Pre-Hand-in

Program description

Supported features

- Initial event state (included, pending, executed)
- All relation types
- Loading of workflows from files

Unsupported features

- Milestones
- Groups
- Passing of multiple roles when attempting to execute or get visible events
- Deletion of events and workflows
- Multiple workflows
- Sub-workflows
- Other data associated with events or relations

REST API

GET http://localhost:8080/<workflow>/<event>/<included | pending | executable | executed>
POST http://localhost:8080/<workflow>
GET http://localhost:8080/<workflow> data: <role>
POST http://localhost:8080/<workflow>/<event> data: <0/1><0/1><0/1>
[role1 [role2 [roleN…]]]
PUT http://localhost:8080/<workflow>/<event>/<relationtype> data: <eventname>
PUT http://localhost:8080/<workflow>/<event>/executed data: <role>

Workflow.fs

This is the “back-end” of the workflow server. We are currently only allowing the server to keep one active workflow, but it should easily be possible to extend it to support more running instances and different workflow types. As this is still not a well-defined requirement, we do not support it yet. The module uses an immutable data structure to ensure that nothing is subtly mutated. It models the workflow as a map of names to a record type for an event. The module only exports a limited interface (using Workflow.fsi), so as to ensure proper encapsulation. We try to convert ‘generic’ input (strings, numbers, etc.) to ADTs as often as possible, so that we might properly match all our possible input cases without subtle errors. This ensures that as few logic errors as possible persists, and that we take as much advantage of the type and match checking of the F# compiler, helping us with avoiding many common errors of more mutable or more generic languages. The way that we ensure that updates are executed in the correct order, is by recursively interpreting a list of update commands (represented using ADTs), and prepending more commands when an action has

consequences. In our case that is mainly when an event is executed, and its dependent events (to which it has a ‘condition’ relationship) are notified of this to update their executability status. We have chosen to model the workflow in a way in which events do not check “backwards”, but instead update “forward” when their state changes, This is probably mainly possible since we use a completely isolated simulation, since a networked simulation would have to go to greater extents to ensure that all data is consistent, and that the server(s) handles transactions correctly.

Event.fs

This is the “front-end” of the workflow server. This module is responsible for receiving REST calls, and delegating them to the back-end as required, while also responsible for providing error messages for the user. In this module we use F#’s list matching capabilities to branch properly based on the shape of the requested URL.

Client.fs and EasyClient.fs

Client and easyclient are both based on the same concept, that is to give the user a way of interacting with the events. The client is role-based whereas the easyclient is not. Instead easyclient has more focus on requiring as little action from the user as possible. To use these programs, one executes them and a list of choices will always be presented, expecting the user to write a number+enter for the choice of interest. The client will start by requesting the user for the role to be used whereas the easyclient won’t.

Client.fs

This class is the module that allows for interaction between users and events. It provides a list of choices to the user and using F# pattern matching and recursive functions allows for the logic to be illustrated in a neat way. F# list matching capabilities also allows for functions such as a loop that outputs each event status to return a new list containing only a working events. The class is rather manual in the sense that it requires the user to even tell it to download the newest list of events, but in that sense it also accounts for the case with multiple clients running and users wishing to keep the event list anew as they please. The program has try/catches for every case that is believed to be returning a network related error. As such the program can handle communication with the rest of the system without crashing.

EasyClient.fs

This class is the module that allows for interaction between users and events. Its a stripped down version of Client.fs that tries to require as little from the user as possible. Functionality is the same except that removed choices such as “Get all events” and “View status” has been depleted, Instead these choices will be executed automatically when starting the program or executing an event.

Testing

We use the NUnit unit testing framework to test our program, though the program is written in F#. So far we only have tests for the responses of our server (a “black box” test), but we plan to add more later.
We have mainly tested two points

- Whether the state of the workflow is correct after various operations
- Whether the debug functionality works correctly

Things we still need to test

- Whether the server is truly RESTful
- Whether initial states work correctly
- Whether the server handles all requests without raising an exception
- Whether wrong operations return the right error messages

Instructions

Running the tests

1. Use NUnit to run the tests located in 'RESTfulWorkflow/Event.Test'
    a. This automatically compiles the workflow server in 'Event' and runs it.
    b. New windows might open and close at this point.

Running the program 'normally'

1. Compile and run the manager (setup program) located in ''RESTfulWorkflow/Manager"
    a. This automatically compiles the workflow server in 'Event' and runs it.
    b. Please note that the workflow server window must be kept open for the client program to be able to communicate with it.
2. Choose the file to load a workflow from by writing a number in the terminal and press enter
3. Now choose whether you want to use roles in the workflow
    a. If you choose to use roles, the execution of events will require the correct role.
    b. Since multiple roles are not supported by our client, you will most likely not be able to finish a workflow if the workflow uses more than one role.
4. Compile and run the client located in "'RESTfulWorkflow/Client"
    a. Follow the manual below.

Client manual

Available choices for is as follows.

1. Get all events (role-based client only, easyclient will do this automatically).
    a. Downloads the list of events for the given role.
2. Execute an event
    a. Requests the user to write the name of an event and it will be executed if possible.
3. Change BaseURL
    a. Requests the user to write a new baseURL to use aside from the default value of "http://localhost:8080/Test".
4. Change Role (role-based client only)
    a. Requests the user to write a new role to use.
5. View task/status (role-based client only, easyclient will automatically output this at startup and after executing an event).
    a. Prints a list of executable events followed by executed events.
6. Exit program
    a. Terminates the program.

Running the simple version (included in this hand-in)

Setup

use the two .exe file provided in the pre-handin

1. run "SetUp.exe" and let the resulting window run in the background.
2. run "Client.exe".

Usage

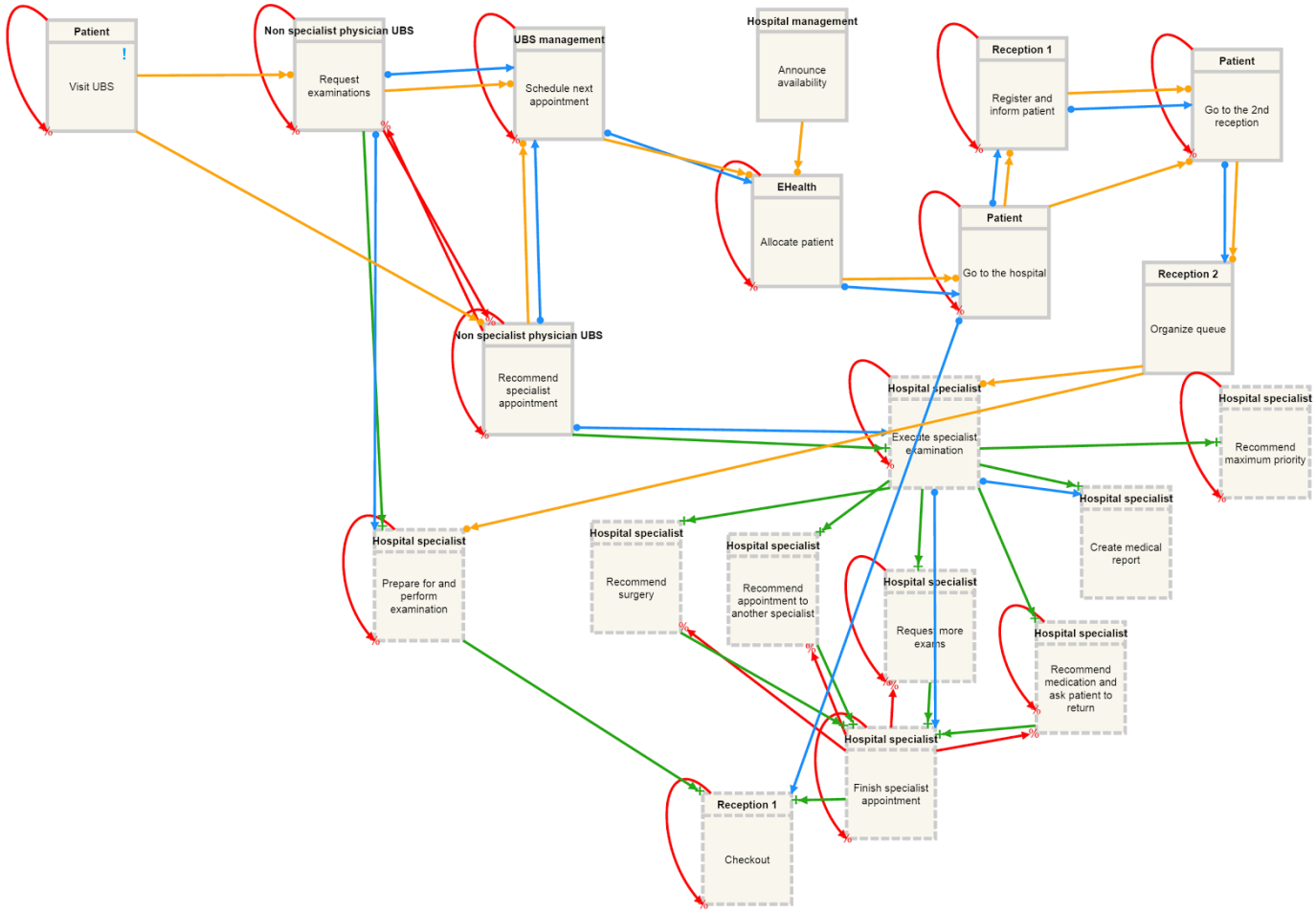
The Client will present all executable events.

1. Press "1" and enter to execute an event.
2. Write the name of the event to execute and press enter.

The Client will present a new list of executable events.

3. repeat step 1 and 2 to traverse the DCR Diagram

DCR Graph



It can be seen from the given DCR-Graph that the only pending task, is that a patient initially visits the medical care unit UBS. Furthermore, at all times the hospital might announce availability of spaces to the E-health system. After the patient visits the clinic, the first part of the workflow is completed. At this point, we are presented with two mutually exclusive possibilities, of the patient either being sent for a more specialized examination, or being referred to see some specialist, both are performed by the physician at the UBS.

Say that the physician requests further examination after the patients visit. This initiates a new part of the workflow with two pending tasks. The first is the actual performance of the examination needed, according to the check at the UBS. This task will not be executable before the second pending task is executed, which is sequence of prerequisite administrative tasks. Once these pending tasks have been performed the “Checkout” task will be included and pending. Performing this task will finish the workflow.

On the other hand, say the UBS physician recommends a specialist for the patient, and therefore schedules an appointment. The condition for proceeding is that there has been announced availability at the hospital. This initiates a pending task which is for the specialist to perform the examination, meaning the examination must occur, but it is not available before multiple other administrative steps have been completed.

Once these self-excluding tasks have been executed the task of examination excludes itself, and a list of tasks are included in the process including additional surgery, checkups and medication. Two pending tasks are now available, being the opportunity to finish the appointment or write up the medical report. The request for additional exams and medication to take can be executed once, before it excludes itself, while surgery and referrals can be performed any given number of times, until the specialist is finished with the appointment, and the patient moves to check out. However, if the specialist has not written the medical report yet, this task will still be pending and the workflow will not be completed before this task has been dealt with, since that is required for this part of the patient’s case to be considered closed.

Conclusion

We have written a DCR workflow system using F#, which supports generic workflows and created a DCR workflow based on our interpretation of the information in HealthCareProcess.pdf, provided by Eduardo from Brazil. The graph models the workflow following a patient in Brazil, from he visits a UBS for a check, and until his or her checkout from the hospital, including the exams and other events he or she might go through.

Questions

- Who makes an examination? Is it a Hospital specialist or someone else?
- What file types are most used at the hospital?
- Have you worked with DCR workflows before?