Early Validation of Functional Requirements and Transition to Source Code with Event-B

Nestor Catano

***Abstract*—The cost of fixing software requirements errors after deployment is so high that it is vital to come up with ways to find and fix requirements errors early in the life-cycle of a project. The work in this paper advocates for the use of formal methods as an alternative approach to guarantee the correctness of the software from requirements to code. We present a formal-methods based approach for the early validation of functional requirements. Our approach relies on formal methods techniques such as program refinement, correctness-by-construction (CbyC), and automated code generation. We present two case studies that showcase our approach; for the case studies, we discuss design decisions, flaws encountered, and lessons learned.**

***Index Terms*—Event-B, Formal Methods, Formal Specifications, Software Requirements, Verification.**

I. Introduction

Software errors result in significant costs, schedule overruns [11], [10], [3], [4], quality issues, and failures to deliver the specified functionality. While there have been various techniques to enhance requirements analysis and detection of software defects, software products are shipped with various defects that are rooted in the requirements. Such problems are exacerbated in the presence of safety-critical, autonomous, and mission-centric systems.

We present a novel approach to validate software requirements in the early phases of software development with the EventB formal method [1]. We advocate for an approach that not only allows early validation of software requirements but also enables the accurate implementation of functional requirements. Early validation of software requirements seeks to reduce the high cost of fixing bugs in later software development phases [4], [3]. Hence, we model the functional software requirements of the application in EventB, we refine the model with the Rodin platform [2], and we use the EventB2Java code generator [12], [6] to generate a prototype implementation of it, and optionally unit-test the generated code. The generated code is not meant to be a final implementation of the EventB model, but a code used for validation purposes only.

We use two case studies to showcase our ideas on validation of software requirements and generation of source code for functional requirements. The first case study is about the use of the EventB formal method and unit-testing techniques to validate the functional requirements of RoadFighter, an Android car racing game. The second case study is about the validation of the software requirements of a chat system (similar to WhatsApp) fully in EventB, without resorting to the use of unit-testing.

This paper is organized as follows: Section II presents our proposed approach for validating software requirements with EventB. Section III presents our first case study, an Android car racing game. Section IV discusses our second case study, a chatting system. Section V discusses some work that relates to the validation of software requirements. Finally, section VI presents conclusions and discusses future work.

II. The Methodology

Figure 1 presents our methodology to validate the functional software requirements of applications. Informal requirements are elicited as User Stories, which are then utilized to write a formal model of the system in EventB. The EventB model is refined as required, and Proof Obligations are discharged along the way to ensure that the model is sound. We use the EventB2Java tool to generate a prototype implementation of the application. We manually write unit tests to validate the implementation of the application. The prototype is run, unit tested, and thus the software requirements validated. If a test fails, the informal or the formal specification is evolved and the whole validation process is restarted.

We encode informal requirements as User Stories, which are textual templates of the form "as a <Role>, I want to <Goal/Desire> so as to <Why>" that encode functional software requirements. Although User Stories are typical of Agile methodologies, we use them here as their syntax closely follows the syntax of EventB. A User Story includes some Acceptance Criteria against which it should be validated. An acceptance criterion has 3 components, a Given part that describes when the functionality may be triggered/executed (which depends on the internal state of the system), 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 describes how the state of the system changes when that functionality executes.

III. RoadFighter

The goal of this case study is to demonstrate how refinement calculus with EventB and unit testing techniques can be combined to sanitize and validate the functional software requirements of a car race Android app. RoadFighter is a racing arcade game. The goal of the game is to reach the finish line without running out of time or hitting other cars. The car racing game consists of a single *car* controlled by the user, the car is placed on a *lane* that has a *finish line* and *borders*. Cars run over lanes. Several static and dynamic objects called *obstacles* as well as other artificially controlled

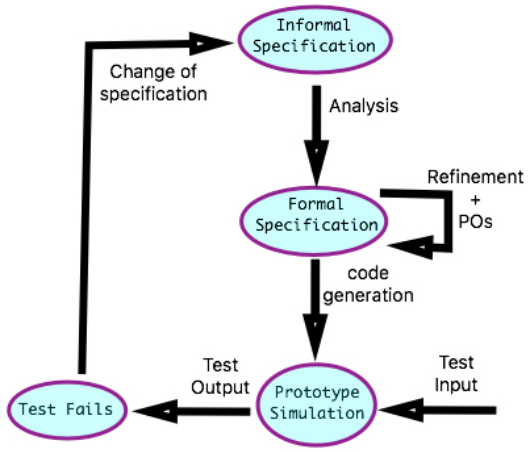

Fig. 1. Chat: validation of software requirements

cars called *opponents* are placed on the *lane*. The simulation of the physics involved in driving a car can be as accurate as the developers want and the hardware allows. A scoring system is modeled to reward players. As games are meant to attract people, variations of the scoring system and rules can be implemented in order to achieve market differentiation. Two important aspects of physics involved in artificially simulating a car race are kinetic movement and collision detection. A straight transformation of physics equations of accelerated movement into algorithms (implemented in Java for instance) is not possible because of their continuous nature. Game programmers often opt to apply a discrete approach based on Euler Discretization. The following equations model a uniformly accelerated movement. You may see that these equations are recursive, and require an initial value, similar to a state machine. Thus, Δ_t is the time step-value of the discrete system.

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

A great diversity of algorithms exist to calculate collisions, which may use boxes, spheres or ellipses as their geometrical enclosing form to calculate the intersection of objects. Because exhaustive collision detection is time demanding, the trade-off thus is the speed of processing vs. exactitude [8]. We considered a two-dimensional box collision detection algorithm as shown below.

$$
|pos_x(obj_1) - pos_x(obj_2)| < \frac{width(obj_1)}{2} + \frac{width(obj_2)}{2} \quad \wedge \quad (1)
$$

$$
|pos_y(obj_1) - pos_y(obj_2)| < \frac{height(obj_1)}{2} + \frac{height(obj_2)}{2} \quad (2)
$$

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

A. The Event B Model

Figure 2 shows RoadFighter's EventB Model. Machine RoadFighter sees context ctxt_0 (not shown here), which declares two carrier sets OBJECTs and LANES representing all the possible objects and race lanes in the game, respectively. Variables objects and lanes are the current objects of the game and the existing race lanes; obstacles is the set of static objects of the game, and cars is the set of dynamic objects. Every object has a horizontal and vertical position (posX and posY), a total function that maps the object with an integer number representing the position. Objects are two-dimensional with a height and a width. Variables vel (velocity), acc (acceleration), and lean (bending from the straight-up position) keep track of the cars' kinetic. Variable collided (a set) keeps track of the collided objects.

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

B. Unit Testing

We wrote 29 JUnit tests to check the absence of errors in code. As EventB2Java translates each event into an independent Java class, we used each unit test to check the functionality of each event. When a test failed, we checked the code of the respective event in the Rodin platform, made the respective changes, and generated code again. All the tests passed at the end.

The JUnit test below checks that the delete_car event properly deletes a car that has previously been created. The way events are structured allows us to write unit tests easily. The car object needs to be given proper initial values following the respective event guard in order for the object to be added to the list of cars. EventB2Java generates two Java methods for each event, a "guard" method that checks if the event guard holds, and a "run" method that executes the event body in case its guard holds. If the test fails, likely, the guard did not hold initially thus the event body did not execute.

```
@Test
public void delete_car() {
 // initial values for Car, Desc, H, etc.

 ADD_CAR ac = new ADD_CAR(machine);
 ac.run_ADD_CAR(Car,Desc,H,W,X,Y,F,M);
 assertTrue(machine.get_cars().has(Car));

 DELETE_CAR dc = new DELETE_CAR(machine);
```

```
machine RoadFighter
sees ctxt_0
variables objects lanes obstacles cars width height
  posX posY vel acc lean score collided active
invariants
  objects ⊆ OBJECTS
  lanes ⊆ LANES
  obstacles ∪ cars = objects
  obstacles ∩ cars = ∅
  posX ∈ objects → ℤ
  posY ∈ objects → ℤ
  width ∈ objects → ℕ
  height ∈ objects → ℕ
  vel ∈ cars → ℤ
  acc ∈ cars → ℤ
  lean ∈ cars → {-1, 0, 1}
  score ∈ cars → ℤ
  collided ⊆ cars
  active ⊆ obstacles ∪ cars
events

 event initialisation
 then
  objects := ∅  lanes := ∅  obstacles := ∅  cars := ∅
  posX := ∅  posY := ∅  width := ∅  height := ∅
  vel := ∅  acc := ∅  lean := ∅  collided := ∅  active := ∅
 end

 event update_pos
  any car  elapsed
  where
    car ∈ cars
    elapsed ∈ ℕ
  then
   posX(car) := posX(car) + lean(car) × elapsed ÷ 1000 * 50
   posY(car) := posY(car) + vel(car) × elapsed ÷ 1000
  end
```

Fig. 2. ROADFIGHTER: EVENTB model

```
 dc.run_DELETE_CAR(Car);
 assertFalse(machine.get_cars().has(Car));
}
```

C. Experimental Results and Lessons Learned

We developed a prototype of ROADFIGHTER in Android Studio SDK version 29 using Java with OpengGL 1.0. We used Eclipse Java 2020 to JUnit tests. We wrote the EVENTB model of the car racing in Rodin version 3.3. We used the last version available of the EVENTB2JAVA Rodin plugin to generate code for the EVENTB Model of ROADFIGHTER (the update site of EVENTB2JAVA is http://svrrodin.javerianacali.edu.co/EventB2Java_UpdateSite_Rodin3.2/). The Android Studio's ROADFIGHTER implementation makes use of the Java code generated by EVENTB2JAVA. The Rodin project, the Eclipse project with the JUnit tests, and the implementation in Android Studio are available from https://github.com/ncatanoc/carracinggame.

Table I summarizes ROADFIGHTER's development effort in EVENTB and Rodin. LOC stands for Lines of Code, and POs for the number of Proof Obligations generated by Rodin. The EVENTB model includes 1 abstract machine and 3 refinement machines. Rodin's provers discharged all the POs automatically. A Master student in Computer Science spent 2 weeks developing a previous and much simpler version of the model that contained only 1 machine. He spent some additional 2 weeks developing the Android Studio project that reuses the code generated by EVENTB2JAVA from the EVENTB model. Then, the first authors spent additional 2 weeks extending the previous EVENTB model to include the 4 current machines. The Android Studio remained unchained (except porting file configurations to a more recent SDK version) as the first author kept essentially the same events of the previous version. Lastly, the first and second author spent 2 days writing the JUnit tests in Eclipse. The first author discovered a small error in the definition of an event guard that he corrected in EVENTB and code-generated afterward. One of the most striking facts of our software development with EVENTB is that Rodin was able to discharge all the POs automatically. This is mainly due to the high performance of Rodin's SMT prover (10 years ago it was a different story), but also at the way the EVENTB model is written, e.g., the model is defined in terms of sets and relations and we put effort into not using quantified logical expressions.

Machine	LOC	# POs	% Aut.
RoadFighter	82	22	100
ref0_circuit	187	67	100
ref1_kinectic	203	57	100
ref2_scoring	95	5	100

TABLE I
ROADFIGHTER: STATISTICS FOR THE EVENTB MODEL

Our prototype implementation of ROADFIGHTER underwent a two-steps sanitization process. First, we discharged all the Proof Obligations in Rodin. Second, we manually wrote Java Unit tests for the code generated by EVENTB2JAVA. The first process checks that the EVENTB model is sound, the second process verifies that our understanding of the model is correct. Often times, when writing the unit tests for an event, we realized that the event guards are not correctly defined in Rodin, making the test fail. Therefore, we corrected the event in Rodin, generated Java code again for the whole event, and re-run all the tests automatically in Eclipse.

We summarize below some key aspects of this case study.

(i.) Game apps (that are similar to ROADFIGHTER) use floating-point values to model features such as object positions or dimensions. EVENTB implements integer arithmetic as opposed to floating-point arithmetic that one would need to implement the physical elements involved in a game.

(ii.) Rodin's provers managed to discharge all the POs of our EVENTB model automatically. Of course, the user's expertise is required for proper modeling software applications in EVENTB.

(iii.) Formal methods can be applied to non-critical systems such as ROADFIGHTER, unlike the popular disbelieve that advocates the use of formal methods for critical applications mostly.

(iv.) Formal specifications help developers reason about requirements even before writing a single line of code. Analysis of requirements at the early stages of software development promotes the development of trustworthy software.

(*v.*) The use of set theory in formal verification helped us write mathematically sound specifications and model complex business logic in a very simple way.

(*vi.*) EVENTB specifications provide great insight into how to write unit tests. A developer who has a basic understanding of formal methods can easily port EVENTB specifications into unit tests. We think that a formal methods tool that generates unit tests from EVENTB models automatically must be implemented.

IV. A CHAT APPLICATION

The chat application is an application with functionalities similar to the WhatsApp app. It features functionality to send content, delete content, forward messages, etc. The chat application brings further challenges to the process of validation of functional software requirements with EVENTB, e.g., stronger interaction of events and complex machine invariants. The chat application shares some similarities with the car racing game app in Section III, e.g., both apps are Android apps. However, unlike for ROADFIGHTER, we do not develop an interface for the chat system here.

The main goals of this case study are the following:

(i.) To demonstrate how software requirements for a real-life application can unambiguously be specified.

(ii.) To demonstrate how refinement and code generation techniques can effectively be used in practice to validate software requirements early in software development. Early validation of software requirements seeks to reduce the high cost of fixing bugs in later software development phases [4], [3].

(iii.) To give software developers alternatives to the use of formal methods in application development in a way that is cost-effective.

A. Functional Software Requirements

Functional software requirements for the chat application are similar to those of WhatsApp (available from https://www.whatsapp.com/android/). We elicited the requirements for the chat application from our own experience using (the SmartPhone version of) WhatsApp. The US-01 User Story below introduces a requirement that describes the functionality used for creating a chat session between "Me" and "You". The chat may not yet exist (it is a fresh chat session). In addition to the User Stories, we wrote the 13 invariants. Section IV-B discusses how to encode some of these invariants in EVENTB.

US-01	create-chat-session
Description	As a user, I want to create a chat session to communicate with You
Acceptance Criterion	Given: A chat session between Me and You does not exist When: I decide to create a chat session with You Then: A chat session between Me and You is created, but not between You and Me.

B. The Event B Model

Table II presents the machine structure of our chat application in EVENTB. The abstract machine (MachineZero) observes basic functionality for chat sessions. The first machine refinement includes functionality to check whether chat content (text, video, picture) can be read or not. The second machine refinement adds implementation details. For instance, it represents chat content as a sequence (rather than a set) of content items.

Machine	Observations
MachineZero	Basic functionality
MachineOne	Read and unread status
MachineTwo	Vertical refinement

TABLE II
CHAT: EVENTB MACHINES STRUCTURE

The event create-chat-session below encodes US-01 in EVENTB. It creates a chat session for user Me to chat with user You. The symbol $\in$ is the set membership operator, user is the set of users who have signed up to the chat application. The symbol $\wedge$ is the logical-and operator. Me $\mapsto$ You is a pair of elements, and chat is a mathematical relation that keeps track of the chat sessions that have been created. Event guard @grd2 formalizes the Given condition of US-01. Action @act1 encodes the Then expression; it adds the pair Me $\mapsto$ You to the set of existing chats. Guard @grd1 is a typing condition, it helps parsers and compilers infer the type of Me and You.

```
event create-chat-session
any Me You
where
 @grd1 Me ∈ user ∧ You ∈ user
 @grd2 Me ↦ You ∉ chat
then
 @act1 chat := chat ∪ {Me ↦ You}
end
```

We wrote in total 10 User Stories that relate to the abstract machine (MachineZero), for creating a chat session, selecting a chat, chatting, deleting content, deleting a chat session, muting a chat, unmuting a chat, broadcasting content, forwarding content, and unselecting a chat, respectively. Each User Story relates to the static part of the machine (variables, data structures, invariant, axioms), to its dynamic part (1 or more events), or both parts.

Table III summarizes our EVENTB development effort in Rodin for the chat system. The two last columns tell the number and percentage of Proof Obligations that are automatically discharged by Rodin's provers. As one might expect, the last refinement has a lower success rate in discharging Proof Obligations automatically. This is partly because the last refinement machine introduces several data refinements.

We present below the formalization of Invariant "it is never the case that chat content exists associated to a pair of users for which no chat session exists". Expression chat[{u}] returns

Machine	LOC	# POs	# Aut.	%
MachineZero	170	53	51	96
MachineOne	91	46	46	100
MachineTwo	114	39	33	85

TABLE III
CHAT: STATISTICS FOR THE EVENTB MODEL

the set of users with whom user u is chatting. Hence, the chatcont EVENTB.

$$\text{@inv11}\ \forall u,c,s \cdot u \in user \land c \in content \Rightarrow (u \mapsto c \mapsto s \in chatcontent \Rightarrow s \subseteq chat[\{u\}])$$

Implementing Invarint "A chat session between two users may have a set of associated content available to either or both of them" requires a subtler analysis as it relates content, the sender, and the recipient of the content. We formalize this invariant as @inv6 below.

$$\text{@inv6}\ chatcontent \in user \pfun (content \pfun \mathbb{P}(user))$$

C. Software Design Decisions

Events delete-content and remove-content are subtle events as their executions can easily break the machine invariants, where $\oplus$ is EVENTB's overriding operator. Event delete-content below does not remove c from chat You $\mapsto$ Me, e.g., @act2 content := content $\setminus$ {c}). If it does, it might break invariant @inv6 in Page 5: it might be the case that some user has sent that content to a group of users previously. If one removes that content, one would need to demonstrate that for any user u other than Me the range of chatcontent(u) is a partial function from content$\setminus${c} to $\mathbb{P}$(user), which is unprovable. This means that any direct implementation of a functionality deleting chat content either cannot remove the content from the lists of contents, or may delete it only if it does not exist in any other chat, which might be time-consuming to validate.

```
event delete-content
any Me You c
where
 @grd1 Me ∈ user ∧ You ∈ user
 @grd2 Me ↦ You ∈ active
 @grd3 Me ∈ dom(chatcontent)
 @grd4 c ∈ dom(chatcontent(Me))
 @grd5 You ∈ chatcontent(Me)(c)
then
 @act1 chatcontent(Me) := chatcontent(Me) ⊕
                          {c ↦ (chatcontent(Me)(c)\{You})}
end
```

D. Lessons Learned

The Rodin project of the chat case study is reachable at https://github.com/ncatanoc/whatsapp. In the following, we summarize some key aspects of this case study.

(i.) We advocate for the use of formal methods tools at the early stages of software development. The use of formal tools helped us validate the software requirements of the chat system and check for possible inconsistencies.

(ii.) The case study presented here solely uses EVENTB (and not unit testing) to validate functional software requirements expressed as User Stories. There are a few advantages of using EVENTB for that purpose. (a) User Story acceptance criteria match the syntax of events in EVENTB. This enables the design and implementation of software tools for the (semi-) automatic translation of informal specifications to formal specifications. (b) EVENTB specifications can be refined as desired. In other words, we can make our software design as abstract or concrete as we want.

(iii.) Expressing informal software requirements as User Stories helped us express them as events in EVENTB since their structure match. On the other hand, software invariants are not typically written by software engineers as part of their software requirements documents, yet they are critical to EVENTB.

(iv.) The correct behavior of a chat system depends on the correct functioning of multiple people chatting concurrently. Our work does not account for the performance of several chats running in parallel. Performance would require an architectural analysis.

V. RELATED WORK

E. Stachtiari *et al.* [14] introduce an approach for the specification of system requirements, and the derivation of system properties, which should be implied by the system structure. System design is carried out with the BIP (Behaviour-Interaction-Properties) framework. Their approach is to be applied during the early stages of software development. They can build an executable model of a system starting with their system requirements. BIP and the EVENTB language that we use in our work are both models of transition systems. One advantage of using EVENTB is that it enables the definition of software refinements from the most abstract machine all the way down to code.

The use of textual boilerplates to express software requirements is not a new idea. M. Śmiałek *et al.* [13] propose a notation to express software requirements suitable for their translation into design models and code. Requirements are written in an SVO (Subject-Verb-Object) format and then transformed into a sequence of User Stories. SVO notation could be incorporated in our work to initially express software requirements, which then are translated to our User Story textual templates, and then to EVENTB.

S.-P. Miller *et al.* [9] describe a case study conducted to determine if formal methods can be used in practice to validate requirements early in the software development process at some *reasonable* cost. The formal methods RSML and NuSMV are used to model the software requirements and the PVS system to conduct the underlying proofs. They claim their case study demonstrates that formal methods and formal models can effectively be used in realistic systems (an avionic system with several hundred requirements) to find

errors before implementation starts. Their case study is for a flight guidance system. Our work is not necessarily related to safety-critical systems.

VI. Conclusion

Our work focuses on the use of Formal Methods for model validation (as opposed to using Formal Methods for program verification). We manually translate User Stories to formal specifications in EventB. Our first validation check is carried out with Rodin's type-checker, which checks for each EventB model that all its variables, constants, events, and the rest of its constructs are well-typed. Rodin raises an error in case it cannot infer the type of any construct. The second validation check is achieved through the generation of Proof Obligations that attest to the correctness of the EventB model. Rodin generates several types of Proof Obligations [7]. It generates a feasibility Proof Obligation for each action of every event stating that a solution (a program) for the assignment exists. It also generates Proof Obligations that relates a blueprint and its refinement. For instance, simulation Proof Obligations ensure that abstract event actions are correctly simulated by concrete actions. Rodin provers are semi-automatic in that they only assist users in discharging the Proof Obligations.

The last validation check relies on the soundness of the code generator. The translation from EventB to Java (or any other implementation language) must be sound in order for the Java program to attest to the EventB formal specification. A formal soundness proof for the code generation performed by EventB2Java has already been conducted [5].

[10] Nur Nurmuliani, Didar Zowghi, and Sue Fowell. Analysis of Requirements Volatility during Software Development Life Cycle. In *15th Australian Software Engineering Conference (ASWEC 2004), 13-16 April 2004, Melbourne, Australia*, pages 28–39. IEEE Computer Society, 2004.

References

[1] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Design*. Cambridge University Press, New York, NY, USA, 2010.

[2] 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. *International Journal on Software Tools for Technology Transfer*, 12(6):447–466, 2010.

[3] Barry Boehm. Software Engineering. *IEEE transactions on computers*, 25(12):1226–1241, December 1976.

[4] Barry Boehm and Philip Papaccio. Understanding and Controlling Software Costs. *IEEE Transactions of Software Engineering*, 14(10):1462–1477, October 1988.

[5] Néstor Cataño and Shigeo Nishi. Soundness Proof of EventB2Java. In *Seventh Latin-American Symposium on Dependable Computing (LADC)*, volume 0 of *IEEE Digial Library*, pages 25–34, Cali, Colombia, October 19–21 2016. IEEE Computer Society.

[6] 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.

[7] Néstor Cataño, Víctor Rivera, and Rustan Leino. The EventB2Dafny Rodin Plug-In. In *2nd Workshop on Developing Tools as Plug-ins (TOPI)*, pages 49–54, Zurich, Switzerland, June 3 2012. IEEE Xplore.

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

[9] Steven P. Miller, Alan C. Tribble, Michael W. Whalen, and Mats P. E. Heimdahl. Proving the shalls: Early Validation of Requirements Through Formal Methods. *International Journal on Software Tools for Technology Transfer*, 8(4):303–319, 2006.

[11] Bashar Nuseibeh. Conflicting Requirements: When the Customer is Not Always Right. *Requirements Engineering*, 1(1):70–71, 1996.

[12] Víctor Rivera, Néstor Cataño, Tim Wahls, and Camilo Rueda. Code Generation for Event-B. *International Journal on Software Tools for Technology Transfer*, 19(1):31–52, February 2017.

[13] Michał Śmiałek. From User Stories to Code in One Day? In Hubert Baumeister, Michele Marchesi, and Mike Holcombe, editors, *Extreme Programming and Agile Processes in Software Engineering*, pages 38–47, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[14] Emmanouela Stachtiari, Anastasia Mavridou, Panagiotis Katsaros, Simon Bliudze, and Joseph Sifakis. Early Validation of System Requirements and Design Through Correctness-by-Construction. *Journal of Systems and Software*, 145:52–78, 2018.