# Can Learning Formal Specification Be Fun? —Experience and Perspective

I. S. W. B. Prasetya
*Utrecht University*, Netherlands
s.w.b.prasetya@uu.nl, Orcid: 0000-0002-3421-4635

Craig Q.H.D. Leek
*Utrecht University*, Netherlands
cqhd.leek@gmail.com

Roan Oosenbrug
*Utrecht University*, Netherlands
roan.oosenbrug@gmail.com

Petar Kostic
*Utrecht University*, Netherlands
petarkostic1995@gmail.com

Mike de Vries
*Utrecht University*, Netherlands

***Abstract*—Writing formal specifications is a useful skill for students to develop and to grow a positive mindset towards it. Unfortunately this is hampered by the stereotyping of formal method as dry and boring. In this short paper we discuss our experience in using of a computer game called FormalZ as an attempt to introduce some fun in teaching the skill. Two setups are discussed: as an embedded part of a course, and as a loose tutorial, afterwhich we will conclude with the lessons learned.**

***Index Terms*—teaching formal method, gamification in teaching formal method, gamification in teaching software engineering**

## I. Introduction

Writing formal specifications should be part of software engineering's good practices. To paraphrase Spivey in his 1989 *Introduction to Z and Formal Specifications* [1], formal specifications can provide written and reliable reference point for developers and testers as to what the functionality they are supposed to implement and verify. Advances in technologies since 1989, in particular the rise of model checkers and automated testing tools, only makes the case stronger. Having formal specifications would then allow us to verify the correctness of programs —or at least to automatically test them. So, why do developers still ignore formal specifications?

Developers typically point out that using formal methods requires lengthy, and costly, mathematical proofs. Furthermore, many formal methods employ mathematical notations and concepts alien to programmers; to quote Parnas [2]: "*There is no quicker way to lose the attention of a room full of programmers than to show them a mathematical formula*". These are valid concerns, but they are no longer as big obstacles as they used to be. The aforementioned technologies (automated testing etc) mean that writing formal proofs is no a longer an absolute requirement. True, these technologies are fundamentally incomplete. But it does not mean that they are useless. On the contrary, they would greatly strengthen traditional manual testing, as already evidenced by e.g. success stories of fuzzing [3]; imagine what it can add if we also have formal specifications. Knowledge on mathematical notation is no longer a requirement either. Modern programming languages like Java or C# incorporate $\lambda$-expressions, allowing mathematical concepts like $\forall$ and $\exists$ to be expressed natively and cleanly in these languages themselves; Figure 1 shows a simple example of this.

```
void getMaxSpec(int[] a) {
   pre(a!=null);
   pre(a.length > 0);
      // call to actual implementation
      int retval = getMax(a);
   post(exists(a, i -> a[i]==retval));
   post(forall(a, i -> a[i]<=retval));
}
```

Fig. 1. *A pre/post-condition specification in Java of a program called* getMax *to return the greatest element of an integer array. The syntax $x \rightarrow e$ denotes a $\lambda$-expression, specifying (above) a predicate over the indices of an array.*

So, if math is no longer an obstacle, why do developers still ignore formal specifications? There are various explanations for the situation. Still, some of the blame lies in the Education itself, for failing to make students enthusiastic about formal methods, or at least about formal specifications. Students experience the subject as dry, and they do have a point. Learning to program is an engaging experience because programs produce tangible results that evoke one's sense of satisfaction (e.g. a program that produces nice Fractal graphics). In contrast, specifications are predicates. The only value they will produce is either a "true" or a "false". This is hardly exciting. A solution might be found in *gamification*. That is, using games or elements of games to maximize enjoyment and engagement in learning [4]. It is at least a hypothesis worth investigating.

In a previous paper [5] we presented a game to train the basics of writing formal specifications in the form of pre- and post-conditions called *FormalZ*[1]. Unlike existing Software Engineering themed education games like Pex [6], Train-Director-B [7], and Code Defenders [8], FormalZ takes a deeper gamification approach [9], where 'playing' (doing things for the fun of it, as opposed to behaving in a fully goal-driven way) is given a more central role. After all, what makes games so engaging is not merely the awarded scores and badges, but also the experience of playing them.

In this paper we will discuss our experience in actually using FormalZ in teaching in two setups: (1) using the game alongside a regular course on Software Testing, and (2) as a separate tutorial session. In Section II we will first explain the main concepts of FormalZ as in [5], and highlight design de-

Funded by the EU Erasmus+ grant 2017-1-NL01-KA203-035259.

[1] https://github.com/FormalZ

cisions made towards creating a formalization game. Sections III and IV present and discuss the two setups/cases. Section V concludes, summarizing the lessons learned.

## II. The Design of a Formalization Game

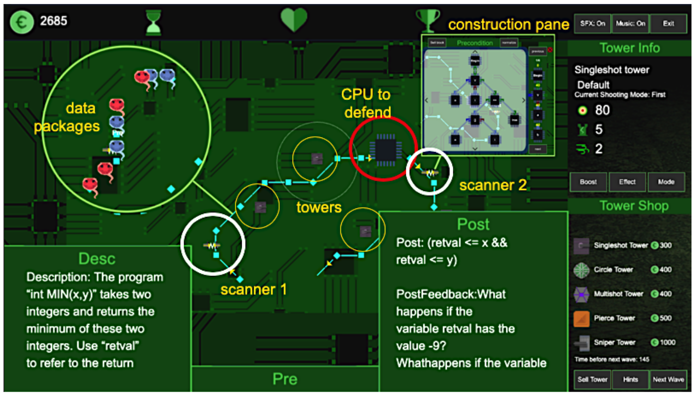
The functional purpose of FormalZ is to train users to turn informal specifications of programs into formal pre- and post-conditions. The teacher provides the informal specifications along with the corresponding solutions; e.g. this could be an informal specification of the program in Fig. 1:

> *”Given a non-empty array a, the program* **int** getMax(**int**[] $a$) *returns the greatest element in the array.”*

The student’s task is to construct formulas capturing the pre- and post-conditions of the program. FormalZ checks if they are semantically equivalent with the teacher’s solution. The theorem prover Z3 [10] is used to check this. If this were all that it does, it would be just as boring as a traditional quiz, with the only difference that the answer is checked by the computer. This saves the teacher’s time, but is hardly exciting for students. While it is possible to improve the excitement through the usual gamification elements such as graphics, scores, and badges, we would like to have something innovative that would trigger students’ curiosity.

To allow more game elements (beyond scores and badges) to be introduced, we cast the original formalization problem as a variation of the tower defense game. This provides concepts such as attackers and defenders, creating conflicts in the game as an instrument to build excitement. It is also a popular game genre, hence many users may already be familiar with its concepts. Figure 2 shows an annotated screenshot of FormalZ, showing the Printed Circuit Board (PCB) of some computer. On the PCB, there is a unit called ”CPU” (red encircled in Fig. 2) that has been programmed to implement some functionality informally described in the *Desc*-tab (left under). Input and output wires (bright blue) are connected to/from the CPU, and they transport data packages. Unfortunately, hackers may find a way to corrupt the packages. To guard against this, the PCB has two scanners (white encircled) that the player can program to identify correct inputs and outputs. In other words, they must be programmed to reflect the CPU’s pre- and post-conditions. Data packages not satisfying the scanners are *marked*. The player should buy and place defense towers (yellow encircled) on the PCB to ”destroy” marked packages before they reach the CPU, or the end of the output wire, if they are output packages. These defense towers are not directly related to the game’s learning goal. Instead, they represent the playful aspect of the game. The game can be completed without them, if the player manages to build the correct scanners right away. With towers a player can however last longer, hence has more opportunities to fix his/her scanners. Moreover, it is usually more satisfying to see corrupt packages are being shot at from all sides.

Rather than just providing a dry binary feedback (correct/incorrect), the game gives four types of visual feedback as well as textual, more technical feedback. The visual feedback comes in the form of the coloring of the incoming/outgoing data packages; this is summarized in Figure 3. Corrupted data packages are red; these represent incorrect data. Correct data are blue. Crossed data packages represent those that the scanners have marked. So, the presence of marked red packages and unmarked blue packages visually cue the player that the scanners are doing well. On the other hand, the presence of unmarked red packages is a cue to the player that the corresponding scanner is under specified, whereas the presence of marked blue packages means that the scanner is over specified. The textual feedback shows the ’content’ of an incorrectly marked or incorrectly unmarked data package (in other words: a counter example) to further help the player to fix his/her scanners.

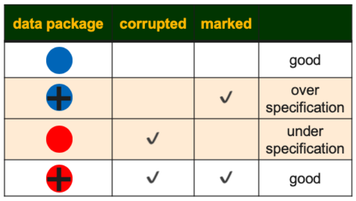


Description: The program “int MIN(x,y)” takes two integers and returns the minimum of these two integers. Use “retval” to refer to the return

Post: (retval <= x && retval <= y)

PostFeedback:What happens if the variable retval has the value -9? Whathappens if the variable

Fig. 2. *A screenshot of FormalZ, with its main components annotated.*



| data package | corrupted | marked | |
|---|---|---|---|
| 🔵 | | | good |
| 🔵✚ | | ✓ | over specification |
| 🔴 | ✓ | | under specification |
| 🔴✚ | ✓ | ✓ | good |

Fig. 3. *Different types of data packages.*

*Constructionism:* To construct the pre- and post-conditions (the scanners) the student gets a special construction pane (up and somewhat to the right in Fig. 2) where blocks, representing either a variable, a constant or an operator, can be placed and connected to construct a tree representation of a formula; its so-called Abstract Syntax Tree (AST). Simply typing the formula is deliberately disallowed. We want students to give more appreciation to their own formalization effort. Studies show that one way to do that is by letting the subjects put more labor into it (also known as the ’IKEA effect’) [11]. Moreover, this applies the Constructionism theory of learning [12].

The theory believes that humans learn by constructing knowledge in their mind. Familiar physical objects play a key role, because the learner already has knowledge on how they work [13]. Framing new knowledge in terms of interactions with these familiar objects helps the learner to construct the new knowledge in his mind. The theory was originally proposed by Papert and Harel [12]. Papert used LOGO as an example, which he used to teach programming to children. A learner can easily relate the 'turtle' in LOGO with his/her own physical body which can turn and move forward and draws upon this analogy to learn LOGO programming concepts.

In FormalZ, the blocks (in the formula building) are visually depicted as electronic hardware components, which are existing concepts in the physical world (as opposed to e.g. a ”variable”, which is physically intangible) and are probably recognizable to most computer science students. Likewise, the wires that connect the blocks relate well to our physical experience, where electronic components always need to be connected by wires. By requiring the user to first locate the right block, drag it to the construction pane and then explicitly connect it to other blocks with wires, we enforce more self-conscious interactions by the user, hence creating a more gradual and deliberate process of knowledge construction, as opposed to just letting the user type in formulas.

## III. STV Case: FormalZ in a Course

We deployed FormalZ to help students train their formalization skill in the course Software Testing and Verification (STV) at Utrecht Univeriosty in 2019. About 70 2nd or 3rd year Computer Science students participated in the course. The course takes a half semester in duration, with 7.5 European Credit (EC) load. There are 4 hours of lectures and 4 hours of lab-sessions per week. The first half of the course is spent on software testing, and the second half on Hoare logic.

The students already have 7.5 EC in set theory and logic in their first year. However, since the learning goals of the Logic course focus on proof systems and techniques, it cannot be taken for granted that students would then know how to apply the gained knowledge to the programming context. E.g. in the Logic course domain details are completely abstracted as nondescript symbols like $p$ or $P(x)$, whereas in programming we would have arrays that conceptually are quite different from plain integers. For example when asked to formalize ”the array $a$ is non-empty”, a participant proposed the following:

$$(\forall i : 0{\leq}i{<}a.\mathsf{length} : a[i]{=}0)$$

which highlights the point that knowing the theory does not automatically means that one knows how to apply it.

There is also some difference in the mindset that one should take when programming and when formalizing. For example, when asked to formalize the post-condition ”the program $m(x, y)$ returns the minimum of the two parameters” a participant proposed:

$$\mathsf{retval}{\leq}x \ \&\& \ \mathsf{retval}{\leq}y$$

Formally, the specification is incomplete. However, from a programmer's perspective it makes sense. Most programmers would understand that $m$ should obviously return either $x$ or $y$. So to them, this is an automatic axiom, which of course should not be taken for granted when we switch role to formalizing the program's specification.

The above two examples illustrate the rationale of explicitly including writing formal specifications as a learning goal of the course STV. To work on this goal a set of 6 formalization problems are offered as homework to the students over a period of three weeks, and two more were given as part of the midterm exam. They are listed in Table I. The first four are also offered in FormalZ. The column ”dif” indicates the problem's difficulty in terms of the total number of quantifiers (∀ or ∃) appearing in their model solutions.

TABLE I
*The set of formalization problems in STV and the students' performance on the scale 0..1. The column ”week” gives the week on which the problem is being worked on. Problems marked with (E) appear in the exam, those with (F) are optionally offered in FormalZ.*

| week | problems | dif | $N$ | $N$-FZ | avrg | |
|---|---|---|---|---|---|---|
| | | | | | non-FZ | FZ |
| 2 | MIN | 0 | 64 | 31 MIN(F) | 0.91 | 0.82 |
| 2 | isMIN | 0 | 58 | 12 isMIN(F) | 0.75 | 0.75 |
| 3 | TRI | 0 | 50 | 8 TRI(F) | 0.83 | **1.0** |
| 3 | getMIN | 2 | 51 | 8 getMIN(F) | 0.53 | **0.86** |
| 4 | COMMON | 2 | 43 | 5 getMIN(F) | 0.66 | **0.8** |
| 4 | SORT | 6 | 43 | 5 getMIN(F) | 0.21 | **0.5** |
| 5 (E) | checkAtMost | 1 | 69 | 8 getMIN(F) | 0.73 | **0.78** |
| 5 (E) | getMinIndex | 2 | 69 | 8 getMIN(F) | 0.57 | **0.65** |

Using FormalZ is not forced, though encouraged. The column $N$ shows how many students did the corresponding problem, and the column $N$-FZ shows how many of these did it with FormalZ. All submissions/solutions are graded. The avarages are shown in Table I, with their scales normalized to 0..1. The column avrg-non-FZ shows the average of how well students without FormalZ perform on the corresponding problem, and the column avrg-FZ shows the average for students that also used FormalZ. Since the last four problems do not have FormalZ option, in Table I we compare the performance of these two groups: for each of these problems $Q$, we compare the group of students that did the getMIN problem on FormalZ as well as $Q$ itself, versus the group of students that did $Q$ but did not do getMIN with FormalZ.

What we can see in Table I is that as the weeks progressed, the use of FormalZ decreased. This is unfortunate. Usability issues are a likely cause for this, though we suspect the tight schedule of the course also had much influence on this. During the three weeks allocated to train on formalization (week 2-4), three chapters from Ammann & Offutt's Introduction to Software Testing [14] have to be covered as well as parallel learning goals, and along with that a project that involves development and unit testing. Towards the end of these three weeks, students would have to start preparing for the midterm exam in week-5, as well as the deadline of the project.

The results in Table I hint that playing FormalZ leads to learning, or at least to improved results, bearing in mind that this cannot be statistically confirmed due to the decreasing participation. The question on whether students had fun using FormalZ is more difficult to address as 'fun' is highly subjective and hard to quantify. The fact that not all students abandon FormalZ right away despite issues mentioned above can be seen as a good indication. A look at the backend logs also indicates that users do spend time on putting towers, which is the more playful aspect of FormalZ. We conducted a user

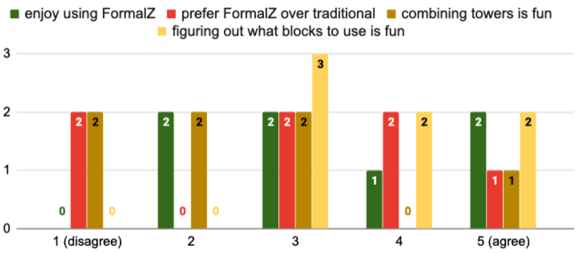


Fig. 4. *User survey 1.* **Green***: if the user enjoys using FormalZ.* **Red***: if the user would prefer FormalZ over written exercises.*

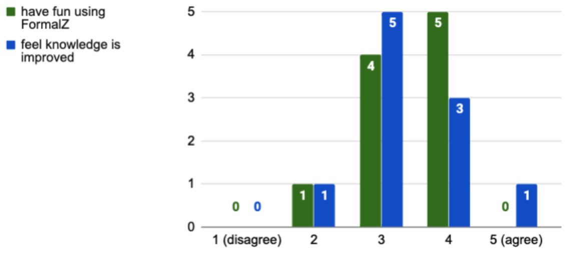


Fig. 5. *User survey 2.* **Green***: if the user had fun using FormalZ.* **Blue***: if the user feels that his/her knowledge in formal specification is improved.*

survey at the end of the course. Only a handful of students responded to the survey, but all respondents have played FormalZ in average 12 times and at least 8 times. Selected results are shown in Figure 4. When asked whether the users enjoy using FormalZ the responses were mixed (green). Constructing formulas using blocks seems to be experienced as fun (yellow), while playing with defense towers less (brown). When asked whether FormalZ would be preferred over traditional written exercises (red), we see that some respondents disagree; these respondents are also the ones that least enjoy the game.

We believe enjoyment is something that can be improved if we fix the game usabilities issues —in fact one of the disagreeing respondent indeed elaborated that that is indeed why he/she does not like the game. On the positive note, one of the respondent also wrote: "*I had some trouble initially understanding how to think about the logic, but once that (finally) clicked it was* **fun** *to do.*"

## IV. UCM Case: Dedicated Tutorial

In the second setup we use FormalZ to simply introduce, rather than train, the concept of formalizing informal specifications. This is done in a 1.5 hour tutorial session. The teacher first explains the idea and basics of FormalZ (0.5 hour). Then the students get one hour to work on one problem with difficulty level (as defined before) 0 and another of level 1. There are two major differences with the STV setup: (1) it is a dedicated session, so the students' attention is not distracted by other learning goals, and (2) major usability issues from the STV experience were fixed. The tutorial was given at the Universidad Complutense de Madrid (UCM), and was attended by 10 Computer Science master students. At the end of the tutorial a user survey was conducted; selected results are shown in Figure 5. Although not everyone agrees, the users do seem to be more positive than the experience in the STV course.

## V. Conclusion

The described cases suggest that with a well designed and crafted game, learning writing formal specifications with fun should be possible. The STV Case suggests that playing does lead to learning, but it also shows that students are sensitive to usability issues (in fairness, most game players probably are), while the UCM Case confirms that fixing usability issues visibly improves the game enjoyment. Usability should not be underestimated, but it is also something that is difficult to address through a research team. Professional, or at least community based, game development and maintenance are likely needed to support long term and structural use of deep gamification like FormalZ in education. Balance is always important in games, and education games are no exception. Finding a good balance is not trivial. In FormalZ it is the balance between its training goal and its more playful aspects such as defense towers, but also the overall balance of the game embedding in a course. If the goal is to let students learn and to have fun while doing it, then space and opportunities should be created to let them have the fun; it is not something we can just pluck from the air.

*Acknowledgement:* We are very grateful to EU Erasmus+ for the funding they provide. We also thank Facultad de Informatica UCM for facilitating the FormalZ tutorial at UCM.

## References

[1] J. M. Spivey, "An introduction to Z and formal specifications," *Software Engineering Journal*, vol. 4, no. 1, pp. 40–50, 1989.

[2] D. L. Parnas, "Really rethinking formal method," *Computer*, no. 1, pp. 28–34, 2010.

[3] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, 2019.

[4] K. Robson, K. Plangger, J. H. Kietzmann, I. McCarthy, and L. Pitt, "Is it all a game? understanding the principles of gamification," *Business Horizons*, vol. 58, no. 4, pp. 411–420, 2015.

[5] I. S. W. B. Prasetya, C. Q. H. D. Leek, O. Melkonian, J. t. Tusscher, J. van Bergen, J. M. Everink, T. van der Klis, R. Meijerink, R. Oosenbrug, J. J. Oostveen, T. van den Pol, and W. M. van Zon, "Having fun in learning formal specifications," in *Proc. 41st Int. Conf. on Software Engineering (ICSE)*. IEEE, 2019.

[6] N. Tillmann, J. de Halleux, and T. Xie, "Pex for fun: Engineering an automated testing tool for serious games in computer science," MSR-TR-2011-41, Tech. Rep., 2011.

[7] Š. Korečko and J. Sorád, "Using simulation games in teaching formal methods for software development," in *Innovative Teaching Strategies and New Learning Paradigms in Comp. Prog.* IGI Global, 2015.

[8] B. Clegg, J. M. Rojas, and G. Fraser, "Teaching software testing concepts using a mutation testing game," in *Proc. of the International Conference on Software Engineering : Software Engineering and Education Track (ICSE-SEET) 2017*. IEEE Press, 2017, pp. 33–36.

[9] A. K. Boyce, "Deep gamification: Combining game-based and play-based methods," Ph.D. dissertation, North Carolina State Univ., 2014.

[10] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[11] M. I. Norton, D. Mochon, and D. Ariely, "The ikea effect: When labor leads to love," *Journal of consumer psychology*, vol. 22, no. 3, 2012.

[12] S. Papert and I. Harel, *Constructionism.* Ablex Publishing, 1991.

[13] Y. B. Kafai, *The Cambridge Handbook of the Learning Sciences*. Cambridge University Press, 2005, ch. Constructionism.

[14] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.