Template for Game Based Exercises in Formal Structures
a Small Take Away from FormalZ

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

Abstract— **Computer science students often find doing exercises in writing formal structures to be boring, and yet this is a useful skill to have, both academically and practically. Providing some of the exercises as a game may help in making the learning process more engaging, while in addition is also useful as an automated trainer. However, developing an education game is quite an endeavour. Additionally, after the development we also have to maintain the system. This paper presents a template that can be instantiated for a given target formal system to give a minimalistic set of game concepts to turn formalization exercises in this formal system into game based exercises. The template is derived from a game called FormalZ, as we observe that its core game concepts are actually general and can be re-applied to other use cases. Along with the template this paper also proposes a light weight architecture that would allow game based exercises to be developed maintained with less effort.**

Note for the reviewers: this is an 'Idea Paper'.

I. Introduction

Teaching Computer Science often involves teaching formal theories such as predicate logic or automata theories to students. While students are usually excited on doing programming projects, they are often less excited when learning formal theories. Exercises are often offered to help students to master the materials, but compelling them to actually do the exercises often proves to be challenging. A good example is learning to write formal specifications. Formal specifications provide a reference for developers defining what they should build, and how to test the resulting programs. On the other hand, to actually urging people to write specifications is also not easy. Developers typically point out that using formal methods requires lengthy mathematical proofs involving mathematical notations and concepts alien to programmers; to quote Parnas [1]: "*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. Modern programming languages like Java or C# incorporate λ-expressions, allowing programmers to natively express mathematical concepts like $\forall$ and $\exists$. To check the specifications, there are technologies like bounded model checking [2], [3] and automated testing [4], [5] that can do this without requiring any formal proof. Indeed the used techniques are theoretically incomplete, but they do greatly strengthen manual testing, at almost no labour cost.

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

Just because all the above possibilities exist does not however mean that developers will actually be excited about them. The first opportunity to put down a proper knowledge and attitude foundation would be during their programming or software engineering education. Unfortunately, subjects related to formal method are often seen as boring by students, and too exotic to be actually be used in practice. If left uncorrected, the attitude will persist when the students later grow to become real developers.

A possible option is to implement selected exercises as computer games to provide students with a more exciting setup to do the exercises. The prospect is indeed enticing. However, one should be cautioned that developing an education game would require quite some investment. It is also hard to get it right the first time. The success of a game, and education games are no exception, often depends on non-functional quality, e.g. its look, the smoothness of its interface, and the overall game balance, which may require several iterations to get it right. On top of that, education games often need additional non-game related functionalities. E.g. teachers would want to be able flexibly deploy exercised to students, and then to track their progress. This implies the need of having some central service where a game application can connect to and obtain current exercises, and to send back its player statistics. This service needs to be developed as well, and maintained. Then along with it we typically also need a website, where teachers can manage the exercises (e.g. to create a new one, or edit existing ones) and their classes, and so on. Given these engineering challenges, advices e.g. in the form of good practices, experience, or design templates can save development time, or even help one avoids project failure.

In a recent project called IMPRESS we studied the use and development of education games to assist in teaching software engineering [6], [7], [8], [9]. One of the games developed in IMPRESS is a game called FormalZ to help Computer Science students to learn writing pre- and post-conditions of programs [9], [10]. The game features a graphical game client, a class and exercise management portal, and analytics, with the whole code base consisting of almost 19K lines of code. Although FormalZ has a specific use case, we notice that its underlying game concepts can be generalized to be applied to other kinds of formal exercises. In this paper we try to capture and explicate these concepts as a design template that we hope to be reusable and useful for the community to create other game-based learning tasks.

The presented template is a design template for *game-based exercises* in writing formal constructs. It can be instantiated for exercises on any formal structures (e.g. functions, finite state automata, LTL formulas) that have set semantics, and for which a decision procedure exists to check satisfiability. The template incorporates fund and score as both game elements and learning instruments, and also features abstract feedback which would be more suitable for games (as opposed to detailed feedback as one would get from e.g. a debugger). We will also discuss some high level architectural decisions when using the template to develop a full fledged educational game while minimizing the needed development and maintenance effort.

II. Gamifying Formalization Problems

Imagine that we want students to learn to write solutions in some formal language $\mathcal{F}$ —e.g. this can be the language of predicate logic for formulating pre- or post-conditions as in FormalZ [10], or a language for describing finite state automata. To help the students to achieve this learning goal we provide exercises.

Each exercise would formulate a problem. The students proceed by constructing their own solutions. In return, they get feedback on the correctness of their solutions. The process is iterated until the students can produce an acceptable solution. The teacher is assumed to provide a model solution $F{\in}\mathcal{F}$. The students are expected to formulate their solution F' in $\mathcal{F}$ as well. The solution is accepted as correct if it is equivalent to F, and else it is rejected. Obviously, to be able to produce automated feedback, even without gamification, we need to have a checker (to check if $F \equiv F'$), at least for the space of candidate solutions that students can possibly construct. A plain "yes/no" checker will make very poor feedback though; we will return to this later.

Let us now consider what extensions we can do to turn an exercise such as above into a game. Rather than discussing e.g. how the game should visually looks like or what kind of game technology we should use, let us focus on conceptual extensions. Once the concepts are set, developers can always design and implement their own visual interpretation of the game. This is comparable to e.g. the game Monopoly, that has well defined rules, and thanks to that it allows companies, hobyists, or even students to create their own Monopoly computer games. Each implementer can decide if he/she would prefer a Monopoly with a simple but functional graphics, or an elaborate one, depending on his/her own budget.

Every game should have rules, else there is no challenge to play it. So, to 'gamify' a learning problem, that is, to turn it into a game, we will need to add some rules Having more rules can make a game more interesting and strengthen players' sense of achievement, but we should not over-do it either. Having too many rules may make the game too distracting with respect to the original learning goal, or too complicated for its intended users. For example in FormalZ, the player is required to also defend his/her proposed F' against intrusions (attempts to show that it is wrong) by placing defense towers. Rather than just placing basic towers, the player can buy more sophisticated towers or upgrade existing towers. These features were deliberately introduced to allow users to wander off from the game's actual learning goal to simply have fun shooting down intruders. It turned out that most students never actually explore these features, which suggests that keeping the game close to the learning goals is preferred.

There are three game rules in FormalZ that are closely aligned to its learning goal. These rules are simple, and general enough to gamify any $\mathcal{F}$ exercise. We will discuss the first two below. The third one, that concerns how the game gives its feedback, will be discussed in the next section.

- **Construction cost rule.** The idea is that constructing a solution costs some fund, which the player only has in limited amount. To construct a formula in $\mathcal{F}$ we assume $\mathcal{F}$ to provide syntactical constructors/operators. This rule defines the price of using each operator. For example, if $\mathcal{F}$ is the language of predicate logic, the rule would specify the cost of using operators $\vee, \neg, \forall$ etc in a proposed solution F'.

  This rule gives extra depth to the game. By limiting the initial fund given to the player the game can signal to the player that the solution they are considering are just too complicated (they would use more operators than necessary, and because of that the players would run out of fund) and that they should thus consider a different approach. For example when asked to write a post-condition for the program $\mathsf{max}(x, y)$ some students came with a solution like this (retval below refers to the program's return value):

  $$(x > y \;\equiv\; \mathsf{retval}{=}x) \;\wedge\; (y > x \;\equiv\; \mathsf{retval}{=}y) \;\wedge\; (x{=}y \;\equiv\; \mathsf{retval}{=}x) \tag{1}$$

  The solution is correct, and in a way is also intuitive as it seeks to describe the post-condition in terms of three disjoint cases of max's inputs. However, it is unnecessarily complicated. If each operator costs 10 units, the solution below, which is 40 units cheaper, would be preferred:

  $$(x \geq y \;\equiv\; \mathsf{retval}{=}x) \;\wedge\; (y > x \;\equiv\; \mathsf{retval}{=}y) \tag{2}$$

  Also, for new players, lowering the price of some operators can be used to nudge them to use the right operators. In the above example, the use of $\equiv$ is unnecessary. The formula is also more difficult to understand due to the bi-directional nature of $\equiv$. Lowering the price for 'simple' boolean operators like $\wedge$ and $\vee$ would nudge the players towards the solution below, which is as short as the previous one, but avoids the use of directional operators like $\equiv$:

  $$(x \geq y \;\wedge\; \mathsf{retval}{=}x) \;\vee\; (y > x \;\wedge\; \mathsf{retval}{=}y) \tag{3}$$

  When the players become more experienced, doing the opposite would nudge them towards trying to find a different kind of solution as an extra challenge. For

example if we want players to explore the use of $\Rightarrow$ rather than sticking with simple boolean operators, we can make the the latter expensive to nudge them to look for a solution like the one below:

$$(x \geq y \;\Rightarrow\; \textsf{retval}{=}x) \;\wedge\; (y > x \;\Rightarrow\; \textsf{retval}{=}y) \quad (4)$$

These examples also exemplify 'street wisdom' that we know as teachers. They are not essential for understanding the main concepts, but can help to improve the student's appreciation towards the subject. In the end, formal specifications also need to be read by human (e.g. other developers), so simplifying them and considering different styles of formulation do matter. Sharing this wisdom manually (e.g. by writing them as comments in submitted paper solutions) is obviously very time consuming. But embedding it as a game would allow the students to discover that wisdom themselves, and having fun in learning it as well.

- **Scoring rule.** Games typically have a concept of 'score'. Scores add excitement and boost the players sense of achievement. But even in learning, score is also important as a tool to directly reward the learner for doing certain steps. In other words, in a learning setup score also provides some learning feedback.
  In our setup, obviously, a correct solution should be awarded with points. We can however put more information in the score. Above we suggest to use fund and pricing to steer players towards a certain type of solutions. So, the amount of remaining fund when a solution is proposed is a metric of how well the solution matches towards the direction the teacher had in mind. This should be awarded as well, e.g. by translating it to some bonus point to be added to the score. So, the score can be e.g. as follows:

$$\textsf{score} \;=\; \begin{cases} S_W * solved \\ + \; \textsf{min}(C * fund, S_W/2 \;-\; 1) \\ + \; \textsf{min}(S_t, S_W/2 \;-\; 1) \end{cases} \quad (5)$$

  where $fund$ is the amount of fund that the player has left, S_W is the base score earned when the player solves the problem, and $solved = 1$ if the player solves the problem, and otherwise it is 0. C is some constant multiplier defining how significant the fund-bonus is. The S_t is time-related bonus that we will explain later. Notice that above scoring scheme award points to the player, even if he/she does not solve the problem. This signals that trying is also part of learning, and therefore should be awarded. Building an incorrect solution, but using the right operators (according to the pricing policy) would still earn points. The scheme still guarantees though, that the score of an incorrect solution can never be higher than a correct one.
  To add a twist to the game, bonus points can be added to the score based on the time t the player needs to construct F', e.g. this can be:

$$S_t \;=\; D * \textsf{max}(t_{\textsf{max}} - t, 0) \quad (6)$$

  where $t_{\textsf{max}}$ is some estimated maximum time that would still warrant bonus, and D is a constant multiplier.

III. Feedback

Obviously providing a feedback that just says 'yes' or 'no' whether the proposed solution is correct is not very useful as it does not provide any suggestion how the student can improve his/her solution. This can be improved if $\mathcal{F}$ allows a set semantic and has a SAT solver. That is, there is a semantic function $[\![.]\!]$ such that for every $F \in \mathcal{F}$, $[\![F]\!]$ is a set. The solver is assumed to be able to find an $\bar{x}$ such that $\bar{x} \in ([\![F]\!] \cup [\![F']\!])/([\![F']\!] \cap [\![F]\!])$, if one exists. Here, F is the teacher's solution and F' is a student's proposed solution. Note that of no such $\bar{x}$ exists, it implies that $[\![F]\!] = [\![F']\!]$; so in other words, F and F' are semantically equivalent. If an instance of $\bar{x}$ exists, it is a witness of the misalignment of F' towards F (or a 'counter example' of the hypothesis that they are equivalent). This value can be offered to the player as feedback to help him/her improving F'. So far this is a quite common practice when a SAT solver is used in automated exercises, e.g. as used by the education tool Pex [11].

Examples of formal structures that satisfy the above requirement are the language of first order predicate logic (under some restrictions), LTL formulas, and finite state automata.

There is however one problem: such a feedback is rather low level and tedious to 'debug'. For example if $\mathcal{F}$ is a language for describing pre- and post-conditions of programs as in FormalZ, a counter example $\bar{x}$ would be a vector describing the value of each program variable of the target program. The player will then have to figure out which of these values is or are wrong. Additionally, just from the value of $\bar{x}$ itself we cannot see in what way $\bar{x}$ is a counter example. It could be a value that satisfies the model solution F, but disallowed by the student's proposal F', but it can also be the other way around. Of course, the student can check this out himself, but this will not work well in a game setup as that would introduce too much interruption to the game flow. Indeed, during our experiments with FormalZ students tend to ignore this feedback.

In FormalZ counter examples are optional feedback, which the user can get if he/she wants it. The primary feedback is given, essentially, in terms of four binary indicators [10] shown in Figure 1 which can easily be displayed as some graphical sprites or icons as in FormalZ, or as simply as texts if we have no graphical interface. These indicators provide faster signaling to the players and hence they benefit the game flow.

To support these indicators we do need $\mathcal{F}$ to support a $/$ (subtraction) operator. That is, if F and G are formulas from $\mathcal{F}$, so is F/G. Our intent is to use this operator to test if there is some part of F that over approximates G, namely by checking the non-emptiness of F/G. This operator does not have to be the 'natural' subtraction operator, but it should be consistent with our concept of equivalence. More precisely, it should satisfy:

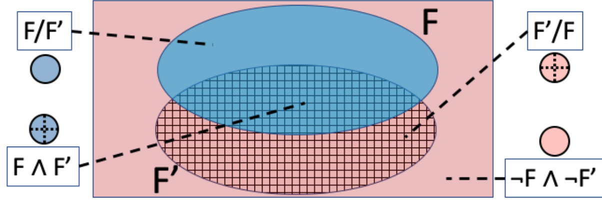

Fig. 1. *Four-indicators feedback a la FormalZ, each indicates if the corresponding partition is non-empty.*

$$[\![F]\!] = [\![F']\!] \quad \text{if and only if} \quad [\![F/F']\!] = \emptyset \ \text{ and } \ [\![F'/F]\!] = \emptyset \tag{7}$$

For example, if $\mathcal{F}$ is the language of predicate logic, then F/G can be defined as $F \wedge \neg G$. If $\mathcal{F}$ is the language of functions of type $\mathsf{int} \rightarrow \mathsf{int}$, we can define $[\![F/G]\!]$ as $\{(x, y) \mid x \in \mathsf{int}, F(x) > G(x)\}$.

Let us also assume $\mathcal{F}$ to contain an literal $\top$ such that $[\![\top]\!]$ is the set that covers the entire semantical domain of $\mathcal{F}$ (analogous to the literal `true` in predicate logic). This allows us to define few other operators:

$$\begin{array}{lcl} \neg F & = & \top/F \\ F \wedge G & = & F/\neg G \end{array}$$

$\mathcal{F}$'s SAT solver is assumed to be able check if $[\![F]\!]$ is empty or non-empty, for any $F \in \mathcal{F}$. The previously mentioned indicators are defined as follows:

- Indicate whether $[\![F \wedge F']\!]$ is non-empty. If so, this tells the player that at least some element in his/her solution is good. If this indicator signals negatively, it means that the player's solution is completely off.
- Indicate whether $[\![F/F']\!]$ is non-empty. If so, this tells the player that some aspect of his/her solution under approximates the real solution.
- Indicate whether $[\![F'/F]\!]$ is non-empty. If so, this tells the player that some aspect of his/her solution over approximates the real solution.
- Indicate whether $[\![\neg F \wedge \neg F']\!]$ is non-empty. Typically this set would be non-empty. But if it is empty, and assuming the player does not just submit the trivial $\top$ as his/her F', this tells that therefore F must be $\top$ itself.

Although each indicator is just a binary yes/no value, note that each carries additional information, e.g. whether the proposed F' is actually completely off, rather than just knowing that it is wrong.

Furthermore, despite their simplicity, these indicators now allow the player to build up his/her solution incrementally:

1) The player starts with some initial $F' = F'_0$. Let $k = 0$.
2) The player improves F'_k until F' does not under approximate F (so, until $[\![F/F']\!]$ is empty).
3) If F' also does not over approximate F (so, if $[\![F'/F]\!]$ is empty), the F' is correct and the player has solved the exercise.
4) Else, the player extends F' by adding a new fragment F'_{k+1}. Let $F' \leftarrow F' \wedge F'_{k+1}$ and $k \leftarrow k{+}1$. Go back to step 2.

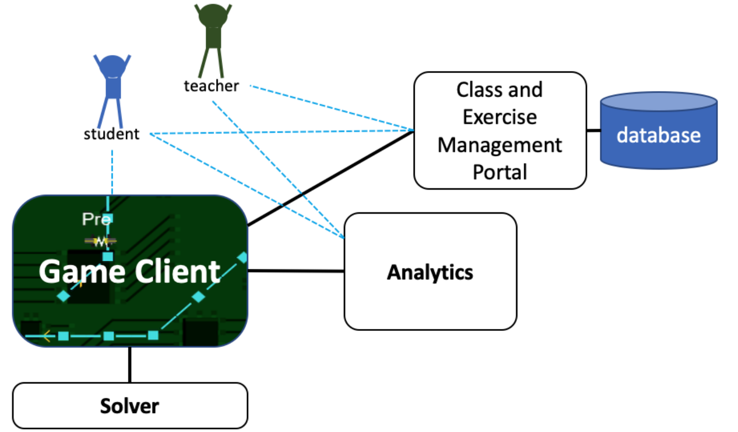

Fig. 2. *The architecture of the game FormalZ. The 'game' itself is a browser-based application (the 'game client' in the picture above, but it also needs the support of other remote services (the white boxes). The Management Portal is used to allow teachers to create classes, to author and deploy exercises, and to manage these classes and exercises. Users need to interact with all these components (indicated by the dashed blue lines) except the solver.*

Notice that the algorithm above allows the player to build his/her solution by adding one conjunctive fragment at a time, and focuses on only improving the new fragment. Note that the way of employing the SAT solver as was suggesting at the beginning of this section, although perhaps a more straight forward way of employing it, would not enable such an incremental learning process. Having indicators to signal partial over or under approximation is crucial for such a process.

IV. Architecture Consideration

The previous section has presented a template of gamified formalization exercises. For a given $\mathcal{F}$, the template can be fixed, and would then functionally and fully define a game. To turn it into an actual computer game, someone has to implement it. At that point we would then want to consider aspects such as the look and feel of the game, the interaction technology to use, and so on. We will put these aspects outside this paper's scope though, and will instead turn our attention to architecture. Architecture is also important, as choices made there may have a far reaching consequence.

In line with what was remarked in the Introduction, simply turning a set of game rules into a game implementation is typically not enough to make an education game. Whereas an ordinary game only needs to interact with its player, in an education game we also have a teacher which also needs to be somehow included in the system.

As an example of a full fledged education game, Figure 2 shows the architecture of FormalZ. Other education games such as Code Defenders [8] and Pex [11] also have a similar client-server architecture. As can be seen, it involves multiple components. The total code base consists of almost 19K lines

of code, developed in a semester by a team of 12 students including two artists. Additionally, the components in white are services. So they also need regular maintenance effort to keep them running. It is fair to say that developing, and after that also maintaining, an education game like FormalZ is not a light weight endeavor.

To foster the creation of games based of the template proposed before we therefore also propose a simpler architecture. The least complicated component in FormalZ is actually the Class and Exercise Management Portal. This is a website where teachers can create a class and add students to this class, and create exercises. Students can browse through available exercises and launch the game client by selecting one. Students' solutions and scores are also stored in a database managed by this component. Yet, the component takes 25% of the code, and furthermore also requires regular maintenance. In the simplified architecture in Figure 3 we propose to remove this component. Rather than having this separate management component the users are now responsible for managing their own exercises.

In the simplified architecture, the teacher can create an exercise in plain text, so there is no need to have a special content creation software. He/she can publish the exercises in his/her own website or mail them to the students. Each exercise would contain the textual description of the exercise, a unique exercise id, and is 'hashed'. The latter means that it also contains an encrypted solution that only the solver can open, and furthermore also allows the solver to verify that the solution indeed belongs to the said exercise.

To work on an exercise a student simply loads a hashed exercise into a game client. Each time the student proposes a solution, the game will need a solver to evaluate the proposal. In this architecture, the solver is assumed to be deployed as a service. To check a proposal the game client will send a tuple (i, E, F') where E is the exercise itself, F' is the proposed solution, i is some anonymized id of the student, and may furthermore identify the class to which he/she belongs to. The tuple is hashed so that only the solver check its integrity. In addition to sending back feedback, the solver will also collect all incoming (i, E, F') packages, along with their scores. It only collects them, and does not try to manage them.

Notice the new architecture only requires one service which is only required to store incoming users' solutions. The responsibility to post-process the data is put on the users themselves (or to some party or application on their behalf). Users, and other services as well (e.g. an analytic service), can query the data collected by the Solver, and to present the data in anyway they want. E.g. students would be more interested in viewing his own data, whereas a teacher would be interested in the data of all students doing some given exercise. Privacy is protected as the users' actual identity never occurs in the collected solutions, and the integrity of the data is protected through hashing.

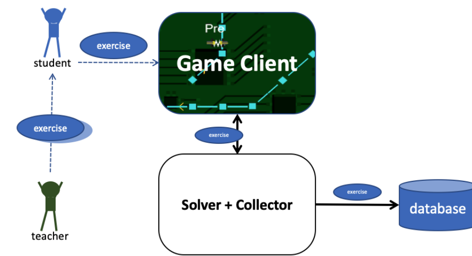

Fig. 3. *simple architecture*

V. Conclusion and Future Work

We have presented a general template and a minimalistic architecture for game based exercises for learning how to write solutions formally. Although these are inspired by our experience in developing a real education game, we have yet to try if this template would indeed work in practice. So, as future work we want to build a demonstrator as a proof of concept that a working education game can be created from the template without using excessive amount of effort. We can also compare it with e.g. FormalZ to see if a game created as such does not result in degrading level of user engagement.

References

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

[2] D. Kroening and M. Tautschnig, "Cbmc–c bounded model checker," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014, pp. 389–391.

[3] J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, and J. Weiser, "Divine 3.0–an explicit-state model checker for multithreaded c & c++ programs," in *International Conference on Computer Aided Verification*. Springer, 2013, pp. 863–868.

[4] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," *ACM Sigplan Notices*, vol. 46, no. 4, pp. 53–64, 2011.

[5] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[6] T. E. Vos, I. Prasetya, G. Fraser, I. Martinez-Ortiz, I. Perez-Colado, R. Prada, J. Rocha, and A. R. Silva, "Impress: Improving engagement in software engineering courses through gamification," in *International Conference on Product-Focused Software Process Improvement*. Springer, 2019, pp. 613–619.

[7] I. J. Perez-Colado, D. C. Rotaru, M. Freire-Moran, I. Martinez-Ortiz, and B. Fernandez-Manjon, "Multi-level game learning analytics for serious games," in *10th Int. Conf. on Virtual Worlds and Games for Serious Applications (VS-Games)*, 2018.

[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] 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.

[10] I. S. W. B. Prasetya, C. Q. H. D. Leek, R. Oosenbrug, P. Kostic, and M. d. Vries, "Can learning formal specification be fun? —experience and perspective," in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2020.

[11] 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.