CIS 842: Specification and Verification of Reactive Systems

Lecture Specifications:

Sequencing Properties 2

Objectives

- To understand the goals and basic approach to specifying sequencing properties
- To understand the different classes of sequencing properties and the algorithmic techniques that can be used to check them

Outline

- What is a sequencing specification?
- What kinds of sequencing specifications are commonly used?
  - Safety vs. Liveness
- In depth on safety properties
  - How to specify them
  - Examples
  - How to check them

Stack Trace Checking

A safety property is just a violating prefix of a program trace

Naïve Algorithm:

- At every state consider the stack trace leading to that state
- If it is a string in the language of the violation property, stop and issue an error message

Stateful Search

- Our stateful search matches and records states based on the values of state variables
  - State values do not encode the path that reached the state only the *effect* of that path
  - This can lead to missing an erroneous trace

Example : Missed Error

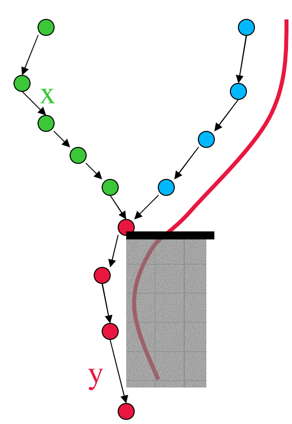
Consider the property

y is always preceded by x

Imagine a satisfying trace that is broken into two parts

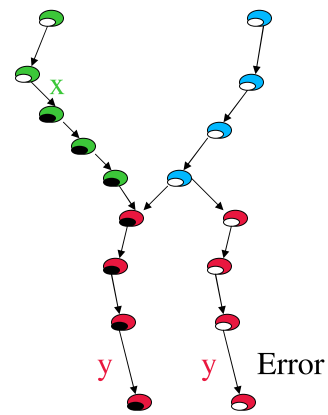
Imagine a violating trace leading to the dividing point in the original trace

Violation is only detected in suffix that has alread been searched!



Solution

- We need to distinguish states that are logically equivalent but different relative to the path that reaches them and the property being checked
- Note that the naïve algorithm works just fine for state-less search



Checking Safety Properties

- Think of it as a language problem
  - Program generates a language of strings over observables (each path generates a string) – L(P)
  - Property generates a (regular) language – L(S)
- Test the languages against each other
  - Language containment – $L(P) \subseteq L(S)$
  - Non-empty language intersection -- $L(P) \cap \overline{L(S)} \neq \emptyset$
  - Interchangeable due to complementation of finite-state automata

Checking Safety Properties

- Two basic approaches
  - Both require a deterministic finite-state automaton for the violation of the property
  - Easy to get via complementation and standard RE->DFA algorithms
- Instrument the program with property
- Check reachability in the product of the program and property

Instrumentation

- Assertions instrument the program
  - They are inserted at specific points
  - They perform tests of program state
  - They render an immediate verdict that is determined completely locally
- The same approach can be applied for safety properties
  - Instrumentation determines a partial verdict
  - Need a mechanism for communicating between different parts of the instrumentation

Example

```
boolean fork1, fork2;
thread Philosopher1() {
  loc pickup1: live {} when !fork1
    do { fork1 := true; } goto pickup2;
  loc pickup2: live {} when !fork2
    do { fork2 := true; } goto eating;
  loc eating: live {} do {} goto drop2;
  loc drop2: live {}
    do { fork2 := false; } goto drop1;
  loc drop1: live {}
    do { fork1 := false; } goto pickup1;
}
```

Consider the property:
a philosopher must pickup a fork before dropping it
e.g., [- P1.pickup1]*; P1:drop1; .*

Example

```
boolean fork1, fork2;
thread Philosopher1() {
  loc pickup1: live {} when !fork1
    do {
      // record that a pickup of 1 happened
      fork1 := true;
    } goto pickup2;
  loc pickup2: live {} when !fork2
    do { fork2 := true; } goto eating;
  loc eating: live {} do {} goto drop2;
  loc drop2: live {}
    do { fork2 := false; } goto drop1;
  loc drop1: live {}
    do {
      // check that a pickup of 1 happened
      fork1 := false;
    } goto pickup1;
}
```

Example

```
boolean fork1, fork2, sawpickup;
thread Philosopher1() {
  loc pickup1: live {} when !fork1
    do {
      sawpickup := true;
      fork1 := true;
    } goto pickup2;
  loc pickup2: live {} when !fork2
    do { fork2 := true; } goto eating;
  loc eating: live {} do {} goto drop2;
  loc drop2: live {}
    do { fork2 := false; } goto drop1;
  loc drop1: live {}
    do {
      assert(sawpickup);
      fork1 := false;
    } goto pickup1;
}
```


Instrumentation Approach

No change to the checking algorithm!

- Safety checking has been compiled to assertion checking

Adds state variables

- That record property related history to distinguish states
- These *multiply* the size of the state space

Recall : DFS Algorithm

```
seen := {s0}
pushStack(s0)
DFS(s0)

DFS(s)
workSet(s) := enabled(s)
while workSet(s) is not empty
    let α ∈ workSet(s)
    workSet(s) := workSet(s) \ {α}
    s' := α(s)
    if s' ∉ seen then
        seen := seen ∪ {s'}
        pushStack(s')
        DFS(s')
        popStack()
end DFS
```

α : `assert(s) = false`; stop

α : otherwise; proceed

Instrumentation Approach

- Instrumenting programs is
  - Laborious – must identify all points that are related to the property (may be conditional on data)
  - Error prone – lack of instrumentation at a state change (false error), lack of instrumentation at a state check (missed error)
  - Property specific – must be done for each property
- Automate it via product construction

Example

```
boolean fork1, fork2;
thread Philosopher1() {
  loc pickup1: live {} when !fork1
    do { fork1 := true; } goto pickup2;
  loc pickup2: live {} when !fork2
    do { fork2 := true; } goto eating;
  loc eating: live {} do {} goto drop2;
  loc drop2: live {}
    do { fork2 := false; } goto drop1;
  loc drop1: live {}
    do { fork1 := false; } goto pickup1;
}
```

Consider the property:
a philosopher must pickup a fork before dropping it
e.g., [- P1.pickup1]*; P1:drop1; .*

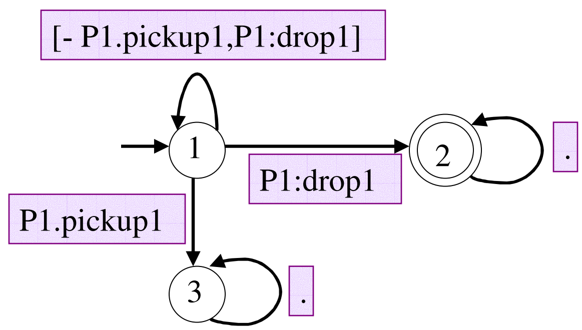
Finite-state Automaton

A *finite state automaton* (FSA) is a 5-tuple (Q, Σ, δ, S, F) where

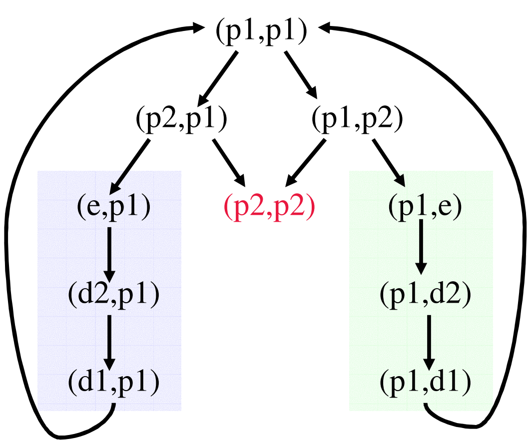
- Q is a finite set of states
- Σ is a finite set of symbols, the alphabet
- $S \subseteq Q$ the set of start states
- $F \subseteq Q$ the set of final states
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function

Example : Property

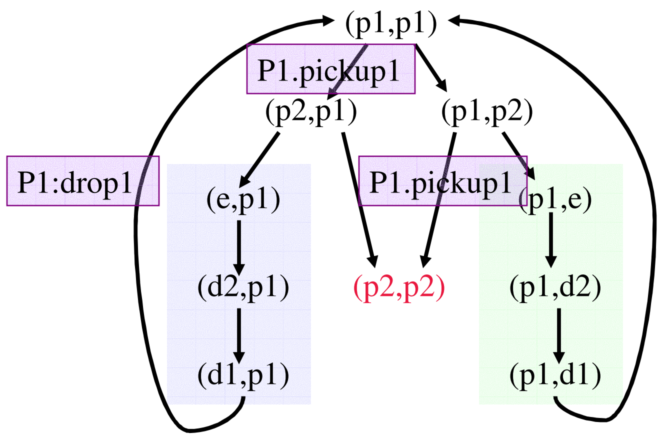
Consider the property:
a philosopher must pickup a fork before dropping it
e.g., [- P1.pickup1]*; P1:drop1; .*



Example : System



Example : System Automaton



Product

Synchronous Product of Two Automata

$$(Q, \Sigma, \delta, S, F) \times (Q', \Sigma', \delta', S', F')$$

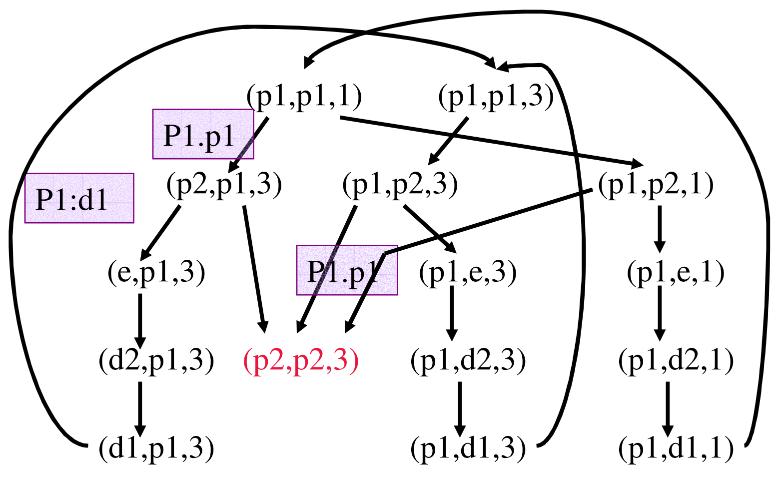
Is a new Automaton

$$(< Q, Q' >, \Sigma \cup \Sigma', \Delta, < S, S' >, < F, F' >)$$

where

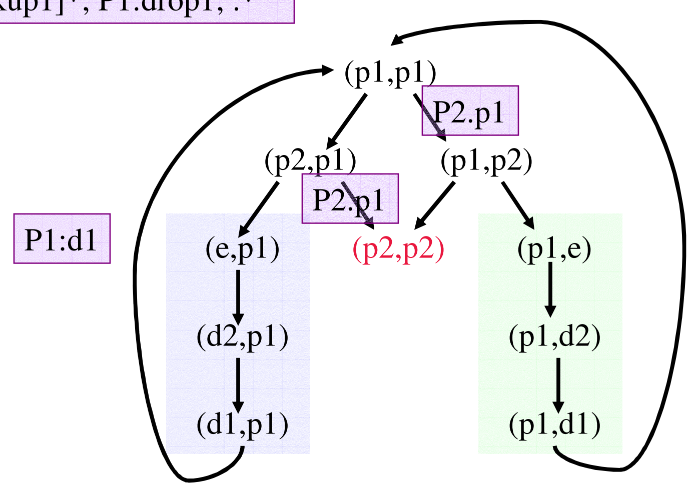
$$\Delta(< s, s' >, a) = \begin{cases} < \delta(s,a), s' > & \textit{if } a \in \Sigma \wedge a \notin \Sigma' \\ < s, \delta'(s', a) > & \textit{if } a \notin \Sigma \wedge a \in \Sigma' \\ < \delta(s,a), \delta'(s',a) > & \textit{if } a \in \Sigma \wedge a \in \Sigma' \end{cases}$$

Example : Product Automaton

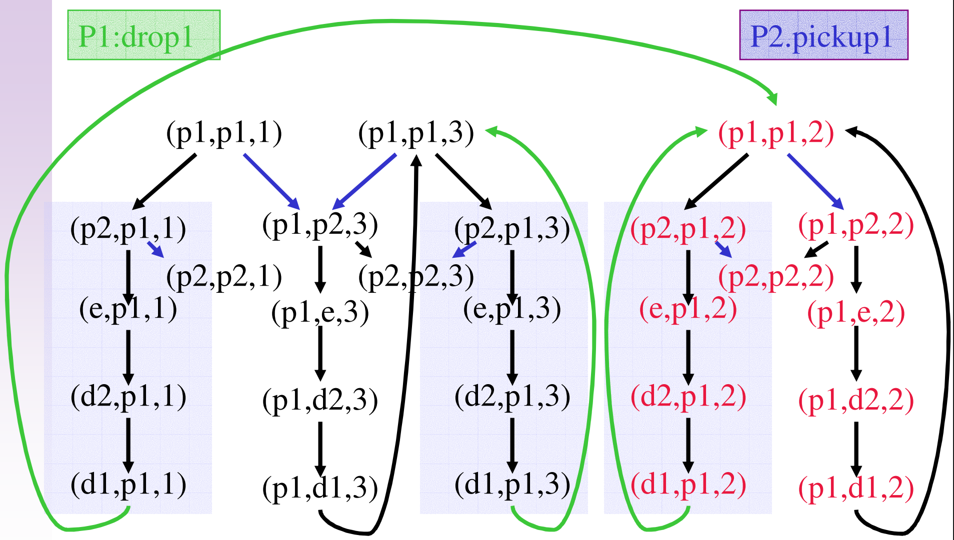


Example : Wrong Property

[- P2.pickup1]*; P1:drop1; .*



Example : Product Automaton



For You To Do

```
seen := {s0}
pushStack(s0)
DFS(s0)

DFS(s)
workSet(s) := enabled(s)
while workSet(s) is not empty
    let α ∈ workSet(s)
    workSet(s) := workSet(s) \ {α}
    s' := α(s)
    if s' ∉ seen then
        seen := seen ∪ {s'}
        pushStack(s')
        DFS(s')
        popStack()
end DFS
```

How would you modify this algorithm to construct the product on the fly?

Some Observations

- Alphabet of property is always a subset of alphabet of system

$$\Sigma_{property} \subseteq \Sigma_{system}$$

- This means we can drive transitions in the property automaton from transitions in the system automaton

DFS Automaton Checking

```
seen := {s0}
pushStack(s0)
DFS(s0)

DFS(s)
workSet(s) := enabled(s)
while workSet(s) is not empty
    let α ∈ workSet(s)
    workSet(s) := workSet(s) \ {α}
    s' := α(s)
    if s' ∉ seen then
        seen := seen ∪ {s'}
        pushStack(s')
        DFS(s')
        popStack()
end DFS
```

See if this transition is a symbol in the property alphabet

If so them drive a transition in the property automaton

DFS Automaton Checking

New property state component

Conditional update of property state

Test for acceptance state of violation property

```
seen := {(s0, p0)}
pushStack((s0, p0))
DFS((s0, p0))

DFS((s, p))
workSet(s) := enabled(s)
while workSet(s) is not empty
    let α ∈ workSet(s)
    workSet(s) := workSet(s) \ {α}
    s' := α(s)
    p' := (α ∈ Σp) ? δ(p, α) : p
    assert(p' ∉ Fp)
    if (s', p') ∉ seen then
        seen := seen ∪ {(s', p')}
        pushStack((s', p'))
        DFS((s', p'))
        popStack()
end DFS
```


Bogor Safety Checking

- Bogor-lite has been extended to check properties specified as *deterministic* FSAs
  - i.e., single transition on any given symbol
- You specify a function that encodes the FSA for the violation
  - Option: `fsaFunctionId=<function-id>`
  - where locations with name "`bad$...`" indicate errors

Example : System

```
thread MAIN() {
  loc open: live {}
    do { } // open
    goto run;
  loc run: live {}
    do { } // run, call close
    goto close;
  loc close: live {}
    do { } // close
    goto open;
}
```


Example : Property

```
extension Property for edu.ksu.cis.projects.bogor.ext.lite.property.Property
   { expdef boolean transformation(string, string); }

   function FSASpec() {
    loc init: live {}
         when Property.transformation("MAIN", "open") do { } goto opened;
         when Property.transformation("MAIN", "close") do { } goto bad$State;
    loc opened: live {}
         when Property.transformation("MAIN", "open") do { } goto bad$State;
         when Property.transformation("MAIN", "close") do { } goto init;
    loc bad$State: live {} // bad state
         do { } goto bad$State;
   }
```

For You To Do

- Take your instrumentation examples from last lecture and encode those properties as violation automata
- Check them with the Bogor-lite support for on-the-fly product construction