email: egonzale@vub.ac.be
office: 10F731

7 goShopping: Debugging AmbientTalk programs with REME-D

Lab session material available at Pointcarre, and at http://soft.vub.ac.be/amop/teaching/dmpp

7.1 Idea

The purpose of this exercise is get you familiarized with reflective programming in AmbientTalk by means of REME-D [1], a distributed debugger designed for AmbientTalk applications, and reflective programming in AmbientTalk. REME-D has been entirely implemented in AmbientTalk using the language reflective capabilities and it has been integrated in the Eclipse AmbientTalk plugin (IdeAT) using the Eclipse Debug Framework. The session consists of two separate parts:

- First, you will get familiarized with REME-D features and Eclipse GUI. The lab material provides you with an application that contains errors. You should fix them using REME-D debugging features.
- Afterwards, you will extend REME-D's functionalities with a new sort of breakpoint, and step command.

7.2 Part 1: Finding bugs in the goShopping application

The provided application is a sample shopping application that needs to process purchase orders. Before the shop can acknowledge the order, it must verify three things: 1) whether the requested items are still in stock, 2) whether the customer has provided valid payment information and 3) whether a shipper is available to ship the order in time. Figure 1 depicts this application which consists of 5 actors. The application's default actor is supposed to host the application GUI, while the buyer actor processes order purchases. In response to a `go` message, the buyer actor sends out three messages to the product, account and shipper actors (as also shown in the figure).

To keep this part simple, we do not make use of futures. Instead, the buyer makes use of an *AsyncAnd* abstraction. The constructor of an `AsyncAnd` object takes two parameters: a number indicating how many affirmative replies the `AsyncAnd` should receive before it invokes `callback<-run(true)`, and the callback object to notify. The callback object thus needs to implement the message `run(boolean)`. In the `goShopping` application, all three actors simply send an affirmative reply to the `AsyncAnd` callback. As a result, when you run the application, it should print the following to the console "Got answer: true".

Your task consists on fix and improve this application as follows:

[1] read as remedy

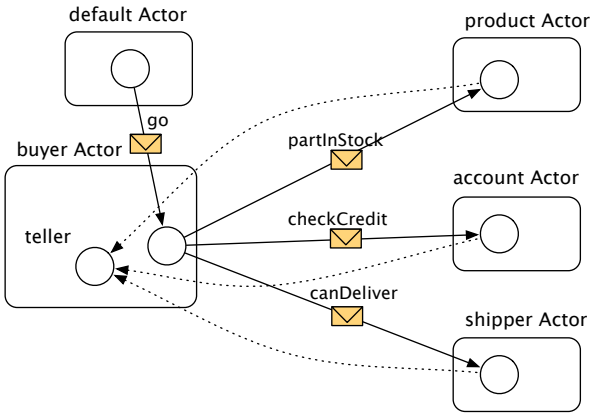

Figure 1: The shopping application

(a) Currently the application contains a bug. If you run it, it prints "Got answer: false" rather than "Got answer: true" . Run again the application now in debug mode and fix the bug using REME-D debugging features.

Hint: You may want to set a breakpoint in the `go` message send and stepping into its execution to check whether the contacted actors reply to the `AsyncAnd` abstraction.

Hint 2: If you didn't you catch the bug yet, you may want to set a breakpoint in the `run` method to check whether the `AsyncAnd` abstraction works properly.

(b) Let's use REME-D epidemic features to catch another bug. To this end, you will need to use `goShoppingSectionB.at` file instead. It works similarly than the described `goShooping` application but it employs futures instead of using the explicit `AsyncAnd` abstraction. In addition, once the shop acknowledges the order, it contacts a warranty broker to suggest the client a warranty for the purchases item by means of the `getExtendedWarranty` asynchronous message. To do so, the warranty broker contacts several insurance agencies, and returns the best quote. However, it currently always returns an negative quota. Use REME-D debugging features to fix this bug.

To reproduce the bug you need to:

1. run the warranty broker code with (stored in the `warrantyBroker.at` provided in the lab session material) with -Xdebug option (so that this code is debuggeable).
2. run in debug mode the `goShoppingSectionB.at` file. Note that this file uses the `goWithInsurance`, rather than the `go` message which used the explicit `AsyncAnd` abstraction.

Hint: Since after fixing the previous bug you know that the default actor gets correctly the acknowledgement of the purchase order, you may want to set a breakpoint on the `getExtendedWarranty` message send.

7.3 Part 2: Extending REME-D's functionality

Now you are going to take the first steps to learn the reflective model of AmbientTalk by modifying REME-D implementation to extend its functionality with the following two new features:

Symbol breakpoints. A symbol breakpoint defines a breakpoint on a method name corresponding to the given symbol. The actor execution pauses before the receiver invokes a method whose name is the given symbol. For example, when setting a symbol breakpoint on the AmbientTalk symbol `'foo`, the AmbientTalk actors running in debug mode will stop when they receive any asynchronous message whose name corresponds to `'foo`.

Step Until command. A step until command defines a step command which steps over the execution of a number of messages until the given message name is found. For example, imagine that an AmbientTalk actor is paused and its message queue contains `<o<-foo | o<-bar() |o<-baz()>`. If the user instruments the debugger to make a step until the symbol `'baz`, the actor will be resume and execute its messages, and pause again when `o<-baz()` reaches the top of the message queue.

Your task consists on adding a symbol breakpoint, and step command as follows:

(a) Extend REME-D's breakpoints module to include the implementation of the symbol breakpoint:

1. Add the definition of the symbol breakpoint in the `breakpoints.at` file provided in the lab session material.

   Hint: A symbol breakpoint can be implemented as an extension to a conditional breakpoint tagged with the `ReceiverBreakpoint` breakpoint type.

2. Since the IdeAT plugin does not offer GUI support for symbol breakpoints, you will need to manually inject the breakpoint to the debugger manager to test that the added breakpoint works. To this end, add a `setSymbolBreakpoint` method to the `localInterface` object of the debugger manager (implemented in the `debuggerManager.at` file). The method should take as parameter a symbol, and it should first create a symbol breakpoint and then, send it to all actors currently being debug.

   Hint: The debugger manager provides a `listLocalManagers` method that returns a table with all local managers currently in the debug session.

3. To test the newly added breakpoint type in REME-D, you can call the `setSymbolBreakpoint` method by means of the "AT Debugger Manager" console in Eclipse. You can access the debugger manager as follows `eclipseController.debuggerSessionBhv`.

If your breakpoint implementation works, you should see that the GUI stops before executing a method invocation with the given name :)

(b) Extend REME-D stepping behaviour to include the implementation of the symbol breakpoint as follows:

1. Add a `stepOverUntil` method on the `interfaceDebuggerManager` object of the local manager which works similarly to the other step command, i.e sets the pause state to STEPUNTIL but schedules all messages from the actor inbox.
2. Modify the `schedule` meta level method to check if an actor is in step until mode when an actor is asked to schedule a message from its inbox. If this happens you will need to check whether the message scheduled is the one that has to pause again the actor's execution.

   Hint: You can make use of a symbol breakpoint to check if the message needs to pause again the execution of the actor.
3. To test the newly added step command, again, you will need to first add a `stepOverUntil` method to the `localInterface` object of the debugger manager and call it by means of the "AT Debugger Manager" console in Eclipse.