CIS 842: Specification and Verification of Reactive Systems

Lecture Specifications: Sequencing Properties

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

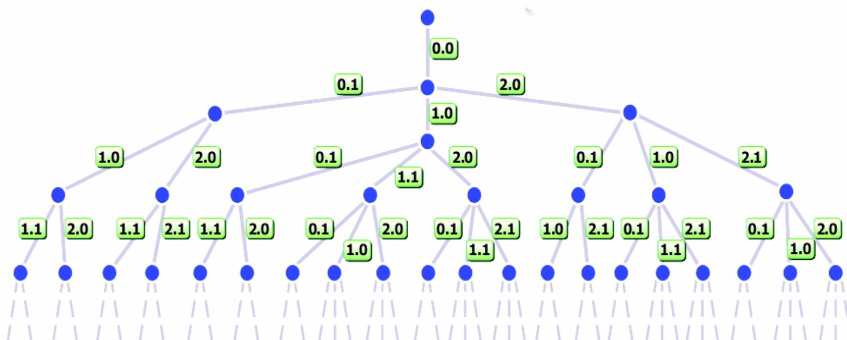
- What is a sequencing specification?
- What kinds of sequencing specifications are commonly used?
  - Safety properties
  - Liveness properties
- In depth on safety properties
  - How to specify them
  - Examples
  - How to check them

What is a Sequencing Specification?

- We've seen specifications that are about individual program states
  - e.g., assertions, invariants
- Sometimes we want to reason about the relationship between multiple states
  - Must one state always precede another?
  - Does seeing one state preclude the possibility of subsequently seeing another?
- We need to shift our thinking from states to paths in the state space

Paths (Traces)

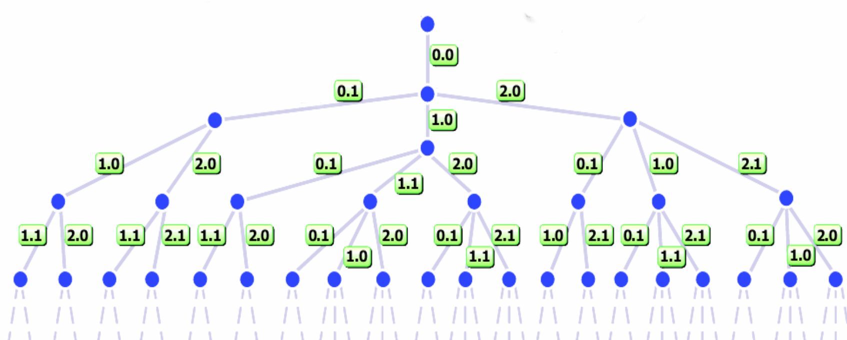
Recall that the system's executions can be viewed as a tree. We want to determine the set of paths in that tree that match a given pattern.



5

Paths (Traces)

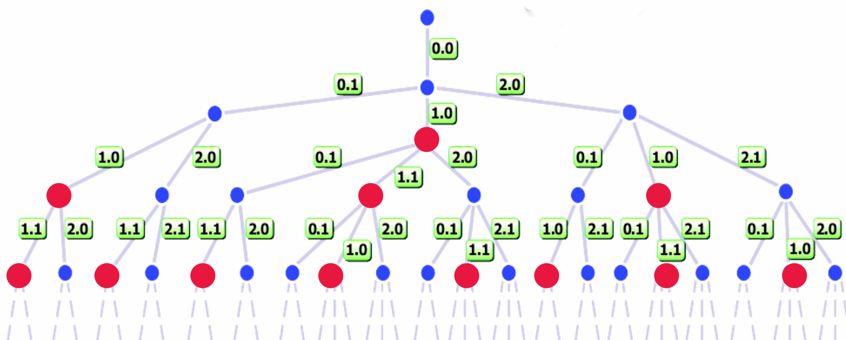
Consider the pattern:
a 1.* is followed by a *.1



6

Paths (Traces)

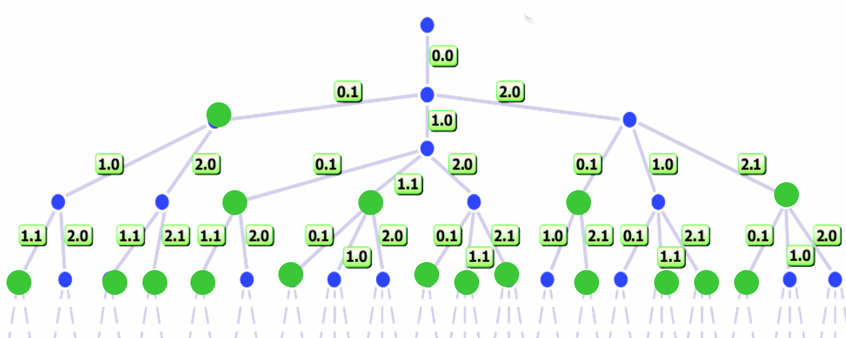
Here are all the states that immediately follow a 1.*



7

Paths (Traces)

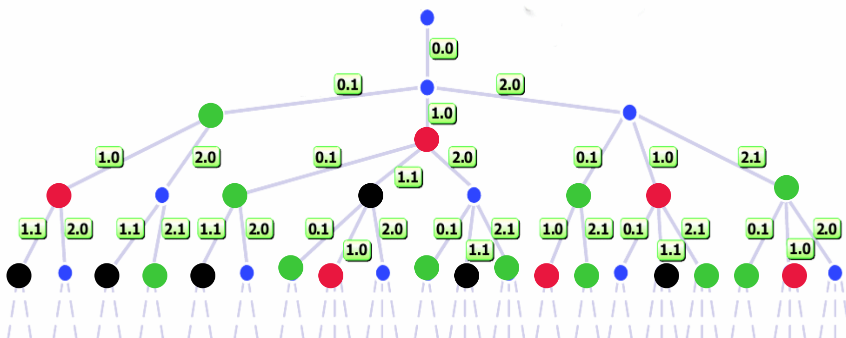
Here are all the states that immediately follow a *.1



8

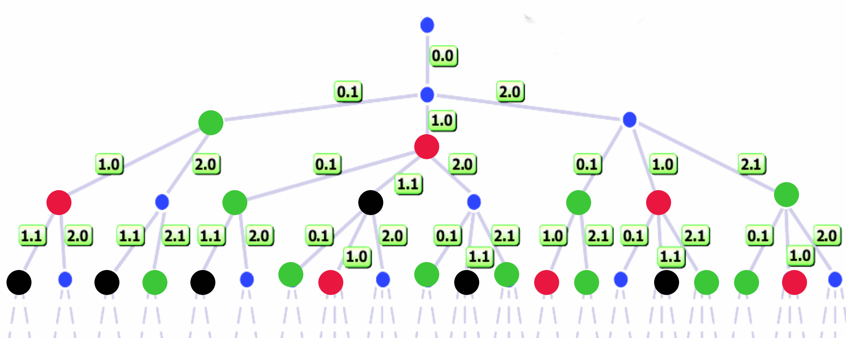
Paths (Traces)

Here are both 1.* and *.1 successor states (with black indicating both)



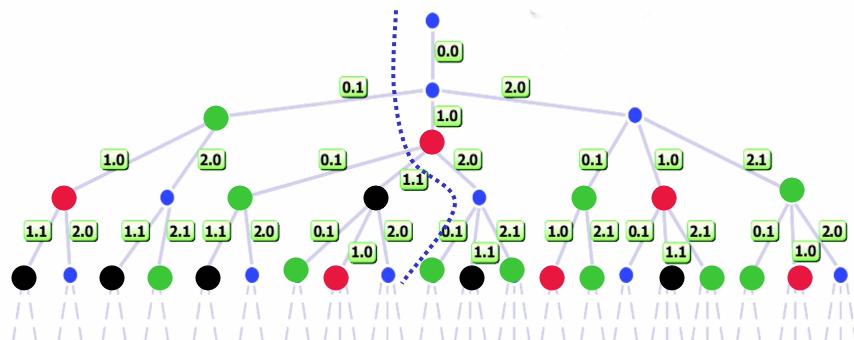
Paths (Traces)

Do all paths conform to the pattern:
a 1.* is followed by a *.1



Paths (Traces)

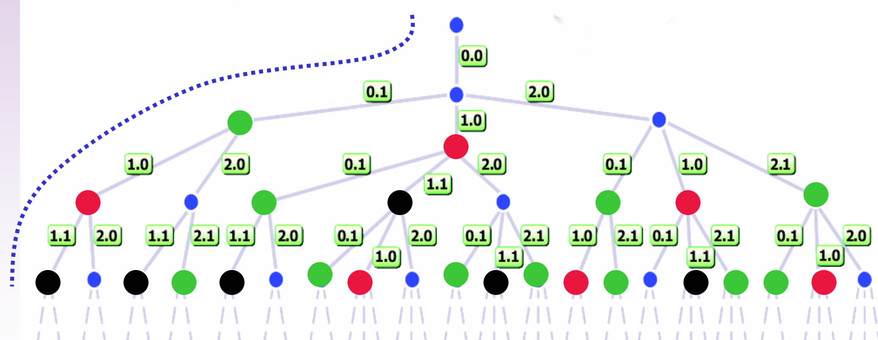
Some paths obviously match the pattern



11

Paths (Traces)

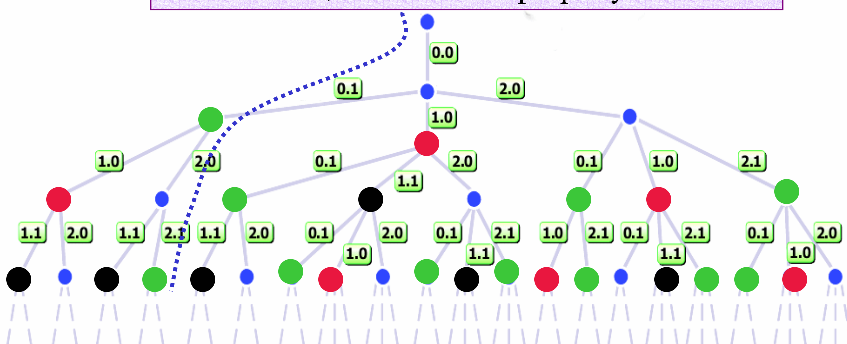
For others it is more interesting
can the 1.* follow a *.1?
what if the 1.* and *.1 coincide?



12

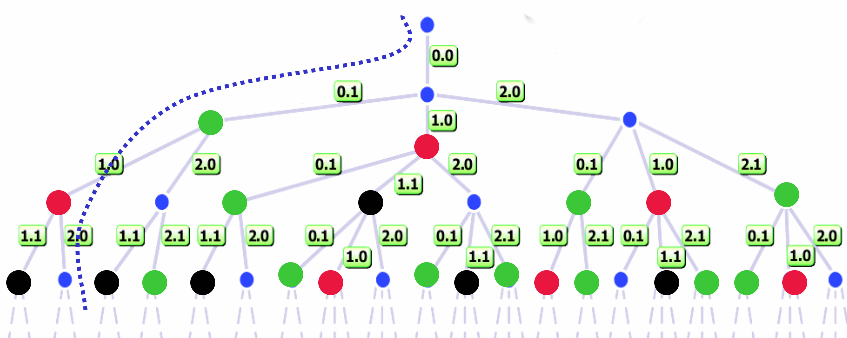
Paths (Traces)

For still others we can't tell yet
will we ever see a 1.*?
if not, then does the property hold?



Paths (Traces)

For still others we can't tell yet
will we ever see a subsequent *.1?



We need …

- A language for describing sequencing patterns
  - There are many such languages with different strengths and weaknesses
- An algorithm for exhaustively considering whether all paths match the pattern
  - Currently we've only seen the exhaustive consideration of individual states

A classic distinction …

- Safety properties
  - "nothing bad ever happens"
  - are violated by a *finite* path prefix that ends in a bad thing
  - are fundamentally about the *history* of a computation up to a point
- Liveness properties
  - "something good eventually happens"
  - are violated by *infinite* path suffixes on which the good thing never happens
  - are fundamentally about the *future* of a computation from a point onward

Examples

- A use of a variable must be preceded by a definition
- When a file is opened it must subsequently be closed
- You cannot shift from drive to reverse without passing through neutral
- No pair of adjacent dining philosophers can be eating at the same time
- The program will eventually terminate
- The program is free of deadlock

Examples

- A use of a variable must be preceded by a definition -- Safety
- When a file is opened it must subsequently be closed -- Liveness
- You cannot shift from drive to reverse without passing through neutral -- Safety
- No pair of adjacent dining philosophers can be eating at the same time -- Safety
- The program will eventually terminate -- Liveness
- The program is free of deadlock -- Safety

For You To Do

- Think of three more properties
  - Classify them as safety or liveness
  - How many observations are being made in the properties
- Try to think of at least one positive property
  - i.e., saying what the system can do
- … and one negative property
  - i.e., saying what the system cannot do
- Is an invariant a safety or liveness property?

Expressing Safety Properties

- Let's simplify things to start with …
- We can observe the location of a BIR-lite thread, e.g.,

```
thread MAIN()  {
   loc open: live {}
        do { … } goto run;
   loc run: live {}
```

- Name observables as a pair
  - e.g, **MAIN:open**, **MAIN:run**
- Such an observable is true when the named thread enters the named location

Regular Expressions

- Regular expressions can be used to specify safety properties
  - Symbols are observables – `MAIN:open`
- Basic Operators
  - Concatenation – `e ; e`
  - Disjunction – `e | e`
  - Closure – `e*`
  - Grouping – `(e)`

Regular Expressions

- Some Useful Derived Operators
  - Option – `e?`
  - Positive closure – `e+`
  - Finite closure – `e^k`
  - Any symbol – `.`
  - Symbol sets – `[e, f, …]`
  - Symbol exclusion – `[- e, f, …]`

Example

```
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;
}
```


A property

- *Opens and closes happen in matching pairs*
- Positive specification

  `(MAIN:open; MAIN:close)*`
- Negative specification (i.e., violation)

  ```
  MAIN:close; .* |
  .*; Main:open; Main:open; .* |
  .*; Main:close; Main:close; .*
  ```

Example

```
system TwoDiningPhilosophers {
   boolean fork1;
   boolean 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;
   }
   thread Philosopher2() {…}
}
```


A property

- *Whenever philosopher 1 is eating, philosopher 2 cannot eat, until philosopher 1 drops his first fork*
- Positive specification

```
[- P1:eating]*;
(P1:eating; [- P2:eating]*; P1:drop1)*
```

- Negative specification (i.e., violation)

```
.*; P1:eating; [- P1:drop1]; P2.eating; .*
```

For You To Do

- Make up an alphabet and specify the following properties as regular expressions
  - *A use of a variable must be preceded by a definition*
  - *You cannot shift from drive to reverse without passing through neutral*
- Give positive and negative formulations

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;
}
```

Does this capture the correctness property?

Instrumentation Approach

- Works well when you only want to check conditions at specific points
- What if you want to exclude some action from a region of program execution?

```
[- P1:eating]*;
(P1:eating; [- P2:eating]*; P1:drop1)*
```

- Need to use invariants

Example

```
boolean fork1, fork2, p1eating;
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 {
      p1eating := true;
    } goto drop2;
  loc drop2: live {
    do { fork2 := f
  loc drop1: live {
    do {
      fork1 := false;
      p1eating := false;
    } goto pickup1;
}
```

Same instrumentation for Philosopher2

Check invariant:

p1eating -> !p2eating

Instrumentation Approach

- No change to the checking algorithm!
  - Safety checking has been compiled to assertion checking
  - Additional property state variables increase cost
- Instrumenting programs is
  - Laborious – must identify all points that are related to the property (may
  - Error prone – lack   change (false error), lack of instrumentation at a state check (missed error)
  - Property specific – must be done for each property

Automate it!!

For You To Do

- Pick your favorite BIR-lite program
- Develop two safety properties for it
- Instrument the program with those properties
- Check them with Bogor

Product Reachability

- Next time