

CIS 842: Specification and Verification of Reactive Systems

Lecture Specifications: Sequencing Properties 2

Copyright 2001-2004, Matt Dwyer, John Hatcliff, and Robby. The syllabus and all lectures for this course are copyrighted materials and may not be used in other course settings outside of Kansas State University in their current form or modified form without the express written permission of one of the copyright holders. During this course, students are prohibited from selling notes to or being paid for taking notes by any person or commercial firm without the express written permission of one of the copyright holders.

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

CIS 842: Spec Basics and Observables

5

Example : Missed Error

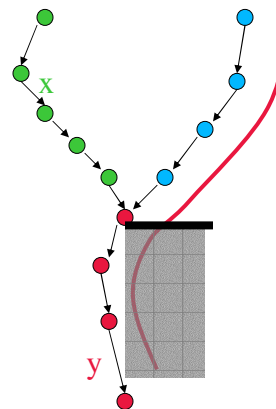
Consider the property

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 already been searched!

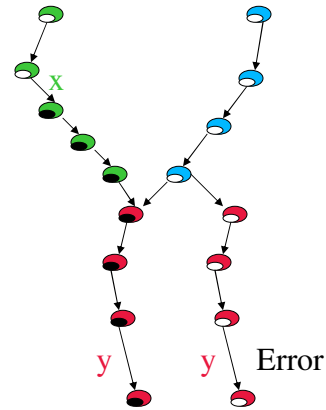


CIS 842: Spec Basics and Observables

6

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



CIS 842: Spec Basics and Observables

7

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

CIS 842: Spec Basics and Observables

8

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- \rightarrow DFA algorithms
- **Instrument** the program with property
- Check reachability in the **product** of the program and property

CIS 842: Spec Basics and Observables

9

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

CIS 842: Spec Basics and Observables

10

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., $[- P1.pickup1]^*; P1.drop1; .^*$

CIS 842: Spec Basics and Observables

11

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;
}
```

CIS 842: Spec Basics and Observables

12

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;
}
```

CIS 842: Spec Basics and Observables

13

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

CIS 842: Spec Basics and Observables

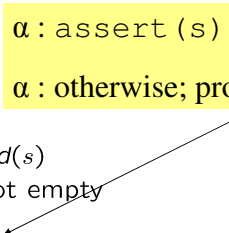
14

Recall : DFS Algorithm

```
1 seen := {s0}
2 pushStack(s0)
3 DFS(s0)

DFS(s)
4 workSet(s) := enabled(s)
5 while workSet(s) is not empty
6   let  $\alpha \in \textit{workSet}(s)$ 
7   workSet(s) := workSet(s) \ { $\alpha$ }
8   s' :=  $\alpha(s)$ 
9   if s'  $\notin$  seen then
10     seen := seen  $\cup$  {s'}
11     pushStack(s')
12     DFS(s')
13   popStack()
end DFS
```

$\alpha : \text{assert}(s) = \text{false}; \text{stop}$
 $\alpha : \text{otherwise}; \text{proceed}$



CIS 842: Spec Basics and Observables

15

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

CIS 842: Spec Basics and Observables

16

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., $[- P1.pickup1]^*; P1.drop1; .^*$

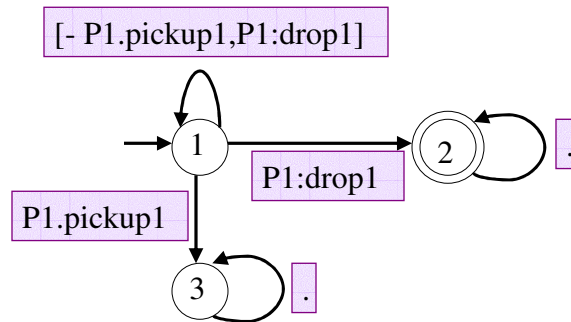
Finite-state Automaton

A *finite state automaton* (FSA) is a 5-tuple $(Q, \Sigma, \delta, S, F)$ where

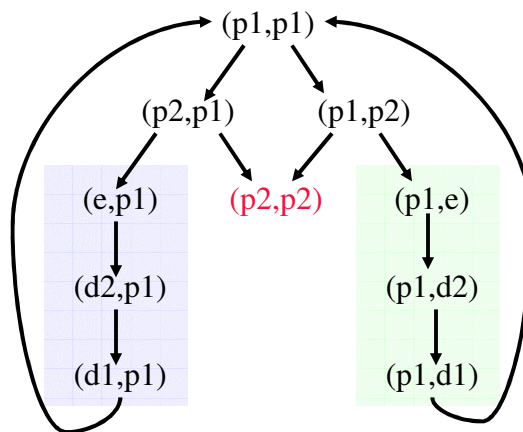
- Q is a finite set of states
- Σ is a finite set of symbols, the alphabet
- $S \subseteq Q$ the set of start states
- $F \subseteq Q$ the set of final states
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function

Example : Property

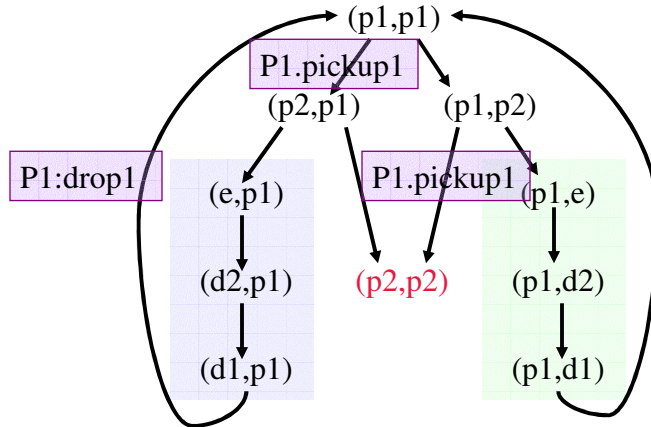
Consider the property:
a philosopher must pickup a fork before dropping it
e.g., $[- P1.pickup1]^*; P1.drop1; .^*$



Example : System



Example : System Automaton



CIS 842: Spec Basics and Observables

21

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

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

CIS 842: Spec Basics and Observables

22

Example : Product Automaton

The diagram illustrates a product automaton with two processes, P1 and P2. The states are represented as pairs $(p1, p2)$, where $p1$ is the state of P1 and $p2$ is the state of P2. The transitions are labeled with actions: $drop$ for P1 and $pickup$ for P2.

The diagram is divided into two main sections by a vertical line. The left section represents the state space where P1 is in a $drop$ state (green box) and P2 is in a $pickup$ state (blue box). The right section represents the state space where P1 is in a $pickup$ state (blue box) and P2 is in a $drop$ state (green box).

States are arranged in a grid-like structure. Transitions are shown as arrows. Some transitions are labeled with $drop$ or $pickup$ and are colored to match the box they lead to. A large green arrow at the top points from the right section to the left section, and a large black arrow at the bottom points from the left section to the right section, indicating a global transition or a reset.

Left Section (P1:drop1, P2:pickup1):

- States: $(p1, p1, 1)$, $(p1, p2, 3)$, $(p2, p1, 1)$, $(p2, p2, 1)$, $(p1, e, 3)$, $(p2, p2, 3)$, $(e, p1, 1)$, $(e, p1, 3)$, $(d2, p1, 1)$, $(d2, p1, 3)$, $(d1, p1, 1)$, $(d1, p1, 3)$.
- Transitions: $(p1, p1, 1) \rightarrow (p2, p1, 1)$ (black), $(p1, p1, 1) \rightarrow (p1, p2, 3)$ (blue), $(p1, p2, 3) \rightarrow (p2, p2, 1)$ (black), $(p1, p2, 3) \rightarrow (p2, p2, 3)$ (black), $(p1, p2, 3) \rightarrow (p1, e, 3)$ (black), $(p2, p1, 1) \rightarrow (p2, p2, 1)$ (blue), $(p2, p1, 1) \rightarrow (e, p1, 1)$ (black), $(p2, p2, 1) \rightarrow (e, p1, 1)$ (black), $(p2, p2, 1) \rightarrow (d2, p1, 1)$ (black), $(p2, p2, 3) \rightarrow (e, p1, 3)$ (black), $(p2, p2, 3) \rightarrow (d2, p1, 3)$ (black), $(e, p1, 1) \rightarrow (d2, p1, 1)$ (black), $(e, p1, 1) \rightarrow (d1, p1, 1)$ (black), $(e, p1, 3) \rightarrow (d2, p1, 3)$ (black), $(e, p1, 3) \rightarrow (d1, p1, 3)$ (black).

Right Section (P1:pickup1, P2:drop1):

- States: $(p1, p1, 2)$, $(p1, p2, 2)$, $(p2, p1, 2)$, $(p2, p2, 2)$, $(p1, e, 2)$, $(p2, p2, 2)$, $(e, p1, 2)$, $(e, p1, 2)$, $(d2, p1, 2)$, $(d2, p1, 2)$, $(d1, p1, 2)$, $(d1, p1, 2)$.
- Transitions: $(p1, p1, 2) \rightarrow (p2, p1, 2)$ (black), $(p1, p1, 2) \rightarrow (p1, p2, 2)$ (blue), $(p1, p2, 2) \rightarrow (p2, p2, 2)$ (black), $(p1, p2, 2) \rightarrow (p1, e, 2)$ (black), $(p2, p1, 2) \rightarrow (p2, p2, 2)$ (blue), $(p2, p1, 2) \rightarrow (e, p1, 2)$ (black), $(p2, p2, 2) \rightarrow (e, p1, 2)$ (black), $(p2, p2, 2) \rightarrow (d2, p1, 2)$ (black), $(e, p1, 2) \rightarrow (d2, p1, 2)$ (black), $(e, p1, 2) \rightarrow (d1, p1, 2)$ (black).

For You To Do

1 $seen := \{s_0\}$
2 $pushStack(s_0)$
3 $DFS(s_0)$

$DFS(s)$
4 $workSet(s) := enabled(s)$
5 while $workSet(s)$ is not empty
6 let $\alpha \in workSet(s)$
7 $workSet(s) := workSet(s) \setminus \{\alpha\}$
8 $s' := \alpha(s)$
9 if $s' \notin seen$ then
10 $seen := seen \cup \{s'\}$
11 $pushStack(s')$
12 $DFS(s')$
13 $popStack()$
end DFS

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

CIS 842: Spec Basics and Observables

26

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

```
1 seen := {s0}
2 pushStack(s0)
3 DFS(s0)

DFS(s)
4 workSet(s) := enabled(s)
5 while workSet(s) is not empty
6   let  $\alpha \in \text{workSet}(s)$ 
7   workSet(s) := workSet(s) \ { $\alpha$ }
8   s' :=  $\alpha(s)$ 
9   if s' ∉ seen then
10     seen := seen ∪ {s'}
11     pushStack(s')
12     DFS(s')
13   popStack()
end DFS
```

See if this transition is a symbol in the property alphabet

If so then 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

```

1  seen := {(s0, p0)}
2  pushStack((s0, p0))
3  DFS((s0, p0))

DFS((s, p))
4  workSet(s) := enabled(s)
5  while workSet(s) is not empty
6    let α ∈ workSet(s)
7    workSet(s) := workSet(s) \ {α}
8    s' := α(s)
8.1  p' := (α ∈ Σp) ? δ(p, α) : p
8.2  assert(p' ∉ Fp)
9    if (s', p') ∉ seen then
10     seen := seen ∪ {(s', p')}
11     pushStack((s', p'))
12     DFS((s', p'))
13     popStack()
end DFS
    
```

CIS 842: Spec Basics and Observables

29

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

CIS 842: Spec Basics and Observables

30

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;  
}
```

CIS 842: Spec Basics and Observables

31

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;  
}
```

CIS 842: Spec Basics and Observables

32

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