Projects for the MDE course

Rules:

1. Projects are to be performed by groups of 1 to 3 people.
2. Projects will be assigned on a first-come first-serve basis. In principle, no two groups can work on the same project. Please write an e-mail to juan.delara@uam.es as soon as you have formed a group and decided the project you'd like to work on.
3. Each project will be presented by both group members at the end of the course. The tentative date is **January 20th**. The presentation will be of 20 minutes (max), and should include a small demo of the project performed. The slides and the presentation can be in English or Spanish.
4. By the presentation day, each group needs to hand in: (a) the code of the project, (b) an explanatory document in the form of a research article.
5. The research article will be in Lecture Notes in Computer Science (LNCS) format (LaTeX template here, and Word templates here and here) with a length between 5 and 10 pages. The paper can be written in English or Spanish.
6. Typical sections of a research article include: abstract (a summary), introduction, approach, tool support, evaluation, related work and conclusions. As this is a small project, you may skip the related work section, but including it is a plus.
7. The project descriptions in this document include the minimal expected requirements, but you are free to add extensions, which is a plus.

Please ask the professors for possibilities to extend the chosen project to a master thesis, or possibilities for similar projects for master theses.

List of eligible projects

1. A DSL to design conversational games

The objective of this project is to facilitate the definition of conversational games and automate their execution. In this type of games, the protagonist goes through a maze made of rooms, where s/he finds objects with which s/he can interact in different ways. Zork (http://en.wikipedia.org/wiki/Zork) was one of the first games of this style.

In this project, you will create a textual DSL to configure conversational games with the characteristics that are detailed below. In addition, you will create a program that, given the description of a game created with the DSL, is responsible for executing it.

Requirements

In a conversational game, there is a labyrinth of connected rooms where there can be objects and characters. The player gives instructions to the protagonist with simple phrases to move around the rooms, interact with other characters, and perform actions on the objects.

Below, you can see a moment in the execution of a game. The red text after the symbol > corresponds to phrases that the player writes, while the black text in italics is shown by the game engine:

```
1  You have entered a room with a door to the north and a door to the east. You
2  can see a potion and a magician.
3  The magician greets you: "Abracadabra"
4  > ask magician about potion
5  You ask the magician: "Should I drink this potion?"
6  The magician answers: "Yes, it will increase your health"
7  > take potion
8  Potion taken
9  > water potion
10 That action is not possible.
11 > drink potion
12 Potion drunk. Your level of health increases by 10.
13 > go north
14 You have entered a room. You can see a box.
15 > open box
16 The box does not open.
17 > open box with key
18 The box has been opened. It has a message inside.
19 > take message
   …
```

As the example shows (lines 1-3), when the protagonist enters a room, the game lists the doors the room has, the objects and characters in the room, and if there is some character, her/his greeting phrase.

The interaction with the characters is limited to ask about the objects that are in the room (line 4). The character may respond that s/he does not know anything about the object, that the protagonist should do some action on it (e.g., drink potion, as in the example, because it will increase the level of health), or that the protagonist should not make a certain action (e.g., do not drink the potion, if it reduces the level of health).

Only a limited number of actions can be performed on each object found, depending on the type of the object. For example, it is possible to take a potion (line 7) or drink a potion (line 11), but it is not possible to

water a potion (line 9). For some of these actions, it is also necessary to indicate one or more additional objects that the protagonist owns. For example, to open a box, a key is required (lines 15 and 17), which the protagonist must have previously collected in another room. Performing an action on an object yields a result. For example, “take potion” adds the potion to the list of belongings of the protagonist, “drink potion” increases one of the characteristics of the protagonist (the level of health), and “open box with key” makes available a new object (message) that previously did not exist in the room.

To complete the game, the protagonist must achieve a predefined goal, such as reaching a room containing a treasure carrying a key and having a health level > 50. The game may also end unsuccessfully for various predefined reasons, e.g., when the health level of the protagonist reaches 0, or if the protagonist performs a certain action such as “drinking poison”.

The DSL to be developed should allow the configuration of (at least) the following characteristics of a game:

- Rooms of the game and how they are connected.
- Properties of the protagonist (e.g., level of health) and initial value (e.g., level of health = 100).
- Existing objects in the game (e.g., key, box, potion, etc.).
- Admissible actions on each object (e.g., drink potion, open box with key, etc.), indicating what happens when doing it: a text is printed on the screen, a certain characteristic of the protagonist is modified, a new object becomes available, etc.
- Characters in the game (e.g., magician, jester, ghost, etc.), indicating how they greet when entering a room they are in (e.g., "Abracadabra"), as well as their response to questions about specific objects.
- Distribution of the elements of the game in the rooms.
- Conditions for ending the game, in case of success or failure.

In addition to the DSL, the project needs to include a program that runs the game based on its definition. The interaction with the program should be similar to the listing shown before (**but the game itself can be in Spanish**). You will probably need to write a code generator to synthesize configuration code to be used by the game engine.

Finally, the project should include a model transformation from the game design into Petri nets, to verify that winning/losing the game is feasible.

2. A DSL to create REST services

The REST (REpresentational State Transfer) architectural style for web services advocates creating services without state (stateless), client-server and cacheable, normally using HTTP as the transport protocol. The REST style is a very popular mechanism on which services such as Twitter, Facebook or Flickr offer access to their data through an API.

In general, a REST API over HTTP uses the POST, GET, PUT and DELETE methods to perform CRUD (Create-Read-Update-Delete) operations, that is, create the resource if it does not exist, obtain the data, update the resource or eliminate it. All resources must be uniquely identified by a URL.

To build REST services you can build a simple HTTP server or use specific libraries available for almost all languages. Below, there is a simple example using the Spark library for Java, taken from the official website http://code.google.com/p/spark-java/ (however, you are free to use other libraries for REST services, like Jersey). The example is a service to store and retrieve information from books.

```
import static spark.Spark.*;
import java.util.*;
import spark.Request;
import spark.Response;
import spark.Route;


// A simple RESTful example showing howto create, get, update and delete book resources.
public class Books {
    // Map holding the books
    private static Map<String, Book> books = new HashMap<String, Book>();

    public static void main(String[] args) {
        // Creates a new book resource, will return the ID to the created resource
        // author and title are sent as query parameters e.g. /books?author=Foo&title=Bar
        post(new Route("/books") {
            Random random = new Random();
            @Override
            public Object handle(Request request, Response response) {
                String author = request.queryParams("author");
                String title = request.queryParams("title");
                Book book = new Book(author, title);

                int id = random.nextInt(Integer.MAX_VALUE);
                books.put(String.valueOf(id), book);

                response.status(201); // 201 Created
                return id;
            }
        });

        // Gets the book resource for the provided id
        get(new Route("/books/:id") {
            @Override
            public Object handle(Request request, Response response) {
                Book book = books.get(request.params(":id"));
                if (book != null) {
                    return "Title: " + book.getTitle() + ", Author: " + book.getAuthor();
                } else {
                    response.status(404); // 404 Not found
                    return "Book not found";
                }
            }
        });

        // Updates the book resource for the provided id with new information
        // author and title are sent as query parameters e.g. /books/<id>?author=Foo&title=Bar
        put(new Route("/books/:id") {
            @Override
```

```
            public Object handle(Request request, Response response) {
                String id = request.params(":id");
                Book book = books.get(id);
                if (book != null) {
                    String newAuthor = request.queryParams("author");
                    String newTitle = request.queryParams("title");
                    if (newAuthor != null) {
                        book.setAuthor(newAuthor);
                    }
                    if (newTitle != null) {
                        book.setTitle(newTitle);
                    }
                    return "Book with id '" + id + "' updated";
                } else {
                    response.status(404); // 404 Not found
                    return "Book not found";
                }
            }
        });

        // Deletes the book resource for the provided id
        delete(new Route("/books/:id") {
            @Override
            public Object handle(Request request, Response response) {
                String id = request.params(":id");
                Book book = books.remove(id);
                if (book != null) {
                    return "Book with id '" + id + "' deleted";
                } else {
                    response.status(404); // 404 Not found
                    return "Book not found";
                }
            }
        });

        // Gets all available book resources (id's)
        get(new Route("/books") {
            @Override
            public Object handle(Request request, Response response) {
                String ids = "";
                for (String id : books.keySet())
                   ids += id + " ";
                return ids;
            }
        });
    }
}
```

As you can see in the example, the elements of a REST service are:

- **Routes**. They identify resources, such as books in this case. Routes are made of segments. For example, "/books" has one segment and refers to the resource for books; while "/books/:id" has two segments, and the second one is a parameter whose value will be given when writing the URL to refer to a specific book.
- **Type of requests**. They correspond to the HTTP methods GET, PUT, POST and DELETE. The same route will behave differently depending on the method applied (e.g., GET vs PUT). The data can be sent from the client to the server in two ways: (i) as parameters in the URL, or (ii) as "attachments", part of a PUT or POST request, e.g., in XML or JSON (a string with the formatted data).
- **Answers**. Answers have a numeric code that indicates the result of the processing (see http://en.wikipedia.org/wiki/List_of_HTTP_status_codes). It also returns a string formatted in some way, such as an HTML page, although REST services usually return the data in JSON or XML format.
- **Data description**. Sometimes, requests and/or responses return a certain type of data (e.g., a book or a list of books). In such cases, the programmer needs to write code to serialize or deserialize JSON or XML from the in-memory data described by a Java class.

- **Advanced services**. There are more advanced services which include mechanisms for authentication, sending attachments (MultiPart), etc.

Requirements

Build a textual DSL to describe REST services taking into account the issues discussed (except the advanced services, which are optional). The language should allow representing requests, routes and the data that can be used as a request/response. Define suitable OCL constraints or validators if they are necessary to complete the static semantics of the meta-model.

Build a code generator that synthesizes a complete REST service from a program written with the DSL. We recommend that the generated code be supported by an existing framework such as Spark or Jersey. You must also generate code for the data model (de-)serialization to XML or JSON.

Create a model-to-model transformation of the data model into an UML class diagram or an Ecore meta-model, to obtain a design-view of the data used by the service.

3. An agent-based simulation language

The goal of this project is to facilitate the definition of multi-agent systems and automate their simulation. In this type of systems, there are multiple “intelligent” agents that can interact with each other and move through a territory. You can see examples of multi-agent systems here: http://agentbase.org/.

In this project, you have to develop a DSL to configure a multi-agent system with the characteristics shown below. You must also implement a simulator that executes the agent system created with the DSL.

Requirements

A multi-agent system contains various agents that interact with each other and move through a territory. The territory is a matrix of configurable size, where each coordinate may contain several elements. Agents have the ability to move to an adjacent coordinate within the territory, query which elements are in their environment, modify the elements of their environment, and interact with other agents. There may be agents of various kinds, each with particular properties (0 or more), and with a specific behavior defined by rules (e.g., when it can move, or when it can modify its properties or the environment). There may also be rules of evolution of the territory, which modify the elements that the territory contains.

As an example, we might want to define a multi-agent system to simulate ant colonies. The system would have two types of agent: worker-ant and queen-ant. Worker-ants have the property "load" to indicate the amount of food they carry, while queen-ants have the property "food" to indicate how much food they have eaten. The behavior of worker-ants is defined by four rules: (1) if their current position contains food and their load is less than 3, they take a unit of food to add it to their load; (2) if their current position has no food and their load is less than 3, they move to any adjacent position; (3) if their load is 3 and there is a queen-ant in their current position, they deliver the load to the queen, who increases her property "food" by 3 units, while the load of the worker-ant becomes 0; (4) if their load is equal to 3 and there is no queen-ant, they move to the adjacent position that has more pheromones, and increase that pheromone amount by 1. On the other hand, the behavior of queen-ants is defined by two rules: (1) if they have between 1 and 50 food units, they subtract 1 food unit, and introduce a new worker-ant into the system; (2) if they have less than 0 food units, they die and disappear from the system. The system would also include a rule of evolution of the environment, responsible for decreasing the pheromones of each coordinate of the territory if its value is greater than 0.

The DSL to be developed must support to definition of:

- The types of agents in the system (e.g., worker-ant and queen-ant).
- The properties of each type of agent (e.g., "load" in the case of worker-ants). For simplicity, you may assume that all properties are integers.
- The size of the territory (e.g., 20 x 20).
- The properties of the coordinates of the territory (e.g., food units, number of pheromones). For simplicity, you may assume that all properties are integers.
- Behavior rules of each type of agent (e.g., if a queen has zero food units, it dies).
- Evolution rules of the territory (e.g., decrease by 1 the number of pheromones of a cell).
- The initial distribution of elements and agents in the territory (e.g., 10 units of food in the coordinate (0,0) of the territory, 1 worker-ant with load 0 in the coordinate (10,0)).

In addition to the DSL, you have to develop a simulator. This will receive as input a multi-agent system and a number of simulation steps. Each simulation step consists in the successful application of a behavior rule for a randomly chosen agent, followed by the application of all evolution rules in every position of the territory. The simulation ends when the indicated number of simulation steps has been executed, or when no rule can be executed for any agent.

For this project, we suggest two possible solutions:

1) A textual DSL + a generic simulation engine written in a programming language + a code generator to configure the simulator.
2) A textual DSL + a model transformation to generate an Ecore meta-model for the agent system + a model transformation to generate Henshin graph transformation rules (typed by the Ecore meta-model) that implement the simulator.

4. Automating the creation and test of form-based web applications

Selenium is a free (open-source) automated testing suite for web applications across different browsers and platforms (https://www.seleniumhq.org/). That is, Selenium enables developers to emulate user interaction with a web page. Selenium WebDriver API is used to develop test scripts to interact with page and application elements. It works with many browsers (Firefox, Chrome, Safari, Explorer, etc.), programming languages (java, C#, JavaScript, PHP, Python, etc.) and testing frameworks. There are seven basic steps when creating a Selenium test script, which apply to any test case and application under test:

1. Create a WebDriver instance.
2. Navigate to a Web page.
3. Locate an HTML element on the Web page.
4. Perform an action on an HTML element.
5. Anticipate the browser response to the action.
6. Run tests and record test results using a test framework.
7. Conclude the test.

The example below shows the process to open Google page with Chrome, and check that the title of this page is “Google”.

```
import java.util.concurrent.TimeUnit;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

public class MyMain {
	private WebDriver driver;

	public static void main (String[] args) {

		System.setProperty("webdriver.chrome.driver","C:\\Drivers\\chromedriver.exe");
		ChromeDriver driver = new ChromeDriver();

		String baseUrl = "https://www.google.com/";
		String expectedTitle = "Google";
		String actualTitle = "";

		driver.get(baseUrl);
		driver.manage().window().maximize();
		driver.manage().timeouts().pageLoadTimeout(5, TimeUnit.SECONDS);

		actualTitle = driver.getTitle();

		if (actualTitle.contentEquals(expectedTitle)){
			System.out.println("Test Passed!");
		}
		else {
			System.out.println("Test Failed " + actualTitle.toString());
		}

		driver.quit();
	}
}
```

Description

The objective of this project is to automate the creation of HTML forms, as well as their corresponding user interface tests. HTML forms can contain text input boxes, combo boxes, buttons, radio buttons, and multiple choice fields (checklists). Text input boxes can be either mandatory or optional. All forms must have a set of buttons to check input data, save data in a file, cancel the forms, and other non-predefined actions (i.e., they will be manually implemented after generating the form).

The visibility and enabling of any of the elements of a form can depend on whether certain fields of type check or radio are selected or not. For example, you could have a form where the text fields "card number" and "holder" will not be visible when you select the choice field "cash payment" and will be visible once you deselect it. A form could have fields initially invisible or disabled, provided that it is guaranteed that there is a selection of fields that allows them to be enabled and made visible.

Finally, a simple mechanism must be provided to be able to define the placement of the elements in the form. For example, a CSS grid layout could be used, so that it could be indicated in which cell of the grid each element is located.

Next to the definition of a form, it should be possible to specify test cases of its user interface. Each test case would define a set of actions to be carried out on the elements of the form (e.g., pressing a button, entering a value in a text field, selecting an entry in a drop-down, etc.), and assertions about the status of the elements of the form (e.g., checking if an element is enabled, disabled, visible, invisible, selected, deselected, has some value, etc.). For example, a test case could consist of the following steps: selecting the check type field "cash payment", assert that the text fields "card number" and "owner" are no longer visible, deselecting the check type field "payment in cash", and assert that the fields "card number" and "owner" are visible again. In addition, it should be possible to activate or deactivate the verification of the following coverage criteria for the tests of a form: (a) all the fields of the form appear in some assertion of a test case, (b) some action is performed on all the fields of the form in some test case.

Requirements

In this project, you will have to develop a textual DSL to describe HTML forms and their corresponding user interface tests using Selenium. Then, you will have to build a code generator that generates HTML forms from the DSL, as well as the tests written as JUnit unit tests with Selenium WebDriver to check the form functionality.

5. A DSL for creating Q&A active games on Twitter

In this project, the goal is to design a DSL that facilitates the creation of Twitter bots for playing question/answer active games through the Twitter (http://twitter.com) social network. The project should provide: (a) a textual DSL to configure the games; (b) support for analyzing the game designs based on Petri nets; and (c) execution support for the games, based on APIs like Twitter4J (http://twitter4j.org/en/).

Requirements

The games we target will consist of tests that are to be solved across different spots within a city, within a given time limit. Each test can be configured with a place where the answer needs to be given, a time limit, the text of the question, a set of possible answers, a number of attempts, a set of hints (to be provided by the bot when the answer is wrong), and a number of points to be awarded if the answer is correct. In addition, the designer of the game needs to specify the next tests when the answer is correct, when the answer is wrong, and when no answer is provided within the time limits. Each game should have exactly one initial test and one or more final tests.

The bot should wait until a Twitter user follows the bot's account. Then, the bot gives a welcome message (as a direct message) with the playing instructions. Then, the bot should send information about the initial test, including the location, maximum time and allowed number of attempts. A test is successfully fulfilled by a user if: (i) the answer is correct; (ii) the position of the user when the tweet is sent is within the coordinates of the location; and (iii) the answer was given within the time limit. The tweet text, timestamp and position can be extracted using Twitter4J.

In addition, to validate the design of the game, a transformation into (timed) Petri nets should be provided (where users are represented by tokens and tests by places). Using such Petri net, the game designer will be able to check reachability of final tests, calculate mean completion times of the game, etc.

As an example, a possible syntax for the DSL could be as follows:

```
location Sol "Puerta del Sol" [40.4167278, -3.7033387]  // we could also give a maximum deviation
location PlazaMayor "Plaza Mayor" [40.415363, -3.707398]

welcome: "Welcome to the game 'El Madrid de los Austrias' with quizzes about the history of Madrid"

initial Test t1 at Sol with duration 20 {
        Question: "Who is represented in the equestrian statue?"
        Answers: ["Carlos III", "Charles the third"]
        Attempts: 2
        Hints:  "A king whose nickname was `The best Mayor of Madrid'"
        Points: 10
} on correct: t2, on fail: t3, on timeout: lose

Test t2 at PlazaMayor with duration 20 { // one attempt by default, and hence no hints needed
        Question: "When was the square renovated?"
        Answers: ["1961", "61"]
        Points: 20
} on correct: t4, on fail: t5, on timeout: t2
```

6. A graphical DSL to define and analyse manufacture systems

The goal of this project is to develop a graphical DSL to define manufacture systems modelling the production and assembly of products made of different kinds of parts. The project should provide: (a) a graphical DSL developed with the Sirius framework to configure the manufacture systems; (b) a simulator of manufacture systems; and (c) support for analyzing the manufacture systems based on timed Petri nets.

Requirements

A manufacture system can include three kinds of machines: part generators, assemblers and packers. These are connected through conveyors which have a maximum capacity indicating how many parts and products they can contain at the same time. Conveyors can also be interconnected to create larger conveyor chains. Part generators produce a certain number of parts (e.g., 4) of a certain kind (e.g., sheet or screw) every a certain amount of time (e.g., every 4 seconds). All these properties are customizable for each generator. Assemblers assemble parts or other products, to build new products. This way, each assembler needs to indicate how many parts or products of each kind are needed to build the new product (e.g., 2 sheets and 6 screws) and the kind of created product (e.g., a hinge). The assembly of a product takes a certain number of seconds. Finally, packers create packages of products. Each packer needs to specify the number of products per package (at least one), and the time required to create a package (e.g., 10 seconds).

The DSL should define suitable OCL constraints or validators to complete the static semantics of the meta-model. Examples of validations include checking that there are generators of every kind of part used by the assemblers.

In addition, to validate the design of a manufacture system, the following two artefacts should be provided:

- a graph grammar developed with Henshin that simulates the manufacture systems (i.e., generation of parts, assembly of parts and products, packaging of products, and transport of parts and pieces by conveyors).
- a transformation of manufacture systems into timed Petri nets. Using the generated Petri net, the designer will be able to check overflows, mean time of creation of products, etc.

7. Open project

Find a problem within some domain of your interest, and apply MDE techniques. Typically, we expect the project to include the design of a DSL, a model transformation and a code generator.

If you choose this project, it is mandatory that you first agree on the project scope and requirements with some of the professors of the course.