CIS 842: Specification and Verification of Reactive Systems

Lecture Specifications: Basics and Observables

Objectives

- To understand the goals and basic elements of every formal specification formalism
- To understand the variety in ways that aspects of program behavior can be observed

Outline

- What is a specification?
- Why write specifications?
- What are the building blocks of specs?
  - What kinds of variation is there across specification languages?
- How can we define the execution behavior of a system that is relevant from the point of view of a specification?
  - What do we want to observe?

What is a Specification?

- A detailed, exact statement of particulars, especially a statement prescribing materials, dimensions, and quality of work for something to be built, installed, or manufactured.

  American Heritage Dictionary 2000
- needs to be precise
- describes what is to be done

What is a Formal Specification?

- A *formal specification* is the expression, in some formal language and at some level of abstraction, of a collection of properties that some system should satisfy

  Axel van Lamsweerde, Future of Software Engineering, 2000

- formal language => precise
- properties … system should satisfy
  - is a prescription!

Why Write Specifications?

- To drive the implementation of a system
  - Rare - usually driven from informal requirements
  - Would require a complete specification - expensive
- To provide a redundant description of intent so we can check an implementation against something
  - Generate tests
  - Perform rigorous inspections
  - Model check

A Spec for Spec Languages

- Concise : if the spec is as large and complex as the system, you lose
- Understandable : spec needs to be right, so you better be able to read it
- General : want to be able to describe a wide range of system characteristics (here we stick to *functionality*)
- Think Different : forces you to express properties differently than solutions

What's a Good Spec?

- Consistent : no internal contradictions
- Unambiguous : has a clear meaning
- Complete : captures all of the essential aspects of the problem that are described elsewhere
- Minimal : doesn't state irrelevant or implementation-specific properties

Essential Parts of a Specification

Components of the system that are related to the property (related to abstraction)

x

Constraints define what is demanded, desired, or restricted of the components

x>0

Order describes how, if at all, the constrained-components related to one another

if x>0 then after x++, x>0

An Example

Think about making a phone call

What components of the phone are relevant?

What characteristics of those components do we care about?

How does the order in which components attain those characteristics influence the making of a phone call?

Variations in Specification Style

- State-based : a condition or mode of being
  - phone is off the hook
  - call is connected
- Event-based : something that happens at a given place and time
  - phone is lifted
  - number 3 is dialed

For You To Do

- Consider the property:

  Dial 532-6350 to connect to CIS
- Give a state-based specification
- Give an event-based specification
- Don't forget to mention any implicit parts or constraints that are relevant

States and Events

- Changes in state are caused by events

$x==5 \xrightarrow{x++} x==6$

- Not all events cause a change in state

$x==5 \xrightarrow{x=x+0} x==5$

Mixed States and Events

- When the door is open and the key is not in the ignition, the alarm beeps.

  door==open

  ignitionKey!=in

  beep

- Assigning x to 7 makes x greater than 0.

  x=7

  x>0

Variations in Specification Style

- Allowable behavior : define what a correctly functioning system is able to do

  offhook, number[7], connected

- Violations : define what a correctly functioning system can never do

  onhook, … anything but offhook …, connected

Observing System Behavior

- A specification may observe
  - the value of a component
  - an operation applied to a component
- The language of value comparisons and operations is
  - System description-language dependent, e.g., Java, BIRLite

BIR-lite Observables

- Currently BIR-lite supports only 2 state observables
- Global variables
  - any legal boolean-valued BIR expression may be used to define the constraints on a variable that are observed
- Existential thread program counter

  `Property.existsThread(<thread>,<label>)`
  - there exists an instance of the named thread that is about to execute the guarded comment with the given label
  - Need this (in part) because there is no explicit PC variable for a thread

Example : BIR-lite

```
system TwoDiningPhilosophers {
  extension Property for edu.ksu.cis.projects.bogor.ext.Property {
    expdef boolean existsThread(string, string);
  }
  boolean fork1;
  boolean fork2;
  thread Philosopher1() {
    loc loc0: live {} when !fork1 do { fork1 := true; } goto loc1;
    loc loc1: live {} when !fork2 do { fork2 := true; } goto loc2;
    loc loc2: live {} do { fork2 := false; }  goto loc3;
    loc loc3: live {} do { fork1 := false; }  goto loc0;
  }
  thread Philosopher2() {…}

  main thread Main() {
    loc loc0: do {
      assert(!(Property.existsThread("Philosopher1", "loc1") && !fork1))); }
      return;
  }
}
```

Example : BIR-lite

```
system TwoDiningPhilosophers {
  extension Property for edu.ksu.cis.projects.bogor.ext.Property {
    expdef boolean existsThread(string, string);
  }
  boolean fork1;
  boolean fork2;
  thread Philosopher1() {
    loc loc0: live {} when !fork1 do { fork1 := true; } goto loc1;
    loc loc1: live {} when !fork2 do { fork2 := true; } goto loc2;
    loc loc2: live {} do { fork2 := false; }  goto loc3;
    loc loc3: live {} do { fork1 := false; }  goto loc0;
  }
  thread Philosopher2() {…}

  main thread Main() {
    loc loc0: do {
      assert(!(Property.existsThread("Philosopher1", "loc1") && !fork1))); }
      return;
  }
}
```


Specification Extensions

Extensions are useful for implementing systems and for defining specifications

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

Build Your Own

- Bogor allows one to define new observables ala extensions

```
Property.allThread(<thread>,<label>)
Property.noThread(<thread>,<label>)
Property.samePlace(<thread>)
```

- Not part of BIR-lite, but this gives you access to the entire state representation
  - Can build powerful observables
  - You'll learn how to do this later

BIR Functions

- BIR provides constructs for defining recursive functions over BIR variables

```
fun p() returns boolean =
   Property.existsThread("Philosopher1",
                         "loc1") &&
   !fork1;
```

- This allows us to use named observables which aids readability

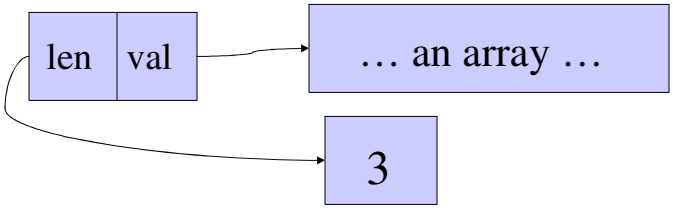
```
assert(!p())
```

Java Observables

- Value of a local variable
  - Scalars
  - References
- Value of object fields
  - Explicit
  - Implicit : locks
- Method call
  - Qualified by parameter values

Naming Instances

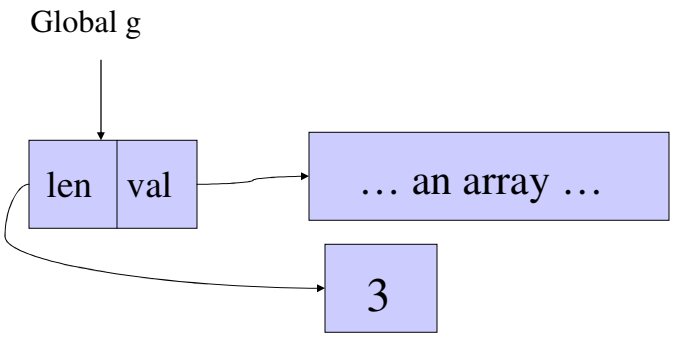
- existsThread(…) is used because threads are anonymous
  - i.e., there is no variable that holds their value
- This is a general problem for heap allocated data



Naming Instances

- It's easy to talk about global references

  g.len>0 → g.val!=null



Naming Instances

- How else can we talk about the instances of this class?
  - All/some of the allocated instances
  - All/some of the instances reachable from g
  - All/some of the instances reachable by a reference chain of the form g.f.h
  - …
- Need a way of defining these for easy use in specifications
  - Use BIR extensions

Naming Yields Independence

- Several of these notions of naming heap instances are quite abstract
  - *The set of instances of C that are reachable from g*
- These make for good specifications since they are independent of implementation details
  - g points to an array
  - g points to the head of a linked list
  - g points to the root of a tree

Observing The Java Heap

- Shapes
  - Tree, dag
- Values
  - Ordered leaves of a tree
- Approximations
  - Reach
  - containment

Assertions

- Assertions are a composite specification
  - Implicit location constraint
  - Explicit data constraint

```
loc loc7: do {
      assert(x<5 && y>3); }   // && PC == loc7
      goto loc14;
  }
}
```


Invariants

- Are constraints that should hold in all system states
  - explicit data constraint
  - location constraints via thread location query extensions
- You've seen the assertion-in-a-thread trick
  - Persistently evaluate the assertion thereby enforcing it in every state

Invariants as Bogor Options

- It is possible to express invariant checks directly in Bogor
- If you have a boolean function expression that takes no parameters, e.g.,

```
fun p() returns boolean =
    Property.existsThread("Philosopher1",
                          "loc1") &&
    !fork1;
```

- Define an option
  - …Isearcher.invariants=p

Using Invariants

- An invariant is a very strong specification
  - It says that the constraint is true in every system state
  - e.g., the initial state, the final state, …
- Often times we only want a constraint to be enforced during some *region* of the system's execution
- We can specify this using an implication as an invariant

  regionSpec → constraintSpec

Examples

- Coherent variables
  - e.g., head and tail pointers to a buffer
  - e.g., length and array contents
- Invariants that describe relationships
  - e.g., tail < head
  - e.g., length = # of non-null elements
- Want to disable their enforcement when the values are being updated
  - e.g., (!add && !take) → tail < head

Summary

- Specifications are an essential element of rigorous system analysis
- Observables are boolean expressions that define constraints on component values that are relevant to a specification
- Specifications, like assertions and invariants, are built up from observables
- BIR-lite has primitive support for defining observables and writing specifications
- Bogor's extension facility allows one to significantly enrich the spec language