8 on translating LTL into Automata
- We will see how to apply the algorithm to an example