Estimated workload: 1.5 days full time (2 students)

Laboration 4.1

Your task is first to extend the type List defined in [Sommerville10] (and presented at the lecture) with an operation CircShift that circularly shifts lists. Similarly to in Laboration 1, circularly shifting a list means removing the first element and appending it to the end of the list. More specifically, the operation shall take as parameters a list and an integer number n, and return the result of circularly shifting the list n times. For example, if myList = [x, y, z] then

CircShift(myList, 1) = [y, z, x]

CircShift(myList, 2) = [z, x, y]

CircShift(myList, 3) = CircShift(myList, 0) = [x, y, z]

Extend the type with another operation Count that, given a list and an element, returns the number of times the element occurs in the list. For example, Count([1, 2, 1], 1) should return 2, Count([1, 2, 1], 2) should return 1, and Count([1, 2, 1], 3) should return 0.

Provide the signatures of the operations and axioms defining them in terms of the already given constructors and/or operations. For simplicity, your specification need only be correct for non-empty lists and non-negative values of n.

Check your specifications in Visual Studio by first translating them to the F# programming language. For your convenience, the file list.zip is a Visual Studio solution with an F# version of the basic List type.

Hint

For those who have never used recursion to specify (or implement) an operation that repeats an action n times, the following example may be helpful. Consider an operation \_GetByIndex : (List(Elem), Integer) ? Elem that returns the nth element in a list. Assuming that the operation only needs to work for 1 = n = Length(L), it can be defined by the following two axioms:

\_GetByIndex(L, 1) = Head(L)

\_GetByIndex(L, n) = \_GetByIndex(Tail(L), n - 1)

As you may already have figured out, an operation GetByIndex that also checks that the parameters are valid, can now be defined as follows:

GetByIndex(L, n) = if n < 1 or n > Length(L) then Error else \_GetByIndex(L, n)

Laboration 4.2

You will now work with a formal specification for an interface of a class (or subsystem) that will implement the central functionality of an automatic teller machine (ATM). You will use the Object Constraint Language (OCL) to specify a state model and pre- and post-conditions for the operations of the interface. A useful source for information about OCL is Chapter 7 of [OMG10].

To write and check the specification, you will use the free tool Object Constraint Language Environment (OCLE). Start by downloading the file ocle-2.0.4.zip and unzip it into a new folder (e.g. a subfolder of "My Documents" if you are working in the computer rooms). Test that OCLE works by executing the script run\_windows.bat (there is also a script run\_linux.sh that can be used to run OCLE on Linux, but that requires a little manual editing).

The syntax of the interface has already been specified in UML and the desired behavior has been partly specified in OCL. Download the file atm.zip and unzip it into a subfolder named "Assignment 4" of your Subversion trunk. The files in the archive are:

atm.uml: StarUML project file specifying the syntax of the ATM interface and two auxiliary data types for the OCL specification

atm.pdf: printout of the diagram from atm.uml

atm.xml: XML Metadata Interchange (XMI) file with the same interface and data types as atm.uml, exported from StarUML

atm.bcr: text file with OCL specifications

atm.oepr: OCLE project file with references to atm.xml and atm.bcr

Open atm.uml in StarUML and look at the class diagram to see the interface and data types. (If you cannot run StarUML, open atm.pdf instead.) In OCLE, open the project atm.oepr and verify that you can see the interface and data type definitions by selecting the UserModel tab. Next, select the Project tab and double-click atm.bcr under Constraints to open the file with the OCL specifications. Check that the specifications contain no syntax or type errors by pressing the "Compile project specifications" button (move the mouse over the buttons just below the menu to see a description of each).

As you can see, the file contains specifications of a state model - comprising the variables state, accounts and currentAccount - and the operations InsertCard, EnterPIN, ChangePIN, and Cancel. Notice that the specification of these operations only considers the behavior of the system when the operations are invoked while the system is in certain states. For instance, the specification of InsertCard only considers the case that the operation is invoked while the state is WaitingForCard. Also, notice that the formal model only specifies how the state variables are changed by each operation and not what actions are performed by the system. For instance, at the end of the specification of InsertCard, the fact that the system ejects the card if it is not recognized is stated in a comment but not expressed in the formal model.

The meaning of the state variables is as follows:

state - Can have six different values:

WaitingForCard - The ATM is ready to accept a card.

WaitingForPIN - A valid card has been inserted and the ATM is waiting for a PIN to be entered.

WaitingForCommand - A valid card has been inserted and the correct PIN entered. The ATM is waiting for a user to select a command (withdraw money, change PIN or cancel).

WaitingForCardAndCashRemoval - The user has selected to withdraw an amount which has been allowed. The ATM has ejected the card and is waiting for the user to remove it before dispensing the cash.

WaitingForCashRemoval - The ATM has dispensed some cash and is waiting for the user to remove it before accepting new cards.

WaitingForCardRemoval - The ATM has ejected a card and is waiting for the user to remove it before accepting new cards.

accounts - A set of ATLState objects, representing the set of accounts that the ATM is currently managing. The card numbers associated with the objects in accounts are unique. When a card is confiscated after three incorrect PINs in a row, the corresponding object is removed from accounts.

currentAccount - The account object of the user currently served by the ATM.

Your task is now to specify the remaining operations in the interface:

Withdraw - Can be invoked after the user has entered a correct PIN. If the amount is positive and not greater than the balance, the system shall update the balance, eject the card and wait for the user to remove it before dispensing the cash. If the amount is greater than the balance, the system shall let the user select another command.

RemoveCard - Can be invoked after the system has ejected a card. Depending on the state (WaitingForCardRemoval or WaitingForCardAndCashRemoval), the system shall either be ready to accept a new card or dispense the customer's cash.

RemoveCash - Can be invoked after the system has dispensed some cash. After the operation, the system shall be ready to accept insertion of a new card.

For each operation, write one or more specifications with pre-/post-conditions. Include comments explaining the meaning of the conditions and any actions the system performs. Actions - such as ejecting the card, dispensing the cash, printing an error message, etc. - do not need to be expressed in the formal model, but should be stated in comments.

The operations described above are of course a simplified model of the functionality you would expect from an ATM. For instance, a more realistic model would have to consider what should happen if the system is not able to deliver a selected amount (because it does not have enough bills of the required denominations) or if a user does not remove an ejected card or dispensed cash within some reasonable time. Given these limitations, it should be possible to specify the operations using only the state variables already defined. You are however allowed to extend the state model by introducing new variables or extending the auxiliary data types (which can be done directly in OCLE or in StarUML).

As you write your specifications, you should periodically check that they are free of syntax and type errors by executing the "Compile project specifications" command. Make sure to check the specifications before you submit your solution.

Hint

A common mistake is to treat the "=" sign in post-conditions as an assignment (as in C/C++/C#) and, consequently, to neglect using the "@pre" notation. For example, one might write "x = x + 1" instead of "x = x@pre + 1". Keeping in mind that both pre- and post-conditions are equations and not executable code, it is easy to see that the first is equivalent to false, since solving the equation for x yields "0 = 1". Note that the OCLE syntax checker will not help you to avoid such mistakes, since "x = x + 1" (as well as "0 = 1") is valid OCL syntax.

Reporting the assignment

Make sure your Subversion repository has a subfolder called "Assignment 4" containing the following:

A document (Word or PDF) with your solution to Assignment 4.1

A subfolder with your F# project to check the solution to Assignment 4.1

A subfolder with the files of the OCLE project you have created/modified to solve Assignment 4.2 - at least three files of type .oepr, .xml, and .bcr, respectively

After you have uploaded your assignment in your Subversion repository, you should notify us in Blackboard that you are done. To notify us use submit option under Laboration 4. As a message you have to write your group number. Each group member has to submit the assignment via Blackboard individually.

Attached Files:

File atm.zip (26.614 KB)

File ocle-2.0.4.zip (2.749 MB)

File List.zip (3.141 KB)