Modelling a Dynamic System

Phone Book Example

Modelling dynamic systems in Alloy

- Alloy has no built-in notion of state.
- Variables are mathematical variables, not programming variables. There is no way to assign a new value to a variable.
- Solution: a change of state is modelled by using two variables, representing the “before” state and the “after” state.
- We shall now look at an example of this: a specification of a phone book.

Phone Book Example

This example will help with the assignment. We model a simple phone book with these properties:

- The phone book is to be used to keep track of people's names and telephone number(s).
- A name may be linked to 0 or more phone numbers. Multiple names may be linked to the same number.
- Operations to be specified:
  - Look up the number(s) linked to a given name
  - Add a new name (initially without any number)
  - Add a new number to a known name
  - Delete a name+number combination
  - Delete a name and all associated name+number pairs
  - Delete a number and all associated name+number pairs

Specifying the state of the system

Initial questions:

- What information is needed to describe the **state** of the phone book?
- How can we represent this information in terms of sets and relations?
- Constraints? Are there any properties that the system must always satisfy (*invariants*)?

Specifying the state of the system

Initial questions:

- What information is needed to describe the **state** of the phone book?
    - The names that are known
    - The mapping from names to phone numbers
- How can we represent this information in terms of sets and relations?
    - A set of known names
    - A relation, mapping known names to phone numbers
- Constraints? Are there any properties that the system must always satisfy (*invariants*)?
    - There are no obvious invariants

Specifying the state of the system

```
module PhoneBook

sig Name { }         // set of names
sig Phone { }        // set of phone numbers

sig PhoneBook {                    // a phone book
    known: set Name,               // names known
    number: known -> Phone  // maps names to numbers
}
```

Specifying a static operation

If an operation cannot change the state of the system (it is *static*), we can specify it using a predicate which relates the current state of the system to any inputs or outputs used by the operation.

We need to answer these questions:

- Does the operation require any inputs?
- Does it produce any outputs?
- When is it possible to perform this operation (preconditions)?
- What is the relationship between the current state, the inputs, and the output (postconditions)?

Specifying a static operation

Consider the operation of looking up the numbers linked to a given name.

- Does the operation require any inputs?
- Does it produce any outputs?
- When is it possible to perform this operation (preconditions)?
- What is the relationship between the current state, the inputs, and the output (postconditions)?

Specifying a static operation

Consider the operation of looking up the numbers linked to a given name.

- Does the operation require any inputs?
  - The name being looked up
- Does it produce any outputs?
  - The set of numbers related to that name
- When is it possible to perform this operation (preconditions)?
  - Anytime? Or only if the name is stored in the phone book? Let’s assume the latter.
- What is the relationship between the current state, the inputs, and the output (postconditions)?
  - The output is the set of numbers related to the input name.

Specifying a static operation

```
pred lookupNumbers [
  pb: PhoneBook,
  n: Name,
  result: set Phone
]
{
   n in pb.known                 // precondition: name is already known
   result = pb.number[n]   // postcondition
}
```

Specifying a dynamic operation

If an operation can change the state of the system (it is *dynamic*), we specify it using a predicate which relates the state of the system before the operation to the state afterwards.

We need to answer these questions:

- Does the operation require any inputs?
- Does it produce any outputs?
- When is it possible to perform this operation (preconditions)?
- How does the operation change the system (postconditions)?
- Are there parts of the system that are unaffected (frame conditions)?

Specifying a dynamic operation

Consider the operation of adding a new name to the phone book.

– Does the operation require any inputs?

– Does it produce any outputs?

– When is it possible to perform this operation (preconditions)?

– How does the operation change the system (postconditions)?

– Are there parts of the system that are unaffected (frame conditions)?

Specifying a dynamic operation

Consider the operation of adding a new name to the phone book.

- Does the operation require any inputs?
  The name being added
- Does it produce any outputs?
  No. (In practice, there could be a confirmation message.)
- When is it possible to perform this operation (preconditions)?
  The name should not already be in the phone book.
- How does the operation change the system (postconditions)?
  The new name is added to the set of known names.
- Are there parts of the system that are unaffected (frame conditions)?
  The relationship between names and numbers does not change.

Specifying a dynamic operation

```
pred addName[
   pb, pb' : PhoneBook,
   n:Name
]
{
   n not in pb.known                  // precondition: name is previously unknown
   pb'.known = pb.known + n           // postcondition: new name added
   pb'.number = pb.number             // frame condition: number relation is unchanged
}
```

Sanity checks

There are a number of ways that we can check if the specification makes sense:

- Write and run a predicate (often called show) to see models of the system.
- Run a predicate specifying an operation to see a model of what that operation does.
- Write and run new predicates to see how an operation works in some specific situation.
- Write and check assertions to verify general properties of the system and its operations.

Sanity checks

We can look at examples of phone books that model our specification by running a “show” predicate:

```
pred show { }
run show          // to see models of the phonebook system
```

We can see what the operation of adding a name does by running the predicate that specifies it:

```
run addName   // to see models of the addName operation
```

Sanity checks

To see how an operation works in a specific situation, we can run a new predicate describing that situation. For example, we might want to see if it is possible to add a new name to an empty phone book:

```
pred addNameToEmptyBook [
  n:Name,
  pb, pb' : PhoneBook ]
{
   pb.known = none        // pb has no known names
   addName[pb, pb', n]   // pb' is result of adding name n to pb
}
run addNameToEmptyBook
```

Sanity checks

We can write assertions to check general properties that (we hope) are implied by our specification. For example:

```
assert lookupNewName {
    all pb, pb' : PhoneBook |
    all n:Name |
    all result: set Phone |
       (addName[pb, pb', n] and          // If we add a new name,
        lookupNumbers[pb', n, result])    // then look up that name,
          implies result = none         // the result is the empty set.
}
check lookupNewName
```

Adding more operations

The rest of this lecture will be done interactively in class using Alloy (if time permits).

We will choose some other operation to specify.

The results will be posted on the CSC9P6 website after the lecture.

During the next practical class, you will work on extending the specification to include more operations.