Final Exam:
The RoadFighter2 Android Car Racing Game

Néstor Cataño **Due: November 21, 2016**

General Description

The purpose of this final exam is to re-implement RoadFighter2 using various of the formal methods techniques learnt during our course. RoadFighter2 is a new Android based implementation of the traditional RoadFighter car racing game. RoadFighter2 implementation follows the standard MVC (Model-View-Controller) design pattern. RoadFighter2's VC part is based on OpenGL, and its M's part has initially been modelled in Event-B and then code-generated to Java using the EventB2Java formal methods tool. RoadFighter2's current implementation has several problems that you will need to fix. $i.$) The current implementation of the interface only works on an old version of Android, so you will need to port the interface to the last version of the Android platform available; $ii.$) RoadFighter2's interface is not usable and it only implements an archaic user-game interaction that relies on the use of the keyboard; and, $iii.$) the M part of RoadFighter2's implementation doesn't cater for safety properties modelled in Event-B.
You are advised to use Android Studio (`https://developer.android.com/studio/index.html`) to debug RoadFighter2 car racing game and to provide a new implementation of its missing functionality. You will demonstrate your final implementation of the game on a Sony S tablet that I will make available to you and to all the other teams. You will need to install the last version of Android on the tablet.
We will be working together in this final exam. You're encouraged to approach me with questions on the exam, and ask me all the kinds of questions that are relevant for your work. My goal is not only to guide you in this process, but to learn from you and your experiences in fixing and evolving RoadFighter2.

The Car Racing Game

The Android platform was introduced by Google in 2008 as an operating system for mobile devices. It supports interfacing with common hardware found in embedded devices as well as more general-purpose programming libraries for threads and networking. The Android SDK has a wide support for programming and includes extensive examples and documentation. It supports technologies commonly found in video game development, *e.g.* 3D rendering pipelines (through OpenGL), raster graphics, and input device interaction (keyboard and touch screen). When developing large-scale projects, an IDE is recommended for features as code refactoring, SVN support, repositories, automatic compilation, etc. Android is supported by Eclipse and Android Studio.
Android programs have to adhere to a special program structure for interactive applications like our car racing game. Hence, any program in Android that requests a visual interface must create a main *Activity*. This activity has access to a variety of widgets and visual structures. Android's graphical nature makes the MVC design pattern widely used among its apps. It includes three big modules in a software project, they are; the *Model*, which specifies the internal and logical behaviour of the app, the *View*, which implements its visual aspect, and the *Controller*, which synchronises and communicates the *View* and the *Model*.
The car racing game consists of a single *car* controlled by the user, the car is placed on a *track* that has a *finish line* and *borders*. Cars run over lanes (tracks). Several static and dynamic objects called *obstacles* as well as other artificially controlled cars called *opponents* are placed on the *track*. The goal of the car racing game is to reach the finish line in the least amount of time possible, hence collisions with the opponents and track borders must be avoided. The simulation of the physics involved in driving a car can be as accurate as the developers want and the hardware allows. A score system is modelled to reward players. As games are meant to attract people, variations of the scoring system and rules can be implemented in order to achieve market differentiation.

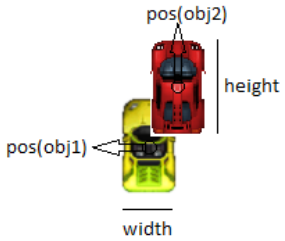

Figure 1: 2D box collision detection scheme

Two important aspects of physics involved in artificially simulating a car race are kinetic movement and collision detection. A straight transformation of physics equations of accelerated movement into algorithms (implemented in Java for instance) is not possible because of their continuous nature. Game programmers often opt to apply a discrete approach based on Euler Discretisation[1]. The following equations model an uniformly accelerated movement. You may see that these equations are recursive, and require an initial value, similar to a state machine. Δ_t is the time step-value of the discrete system.

$$\begin{aligned} acc(obj) &= a \\ vel(obj) &= vel(obj) + acc(obj) \times \Delta_t \\ pos(obj) &= pos(obj) + vel(obj) \times \Delta_t \end{aligned}$$

A great diversity of algorithms exist to calculate collisions, which may use boxes, spheres or ellipses as their geometrical enclosing form to calculate intersection of objects. Because exhaustive collision detection is time demanding, the trade-off thus is speed of processing vs. exactitude. The car racing game implements a two-dimensional box collision detection algorithm (see Figure 1). Let $Obj1$ and $Obj2$ be two objects inside the game's world, we wish to know if they are actually colliding and take action upon this information. Because we are dealing with two dimensional figures, the position of an object can be described as $pos_x(obj)$ and $pos_y(obj)$. The following logical proposition holds when a collision between obj_1 and obj_2 occurs.

$$\begin{aligned} |pos_x(obj_1) - pos_x(obj_2)| &< \frac{width(obj_1)}{2} + \frac{width(obj_2)}{2} \quad \wedge \\ |pos_y(obj_1) - pos_y(obj_2)| &< \frac{height(obj_1)}{2} + \frac{height(obj_2)}{2} \end{aligned}$$

We enclose each object into a 2D box of width and height dimensions and then test if there is intersection between the two. We use boxes in our car racing game, but circles or ellipses might be the best choices for other scenarios with objects such as balls (Billiard game) or humans.

As it's often the case in professional games to request hardware video acceleration, Android provides an OpenGL abstraction layer (either version 1.0 or 2.0) to interact with the video hardware. The activity created inside the application thus must posses an extended `GLSurfaceView` to interact with the user and offer video acceleration. This part of the structure serves as the Controller module in the MVC design pattern. The View module is implemented inside as `GLSurfaceView.renderer`, a part of `GLSurfaceView`, all visual feedback to the user is provided here. The Model lies outside this structure as it is implemented independently, however, it is used here.

[1]E. Lengyel. Mathematics for 3D Game Programming and Computer Graphics. Course Technology PTR, 2011

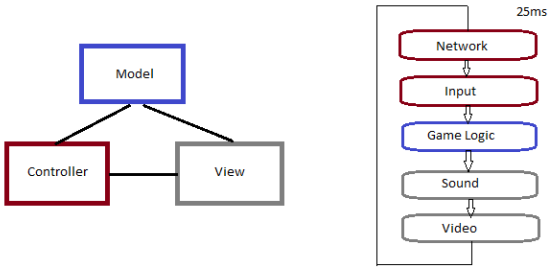

Figure 2: MVC (left) and Game Loop (right). Both indicate the flow of information and behaviour, same purpose blocks are drawn in the same colour.

Syntax	**Name**	**Definition**	**Short Form**
$q;r$	forward composition	$\{(x,z) \mid \exists y \cdot (x,y) \in q \land (y,z) \in r\}$	$q;r$
$\mathsf{id}(s)$	identity relation	$\{(x,y) \mid (x,y) \in s \times s \land x = y\}$	$\mathsf{id}(s)$
$s \lhd r$	domain restriction	$\{(x,y) \mid (x,y) \in r \land x \in s\}$	$\mathsf{id}(s);r$
$s \lhd\!\!\!- r$	domain subtraction	$\{(x,y) \mid (x,y) \in r \land x \notin s\}$	$(\mathsf{dom}(r) - s) \lhd r$
$r \rhd s$	range restriction	$\{(x,y) \mid (x,y) \in r \land y \in s\}$	$\mathsf{id}(s);r$
$r \rhd\!\!\!- s$	range subtraction	$\{(x,y) \mid (x,y) \in r \land y \notin s\}$	$r \rhd (\mathsf{ran}(r) - s)$
$r[s]$	relational image	$\{y \mid (x,y) \in r \land x \in s\}$	$\mathsf{ran}(s \lhd r)$
$r^{\sim}$	inverse relation	$\{(x,y) \mid (y,x) \in r\}$	$r^{\sim}$

Figure 3: Basic Event-B mathematical notation.

Game design employees use a similar MVC-alike architecture based on a concept called the *game loop*. Both architectures are based on *real time interactivity* (see Figure 2). Therefore, the Model part in MVC is analogous to the *logic* part in game loop. This is the critical part defining how the information is processed both from and to the users. The functional requirements are largely implemented here therefore this work explains how to generate *logic* code that is both correct and consistent with the software requirements.

The Event-B Notation

The *Model* part of the car racing game has been first modelled in the Event-B formal language, and then code-generated to Java using the EventB2Java tool. We give here a short introduction to Event-B. Event-B is a formalism similar to Z. Event-B provides a complete battery of set and relation operators (see Figure 3). We use square brackets to apply a relation to a set of elements. For instance, $r[s]$ applies relation r to all the elements in set s. The result of $r[s]$ is a set of elements in the range of r. Event-B provides standard notations for set union, intersection, difference, etc. Symbol $\times$ denotes the cross product between two sets. The operator dom returns the domain of a relation, and ran its range. The operator id denotes the identity relation over a set of elements s. Applying the forward composition relation $q;r$ to an element x in the domain of relation q returns a set of elements calculated as the result of applying r to $q[\{x\}]$. In the case that q is a function, then $q[\{x\}]$ becomes $q(x)$. The domain restriction relation $\lhd$ restricts the domain of a relation r to consider only elements in a subset s of its domain. The range restriction relation $\rhd$ restricts the range of a relation r to consider only elements in a subset s of its range. Domain (range) subtraction is defined similarly to domain (range) restriction, except that the elements in the set s are disregarded rather than considered. Event-B also offers the $^{\sim}$ notation for the inverse of a relation.

How Does Event-B Compare to Z?

Event-B modeling language is based on predicate logic and set theory. Although Event-B shares essentially the same modeling language for stating state properties as Z, Event-B and Z offer different modeling mechanisms that are specialized in distinct mathematical aspects. Event-B and Z are both models for state transition systems. Event-B's language for expressing the dynamic behaviour of state machines is based on events. On the other hand, Z uses a rich schema calculus mechanism for expressing the dynamic behaviour of models. Z Schema calculus and Event-B events coupled with model refinement are different mechanisms. Z also offers refinement, but in practice, Z focuses more on formal specification and Event-B focuses more on model refinement and coding (whereas manual or tool generated, e.g. with EventB2Java).

Another major difference between Event-B and Z is about the undertaking and use of invariants. In Event-B, each event definition produces proof obligations that attest to the correctness of machine invariants. These invariants might encode safety properties. On the other hand, in Z, invariants are incorporated to the model definitions, altering their meanings. They do not generate proof obligations. Other differences between Event-B and Z are about the notation used. Event-B does not have a explicit notation for variable post-state as Z does. Z uses primed variable notation to denote the post-state of a variable. In Event-B, the use of a variable on the right hand-side of an assignment denotes the value of the variable in the pre-state of the event where the assignment is declared, and its use on the left hand-side denotes its value in the post-state of the execution of the event. Z uses a convention whereby the name of a schema parameter ends by a question mark symbol. In Event-B, parameters of an event are declared within an any symbol. Event preconditions can implicitly be encoded with the aid of event guards, schema preconditions are encoded with schema invariants that might make use of schema parameters.

Rodin

Rodin is an Eclipse based platform that provides support to Event-B, e.g. writing Event-B models, defining safety invariant properties, and discharging proof obligations using existing decision procedures. The last version of Rodin is available at `https://sourceforge.net/projects/rodin-b-sharp/`. The standard way of installing components in Rodin is through the menu *Help*, *Install New Software ...*, *Add ...*, and then adding an "Update Site". After installing Rodin, you must install install $i.$) the Atelier B Provers (the Update Site is `http://methode-b.com/update_site/atelierb_provers`), $ii.$) the ProB model checker (the Update Site is `http://www.stups.hhu.de/prob_updates_rodin3`), and $iii.$) the EventB2Java Java code generator (the Update Site is `http://poporo.uma.pt/Projects/EventB2Java_3_1_plugin_update`). Optionally, you might want to install Rodin Handbook (the Update Site is `http://handbook.event-b.org/updatesite`). I strongy recomend to use Camille to write and edit Event-B machines. The Update Site is `http://www.stups.hhu.de/camille_updates`.

Figure 4 presents a snapshot of Rodin displaying the `ref0_circuit` machine of the car racing game. This machine *refines* machine `ctx_0` and declares variables such as `objects`, `obstacles`, and `cars`. It also declares some invariants which, in this case, are mainly used to type the machine variables. The machine also declares some *events* (not shown there) that define the dynamic part of the machine. The bottom right part of the figure shows the *Symbol* tool-box with different mathematical Event-B symbols.

Rodin and all Eclipse based editors rely on the concept of *perspective*. The upper right part of the Figure shows that the Event-B machine's code is presented in *Event-B* perspective. You can either manually switch to a different perspective (*Window*, *Open Perspective*) or pursue an action that makes the editor switch to a default perspective. If you double-click a proof obligation (shown on the left panel), then the proof obligation is open in the *Proving* perspective of Rodin (see Figure 5). A green face means that the underlying proof obligation (PO) has successfully been discharged (proven). Undischarged POs are shown in a red grumpy face.

Rodin relies on several decision procedures that help the user discharge proof obligations automatically. A particular decision procedure may not succeed, in which case you are obliged to use another decision procedure to discharge the PO, or you can attempt to discharge the PO manually. The figure shows `pp`, `nPP`, `p0`, `p1`, and `ml` decision procedures. You must consult Rodin Handbook on details how these decision procedures work and how to discharge POs in Rodin manually and automatically.

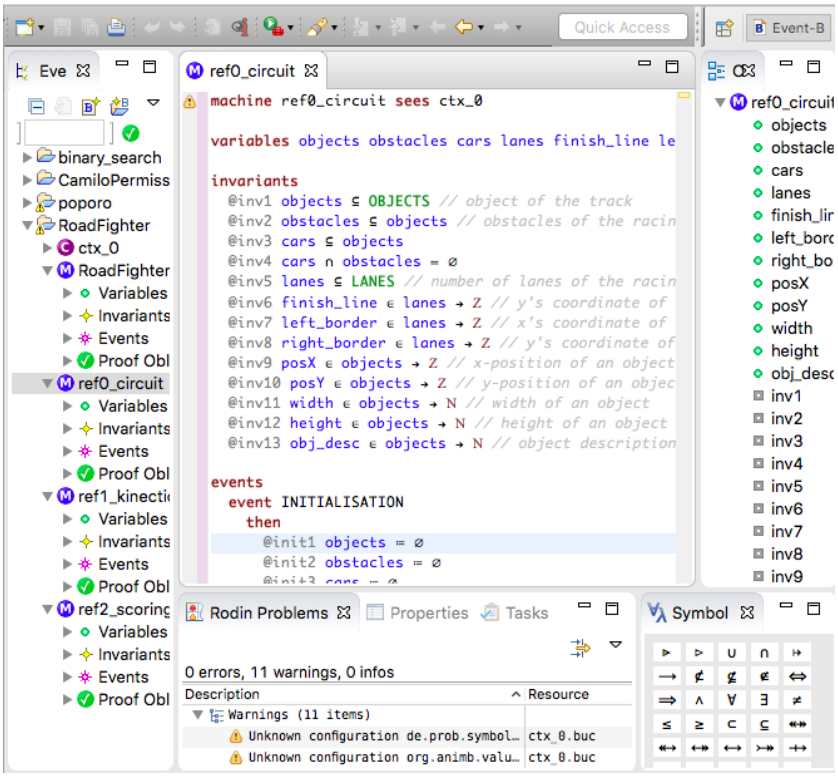

Figure 4: Rodin IDE

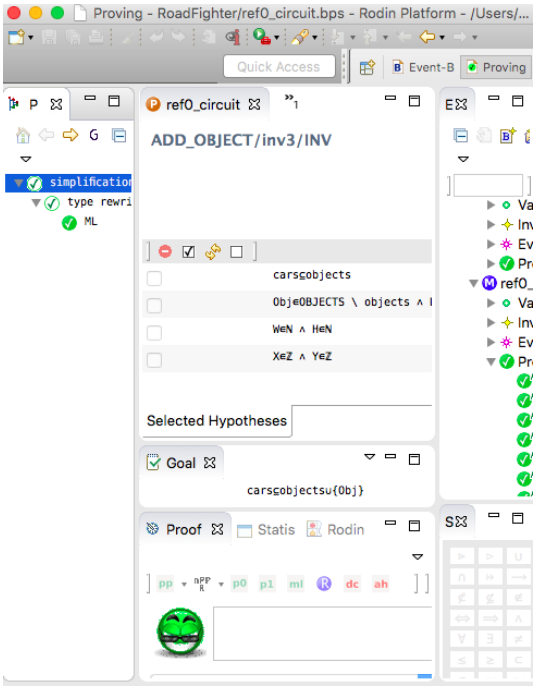

Figure 5: Rodin Proof Obligation Editor

EventB2Java

The EventB2Java tool is a Rodin Eclipse plug-in. To install and use EventB2Java you must install Rodin first. EventB2java is available at `http://poporo.uma.pt/EventB2Java/EventB2Java_download.html`. This web site includes detailed instructions on how to install and use the tool. The EventB2Java Eclipse plug-in's update site is `http://poporo.uma.pt/Projects/EventB2Java_3_1_plugin_update`, and EventB2Java has been tested on Rodin version 3.2. To install EventB2Java, select *Help*, *Install New Software ...* and add EventB2Java's Update Site. Code generation with EventB2Java works in *Event-B* perspective, but the generated code is available in the *Resource* perspective. In the Resource perspective, you must refresh your project to see the generated code. In the Event-B perspective, right-click the machine you want to generate Java code for (usually the last refinement machine), select *Java Code Gen*, and then select the type of implementation that you want to generate, either sequential or multi-threaded.

Task 0 (20 points)

The goal of this task is to run the existing version of RoadFighter2 in Android Studio.

- Download and Install Android Studio from `https://developer.android.com/studio/index.html`.
- If prompted, install all the updates of Android Studio, and install the `gradle` project manager system.
- Download RoadFighter2 from `http://poporo.uma.pt/modelsSs/roadfighter/RoadFighter2.zip`, unzip it into a folder of your preference, and open that project in Android Studio.
- Run RoadFighter2 in Android Studio. I have tested RoadFighter2 in two emulators, Nexus One API N, and Genymotion. Other virtual devices might be better options for running RoadFighter2.

- Document (write) all the technical problems that you found, your thoughts on the initial challenges you're faced, and if it's possible recommendations based on your first impressions of RoadFighter2.

The final part of this document includes a list of useful links and hints that will help you install and run RoadFighter2 in Android Studio, but be aware of that you're expected to conduct independent studies and learning on Android.

Task 1 (20 points)

The goal of this task is to conduct a *sanity check* of the Event-B model of RoadFighter2 with the ProB model checker.

- Install Rodin following previous instructions.
- Download the Event-B model of the car racing game from `http://poporo.uma.pt/modelsSS/roadfighter/RoadFighterEB.zip`. In the next tasks, you will be extending this model, but here in this task, you will model-check the Event-B model of the car racing game for *deadlocks* using the ProB model checking tool.
- Install the ProB model checker. ProB is a Rodin plug-in, so it installs the usual way through the *Help*, and *Install New Software* options of Rodin. ProB Update Site is `http://www.stups.hhu.de/prob_updates_rodin3`.
- Run ProB automatically and step-by-step to check RoadFighter2 for deadlock conditions. Deadlocks arise when an execution cannot make progress because none of the *event guards* hold. If you find a deadlock, then modify the Event-B model, e.g. change some of the event guard conditions to avoid the deadlock. However, any modification needs to make sense in terms of the car racing game. Document all your conclusions, and changes to the model regardless of whether you find deadlocks or not. Your conclusions are valuable to me.
- Make sure that every single event of the machine are reachable. If you find an unreachable event, then you will need to modify the Event-B model of RoadFighter2 to make the event reachable. Document your decisions and conclusions.
- Any time that you modify your model, you will need to discharge all the generated POs. But, most of the time you will just need to execute again the existing proofs to get a green face.

Notice that because you will extend and modify the Event-B model of the car racing, you will need to go back gain to this task and to check that your extended model doesn't deadlock and that all the machine events are reachable.

Task 2 (20 points)

The goal of this task is to add a basic safey property to the Event-B model of RoadFighter2.

- Add an invariant property to the RoadFighter2 that states that "no two different objects of the game can occupy the same position at the same time". You must choose the right machine to add this invariant safety property, however, be aware that adding an invariant to a machine not only generates POs for that machine itself but also for all the refinement machines.
- Discharge all the POs obligations generated in the previous step and for all the machines.
- Don't forget to document this task and describe all the decisions you made and the problems you faced.

You will use the EventB2Java tool to generate Java code for your extended model of RoadFighter2 in one of the final tasks of this exam, however, you need to discharge all the POs first, otherwise the EventB2Java tool cannot assure that the generated code adheres to the safety property you have incorporated to the Event-B model.

Task 3 (20 points)

The goal of this task is to *animate* (run) the Event-B model of RoadFighter2. The main advantage of implementing an application using the MVC design pattern is that you can evolve the Model part of your application independently from its View.

- Use the EventB2Java tool to generate code for the extended Event-B model of RoadFighter2 that you wrote in the previous task. Check previous sections of this document on instructions on how to install and use EventB2Java.
- The EventB2Java tool generates an Eclipse project that you can import and run from Eclipse or run from the command line. The tool generates a `TestXXX` Java class that you will edit and use to animate the Java Model of RoadFighter. The first time that you import the Eclipse project, you will see that the `TestXXX` class doesn't compile. Manually correct any errors on duplicate local variable declarations (if any): those are bugs of the EventB2Java tool.
- EventB2Java generates a *prelude* library that includes a Java implementation for sets (class `BSet`), relations (class `BRelation`), and operations over them. Check the following paper for documentation about these classes `http://poporo.uma.pt/docs/eventb-codegen.pdf` and the EventB2Java code generator.
- Edit the `main` method of the `TestXXX` class and write code that animates (uses, runs) the Java code generated by EventB2Java. Animation works in a similar way as testing. The goal of animation is to check if the model that you wrote in Event-B (and that's generated to Java) is the actual model that you had in mind. Write animation code between the comments */** User defined code that reflects axioms and theorems: Begin ... End **/*. Your animation code must give different values for all the machine (class) variables including `radom_USER_LANE`, and `random_USER_CAR`. Those values exercise classes `BSet` and `BRelation`.
- Consult examples of how to animate Java code generated by EventB2Java from the link `http://poporo.uma.pt/docs/tokeneer-tests.zip`. Check the `main` method of the `Test_Ref6_admin` class.
- If after animating your Java code you notice any inadequate behaviour in RoadFighter2, go back to the previous task, modify the Event-B model of the game, and generate code again with EventB2Java.
- Document (write in your report) any decision that you made, any changes you made to the Event-B model, any conclusions that you made about this task.

I currently use Eclipse Luna for running the code generated by EventB2Java, but you can decide to use a different yet compatible version of Eclipse. To import a project in Eclipse, select *File*, *Import*, *General*, *Existing Projects into Workspace*, *Select root directory*, and follow the instructions.

Task 4 (20 points)

The goal of this task is to fix the implementation of the interface of RoadFighter2. At the current moment I am not aware of all the challenges involved in this task, so you will need to find them out by yourselves.

- The current implementation of the interface works on an old version of Android, so you will need to port RoadFighter2 to the last version of Android available out there.
- The usability of your implementation of the interface of RoadFighter2 will be assessed on a Sony S Tablet. Make sure you test RoadFighter2 on that tablet and you install on it the last version of the Android platform.
- Usability aspects of your implementation of RoadFighter2 must include the ability to interact with the game using the touch screen.
- Add the Java code generated in tasks 2, 3 to the final version of RoadFighter2 before you demonstrate your final implementation of the game to me.

- Document all your decisions, challenges and conclusions for this task.

Additional conclusions on the whole final exam exercise are welcome and will help me improve exam for the sake of future students' benefit.

Useful Links and Hints

- Setting up the `PATH` environment variable
  edit `~/bash_profile` and add the following line to the `PATH`
  `PATH=~/Library/Android/sdk/tools:~/Library/Android/sdk/platform-tools:$PATH`
- Listing the exiting platforms in the SDK
  `tools/android list targets`
- creating an avd
  `android create avd -n android-22-armeabi-v7a -t 1 --abi android-tv/armeabi-v7a`
- Creating a Project (that can be compiled with Ant)
```
android create project \
--target 3 \
--name MyAndroidApp \
--path ./MyAndroidAppProject \
--activity MyAndroidAppActivity \
--package com.example.myandroid
```
- Running the SDK Manager
  `tools/android`
- Creating a debug APK version of the Project
  from within the Studio Project
  `chmod +x gradlew ./gradlew assembleDebug`
  this creates `app/build/outputs/apk/app-debug.apk`
- Installing the APK in an Android Device (using Android Debug Bridge - ADB)
  `adb -d install app/build/outputs/apk/app-debug.apk`
  or if an emulator is running
  `adb install app/build/outputs/apk/app-debug.apk`
- Running the APK on the started emulator
  `adb install app/build/outputs/apk/app-debug.apk`