Formal Software Development of Android Games

Salim Perchy
Pontificia Universidad Javeriana
Cali, Colombia
ysperchy@javerianacali.edu.co

Néstor Cataño
The University of Madeira
Carnegie Mellon Portugal
ncatano@uma.pt

ABSTRACT

We present a formal software development methodology for implementing Android video games. We use formal methods embedded within a Model-View-Controller (MVC) design pattern for software development. The Model of the game is written in Event-B, which is then translated to JML (Java Modeling Language) by using the EventB2Jml tool. A Java code implementation is manually written that closely follows the JML specification. Writing a formal model in Event-B avoids the costly testing phase in traditional incremental software development methodologies for games. We validate our ideas on the proposed methodology by presenting the software development of a car racing game for the Android platform. Our methodology incorporates rigour in a number of ways. All the proof-obligations of the Event-B model are discharged prior to the generation of JML specifications, and we use OpenJML to verify the Java implementation against the JML specification obtained from Event-B.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*methodologies, tools*

; D.2.4 [**Software/Program Verification**]: [validation, formal methods]

General Terms

automated translation, combined formal methods

Keywords

Android, Event-B, Formal Methods, Game Design, Game Development, Java, JML, Software Engineering

1. INTRODUCTION

Software engineering methods provide a disciplined approach to software development, yet software is often flawed, and even small software development projects have an inherently high level of complexity that requires the coding of mathematical well-founded algorithms. Software development projects are often composed of multi-disciplinary teams with diverse expertise in different tools and techniques. This is particularly true for teams developing video games, e.g. for the Android platform [12]. Video games usually follow incremental software development methodologies that include heavy-weight beta testing and debugging final phases for the release candidate [20]. This testing phase demands a high amount of human and monetary resources for teams to meet deadlines. On top of that, the functionality of the game should always be re-tested after the release.

We present a software development methodology for implementing Android games following the traditional Model--View-Controller design pattern. Therefore, the View is developed using *usability engineering* techniques as advocated by Jakob Nielsen in [18]. The code for the Model is semi-automatically generated. One writes the model in Event-B [3], then uses the EventB2Jml tool [6] to automatically generate a JML specification of the model, and finally writes Java code that verifies against the JML specification. The translation from Event-B to JML is automated and tool-supported. And, although going from JML to Java is manual and laborious, it is intuitive as only requires removing JML specification markers. The *Controller* includes wrapping code that communicates with the View and the Model. The relationship between the View and the Model is implemented by generating a Java field for every local event variable, as well as accessors and mutators for these fields.

Our methodology incorporates formal rigour in a number of ways: *i.*) all proof obligations of the Event-B model are discharged prior to the generation of JML specifications; *ii.*) the soundness of the translation from Event-B to JML has been proven [7] and *iii.*) OpenJML [9] is used to verify the Java implementation against the JML specification obtained from Event-B. Our methodology permits the use of different software validation and development techniques – Refinement Calculus in Event-B and Design-by- Contract as advocated by JML, and Usability Testing for the View.

The contributions of this paper are two-fold: (1) we propose a methodology that integrates formal methods to traditional usability engineering techniques for the development of games for the Android platform that (2) avoids the costly testing phase used in incremental software development of games. We validate our approach by presenting the software development of an Android race car game available at [1].

In the following, Section 2 introduces Android, Event-B and JML, and describes the Android game. Section 3 describes the proposed methodology. Section 4 presents

the Event-B model for the game. Section 5 describes the JML specification generated by the EventB2Jml tool on the Event-B input. Section 6 describes how Java code is written that verifies against JML, Section 8 presents related work, and Section 9 concludes.

2. PRELIMINARIES

2.1 Event-B

The B method for software development, introduced by J.-R Abrial [2], is a strategy for software development in which an abstract model of a system is transformed into an implementation via a series of *refinement* steps, where the behaviour of each refinement is provably consistent with the behaviour of the previous step. Each refinement adds more details to the description of the system. A derivative of the B method, also introduced by J.-R Abrial [3], is called Event-B. Event-B models are complete developments of discrete transition systems composed of machines and contexts. Machines contain the dynamic part of a model (variables, invariants, and events). Contexts contain the static part of a model (carrier sets and constants). Three basic relationships are used to structure a model, namely, a machine *sees* a context or *refines* another machine, and a context *extends* another context. The B language for stating properties, essentially predicate logic plus set theory, and the B language for specifying dynamic behaviour (*i.e.* programs) are seamlessly integrated.

2.2 The Java Modeling Language

JML [5] is a model-based language for specifying the behaviour of Java classes. JML specifications are typically embedded directly into Java class implementations using special comment markers `/*@ ... @*/` or `//@`. Specifications include various forms of invariants and pre- and postconditions for methods. JML syntax is intentionally similar to that of Java so that it is less intimidating for developers. In particular, the mathematical types that are heavily used in other model based specification languages (sets, sequences, relations and functions) are provided in JML as classes, and the operations on those types are specified (in JML) and implemented as Java methods. In JML, methods are specified using `requires`, `assignable` and `ensures` clauses, which respectively give the precondition, the frame (what locations may change from the pre- to the poststate) and the postcondition. A method specification can also include an `exsures` or `signals` clause to specify conditions under which the method could throw an exception. The specification of a method appears immediately before its declaration. Class invariants can also be given to constrain the states of legal class instances.

2.3 The Android Platform

The Android platform [12] 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 [16] 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. The Eclipse programming environment supports Android.

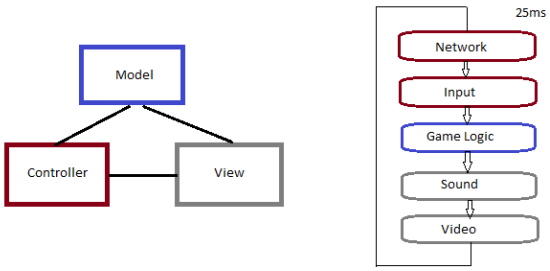

Figure 1: MVC (left) and Game Loop (right). Both indicate the flow of information and behaviour, same purpose blocks are drawn in the same colour.

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 synchronizes and communicates the *View* and the *Model*.

2.4 Game Development in Android

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.

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 1). 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 race car game model and the implementation presented in this paper are kept to a minimum for space and simplicity reasons. The full Android game application including aspects such as scoring, road friction, maximum velocity, track collision, dynamic objects management, textures, user input and acceleration is available at [1]. This game software was implemented using the Eclipse framework and developed for the Android platform version 2.3

using Java with OpenGL 1.0.

2.5 The Race Car Game

A race car game consists of a single *car* controlled by the user, the car is placed on a *track* that has a *finish line* and *borders*. 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 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.

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 Discretization [14]. The following equations model an uniformly accelerated movement. The reader may see that these equations are recursive, and require an initial value, similar to a state machine. The 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 [14].

We implement a two-dimensional box collision detection algorithm (see Figure 2). 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 race car game, but circles or ellipses might be the best choices for other scenarios with objects such as balls (Billiard game) or humans.

3. THE METHODOLOGY

We envision the use of formal methods within a typical Model-View-Controller (MVC) design pattern for developing software applications. This comprises an interface (the View) that interacts with the user, a functional core (the Model) that implements all the core functionality of the application, and a linking part (the Controller) handling all the user's requests as understood by the Model. The Controller includes wrapping code for communicating between the View and the Model. Event-B and the translation to JML are used in developing the Model. In producing Java code from the JML specification, a Java class field is generated for every local event variable as well as accessors and mutators for these fields. These fields are used in writing methods to check whether the guard of an event is satisfied before invoking (the translation of) the event. The proposed methodology comprises the following steps:

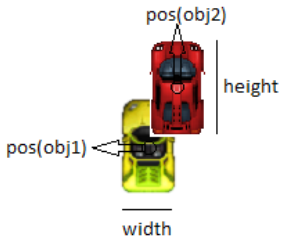

Figure 2: 2D box collision detection scheme

1. The developer chooses the desired level of abstraction for a formal description of the system in Event-B as a hierarchy of machine refinements.
2. All proof obligations of the above Event-B model are discharged in Rodin.
3. The Event-B model, provably correct, is automatically translated into JML by using the EventB2Jml tool.
4. Java code for the Model, complying with the JML specification, is produced.
5. The View is developed using *Usability Engineering* techniques [18]. In particular, every local event variable in the Event-B model is implemented as a Java field in the View, and accessors and mutators are included in the Java code. This enables communication between the Model and View implementations.
6. If a Java implementation is produced manually (rather than being automatically generated), a tool such as OpenJML [9] can be used to verify the Java code against the JML specification.

4. THE EVENT-B MODEL OF THE GAME

Figure 3 presents a simplified model of the car racing game in Event-B. Machine RacingGameMachine sees the RacingGameContext context (not shown here), which declares two carrier sets OBJECT and TRACK representing all the possible objects in the game and all the possible race tracks. Variables *objects* and *tracks* are the current objects of the game and the existing race tracks; *obstacles* is the set of static objects of the game, and *cars* is the set of dynamic objects. Invariants $obstacles \cup cars = objects$ and $obstacles \cap cars = \varnothing$ ensure that *obstacles* and *cars* make up a partition of

machine RacingGameMachine
sees RacingGameContext
variables $objects\ tracks\ obstacles\ cars\ width\ height$
$posX\ posY\ vel\ acc\ lean\ score\ collided\ active$
invariants
$objects \subseteq \text{OBJECT}$
$tracks \subseteq \text{TRACK}$
$obstacles \cup cars = objects \wedge obstacles \cap cars = \varnothing$
$posX \in objects \to \mathbb{Z}$
$posY \in objects \to \mathbb{Z}$
$width \in objects \to \mathbb{N}$
$height \in objects \to \mathbb{N}$
$vel \in cars \to \mathbb{Z}$
$acc \in cars \to \mathbb{Z}$
$lean \in cars \to \{-1, 0, 1\}$
$score \in cars \to \mathbb{Z}$
$collided \subseteq cars$
$active \subseteq obstacles \cup cars$
events
event *initialisation* then
$objects := \varnothing\ tracks := \varnothing\ obstacles := \varnothing\ cars := \varnothing$
$posX := \varnothing\ posY := \varnothing\ width := \varnothing\ height := \varnothing$
$vel := \varnothing\ acc := \varnothing\ lean := \varnothing\ collided := \varnothing\ active := \varnothing$
end
event *update_pos*
any $car\ elapsed$
where
$car \in cars$
$elapsed \in \mathbb{N}$
then
$posX(car) := posX(car) + lean(car) \times elapsed \div 1000 * 50$
$posY(car) := posY(car) + vel(car) \times elapsed \div 1000$
end

Figure 3: An Event-B model for car racing.

objects. Every object has an horizontal and vertical position (*posX* and *posY*), a total function that maps the object with an integer number representing the position. Objects are two-dimensional with a *height* and a *width*. Variables *vel* (velocity), *acc* (acceleration), and *lean* (bending from the straight up position) keep track of the kinetic system of cars. Variable *collided* (a set) keeps track of the collided objects.

The event *update_pos* updates the *car*'s position according to Euler discrete movement approach, popular in real-time applications, where *elapsed* is a global measure of the time passed since the last update to the object's position, expressed in milliseconds (converted to seconds in the model). Events may only be triggered when the guard (the where condition) holds. The event *update_pos* models an unbounded substitution in Event-B (the any clause), so it allows the implementer of the event to choose any value for *car* and *elapsed* that verifies the guard. The symbol ":=" represents simple assignments in Event-B, $\varnothing$ the empty set, and $\to$ a total function.

The initialisation event describes what happens at the initial state of the machine. Machine invariants must hold after the initialisation event, and machine events must maintain the machine invariants, so they should still hold after the assignments in the body of events.

5. THE JML SPEC OF THE GAME

Figure 4 presents a partial JML output of applying the EventB2Jml tool [6] to the Event-B model in Figure 3. An Event-B abstract or refinement machine is translated as a JML abstract specification class. Event-B carrier sets and machine variables of type set are translated as `model` fields of type `BSet<Integer>`, a JML specification for sets. `model` variables are specification only variables, they exist in JML but not in Java. Event-B machine variables of type relation are translated as JML `model` variables of type `BRelation`, a JML implementation for relations in Event-B, along with invariants that model the type of the variable. For instance, the conditions `posX.isaFunction()` and `posX.domain().equals(objects)` ensure that `posX` is a total function.

The EventB2Jml tool produces a JML abstract method for every event of the Event-B model. The method includes a post-condition (the JML `ensures` clause) and a frame condition (the `assignable` clause). The frame-condition specifies the variables assigned by the event, *i.e.* `posX` and `posY`. Bounded variables, *i.e. car* and *elapsed*, are existentially quantified in JML. Method `override` is part of `BRelation` and implements ":=', and the JML `\old` operator returns the value of an expression evaluated before the method call.

The initialisation event is translated to a JML `initially` clause, which is an assertion that the initial values of the class fields must satisfy.

```
public abstract class RacingGameMachine {
 /*@ public model BSet<Integer> OBJECT; @*/
 /*@ public model BSet<Integer> TRACK; @*/

 /*@
  public model BSet<Integer> objects;
  public model BSet<Integer> tracks;
  public model BSet<Integer> obstacles;
  public model BSet<Integer> cars;
  public model BRelation<Integer,Integer> width;
  public model BRelation<Integer,Integer> height;
  public model BRelation<Integer,Integer> posX;
  public model BRelation<Integer,Integer> posY; ... @*/

 /*@ public invariant
  objects.isSubset(OBJECT) && tracks.isSubset(TRACK) &&
  obstacles.union(cars).equals(objects) &&
  obstacles.intersection(cars).equals(BSet.EMPTY) &&
  width.isaFunction() && width.domain().equals(objects) &&
  width.range().isSubset(NAT.instance) &&
  height.isaFunction() && height.domain().equals(objects) &&
  height.range().isSubset(NAT.instance) &&
  posX.isaFunction() && posX.domain().equals(objects) &&
  posX.range().isSubset(INT.instance) &&
  posY.isaFunction() && posY.domain().equals(objects) &&
  posY.range().isSubset(INT.instance) && ... @*/

 /*@ initially
   objects.isEmpty() && tracks.isEmpty() &&
   obstacles.isEmpty() && cars.isEmpty() && ... @*/

 /*@ assignable posX, posY;
  ensures (\exists Integer car;
   (\exists Integer elapsed; \old((cars.has(car) &&
     NAT.instance.has(elapsed))) &&
     posX.equals(\old(posX.override((
      new BRelation<Integer,Integer>().singleton(car,
  posX.apply(car) + lean.apply(car) * elapsed / 1000 * 50))))) &&
     posY.equals(\old(posY.override((
      new BRelation<Integer,Integer>().singleton(car,
  posY.apply(car) + vel.apply(car) * elapsed / 1000))))))); @*/
 public abstract void update_pos();
}
```

Figure 4: The JML spec for the racing car.

6. THE IMPLEMENTATION OF THE GAME

Figure 5 shows the Java code written for the JML example in Figure 4. A Java class is created that extends the JML abstract specification class. All JML `model` variable declarations are naturally transformed into Java variable declara-

```
class RacingGame extends RacingGameMachine {
 BSet<Integer> OBJECT;
 BSet<Integer> TRACK;

 BSet<Integer> objects;
 BSet<Integer> tracks;
 BSet<Integer> obstacles;
 BSet<Integer> cars;
 BRelation<Integer,Integer> width;
 BRelation<Integer,Integer> height;
 BRelation<Integer,Integer> posX;
 BRelation<Integer,Integer> posY;
   ...

 RacingGame( ) {
  OBJECT = new BSet<Integer>();
  TRACK  = new BSet<Integer>();
  objects = new BSet<Integer>();
  obstacles = new BSet<Integer>();
   ...
 }

 Integer car, elapsed;

 public abstract void update_pos() {
  if(cars.has(car) &&  NAT.instance.has(elapsed)) {
   BRelation tempPosX = posX.override(
    new BRelation<Integer,Integer>().singleton(car,
     posX.apply(car) + lean.apply(car) * elapsed / 1000 * 50));
   BRelation tempPosY = posY.override(
    new BRelation<Integer,Integer>().singleton(car,
     posY.apply(car) + vel.apply(car) * elapsed / 1000));

   posX = tempPosX;
   posY = tempPosY;
  }
 }
}
```

Figure 5: The Java code for the racing game.

tions since `BSet` and `BRelation` are indeed Java implementations for sets and relations. A typical event in Event-B is represented through an unbounded substitution (the ANY construct). For every event, variables bounded by the unbounded substitution - which are existentially quantified in JML, are declared as class fields in Java. These variables communicate the View with the Model, and accessors and mutators are implemented in Java (not shown here) to set the communication. JML postconditions are rewritten to produce the body of the Java method, and the constraints on the bounded variables (the guard of the event) are written as the guard of an if-statement within the body.

The JML `\old` operator is implemented in Java by creating temporary variables `tempPosX` and `tempPosY` that are evaluated before any assignment is made. This and the fact that left hand-side variables in Event-B machines are all different guarantee that semantics of Event-B simultaneous assignment is preserved.

No Java code is written for the JML invariants since all the proof obligations of the Event-B model from which the JML spec has been obtained have all been discharged. That is, as the translation from Event-B to JML is sound, the JML methods specifications comply with the Event-B machine invariants. However, if additional methods are added to the Java code, then one needs to be sure they do not break the invariants. To check whether Java code complies with the class invariants, JML provides a `represents` clause that relates JML specifications (invariants) with written code. In addition to this, the Java constructor must fulfil the JML `initially` clause. Event-B sets variables initialised as empty are allocated memory in Java by the constructor but no elements are initially added to these sets. To increase the confidence on the produced Java implementation, we further used the OpenJML tool [9] to verify the JML specification generated by Event2Jml.

	LOC	**% of Total**	**Generated**
Event-B Machine	253	22.7%	Manually
JML Abstract Model	226	20.3%	Automatically
Implemented Model	192	17.2%	Manually
Android+OpenGL code	443	39.8%	Assisted
TOTAL	1114	100%	N.A.

Table 1: Statistics of the car racing project

Machine	**LOC**	**%**	**POs**	**Aut. POs**
Abstract	132	52%	43	43 (100%)
Refinement	120	48%	38	35 (92%)
TOTAL	253	100%	81	78 (96%)

Table 2: Event-B Code statistics of the model

7. EXPERIMENTAL RESULTS

Tables 1 and 2 summarise the experimental results of the software development project presented in this paper. LOC stands for Lines of Code, and POs for the number of proof obligations generated. The first table shows how much of the project is automatically generated and how much code (60.2% = 22.7% + 20.3% + 17.2%) was needed to develop the critical part (the Model). The second table shows the scaling and evolution of the Event-B model. Regarding the third line of Table 1, much of the method implementations are a copy of the respective JML specification with a few details changed (following Section 6). Small parts of the fourth line of Table 1 are automatically generated by Eclipse. Most proof obligations in Table 2 are automatically discharged (78 out of 81); the other proof obligations require a considerable amount of effort and knowledge to be discharged.

8. RELATED WORK

In [8], the second author presents the translation from B to JML that is part of the core implementation of the EventB2Jml tool. Mèry and Singh [17] define a translator (implemented as a Rodin plug-in) that automatically translates Event-B machines into several different languages: C, C++, Java and C#. Unlike EventB2Jml, which utilises Rodin to check the consistency of the model, their tool checks syntax and type consistency before generating the target programming language code. Also, Wright [19] defines a B2C extension of the Rodin platform that translates Event-B models into C code. This work considers only simple translations of formal concrete machines. The main issue with these tools is that the user has to provide a final (or at least an advanced) refinement of the system so that it can be directly translated to code. Edmunds and Butler [10, 11] present a tasking extension for Event-B that generates code for concurrent programs (targeting multitasking, embedded and real-time systems). Jin and Yang [13] outline an approach for translating VDM-SL to JML. Their motivations are similar to the ones in [8] in that they view VDM-SL as a better language for modelling at an abstract level (much the way that we view Event-B), and JML as a better language

for working closer to an implementation level. In fact, they translate VDM variables to Java fields, thus dictating the fields of an implementation. In [4], Sue Black *et al.* show how formal methods can be integrated into agile methodologies of software development, and in [15], Qaisar Malik *et al.* propose a method for generating Java template implementations for Event-B models.

9. CONCLUSION

In our methodology, the Model part is written in Event-B and automatically translated to JML, which is manually coded into Java (Android's language), briefly modifying the JML spec itself. We used formal methods to develop the logic of the game, and proved the soundness of the model by discharging all its proof obligations in Rodin. All together, the final result down to Java code is sound (using translation tools also sound). This avoided the authors the time of debugging the application for internal consistency and logical errors (such as flawed collision detection or erroneous scoring). Avoiding debugging time and software maintenance on costly and long-cycled projects like video games (6 months through 3 years usually) reduces budgetary concerns, code maintenance efforts, and also improves public confidence on entertainment software developers.

In the following, we list a few hints following our experience developing the game project. *i*) Event-B's programming paradigm differs greatly from that of Java's, so experience in knowing the outcome of the Event-B to JML translation is needed. *ii*) Discharging proofs in Rodin is laborious and requires specific mathematical knowledge which a video-game company normally would not have. Nevertheless, in our case study, most of the proof obligations (78 out of 81) were discharged automatically by Rodin. *iii*) Game development generally uses floating-point values to model features such as position or dimension, so careful attention was needed to treat these features in the discrete mathematics world of Event-B. *iv*) The translation of one (refinement) machine results into a large Java class. Programming the complete logic of a professional game in just one class is cumbersome. We recommend strategically partitioning the logic of the Event-B model in various machines.

As future work we plan to use standard Java classes to implement set and relations rather than the `BSet` and `BRelation` JML classes, which are more suitable for verification purposes than for performance. The translation to these Java classes might be automatic. We plan to undertake a case study in which we compare the time required to refine an abstract Event-B model to a machine that can directly be translated to Java code with the time needed to produce a Java implementation by using the EventB2Jml tool to translate the same Event-B model to JML, implementing the JML specification by hand, and verifying the Java code against the JML specification.

10. REFERENCES

[1] The racing car game, 2012. Available at `http://cic.javerianacali.edu.co/~ysperchy/formal-game`.

[2] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[3] J.-R. Abrial. *Modeling in Event-B: System and Software Design*. Cambridge University Press, 2010.

[4] S. Black, P. Boca, J. P. Bowen, J. Gorman, and M. Hinchey. Formal versus agile: Survival of the fittest. *IEEE Computer*, 42(9):37–45, 2009.

[5] C.-B. Breunesse, N. Cataño, M. Huisman, and B. Jacobs. Formal methods for smart cards: An experience report. *Science of Computer Programming*, 55(1-3):53–80, March 2005.

[6] N. Cataño and V. Rivera. The EventB2JML tool, 2012. Available at `http://\-poporo.\-uma.\-pt/\-Projects/\-favas/\-eventb2jml.html`.

[7] N. Cataño and C. Rueda. A machine-checked proof for a translation of event-b machines to jml specifications. Technical Report CIC-007-12, Pontificia Universidad Javeriana, `http://cic.puj.edu.co/wiki/doku.php?id=grupos:secsy:secsy`, 2012.

[8] N. Cataño, T. Wahls, C. Rueda, V. Rivera, and D. Yu. Translating B machines to JML specifications. In *27th ACM Symposium on Applied Computing, Software Verification and Testing track (SAC-SVT)*, Trento, Italy, March 26-30 2012.

[9] D. R. Cok. OpenJML: JML for Java 7 by extending OpenJDK. In *NASA Formal Methods Symposium*, pages 472–479, 2011.

[10] A. Edmunds and M. Butler. Tool support for Event-B code generation. In *WS-TBFM2010*, 2010.

[11] A. Edmunds and M. Butler. Tasking Event-B: An extension to Event-B for generating concurrent code. In *PLACES 2011*, 2011.

[12] google Inc. The android platform. `http://developer.android.com/design/index.html`, 2012.

[13] D. Jin and Z. Yang. Strategies of modeling from VDM-SL to JML. In *International Conference on ALPIT*, pages 320–323, 2008.

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

[15] Q. A. Malik, J. Lilius, and L. Laibinis. Scenario-based test case generation using Event-B models. In *International IEEE Conference on Advances in System Testing and Validation Lifecycle (VALID)*. IEEE Computer Society, 2009.

[16] R. Mejer. *Professional Android 4 Application Development*. Wrox, 2012.

[17] D. Mèry and N. K. Singh. Automatic code generation from Event-B models. In *Proceedings of the Second SoICT*, SoICT '11. ACM, 2011.

[18] J. Nielsen. *Usability Engineering*. AP Professional, 1993.

[19] S. Wright. Automatic generation of C from Event-B. In *Workshop on IM_FMT*. Springer-Verlag, 2009.

[20] V. Young. *Programming a Multiplayer FPS in DirectX*. Charles River Media, 2004.