# Having Fun in Learning Formal Specifications

I.S.W.B. Prasetya , Craig Q.H.D. Leek , Orestis Melkonian , Joris ten Tusscher , Jan van Bergen , J.M. Everink , Thomas van der Klis , Rick Meijerink , Roan Oosenbrug , Jelle J. Oostveen , Tijmen van den Pol , Wink M. van Zon

*Dept. of Information & Computing Sciences, Utrecht University*

***Abstract*—There are many benefits in providing formal specifications for our software. However, teaching students to do this is not always easy as courses on formal methods are often experienced as dry by students. This paper presents a game called FormalZ that teachers can use to introduce some variation in their class. Students can have some fun in playing the game and, while doing so, also learn the basics of writing formal specifications. Unlike existing software engineering themed education games such as Pex and Code Defenders, FormalZ takes the deep gamification approach where playing gets a more central role in order to generate more engagement. This short paper presents our work in progress: the first implementation of FormalZ along with the result of a preliminary users' evaluation. This implementation is functionally complete and tested, but the polishing of its user interface is still future work.**

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

## I. Introduction

In the world of fast churning software industry, we might wonder whether applying formal methods is a viable option, since formal proofs are arguably hard to produce. Even if we can get programmers with the needed mathematical skill to produce them, the process is too slow to keep up with the pace of modern agile development. On the bright side, we do not need to exercise the full scale of formal methods to reap its benefit. Much can already be gained just by writing formal specifications. This does not require sophisticated mathematical skills, and nowadays not even a separate specification language and a separate tool chain anymore. Many modern programming languages support $\lambda$-expressions, which allows, for instance, predicate logic formulas to be expressed natively in the programming languages themselves.

Having formal specifications enables verification through automated testing or bounded model checking[1]. While this benefit might be clear in the eyes of a computer scientist, convincing practitioners to write formal specifications is still not easy. The myth that any form of exercising formal methods requires sophisticated math remains, and this is perhaps also a message that programmers somehow picked up from their education, where lectures in formal methods tend to be terse and dry. Students receive more points from being able to construct formal proofs. Demonstrating the ability to write formal specifications receives less points, hence creating the perception that it is less important (whereas we just argued that it is a more usable skill).

This paper presents a browser-based education game called FormalZ that teachers can use to break the typical monotony in a class on formal method by allowing students to have some fun playing the game, while also learning the basic of writing formal specifications. Unlike existing software engineering themed education games like Pex [1] and Code Defender [2], FormalZ takes a deeper gamification approach [3], where 'playing' is given a more central role. After all, what makes games so engaging is not merely the awarded scores and badges, but primarily the experience of playing them. The ultimate research question that intrigues us is whether such an approach will actually make a difference towards the game's ultimate learning goal. This is still on-going work. Currently, we have a fully functional implementation of FormalZ, but polishing the user interface is still future work. In this short paper we will present its game concepts and the result of a preliminary users' evaluation.

## II. FormalZ Game Concepts

In FormalZ, teachers provide exercises in the form of program headers and informal descriptions of the intended program behaviour. For each exercise, the student is asked to translate the informal description into formal pre- and post-conditions. An example of such an exercise is shown below:

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

The teacher accompanies the exercise with a solution in the form of a formal specification, written in a Domain Specific Language (DSL) that closely resembles predicate logic formulas. The DSL is embedded in Java, so the teacher can simply use Java SDK to test the solution (if it reflects what he/she has in mind), before deploying the exercise to the class. Section II-C will provide more details on this DSL.

A student solves the exercise by offering a pre- and post-condition which are equivalent to the teacher's solution. FormalZ frames this in a game of defending a CPU, which is a re-interpretation of the popular tower defence genre of games. The CPU symbolizes the program that is being specified. The story line is that hackers manage to find a way to influence data

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

[1]While one can employ these techniques in the absence of a formal specification, only the correctness with respect to general properties, such as absence of crash or abnormal CPU usage, can be verified.

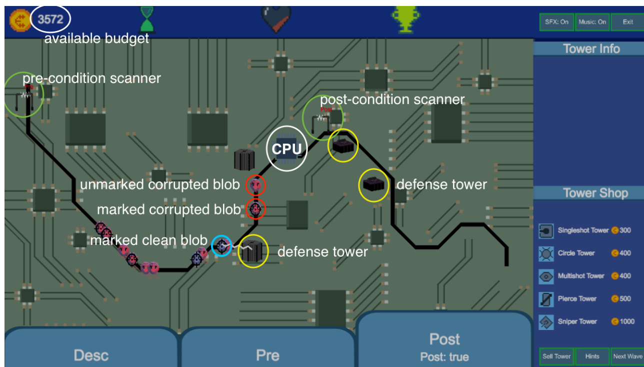



Fig. 1. *A screenshot of FormalZ. The CPU that has to be defended is in the middle. The small red and blue blobs represent data coming to or leaving the CPU. The green circles are the pre- and post-condition scanners, used to mark blobs — we can see that some red blobs are left unmarked (not good), and some blue blobs are marked (also not good). Defence towers (yellow circles) must be placed to shoot down marked blobs; one tower can be seen as 'zapping' at a wrongly marked blue blob. More towers can be placed to get higher score, subject to the available budget (top left).*

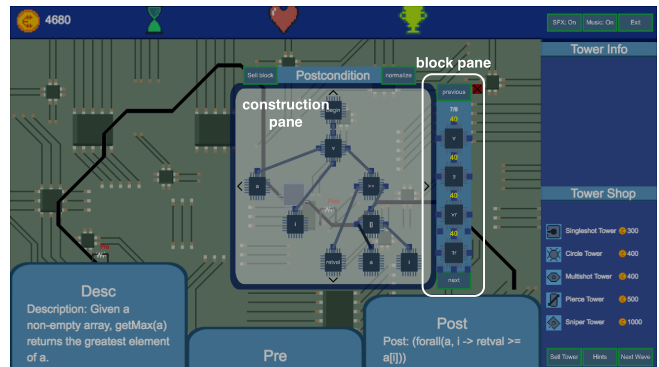



Fig. 2. *The construction pane of FormalZ. The user can construct a pre- or post-condition by dragging blocks, essentially the AST of the formula that the user has in mind.*

packages that flow into and out from the CPU. Data packages are represented by blobs, see Figure 1: red blobs are data that the hackers manage to corrupt to incorrect values, and blue blobs are those that are still 'clean'. No incoming red blob should reach the CPU, nor leave the CPU to reach the environment. To defend against this attack, the CPU's circuit board provides two scanners, one on the input side of the CPU, and one on its output side. Both can be programmed to identify and mark certain blobs. Other hardware called 'defence towers' can be added to the circuit board to discard marked blobs.

The scanners symbolize the pre- and post-conditions that the student should construct. An incorrectly programmed scanner would leave some red blobs unmarked, and may also wrongly mark blue blobs. The latter is also bad, since they represent good inputs or outputs, which the towers would subsequently wrongly discard.

### A. Constructionism

To construct the pre- and post-conditions the student gets a special construction panel where blocks can be placed to construct a formula; see Figure 2. Each block represents either a variable, a constant or an operator. Wires are used to connect the blocks to construct the tree representation of the formula; its so-called Abstract Syntax Tree (AST). Simply typing the formula is deliberately disallowed; let us motivate this from the perspective of the Constructionism theory of learning [4].

The theory believes that humans learn by constructing knowledge, rather than by simply transferring this knowledge from a teacher into the head of a watching learner. Familiar physical objects play a key role in this process, because the learner already has knowledge on how they work [5]. When new knowledge is framed in terms of interactions with these familiar objects, it helps the learner to construct the new knowledge in his mind. The theory was originally proposed by Papert and Harel [4]. 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 concepts that do exist in the physical world (as opposed to e.g. a "variable" which does not really have tangible physical existence) and can be assumed to be 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.

### B. Deep gamification

Introducing gamification just by adding game components like points, badges and leaderboards (what in [3] is called shallow gamification), would miss some key aspects such as play and fun, which are important to make a game engaging [6]. For example, awarding a badge to a student for completing a certain task acknowledges a certain achievement, but it does not mean that the task is fun to do. Making the task fun would evoke more engagement, which we believe will improve learning as well (an opinion also shared by Prensky in his classic work "*Digital Game-Based Learning*" [6]).

Although constructing formulas using blocks is to some degree fun, it is still a quite formal task as it has to abide by a whole set of rules (e.g. we cannot connect an `&&` block to an integer block, nor connect it to more than one parent) and is therefore not really 'playful'. An important characteristic of a play, as Caillois [7] puts it, is that it is not obligatory; if it were, it will lose its joyous quality. Although playing FormalZ is not obligatory (at least, we do not envision it to be so), having more rules does evoke some sense of being obliged to do things in a certain way. To create play, FormalZ

therefore frames the challenge of constructing pre- and post-conditions as a defence game where the goal is to prevent corrupted blobs from reaching their destination, while letting through as many clean blobs as possible. To do this the player also has to strategically place defence towers on the circuit board to shoot and remove marked blobs. There are different kinds of towers, each with unique features. The selection and placement of the towers are subject to almost no restriction. The student is free to experiment to figure out which strategies lead to better scores, though ultimately best scores are only attainable with the help of correct pre- and post-conditions.

### C. Java DSL

To specify pre- and post-conditions in a model solution, the teacher must write them in a DSL. The main design criterion was easy integration with the Java ecosystem to allow teachers to test out solutions using the Java SDK and enable future extensions where students are also asked to implement the specified program. To this end we chose to embed the DSL in Java, rather than follow a more extrinsic approach (e.g. custom Java pre-processor). To specify pre-conditions or post-conditions, we use the `pre` and `post` functions, supplied with a **`boolean`** expression as the argument. These expressions must be Java expressions, constructed using known boolean operators such as negation (`!`), conjunction (`&&`) and disjunction (`||`). To aid readability, we allow multiple pre/post statements, which are conjunctively interpreted (e.g. `pre(f1);pre(f2)` $\equiv$ `pre(f1&&f2)`).

The DSL also allows implication (`imp(f1,f2)` $\equiv$ f1 $\Rightarrow$ f2) and polymorphic equality (`==`) between a fixed universe of types, namely integers (**`int`**, **`short`**, **`long`**), reals (**`float`**, **`double`**), and multi-dimensional arrays of the previous types (e.g. **`float`**`[][]`). Any equality check on expressions of a different type is equivalent to **`false`**, except when Java semantics allow coercions between different numeric representations.

The usual representation-agnostic numeric operations are supported: `+`,`-`,`*`,`/`,`%` and comparisons (`<`, `<=`, `>`, `>=`).

To allow formal specifications of programs working on arrays, we additionally support the following array operations:

- Retrieving an array's length, as in `a.length > 0`.
- Indexing an array, as in `a[5] == 0`.
- Universally/existentially quantifying over the elements of an array, as in `forall(a, i -> a[i]==0)`.

Notice that we use $\lambda$-expression `args->body`, which was added to Java since Java-8, to model universal/existential quantification. Below is a possible specification of the exercise shown in Section II:

```
public static void getMax_spec(int[] a) {
  // pre-conditions
  pre(a != null);
  pre(a.length > 0);

  // call to actual implementation
  int retval = getMax(a);

  // post-conditions
  post(exists(a, i -> a[i] == retval));
  post(forall(a, i -> a[i] <= retval));
}
```

### D. Checking specifications

Given a teacher specification $M$ and a student specification $S$, FormalZ's backend will try to determine whether the two specifications are *semantically* equivalent: $M \equiv S$. While this is ultimately what the student is aiming for, the backend also provides helpful feedback in the case of an incorrect student solution. Specifically, when $M \not\equiv S$ it also gives the following information:

- If the student solution is too strong: $S \Rightarrow M$
- If the student solution is too weak: $M \Rightarrow S$
- Which of these combinations are satisfiable: $M \wedge S$, $M \wedge \neg S$, $\neg M \wedge S$, $\neg M \wedge \neg S$. This information controls the generation of the blobs and their marking. E.g. if $\neg M \wedge S$ is satisfiable the game will generate at least one unmarked red blob, and if $M \wedge \neg S$ is satisfiable then at least one marked blue blob is generated.

FormalZ implements two backends.

*Z3:* The principal Z3 backend converts the abstract syntax tree (AST) of Java expressions to the AST used by the Z3 theorem prover [8]. We can then freely invoke the Z3 solver to *try* to prove that the queried logical formula is actually true.

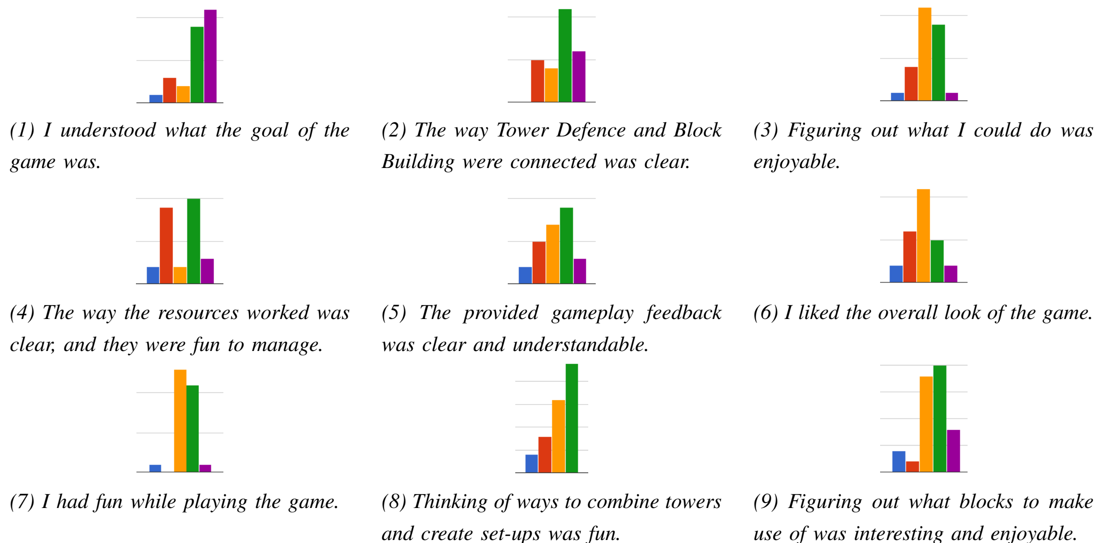
*Random testing:* While most teacher examples are expected to lie in Z3's decidability range, in general they are undecidable. So, we also provide a second backend that employs random testing to approximate the aforementioned truth-values.

The random testing backend first looks at $M$ and $S$ to identify clauses that are present in both, possibly in different but equivalent forms (e.g. the teacher wrote $a \Rightarrow b$ whereas the student wrote $\neg a \vee b$). It then removes these clauses, since they cannot possibly influence the validity of $M \equiv S$, and would result in simpler formulas on which an randomized equivalence test can be performed more quickly.

After the elimination step, the random testing backend starts a repeating process. In every iteration of the process, it first generates random values for all variables present in either $M$ or $S$, including non-primitive variables like (multidimensional) arrays. It then substitutes these variables in both $M$ and $S$ for the generated random values and evaluates the two formulas. During the evaluation step, universal and existential quantifiers are only partially evaluated, meaning that only a limited number of iterations of the quantifier are actually evaluated, for the sake of running time. After the evaluation step, $M$ and $S$ will reduce to a single boolean literal, from which the backend concludes whether $M \equiv S$ is satisfied, or if that is not the case, which other cases (see above) do apply. This approach is repeated multiple times to increase the confidence, and the combined results are returned by the backend.

## III. Preliminary Evaluation

To evaluate how our concept worked with students we ran a playtest with students that were taking the bachelor course Software Testing and Verification at Utrecht University. This group was chosen as this is the typical target audience for the game. They played a previous version of the game and



(1) *I understood what the goal of the game was.*

(2) *The way Tower Defence and Block Building were connected was clear.*

(3) *Figuring out what I could do was enjoyable.*

(4) *The way the resources worked was clear, and they were fun to manage.*

(5) *The provided gameplay feedback was clear and understandable.*

(6) *I liked the overall look of the game.*

(7) *I had fun while playing the game.*

(8) *Thinking of ways to combine towers and create set-ups was fun.*

(9) *Figuring out what blocks to make use of was interesting and enjoyable.*

Fig. 3. *The evaluation results. Each question takes the form of an assertion, to which a student can disagree (blue –left most bar), somewhat disagree, neither agree nor disagree, somewhat agree, or agree (purple –right most bar).*

feedback received in this session has already been incorporated in the game as it is now. After they played the game for approximately 30 minutes, including a short tutorial they could take to get introduced to the basics of the game, they were asked to complete a questionnaire. This resulted in 26 responses. Key results are shown in Figure 3.
We can see that the ultimate goal of the game was understood by most students (graph 1), as well as a majority being able to see the link between the separate key elements (defence towers and block building, question 2) of the game. However when we went deeper into the different aspects of the game, it was clear that students struggled slightly more with understanding how everything worked. An example of this is question 4 on resource (e.g. money and towers) management where the result shows a clear split: 13 people agreed to an extend, while 11 disagreed. We are not sure what exactly causes this split; it might be related to prior gaming experience of the students. FormalZ's way of giving feedback does resonate with students in general, as they were positive on average about it (graph 5). Improvement is still called for, as quite a few people also did not think the feedback was clear. Students were less positive about the overall look of the game (graph 6). This is something that is actively worked on as a priority.

The students are positive about having room to figure things out themselves (that is, to be allowed to 'play', which is a point deep gamification tries to put more emphasis on), as is shown in graph 8 for the Tower Defence aspect specifically and graph 9 for the Block Building. Overall, graph 7 shows that students are neutral or positive about the enjoyability of the game on average.

## IV. Conclusion and Future Work

We presented a fully functional implementation of the game FormalZ, which teachers can use as an alternative medium to help them teach students how to write formal specifications. FormalZ adopts a combination of the constructionist and the deep-gamification approaches. Our preliminary evaluation indicated that most our subjects perceived the approach positively. Despite the greater emphasis on the 'play' element, most subjects at least understood what the game's goal was. We cannot however confirm yet, if such an approach would indeed improve the students' learning. This requires further experiments, which for now are left as future work. Additionally, the game needs some cosmetic improvement to make it look more pleasing and presentable; this is also future work.


## Acknowledgement

We thank Sergey Sosnovsky for his useful feedback throughout the development of FormalZ.


## References


[1] 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, March, Tech. Rep., 2011.

[2] J. M. R. Benjamin Clegg 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.

[3] A. K. Boyce, "Deep gamification: Combining game-based and play-based methods." Ph.D. dissertation, North Carolina State University, 2014. [Online]. Available: https://repository.lib.ncsu.edu/bitstream/handle/1840.16/9788/etd.pdf?sequence=2

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

[5] Y. B. Kafai, *The Cambridge Handbook of the Learning Sciences*. Cambridge University Press, 2005, ch. Constructionism, look this up at UU univ lib.

[6] M. Prensky, *Digital Game-Based Learning*. Paragon House, 2003, ch. Fun, play and games: What makes games engaging.

[7] R. Caillois, *Man, play, and games*. University of Illinois Press, 2001.

[8] 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.