# 272: Software Engineering
# Fall 2018

Instructor: Tevfik Bultan

## Lecture 15: Runtime Monitoring

# How to Enforce Specifications?

- We discussed design-by-contract approach which provides a way of organizing and writing interface specifications for object oriented programs

- Today we will also discuss temporal logics which provide a way of specifying expected ordering of events during program executions

- We discussed that one can infer specifications of program behavior by observing a set of program executions

Bottom line: All these approaches can be used to obtain a set of specifications about the expected behavior of a program

- What are we going to do with these specifications?
  - Shouldn’t we make sure that the program behaves according to its specifications?
    - How are we going to do that?

# Runtime Monitoring

- The basic idea in runtime monitoring is to observe the program behavior during execution and make sure that it does not violate the specifications
  - Sometimes it is called *runtime verification*

- We already discussed this for the design-by-contract approach
  - The pre, post-conditions and class invariants written within the scope of the design-by-contract approach can be monitored at runtime by instrumenting the program and checking the specified conditions at appropriate times
  - Eiffel compiler supports this (since Eiffel languages supports the the design-by-contract approach)
  - There are tools for other programming languages (like Java) that automatically instrument Java programs for runtime monitoring of design-by-contract specifications

# Runtime Monitoring of Assertions

- In general, monitoring of design-by-contract specifications correspond to monitoring of assertions
  - Create an assertion for pre-condition (and class invariant) checks at each method call location
  - Create an assertion for post-condition (and class invariant) checks at each method return location
  - For each assertion, when the program execution reaches the location of the assertion, evaluate the assertion.
    - If the assertion evaluates to true continue execution (no violation).
    - If the assertion evaluates to false, stop execution and report the assertion violation.
  - When reporting the assertion violation in design-by-contract approach, we can also appropriately assign the blame:
    - Pre-condition violation: Blame the caller
    - Post-condition violation: Blame the callee

# Runtime Monitoring of Assertions

- While converting design-by-contract specifications to assertion checks, we need to take care of **old** and **result** primitives in the post-condition specifications
  - Store values of variables that are referenced with the **old** primitive at the method entry
  - Compute the return value before evaluating the post-condition

- For runtime monitoring of JML specifications, expressions that involve quantification (forall, exists, sum, etc.) must be converted to code that evaluates the expression

# Beyond Assertions

- What if we want to do more than monitoring assertions?

- For example, we may have specifications such as:
  - The method “close-file” should only be called after the method “open-file” is called
  - This specification is not an assertion
  - It is specifying an ordering of events, not a condition that needs to hold at a specific point in program execution (which is what an assertion does)

# Temporal Logics

- We can use temporal logics such LTL (linear temporal logic) to specify ordering of events

- There are different variants of LTL for runtime monitoring:
  - Past time LTL has temporal operators such as
    - Previously
    - Eventually in the past
    - Always in the past
    - Since

- The question is how do we monitor temporal properties?
  - Temporal logic specifications can be converted to state machines (finite state automata)

# Execution Paths

- An execution path is an infinite sequence of states

  $x = s_0, s_1, s_2, \ldots$

  such that

  $s_0 \in I$ and for all $i \geq 0$, $(s_i, s_{i+1}) \in R$

Notation: For any path x

$x_i$ denotes the i'th state on the path (i.e., $s_i$)

$x^i$ denotes the i'th suffix of the path (i.e., $s_i, s_{i+1}, s_{i+2}, \ldots$ )

# Temporal Logics

- Pnueli proposed using temporal logics for reasoning about the properties of reactive systems

- Temporal logics are a type of modal logics
  - Modal logics were developed to express modalities such as “necessity” or “possibility”
  - Temporal logics focus on the modality of temporal progression

- Temporal logics can be used to express, for example, that:
  - an assertion is an invariant (i.e., it is true all the time)
  - an assertion eventually becomes true (i.e., it will become true sometime in the future)

# Temporal Logics

- We will assume that there is a set of basic (atomic) properties called AP
  - These are used to write the basic (non-temporal) assertions about the program

- We will use the usual boolean connectives: $\neg$ , $\wedge$ , $\vee$

- We will also use four temporal operators:

| | | | | |
|---|---|---|---|---|
| **Invariant** *p* | : | G *p* | (aka $\Box p$) | (Globally) |
| **Eventually** *p* | : | F *p* | (aka $\Diamond p$) | (Future) |
| **Next** *p* | : | X *p* | (aka $\bigcirc p$) | (neXt) |
| *p* **Until** *q* | : | *p* U *q* | | |

# Linear Time Temporal Logic (LTL) Semantics

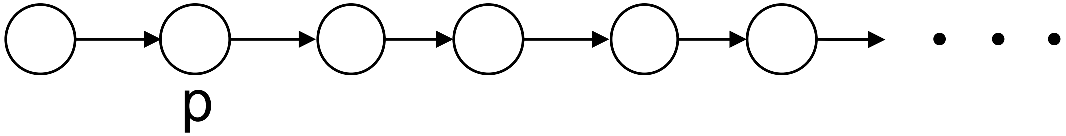
Given an execution path x and LTL properties p and q

$x |= p$ iff $L(x_0, p) = \text{True}$, where $p \in AP$
$x |= \neg p$ iff not $x |= p$
$x |= p \wedge q$ iff $x |= p$ and $x |= q$
$x |= p \vee q$ iff $x |= p$ or $x |= q$

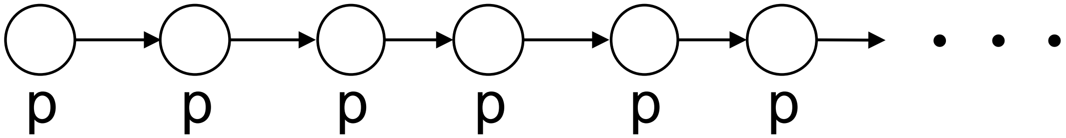
$x |= X\,p$ iff $x^1 |= p$
$x |= G\,p$ iff for all i, $x^i |= p$
$x |= F\,p$ iff there exists an i such that $x^i |= p$
$x |= p\,U\,q$ iff there exists an i such that $x^i |= q$ and
for all $j < i$, $x^j |= p$

# LTL Properties

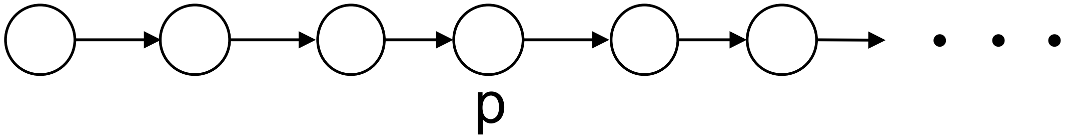
X p



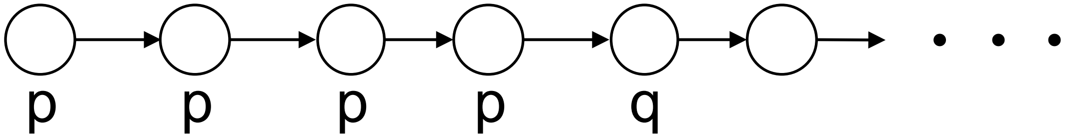
G p



F p



p U q



# Example Properties

mutual exclusion:

Assume that pc1 is the program counter for process 1 and pc2 is the program counter for process 2

Then, mutual exclusion can be specified in LTL as:

$G\ (\neg\ (pc1=c \wedge pc2=c))$

Two processes are not in the critical section at the same time

starvation freedom:

$G(pc1=w \Rightarrow F(pc1=c)) \wedge G(pc2=w \Rightarrow F(pc2=c))$

# Example Properties

starvation freedom:

$$G(pc1=w \Rightarrow F(pc1=c)) \wedge G(pc2=w \Rightarrow F(pc2=c))$$

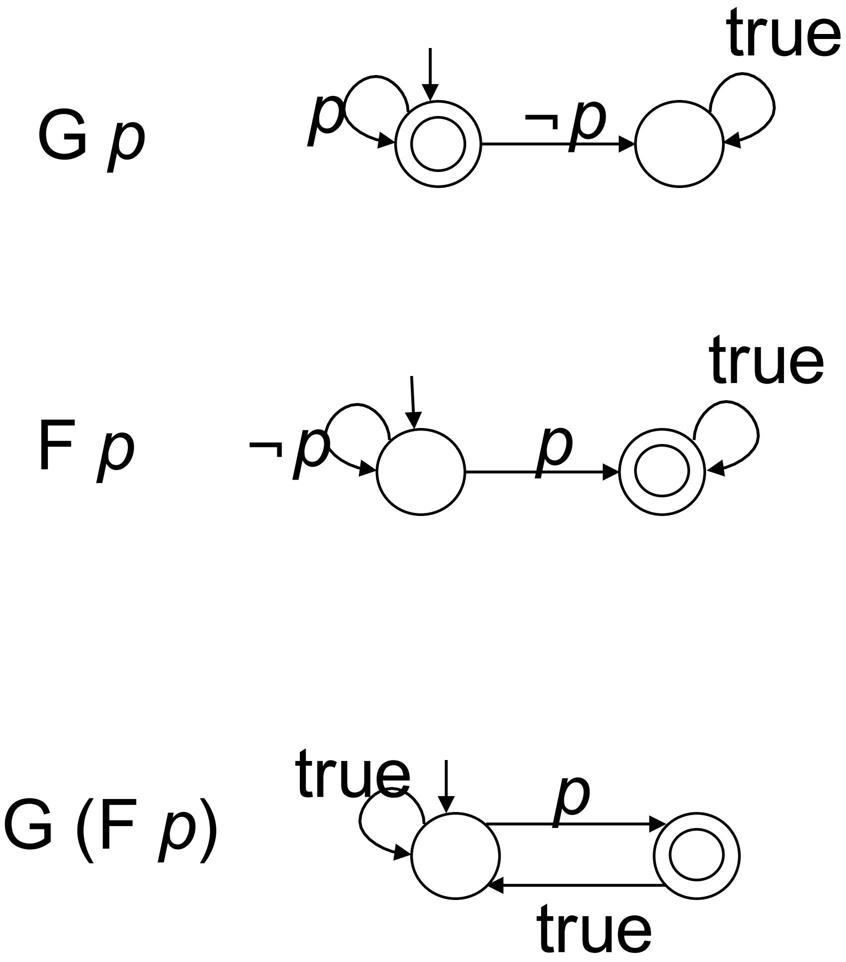
If a process starts waiting to enter the critical section (pc1=w), then it will eventually get in the critical section (pc1=c).

# LTL Properties ≡ Büchi automata

- Büchi automata: Finite state automata that accept infinite strings

- A Büchi automaton accepts a string when the corresponding run visits an accepting state infinitely often

- The size of the property automaton can be exponential in the size of the LTL formula



# Temporal Logics to State Machines

- We can convert temporal logic specifications to automata and track the current state of the specification automata during the program execution
  - If the specification automaton goes to a sink state, then we can report a violation
    - a sink state is a state from which there is no path to any accepting state
  - At the program termination, we can check if the specification automaton is at an accepting state
    - This is assuming that we are using finite path semantics
      - recall that standard semantics for temporal logics assume infinite paths, but it is also possible to define finite paths semantics

# State Machines as Specifications

- We can also use state machines directly as specifications

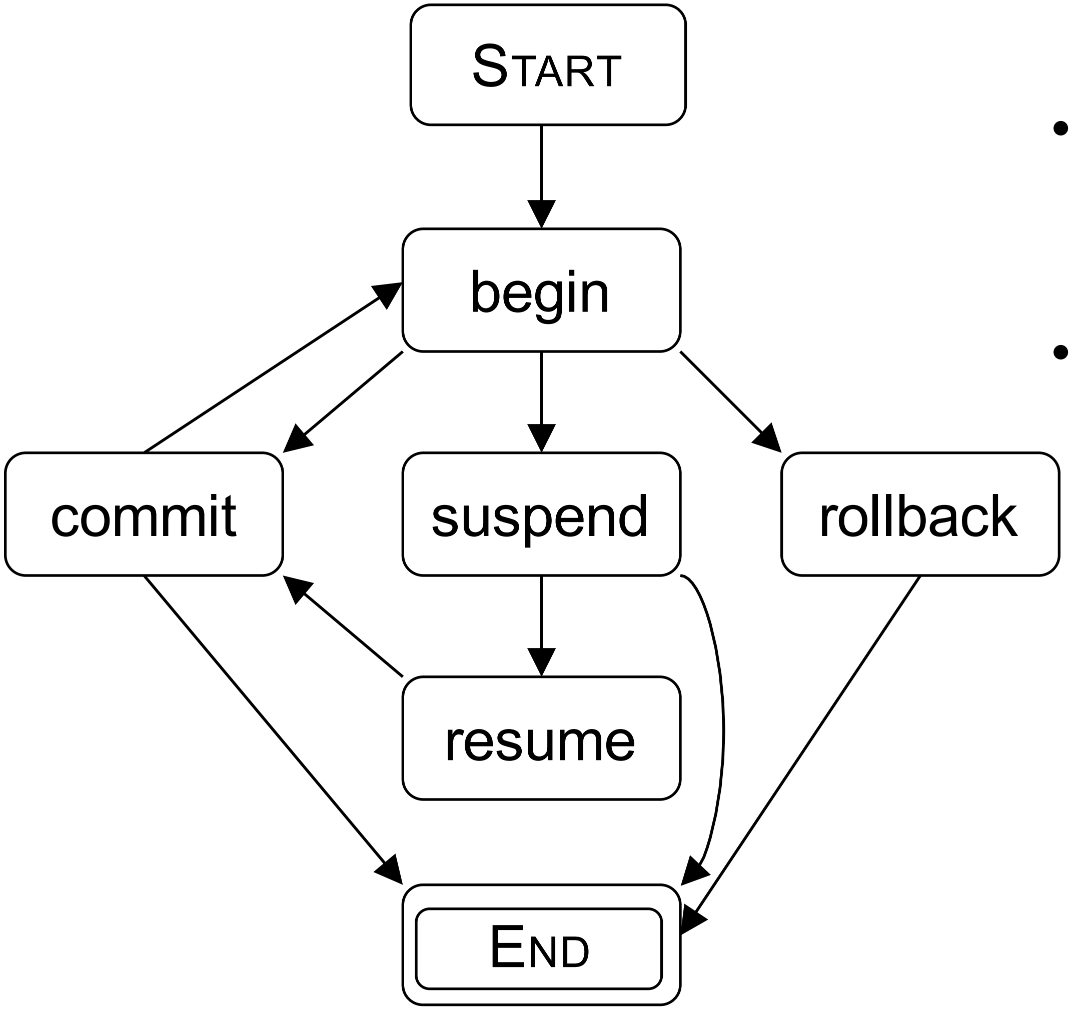
- State machines are useful for specifying ordering of events and can be useful for specifying interfaces

- For examples, given a class, we may want to figure out what are the allowed orderings of method calls to the methods of that class
  - This can be specified as a state machine

- There has been research on automatically extracting such interfaces from existing code
  - Dynamically: By observing program execution and recording ordering of method class
  - Statically: By statically analyzing code and identifying the method call orderings that do not cause exceptions

# An automatically extracted state machine

J2EE TransactionManager
class interface

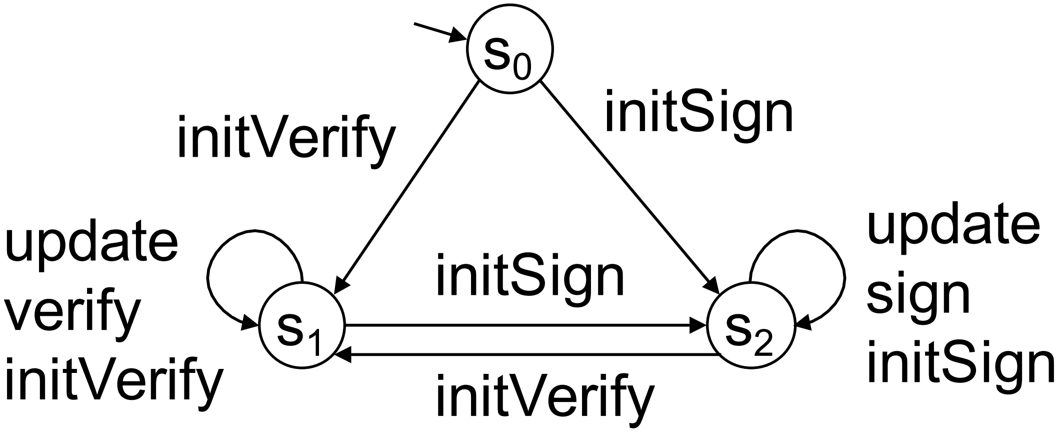


- An example state machine that is dynamically generated
- It provides a specification for the stateful interface of a class
- The states denote the method calls (Start and and End states are special states)
- The paths from start to end identify the acceptable method call orderings

# Another automatically extracted state machine

- A statically extracted interface for the Java class Signature
- The method calls are represented by the transitions
- The paths from the initial state identify the acceptable method call orderings

Signature class
interface



# Beyond State Machines

- As you know, finite state automata can only specify regular languages
- For example, an ordering constraint that specifies nested matching of events cannot be specified using finite state machines
  - For example, each “acquire” call must be matched with a “release” call and “acquire” and “release” calls can be nested
  - This ordering of events is not a regular language
    - It is context free, so it can be specified using a context free grammar (CFG)
- So we can specify such ordering using context-free grammars
  - Then, the question is how can we monitor such ordering constraints at runtime

# Runtime Monitoring with JavaMOP

- This is the problem studied in the following paper:
  - ``Efficient Monitoring of Parametric Context-Free Patterns,” Patrick O'Neil Meredith, Dongyun Jin, Feng Chen and Grigore Rosu. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008).

- There is a tool called JavaMOP, developed by the authors of this paper
  - http://fsl.cs.uiuc.edu/index.php/MOP

- JavaMOP instruments Java programs for runtime monitoring of specifications written using a variety of formalisms including temporal logics, finite state machines, context-free grammars, etc.

# Runtime Monitoring

- There are three ingredients for runtime monitoring systems:
  1. A specification formalism for specifying expected behaviors of the program
  2. A monitor synthesis algorithm that convert specifications about the program behavior to monitors that can be executed with the program and that track if any specified property is violated
  3. A program instrumentor that embeds the synthesized monitors to the program

# Context Free Patterns

- Specifies the appropriate ordering for method calls to a transaction manager

$$\begin{array}{lcl} \textit{Start} & \rightarrow & \textit{Base} \\ \textit{Base} & \rightarrow & \mathbf{\texttt{begin}}\ \textit{Tail Base} \\ & | & \varepsilon \\ \textit{Tail} & \rightarrow & \mathbf{\texttt{commit}} \\ & | & \mathbf{\texttt{rollback}} \end{array}$$

  - Method calls are the events which correspond to the terminal symbols of the grammar

# An Example

- Consider the call sequence

**`begin rollback begin commit`**

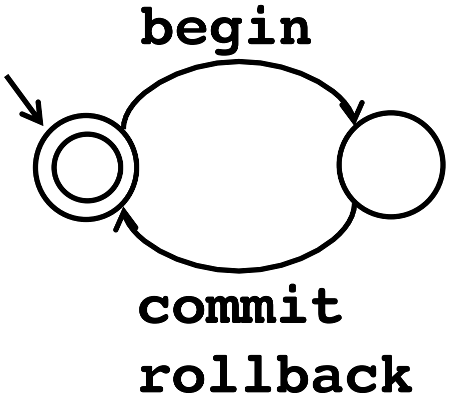
- Here is a derivation:

*Start*
⇒ *Base*
⇒ **`begin`** *Tail Base*
⇒ **`begin rollback`** *Base*
⇒ **`begin rollback begin`** *Tail Base*
⇒ **`begin rollback begin commit`** *Base*
⇒ **`begin rollback begin commit`**

*Start* → *Base*
*Base* → **`begin`** *Tail Base*
| ε
*Tail* → **`commit`**
| **`rollback`**

# Another Example

- This interface can also be specified as a finite state machine (finite state automata)



- However, the following grammar, which specifies ***nested transactions***, cannot be specified as a FSM

*Start* → *Base*
*Base* → **`begin`** *Base Tail Base*
| ε
*Tail* → **`commit`**
| **`rollback`**

# Nesting Requires Context Free Patterns

- If there is a nesting constraint in the property we wish to specify then finite state machines will not work
  - We need to use context free patterns

- Another example, assume that we have “acquire” and “release” calls for a lock
  - Assume that the lock is *reentrant*
  - This means that you can call “acquire” even when you have the lock
    - This is how the locks are in Java
  - The lock is released when the “acquire” and “release” calls cancel each other out

- This cannot be expressed using finite state machines
  - It is a context free pattern

# Monitoring Context Free Patterns

- Given a CFG as a specification
  - Any execution trace that is not a prefix of a word (i.e., a sequence) that is recognized by the CFG violates the specification

- JavaMOP generates monitors from CFG specifications that check the above condition

- The specifications that JavaMOP handles are parametric:
  - There are parameters that can be bound to different objects at runtime
  - So one CFP specification can instantiate multiple monitors at runtime
    - For example, generate a monitor for each lock object or each transaction object based on the specifications we discussed earlier

# Total matching vs. Suffix matching

- An execution is a sequence of events observed up to the current moment (hence, they are always finite)
- *Total matching* corresponds to checking the desired property against the whole execution trace
  - Total matching returns:
    - Valid: the trace is a prefix of a valid trace
    - Violation: the trace is not a prefix of any valid trace
    - Unknown: otherwise
- *Suffix matching* corresponds to checking if the desired property holds for a suffix of a trace
  - Suffix matching returns:
    - Valid: the trace has a suffix which is prefix of a valid trace
    - Unknown: otherwise
- A suffix matching monitor can be implemented using total matching monitors for the same pattern by creating a new monitor instance at each event

# Context Free Patterns in JavaMOP

- JavaMOP supports LR(1) grammars
  - A well-known subset of context free grammars supported by tools like yacc
  - Correspond to deterministic context free languages
  - Can be parsed in linear time using the LR(1) parsing algorithm
- LR(1) parsing algorithm basically generates a deterministic push-down automaton (DPDA) that can recognize every word that is accepted by the input LR(1) grammar without any back-tracking
  - The transition system of the DPDA is encoded as action and go to tables which are constructed using the LR(1) parser construction algorithms
- The monitor synthesis algorithm basically uses the LR(1) parser construction algorithm and returns the resulting DPDA as the monitor

# Stack Cloning

- LR(1) parsing algorithm assumes that there is a single input trace
- However, in runtime monitoring the current trace is extended when a new event is observed
  - It would be inefficient if addition of each event started the parsing process from the beginning
- To handle this, before a reduction is made that uses the terminal symbol as the look-ahead, the parse stack is cloned (saving the parser state until that point)
  - When a new event is added to the end of the input, the parser can start back from the cloned state

# CFG Monitors

- When the synthesized CFP monitor is used for runtime monitoring, it guarantees the following:
  - For every finite prefix of a (possibly infinite) program trace, and a CFG pattern, JavaMOP will report
    - violation of the pattern if the LR(1) parsing algorithm would indicate a parse failure due to a bad token, and
    - validation of the pattern if the LR(1) parsing algorithm would return success given that prefix as the total input

- Suffix matching is implemented by identifying a subset of events that trigger the monitor creation
  - Events in the first set of the start symbol for the grammar are used for monitor creation

# Some Properties Checked in Experiments

- HashMap: An object’s hash code should not be changed when the object is a key in a HashMap;
- HasNext: For a given iterator, the hasNext() method should be called between all calls to next();
- SafeIterator: Do not update a Collection when using the Iterator interface to iterate its elements.
- ImprovedLeakingSync: Specifies correct synchronization behavior and allows calls to the unsynchronized methods so long as they happen within synchronized calls.
- SafeFileInputStream: It ensures that a FileInputStream is closed in the same method in which it is created.
- SafeFileWriter: It ensures that all writes to a FileWriter happen between creation and close of the FileWriter, and that the creation and close events are matched pairs.

# Results of Experiments

- Given 66 program/property pairs, the average runtime overhead of runtime monitoring with JavaMOP is 34%
- If the two cases with the largest overhead are removed, for the remaining 64 program/property patterns, the average runtime overhead is 8%
- Average memory overhead is 33% with a 4% median
- Overall JavaMOP has less overhead compared to other tools (PQL, Tracematches)