Improving the Reliability of Mobility Applications

Néstor Cataño

nestor.catano@gmail.com

Abstract. The Android platform [13] was introduced by Google in 2008 as an operating system for mobile devices. Android’s SDK [15] provides a wide support for programming and extensive examples and documentation. Reliability is an increasing concern for Smart Phone applications since they often feature personal information and data. Therefore, techniques and tools for checking the correct behaviour of apps are required. This paper shows how the Event-B method can be used to reason and to verify the design of Android apps and how this can be used to document implementation decisions. Our approach consists in modelling the core functionality of the app in Event-B and using the evidence shown by the Proof Obligations generated to reason about the design and the implementation of the app. Although we don’t propose a novel approach, we prove that heavyweight Formal Methods (FMs) techniques with Event-B can effectively be used to support the development of correct Android apps. We present a case study in which we design the core functionality of WhatsApp in Event-B, we encode it over three machine refinements modelling basic functionality (chatting, deleting content, forwarding content, deleting a chat session, etc.), read and unread status of chat sessions, and implementation details, respectively. We report and discuss on underlying challenges in the design and implementation of the core functionality.

Keywords: Android, Event-B, Formal Methods, Mobile Applications, Refinement Calculus, Verification, WhatsApp

1 Introduction

Mobile phones have never been more popular and exciting for programmers. Android operating system [13] was first released by Google in October 2008, and it’s today an ongoing development effort led by the Open Handset Alliance (OHA) and Google. On September 2013 Google declared that more than one billion of Android devices were in use in the world, with over one million of Android apps published in the Google Play store. Android is built on top of an open source framework that features powerful libraries for mobile development, primarily designed for touchscreen smartphones and tablets. One of the most appealing aspects of Android is that it allows developers to use on a smartphone services like Gmail or Calendar which are typically used online. Reliability
of Smart Phone apps is a significant concern as they often manipulate personal information and data. This is exacerbated by the fact that the increase of manipulated data by Smart Phones also brings opportunities for privacy and security breaches. Therefore, sound techniques for the development of correct Android apps are required.

In spite of some misconception about their cost-effectiveness (results do not outweigh the investment in time and money), Formal Methods (FMs) have proven their potential to dramatically increase the quality of software systems as conceived and developed by the IT industry as shown in various case studies presented by the author [7,8,9,11]. This paper discusses about the use of heavyweight FMs techniques to check the design of Android applications. We use the Event-B formalism [4] to model the core functionality of the Android app. Event-B language is based on set-theory and predicate logic and it’s that what makes it of great value as it can be used by theorem provers to reason about underlying properties. We present a case study in which we verify the design of WhatsApp formally in Event-B using the Rodin IDE [3]. WhatsApp is a popular freeware instant messenger service for Smart Phones, available from Google Play Store (https://www.whatsapp.com/android/). We adopt a Software Engineering (SE) approach to reason about the design of WhatsApp, starting by a discussion about its software requirements, their formalisation in Event-B, the verification of underlying Proof Obligations (POs) with Rodin, and the use of POs for design and implementation decisions.

The contributions of this paper are two-fold. (i.) We demonstrate how the use of heavyweight FMs techniques with Event-B can be employed to verify the design of the core functionality of an Android app, and how implementation decisions can be made based on the evidence shown by the Event-B modelling of the Android app. (ii.) FMs techniques and languages such as Event-B have traditionally been used to develop critical systems. This paper is unique in showing how discrete mathematics and program refinement techniques with Event-B can effectively be used to support the development of Android applications.

2 Background

Event-B is based on Action Systems [6], a formalism describing the behaviour of a system by the (atomic) actions that the system carries out. An Action System describes the state space of a system and the possible actions that can be executed in it. Event-B models are composed of contexts and machines. Contexts define constants, uninterpreted sets and their properties expressed as axioms, while machines define variables and their properties, and state transitions expressed as events. The initialisation event computes the initial state of a machine. An event is composed of a guard and an action. The guard (written between keywords where and then) represents conditions that must hold in a state for the event to trigger. The action (written between keywords then and end) computes new values for state variables, thus performing an observable state transition. If the system reaches a state where no event guard holds, it halts and is said
to have *deadlocked*. There is no requirement that the system should halt, and indeed, most Event-B models represent systems that run forever. If halting is desired, the system can be modelled using *convergent* events that monotonically decrease the value of a natural number expression called the machine *variant*. Such events can only be triggered in states where the value of the *variant* is non-negative. Additionally, the system may reach a state where the guards of more than one event hold. In this situation, the system is said to be *non-deterministic*: Event-B semantics allows any of the events whose guards are satisfied to be triggered.

In Event-B, systems are typically modelled via a sequence of refinements. First, an abstract machine is written to verify whatever correctness and safety properties are desired. Refinement machines are used to add more detail to the abstract machine until the model is sufficiently concrete for hand or automated translation to code. Refinement Proof Obligations are discharged (proven) to ensure that each refinement is a faithful model of the previous machine, so that all machines satisfy the correctness properties of the original.

Figure 1 presents a simplified version of an Event-B model of WhatsApp. The *initialisation* event starting on line 12 gives initial values to the state (machine) variables. One further event is shown: one that is triggered when any user creates a chat session between two users $u_1$ and $u_2$ (Line 21 and 23). Guard $@grd2$ checks that the chat does not already already. Action $@act1$ modifies the chat to contain the pair of elements $u_1 \mapsto u_2$. Action $@act1$ makes the new added chat active for $u_1$. The construct:

$$\text{any } x \text{ where } G(s, c, v, x) \text{ then } v := A(s, c, v, x) \text{ end}$$

specifies a *non-deterministic* event that can be triggered in a state where the guard $G(s, c, v, x)$ holds for some bounded value $x$, sets $s$, constants $c$, and machine variables $v$. When the event is triggered, a value for $x$ satisfying $G(s, c, v, x)$ is non-deterministically chosen and the event action $v := A(s, c, v, x)$ is executed with $x$ bound to that value. The correctness condition of the event requires that, for any $x$ chosen, the new values of the state variables computed by the action of the event maintain the invariant properties of the machine.

### 2.1 Event-B Mathematical Notation

Event-B provides a full battery of set and relation notation. Figure 2 shows some of the Event-B mathematical notation used by Event-B. We use square brackets to apply (evaluate) a relation to (over) a set of elements as mentioned above. For instance, $r[s]$ applies relation $r$ to all the elements in set $s$. The result of $r[s]$ is a set of elements in the range of relation $r$. Event-B provides standard notations for set union, intersection, difference, etc. Symbol $\times$ denotes the cross product between two sets. The operator $\text{dom}$ returns the domain of a relation, and $\text{ran}$ its range relation. The expression $\text{id}[s]$ denotes the identity relation over a set of elements $s$. Applying the forward composition relation $q;r$ to an element $a$ in the domain of relation $q$ returns a set of elements calculated as the result
machine machine0 sees ctx0

variables user content chat active chatcontent

invariants
@inv1 user ⊆ USER
@inv2 content ⊆ CONTENT
@inv3 chat ∈ user ↔ user // chat sessions
@inv4 active ∈ user → user // active chat session

events
event initialisation
then
@init1 user := ∅
@init2 content := ∅
@init3 chat := ∅
@init4 active := ∅
end

event create-chat-session
any u1 u2
where
@grd1 u1 ∈ user ∧ u2 ∈ user
@grd2 u1 ↦→ u2 ⊈ chat
then
@act1 chat := chat ∪ {u1 ↦→ u2}
@act2 active(u1) := u2
end
end

Fig. 1. A simplified WhatsApp abstract machine in the Event-B language.
of applying $r$ to $q\{\{a\}\}$. When a relation $q$ is a function, $q\{\{a\}\}$ should be used as $q(a)$. The domain restriction relation expression $s \circlearrowleft r$ restricts the domain of a relation $r$ to (consider only elements in) a subset $s$ of its domain. The range restriction relation expression $r \circlearrowright s$ restricts the range of relation $r$ to consider only elements in a subset $s$ of its range. Domain (range) subtraction is defined similarly to domain (range) restriction, except that the elements in the set $s$ are disregarded rather than considered.

### 2.2 Event-B Relations and Functions

Event-B relations are encoded as a set of pairs. A relation $f$ with domain $A$ and range $B$ is denoted $f: A \rightarrow B$. If $f$ is a function defined for all values of $A$, we say that $f$ is a total function, and we write $f: A \rightarrow B$. If $f$ is defined for some values of $A$, we say that $f$ is a partial function, and we write $f: A \rightarrow B$. If $f$ is a function such as no element in the range of $f$ is associated with more than one element in the domain of $f$, then we say that $f$ is a one-to-one or injective function, and we write $f: A \rightarrow B$. If $f$ is a function whose range is $B$, we say that $f$ is an onto or surjective function, and we write $f: A \rightarrow B$. If $f$ is both one-to-one and onto, we say that $f$ is a bijection, and we write $f: A \rightarrow B$.

### 2.3 Rodin and EventB2Java

Rodin [3] is an Eclipse based platform that provides support to Event-B, for instance, for writing Event-B models, defining safety invariant properties, and for discharging POs using back-end provers. EventB2Java [10] is a plugin of the Rodin IDE. It generates Java implementations for Event-B programs. EventB2Java translates a *machine* as a Java class. In translating a machine, EB2Java not only considers the information provided by the machine, but also the *contexts* the machine sees. Refinement machines are translated in the same way as abstract machines since Rodin properly adds abstract machine components to the

| Syntax | Name | Definition | Short Form |
|--------|------|------------|------------|
| $q; r$ | forward composition | $\{ (x, z) \mid \exists y \cdot (x, y) \in q \land (y, z) \in r \}$ | $q; r$ |
| $\text{id} \{ s \}$ | identity relation | $\{ (x, y) \mid (x, y) \in s \times s \land x = y \}$ | $\text{id} \{ s \}$ |
| $s \circlearrowleft r$ | domain restriction | $\{ (x, y) \mid (x, y) \in r \land x \in s \}$ | $\text{id} \{ s \}; r$ |
| $s \leftarrow r$ | domain subtraction | $\{ (x, y) \mid (x, y) \in r \land x \notin s \}$ | $(\text{dom}(r) \setminus s) \leftarrow r$ |
| $r \circlearrowright s$ | range restriction | $\{ (x, y) \mid (x, y) \in r \setminus (x, y) \notin s \}$ | $\text{id} \{ s \}; r$ |
| $r \leftarrow s$ | range subtraction | $\{ (x, y) \mid (x, y) \in r \setminus (x, y) \notin s \}$ | $r \leftarrow (\text{ran}(r) \setminus s)$ |
| $r \oplus q$ | relational overriding | $\{ (x, y) \mid (x, y) \in q \lor ((x, y) \in r \land \exists z \cdot (x, y) \notin q) \}$ | $q \uplus (\text{dom}(q) \leftarrow r)$ |
| $r^\sim$ | inverse relation | $\{ (x, y) \mid (y, x) \in r \}$ | $r^\sim$ |

**Fig. 2.** Basic Event-B mathematical notation.
internal representation of the refining machine. Refining and extending events (defined using \textit{refines} and \textit{extends}, respectively) are translated in the same manner as abstract events. Each \textit{event} is translated to a separate Java class. The translation of each event includes an object reference to the machine class. The translation of a standard event includes a \texttt{guard evt} method that tests if the guard of the event \texttt{evt} holds, and a \texttt{run evt} method that models the execution of \texttt{evt}.

### 3 Software Development with Event-B

Software development with Event-B starts with the definition of an initial \textit{blueprint} of the system one wants to model. This blueprint represents the future system implementation. Blueprints give insight on some but not all the aspects of the future system. A blueprint then goes through a series of stages called refinements \cite{5}. A blueprint refinement adds details to the blueprint. Refinements provide a hierarchical organization of the blueprints. The design of the initial system blueprint and its subsequent refinements is based on the description contained on an existing software requirements document. Each stage of the organization of a blueprint serves a different purpose. At higher levels, blueprints are used to state key system properties. At lower levels, blueprints implement the system behaviour. It is crucial that the initial blueprint and its refinements are consistent with each other, and that they are coherent with respect to the system specification. A refinement step generates Proof Obligations (POs) expressed in predicate logic to assert that the blueprint refinement is a refinement of the blueprint. That is, POs guarantee that the blueprint and its refinements are models of the same system.

Event-B caters for two types of blueprint refinements, \textit{horizontal refinement} (discussed above) and \textit{vertical refinement} \cite{1}. Horizontal refinement is also called \textit{superposition} in literature. Horizontal refinements add state transitions to the system or enrich existing transitions. The horizontal refinement stage is complete when all the software requirements are considered in the model. Through horizontal refinement a blueprint (a machine) can:

- strengthen an event guard,
- add new event guards,
- add more actions to some events, or
- add more events.

Vertical refinement is \textit{data refinement}. It does not add more details to the system, but it transforms the model into something that can easily be implemented. For instance, vertical refinement can transform finite sets into Boolean arrays. A key aspect of a vertical refinement is the definition of a \textit{gluing invariant} that bridges the abstract state of the system to the concrete state of the system by stating properties of the combined behaviour of both state models. Although horizontal and vertical refinements can be combined together in a single refinement step, a final vertical refinement single step is typically realized with the aid of a code generation tool such as EventB2Java \cite{10,11,12}.
The definition of the most abstract machine above and all its refinements are based on an existing software requirements document. The Rodin tool provides support for Event-B and Event-B model refinement definition [5]. Rodin generates safety and consistency POs in each refinement stage. Rodin includes several semi-automatic theorem provers that provide users assistance with proof discharging.

4 WhatsApp Software Requirements

Software development with Event-B relies on the parachute strategy for software development. Development starts with an initial abstract blueprint of the system in Event-B, and then, as the paratrooper descends, more details become clearer to him, and so he's able to add them to the system description. There is no automated mechanism or magical recipe that tells us how English written functional or non-functional requirements must be ported to Event-B. Nonetheless, one can stick to strict guidelines for writing software requirements as described below.

Software requirements might be related to the static part of Event-B, its dynamic part, or both. One needs to model the context first, which is related to the static part of the machine. A machine context typically includes constants, sets, and axioms that are used in the abstract machine and its refinements to declare invariants that typeset the machine variables. Machine variables are the static part of a machine.

I elicited the software requirements by directly installing the version of WhatsApp that is available from Android's store. Requirements do not cater for the Web version of WhatsApp but only for its Smartphone version. They cater for the most basic functionality of WhatsApp. When one writes software requirements in Event-B one should write an abstract machine (model) first and then successively write refinement machines [5]. For each refinement machine Proof Obligations (POs) are to be discharged in the Rodin platform [3] to ensure that each machine is a proper refinement of the most abstract machines. Only once all the machines are written and all the POs are discharged one can consider the underlying system has completely been modelled.

Table 1 presents WhatsApp machines hierarchy, as they will be modelled in Event-B. The abstract machine observes basic functionality for chat sessions including the functionality for creating a chat session, selecting or un-selecting a chat, chatting, deleting content (text, video, photos), removing content, deleting a chat session, muting and un-muting a chat, and broadcasting and forwarding network content. The first machine refinement includes functionality to check whether chat content has been read or not. The second machine refinement adds implementation details, for instance, it represents content as a sequence (rather than a set) of content items. This is important for us because the graphical interface of a chat session is implemented as an ordered sequence of content items that reads from the beginning to the end. Additionally, it would help us state a property that says that for any chat session, the chat content as seen by
one of the two chat members reads exactly the same as it is seen by the other
chat member. This is a safety invariant property, it says that “some desirable
situation always holds, or, that nothing bad happens”. This safety property in
particular is not a property of the Smart Phone version of WhatsApp.

| Machine | Observations                          |
|---------|---------------------------------------|
| machine 0 | Basic functionality for chat sessions |
| machine 1 | Read and unread status                |
| machine 2 | WhatsApp’s implementation             |

Table 1. WhatsApp’s Event-B machines hierarchy

I present WhatsApp’s requirements as User Stories (US). Each software re-
quirement is checked against an acceptance criterion that has 3 main compo-
nents, a Given part that describes when the functionality may be triggered/executed
(which depends on the internal state of the system or program), a When part that
tells us when the functionality is to be executed (which depends on the user’s
decision), and a Then part that tells how the state of the system is changed
when the functionality changes. US are typical of Agile methodologies, yet I use
them here since their structure fit the structure of events in which the Given
part is encoded through event guards, the When part is the event itself that is
triggered, and the Then part is encoded via event actions. In writing the US, in
general, I try to keep myself away from the interaction user-interface and focus
on the core functionality that WhatsApp needs to provide. However, a simple
user-interaction may involve the working and interplaying of multiple core func-
tionality.

In what follows, Sections 4.1 and 4.2 present the basic functionality for chat
sessions (first row in Table 1). Section 5 presents the modelling of that basic
functionality in Event-B, and Section 6 discusses design and implementation
decisions related to the basic functionality of WhatsApp. Second row in Table
1 is not discussed in this paper. Section 4.3 presents WhatsApp’s functionality
for the third row in Table 1 Section 7 discusses its modelling in Event-B, and
Section 8 discusses related design and implementation issues.

4.1 Basic Functionality for Chat Sessions

US-01 describes the functionality for creating a chat session between Me and
Another-User. The chat may not exist already.
US-01 create-chat-session
Description: As a user, I want to create a chat session so that I can communicate with Another-User
Acceptance
Given: A chat session between Me and Another-User does not exist
Criterion: When: I decide to create chat session with Another-User
Then: Chat session between Me and Another-User is created

US-02 describes the functionality for selecting a chat session. The effect of having two Given conditions is the condition obtained as the conjunction of both.

US-02 select-chat
Description: As a user, I want to select a chat session so that I can start chatting with Another-User
Acceptance
Given: A chat session between Me and Another-User exists
Criterion: Given: A chat session between Me and Another-User is not active
When: I select a chat session with Another-User
Then: The chat session between Me and Another-User is made active

US-03 introduces the functionality used for Me to chat with Another-User. Sent content is made available for both users Me and Another-User.

US-03 chatting
Description: As a user, I want to send some content during a chat session with Another-User so that I can transmit some information
Acceptance
Given: Chat session with Another-User is active
Criterion: When: Content is produced and sent by Me
Then: Content is made available to Me as well as to Another-User

WhatsApp implements two different behaviours for erasing exchanged content: “Remove For Me” and “Remove For Everyone”. If the sender of the content wants to remove some content, he is offered the option to remove it from his chat or to remove it from his chat and from the chat as seen by the user he’s chatting with. On the other hand, if the receiver of the content wants to delete it, he can only do it from his chat. These two behaviours are described by US-04a and US-04b, respectively. Erasing is always the type of subtle functionality difficult to encode in logic as one can easily break the machine invariants, for instance, if one erases content from one side of the chat and not from the other, one would
break any invariant on the equivalence of content read by both users of a chat session. One would then need to add an event guard (a Given condition) that prevents such behaviour or rephrase the invariant properly.

| US-04a | delete-content |
|--------|----------------|
| Description | As a user, I want to delete some content exchanged with another user during a chat session so that I unclutter my chat |
| Acceptance | Given: Content exists |
| Criterion | When: Me decides to delete the content he has received |
| | Then: Me’s content is deleted |

| US-04b | remove-content |
|--------|----------------|
| Description | As a user, I want to remove some content exchanged with another user during a chat session so that I unclutter my chat |
| Acceptance | Given: The content exists |
| Criterion | When: Me decides to remove the content he has sent |
| | Then: The content is deleted from Me and anyone to whom Me has sent the content |

Chat sessions and associated content can be deleted as well. What would it happen with the content seen by Another-User if the session between Me and Another-User is deleted. Will that content be deleted from Another-User as well? Deleting a chat session between Me and Another-User does not delete the content as seen by Another-User, regardless of who sent the content to whom, however, a remove-content US exists that deletes the content both ways.

| US-05 | delete-chat-session |
|--------|----------------------|
| Description | As a user, I want to delete a chat session with Another-User |
| Acceptance | Given: A chat session between Me and Another-User exists |
| Criterion | When: I select to delete the only active chat session |
| | Then: The chat session is deleted as well as its associated content |

When a chat session has been muted, communication between the two chat users is disabled both ways. Nevertheless, communication can be enabled later on.
Number | Invariant
---|---
1 | Users are uniquely identified throughout the system.
2 | Content is uniquely identified throughout the whole system.
3 | Chat sessions are uniquely identified throughout the system.
4 | A chat session relates exactly two users.
5 | Only one chat session maximum can be established between two users.
6 | A chat session between two users may have a set of associated content available to either or both of them.
7 | Content is associated to a chat session only if one the users of the session has sent the content to the other user or vice-versa.
8 | Active and muted are disjoint concepts. That is, it is never the case that the same system reaches a state in which user A muted user B and either is actively chatting with the other one.
9 | Chat sessions are not symmetric. That is, the fact that user A has created a chat session so as to chat with user B, does not necessarily mean that user B has a created session so as to chat with user A.
10 | Active chat sessions are no symmetric. That is, the fact that user A is actively chatting with user B does not necessarily mean that user B is actively chatting with user A.
11 | It is never the case that chat content exists associated to a pair of users for which no chat session exists.
12 | Several chat sessions can be created, but only one (or none) created chat session may be active per user.
13 | Chat communication with a muted user is no feasible: no content exchange is feasible from or to a muted chat.

**Table 2.** Local invariants for machine0

| US-06 | mute-chat |
| --- | --- |
| Description | As a user, I want to mute a chat session so that I can prevent communication with and from Another-User |
| Acceptance Criterion | **Given:** Chat session between Me and Another-User exists  
When: I select to mute a chat session  
Then: Chat session is muted and no communication from Me to the muted user or vice-versa is permitted |

US-07 is about to re-establish communication between two users of a muted chat. Only the user who muted the chat can unmute it.

| US-07 | create-chat-session |
| --- | --- |
| Description | unmute-chat |
| Acceptance Criterion | **Given:** Chat session between Me and Another-User is muted  
**Given:** I had muted the chat session previously  
When: I select to unmute a chat session  
Then: Communication between Me and Another-User is re-established |
US-08 and US-09 describe the situation whereby some content is sent to a group of users; forwarding a content requires that respective chat sessions between Me and the group of users exist, broadcasting creates new chat sessions if they do not exist already.

| US-08  | broadcast |
|--------|-----------|
| **Description** | As a user, I want to broadcast a content to a group of users so that I can communicate with all of them quickly |
| **Acceptance Criterion** | **Given**: Me wants to broadcast some content  
**When**: Me decides to broadcast the said content to Other-Users  
**Then**: The content is sent to Other-Users |

| US-09  | forward |
|--------|---------|
| **Description** | As a user, I want to forward a content to a group of users so that I can communicate with all of them quickly |
| **Acceptance Criterion** | **Given**: Me wants to forward some content  
**Given**: Respective chats between Me and Other-Users exist  
**When**: Me decides to forward the said content to Other-Users  
**Then**: The content is sent to Other-Users |

US-10 is the counterpart of US-02, unselecting a chat requires the chat to be active.

| US-10  | unselect-chat |
|--------|---------------|
| **Description** | As a user, I want to unselect a chat session so that I can chat with Another-User |
| **Acceptance Criterion** | **Given**: A chat session between Me and Another-User exists  
**Given**: A chat session between Me and Another-User is active  
**When**: Me wants to make session Another-User inactive  
**Then**: Chat session between Me and Another-User becomes inactive |

Notice that select-chat and unselect-chat could have been written without requiring the chat to be inactive or active, respectively. Thinking about their final encoding, the two events can eventually be encoded by adding a respective
checking if-condition that does nothing in case the condition is not fulfilled. On the contrary, by imposing those Given conditions in the US and eventually in their respective Event-B models I adopt a defensive style of modeling in which the system is required to be at the right state in order to be able to transition to another state.

4.2 Local Invariants for WhatsApp’s Basic Functionality

When modelling a system in Event-B in addition to the machine’s core functionality, one should write a series of safety invariant properties that describe the desirable behaviour of the system. Table 2 in Page [11] presents all the safety invariants that I have elicited for WhatsApp’s abstract machine.

4.3 Basic Functionality with Implementation Details

EX-02 offers a general description for the functionality for reading a chat session. Chat content is read in an orderly fashion.

| EX-02 | reading-chat |
|-------|--------------|
| Description | As a user, I want to read a chat session |
| Acceptance | Given: A chat session between Me and Another-User exists |
| Criterion | When: I read a chat session between Me and Another-User |
| Then | The content associated to the chat session between Me and Another-user is made available to Me |

5 Basic Functionality of WhatsApp in Event-B

We start by looking at the context of the abstract machine, which introduces two carrier sets, namely, USER and CONTENT that are used to typeset all the users registered in WhatsApp and all the content that it manipulates.

```
context ctx0
  sets USER CONTENT
end

machine machine0 sees ctx0
  variables user content chat active chatcontent muted
  // machine invariants...
  event initialisation
  then
    @init1 user := Ø    @init2 content := Ø
    @init3 chat := Ø    @init4 active := Ø
    @init5 chatcontent := Ø    @init6 muted := Ø
  end
  // rest of machine events...
end
```
Two variables in our model implement the two first invariants in Table 2. The first variable stores the registered users and the second one the content exchanged.

\texttt{invariants}
\begin{align*}
@inv1 & \text{user} \subseteq \text{USER} \\
@inv2 & \text{content} \subseteq \text{CONTENT}
\end{align*}

Invariant 3 in Table 2 is implemented as an Event-B invariant that declares chat as a relation between users. Invariant 4 is implemented by the fact that chat is a binary relation. Having modelled chat as a set enforces the fifth invariant in Table 2, therefore, no pair of elements in a chat session is repeated.

@inv3 \text{chat} \in \text{user} \leftrightarrow \text{user} // chat sessions

Implementing invariant 6 in Table 2 requires a subtler analysis as it relates content, the sender and the receiver of the content. Variable chatcontent below introduces chat content. The variable is defined as a partial function with domain user (the person who sends the message) and range content \( \rightarrow \mathcal{P}(\text{user}) \), where content is the content sent and \( \mathcal{P}(\text{user}) \) is the set of users to whom the content has been sent. chatcontent is a partial function, therefore, it might be the case a user exists that has not chatted with any one. The range of chatcontent is again a partial function, therefore, it might be the case a user exists that has not chatted with some particular user. Since chatcontent and its range are functions, the set of users to whom user \( u_1 \) has sent some content \( c \) is uniquely represented as chatcontent\( (u_1)(c) \), given that \( u_1 \) exists in the domain of chatcontent and \( c \) exists in the domain of chatcontent\( (u_1) \). The set of users with whom \( u_1 \) has chatted is represented as ran(chatcontent\( (u_1) \)), and the set of content items sent by \( u_1 \) (to anyone) is represented as dom(chatcontent\( (u_1) \)), given that \( u_1 \) exists in the domain of chatcontent.

@inv4 chatcontent \in \text{user} \rightarrow (\text{content} \rightarrow \mathcal{P}(\text{user}))

Next, we proceed to encode invariant 8 in Table 2 which says that active and muted chats are disjoints. @inv5 encodes the set of active chat sessions; active is a partial function, hence, a user has one active chat session maximum (the “function” part), but it might be the case he has no active chat session at all (the “partial” part). @inv7 states that it is never that case an active chat session is not a chat session, and @inv8 states that it is never the case that a muted chat session is not a chat session, that is, elements from muted chats are taken from chats. @inv9 encodes invariant 8 in Table 2.
Invariants 9 and 10 in Table 2 state that chat and active sessions are not symmetric necessarily. This invariants are modelled by not imposing further constraints over chat and active. In other words, if we wanted them to be symmetric, we have needed to enforce further invariants in Event-B. Invariant 11 in Table 2 is implemented by @inv10 below. Expression chat[{u}] returns the set of users with whom user u is chatting.

@inv10 \forall u,c,s \in user \land c \in content \Rightarrow (u \mapsto c \mapsto s \in chatcontent \Rightarrow s \subseteq chat[{u}])

Invariant 12 in Table 2 is enforced by the fact that active is a function.

Event-B models are composed of a static part defining observations (variables, constants, parameters, etc.) of the system and their invariant properties, and a dynamic part defining operations (events) changing the state of the system. Definitions introduced up to now are all static, and the next definitions are the dynamic part of the abstract machine (machine0) of our model. Invariant 13 in Table 2 is dynamic. It requires us to add an event guard to every event that otherwise might modify chatcontent of a muted chat.

Next, we implement the basic functionality of chat sessions in Event-B. Event create-chat-session implements US-01. It creates a chat session for user u1 to chat with user u2. The Given condition in US-01 is encoded by guard grd1. Guard grd1 helps Rodin to infer the type of u1 and u2. The pair u1 \mapsto u2 is added to the set of existing chats. The content associated to the chat between u1 and u2 is empty. Notice that event create-chat-session does not create a chat for u2 to chat with user u1.

```plaintext
event create-chat-session // US-01
    any u1 u2
        where
            grd1 u1 \in user \land u2 \in user
            grd2 u1 \mapsto u2 \notin chat
        then
            act1 chat := chat \cup \{u1 \mapsto u2\}
            act2 active(u1) := u2
end
```

Event select-chat implements US-02. Act1 uses the relational overriding operator \(\oplus\) instead of the set union operator \(\cup\), in this way u1 can have an active chat session only with one user. Act4 implements a defensive style of programming as explained before. Act3 makes sure that a muted chat session is never active. Act1 typesets u1 and u2. Act2 implements the first Given condition in US-02, and guard Act4 implements the second one. Act1 uses
the overriding operator ⊕ instead of the union operator ∪ to make sure we don’t break @inv5 so that active remains a function. Had we added u1↦→u2 to active using the union operator ∪, we would have probably ended up with active mapping u1 to two different users. Rodin would have detected this mistake by generating an improvable Proof Obligation (PO).

```
event select-chat // US-02
any u1 u2
where
  @grd1 u1∈user ∧ u2∈user
  @grd2 u1 ↦→ u2 ∈ chat
  @grd3 u1 ↦→ u2 /∈ muted
  @grd4 u1 ↦→ u2 /∈ active
then
  @act1 active := active ⊕ \{u1↦→\}
end
```

Event chatting implements US-03 whereby user u1 chats with user u2. It implements the scenario whereby u1 sends some content c to u2. @grd2 encodes the Given condition. The first part of guard @grd4 typesets variable c and the second part requires it to be a fresh content. Because c is a fresh content, @act1 adds it to the set of contents. @act2 creates a chat instance for u2 ↦→ u1 in case it does not exist already. If it exists, chat remains unchanged as it is a set. This matches the actual behaviour of WhatsApp in which a chat window is created for u2 the first time a user u1 sends her some content. The second line in @act3 adds c to the existing chat content between u1 and u2. Notice that chatcontent(u1) remains a function after the assignment in @act3 since c is not in its domain.

```
event chatting // US-03
any u1 u2 c
where
  @grd1 u1∈user ∧ u2∈user
  @grd2 u1 ↦→ u2 ∈ active
  @grd3 u1 ↦→ u2 /∈ muted ∧ u2 ↦→ u1 /∈ muted
  @grd4 c ∈ CONTENT ∧ c /∈ content
  @grd5 u1 ∈ dom(chatcontent)
then
  @act1 content := content ∪ \{c\}
  @act2 chat := chat ∪ \{u2 ↦→ u1\}
  @act3 chatcontent := chatcontent ⊕
    \{ u1 ↦→ (chatcontent(u1) ∪ \{c ↦→ \{u2\}\}) \}
end
```

We present below the encoding of US-04a and US-04b, therefore, guard @grd2 verifies that the user u1 who deletes or removes the content is actively chatting with u2. delete-content uses the functional overriding operator ⊕ to
override u1’s chat content. It removes u2 from chatcontent(u1)(c) so that u2 no longer appears as having received content c from u1.

**event** delete-content // US-04a
any u1 u2 c
where
@grd1 u1∈user ∧ u2∈user
@grd2 u1→u2 ∈ active
@grd3 u1 ∈ dom(chatcontent)
@grd4 c ∈ dom(chatcontent(u1))
@grd5 u2 ∈ chatcontent(u1)(c)
then
@act1 chatcontent(u1) := chatcontent(u1) ⊕ {c ↦ (chatcontent(u1)(c) \ {u2}))
end

remove-content removes c from the domain of chatcontent(u1), therefore, c no longer appears as having been sent by u1.

**event** remove-content // US-04b
any u1 u2 c
where
@grd1 u1∈user ∧ u2∈user
@grd2 u1→u2 ∈ active
@grd3 u1 ∈ dom(chatcontent)
@grd4 c ∈ dom(chatcontent(u1))
@grd5 u2 ∈ chatcontent(u1)(c)
then
@act1 chatcontent(u1) := {c} ⊲ chatcontent(u1)
@act2 content := content \ {c}
end

Event mute-chat encodes US-06. It mutes the chat between u1 and u2; more concretely @act1 adds the pair u1→u2 to the set of muted chats. @act2 forbids a muted chat from being active. Alternatively, we could have added a guard @grd4 u1 → u2 \ active, but then this does not reflect the actual behaviour of the graphical interface of WhatsApp in which a user u1 can indeed mute a user u2 when actively chatting with her.
Event `mute-chat` implements US-06. It unmutes the chat between `u1` and `u2`. `@grd3` checks that user `u1` (who muted `u2`) is the only one who can unmute `u2`. `u2` is unique since `muted` is a function. `@grd2` is redundant: it can be deduced from `@grd3` and the fact that `muted \subseteq chat`.

Event `unmute-chat` implements US-07. It unmutes the chat between `u1` and `u2`. `@grd3` checks that user `u1` (who muted `u2`) is the only one who can unmute `u2`. `u2` is unique since `muted` is a function. `@grd2` is redundant: it can be deduced from `@grd3` and the fact that `muted \subseteq chat`.

Event `forward` below implements US-09 whereby user `u` forwards content `c` to a set of users `us`. Guards `@grd1`, `@grd2`, and `@grd5` typeset `u` and `us`. `@grd4` typesets `c`. `@grd6` checks that `u` indeed possesses chat sessions with every user member of `us`. Expression `muted[{u}] \cap us = \emptyset` checks that no member of the set `us` is part of the set of users that `u` has muted. Expression `muted[us] \cap \{u\} = \emptyset` checks that `u` has not been muted by any member of `us`. Body action `@act2` creates respective chat sessions for each member of the set `us` to chat with `u`. Action `@act1` overrides `chatcontent` to include content item `c` into the chat sessions between `u` and each element of `us`. Event `forward` itself does not encode a notion of order among the content items of a chat session, hence, at the abstract level as implemented by the abstract machine one cannot establish which chat content reads first or after another. Event `broadcast` for US-08 is implemented in a similar way to event `forward`. The major difference between their implementations is that guard `@grd6` for event `forward` is not included by event `broadcast`. 
**Event forward** // US-09

any u us c

where

@grd1 u ∈ user
@grd2 us ⊆ user
@grd3 muted[\{u\}] ∩ us = ∅ ∧ muted[us] ∩ \{u\} = ∅
@grd4 c ∈ content
@grd5 u ∈ dom(chatcontent)
@grd6 us ⊆ chat[\{u\}]

then

@act1 chatcontent := chatcontent ⊕ \{u ↦→ (chatcontent(u) ∪ \{c ↦→ us\})\}
@act2 chat := chat ∪ (us × \{u\})

end

Event unselect-chat implements US-10. It unselects the chat between u1 and u2 by dropping u1 ↦→ u2 from active.

**Event unselect-chat** // US-10

any u1 u2

where

@grd1 u1 ∈ user ∧ u2 ∈ user
@grd2 u1 ↦→ u2 ∈ chat
@grd3 u1 ↦→ u2 ∈ active

then

@act1 active := active \ \{u1 ↦→ u2\}

end

6 Design and Implementation Decisions Regarding machine0

Event create-chat-session. What are the consequences of making a1 ↦→ a2 active? Rodin discharge its POs automatically, hence create-chat-session is correct with respect to the invariants defined in machine0. The consequences of making or not a1 ↦→ a2 active are rather related with its inter-playing with other events, for instance, with events chatting and select-chat. If create-chat-session doesn’t make a1 ↦→ a2 active then select-chat should execute later on before start chatting. The analysis of the inter-playing in the execution of several events in an Event-B model can typically be performed using ProB [14]. This tool checks for deadlocks. It checks if after executing any event the system can always make progress or not.

Event chatting. The first decision to make is whether or not we want to add content c to the chat between u2 and u1 (in addition to the chat between u1 and u2). If we want to do so, we should extend the second line of @act3 with u2 ↦→ (chatcontent(u2) ∪ \{c ↦→ \{u1\}\}). Intuitively, adding this line means that the content c that u1 sends to u2 is not only seen by u1 but also by u2. However, if we choose to extend @act3 that way, Rodin provers would generate a PO henceforth
u2 must be in the \textit{dom}(\textit{chatcontent}) so that sub-expression \textit{chatcontent}(u2) is well-typed. This requirement can be solved by adding an event guard @\textit{grd}6 \( u2 \in \textit{dom}(\textit{chatcontent}) \). The downside of this solution is that @\textit{grd}6 does not hold the first time when \( u2 \) hasn’t sent any content to anyone (not just to \( u1 \)) yet. In other words, the first time that \( u1 \) chats with \( u2 \) no chat session \( u2 \mapsto u1 \) exists yet.

The above downside would suggest that one could add default chat content associations the first time that one creates \( u2 \) (or any user, in general). The event \texttt{add-user} below adds user \( u \) to the set of current users. Action @\texttt{act}2 adds default chat content associations for user \( u \) with respect to any existing content. The soundness of @\texttt{act}2 is corroborated by Rodin provers by discharging all the associated POs automatically; in particular, @\texttt{act}2 adheres to @\texttt{inv}4 in Page 13.

\begin{verbatim}
event add-user
    any u
    where
        @\texttt{grd1} u \in USER \setminus user
    then
        @\texttt{act1} user := user \cup \{u\}
        @\texttt{act2} chatcontent(u) := content \times \{\emptyset\}
end
\end{verbatim}

What would it happen with the association encoded by @\texttt{act}2 above the next time that we add (create) a new content item? Event \texttt{add-content} is shown below. For each and every existing \texttt{user}, @\texttt{act}2 associates the fresh content \( c \) to the empty set, in other words, content item \( c \) appears as been sent by the whole set of users \texttt{user} to anyone.

\begin{verbatim}
event add-content
    any c
    where
        @\texttt{grd1} c \in CONTENT \setminus content
    then
        @\texttt{act1} content := content \cup \{c\}
        @\texttt{act2} chatcontent := chatcontent \cup (user \times \{c \mapsto \emptyset\})
end
\end{verbatim}

Summing up on event \texttt{chatting}, if we wanted to add content item \( c \) to chat \( u2 \mapsto u1 \) in addition to chat \( u1 \mapsto u2 \), then we would incur into a computationally expensive task: we would need to associate \( \emptyset \) to every existing content item every time we add a user to the system, and we would need to associate every single user to \( \{c \mapsto \emptyset\} \) every time we needed to add a fresh content item \( c \). This type of analysis on the complexity of associating chat content to \( u2 \mapsto u1 \) is not very intricate, in general; this analysis can be performed through careful code inspection or testing. But, writing the formal specification of WhatsApp in Event-B forces one to do code-inspection, and having Rodin theorem provers...
ensures that all cases are considered when performing automatic checking of
Event-B specifications with Rodin, without having to put effort into writing
appropriate test scenarios.

Notice that expressing \texttt{@act3} in \texttt{chat} as below does not work since the
last overriding expression forgets about \texttt{chatcontent(u2)}, which amounts to delet-
ing it. This issue cannot be spotted by Rodin (in particular regarding invariant
\texttt{@inv4}) as the new association for \texttt{u2} would still be a partial function. This can
only be spotted by a domain expert who knows that she does not want her chat
to be deleted whenever a content is sent to her.

\texttt{@act3 \ chatcontent := chatcontent}
\begin{itemize}
  \item \{u1 \mapsto (chatcontent(u1) \cup \{c \mapsto \{u2\}\})
  \item \{u2 \mapsto \{c \mapsto \{u1\}\}\}
\end{itemize}

The final solution is to use a comprehension set expression to expre-
sing the new value of \texttt{chatcontent(u2)} as indicated in the last overriding expression be-
low. The downside of this solution is that this expression is not directly encoded
with sets, relations and their operators (domain restriction, domain subtraction,
inverse, etc.), which are, for instance, directly encoded into Java by Event-B
code generators like EventB2Java \cite{10,12,16}.

\texttt{@act3 \ chatcontent := chatcontent}
\begin{itemize}
  \item \{u1 \mapsto (chatcontent(u1) \cup \{c \mapsto \{u2\}\})\}
  \item \{cc,s \cdot u2 \mapsto \{cc \mapsto s\} \in chatcontent \lor (cc=c \land s=\{u1\})
       \land u2 \mapsto \{cc \mapsto s\}\}
\end{itemize}

\texttt{delete-content} and \texttt{remove-content} are two of the subtlest functionality of
WhatsApp in the sense that performing either of them can potentially break
invariants all around. Notice that \texttt{delete-content} does not remove \texttt{c} from chat \texttt{u2}
\mapsto \texttt{u1}. Under which circumstances should one add the following action to event
\texttt{delete-content}?

\texttt{@act2 \ content := content \setminus \{c\}}

If we add \texttt{@act2} to \texttt{delete-content}, Rodin will generate an unprovable PO.
The PO is related to \texttt{@inv4} in Page 14. One would need to demonstrate that
for any user \texttt{u} other than \texttt{u1} the range of \texttt{chatcontent(u)} is a partial function
from \texttt{content\setminus\{c\}} to \texttt{P(user)}, which is not possible because it might be the case
that \texttt{u} has sent (forwarded or broadcasted) \texttt{c} to another user previously.

A turn-around to this problem is to express \texttt{content} as below. However, to
calculate the value of \texttt{chatcontent} that way one should traverse \texttt{user} twice and
\texttt{content} once, which might be time consuming depending on the type of struc-
tures used to store \texttt{chatcontent} or to represent sets in general.

\texttt{@act2 \ content := \{cc,a,b,s \cdot a \in \text{dom(chatcontent)} \land
  cc \mapsto s \in chatcontent(a) \land b \in s \land \neg(a=u1 \land b=u2 \land cc=c) \mid cc\}}
Notice that if \( u_1 \) is chatting with \( u_2 \), and \( u_2 \) with \( u_3 \), and \( u_1 \) sends \( c \) to \( u_2 \), and \( u_2 \) sends \( c \) to \( u_3 \), calling `remove-content` with parameters \( u_1, u_2, \) and \( c \) does not remove \( c \) from the chat between \( u_2 \) and \( u_3 \), but only from the chat between \( u_1 \) and \( u_2 \) and between \( u_2 \) and \( u_1 \). For this reason `remove-content` does not implement a second action: @act2 \( \text{content} := \text{content}\setminus\{c\} \). To express the new value of \( \text{content} \) we can adopt the same approach as above and add the following line to the event `remove-content`.

@act2 \( \text{content} := \{cc,a,s : a \in \text{dom}(	ext{chatcontent}) \land cc \mapsto s \in \text{chatcontent}(a) \land \neg (a = u_1 \land cc = c) | cc\} \)

Event `forward`. It presents the same problem as event `chatting`. That is, if we additionally want to augment \( \text{chatcontent} \) with triplets \( u_2 \mapsto \{c \mapsto \{u\} \} \) for each \( u_2 \in \text{us} \), then we would need to add a second overriding expression like the one shown below. Again, set comprehension expressions are not implemented by tools like the EventB2Java Java code generator and hence that expression would need to be (machine) refined before it can be translated to a language like Java.

@act1 \( \text{chatcontent} := \text{chatcontent} \)
\[ \oplus \{u \mapsto (\text{chatcontent}(u) \cup \{c \mapsto \{u\}\})\} \]
\[ \oplus \{u_2,cc,s : (u_2 \mapsto \{cc \mapsto s\}) \in \text{chatcontent} \lor (cc = c \land s = \{u\} \land u_2 \in \text{us}) \land u_2 \mapsto \{cc \mapsto s\}\} \]

Event `mute-chat`. It encodes a *defensive* style of programming whereby only unmuted chats can then be muted. If the chat \( u_1 \mapsto u_2 \) is muted, then event `mute-chat` does not execute. However, notice that if we were to execute @act1 with a muted chat, then state variable `muted` would be remain unchanged as sets do not contain repeated elements (see @act1).

### 7 Extended Functionality with Implementation Details

*Machine refinement* is the mechanism that Event-B offers to extend or to detail the behaviour and the functionality of a machine. In Event-B, all the components of a refined, machine variable initialisations, guards and actions of a refining event defined using refines) or implicitly (invariants, guards and actions of a refining event defined using extends). We don’t give details here about `machine1` (the first machine refinement) but rather focus on the encoding of `machine2`. This machine adds implementation details to our Event-B model of WhatsApp. The goal of this machine is to leave the Event-B model into a way that is close to implementation for it to be translated to Java using the EventB2Java tool. Variables of a refined machine can appear in an invariant of a refinement machine. When this happens, the invariant is called a *gluing invariant* as it relates the state space of the abstract (refined) machine with the state space of the refinement machine. Until now, we have worked `content`, `users`, and `chat content` with an abstract data structure `set`. This structure was chosen for clarity rather than for its ability to be implemented in a computer. In their implementation in `machine2` we want to represent (some of) these structures with `sequences`. 
In Event-B, a segment of natural numbers can be expressed using the \( a..b \) notation, which defines the set of natural numbers between \( a \) and \( b \) inclusive.

\[
a..b = \{ x \mid x \in \mathbb{N} \land a \leq x \land x \leq b \}
\]

We can hence use the \( 1..n \) notation to model a sequence of type \( T \) and size \( n \) as a total function from \( 1..n \) to \( T \). By requiring \( \text{sequence} \) to be a total function we enforce it to have no holes in its domain.

\[\text{inv} \quad \text{sequence} \in 1..n \rightarrow T\]

Variable \text{contents} below encodes \text{content} as a sequence. \text{csize} represents the number of content items in \text{contents}. \text{content} is the type of \text{contents}. The domain of \text{contents} is \( 1..\text{csize} \), hence, when \text{csize} is 0, \text{contents} is empty. \@invr22 and \@invr23 are together a gluing invariant that relates \text{contents} with \text{content}.

\[
\begin{align*}
@\text{invr21} & \quad \text{csize} \geq 0 \\
@\text{invr22} & \quad \text{contents} \in (1..\text{csize}) \rightarrow \text{content} \\
@\text{invr23} & \quad \text{content} = \{ n,c \mid n \mapsto c \in \text{contents} \mid c \}
\end{align*}
\]

Next, we choose to implement \text{chatcontent} as the variable \text{screen}. This refined variable makes content-sent sequential, but not the sender or the receiver of the content. This is because we mainly use \text{screen} to display content exchanged in an orderly fashion, for which the pair of users do not need to be ordered, just the content items.

\[\text{invr24} \quad \text{screen} \in \text{user} \mapsto (\text{user} \mapsto \mathbb{P}(\text{contents}))\]

We present the refined version of event \text{chatting} below. Parameter \( k1 \) is the position at which content \( c \) is placed at \( u1 \)'s screen. For \( u1 \)'s chat content to be shown in an orderly fashion, \( k1 \) must be greater than any value in \( \text{dom}(\text{screen}(u1)(u2)) \) every time that \text{chatting} executes. Likewise, \( k2 \) is the position of content item \( c \) in \( u2 \)'s chat screen with \( u1 \). The last conjunction in \@grdr21 and \@grdr22 together ensure that \( \mathbb{P}(\text{contents}) \) in \@invr24 is a function. Action \@actr22 increases the number of existing content items. Action \@actr23 adds \( c \) at position \( \text{csize}+1 \) of sequence \text{contents}. \@actr21 adds \( c \) at position \( k1 \) (\( k2 \)) of \( u1 \)'s (\( u2 \)'s) chat screen with \( u2 \) (\( u1 \)).
event chatting extends chatting // US-03
any k1 k2
where
  @grdr21 u1 ∈ dom(screen) ∧ u2 ∈ dom(screen(u1)) ∧
  k1 \not\in dom(screen(u1)(u2))
  @grdr22 u2 ∈ dom(screen) ∧ u1 ∈ dom(screen(u2)) ∧
  k2 \not\in dom(screen(u2)(u1))
then
  @actr21 screen := screen ⊕
  \{u1 \mapsto (screen(u1) \oplus \{u2 \mapsto (screen(u1)(u2) \oplus \{k1 \mapsto c\})\})\},
  u2 \mapsto (screen(u2) \oplus \{u1 \mapsto (screen(u2)(u1) \oplus \{k2 \mapsto c\})\})\}
  @actr22 csize := csize+1
  @actr23 contents := contents \oplus \{(csize+1) \mapsto c\}
end

Event delete-content declares two parameters i and k for the position of content c in the sequences contents and screen, respectively. @grdr21 checks that contents(i) = c. The last conjunct of @grdr22 checks that c is displayed at position k of the chat screen between u1 and u2. @actr21 deletes k \mapsto c from screen(u1)(u2).

event delete-content extends delete-content // US-04
any i k
where
  @grdr21 i \mapsto c \in contents
  @grdr22 u1 ∈ dom(screen) ∧ u2 ∈ dom(screen(u1)) ∧
  k \mapsto c \in screen(u1)(u2)
then
  @actr21 screen(u1) := screen(u1) \oplus \{u2 \mapsto (\{k\} \triangleleft screen(u1)(u2))\}
end

8 Design and Implementation Decisions Regarding machine2

Event chatting. Some of the previous discussions in Section [4] are revisited in this section. We could opt to define in machine2 an additional chatting-first-time event that works the first time that user u1 sends any content to user u2 (hence u1 doesn’t exist in the domain of screen(u2)). Therefore, we redefine guard @grdr22 to reflect the case when u1 \notin dom(screen(u2)), and modify @actr21 not to refer to screen(u2)(u1).
event chatting-first-time extends chatting // US-03
any k1 k2
where
  @grdr21 u1 ∈ dom(screen) ∧ u2 ∈ dom(screen(u1)) ∧ k1 /∈ dom(screen(u1)(u2))
  @grdr22 u2 ∈ dom(screen) ∧ u1 /∈ dom(screen(u2)) ∧ k2 ∈ Z
then
  @actr21 screen := screen ⊕
  { u1 → (screen(u1) ⊕ { u2 → (screen(u1)(u2) ⊕ { k1 → c } ) } ) ,
    u2 → (screen(u2) ⊕ { u1 → { k2 → c } } ) }
  @actr22 csize := csize+1
  @actr23 contents := contents ⊕ { (csize+1) → c }
end

If one wants to show the content of a chat in an orderly fashion, then chatting should always be fed with indexes k1 and k2 that are greater than any previous index.

Event delete-content. Expressing the new value of contents without recurring to the use of a set comprehension expression is a difficult problem for concrete machine machine2 as well. By simply looking at the type given by @invr24 to screen, if content c is deleted from screen(u1)(u2), one would need to search for c in every element of type P(contents) associated to every pair of users in screen; if c is ever found, then contents remains unchanged, otherwise, contents is modified so that it becomes contents \ { c }. This is an algorithmic solution that can be implemented in a streamline programming language, which would be difficult to express in logic using sets, relations and operators over them only.

Event forward. We show below an unsuccessful attempt to implement event forward1. The event maps c with a new index k /∈ dom(screen(u)(u2)) for each user u2 in the set of users us. However, (i.) k ought to be the maximum element in dom(screen(u)(u2)) for each user u2, and (ii.) content c must be seen in screen(u2)(u) in addition to screen(u)(u2).

event forward extends forward
any ks
where
  @grdr21 u ∈ dom(screen) ∧ us ⊆ dom(screen(u))
  @grdr22 ks ⊆ N ∧ card(k) = card(us)
then
  @actr21 screen := screen ⊕
  { u → (screen(u) ⊕ { u2,k . u2 ∈ us ∧ k /∈ dom(screen(u)(u2)) |
    | u2 → (screen(u)(u2) ⊕ { k → c } ) } ) } |
end

The encoding below addresses issue (i.), hence, as for @grdr22, the user needs to feed event forward with a parameter k that is greater than any ele-

1 card returns the number of elements of a set.
ment in screen(u)(u2). Code generators like EventB2Java will implement event guards through if-conditions. However, coding with an if-condition would negatively affect the performance of the event implementation as the underlying checking would need to be performed every time the event is to be executed. One can therefore think of moving the checking outside the event and entrust it to a function that keeps (or calculates) a maximum k value for each pair of users u and u2 and then calling that function every time a k is needed.

```plaintext
event forward extends forward // US-09
any k
where
  @grdr21 u ∈ dom(screen) ∧ us ⊆ dom(screen(u))
  @grdr22 ∀ u2, i · u2∈us ∧ i∈dom(screen(u)(u2)) ⇒ k>i
then
  @actr21 screen := screen ⊕ { u ↦→ (screen(u) ⊕ {u2 · u2∈us ∧ i∈dom(screen(u)(u2)) ⇒ k>i}) }
end
```

Event broadcast. The same problems for forward apply to broadcast.

9 Conclusion

Formal Methods (FMs) will become more popular in the Smart Phone industry if techniques are developed and tools are implemented that provide support to Software Engineering practices and to the analysis of performance and correctness of mobile apps. The techniques presented in this paper are mainly related to correctness, but some of the discussions relate to performance as well. I consider paramount important to give developers mechanisms to analyse mobile apps way before they start thinking on their implementation or can make any decision about the use of any particular technology. Our approach to the analysis of the design of mobile apps relies on first writing the software requirements of the app as User Stories, and then formalising them directly in Event-B. Writing software requirements in Event-B demand a high level of formality from developers as one should write invariants and predicates in logic, and should use (semi-) automatic theorem provers to validate one’s understanding of the system that one has in mind. This paper abstract away from issues related to the use of Rodin to discharge Proof Obligations. Writing invariants requires certain discipline and training, but then, I judge that it’s the same kind of discipline and training a developer would require to appropriate and master every new technology of interest.

The EventB2Java tool generates Java class implementations for events. EventB2Java blindly translates event actions into Java, hence, if a set or relation expression occurs repeated several times in the right-hand side of an assignment, EventB2Java translates it several times to Java; the tool does not perform any kind of pre-processing. Nevertheless, the Java code can serve as a prototype implementation
and be used to animate and check the actual behaviour of the Event-B model in Java.

In addition to the approach presented in this paper to check the design of Android apps, one can use ProB \[14\] to check for deadlock conditions. For instance, the analysis performed in Section 6 for event chatting can be supplemented with the use of ProB to check for deadlocks in the interleaving of chatting with all the other machine events.

In what follows I give a list of functionality that though wasn’t included in my model of WhatsApp and that it’s worthwhile pursuing as future work as future work: (i.) Archiving chats. It consists in backing up a chat, making it inactive, and hiding it from the user. (ii.) Pinning a chat. This is a more interface-related requirement, it consists in moving a chat up in the list of existing chats so that the user does not need to scroll down to search for that chat in her chat list. (iii.) Contact lists. Chats can only be created out of a local contact list. (iv.) Copying a message or content. The message or content can be pasted thereafter. (v.) Time stamps. Users can check the time stamps of sent or received messages. (v.) Group of users. Users can create groups. These groups can be used as chat groups or be used in phone calls or video conferences. (vii.) Phone calls and video conferences. Users can place phone calls or make video conferences to a person or a group of persons.

References

1. Jean-Raymond Abrial. Faultless systems: Yes we can! IEEE Computer, 42(9):30–36, 2009.
2. Jean-Raymond Abrial. Modeling in Event-B: System and Software Design. Cambridge University Press, New York, NY, USA, 2010.
3. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. Software Tools for Technology Transfer, 12(6):447–466, 2010.
4. Jean-Raymond Abrial and Stefan Hallerstede. Refinement, Decomposition and Instantiation of Discrete Models: Application to Event-B. Fundamentae Informatica, 77(1,2):1–24, 2007.
5. Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. Fundam. Inf., 77(1-2):1–28, 2007.
6. R. Back and K. Sere. Stepwise Refinement of Action Systems. Structured Programming, 12:17–30, 1991.
7. Cess-Bart Breunesse, Néstor Cataño, Marieke Huisman, and Bart Jacobs. Formal methods for smart cards: An experience report. Science of Computer Programming, 55(1-3):53–80, March 2005.
8. Néstor Cataño, Ijaz Ahmed, Radu Siminiceanu, and Jonathan Aldrich. Lightweight verification of a multi-task threaded server. Science of Computer Programming, 80, Part A(0):169–187, 2014.
9. Néstor Cataño and Marieke Huisman. Formal specification of Gemplus’ electronic purse case study. In Lars-Henrik Eriksson and Peter A. Lindsay, editors, Formal Methods Europe (FME), volume 2391 of Lecture Notes in Computer Science, pages 272–289, Copenhagen, Denmark, July 22-24 2002. Springer-Verlag.
10. Néstor Cataño and Víctor Rivera. EventB2Java: A code generator for Event-B. In *Nasa Formal Methods (NFM)*, volume 9690 of *LNCS*, pages 166–171, Minneapolis, MN, USA, June 7–9 2016. Springer.

11. Néstor Cataño and Tim Wahls. A case study on code generation of an erp system from Event-B. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 183–188, Vancouver, Canada, August 2015.

12. Néstor Cataño, Timothy Wahls, Camilo Rueda, Víctor Rivera, and Danni Yu. Translating B machines to JML specifications. In *27th ACM Symposium on Applied Computing, Software Verification and Testing track (SAC-SVT)*, pages 1271,1277, Trento, Italy, March 26-30 2012. ACM.

13. Google Inc. The Android Platform. [http://developer.android.com/design/index.html](http://developer.android.com/design/index.html), 2017.

14. Michael Leuchel and Michael Butler. ProB: A Model Checker for B. In *FME 2003: Formal Methods, Lecture Notes in Computer Science*, Pisa, Italy, 2003. Springer-Verlag.

15. Rato Mejer. *Professional Android 4 Application Development*. Wrox, 2012.

16. Víctor Rivera, Néstor Cataño, Tim Wahls, and Camilo Rueda. Code generation for Event-B. *STTT*, 19(1):31–52, 2017.