
Graham Peter Williams
Dickinson College

Follow this and additional works at: https://scholar.dickinson.edu/student_honors
Part of the Graphics and Human Computer Interfaces Commons, and the Theory and Algorithms Commons

Recommended Citation

This Honors Thesis is brought to you for free and open access by Dickinson Scholar. It has been accepted for inclusion by an authorized administrator. For more information, please contact scholar@dickinson.edu.

by

Graham Williams

Submitted in partial fulfillment of Honors Requirements for the Computer Science Major
Dickinson College, 2016

Tim Wahls, Supervisor
John MacCormick, Reader
Michael Skalak, Reader

March 29, 2016
The Department of Mathematics and Computer Science at Dickinson College hereby accepts this senior honors thesis by Graham Williams, and awards departmental honors in Computer Science.

__________________________________________
Timothy Wahls (Advisor)                        Date

__________________________________________
Michael Skalak (Committee Member)              Date

__________________________________________
John MacCormick (Committee Member)             Date

__________________________________________
Richard Forrester (Department Chair)           Date

Department of Mathematics and Computer Science
Dickinson College

May 3, 2016
Abstract


by
Graham Williams

This project is a case study on formal verification of an Android application used for pain management. When it comes to healthcare applications, the consistency of the application is crucial, as it may affect the wellbeing of the user. The application will check for trends in the user reports of pain, stress, etc. and notify the user accordingly. The model that specifies these trends is defined in Event-B using an Eclipse-based IDE called Rodin. We used the automated and interactive theorem provers built into Rodin to verify that the model always gives the user consistent information. The Event-B model is translated from Event-B to Java using the EventB2SQL tool. The Java class generated from the model forms the core of our Android application. We have recently extended the EventB2SQL tool to generate Android user interface components, further facilitating the development of our application. Since the model is based on a formally verified Event-B model, we can say with confidence that the application will not output any harmful or misleading information to the user. This paper will further elaborate on the research we are doing in Event-B, EventB2SQL, and the advantages of formal verification for a healthcare application.
Acknowledgements

I would like to thank my thesis committee for taking the time and effort to support this project. Particularly I would like to thank my research advisor Professor Tim Wahls for the many hours of weekly, and sometimes daily help with the project. Without his consistent support, only minimal progress would have been possible. I would also like to thank My (Caroline) Nguyen Tra ’16 for her collaboration in the early stages of Event-B and her significant contributions to the EventB2SQL compiler.
Chapter 6: CONCLUSION ........................................................................................................... 37
  6.1 Conclusion ....................................................................................................................... 37

Chapter 7: REFERENCES ........................................................................................................... 38
  7.1 REFERENCES .................................................................................................................... 38
Chapter 1

Introduction

1.1 Introduction

The end goal of this project is an Android application that assists the user in managing their pain. The user inputs several different measurements at some consistent interval (e.g., daily, weekly) and the app checks for trends that warrant concern. When such a trend is detected, a notification or warning is sent to the user indicating that some action should be taken in response to the worrying data. A more detailed walkthrough of how the app works is provided in section 2.7.

The foundation of this project is a mathematical model. This model is created in an Eclipse-based IDE called Rodin using a formal method called Event-B (Abrial et al., 2010). Event-B is explained in more detail in section 2.2. The model is then translated to Java code using a code generator called EventB2SQL (Wang and Wahls, 2014). The resulting Java class is imported into an Android Studio project. This class facilitates the creation and management of a database corresponding to the original mathematical model. Essentially, with these tools, we are able to specify correctness parameters in Event-B and have those parameters upheld in a larger, more capable Android application.

The research areas of this project are a mixture of Computer Science, Mathematics, and Healthcare. More specifically, the research areas include Pain Management, Event-B, EventB2SQL, Android Development, Compilers, Code Generation, Automated Provers, and Formal Verification. The research question of
this project is whether or not EventB2SQL can be used effectively to translate an Event-B model to an Android application. It will also explore what tools would be useful in this translation, and what benefits are evident in using Event-B to develop a model with medical implications.

This project is especially interesting in its mixture of different fields. Event-B stands out in its ability to maintain specific conditions, or invariants. Due to these aspects of Event-B, it has traditionally been used for projects that need to uphold very specific but important conditions. The invariants of the model correspond to medical trends that affect the safety and wellbeing of the user, so formal verification of these invariants is a significant aspect of this project.
Chapter 2

Background

2.1 Background Introduction

The means through which the application was created are complex – there are many different processes that rely on even more technologies. Some are traditional Computer Science tools, like Eclipse or Android Studio. Others are more off the beaten path like Event-B and EventB2SQL. Each of these technologies and processes – along with several others – will be described in depth in the following sections of the Background chapter.

2.2 Event-B

Event-B is not a programming language, but rather a methodology for creating mathematical models. Consequently, it relies heavily on set theory (Abrial, 2010). These models are created in Rodin, an Eclipse-based IDE (Abrial et. Al, 2010). The two main components of Event-B models are machines and contexts. All other components are contained in one of these two. A machine sees a context, not the other way around. That is, the machine uses elements defined in the context in its own definition, whereas the context is defined independently of the machine.

2.2.1 Context

The context describes static information about the model that will not change. The three main components of a context are carrier sets, axioms, and constants. Carrier sets
hold objects of the same type, but the model will know nothing more about that type. In this way, they act like type parameters of a generic Java or C++ class. Axioms describe properties of constants. An example of how carrier sets and axioms are used in conjunction to represent an enumerated type is presented in section 4.2.

2.2.2 Machine

The machine describes and defines the dynamic behavior of the model. The three main components of a machine are invariants, variables, and events. A variable can be one of two primitive data types: Boolean or integer. A non-primitive variable can be a set, function, or relation. It is key to note that the state of the model is determined completely by the values of all of the variables.

Events change the state of the model. That is, they change the value of the variables. Each event has three components: parameters, guards, and actions. The actions are what actually change the state of the model. The guards of the event are the conditions (i.e., mathematical statements about the model) that must be true in order for the actions of the event to be executed. That is, if any of the guards are not true, then none of the actions will be performed, and the state of the model will remain unchanged. There is also an initialization event that acts as a default constructor. That is, it sets all of the variables’ values for the initial state of the model.

Invariants are mathematical statements that must be true for any possible state of the model. That is, given a state of a model, the execution of any event will result in a post-state of that model that does not violate any of the invariants. A simple example of a
traffic light system modeled by Event-B is described in section 2.3 and includes examples of several of the Event-B components described in this section.

### 2.3 Example Event-B Model: Traffic Light System

This section describes the implementation of a very simple Event-B model that represents a traffic light system. Please note that this is a description of a tutorial published by others. Also, keep in mind that the traffic lights described in this model have nothing to do with the traffic lights in my model (to be mentioned in later sections). Because it is so simple, there is only one machine and no context (and thus, no axioms, constants or carrier sets). There are two Boolean variables – cars_go and peds_go – indicating whether or not cars or pedestrians can cross the intersection respectively. The three invariants are shown in Figure 1. The first two just describe the types of the variables by saying that they are elements of the set BOOL (where BOOL is the set \{true, false\}). The third invariant is more relevant and meaningful. It says that the two variables should never be true at the same time for any state of the model. That is, the cars_go and peds_go can never both have the value true. This makes sense in context, as pedestrians and cars cannot cross at the same time (at least not over the same crosswalk).

![Figure 1](http://handbook.event-b.org/current/html/tut_final_model.html)

<table>
<thead>
<tr>
<th>INVARIANTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>o inv1: cars_go ∈ BOOL not theorem</td>
</tr>
<tr>
<td>o inv2: peds_go ∈ BOOL not theorem</td>
</tr>
<tr>
<td>o inv3: ¬(cars_go = TRUE ∧ peds_go = TRUE) not theorem</td>
</tr>
</tbody>
</table>

---

There are several events in this model, but only two are needed to demonstrate the different components of Event-B. The two events are \texttt{set\_peds\_go} and \texttt{set\_peds\_stop} – both shown in Figure 2. The purpose of these two events is to change the value of the \texttt{peds\_go} variable (i.e., \texttt{set\_peds\_go} assigns \texttt{true} to \texttt{peds\_go} and \texttt{set\_peds\_stop} assigns \texttt{false}). Note that \texttt{set\_peds\_stop} has no guards, because there is no risk of violating any of the invariants. The third invariant says two variables cannot be true at the same time, but the action of this event only sets a variable to \texttt{false}. The event \texttt{set\_peds\_go} does require a guard to make sure that the other variable (\texttt{cars\_go}) is not \texttt{true}. Section 2.4 will explain in detail why this guard is necessary and how it helps maintain the invariants of the machine.

\begin{figure}
\centering
\begin{verbatim}
\begin{enumerate}
  \item \texttt{set\_peds\_go: not extended ordinary} \\
       WHERE \texttt{\} \\
       \begin{enumerate}
         \item \texttt{grd1: cars\_go = FALSE not theorem} \\
         \begin{enumerate}
           \item \texttt{act1: peds\_go = TRUE} \\
         \end{enumerate}
       \end{enumerate}
  \end{enumerate}
\end{verbatim}
\end{figure}

\begin{figure}
\centering
\begin{verbatim}
\begin{enumerate}
  \item \texttt{set\_peds\_stop: not extended ordinary} \\
       WHERE \texttt{\} \\
       \begin{enumerate}
         \item \texttt{grd1: peds\_go = FALSE theorem} \\
       \end{enumerate}
\end{enumerate}
\end{verbatim}
\end{figure}

\textit{Figure 2}
2.4 Rodin and Verification

Event-B models are created in an Eclipse-based IDE called Rodin. Along with the basic Rodin application, additional software called Atelier B provers were installed for this project. These provers are used to formally verify that all possible states of a model will not violate any of its invariants. That is, no matter what events are called on what states of the model, the resulting post-state of the model will satisfy all of the invariants. For example, consider the set_peds_go event (Figure 2) and the third invariant (Figure 1) from the traffic light example (Section 2.3). The actions of this event set the peds_go variable to true, but they can only be executed if the guard is true. The guard claims that cars_go must be false, so you cannot set peds_go to true unless cars_go is false. The provers take this information and mathematically prove that set_peds_go will never cause invariant 3 (or any of the other invariants) to be violated. A more detailed example of using the provers is presented in section 3.5.

2.5 Event-B Code Generators

The essence of a code generator is a compiler. Compilers are usually thought of as behind-the-scenes tools that take in source code and output machine code to be executed by the CPU. Event-B code generators are similar to this idea of compilers in that they have a source or input and still output code. However, both the input and output differ from those of a traditional compiler – the input is an Event-B model rather than an executable language and the output is code for a programming language rather than machine code. Event-B generators take an Event-B model as input and output code that
runs in some programming language. The behavior of the generated code must be consistent with the Event-B model.

There are several existing Event-B generators. One example is EventB2Java (Rivera and Catano, 2014), a full-fledged Java code generator that also generates with JML specifications. Another similar generator is B2C (Rivera and Catano, 2014), which generates source code in C. The resulting code is ready to be compiled by C development tools. There is even a universal generator called EB2All (Mery and Singh, 2011), though it is still in its beta stage. Using this generator, one can translate an Event-B model to Java, C, C++, and C# with the same tool. For this project, we used EventB2SQL, which is described in detail in section 2.6.

2.6 EventB2SQL

The name “EventB2SQL” can be deceiving; EventB2SQL doesn’t generate SQL code. It actually generates (at least for the purposes of this project) Java code that creates and manages an SQL database using SQLite. (Wahls, 2015). Since Event-B models are defined in terms of relations, functions, and sets, it is relatively straightforward to represent an Event-B model with an SQL relational database. EventB2SQL has been used in the past for medical applications. Namely, it has been used to create a Medication Checker Android application (Wahls, 2015).

Each carrier set corresponds to a table in the database, and a generic type parameter is generated for each non-enum carrier set. Additionally, for each carrier set, there are methods for iterating over each object in the set and adding objects to the set. For each event, a method implementing the corresponding database transaction is created.
So whenever an event would affect the Event-B model, a database transaction (inside a generated method) will affect the SQL database in a manner consistent with the event. If the guards of the original event hold, the transaction is executed and the method returns true. Otherwise, the method returns false and the transaction is not executed.

Since every database transaction in the generated Java class corresponds to an event in the Event-B model, we know none of the invariants from the original model will be violated in the database. That is to say, because of the provers described in section 2.4 and the proven soundness of the translation (Wahls, 2016), we now have formal verification that these methods will not affect the database in any way that would violate the original invariants of the Event-B model. This is the power gained from starting with Event-B; we now have Java code and can guarantee that our Event-B mathematical invariants will not be violated.

2.7 Application Walkthrough

This project is a proof of concept, but the application is set up in such a way that it models a real application as closely as possible. This section describes exactly how an instance of the app would be customized and used, so that the rest of the paper is easier to understand.

The client of the application (not to be confused with the user) would be a physician. They would specify to the developer the types of trends they want to look for in patient data. For the purposes of this project, we have included three representative trends with hardcoded constants. If an actual application were created, the constants
could be changed trivially. Additionally different types of trends could be created with relative ease.

The user of this application would be a patient of the client (physician). They would have the application on their own device and input measurements daily. Whether or not the physician would see the data they entered was not considered for this project, as the application will not actually be used. The user could check their input data over time. More importantly, the user could check their warning-levels for each type of measurement. That is, for each type of input given, they would have a green (good), yellow (concerning), or red (requires intervention) warning-level. This way, without doing any calculations on their data, the user can easily determine what actions to take.
Chapter 3

Methods

3.1 Healthcare Research

Though the research question of this project is based on Computer Science technologies, significant research in the healthcare field was necessary for creating a realistic application. Two medical consultants were in consistent collaboration with us during the early development stages. One consultant led a project on the Pain Assessment Screening Tool and Outcomes Registry (PASTOR)\(^2\), which forms the basis for many of the variables and constants of this project. One example of how this project influenced the design of the model is the different measurements. We decided to use five different indicators of quality of life, rather than just one indicator of pain, as this was found to be the most modern way of handling pain management. That is, in the field of pain management, clinicians tend to measure other facets of the patients life, so that quality of life does not decrease as pain does. Overall, having direct contact with the consultants was extremely useful in getting first-hand, consistent information and understanding the reasoning behind it. With this information, the end application will be more reasonable and consistent.

3.2 Basic Pain Management Model

This section describes the model as it existed at the end of the fall semester. This model was complete and functional (though with less interesting functionality). One

\(^2\) [http://www.dvcipm.org/clinical-resources](http://www.dvcipm.org/clinical-resources)
limitation of the “basic” model was that the trends it checked and verified were trivial.
Part of the spring semester was used to create a more functional and appropriate model.
The remainder of section 3.2 is very similar to the Research Report. Section 3.3 describes
the current model.

The pain management tracks the measurements for one patient. There are
recordings of pain, sleep, mood, stress, and activity. All five measurements are checked
separately for worrying trends and a corresponding warning-level for each measurement
type is calculated after each transaction. This model used a constant function called
WARNING_MAP.

3.2.1 Basic Model: The Context

The entire context can be seen in Figure 3 below. The carrier set of this context,
the first five constants, and the first axiom are used to make an enumerated type. This is
described later in section 4.2. The other constant WARNING_MAP and the remaining
axioms are used to define a constant function. The uses of this function are to map levels
of pain to warning-levels or notification-levels. This will facilitate the end project in
sending notifications to the user when a trend warrants a concern.
3.2.2 Basic Model: Variables and Invariants

All of the variables and invariants are shown in Figure 4 below. The machine has eight variables. The types of the variables are defined by the invariants that end with the word “Basic”. Each of the recording variables (i.e., painRecording, sleepRecording, etc.) is a total function from (0..nextIndex – 1) to (1..5). This means that it maps all of the integers between 0 and nextIndex=1 inclusive to an integer between 1 and 5 inclusive. It is worth noting that nextIndex is the variable for the next available index of the recording variables (as they are added sequentially). The notification variable is defined to be a total function from the enumerated type MEASUREMENT to 1..3. It maps each of the constants (i.e., PAIN, SLEEP, MOOD,
STRESS, and ACTIVITY) to an integer between 1 and 3 inclusive. For this model, the integer between 1 and 3 inclusive corresponds to a warning-level: low, medium, or high.

The remaining invariants (all of the ones ending in “Low”, “Medium”, or “High”) are all the invariants corresponding to trends. Essentially, it checks the last recording added and makes sure that the image of the notification function under that recording has the appropriate corresponding integer (or warning-level).

**Figure 4**
3.2.3 Basic Model: Events

The pain management model contains three events: addRecording (Figure 5), editLastMeasurement (Figure 6), and an initialization event. All three events are fully verified to maintain the invariants. Section 3.5 describes the verification in more detail.

The addRecording takes six parameters. The first five are each a number between 1 and 5 inclusive (as indicated by the guards) and the last is just a timestamp for when the recording was entered. The actions of the event add the five values of each of the recording parameters to each variable corresponding to the type of measurement. Note that each recording variable maps nextIndex to the new recording and then nextIndex is increased. This makes the variables function as lists.

The editLastMeasurement event changes the values of the last recording of each variable. That is, it maps each recording variable at nextIndex – 1 to a new value. Note that its five parameters are the same as the first five of addMeasurement and that act2 is similar act6. This is because they do very similar things, just using different indices. One difference to note is that editLastMeasurement has a guard for nextIndex. This is because you cannot editLastMeasurement unless there is at least one recording in the current state. This is equivalent to saying that nextIndex is greater than or equal to 1, because it starts at 0 and is incremented by 1 each time a recording is added.
addRecording: not extended ordinary

ANY
  o  pain  
  o  sleep  
  o  mood  
  o  stress  
  o  activity  
  o  timeStamp  

WHERE
  o  grd10: activity £ 1.5 not theorem  
  o  grd1: pain £ 1.5  
  o  grd7: sleep £ 1.5 not theorem  
  o  grd8: mood £ 1.5 not theorem  
  o  grd9: stress £ 1.5 not theorem  
  o  grd11: timeStamp £ 1.31 £ 1.12 £ 2000.3000 £ 0.23 £ 0.59 not theorem  

THEN
  o  act1: nextIndex = nextIndex + 1  
  o  act5: notification = {PAIN ⇒ WARNING_MAP(pain), MOOD ⇒ WARNING_MAP(mood),
                          STRESS ⇒ WARNING_MAP(stress), SLEEP ⇒ WARNING_MAP(sleep),
                          ACTIVITY ⇒ WARNING_MAP(activity)}  
  o  act2: painRecording(nextIndex) = pain  
  o  act4: sleepRecording(nextIndex) = sleep  
  o  act6: moodRecording(nextIndex) = mood  
  o  act7: stressRecording(nextIndex) = stress  
  o  act8: activityRecording(nextIndex) = activity  
  o  act3: date(nextIndex) = timeStamp  

END

Figure 5
editLastMeasurement: not extended ordinary

ANY
  o pain
  o sleep
  o mood
  o stress
  o activity

WHERE
  o grd1: pain ∈ 1.5 not theorem
  o grd7: sleep ∈ 1.5 not theorem
  o grd8: mood ∈ 1.5 not theorem
  o grd9: stress ∈ 1.5 not theorem
  o grd2: nextIndex ≥ 1 not theorem
  o grd10: activity ∈ 1.5 not theorem

THEN
  o act1: painRecording(nextIndex - 1) = pain
  o act3: sleepRecording(nextIndex - 1) = sleep
  o act4: moodRecording(nextIndex - 1) = mood
  o act5: stressRecording(nextIndex - 1) = stress
  o act6: activityRecording(nextIndex - 1) = activity
  o act2: notification = {PAIN ⇒ WARNING_MAP(pain), MOOD ⇒ WARNING_MAP(mood),
                           STRESS ⇒ WARNING_MAP(stress), SLEEP ⇒ WARNING_MAP(sleep),
                           ACTIVITY ⇒ WARNING_MAP(activity)}

END

Figure 6
3.3 Current Pain Management Model

The purpose of the model is the same as that of the basic model. The pain management model keeps track of the measurements for one patient. There are recordings of pain, sleep, mood, stress, and activity. The user of the application enters measurements between 1 and 5 inclusive for each of these. All five measurements are checked separately for worrying trends and a corresponding warning-level for each measurement type is calculated after each transaction. The end application notifies the user of the warning-level using a traffic light UI for each different type of measurement. Note that this traffic light UI has nothing to do with the traffic light tutorial described earlier in section 2.3. At any given time, the warning-level for each type of measurement is green, yellow, or red. An example of how this works in the model is shown briefly in section 3.3.2. The types of trends that are checked to determine the warning-level differ depending on the type of measurement.

3.3.1 Current Model: The Context

The entire context can be seen below in Figure 7. It is simpler than the context of the basic model, as we found it easier to model the warning-levels with variables rather than the constant function WARNING_MAP. Note that there is one carrier set, five constants, and one axiom. All of these are used together to act as an enumerated type. That is, MEASUREMENT has five possible values: PAIN, SLEEP, MOOD, STRESS, or ACTIVITY. The next section describes how variables and invariants use this in their definition. Section 4.2 describes how the axiom, constants, and carrier set work together to make this function as an enumerated type.
3.3.2 Current Model: Variables and Invariants

All of the variables are shown below in Figure 8. The machine has 22 variables. The types of the variables are defined by the invariants ending in “Basic” (Figures 9 and 11). Note that many of the variables come in groups of five – one for each type of measurement. Specifically, there is a “recording”, “green”, “yellow”, and “red” variable for each type of measurement. One may think upon observation of this that there is a more efficient way to model the variables. Indeed, we could have modeled it with a function from \((\text{MEASUREMENT} \times 1..3)\) to \(\text{BOOL}\), but the proofs were made simpler by splitting up the variables.
Figures 9 and 10 show some of the invariants of the machine. Some variable types are simple and primitive, while others require more understanding. Some of the more complicated invariants can even include constants from the machine. The invariant labeled `painRecBasic` in Figure 9 line 29 defines the type of the `painRecording` variable to be a total function from integers onto integers. More specifically, it defines the domain of the function to be $0..\text{nextIndex} - 1$. Since we are working with a total function, this means that all integers between 0 and $\text{nextIndex} - 1$ inclusive are mapped to something in the codomain. In this case, the codomain is $1..5$. That is, all things in this function map to an integer between 1 and 5 inclusive. Figure 9 actually
shows that many variables are described using total functions with these domains and codomains.

Invariants can also define more complex properties. Recall from earlier that one main function of this application is to notify the user when their input measurements warrant some concern. Some of the invariants ensure that the warning-level is consistent with the data. One example is the warning-level invariant on line 45 of Figure 10 labeled `moodMed`. Remember from section 2.2.2 that invariants are mathematical statements that must be true for any state of the model. In this case, it is one implication conjoined with a logical equivalence. The implication is \((\text{nextIndex} = 0) \Rightarrow \text{false}\). This is essentially saying that there must be at least one measurement added before `moodYellow` can be true. The variable `nextIndex` is initiated to 0 and increments after each measurement, so if `nextIndex = 0`, then nothing has been added. The second portion of the invariant (that is, the logical equivalence) is more interesting. This portion considers the case where `nextIndex > 0`. It then checks to see if `moodRecording(nextIndex - 1) = 4`. The logical equivalence says that if this is true, then `moodYellow` is the only warning-level variable that is true. All together, this invariant says: “If a measurement has been entered and the most recent measurement of mood is 4, then the warning-level of the mood measurement is yellow, and vice versa”.

This moodMed is the simplest trend in the model: it just checks the last measurement and depending on its value between 1 and 5, specifies one of the three corresponding warning-levels. The stressMed invariant on line 54 of Figure 10 is defined in the same way. The “Low” and “High” invariants for these variables (i.e., moodLow,
moodHigh, stressLow, and stressHigh) are defined similarly. They also just check the last measurement: if it is less than or equal to 3, then it defines the green variable to be true. If the last measurement is 5, then it defines the red variable to be true.

The painMed and sleepMed invariants on lines 48 and 50 respectively are a bit more complicated. They compare the last measurement with the one before it. If it is the same (i.e., the pain or sleep hasn’t improved or worsened), then the respective yellow variable is defined to be true. The “Low” and “High” invariants for these variables are defined similarly. The “Low” checks to see if the recording has actually improved, while the “High” checks to see if it has worsened.

The activityMed (Figure 10, lines 56-59) is the most complex trend. It actually checks the average recording of the last three measurements. If the average is less than or equal to 4 and greater than 3, then the corresponding yellow variable is defined to be true. If the average is less than this range, then the activityLow invariant defines the green variable to be true. If the average is greater than this range, then the activityHigh invariant defines the red variable to be true. You can see these two corresponding invariants in Figure 11.
3.3.3 Current Model: Events

The pain management model contains two non-initialization events: addRecording and editLastMeasurement (Figure 14). Both are very similar to the basic model. The main thing that changes is the actions. This is because they correspond to invariants, which have changed significantly (as shown in the last section). The next paragraph describes how these actions uphold invariants.

The majority of the actions assign Boolean values to each of the warning-level variables (i.e., any variable ending in “green”, “yellow”, or “red”). In fact, act9 through act20 all have this purpose. Note that it varies depending on the variable and the warning-level, as different trends are checked for different measurement types. Line 128
of Figure 13 shows an action that corresponds to the earlier invariant example (moodMed). It states that the moodYellow variable is assigned a Boolean value corresponding to the statement mood = 4. That is, it is assigned true if the last measurement input was 4, and false otherwise. This upholds the moodMed invariant described earlier in section 3.3.2. Similarly, see that act19 on line 135 of Figure 13 corresponds directly to the invariant defined in section 3.3.2.
Figure 13
The editLastMeasurement is also very similar to the basic model. Again, the only thing that changes significantly are the actions. Upon inspection, it is clear that the actions have been changed so that they correspond to the new invariants.

### 3.4 Interconnection of Technologies

This subsection describes what all of the different technologies do and how they work together throughout the many different steps of this project. As stated earlier, the entire process starts with an Event-B model. The model is created in an Eclipse-based
IDE called Rodin. Within the model are invariants, many of which correspond to necessary medical conditions of the application. These invariants are formally verified using the Atelier B provers installed in Rodin. Some invariants are automatically proved, while others are verified using an interactive prover (more detail in section 3.5).

EventB2SQL is a code translator that is written in Java. It scans and parses the formally verified Event-B model and outputs Java code that creates and manages a database corresponding to the original model. This code is used in an Android Studio project that creates a fully functional application.

3.5 Formal Verification and Proving

An Event-B model is only ready for translation once all possible states of the model are guaranteed to satisfy all of the invariants. The Atelier B provers verify most invariants automatically. This is called automated proving, and the Atelier B provers are often referred to as automated provers for this reason. However, it is important to note that not all invariants are automatically proven, even if it is the case that the invariants will always be satisfied. This is the main limitation of the provers; sometimes the provers will need help from the developer in order to formally verify the correctness of the model. The way a developer does this is with an interactive prover.\footnote{http://handbook.event-b.org/current/html/proving_perspective.html}

The interactive prover is a tool in which the developer can give the provers more information, as well as tell them what information to use. It gives a few different pieces of information to the developer, the most important being the proof tree. The proof tree shows what steps the prover has taken and where in the proving process that the
automated prover halted. At the bottom of this tree is the goal, or the mathematical statement that the prover needs to prove is true in order to complete the entire proof. In addition to the proof tree, the interactive prover shows the user what hypotheses are available to prove the necessary goal. Note that the goal changes as the tree becomes larger. Logically, the provers are attempting to simplify a long predicate to the point that the last node is true.

Figure 15 shows a proof tree in which we are trying to prove that a variable \texttt{sleep} is equal to 4. For now, do not worry about the significance of what this proof is doing, but rather understand that this is the end node of a fairly complex proof. That is, the entire proof (which is much more significant than just showing the value of \texttt{sleep}) will be satisfied if the developer can assist the provers in showing that \texttt{sleep}=4. In order to assist the provers in doing this, we added a hypothesis (Figure 17) that the provers could prove on their own. Once this was done, the provers had an extra hypothesis in their arsenal, which allowed them to complete the proof, as indicated by the green proof tree (Figure 16). Without that assistance, the prover never would have known to add that hypothesis. I had to do some sort of interactive proving similar to this for each of the 30 medically relevant invariants of my project, as well as for a few others.
3.6 EventB2SQL and Android Studio

EventB2SQL is the compiler that takes in an Event-B model and outputs code that creates and manages a database corresponding to that model. EventB2SQL is created and managed in Eclipse. For this project, the generated code will be in Java and will use SQLite to manage the database. The Java file will be imported into an Android Studio
project. As described earlier, the power of translating Event-B to Java is that our Android application satisfies the invariants proven for the model, as the translation is sound.

Only one Java file is generated, but the end application has several files working together. The generated file is not altered once generated in order to preserve the soundness of the translation. Instead, other classes within the Android Studio project will use the methods defined in the generated code. It is easiest to think of the generated file as the model of a Model-View-Controller (MVC) design pattern (Gamma, Helm, Johnson, and Vlissides, 1995), while the other files in the project make up the view and controller. The hand-written code will use methods in the generated code in the same way Controller code would use methods from a model.

The application provides the ability to add measurements as well as view previous data and current warning-levels. Future work will explain the intent to add the ability to edit to the functionality of the application. The page on which a user can add a measurement is composed of five progress bars (one for each type of measurement) that range from 1 to 5, and a submit button. You can see a picture of this screen in Figure 18 on the left. The progress bars are generated elements that are added and formatted programmatically. That is, only using an access method, I was able to generate a progress bar with the correct range for each type of input. This User Interface code generation was added to EventB2SQL during this project, and should be a useful tool in future projects. It is important to note that while there are UI components generated, they have nothing to do with formal verification. The generation of these components is only to provide the programmer with more tools and improve the EventB2SQL tool as a whole.
In addition to functionality directly matching the Event-B model, I also added the option to view all data over time with a linear graph. The graph has five overlapping line series, one for each type of measurement. Under the graph, there are five traffic lights, each representing a warning-level for the user. For example, if the traffic light next to pain is red, this is an indication to the user that action should be taken to ensure his treatment is working properly.

Figure 18
Chapter 4

Results

4.1 Translation

After creating the model and satisfying/completing all of the proofs, I was able to translate the model with just two clicks. From here, I imported the generated code into the appropriate Android Studio project. Due to the changes we have made to the compiler, there is now a significant amount of generated User Interface. However, this project has shown the potential for even more helpful UI components that could be generated. This is discussed more in the future work section. The compiler's ability to recognize and generate enumerated types has proven sufficient and useful. This will be explained more in section 4.2. Ultimately, the effectiveness of EventB2SQL in translating this healthcare model is more than satisfactory.

4.2 Enumerated Types

Enumerated types are not implemented in Event-B, but there is a workaround using carrier sets. Recall that carrier sets hold values of the same type, essentially introducing user-defined data types to the model. In order to represent an enumerated type in Event-B, we make a constant for each enumerated value then partition a set with only singleton sets – each containing one of the constants. Figure 19 shows an example of this workaround by creating an enumerated type MEASUREMENT that is meant to have five different values for the different types of measurement that a user can input: PAIN, SLEEP, MOOD, STRESS, and ACTIVITY. The set MEASUREMENT is partitioned by five
singleton sets, each containing exactly one constant, causing anything in the set MEASUREMENT to always have the value of exactly one of the constants.

At the start of this project, EventB2SQL did not recognize this as the enumerated type that it functions as, but rather as any other carrier set. This causes it to behave undesirably in a number of different ways. To fix this, we extended the compiler in several ways. First, we had to change the fundamentals and split up the carrier sets. Previously, we only had one list of carrier sets. Now, when we parse the axioms of Event-B, we check each carrier set to see if it was created as a standard carrier set or if it is actually meant to function as an enumerated type. After determining the true nature of the carrier set, it is added to the appropriate list. Once we had the carrier sets split up appropriately, we still had to change the ways a lot of the translation occurred. Caroline Nguyen ‘16 has implemented the recognition and handling, which has been very useful for the project. Once parsed correctly, a table is created in the database containing the elements of the enumerated type.
4.3 Proving

The automated provers were not as effective as anticipated; many of the proof obligations generated from the invariants were not discharged by the automated provers. In particular, the prover failed to verify many simple properties of functions and relations. However, the interactive prover built into Rodin has been a very useful tool. I was unaware of this tool when beginning the project, and it was very difficult to master.

In some cases the prover would have trouble with seemingly trivial steps in the proof. For example, I would often have to assist the prover in recognizing that $x + 1 - 2$ was equivalent to $x - 1$. However, after much troubleshooting, our understanding of the interactive prover and how to assist it has grown extensively. We were able to successfully verify all of the proof obligations for both models using the interactive prover in conjunction with the automated provers.
Chapter 5

Challenges and Future Work

5.1 Model

The current model has many medically relevant invariants, all of which are satisfied. In the coming weeks, we should have very little trouble satisfying the remainder of the proof obligations. Furthermore, the current model includes both an addMeasurement and editLastMeasurement event, giving the generated code more than the most basic functionality. However, another event, deleteLastRecording was removed due to some challenges with the provers. The invariants could not be satisfied automatically when this event was included, and it was too difficult (at least for now) to satisfy them with the interactive prover. While I do not think that delete functionality was necessary for this application (as the user is not supposed to skip inputs), I think that it would have proof of concept benefits, as many other applications could use this functionality. This is something that would be beneficial to have in future work.

5.2 Android

The final product of this project is an Android application. The application allows the user to view data, input measurements, and create a database. In the next couple of weeks, I plan to implement the ability to edit measurements into the application, as the event corresponding to this is nearly completed. This way, all of the functionality in the model will be represented in the application.
In addition to the functionality given in the Event-B model, I also added a page in the application to view all of the data. This included both a linear graph that showed the data over time, as well as a traffic light for each type of measurement indicating the warning-level. There were several challenges, and some future work for graphing.

Android has no built in graphing API, so we were forced to use a third party graphing API. The API was surprisingly functional and capable, but there were still some formatting issues that the limit the interface. One thing I plan to attempt in the few weeks remaining is changing the x-axis to correspond to a date variable taken from the generated code. Currently, it corresponds to the index of the function, which is just 0, 1, 2, … and so on.
Chapter 6

Conclusion

6.1 Conclusion

Event-B was ultimately effective in describing a substantial healthcare model. However, the user needed to be fairly experienced in order to take full advantage of the provers. I think that the provers would need to be more capable in certain proving techniques in order to be accessible to first time users of Rodin and Event-B. That being said, they were ultimately able to prove all of the proof obligations for some very complex invariants.

EventB2SQL worked better than expected in its ability to effectively generate code. The resulting Java class was very easy to use and I had all of the functionality of the events of the Event-B model. Furthermore, with the new generated UI and enumerated types, I think that adequate, efficient, and user-friendly Android applications can be created with only trivial work from the programmer after code generation. The blend of all of these technologies was crucial in the creation of an application with formally verified medical constraints.
Chapter 7

References

7.1 References


Yang, F., Jacquot, J., & Souquieres, J. JeB: Safe Simulation of Event-B Models in JavaScript. The 20th Asia-Pacific Software Engineering Conference (APSEC), Dec 2013, Bangkok Thailand. <hal-00908056>