The Shopping Scanner:

A Worked Example using *Perfect Developer*

Classification	Public
Author	D. Crocker, Escher Technologies Ltd.
Issue	1
Date	1 March 2004

1 Introduction

This worked example is based on an assignment presented by Prof. Steve Schneider of Royal Holloway University of London [1]. I am grateful to Prof. Schneider for allowing me to publish this adapted version of his idea.

Some supermarkets provide shoppers with a handheld scanner, allowing shoppers to scan their own purchases and thereby speed up the checkout process. This example concerns the specification of such a scanner.

The scanner has a barcode reader, an LCD display, and buttons labelled "+", "-" and "=". The scanner tracks the items in the shopper's trolley, as follows:

- When the "+" button is pressed, the scanner is activated, the scanned item name and its price are displayed, and the item scanned is assumed to have been added to the trolley;
- When the "-" button is pressed, the scanner is activated, the scanned item name followed by a minus-character and the item price are displayed, and the item scanned is assumed to have been removed from the trolley;
- When the "=" button is pressed, the number of items and the total price of all items in the trolley is displayed.

For the purposes of this exercise, we assume the items in the trolley to be of type *Good* which is an enumeration containing values such as *apple*, *banana*, *tunaroll*. The barcode scanner will be assumed to provide a value of type *Good*. Failure to scan or recognize an item will not be considered.

The scanner maintains a price list, which stores the price of every item of type *Good*. All prices are greater than zero. Prices are defined when the scanner is initialized but are not affected by pressing the buttons. The total price of all items in the trolley is normally the sum of the prices of all the items therein. However, a proposed extension of the specification allows "meal deals" to be defined. A meal deal is a combination of two or more distinct items, such that no item appears in more than one meal deal. The price of a meal deal is greater than zero but less than the total price of its constituent items.

Our task is to specify:

- The state variables of the scanner, including the display;
- The methods *totalPlus*, *totalMinus* and *equals* corresponding to the operation that takes place when the "+", "-" or "=" button (respectively) is pressed.

2 Skeleton

We start by writing a skeleton class to contain the specification, together with any ancillary type declarations we require. Here is such a skeleton:

```
class Good ^= enum apple, banana, tunaroll, crisps, coke end;

class PriceList ^= ?;

class Scanner ^=
abstract
  ?;
interface
  schema !totalPlus(item: Good)
    post ?;

  schema !totalMinus(item: Good)
    post ?;

  schema !equals
    post ?;

  build{initialPriceList: PriceList}
    post ?;
end;
```

None of the three schemas has a precondition, because there is nothing to stop the shopper pressing any of the buttons regardless of the state of the scanner.

3 Scanner state

The scanner state (prior to specifying "meal deals") comprises:

- The displayed text;
- The contents of the trolley;
- The price list.

For now, we will assume no limitation on the length of the displayed text (which is obviously unrealistic!), so the displayed text is just a string.

The trolley contents will obviously be some sort of collection, and we should choose the simplest sort that suffices. A set is unsuitable because the trolley may contain multiple instances of a particular *Good*, but a bag suffices.

The price list maps all possible values of type *Good* to a *Price*. We will represent a price as a natural number, which will be the price in pence (or cents etc. depending on your country).

Here is the class skeleton with the state variables defined and some new type definitions:

```
class Good ^= enum apple, banana, tunaroll, crisps, coke end;
```

```
class Price ^= those x: nat :- x > 0;

class PriceList ^= those pl: map of (Good->Price) :- forall g: Good :- g in pl;

class Scanner ^=
abstract
  var prices: PriceList,
      display: string,
      trolley: bag of Good;

interface
  schema !totalPlus(item: Good)
    post ?;

  schema !totalMinus(item: Good)
    post ?;

  schema !equals
    post ?;

  build{!prices: PriceList}
    post display! = "Ready",
         trolley! = bag of Good{};
end;
```

We have specified that the price list is initialised from a constructor parameter, and defined suitable initialisation for the display and trolley.

4 Specifying the schemas

The *totalPlus* schema is very simple to specify:

```
schema !totalPlus(item: Good)
  post trolley! = trolley.append(item),
       display! = item.toString ++ " " ++ prices[item].toString;
```

In specifying *totalMinus* we need to consider the possibility of trying to remove an item that is not in the trolley:

```
schema !totalMinus(item: Good)
  post ( [item in trolley]:
           trolley! = trolley.remove(item),
           display! = item.toString ++ " " ++ (-prices[item]).toString,
         []:
           display! = "Item not in trolley"
       );
```

In specifying *equals* it is convenient to declare a separate function that yields the total price of all items in the trolley:

```
abstract
  …
  function totalPrice: nat
    ^= ( [trolley.empty]: 0, []: + over (for x::trolley yield prices[x]) );

interface
  …

  schema !equals
    post display! = (#trolley).toString ++ " items, total " ++ totalPrice.toString;
```

In defining *totalPrice*, the return type is **nat** and not *Price* because if the trolley is empty, the total price does not obey the type constraint of type *Price*. Furthermore, we have to deal with the case of an empty trolley separately in computing the total, because the **over** construct requires a non-empty collection as its operand.

The completed specification is therefore as follows:

```
class Good ^= enum apple, banana, tunaroll, crisps, coke end;

class Price ^= those x: nat :- x > 0;

class PriceList ^= those pl: map of (Good->Price) :- forall g: Good :- g in pl;

class Scanner ^=
abstract
  var prices: PriceList,
      display: string,
      trolley: bag of Good;

  function totalPrice: nat
    ^= ( [trolley.empty]: 0, []: + over (for x::trolley yield prices[x]) );

interface
  schema !totalPlus(item: Good)
    post trolley! = trolley.append(item),
         display! = item.toString ++ " " ++ prices[item].toString;

  schema !totalMinus(item: Good)
    post ( [item in trolley]:
             trolley! = trolley.remove(item),
             display! = item.toString ++ " " ++ (-prices[item]).toString,
           []:
             display! = "Item not in trolley"
         );

  schema !equals
    post display! = (#trolley).toString ++ " items, total " ++ totalPrice.toString;
```

```
  build{!prices: PriceList}
    post display! = "Ready",
         trolley! = bag of Good{};
end;
```

5 Adding "meal deals" to the specification

To add meal deals to the specification, we need to:

- Add a state component that defines the meal deals available; and
- Modify the *totalPrice* function to take account of meal deals.

A meal deal comprises a set of *Good* associated with a *Price* for the complete meal deal. The deal price must be less than the sum of the prices of its elements. It will be convenient to define a function *discount* that yields the discount provided by a meal deal.

Here is a modified definition of the state, together with the *discount* method:

```
class MealDealCombo ^= those x: set of Good :- #x >= 2;

class Scanner ^=
abstract
  var prices: PriceList,
      mealDeals: map of (MealDealCombo -> Price),
      display: string,
      trolley: bag of Good;

  invariant
      forall x, y:: mealDeals.dom:- x = y | x ## y;

  function discount(d: MealDealCombo): int
    pre d in mealDeals
    ^= (+ over (for x::d yield prices[x])) – mealDeals[d];

  invariant
      forall d::mealDeals.dom :- discount(d) > 0;
```

In declaring class *MealDealCombo*, we have specified that a meal deal must contain at least two items. We have expressed the constraint that an item cannot be in more than one meal deal by way of the first invariant (although we could equally well have used a type constraint, since it relates to a single abstract variable). Note that the "##" operator means "disjoint". The second invariant declares that every meal deal provides a positive discount.

To determine the total price, we observe that any trolley can be broken down into a number of occurrences of each meal deal plus a remainder, such that the remainder contains no meal deals. The breakdown is unique because of the constraint that no item may appear in more than one meal deal. We will start by defining an operator that defines the number of times that a particular meal deal occurs in the trolley:

```
function occurrencesInTrolley(d: MealDealCombo): nat
```

```
    satisfy d.rep(result) <<= trolley,
            ~(d <<= (trolley – d.rep(result)).ran);
```

The above definition is an implicit one and it is likely that it will need to be refined to a more executable form before *Perfect Developer* can generate code for it. An alternative specification is:

```
function occurrencesInTrolley(d: MealDealCombo): nat
    ^= occurrencesIn(trolley, d);

function occurrencesIn(t: bag of Good, d: MealDealCombo): nat
    decrease #t
    ^= ( [d <<= t.ran]:
            occurrencesIn(t -- d.rep(1), d) + 1,
         []:
            0
       );
```

Now we can revise our definition of *totalPrice*:

```
function totalPrice: nat
   ^= ( [trolley.empty]:
           0,
        []:
           ( let basicPrice ^= + over (for x::trolley yield prices[x]);
             let discounts ^= + over (for x::mealDeals.dom
                                        yield discount(x) * occurrencesInTrolley(x)
                                     );
             basicPrice - discounts
      );
```

Note that there is at least one error in the above specifications!

6 Exercises

The following exercises are left to the reader.

6.1 Verification

Use the Verify function of *Perfect Developer* to check for possible errors in the specification. For each unproven verification condition, either fix the specification, or construct an informal proof argument. If possible, provide your argument to *Perfect Developer* by way of additional assertions, properties or axioms so as to achieve automated proof of all verification conditions.

6.2 Adding properties

A supermarket is interested in purchasing scanners but is very worried that it might be possible to change the price list by pressing the buttons, or that if a customer adds an item and

then removes it, the scanner might fail to restore the trolley to its original state. Add extra post-assertions and/or properties to verify that these things cannot happen.

6.3 Using inheritance

A manufacturer of scanners wishes to specify and develop software for a *BasicScanner* (without meal deals) and a *ScannerWithMealDeals*. In true object-oriented style, the manufacturer wishes to achieve re-use by inheritance.

Modify the *Scanner* specification given earlier to turn it into a *BasicScanner* specification, such that the *totalPrice* method can be overridden in descendant classes. Then define a class *ScannerWithMealDeals* that inherits *BasicScanner* and provides the meal deal functionality. Ensure that any additional post-assertions and properties that you have added are declared in *BasicScanner* and inherited by *ScannerWithMealDeals*, if they are still applicable.

Hint: you will need to add a **confined** section to class *BasicScanner*, containing the declaration of *totalPrice*. You may also need to redeclare some of the state variables as **confined** functions. You may find it useful to use the **super** keyword within the new definition of *totalPrice*.

6.4 Discount for large total

Specify a scanner that behaves like *BasicScanner* but provides a 5% discount if the total price before discount is at least 10 000 pence (or cents etc.). You can again use inheritance to maximise re-use.

6.5 Multibuy

Many supermarkets operate a “multibuy” discount whereby if the trolley contains a certain number of items from a family, a discount is applied. This differs from a meal deal in that there may be just one item in the family, and a specified total number of items belonging to the family must be present to earn the multibuy discount. For example, a multibuy family might comprise the members *coke* and *lemonade*, and the qualifying number might be 3. For every 3 cokes and/or lemonades in the trolley, the discount would be applied. Specify a *MultibuyScanner* (inheriting *BasicScanner*) that provides this functionality.

6.6 Limited display size

If the text string written to the display is more than 20 characters long, it will be truncated. This is obviously undesirable. Constrain the display to hold a maximum if 20 characters. Add other constraints and adjust the specification to ensure this will never be breached. Verify your changes.

6.7 Generate a prototype

Generate prototype code for one of your scanner classes in Java and interface it to a graphical front-end that simulates the buttons and the display. You may find it helpful to modify one of the sample Java front-ends that Escher Technologies provides (e.g. the one in the *Examples/Graphical* subdirectory of a *Perfect Developer* installation).

7 References

[1]. *Formal Methods at Royal Holloway: Perspectives and Pitfalls*, Steve Schneider. Keynote speech of *Teaching Formal Methods: Practice and Experience* workshop, Oxford Brookes University, 12 December 2003. Available at http://wwwcms.brookes.ac.uk/tfm2003/ and at http://www.cs.rhul.ac.uk/research/formal/steve/talks/tfm.ps (March 2004).

End