Sudoku App: Model-Driven Development of Android Apps Using OCL?

Yoonsik Cheon and Aditi Barua

TR #17-91
November 2017

Keywords: class invariant, model-driven development, pre and postconditions, Android, Object Constraint Language (OCL).

1998 ACM CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications—languages; D.2.2 [*Software Engineering*] Design Tools and Techniques—object-oriented design methods, state diagrams; D.2.4 [*Software Engineering*] Software/Program Verification—class invariants, formal methods; F.3.1 [*Logics and Meaning of Programs*] Specifying and Verifying and Reasoning about Programs—assertions, invariants, pre- and post-conditions, specification techniques

A shorter version of this document entitled "Model Driven Development for Android Apps" appeared in the *Proceedings of the 2018 International Conference on Software Engineering Research & Practice, Las Vegas, Nevada, July 30 - August 2, 2018*, pages 17-22, 2018.

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A

Sudoku App: Model-Driven Development of Android Apps Using OCL?

Yoonsik Cheon
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas
ycheon@utep.edu

Aditi Barua
Department of Economics and Finance
The University of Texas at El Paso
El Paso, Texas
abarua@utep.edu

Abstract— **Model driven development (MDD) shifts the focus of software development from writing code to building models by developing an application as a series of transformations on models including eventual code generation. Can the key ideas of MDD be applied to the development of Android apps, one of the most popular mobile platforms of today? To answer this question, we perform a small case study of developing an Android app for playing Sudoku puzzles. We use the Object Constraint Language (OCL) as the notation for creating precise models and translate OCL constraints to Android Java code. Our findings are mixed in that there is a great opportunity for generating a significant amount of both platform-neutral and Android-specific code automatically but there is a potential concern on the memory efficiency of the generated code. We also point out several shortcomings of OCL in writing precise and complete specifications for UML models and suggest a few extensions and improvements to make it more expressive and suitable for MDD. The reader is assumed to be familiar with OCL.**

Keywords— **class invariant, model-driven development, pre and postconditions, Android, Object Constraint Language.**

I. Introduction

Model driven development (MDD) is a paradigm to solve a number of problems associated with the development of a large complex application [25]. It relies on the use of models as the basis for software development and shifts the focus of development from writing code to building models. The key idea of MDD, in particular the Model Driven Architecture (MDA) of the Object Management Group (OMG) [5] [21], is to develop applications as series of transformations on models and to generate code automatically from the models. It is also suggested to construct models at several different abstraction levels, such as computation independent models (CIM), platform independent models (PIM), platform specific models (PSM), and implementation specific models (ISM). The underlying assumption of MDD, however, is the existence of an appropriate model -- a representation that is sufficiently general to capture the semantics of many different domains and yet precise enough to support eventual transformation into code.

Android is one of the most popular mobile platforms today paving the way for the development of a flood of apps; the reported market share of Android-based smartphones in the first quarter of 2017 is 85.0% (http://www.idc.com). Android provides its own operating system, libraries, and application programming interfaces (API), and its apps are written in Java. Despite the use of Java, the Android application domain is sufficiently narrow with several interesting characteristics, such as XML-based UI, event-based reactive apps, life cycles of apps, and the single active app. However, one key difference is that Android devices are resource-constrained in storage capacity and battery lifetime, and thus memory efficiency is an important quality factor for Android apps [20] [24]. Android apps are also relatively small and not as complex as typical enterprise applications. Thus, it is natural to ask whether the ideas of MDD are applicable to the development of Android apps and whether the promised benefits of MDD such as productivity can be obtained even without using MDD-specific tools. There is no well-known commercial quality MDD tools for Android.

MDA of OMG [5] [21], one of the best known MDD approaches, recommends to use well-defined, standard notations like the Unified Modeling Language (UML) [12] [23]. One key requirement of MDD is the availability of a precise model to generate working code from it. A formal notation such as the Object Constraint Language (OCL) [22] [27] of UML can play an important role to build such a precise model. OCL is a textual, declarative notation to specify constraints or rules that apply to models expressed in UML diagrams such as class diagrams. We would like to know whether UML/OCL is suitable for creating precise models that can be used as the basis of MDD.

In this paper, we perform an experiment to answer the above two questions. Our experiment is a small case study of applying the key ideas of MDD to the development of an Android app. However, unlike previous work on the use of MDD for mobile apps (e.g., [3] [14] [16]), the purpose of our study is not to propose new languages, techniques, methods, or toolsets. Our main objective is to study a practical application of the key components of MDD -- precise models and code generation -- in developing Android apps using the standard modeling notation UML/OCL. Another objective is to study suitability of OCL in

creating precise models that can be used as the basis of MDD. We perform our case study manually without using any specific MDD technology or toolset. We use only a free UML tool to draw UML diagrams and write and attach OCL constraints to the UML models [9]. But as said above, it is not our purpose to evaluate the support tools.

The app we develop is for playing Sudoku puzzles, one of the running examples used in [18]. Sudoku is a logic-based, number placement puzzle for a single player. The objective of the game is to fill a 9×9 grid with numbers so that each column, row, and 3×3 sub-grid that composes the grid contains all of the numbers from one to nine. Thus, the same number cannot appear more than once in the same column, row or sub-grid. Figure 1 shows a partially solved Sudoku puzzle; a gray square represents a number that is given and thus cannot be changed. A game starts with a partially filled grid, typically having at least 17 numbers, normally 22-30 numbers. A well-designed puzzle has a single solution.

3	2	9		7		1		
7	8	6	5		1	2		
1	5	4	3			7	9	
	1			4		6		5
2	6				5	9		
						3	2	1
6	7	2				5	1	9
		5		1	9	8	6	
			6		7	4	3	

Figure 1. Sudoku puzzle

Our case study consists of three main steps: (a) creating a precise specification model, (b) creating a detailed design model, and (c) generating Android Java code from the design model. For both specification and design models, we use UML and OCL and create static models (class diagrams) as well as dynamic models (state machines). We generate functioning code manually but systematically from the UML models and accompanying OCL constraints.

The case study steps are reflected in the structure of this paper. In Section II below, we create a specification model of our app to describe precisely what the app has to do. In Section III, we transform the specification model to a detailed design model. We incorporate design decisions to the specification model by extending existing classes and adding new PIM/PSM classes. The design model consists of an architectural design, a UI design and detailed designs of classes including algorithms. In Section IV, we generate functioning Android Java code from our design models manually. In Section V, we share our findings and lessons learned from the case study, and in Section VI, we provide a concluding remark.

II. SPECIFICATION

In this section we create a specification model for our app. Our specification model consists of two UML diagrams, a class diagram and a state machine diagram. The class diagram models the entities and the static structure of our app while the state machine diagram specifies the dynamic behavior of the app, specifying the allowed sequences of operation calls.

Figure 2 show the static model. A Sudoku game consists of a 9×9 grid with numbers, called a *board*. A 3x3 sub-grid of a board is called a *box*, and each cell of the grid is called a *square*. The association between Board and Square is *derived* from the Board-Box and the Box-Square associations; it is calculated from the other associations. All the classes appearing in the diagram will be specified in OCL later in this section.

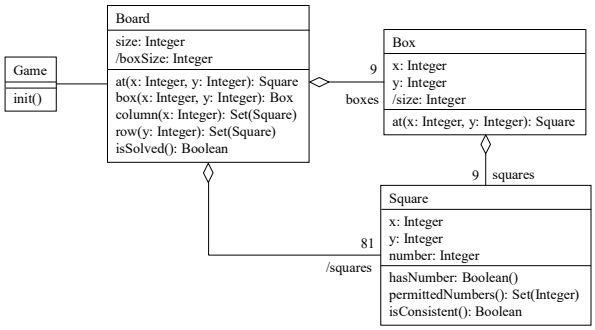
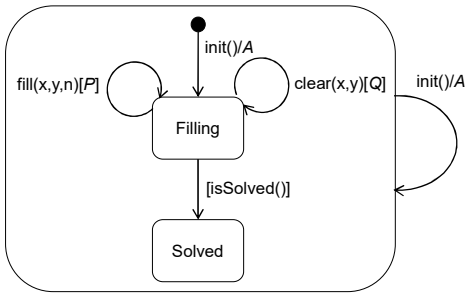


Figure 2. Class diagram

Figure 3 show a dynamic model of the game. It is a protocol state machine for the Game class and specifies the allowed sequence of operation calls.



clear(x,y) ≡ board.at(x,y).setNumber(0)
fill(x,y,n) ≡ board.at(x,y).setNumber(n)
A ≡ given = board.filledSquares()
P ≡ not isSolved() and given->excludes(board.at(x,y))]
Q ≡ not isSolved() and given->excludes(board.at(x,y))
and board.at(x,y).hasNumber()

Figure 3. Protocol state machine

An initial board configuration, partially filled grid of numbers that has a solution, is created by the *init*() operation. Numbers are filled in or removed from the board until the puzzle is solved. The guards *P* and *Q* in the *fill*(*x*,*y*,*n*) and *clear*(*x*,*y*) transitions prevents the fixed number given by the initial configuration from being replaced or removed. The composite transition *init*() allows one to start a new game in any state. Below we specify OCL constraints for classes appearing in the class diagram.

A. Game Class

The key responsibility of the Game class is to create a new game by coming up with a partially filled grid of numbers. As will be shown later, this turns out to be one of the most complicate tasks. The partially filled board should have a solution --

preferably one -- and this is expressed by a newly-introduced operation *isSolvable*().

```
context Game::init()
post: isSolvable() and 0 < filled and filled < n
  where n = board.size * board.size,
    filled = board.squares->select(hasNumber())->size()

context Game
def: isSolvable():Boolean = Board.allInstances()->exists(
  size = self.board.size and isSolved() and
  squares->forAll(let s = self.board.at(x, y) in s.hasNumber()
    implies hasNumber() and number = s.number))
```

A board is *solvable* if there exists a solved version of it, a board with all the empty squares filled with non-conflicting numbers; see Section B below for the specification of the *Board*::*isSolved*() operations. Note that in the specification of *Game*::*init*() we introduced our own extension to OCL, *where* clause, to present a constraint in a more structured fashion. It is a syntactic sugar in that *E where D* can be translated to *let D in E* in the standard OCL.

B. Board Class

A board consists of a set of squares subdivided into boxes. The *size* attribute denotes the width and height of a board. The first invariant below constrains it to be a square, and the second invariant together with the invariant of the Square class (see Section C) asserts that each box is uniquely identified by its row and column indexes.

```
context Board
inv: size >= 9 and Sequence{1..size}->exists(i | i * i = size)
inv: boxes->isUnique(Tuple{col = x, row = y})

context Board::boxSize: Integer
derive: Sequence{1..size}->any(i | i * i = size)

context Board::squares: Set(Square)
derive: boxes.squares->asSet()
```

As shown above, the values of derived attributes and association ends are specified using the *derive* clause. Below we specify several representative operations of the Board class.

```
context Board::at(x1: Integer, y1: Integer): Square
pre: 0 <= x1 and x1 < size and 0 <= y1 and y1 < size
post: result = squares@pre->any(x = x1 and y = y1)

context Board::column(x1: Integer): Set(Square)
pre: 0 <= x1 and x1 < size
post: result = squares@pre->select(x = x1)

context Board::isSolved(): Boolean
body: squares->forAll(hasNumber() and isConsistent())
```

The specification of the *Board*::*isSolved*() operation states that a board is solved if each of its squares has a consistent or non-conflicting number; see Section D for the consistency of a square.

C. Box Class

A box consists of a set of squares, each denoted by a pair of 0-based column (*x*) and row (*y*) indexes. The size of a box is determined by the size of the board.

```
context Box:: size: Integer
derive: board.boxSize

context Box
inv: 0 <= x and x < size and 0 <= y and y < size
inv: xys = coords where
  xys = squares->collect(Tuple{col = x, row = y})->asSet(),
  coords = Sequence{0..size-1}->collect(i|
    Sequence{0..size-1}->collect(j|
    Tuple{col = x*size + i, row = y*size + j}))->asSet()
```

One interesting modeling choice is the indexes of squares (see also Section D below). They can be local and unique in each box, e.g. 0–*Box*::*size*, or global and unique in the board, e.g. 0–*Board*::*size*. We found that use of square indexes outside the Box class are for the second type. This choice affects the formulation of the second invariant above stating that each square of a board be uniquely identified by its column and row indexes.

D. Square Class

A square of a board is uniquely identified by a pair of column (*x*) and row (*y*) indexes. A square may have a number between 1 and *Board*::*size*, inclusive. An empty square is denoted by the number 0.

```
context Square
inv: 0 <= x and x <.board.size and 0 <= y and y <.board.size
inv: 0 <= number and number <= board.size

context Square::hasNumber(): Boolean
body: number <> 0

context Square::permittedNumbers(): Set(Integer)
body: result = all - b - h – v where
  all = Sequence{1..board.size}->asSet(),
   b = box.squares->excluding(self)->collect(number),
   h = board.row(x)->excluding(self)->collect(number),
   v = board.column(y)->excluding(self)->collect(number)

context Square::isConsistent(): Boolean
body: b and h and v where
  b = box.squares->select(s| s <> self and
      hasNumber() and number = self.number)->size() = 0,
  h = board.row(y)->select(s| s <> self and
      hasNumber() and number = self.number)->size() = 0,
```

```
v = board.column(x)->select(s| s <> self and
    hasNumber() and number = self.number)->size() = 0
```

The *permittedNumbers*() operation returns the set of all non-conflicting or allowed numbers for a square. A number is allowed in a square if it doesn't appear in any other squares in the same box, column, and row. The *isConsistent*() operation determines if a square is consistent. A square is *consistent* if its number doesn't appear in any other squares in the same box, column, and row. An empty square is consistent, and this can be inferred from the specification.

III. DESIGN

In this section, we create a detailed design model for our app that conforms the specification model created in the previous section. Our design model consists of a UI design, an architecture, and detailed designs of classes and algorithms. We uses the model-view-control (MVC) architectural pattern, where the model (business logic) is separated from the view (UI) and the control code. This is a good style for Android apps as the UI of an app can be defined in XML (see Section IV). Our design model consists of all the classes from the specification model with some extensions. We also add several new classes. Although we don't make a clear distinction between PIM and PSM, certain extensions to the specification classes and newly introduced classes are PIM while others are PSM in that they are Android-specific extensions or classes.

A. UI

Figure 4 shows the layout of the UI consisting of several buttons (new, delete, and numbers 1-9) and a custom view. A custom view is used to display the current state of the board in 2-D graphics as well as to select a square to fill in a new number or delete the current one.

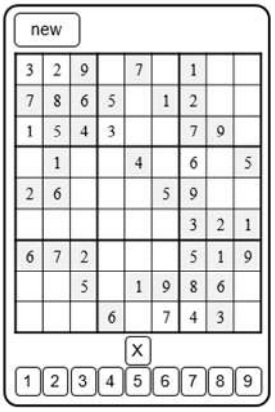


Figure 4. UI design

We can associate UI events such as button click and screen touch with game operations as follows.

- New button: *Game*::*init*()
- Number buttons: *Game*::*fill*(*x*,*y*,*n*), where *x* and *y* is the indexes of the selected square and *n* is the label (i.e., the represented number) of the clicked button.
- Delete button: *Game*::*clear*(*x*,*y*)
- Screen touch: set the square selection

With these bindings, the protocol state machine in Section II can be turned into a state machine describing the allowed sequences of user interactions.

B. Classes

Object-oriented design is all about refining classes from the analysis or specification model as well as introducing new classes by making decision or implementation-oriented decisions. The class diagram in Figure 5 depicts a portion of our specification model extended with several design decisions.

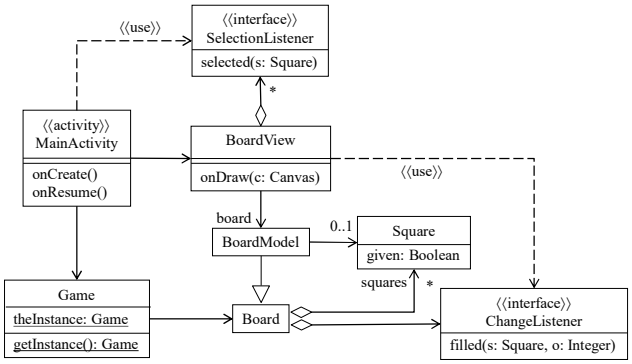


Figure 5. Design class diagram

Several new classes and interfaces are introduced, and existing classes are extended with new attributes, associations and operations. In Android, an *activity* is an app component responsible for a single UI screen. An app may consist of one or more activities. The MainActivity class is the main entry point to our app. The BoardView class is a custom view class to display a board, and the BoardModel is a subclass of Board to contain view-dependent board data, e.g., the currently selected square that is to be displayed differently. The Observer design pattern [11] is used to notify changes in a BoardView and a Board to interested observers such as MainActivity. For this, two interfaces SelectionListener and ChangeListener are introduced. Below we explain in detail some of our design decisions shown in the class diagram.

One of interesting platform-specific design decisions is to use the Singleton design pattern [11] for the Game class. The main reason is to save and restore the game state when the activity is destroyed and later recreated. On Android, an activity may be paused, stopped, and destroyed at any time by the operating system. For example, it is destroyed when the screen orientation changes or another activity is launched. When the main activity is (re)created, the single instance of the Game class is retrieved and thus maintains its previous state automatically. The specification of the *getInstance*() operation is shown below.

```
context Game::getInstance()
post: result = theInstance and
  (if  theInstance.oclIsUndefined()@pre
   then theInstance.oclIsNew()
   else theInstance = theInstance@pre endif)
```

The MainActivity class is the controller for the app and is responsible for configuring the UI and creating various objects such as a board, boxes, and squares. Below are shown the specifications of its two lifecycle operations.

```
context MainActivity::onCreate()
post: game = Game.getInstance()
  and self^setLayout(?: Integer)
  and boardView^addSelectionListener(?: SelectionListener)
  and buttons->forAll(b |
    b^setOnClickListener(?: View::OnClickListener))

context MainActivity::onResume()
post: game = Game.getInstance() and boardView^invalidate()
```

The *onCreate*() operation is defined by the Android framework and is called when an activity is newly created. It is responsible for configuring the UI and creating the initial state of the activity. The UI configuration and the event handler registrations are loosely specified using the OCL message expression. The OCL *hasSent* (^) operator specifies that a specified message should be sent during the execution of the operation [22]. As shown here, our approach for specifying Android framework operations is to focus only on the key state change or to gross over the state change and instead specify the interactions needed to achieve the change (see Section V for a discussion on this). The *onResume*() operation is called when an activity starts interacting with the user. It restores the state of the activity and refresh the board view to display the restored state.

The BoardView class is a custom view class to display a board. It also responds to a touch event to select a square of the displayed board. The BoardView class is associated with a subclass of the Board class, named BoardModel, to remember the view-specific board data such as the currently-selected square.

```
context BoardModel
inv: square <> null implies squares->includes(square)

context BoardView::onDraw(c: Canvas)
post: self^drawLines(c) and (selected <> null implies
  self^drawSquare(c, selected, ?: Paint)) and
  (board.squares->forall(s| s.hasNumber() implies
   self^drawSquare(c, s))
  where selected = board.square

context BoardView::drawSquare(c:Canvas, s:Square, p:Paint)
post: let w = getMeasuredWidth().min(getMeasureHeight())
      / board.size, x = s.x * w, y = s.y * w
  in c^drawRect(x, y, x + board.size, y + board.size, p) and
    (s.hasNumber() implies
      c^drawText(s.number, ?: Real, ?: Real, ?: Paint))
```

The *onDraw*() operation is an Android framework operation overridden to draw a custom view such as BoardView. As shown in its specification, drawing a 2-D representation of a board involves drawing horizontal and vertical lines of the grid (*drawLines*) and drawing the numbers for all the squares of the board (*drawSquare*). The screen coordinates of a square are calculated from its indexes as well as the width and height of the screen. These operations are loosely specified using the OCL message expressions. Their postconditions constrain only that appropriates drawing messages be sent during the execution; they don't say anything about the effect or state changes.

On Android, one way to detect a touch gesture that occurs when a user places one or more fingers on the touch screen is to override the *onTouchEvent*() operation in a view class. As specified below, it determines the square displayed at the touched screen coordinate (see the *locateSquare*() operation), sets the square selection, refreshes the view by invaliding it, and notifies the new selection to all registered listeners. The notification of a new selection can be elegantly specified using the OCL message expression.

```
context BoardView::onTouchEvent(e: MotionEvent)
post: e.getAction() = MotionEvent.ACTION_UP implies
  let s = locateSquare(e.getX(), e.getY()) in
    s <> null implies board.square =
      (if s = board.square@pre then null else s endif)
      and self^invalidate()
      and changeListener->forall(l| l^selected(s))

context BoardView::locateSquare(x: Real, y: Real): Square
pre: 0 <= x and x <= getMeasuredWidth()
pre: 0 <= y and y <= getMeasuredHeight()
post: let w = getMeasuredWidth().min(getMeasureHeight())
      / board.size, m = w * board.size
  in result = if x > m or y > m then null
              else board.at((x/w).floor(), (y/w).floor()) endif
```

In our design, we didn't make a clear distinction between PIM and PSM, but several newly-added classes such as BoardView and MainActivity are PSM while others such as BoardModel and listener interfaces are PIM. Similarly, the Game extension is PSM while the Square extension is PIM.

C. Algorithms

For operations whose pre and postconditions cannot be translated directly to executable code, we design algorithms for them. One such operation is the *Game*::*init*() operation to create an initial board configuration. It creates a partially filled grid of numbers such that each number 1 to 9 occupies each row, column and box just once. Ideally, the created grid should have only one solution. In this section, we design one possible

algorithm for the operation and describe it by drawing a behavioral state machine (see Figure 6).

Our algorithm consists of two steps: creating a solved grid of numbers and removing numbers repeatedly that, upon removal, are likely to produce a single solution. To determine the likelihood of one solution, a candidate number is removed from the board and then the board is solved again. If the solver fills the candidate square with the same number, it is more likely to have a single solution; otherwise, the candidate square is rejected and a new one is attempted. The key of our algorithm is a solver that finds a solution for an empty or partially filled board. Below we design a solver based on backtracking. A backtracking algorithm constructs a solution by making succession of choices. If there is no choice available, it retraces backwards through the choices made, undoing their effect, until an alternative choice is found.

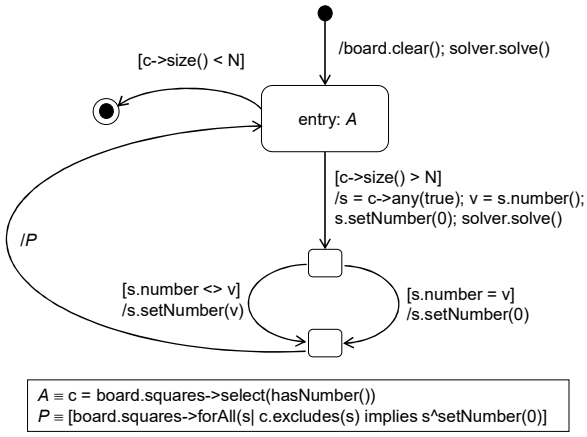


Figure 6. Algorithm for creating a new game

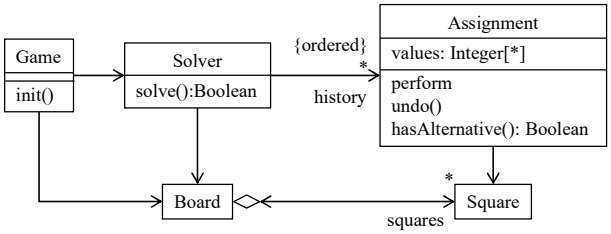


Figure 7. Design of a backtracking solver

As shown in Figure 7, a solver has a history of choices made, where a choice is an assignment of a number to a square. The Assignment class represents a choice as a tuple of a square and alternative numbers assignable to the square. Below we first specify *Solver*::*solve*() operation and then design its algorithm.

```
context Solver::solve(): Boolean
post: result = isSovable()@pre and
  (result implies board.isSolved() and board.squares->forAll(
   hasNumber()@pre implies number = number@pre)

context Solver
def: isSolvable():Boolean = Board.allInstances()->exists(
  size = self.board.size and isSolved() and
  squares->forAll(let s = self.board.at(x, y) in s.hasNumber()
  implies hasNumber() and number = s.number))
```

The specification of the newly-introduced *isSolverable*() operation is similar to that of the Game class. The behavioral machine shown Figure 8 describes a backtracking algorithm for the *solve*() operation.

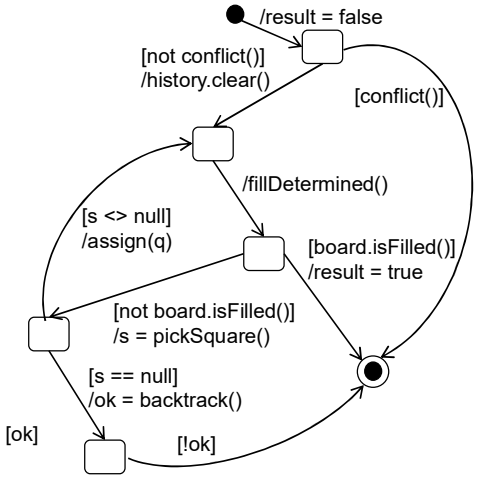


Figure 8. Backtracking algorithm

The key of the algorithm is first to fill all empty squares that have only one permitted number (*fillDetermined*). A number is permitted in a square if it doesn't appear in the row, column, and box of the square. The algorithm then picks an empty square that has permitted numbers (*pickSquare*) and assigns one such number to the square (*assign*). If no square can be picked, the algorithm does backtracking (*backtrack*). Below we specify the helper operations introduced in the state machine diagram.

```
context Solver
def: conflict():Boolean =
  board.squares->exists(not isConsistent())

context Solver::fillDetermined()
post: board.squares->forAll(not hasNumber()@pre and
  permittedNumbers()@pre->size() = 1 implies
  number = permittedNumbers()@pre->any(true))

context Solver::pickSquare(): Square
post: result = if c->size() > 0 then c->any(true) else null endif
  where Square::isPickable():Boolean = not hasNumber()
    and permittedNumber()->size() > 0,
  c = board.squares@pre->collect(s| isPickable() and
      board.squares@pre->forAll(isPickable() implies
        permittedNumbers()->size()
        >= s.permittedNumbers()->size())
```

context Solver::assign(s: Square)
post: **let** a = **new** Assignment(s)
  in a^perform() **and** history = history.append(a)@pre

The *pickSquare*() operation is specified to pick a square that is empty but has at least one permitted value. If there are more than one such square, it picks an arbitrary square with the smallest number of permitted values. The specification of the *Solver*::*backtrack*() operation is complicate and involved despite its straightforward algorithm, so instead of specifying it in OCL we design its algorithm directly (see Figure 9).

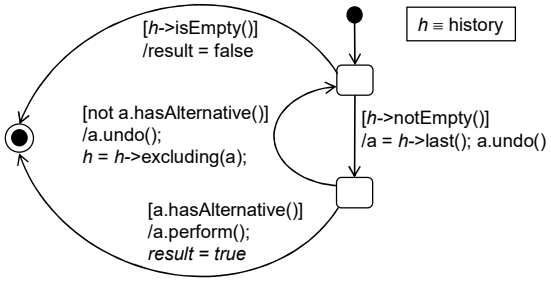


Figure 9. Algorithm for the *backtrack* operation

The specification of the Assignment class are shown below.

context Assignment::Assignment(s: Square)
post: square = s **and** values = s.permittedNumbers()

Assignment::perform()
pre: values.size() > 0
post: **let** n = values.any(true)@pre **in**
  square.number = n **and** values = values->excluding(n)@pre

Assignment::undo()
post: square.number = 0

Assignment::hasAlternative(): Boolean
body: values.siz() > 0

IV. Implementation

In this section, we explain how we translate our design shown in the previous section to functioning Android Java code including both the UI and the functional core. The key of our translation approach is to generate code incrementally on a need basis. We first generate skeletal code from the class diagram considering only the structural aspect such as classes and their attributes. We then translate operations to Java methods, considering one operation at a time. We focus on core operations that are explicitly specified in the model. The underlying idea is to introduce additional structural elements (e.g., fields for associations) and behavior such as helper methods (e.g., getter and setters for fields and association ends) on a need basis. This will let us to generate minimal code with minimal effort.

The UI of an Android app is typically defined declaratively in XML, called a layout. We create our layout XML file using the Layout Editor of Android Studio, a visual editor for creating layouts by dragging widgets. We use a recently-introduced layout called ConstraintLayout as our base layout. It allows us to compose our layout file corresponding to the UI design of the previous section entirely by dragging and dropping (see Figure 10).

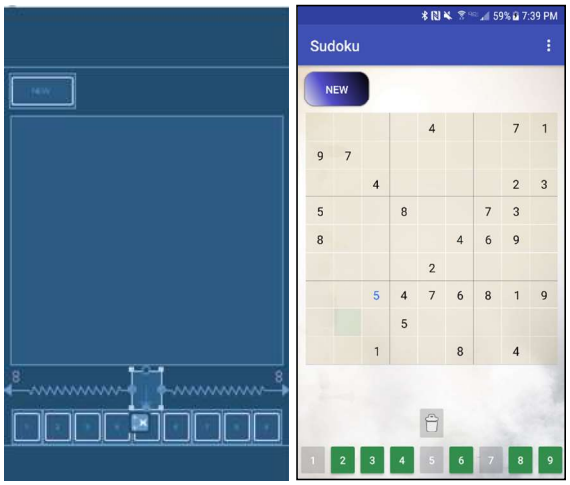


Figure 10. UI implementation

We next translate our detailed design -- consisting of a class diagram, OCL specifications, and behavioral state machines -- to functioning Java source code. This involves translating both the structure (attributes and associations) and the behavior (pre/postconditions and behavioral state machines). As said earlier, we introduce getters and setters for attributes and role names (association ends) on a need basis. For this, we check for the *readOnly* property for an attribute. We also look at each occurrence of an attribute and a role name in OCL constraints to determine if it appears in a read context or a write context. Every occurrence in an invariant and a precondition is a read context. An occurrence in a postcondition is in general a write context unless suffixed with @pre; an unqualified feature denotes its new value in the post-state, which may be different from its pre-state value. For an attribute and an association end with multiplicity bigger than one, its getter returns an immutable view of the collection to prevent mutation, e.g., the getter of the *squares* role of the Box-Square association is defined as:

```
public List<Square> squares() {
  return Collections.unmodifiableList(squares);
}
```

We may also introduce additional helper operations to manipulate a single element of a collection, e.g., to access an element, add a new element, or remove an existing element. For an association, we determine its navigability based on its use in OCL constraints. In general, a derived attribute and association

is translated to a getter; its return value is determined by the OCL *derive* expression of the attribute and association.

While the class diagram is used to generate the basic structural elements of an implementation, it is necessary to use OCL constraints and dynamic models like behavior state machines to generate the detailed functional behavior of an application. Operations of classes are translated to methods based on either their postconditions or behavioral state machines. A behavioral state machine can be automatically translated to functional code if it is written in the style used in this paper. For example, the behavioral state machine of the *Solver*::*solve*() in Section III.C can be translated to the following code.

```
public boolean solve() {
  if (conflict()) { return false; }
  history.clear();
  while (true) {
    fillDetermined();
    if (board.isFilled()) { return true; }
    Square s = pickSquare();
    if (s != null) { assign(s); }
    else if (!backtrack()) { return false; }
  }
}
```

When an operation doesn't have an algorithm specified as a behavioral state machine. We use its postcondition to generate its functional code. Its precondition can also be used to generate a runtime check to determine whether the assumption of the operation is met or not [1] [7]. When the postcondition is written in an explicit style in which the expected change in the value of features are specified definitely, it can be used to derive code systematically. The new features of Java 8 language and APIs such as lambda expressions and the Stream API are of great help in systematically deriving concise code from constraints [13]. The OCL collection iterators such as *forAll* can be translated to the corresponding Java stream operations such as *allMatch* and their argument expressions to Java lambda expressions. For example, shown below is one possible translation of the *Square*::*isConsistent*() operation described in Section II.D.

```
context Square::isConsistent(): Boolean
body: b and h and v where
  b = box.squares->select(s| s <> self and
      hasNumber() and number = self.number)->size() = 0,
  h = board.row(y)->select(s| s <> self and
      hasNumber() and number = self.number)->size() = 0,
  v = board.column(x)->select(s| s <> self and
      hasNumber() and number = self.number)->size() = 0

public boolean isConsistent() {
  return isConsistent(box.squares())
    && isConsistent(board.row(y))
    && isConsistent(board.column(x));
}

private boolean isConsistent(Collection<Square> sqs) {
  return sqs.stream().filter(s -> s != this &&
    s.hasNumber() && s.number() == number()).count() = 0;
}
```

The OCL *select* iterator is nicely translated to the Java stream method *filter* and its argument expression becomes a lambda expression in Java. The resulting code thus has a structure similar to that of the OCL constraint. As shown above, private helper methods may be introduced to factor out common subexpressions of constraints.

Generating code for update operations are more involved than query operations, as one needs to figure out the features whose values have to be changed. As an example, let us consider the *Solver*::*fillDetermined*() operation (see Section III.C).

```
context Solver::fillDetermined()
post: board.squares->forAll(not hasNumber()@pre and
  permittedNumbers()@pre->size() = 1 implies
  number = permittedNumbers()@pre->any(true))
```

The postcondition constrains those squares of the board that are empty and have exactly one permitted value in the pre-state. The *number* attributes of these squares have to be modified to satisfy the postcondition; that is, the modifiable features are uniquely determined. In spite of the use of the *any* operation that picks an arbitrary element of a collection, the post-values of the *number* attributes are not loosely specified because the collections are singletons. Thus, Java code can be generated from the postcondition as shown below. We use the *forEach* method defined on collections.

```
private void fillDetermined() {
  board.squares().forEach(s -> {
    if (!s.hasNumber()
      && s.permittedNumbers().size() == 1)  {
      s.setNumber(s.permittedNumbers().get(0));
    }
  });
}
```

We often have to address platform-specific restrictions or constraints during the implementation. For example, there is a one-to-one association between BoardView and BoardModel in our design model. However, this one-to-one relationship cannot be maintained as an invariant on Android because views are automatically created by the system according to the layout specified in XML. There is no way to provide a board object to the constructors of the BoardView class. Therefore, the multiplicity of the Board end of the association needs to be weakened to 0..1. And all its occurrences in OCL constraints need to be checked for nullness. For example, we need to

change the specification of the BoardView::*onDraw* operation as follow (see Section III.B).

context *BoardView*::onDraw(c: Canvas)
post: board <> null **implies** …

In fact, this revised specification produces more robust code that works harmoniously in the Android environment. Due to the new non-nullness guard, the *onDraw* method should work even if there is no associated board. As the result, Android tools like the Layout Editor will be able to display the BoardView class correctly.

As described above, we translated our specifications and design models to functioning Android Java code sort of semi-automatically without much coding effort.[1] The generated code satisfies the constraints of our models, such as class invariants and operation postconditions. Table 1 provides a summary of the generated classes. For each field and method of a class, we traced its source: specified attribute/operation (spec), derived attributes (deri), associations (assoc), attributes (attr), and helper methods (help).

Table 1. Summary of generated code

Class	Fields			Methods			
	spec	deri	assoc	spec	attr	assoc	help
Game	1		1	4		2	
Board	1	1	2	5		1	6
Box	2	1	2	1		1	
Square	4		2	3	6		5
Solver			2	6			
Assign	1		1	3		1	
BdView	1		2	4		4	7
BdModel			1			3	
Activity	1		4	3			9
Total	11	2	17	29		12	27
			30				74

In our specification and design models, there are nine classes with ten associations among them. All nine classes appear in the implementation with total 30 fields and 74 methods (without counting constructors), and Java source code consists of 1279 lines[2]. Among the 30 fields, more than half (57%; 17/30) are generated from associations. This is generally a good sign for an object-oriented program. It indicates that there exists a significant structure in the program. It may also mean that the functionalities of the program are well distributed among the classes. Indeed, the method *spec* column shows that operations are well distributed among the classes. One interesting observation is that only 39% (29/74) of methods are from the operations explicitly specified in the model. The rest are from attributes, associations, and common subexpressions of operation specifications (helpers). The percentage will decease further with getter and setters for such attributes as *size*, *x*, and *y* of Box and Square classes that are currently translated to final fields. This means that one needs to focus on specifying and designing only interesting, core operations that account for less than 40% of operations, and the rest of the operations -- secondary, uninteresting or helper operations, which are not even specified as operations in the model -- can be generated automatically. In sum, a significant amount of code can be generated automatically.

[1] However, two Android framework-related classes BoardView and MainActivity required manual coding work; their behavior are not completely specified in the model.

[2] About 43% (553 lines) of source code are for two Android framework –related classes: BoardView and MainActivity.

V. Findings and Discussion

A. Code Generation

There is an opportunity for generating a significant amount of code automatically from models. Besides code that can be translated from constraints and behavioral state machines, operations can be derived on a need basis from the static structures expressed in class diagrams, e.g., operations for accessing attributes and traversing associations. In our app, for example, more than 60% of operations are of these and other helper operations mechanically generated from the models. It is also possible to automatically generate platform-specific code from models. An example is code to preserve the state of an app. Remember that when an event like screen orientation change occurs, an activity is destroyed and a new instance is created by the system. One way to preserve an activity's state is to use the Singleton design pattern as we did in Section IV. A new Singleton class with operations to store an activity's state and retrieve the stored state can be generated from a model, and calls to these operations can be added to appropriate framework methods such as *onPause*() and *onResume*(). Thus, one only needs to indicate in the model the attributes whose states have to be preserved, say, using a custom stereotype; such a stereotype can be defined in a UML profile. One key benefit of generating these kinds of code automatically is that one can focus on specifying and designing only more meaningful and interesting operations so called business logic or functional core. Another benefit is that one does not have to consider or even express in the model the detailed design or implementation decisions such as the visibility of features and the navigability of associations. The resulting model is more readable, as it doesn't distract the reader with unimportant design or implementation details.

We learned that the process of creating a precise model provides an opportunity for refactoring the model. In a sense, it forces one to review and evaluate the current model and thus refine and improve it. This happens naturally as part of formulating and writing constraints. For example, one functionality of the BoardView class is to let the user select a square to enter a new number or delete an existing one. In our initial design, we introduced an attribute in the BoardView class to store the selected square. We also introduced a corresponding attribute in the Game class to remember the selected square throughout the app's lifecycle, e.g., when the app is destroyed and recreated. We immediately noticed a deficiency of this design when we started formulating the behavior of the activity

lifecycle methods such as *onPause*() and *onResume*(). Since all such attributes have to be stored in and restored from the game object one by one, the design leads to long, unstructured constraints. We refactored our initial design by introducing a view-specific subclass of the Board class named BoardModel to remember view-specific data (see Section III.B). This produces not only a clean and extensible design but also simple specifications for the *onPause*() and *onResume*() methods. In fact, no additional constraint is needed because it is done automatically by the singleton game object composed of a board.

A precise model also facilitates evolution of an application. We made several enhancements to our app along with options to enable and disable new features: displaying all permitted numbers for empty squares, undoing and redoing assignments of numbers to squares, and the calculation of uniquely determined numbers for a set of squares (see Figure 11). The first enhancement didn't require a significant change in our model. The others required new model classes (e.g., stacks to keep track of undoable/re-doable actions), new UI elements (e.g., undo/redo buttons), Android specific classes (e.g., PreferenceFragment and its activity), and changes to existing classes. Our models provided us a good guidance in developing these enhancements by allowing us to identify the required change along with its impact in the models and the source code as well, e.g., modification of specifications, modification of the design to satisfy the modified specifications, and consequent modification of the source code. The preciseness of the model helped us to come up with an extended design with minimal change in order to support the enhancements.

Figure 11. Enhanced UI

One potential concern for automatically-generated code, however, is its runtime performance. Since Android devices are resource-constrained in storage capacity and battery lifetime, performance is always a concern in developing an app [24]. It is even said that identifying an app's performance bottlenecks and addressing them is critical to the success of the app [20]. For example, Android has a memory conservation mechanism known as Low Memory Killer (LMK) that, upon shortage of memory, starts killing background and inactive processes to reclaim their memory [28]. Another concern is that a performance problem hardly show up in the model. It is revealed only after the model is transformed to code and executed. In fact, we encountered such a performance problem in one of our operations whose constraints was directly translated to Java 8. The *Solver*::*pickSquare*() operation picks a square to fill in a number (see Section III.C), and its OCL specification and translated code are shown below.

```
context Solver::pickSquare(): Square
post: result = if c->size() > 0 then c->any(true) else null endif
  where Square::isPickable():Boolean = not hasNumber()
    and permittedNumber()->size() > 0,
  c = board.squares@pre->collect(s| isPickable() and
      board.squares@pre->forAll(isPickable() implies
        permittedNumbers()->size()
        >= s.permittedNumbers()->size())

private Square pickSquare() {
  List<Square> c = board.squares().stream()
    .filter(this::isPickable).collect(Collectors.toList());
  Square result = null;
  if (c.size() > 0) {
    int m = c.stream().mapToInt(s ->
        s.permittedNumbers().size()).min().getAsInt();
    c = c.stream().filter(s-> s.permittedNumbers().size() == m)
        .collect(Collectors.toList());
    result = c.get(random.nextInt(c.size()));
  }
  return result;
}

private boolean isPickable(Square s) {
  return !s.hasNumber() && s.permittedNumbers().size() > 0;
}
```

As said in the previous section, we use Java 8 features such as lambda expressions and the Stream API to generate code from OCL constraints. The OCL collection iterators such as *collect* are mapped to the Java stream operations such as *filter* and their argument expressions to lambda expressions. The code is a direct translation of the postcondition except for some performance improvements; e.g., in order to pick a square that has the least number of permitted values, it calculates the smallest size of permitted values of all squares and stores it in a local variable instead of comparing a candidate square to all other squares. The above code, however, failed to create a new game. There were too frequent garbage collections (see Figure 12), suspending all threads several times and eventually closing the app.

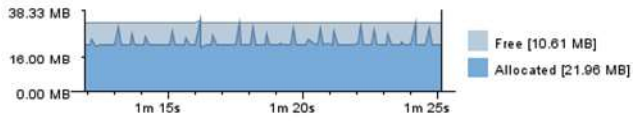


Figure 12. Pattern of garbage collection events

We rewrote our initial code to make it more memory efficient by reducing the number of garbage collections. Since stream operations create hidden objects and cause significant memory overheads [10], we refactored our code to remove them as shown below.

```
private Square pickSquare() {
  Square result = null;
```

```
  int min = Integer.MAX_VALUE;
  for (Square s: board.emptySquares()) {
    int c = s.permittedNumbers().size();
    if (c == 0) { return null; } // not solvable!
    if (c < min || (c == min && random.nextBoolean())) {
      min = c; result = s;
    }
  }
  return result;
}
```

This new code fixed our problem. When a new game is created, it triggers only one or two garbage collection events without causing all threads be suspended. However, we soon learned that the real cause of our problem was not the use of streams though it aggravated the problem. Our new code fixed the problem accidently by the way it was written -- i.e., a case analysis on the number of permitted values. When there is an empty square with no permitted number, it returns null. Thus, it forces backtracking as soon as the board becomes unsolvable without performing sort of exhaustive search for other empty squares. When we made this fix to our original code, it worked although its memory performance was not as good as the new code. We revisited our specification of the *Solve*::*pickSquare* operation to admit only this more efficient code as shown below.

```
result = if board.squares@pre->exists(not hasNumber() and
  permittedNumber() = 0) or c->size() = 0  then null else …
```

We learned that developing a good model is an iterative process. As shown above, coding can be an effective way to find a problem in a model and improve it. It is difficult to verify and validate a model alone while it is often rather straightforward to generate code from a specification and to observe its runtime behavior. Unlike the above example, however, it doesn't have to be a performance-related issue. As an example, consider the *Solver*::*fillDetermined*() operation from Section III.C.

```
context Solver::fillDetermined()
post: board.squares->forAll(not hasNumber()@pre and
  permittedNumbers()@pre->size() = 1 implies
  number = permittedNumbers()@pre->any(true))
```

It is supposed to fill all the empty squares that have a single permitted number. The specification looked good to us though there is some incompleteness (see below). However, when we ran the generated code, the solver behaved strangely. We were able to track the cause to the *fillDetermined*() method, whose code is a simple, direct translation of its specification and is indeed correct (see Section IV). There was a problem in the specification itself. Missing in the specification is a key constraint asserting all the newly assigned squares have to be remembered for backtracking. We fixed our specification as follows.

```
context Solver::fillDetermined()
post: board.squares->forAll(s| not hasNumber()@pre and
  permittedNumbers()@pre->size() = 1 implies
  number = permittedNumbers()@pre->any(true) and
  history^append(new Assignment(s)))
```

We used the OCL message expression (^) to add the new constraint. Note that it is also possible to simplify the whole consequent of the implication (the *implies* operation) to *self^assign*(*s*). Ironically, while tracking the cause of the problem, we also came up with a more efficient algorithm. Filling a number in a square may make other squares to be "determined", and we can fill these squares too as shown below.

```
private void fillDetermined() {
  boolean[] found = { false };
  do {
    found[0] = false;
    board.squares().forEach(s -> {
      if (!s.hasNumber()
        && s.permittedNumbers().size() == 1)  {
        assign(s); found[0] = true;
      }
    });
  } while (found[0]);
}
```

B. OCL

Is OCL a good notation for constructing a precise model for MDD approaches? Our finding is that it is expressive and reasonably good for writing single-state assertions such as class invariants, derived attributes and associations, and behavior of query operations. However, it is hard to write complete specifications in OCL for update operations and constraints involving multiple states, often called history constraints. For example, the specification of the *Solver*::*fillDetermined*() revised in the previous subsection is still incomplete. It constraints only the *number* attributes of those squares that are empty and have a single permitted value in the pre-state. What should be the new values of other attributes? What about other squares of the board or the board itself? What is missing in OCL is a built-in language construct for specifying the so-called *frame axiom* that essentially says "and nothing else changes" [2]. Some common but potentially imprecise approaches are to assume that only those objects mentioned in the postcondition are allowed to change, and objects that are not specified to change in the postcondition do not change [17]. We also found that it is often more intuitive and straightforward to design a step-by-step algorithm than to write pre and postconditions for an operation. For example, instead of writing an OCL constraint, we drew a behavioral state machine to express the behavior of the *Solver*::*backtrack*() operation that does backtracking. In general, when a constraint involves side effects on a series of hidden internal states, it is hard to formulate it because, besides the frame problem, there is no direct way to

refer to the hidden internal states to order or accumulate the side effects. Another example is the improved version of the *fillDetermined*() operation in the previous subsection whose behavioral machine produces a more concise and clearer description. We also learned that OCL are not effective in specifying the behavior of Android framework classes and their operations involving user interactions or collaborations among objects (see below).

We found that OCL message expressions are useful in specifying certain behavior of operations. One common use is to specify in a postcondition that a certain interaction should happen, e.g., invoking a callback operation or notifying an event to an observer. The OCL *hasSent* (^) operator allows one to specify this elegantly at a higher level of abstraction, i.e., without worrying about the state change due to the interaction. As described in Section III.B, for example, the *BoardView*::*onTouchEvent*() operation should notify to its observers when a square is touched and selected, and this behavior is specified as: *listeners->forAll*(*l* | *l*^*selected*(*s*)), where *s* is the selected square. In fact, it is impossible to specify the state change caused by the callback operation such as *selected* in this example because each observer may implement it differently and thus have different behavior. Another common use of the OCL message expression in our specifications is when we know how to achieve the required state change but specifying it is too involved or worthless. For example, the *BoardView*::*onTouchEvent*() operation has to refresh its display when a square is selected (see Section III.B). This behavior can be succinctly specified as: *self*^*invalidate*(); the *invalidate* operation is an Android framework method to force a view to draw itself. As shown above, judicious use of the message expressions in postconditions can improve the clarity and readability of the constraints. However, one has to be cautious when both query expressions and message expressions are used in a single constraint because the specified message sending may happen in any intermediate state, not necessarily in the final state. As an example, consider a constraint, *board.square = s and self*^*invalidate*(), where *s* is a new selection. This constraint is from the postcondition of the *BoardView*::*onTouchEvent*() operation, and the intention is to assert that the operation sets *s* as a new selection and then refresh the display. What is specified, however, is loose in that the *invalidate* message may be sent before the new selection is set; it may happen at any time during the execution of the operation.

We often extended the vocabulary for writing constraints, given by OCL and the class diagram. For example, to specify the behavior of the *Game*::*init*() operation, we introduced a new query operation named *isSolvable*() that tests whether a board configuration has a solution or not (see Section II.A). For this, we used the OCL *def* constraint, in which a helper attribute or operation is defined. We found that this feature of OCL is very useful in writing constraints partly because unlike the *let* expression it enables reuse of attributes and operations in multiple places. We believe that, like specification-only features of other formal specification languages such as JML [6], such attributes and operations do not have to be implemented[3], and thus they can be used to write constraints at a higher level of abstraction than given by the underlying UML models like class diagrams. We also introduced locally scoped functions (query operation) in our own extension to OCL, the *where* clause (see Section II.A).

During the design, we often encountered certain aspects of the underlying UML model that are difficult or simply worthless to formulate. Most of these are Android-specific detailed designs, e.g., specifications of operations such as *onCreate* and *onDraw* that override methods inherited from Android framework classes. Our approach was either to write constraints for only those aspects that are doable in OCL and worthwhile to do so or to focus on interactions (i.e., messages to be sent) by abstracting away from the required state changes. Thus, our specifications are partial or incomplete, lacking the parts that are not formalized. One desirable feature of OCL is a way to include them in a constraint – even if they are not formally written – to make the constraint complete. A construct similar to the *informally* expression of JML [19] is desirable to escape from formality and to combine formal and informal texts in a single constraint. For example, one can write a constraint like the following.

(* buttons from the current layout *)->forAll(**not** isEnabled())

The text enclosed in a pair of (* and *) is an informal expression in that its meaning is not given formally. Even if the constraint cannot be interpreted by a tool, it will be helpful for a human reader or programmer; it is a lot better than completely omitting it.

As said earlier, the OCL message expression was useful to either specify the required collaboration among objects or abstract from the required state changes. For example, the *BoardView*::*setBoard*() operation can be specified as follows.

context BoardView::setBoard(b: Board)
pre: b <> null
post: board = b **and** b^addChangeListener(?: ChangeListener)

The operation sets the board to be displayed and registers a listener to the board to refresh the display when there is a change in the board. However, the postcondition says nothing about refreshing the current display. It only states that the *addChangeLister* operation be invoked during the execution; it doesn't constrain the listener itself passed as the argument. A notation similar to the Java lambda expression [26] would be useful for constraining the callback operation. For example, the above conjunct may be rewritten as:

b^addChangeListener((s,v) -> **post**: self^invalidate())

[3] The OCL standard says that a «definition» constraint is identical to defining an attribute/operation in the UML with stereotype «OclHelper» with an attached OCL constraint for its derivation [22].

A lambda-like notation is used to specify the behavior of the callback operation. The two argument (*s*, *v*) represents the square whose number was changed and its old number as declared in the ChangeListerner interface. Of course, the lambda body consists of OCL constraints such as pre- and postconditions. The registered listener now should invoke the *invalidate* operation when a board change event occurs. A related improvement would be to provide a way to "quote" or refer to another constraint, e.g., the postcondition of another operation [15]. It would allow one to assert a condition or state change stated in another constraint without duplicating it. We believe notations like these be very useful for modeling and specifying Android apps. Android apps tend to become more complex reactive systems, constantly reacting on inputs from user interfaces or sensors as well as communicating with different network protocols. Thus, it is essential to be able to specify these interactions precisely in the model.

We can also image Android platform-specific support for writing OCL constraints. One feature of the Android platform is its use of UI layouts written in XML. An Android activity may be associated with multiple layouts, e.g., one for portrait mode and another for landscape. One handy feature would be an abstract way of referring to views contained in the current layout, e.g., all the buttons or buttons satisfying a certain property. For example, the following expressions can be used to retrieve all buttons or a button with a label 'X'.

```
layout->select(oclIsKindOf(Button))
layout.button->any(text = 'X')
```

Once a suitable abstraction of a layout is determined (e.g., a set of views or a composition of views), a derived attribute or query operation, say *layout*, may be introduced to the Android Activity class so as to be inherited by all user-defined activity classes.

VI. Conclusion

We performed a small case study of developing an Android app by applying the key ideas of MDD – creation of precise models and code generation. Our findings are mixed. There are of course obvious benefits of creating precise models. By writing OCL constraints, for example, one is in fact examining and evaluating one's models constantly even though one may not realize it; e.g., a long or complicate constraint may indicate a deficiency in one's model. Thus, it is more likely that one produces a better model in the end. Even if it is done manually, we generated a significant amount of platform-neutral and Android-specific code from our model, including functioning code derived from OCL constraints and behavioral state machines. An interesting result is that only 39% of methods are from the operations explicitly specified in the model. The rest of the operations are from attributes, associations, and common subexpressions (helper operations) of constraints, all of which are derived straightforwardly. Thus, one benefit of MDD is that it allows one to focus on specifying and designing only important and interesting operations; one does not have to consider or even express in the model uninteresting detailed design or implementation decisions such as the visibility of features and the navigability of associations. Since Android devices are resource-constrained in storage capacity and battery lifetime, however, one problem with automatically generated code is its performance, especially memory efficiency[24]. In fact, we encountered such a problem in one of our operations whose postcondition is directly translated to Java 8. The generated method caused so frequent garbage collection that it suspended all threads several times and eventually closed the app. As in our case, performance problems seldom show up in the model and thus hard to detect during the design; they are revealed only when the generated code is tested. A more fundamental question regarding performance is that specifications are generally written with clarity in mind, not for efficiency. Can the code generated from such specifications be efficient on Android?

Regarding the use of OCL, we learned that OCL can be an effective notation for writing constraints involving a single state, e.g., invariants, derived attributes and associations, query operations, and preconditions of update operations. However, it lacks expressiveness for writing complete specifications of state changes as well as being precise on the required interactions in postconditions. Android apps are becoming highly interactive and more complex reactive systems. For example, 43% of our source code lines are for two Android framework-related UI and control classes; they required most of our manual coding work. It would be challenging to specify in OCL the rich interactions possible on the Android platforms abstractly and at the same time sufficiently detailed so as to generate efficient code.

Is MDD a practical approach for developing Android app? As we did in our case study, the key components of MDD such as precise models and code generations can certainly be incorporated into the development of Android apps. However, one needs to consider the effort as well as the skills needed to create precise models. Thus, it may not be such an attractive approach for developing typical Android apps like our Sudoku app. However, it may be possible to reap the benefits of MDD for a certain types of apps such as health-related apps (e.g., [4] [8]) that require high assurance in meeting functional correctness or satisfying appropriate safety or regulatory requirements.

References

[1] C. Avila, et al., Runtime constraint checking approaches for OCL, a critical comparison, *International Conference on Software Engineering and Knowledge Engineering*, July 1-3, 2010, pp. 293-398.

[2] A. Borgida, et al., "… And nothing else changes": the frame problem in procedure specifications, *15th International Conference on Software Engineering*, pages 303-314, Baltimore, MA, May 1993.

[3] G. Botturi, et al., Model-driven design for the development of multi-platform smartphone applications, *Specification & Design Languages*, Paris, France, September 24-26, 2013.

[4] M. Boulos, et al., Mobile medical and health apps: state of the art, concerns, regulatory control and certification, *Online Journal of Public Health Informatics*, 5(3):229, 2014. doi:10.5210/ojphi.v5i3.4814.
[5] A. Brown. Model driven architecture: principles and practice, *Software and System Modeling,* 3(4):314-327, December, 2004.
[6] Y. Cheon, et al., Model variables: cleanly supporting abstraction in design by contract. *Software-Practice & Experience*, 35(6):583-599, May 2005.
[7] Y. Cheon, et al., Checking Design Constraints at Run-time Using OCL and AspectJ, *International Journal of Software Engineering,* 2(3):5-28, December 2009.
[8] Y. Cheon, R. Romero, and J. Garcia, HifoCap: An Android App for Wearable Health Devices, *8-th International Conference on Applications of Digital Information and Web Technologies*, volume 295 of Frontiers in Artificial Intelligence and Applications, pages 178-192, 2017.
[9] Eclipse Foundation, *Papyrus Modeling Environment*, available from http://www.eclipse.org/papyrus/, retrieved on October 18, 2017.
[10] A. Escobar and Y. Cheon, *Impacts of Java language features on the memory performances of Android apps,* Technical report 17-84, Department of Computer Science, University of Texas at El Paso, El Paso, TX, September 2017.
[11] E. Gamma, et al., *Design Patterns*, Addison-Wesley, 1994.
[12] R. B. France, et al., Model-driven development using UML 2.0: promises and pitfalls. *IEEE Computer,* 39(2):59-66, February 2006.
[13] J. Gosling, et al., *The Java Language Specification, Java SE 8 Edition*, February, 2015, Available from: https://docs.oracle.com/javase/specs/jls/se8/html/index.html.
[14] H. Heitkotter, T. A. Majchrzak and H. Kuchen, Cross-platform model-driven development of mobile applications with md2, *ACM Symposium on Applied Computing*, Coimbra, Portugal, March 18-22, 2013, pages 526-533.
[15] C. B. Jones, Systematic Software Development using VDM, Prentice Hall, 1990.
[16] F. A. Kraemer, Engineering Android applications based on UML activities, *International Conference on Model Driven Engineering Languages*, Wellington, New Zealand, October 16-21, 2011, pages 183-197.
[17] P. Kosiuczenko, Specification of invariability in OCL, *Software & Systems Modeling*, 12(2):415-434, 2013.
[18] K. Lano, *Model-Driven Software Development with UML and Java*. Course Technology, 2009.
[19] G. T. Leavens, A. L. Baker, and C. Ruby, Preliminary design of JML: A behavioral interface specification language for Java, ACM SIGSOFT Software Engineering Notes, 31(3): 1–38, March 2006.
[20] M. Linares-Vasquez, et al., How developers detect and fix performance bottlenecks in Android apps, *IEEE International Conference on Software Maintenance and Evolution*, September 2015, pages 352-361.
[21] T. O. Meservy and K. D. Fenstermacher, Transforming software development: an MDA road map, *IEEE Computer,* 38(9): 52-58, September 2005.
[22] Object Management Group, *Object Constraint Language*, version 2.4, Feb. 2014. Available from http://www.omg.org/spec/OCL/.
[23] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, 2nd ed. Addison-Wesley, 2004.
[24] D. Sillars, *High Performance Android Apps: Improve Ratings with Speed, Optimizations, and Testing*, O'Reilly, 2015.
[25] T. Stahl and M. Volter, Model-Driven Software Development, Wiley, 2006.
[26] R. Warburton, *Java 8 Lambdas, Functional Programming for the Masses*, O'Reilly, 2014.
[27] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*, 2nd ed. Addison-Wesley, 2003.
[28] K. Yaghmour, *Embedded Android: Porting, Extending, and Customizing*, O'Reilly, 2013.