UNIVERSITÀ DI PISA

Università degli Studi di Pisa

Laurea Specialistica in Informatica

Advanced Programming

Final Term Paper

Start Date: 18/01/2016
Submission deadline: 7/02/2016 (send a single PDF file to attardi@di.unipi.it)

Rules:

The paper must be produced personally by the student, signed implicitly via his mail address.

You are allowed to discuss with others the general lines of the problems, provided that each student eventually formulates his own solution. Each student is expected to understand and to be able to explain his solution.

You are allowed to consult documentation from any source, provided that references are mentioned.

It is not considered acceptable:

- **to consult or setup an online forum, to request help of consultants in producing the paper**
- to develop code or pseudo-code with others
- to use code written by others
- to let others use someone's code
- to show or to examine the work of other students.

Violation of these rules will result in the cancellation of the exam and a report to the Presidente del Consiglio di Corso di Studio.

For the programming exercises you can choose a programming language among C++, C# and Java.

The paper must:

1. be in a single PDF file, formatted readably (**font size ≥ 10 pt** with suitable margins, single column), of **no more than 10 numbered pages**, including code: for each extra page one point will be subtracted from the score.
2. include the student name
3. provide the solution and the code for each exercise separately, referring to the code of other exercises if necessary.
4. cite references to literature or Web pages from where information was taken.

Introduction

In this project, you will develop a Workflow Coordination framework, including a Workflow Coordination Language (WCL). Here is an example of a workflow for coordinating a set of jobs expressed in WCL:

```
<workflow>
  <agent id="job1">
    <state name="pause">
       <trigger target="timer" type="alarm" message="%random()%" />
       <event type="timeout">
          <next state="request"/>
       </event>
    </state>
    <state name="request">
```

```
        <trigger target="coordinator" type="request"/>
        <event type="run">
           <next state="running"/>
        </event>
      </state>
      <state name="running">
        <code>…</code>
        <trigger target="coordinator" type="done"/>
        <next state="pause"/>
      </state>

      …
    </agent>
    <agent id="job2"> … </agent>
    …
    <agent name="timer">
      <event type="alarm" name="event1">
        <code> … </code>
      <event>
    </agent>
    <agent id="coordinator">
      <code>..declare code  for methods allowed(), delay(), etc.</code>
      <state name="running">
        <event type="request" name="event2">
            <if test="allowed(event2.sender)">
               <then>
                  <trigger target="%event2.sender%" type="run" />
               </then>
               <else>
                  <code>delay(event2.sender)</code>
               </else>
            </if>
         </event>
        <event type="done" name="event3">
            <code>done(event3.sender)</code>
            <while test="running_possible()">
                  <trigger target="%next()%" type="run" />
            </while>
         </event>
      </state>
    </agent>
  </workflow>
```

Each job can be in either state `pause`, `request`, `running`. When in `pause`, it does nothing, waiting for a timeout event that will change its state into `request`. The timeout is triggered by a message sent to a timer agent. When entering the `request` state a job will trigger an event to the coordinator, submitting a request for a permission to run. Upon a request event the coordinator will decide whether the requester is allowed to run, in which case it will send a run event to the requesting job, otherwise it will delay the request storing it in an internal queue. When a job receives a run event, it performs its code and then advances to state `pause` again, stops running and triggers the event `done` on the coordinator, which will then decide whether there is a delayed job that can be given permission to run.

The behavior of an agent is described by a state machine: within each state, event may cause some action and possibly a state transition, marked by the `<next>` element. Event elements may specify an attribute `name`, which will correspond to the name of a variable to be used for referring to the corresponding Event object. Event objects include a type, the id of the sender and an optional message, which can be referred in the code with e.g. `event1.sender`, `event2.message`.

Attributes values enclosed in %...% indicate expressions to be evaluated at run time, rather than constant literals.

The trigger element produces an event of the given type that is sent to the given target with the given optional body.

A `code` element represents executable code that should be inserted in the generated code. In case the code contains special characters, these should be denoted as HTML entities.

The WCL is used to generate code for the system, by means of a code generator.

Exercise 1

Design a set of classes to represent the syntactic constructs of WCL (e.g. Workflow, Agent, State, etc.) that are expressed in the XML notation.

Exercise 2

Implement a recursive descent parser for WCL without using external libraries or parser generators. Split the parser into a lexical analyzer, recognizing as tokens just tags and text, and a syntax analyzer, as presented in the slides of the course.

Exercise 3

Design the Workflow Coordination Framework, consisting in a set of abstract classes to be used to build an event system and that provide the runtime support for event scheduling, triggering, state transitions and concurrent execution. Each agent should run in its own thread. Explain the role of each class, for example, by a state transition diagram.

Exercise 4

Write a code generator that takes as input a workflow description consisting in the classes defined in Exercise 1, and generates actual executable code for running the workflow. Ensure to use polymorphism in the generator whenever appropriate, in order to make its code modular and reusable.

Exercise 5

Complete the specification of the example in the introduction and show the code produced for it by the code generator (for simplicity show the generated code for a single agent). Assume the coordinator can allow at most *n* jobs to run at the same time.

Exercise 6

Extend the WCL introducing the possibility of spawning new jobs, like this, where `next_id()` returns the next available id for an agent:

```
<spawn>
    <agent id="%next_id()%"> … </agent>
</spawn>
```

Extend the parser and the code generator, in order to handle the new construct.

Exercise 7

Describe the technique of tracing garbage collection. Explain which information the garbage collector needs to know about the runtime data structures and the program data structures in order to perform its task. Explain how the collector can obtain such information. Discuss any relation or similarity between these information required by a garbage collection and that provided by reflection facilities.