

RULER: Prebugging with Proxy-Based Programming

Alexander Repenning
School of Education
FHNW Switzerland
Brugg-Windisch, Switzerland
alexander.repenning@fhnw.ch

Ashok Basawapatna
Mathematics, Computer & Information Science
SUNY Old Westbury
Old Westbury, Long Island, New York, USA
basawapatnaa@oldwestbury.edu

Abstract—While block-based programming has successfully eliminated critical syntactic barriers to programming, it remains unclear how effectively it aids in overcoming semantic, logical, and pragmatic programming challenges that hinder computational thinking. These challenges are likely to far outweigh the syntactic ones. With the goal of creating a highly accessible programming tool for young students using mobile devices, we explored the concept of pragmatic prebugging to begin addressing these challenges. By pragmatic prebugging, we refer to proactive debugging tools designed to prevent logical errors. This article introduces RULER.game as a Computational Thinking Tool with built-in pragmatic prebugging, enabling novice programmers to create games through a paradigm we call proxy-based programming. A small study exploring error rates found statistically significant performance improvements of proxy-based programming compared to block-based programming.

Keywords—block-based programming, computational thinking, programming by example, mobile computing.

I. INTRODUCTION

While block-based programming has removed critical syntactic barriers from programming, it is not clear how much it has contributed towards making computational thinking (CT) truly accessible [61]. If the goal is to teach computational thinking by creating personally interesting artifacts such as games [29], robots [1, 55] and simulations [40], then how many of the overall problem solving challenges are fully addressed by syntactic support mechanisms? Block-based programming is often compared to the ease of use of LEGO (e.g., [38]). However, if a person understands how to snap together two LEGO pieces, and then decides to build a replica of the Taj Mahal, they might soon realize that this “skill” does not automatically scale into the competence to create sophisticated artifacts. Benefits of block-based programming are claimed to include being “intuitive and self-explanatory” [72]. Many block-based programming designs intentionally suppress error messages [24] or provide limited debugging support [10]. This can produce programming errors [4] resulting from logical and pragmatic programming challenges. Most importantly, these kinds of challenges are likely to outweigh the syntactic ones.

Not only are non-syntactic problem solving challenges vast in magnitude, but research also does not have a full understanding of what the precise nature of these challenges really are. Early on, Knuth in his seminal analysis of the root problems of programming bugs uncovered nine problem-independent categories [31]. Only one of these categories is connected to syntax. This early research did not focus on novice

programmers, however, it becomes clear that even if syntactic challenges are completely removed by block-based programming the remaining challenges are critical and need to be addressed for advanced and novice programmers alike. More recently Ben-Yaacov et al. explored the types of errors kids make when using block-based programming languages [4]. They observed many logical bugs, not connected to syntax, when students programmed simple microworlds [42, 45].

Enabling students to create interesting projects [53] is not limited to the acquisition of programming skills but also requires debugging skills including the mastery of debugging tools [34]. The notion of “prebugging” [48, 69] refers to debugging approaches *proactively preventing bugs*. Debugging, in contrast, is reactive and generally considered “an activity that comes after testing” where one finds out where the error exists and how to fix it [34]. The proactive nature of prebugging may be particularly relevant to novice programmers engaging in ineffective debugging strategies such as trial-and-error [36]. Eisenstadt [12] points out that debugging becomes particularly difficult when there is a large temporal as well as spatial chasm between the root cause and the symptom. Block-based programming could be considered *syntactic prebugging*, crossing the syntactic challenges part of the Eisenstadt chasm by proactively preventing syntactic errors. But how can programming environments for children address the more complex aspects of the Eisenstadt chasm dealing with bugs emerging from logic, and pragmatic misconceptions?

Our research explores *pragmatic prebugging* as meta-design [13] philosophy to make programming more accessible and less error prone to novice programmers. In the field of linguistics “pragmatics” refers to *the understanding of what words mean in the context of specific situations* [60]. In computer science we conceptualize pragmatic prebugging as *the proactive support of programmers to understand what programming instructions mean in the context of specific situations*. By situation we mean the state an object is in, including its attributes, and its surrounding microworld.

RULER.game is a new tool implementing pragmatic prebugging through, what we call, *proxy-based programming* (PBP). Proxy-based programming introduces a number of programming affordances [51] not found in typical block-based programming. Most notably, there is the notion of a *proxy* object helping programmers to overcome the Eisenstadt chasm by proactively visualizing the consequence of programming decisions by unifying, typically separate [34], programming and debugging interfaces. The research question of this paper explores the error prevention efficacy of PBP: What is the efficacy regarding the prevention of non-syntactic errors provided by proxy-based programming?

II. RELATED WORK

Proxy-Based Programming (PBP) offers pragmatic programming support similar to programming by demonstration [7, 22] and programming by example [33]. In these demonstrational programming environments, users directly manipulate a situation through a microworld [45]. Demonstrational programming environments such as graphical rewrite rules (e.g., [65]) interpret changes of the situation by the user to generate a program as a collection of rules. For instance, the behavior of a digger (Figure 2) could be demonstrated by direct manipulation [26, 63] making the user rotate and move the digger on the road. A ubiquitous challenge [7, 18, 33] to demonstrational programming is finding the right level of abstraction. Automatically generated behavior may either be too specific, factoring in irrelevant details of a concrete situation, or, vice versa, may be too generic resulting in unwanted overgeneralizations. Some demonstrational programming environments mitigate the abstraction challenge somewhat by making the program visible and editable by users. Proxy-based programming flips the “change the situation” then “adjust the level of abstraction” process around. First users select from a set of conditions holding true in the current situation, and then they select contextualized actions to witness these actions performed in a sandbox by a proxy object.

When looking at proxy-based programming we must pay special attention to how we can better enable tool accessibility, program implementation, execution, and debugging. Live Programming, and mobile tools help develop the *four design principles* that the RULER.game adheres to (described in section III). Live Programming provides programmers immediate visual feedback about the behavior of their programs that helps to understand the cause-effect relationship in their code – an influential aspect for novices learning programming [68]. As reported by McDirmid [35], Live Programming environments enable real-time interaction with programming systems that can significantly improve productivity and make the activity of programming more engaging. However, Huang points out that Live Programming poses an “unavoidable risk” of overwhelming information overload [25]. Proxy-based programming mitigates information overload by focusing the user attention on a single object (the proxy, e.g., Figure 2, right) and only the limited scope of code being currently edited (a single rule).

Programming and debugging is particularly challenging on small mobile devices (see principle #3). There are a number of tools to create programs running on mobile devices such as smartphones and tablets. Some, such as MIT App Inventor [46], Thunkable [21], and AppyBuilder [70], are desktop-based. That is, programming takes place on desktop-based computers producing apps or project files which then need to be copied to smartphones. Tools more similar to RULER.game directly enable programming on smartphones using text- or block-based programming. Text-Based Programming environments for smartphones, aimed typically at high-school and undergraduate students, include Lua and Processing. Boateng [5] points out the ubiquity of smartphones in Africa as the main reason to select smartphones as an educational platform. He reported that by 2021 there would be nearly one billion smartphone users in

Africa. Furthermore, he also found that the use of smartphones for text-based programming of games was not a hindrance [5].

Block-based programming environments can be categorized into closed-project tools, such as LightBot [17] focusing on Hour of Code like puzzles [73], or open-project tools, such as Hopscotch [74], Pocket Code [64] and Scratch Jr. [72], helping users to create their own projects including the creation of their own characters by drawing them using image editors or using camera functions of smartphones. Most of the closed-project puzzle environments, but also Scratch Jr, show the code and the situation simultaneously. In contrast, Hopscotch and Pocket Code do not support the simultaneous view of code and situation. Scratch Jr. appears to have a lower threshold [43] than Hopscotch and Pocket Code.

While the history of debugging can be traced further back to the 1940s when relay-based computing devices were invaded by moths and “debugged” by Admiral Grace Hopper, the modern science of debugging [69] in the context of block-based programming starts as early as block-based programming itself. Glinert [16] proposed the idea of programming constructs represented as jigsaw puzzle pieces that could be snapped together. This could be considered *syntactic prebugging* proactively preventing the creation of syntactically incorrect programs. Debugging hinges on the knowledge and strategies that are necessary for successful debugging [34] including the ability to employ debuggers [59]. Most block-based programming languages provide little debugging support [10] and may intentionally suppress error messages [24]. Some block-based programming tools including Scratch [10], Scratch jr. [72], and Snap! [41], provide basic affordances to test programming blocks. Importantly, these debugging aids are not part of normal programming process. AgentSheets and AgentCubes feature additional static as well as dynamic code analysis tools [48] aiding the debugging process more proactively. Static code analysis [27] visualizes semantic problems such as unreachable code. Dynamic code analysis in AgentCubes [48] visualizes logical problems by showing how the control flow depends on the current situation. Proxy-based programming provides the notion of a proxy object, enabling safe tinkering and programming in a sandbox to overcome syntactic, semantic and pragmatic language challenges.

Finally, if CT education in schools is less about entering a professional computer science career pipeline and more about connecting CT to other K-12 subjects then it may make sense, just as suggested by Johnson [28], to raise programming to higher levels of abstractions. Tools such as Microsoft’s TileCode [2] and Thymio with its VPL programming language [39] employ higher level abstraction programming models that evolved from the concept of graphical rewrite rules [65]. With TileCode a simple Pac-Man game can be created with just a handful of rules edited on a tiny 160 x 120 pixel display. Similarly, RULER.game also features a rule-based programming approach [3, 8, 14, 15, 19, 47, 54] and makes it possible to write highly compact code, e.g., a 9 rule Pac-Man game including multi-agent path finding AI blocks [49], avoiding frequent code smells found in lower-level block-based programming languages such as Scratch [20].

III. PRINCIPLES OF PROXY-BASED PROGRAMMING

Proxy-based programming (PBP) is a new programming paradigm embodied in the RULER.game computational thinking tool [52, 56]. PBP can be considered an example implementation of the pragmatic programming philosophy. The goal of PBP is to make programming more accessible by establishing a highly transparent connection between the *situation* embodied in a microworld [42, 45] and the *program* currently being constructed by a programmer. Eisenstadt [12] urges to address this connection stating “More than 50% of the [programming] difficulties are attributable to just two sources: large temporal or spatial chasms between the root cause and the symptom, and bugs that rendered debugging tools inapplicable.” We define proxy-based programming through four *design principles*:

1. ***The impact of each programming step should be proactively and visibly reflected through an object called the proxy contextualized in the current situation, while avoiding unwanted side effects.*** The proxy is a recognizable copy of the object to be programmed, created when the user starts a programming cycle (e.g., Figure 2). The microworld (or situation) should be annotated in ways that enhance the perception of causality [37] by demonstrating the concrete consequences of executing instructions by the proxy. To prevent unwanted side effects, such as losing an object by accidentally deleting it, the microworld, including object attributes such as size, color, and rotation, is sandboxed. After the user concludes the programming cycle, the proxy's code is transferred into the original object to be programmed. The proxy is then removed, and in case of side effects, the microworld is automatically restored.
2. ***The situation should guide the programming environment.*** Instruction palettes should be demonstrational to reflect the current situation. Conditions found in a *demonstrational condition palette* should be situated, if possible, to reflect true facts. For instance, a condition querying the object ahead of the proxy should default to the actual object found in the current situation while the user is programming the proxy.
3. ***Program and Microworld should be visible at the same time.*** Mobile devices, with their small screens, tempt programming language designers to create interfaces that introduce problematic temporal or spatial chasms [12], separating the program and microworld into views that users must explicitly switch between. Instead, a programming interface should simultaneously display the program and code, with a focus on the relevant parts involved in a causal connection.
4. ***A proactive programming interface should unify code manipulation and debugging interfaces.*** The programming environment should not have to consist of different interfaces for implementation and run-time activities [34]. Users should not need to explicitly ask to enter a debugging tool or mode. While many block-based programming tools include some form of testing blocks there is little evidence that programmers use these mechanisms. With PBP users do not have to take the initiative to invoke testing functions.

IV. PREBUGGING WITH PROXY-BASED PROGRAMMING

While our ultimate goal is to teach computational thinking employing learning designs [32] based on constructionists principles [43, 44], we find it useful to start with basic coding puzzles [53]. The Kodetu computational thinking challenges have been carefully designed and evaluated as progressive set of puzzles aimed at early programmers [4, 11]. Figure 1 shows the RULER.game Home screen including versions of the Kodetu challenges. The Kodetu challenges will be described more in-depth in section V.



Fig. 1. The RULER.game Home Screen provides a Netflix-like Menu of Programming Tutorials including Kodetu Challenges.

A 2nd grade student, let's call her Alicia, has already advanced to the digger challenge 6. RULER.game opens a tutorial explaining that Alicia should program the digger by defining an IF/THEN rule. For the THEN part of the rule, Alicia is instructed to use only the **forward** ⤴, **turnLeft** ⤵ and **turnRight** ⤶ actions (Figure 2, right). These actions are inspired by the *turtle* commands [43] found in the Logo programming language [23, 30]. Similar actions can also be found in the controls of remote control cars or the 4 buttons of programming toys for kindergarteners such as the Bee-Bot [62]. Alicia taps the digger in the world to show its behavior consisting of an empty IF/THEN rule (Figure 2, left).

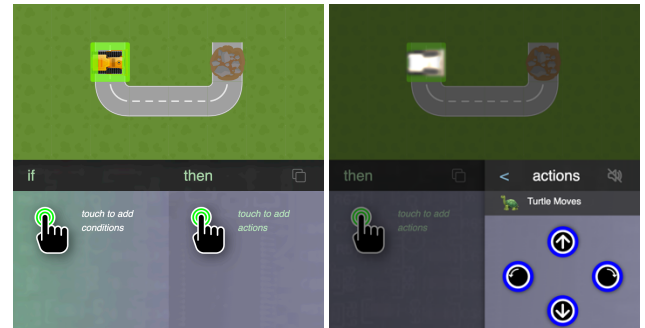





Fig. 2. Left: Tap digger to show code. Right: tap “then” part to create proxy (ghosted digger copy) and show action palette including turtle move actions.

Following the tutorial, she then taps into the THEN box to open the demonstrational action palette. This is when Alicia enters the *proxy programming sandbox*. A proxy object (Figure 2, right) is created by copying the digger. Alicia can now tinker [57] with the proxy by tapping actions in the palette without risking unwanted side effects. For instance, using the erase action will only erase the proxy while preserving the state of the original digger. Before world destructive actions are executed, the world is backed up so that it can be restored.

Alicia needs to first fully understand that the **forward** , **turnLeft**  and **turnRight**  actions operate from the perspective of the digger driver. For instance, selecting the **forward** action in Figure 2, right, would make the digger drive to the right. The purpose of the proxy is to aid imagination through body syntonicity [71], i.e., the idea of becoming the object one wants to program. Alicia quickly taps the **turnRight** and then the **forward** action to make the digger move into the corner. To better understand what each action does, perhaps Alicia cannot read yet, Alicia can enable spoken explanations to receive detailed descriptions. Additionally, while executing and moving the proxy, the action will be annotated with a short explanation tag (e.g. Figure 3 left).

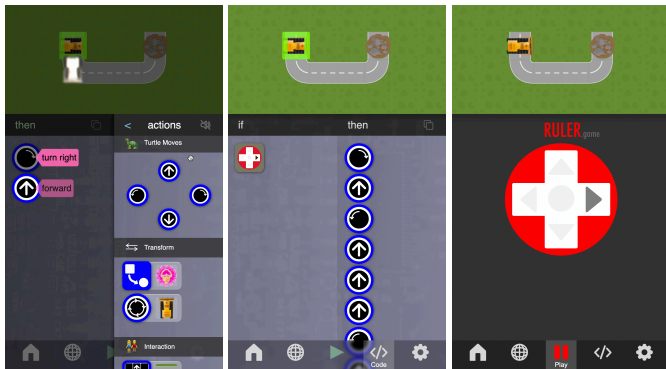


Fig. 3. Left: make proxy turn right, move forward. Middle: after adding D-pad condition. Right: play the game. D-pad shows which keys are programmed.

It is in this situation where she hesitates for a moment because she is not sure if the digger needs to turn left or right. From a bird's eye point of view the digger needs to move to the right. However, from the first person perspective, because the digger is facing down, it needs to turn left. In our usability study we found this stage to be challenging to many of the 10 year old kids. Often we would notice that kids would not only hesitate but also engage in body syntonic behavior such as turning their head, or the tablet, ever so slightly to better understand which way to turn. Alice correctly selects the **turnLeft** action and finishes the action sequence. She taps into the IF part of the rule to trigger the rule with a directional pad (D-pad) key (Figure 3, middle).

Pressing the play button Alicia can now run the complete program. The D-pad indicates that its right button has been programmed (Figure 3, right). After Alicia presses that button the digger successfully drives to the construction site.

Kwame, in 5th grade, has finished the Kodetu challenges and is ready for a more challenging project. He picks the “Maze Solver” project. Unlike with Hour of Code like puzzles, the goal is not to code one specific sequence of instructions to deal with

one specific maze but to become a more advanced computational thinker implementing a universal algorithm applicable to an infinite universe of mazes. The tutorial hints at the so-called “right hand rules” maze-solving algorithm.

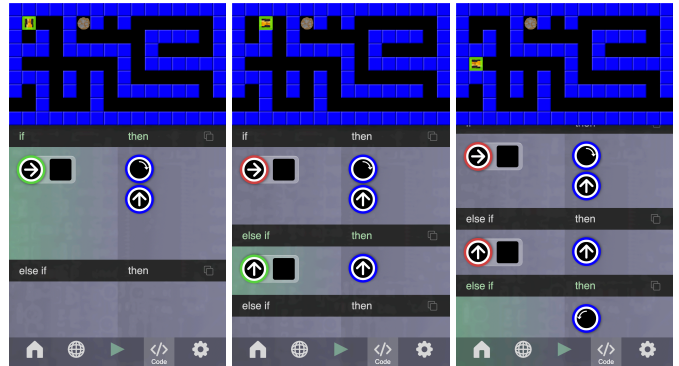


Fig. 4. Progressive Proxy-Based Programming. Left: rule #1 if empty right then turn right and move forward Middle: rule #2 else if empty ahead then move forward Right: else turn left

Engaging in *progressive proxy-based programming*, Kwame fluidly interweaves coding and prebugging to gradually program the right hand rules. Rule #1 is about turning right and advancing one step if the space to your right is empty (Figure 4, left). Kwame runs this rule but the digger is not getting far. Rule #2 is about going straight: If it is empty ahead of you then move forward (Figure 4, middle). These two rules make the digger advance all the way to a cul-de-sac (Figure 4, right). He codes rule #3 to rotate left twice (180 degrees) until Safiya, sitting next to him, points out a more elegant solution with just a single rotate left (Figure 4, right).

The demonstrational condition palettes, mentioned in PBP principle #2, help with the creation of algorithms. When the digger gets stuck in the maze in a particular situation, the demonstrational palettes offer true conditions reflecting the situation. These conditions become likely candidates to predicate pertinent control flow. When looking at the finished rules Kwame can differentiate true or false conditions by their color and recognize the control flow by the pertinent path subtly glowing green (Figure 4).

V. EXPERIMENTAL METHODS

To assess the error prevention efficacy of proxy-based programming we employed documented error rates of established block-based programming challenges as baseline data. The Kodetu challenges [11] employed in the example above have been analyzed by Ben-Yaacov et al. to categorize the types and frequencies of non-syntactic errors novice programmers make when using block-based programming [4]. We used these data as an empirical baseline to run an initial comparative study pilot. While the age group (10 – 12 year old elementary school children) as well as the duration of the interventions match, the sample sizes are different. The Ben-Yaacov study (n=123) is larger than our pilot sample size: 8 users, aged 10-12, with little if any programming experience (n=8, 4 males, 4 females).

Ben-Yaacov et al. defined errors as the event when executing the finished program resulted in a situation where the agent did not successfully reach the goal [4]. The 123

participants executed their programs a total of 2679 times across all challenges. Of these executions, 1033 (39%) resulted in errors. The study categorized logical errors as one of the following: *Counting*, *Orientation*, *Redundancy*, *Conditionals*, *Repetition*, *Premature*, and *Decomposition*. Due to the simple nature of exercises 1-6, the main errors encountered in the original and the comparative study were Counting and Orientation and so we will focus on these. Both Counting and Orientation can be traced back to a well-established body of literature. Counting is a classic example of a boundary, i.e. a “off-by-one” bug [66]. Orientation errors are geometrical misconceptions that can either be traced back to the lack of *preprogramming knowledge* identified by Bonar and Soloway [6] or the inability to correctly put themselves into the perspective of the object to be programmed (syntonicity [71].)

VI. RESULTS AND DISCUSSION

All students in our study noticed every error in their code, through all the exercises, using proxy-based programming. The authors of the original Kodetu study [4] provided us with additional data necessary to calculate statistical relevance for challenges 1-6. Table I compares block-based programming (syntactic prebugging) with proxy-based programming (pragmatic prebugging) rates for orientation and counting errors. Both error reductions, regarding orientation as well as counting, were statistically significant ($p < 0.05$). The total reduction of errors was a factor of 10.

TABLE I. SYNTACTIC AND PRAGMATIC PREBUGGING ERROR RATES

Prebugging	Errors		
	<i>orientation</i>	<i>counting</i>	<i>total</i>
Syntactic: Block-Based Programming	9.71%	13.37%	20.79%
Pragmatic: Proxy-Based Programming	0.00%	2.08%	2.08%

Provided the pronounced error reduction efficacy one needs to ask if proxy-based programming can be considered an effective educational scaffold [67] or should be considered just an crutch [9]? A teacher amazed about how quickly kids were flying through the Kodetu challenges wondered if maybe RULER.game had made these Hour of Code like puzzles too easy. On the one hand, the error reduction resulting from the immediate assistance of proxy-based programming is indicative of an effective affordance [51] to make programming more accessible. On the other hand, however, Roberson et al. [58] conjectured that approaches relying on immediate style interruptions, such as providing the kinds of clues offered by proxy-based programming, could result in an over-reliance on shallow problem-solving strategies. Further research needs to explore the trade-offs between temporary relief versus lasting skills. In other words, we do not really understand the longer term educational effects of proxy-based programming. If proxy-based programming acts like a scaffold with educational benefits then it should be possible to later fade the support of this scaffold with only small negative effects. If, however, proxy-based programming acts like a crutch, then the students’ performance would drop instantly to levels comparable in the Kodetu [4] study when the pragmatic programming support gets removed. Research could explore these more longitudinal learning effects of proxy-based programming through A/B testing contrasting options of RULER.game with and without fading of pragmatic prebugging.

Can proxy-based programming result in code smells [20]? We observed a number of students making orientation mistakes, realizing it instantly but then not fixing it cleanly. For instance, if the digger was programmed, by mistake, to turn left, then some students did not remove that false instruction but simply compensated the “bad” code with additional “good” code. That is, they would either continue turning, 3 times turning left is the same as 1 time turning right, or they would add 2 times turning the other way. In either case, according to the Kodetu challenge, this would not be considered an error. However, these programs include unwanted redundancy. This kind of behavior could either be blamed on tool affordance or on the lack of details provided in the description of the Kodetu challenges which did not explicitly ask for the most compact programming solution. Researchers are encouraged to explore these kinds of questions by using and extending the RULER.game software.

The goal of RULER.game, however, is not to create the ultimate Hour of Code like puzzle solving tool but to explore fundamental affordances [51] enabling novice programmers to create their own games. Affordances need to be combined with strategies to scaffold creative programming project [53]. 5th graders employed multi touch affordances, enabled on mobile devices, to build two player Whack-a-Mole games. Classes consisting of 1st - 4th graders drew their own game characters on paper, scanned them into RULER.game, designed mazes and programmed these characters. Another class with 1st - 4th graders used a new Artificial Intelligence block [49] to implement highly sophisticated pathfinding based on Collaborative Diffusion [50]. While most of these young students did not fully understand how the AI worked they did enjoy the resulting gameplay. In fact, with the ability for ghosts to track down Pac-Man optimally even through the most complex mazes, including portals, and the ghosts collaborating with each other, students quickly were quite busy dumbing down the ghosts AI by making ghosts move much slower to give the human player a fighting chance to win against the AI.

CONCLUSIONS

Proxy-based programming represents a significant evolution of block-based programming. It supports novice programmers not only in overcoming syntactic challenges but also in prevailing over the much more daunting semantic, logical, and pragmatic programming challenges. RULER.game offers pragmatic prebugging; that is, it affords proactive debugging support beyond syntax by employing a sandboxed proxy object that illustrates the concrete consequences of executing code in specific situations. A pilot study, which employed Kodetu programming challenges, compared error performance between block-based programming and proxy-based programming. The study found that the error rates of proxy-based programming were 10 times smaller and that the reduction of error rates in all categories explored was statistically significant.

ACKNOWLEDGEMENT

This work is supported by the Hasler Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the foundation.

REFERENCES

- [1] Ajaykumar, G., M. Steele, and C.-M. Huang, "A survey on end-user robot programming," *ACM Computing Surveys (CSUR)*, vol. 54, pp. 1-36, 2021.
- [2] Ball, T., S. Kao, R. Knoll, and D. Zuniga, "TileCode: Creation of Video Games on Gaming Handhelds," in *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, 2020, pp. 1182-1193.
- [3] Bell, B. and C. Lewis, "ChemTrains: A Language for Creating Behaving Pictures," in 1993 IEEE Workshop on Visual Languages, Bergen, Norway, 1993, pp. 188-195.
- [4] Ben-Yaacov, A. and A. Hershkovitz, "Types of Errors in Block Programming: Driven by Learner, Learning Environment," *Journal of Educational Computing Research*, vol. 61, pp. 178-207, 2023.
- [5] Boateng, G., V. W.-A. Kumbol, and P. S. Annor, "Keep Calm and Code on Your Phone: A Pilot of SuaCode, an Online Smartphone-Based Coding Course," in *Proceedings of the 8th Computer Science Education Research Conference*, 2019, pp. 9-14.
- [6] Bonar, J. and E. Soloway, "Preprogramming knowledge: A major source of misconceptions in novice programmers," in *Studying the novice programmer*, ed: Psychology Press, 2013, pp. 325-353.
- [7] Cypher, A., *Watch What I Do: Programming by Demonstration*. Cambridge, MA: MIT Press, 1993.
- [8] Cypher, A. and D. C. Smith, "KidSim: End User Programming of Simulations," in *Proceedings of the 1995 Conference of Human Factors in Computing Systems*, Denver, CO, 1995, pp. 351-358.
- [9] Daniel, S. M., M. Martin-Beltrán, M. M. Percy, and R. Silverman, "Moving beyond yes or no: Shifting from over-scaffolding to contingent scaffolding in literacy instruction with emergent bilingual students," *TESOL Journal*, vol. 7, pp. 393-420, 2016.
- [10] Deiner, A. and G. Fraser, "NuzzleBug: Debugging block-based programs in scratch," *arXiv preprint arXiv:2309.14465*, 2023.
- [11] Eguiluz, A., M. Guenaga, P. Garaizar, and C. Olivares-Rodriguez, "Exploring the progression of early programmers in a set of computational thinking challenges via clickstream analysis," *IEEE Transactions on Emerging Topics in Computing*, vol. 8, pp. 256-261, 2017.
- [12] Eisenstadt, M., "My hairiest bug war stories," *Communications of the ACM*, vol. 40, pp. 30-37, 1997.
- [13] Fischer, G. and E. Giaccardi, "Meta-Design: A Framework for the Future of End User Development," in *End User Development*, H. Lieberman, F. Paternò, and V. Wulf, Eds., ed Dordrecht, The Netherlands: Academic Publishers, 2006, pp. 427-457.
- [14] Fisher, G. L. and D. E. Busse, "Adding Rule-Based Reasoning to a Demonstrational Interface Builder," in *Proceedings of the ACM Symposium on User Interface Software and Technology*, Monterey, CA, 1992, pp. 89-97.
- [15] Gindling, J., A. Ioannidou, J. Loh, O. Lokkebo, and A. Repenning, "LEGOsheets: A Rule-Based Programming, Simulation and Manipulation Environment for the LEGO Programmable Brick," presented at the *Proceeding of Visual Languages*, Darmstadt, Germany, 1995, 172-179.
- [16] Glinert, E. P., "Towards 'Second Generation' Interactive, Graphical Programming Environments," in *IEEE Computer Society, Workshop on Visual Languages*, Dallas, 1986, pp. 61-70.
- [17] Gouws, L. A., K. Bradshaw, and P. Wentworth, "Computational thinking in educational activities: an evaluation of the educational game light-bot," in *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, 2013, pp. 10-15.
- [18] Halbert, D. C., "Programming by Example," Xerox Office Systems Division, Technical Report OSD-T8402 1984.
- [19] Hayes-Roth, F., "Rule-Based Systems," *Communications of the ACM*, vol. 28, pp. 921-932, 1985.
- [20] Hermans, F. and E. Aivaloglou, "Do code smells hamper novice programming? A controlled experiment on Scratch programs," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, 2016, pp. 1-10.
- [21] Hiasa, F., S. Supadi, E. Agustina, M. Afrodita, L. Lazfihma, and N. Yanti, "Development of android-based learning media assisted by Thinkable Applications in literary history courses," *BAHAstra*, vol. 43, pp. 221-233, 2023.
- [22] Hirzel, M., "Low-code programming models," *Communications of the ACM*, vol. 66, pp. 76-85, 2023.
- [23] Hromkovič, J., D. Komm, R. Lacher, and J. Staub, "Teaching with LOGO philosophy," *Encyclopedia of Education and Information Technologies*, 2019.
- [24] Hromkovic, J. and J. Staub, "The Problem with Debugging in Current Block-based Programming Environments," *Bulletin of EATCS*, vol. 135, 2021.
- [25] Huang, R., K. Ferdowsi, A. Selvaraj, A. G. Soosai Raj, and S. Lerner, "Investigating the impact of using a live programming environment in a CS1 course," in *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education-Volume 1*, 2022, pp. 495-501.
- [26] Hundhausen, C. D., S. Farley, and J. Lee Brown, "Can Direct Manipulation Lower the Barriers to Programming and Promote Positive Transfer to Textual Programming? An Experimental Study," in *VLHCC '06: Proceedings of the Visual Languages and Human-Centric Computing*, 2006, pp. 157-164.
- [27] Johnson, B., Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 672-681.
- [28] Johnson, M., "Generative AI and CS Education," *Communications of the ACM*, vol. 67, pp. 23-24, 2024.
- [29] Kafai, Y., "Playing and Making Games for Learning," *Games and Culture*, vol. 1, pp. 36-40, 2006.
- [30] Kahn, K., "Should LOGO keep going forward 1?," *Informatics in Education-An International Journal*, vol. 6, pp. 307-321, 2007.
- [31] Knuth, D. E., "The errors of TEX," *Software: Practice and Experience*, vol. 19, pp. 607-685, 1989.
- [32] Koper, R., "Current research in learning design," *Journal of Educational Technology & Society*, vol. 9, pp. 13-22, 2006.
- [33] Lieberman, H., *Your Wish Is My Command: Programming by Example*. San Francisco, CA: Morgan Kaufmann Publishers, 2001.
- [34] McCauley, R., S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander, "Debugging: a review of the literature from an educational perspective," *Computer Science Education*, vol. 18, pp. 67-92, 2008.
- [35] McDirmid, S., "Usable Live Programming," presented at the *SPLASH Onward!*, Indianapolis, Indiana, 2013.
- [36] Michaeli, T. and R. Romeike, "Current status and perspectives of debugging in the k12 classroom: A qualitative study," in *2019 IEEE Global Engineering Education Conference (Educon)*, 2019, pp. 1030-1038.
- [37] Michotte, A., *The Perception of Causality*. London: Methuen & Co. Ltd., 1963.
- [38] Mohamad, S. N. H., A. Patel, R. Latih, Q. Qassim, L. Na, and Y. Tew, "Block-based programming approach: challenges and benefits," in *Proceedings of the 2011 International Conference on Electrical Engineering and Informatics*, 2011, pp. 1-5.

- [39] Mondada, F., M. Bonani, F. Riedo, M. Briod, L. Pereyre, P. Rétonnaz, and S. Magnenat, "Bringing robotics to formal education: The thymio open-source hardware robot," *IEEE Robotics & Automation Magazine*, vol. 24, pp. 77-85, 2017.
- [40] Musaeus, L. H. and P. Musaeus, "Computational Thinking in the Danish High School: Learning Coding, Modeling, and Content Knowledge with NetLogo," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 2019, pp. 913-919.
- [41] Newley, A., H. Deniz, E. Kaya, and E. Yesilyurt, "Engaging elementary and middle school students in robotics through hummingbird kit with Snap! visual programming language," *Journal of Learning and Teaching in Digital Age*, vol. 1, pp. 20-26, 2016.
- [42] Papert, S., "Microworlds: transforming education," in *Artificial intelligence and education*, 1987, pp. 79-94.
- [43] Papert, S., *Mindstorms: Children, Computers and Powerful Ideas*. New York: Basic Books, 1980.
- [44] Papert, S. and I. Harel, Eds., *Constructionism*. Norwood, NJ: Ablex Publishing Corporation, 1993, 518 Pages
- [45] Pelánek, R. and T. Effenberger, "Design and analysis of microworlds and puzzles for block-based programming," *Computer Science Education*, vol. 32, pp. 66-104, 2022.
- [46] Pokress, S. C. and J. J. D. Veiga, "MIT App Inventor: Enabling personal mobile computing," *arXiv preprint arXiv:1310.2830*, 2013.
- [47] Repenning, A., "Bending the Rules: Steps toward Semantically Enriched Graphical Rewrite Rules," in *Proceedings of Visual Languages*, Darmstadt, Germany, 1995, pp. 226-233.
- [48] Repenning, A., "Conversational Programming: Exploring Interactive Program Analysis," presented at the 2013 ACM International Symposium on New ideas, New Paradigms, and Reflections on Programming & Software (SPLASH/Onward! 13), Indianapolis, Indiana, USA, 2013, 63-74.
- [49] Repenning, A., "Escaping the Turing Tar-Pit with AI Programming Blocks," presented at the The 19th WiPSCE Conference on Primary and Secondary Computing Education Research, Munich, Germany, 2024.
- [50] Repenning, A., "Excuse me, I need better AI!: employing collaborative diffusion to make game AI child's play," presented at the ACM SIGGRAPH symposium on Videogames, Boston, Massachusetts, 2006, 169-178.
- [51] Repenning, A. and A. Basawapatna, "Smacking Screws with Hammers: Experiencing Affordances of Block-based Programming through the Hourglass Challenge," presented at the Special Interest Group on Computer Science Education Technical Symposium (SIGCSE TS 2021), Toronto, Canada, 2021, 7.
- [52] Repenning, A., A. Basawapatna, and N. Escherle, "Computational Thinking Tools," presented at the IEEE Symposium on Visual Languages and Human-Centric Computing, Cambridge, UK, 2016.
- [53] Repenning, A. and S. Grabowski, "Scaffolding Creative Programming Projects," presented at the The 19th WiPSCE Conference on Primary and Secondary Computing Education Research 2024, Munich, Germany, 2024.
- [54] Repenning, A. and A. Ioannidou, "LEGOsheets: Rule-Based Programming for the MIT Programmable Brick," *CM Transactions on Computer-Human Interaction*, 1998.
- [55] Repenning, A., D. C. Webb, K. H. Koh, H. Nickerson, S. B. Miller, C. Brand, I. H. M. Horses, A. Basawapatna, F. Gluck, R. Grover, K. Gutierrez, and N. Repenning, "Scalable Game Design: A Strategy to Bring Systemic Computer Science Education to Schools through Game Design and Simulation Creation," *Transactions on Computing Education (TOCE)*, vol. 15, pp. 1-31, 2015.
- [56] Repenning A., Basawapatna A.R., and E. N.A., "Principles of Computational Thinking Tools," in *Emerging Research, Practice, and Policy on Computational Thinking*. Educational Communications and Technology: Issues and Innovations, H. C. Rich P., Ed., ed: Springer, Cham, 2017, pp. pp 291-305.
- [57] Resnick, M. and E. Rosenbaum, "Designing for tinkability," *Design, make, play: Growing the next generation of STEM innovators*, pp. 163-181, 2013.
- [58] Robertson, T., S. Prabhakararao, M. Burnett, C. Cook, J. R. Ruthruff, L. Beckwith, and A. Phalgune, "Impact of interruption style on end-user debugging," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2004, pp. 287-294.
- [59] Rosenberg, J. B., *How debuggers work: algorithms, data structures, and architecture*: John Wiley & Sons, Inc., 1996.
- [60] Salinas, A. G., "The Relation between Syntax, Semantics and Pragmatics," *Revista de Humanidades: Tecnológico de Monterrey*, pp. 13-19, 2001.
- [61] Saqr, M., K. Ng, S. S. Oyelere, and M. Tedre, "People, Ideas, Milestones: A Scientometric Study of Computational Thinking," *ACM Transactions on Computing Education (TOCE)*, vol. 21, pp. 1-17, 2021.
- [62] Seckel, M. J., C. Salinas, V. Font, and G. Sala-Sebastia, "Guidelines to develop computational thinking using the Bee-bot robot from the literature," *Education and Information Technologies*, vol. 28, pp. 16127-16151, 2023.
- [63] Shneiderman, B., "Direct Manipulation: A Step Beyond Programming Languages," in *Human-Computer Interaction: A multidisciplinary approach*, R. M. Baecker and W. A. S. Buxton, Eds., ed Toronto: Morgan Kaufmann Publishers, INC. 95 First Street, Los Altos, CA 94022, 1989, pp. 461-467.
- [64] Slany, W., "Tinkering with Pocket Code, a Scratch-like programming app for your smartphone," *Proceedings of Constructionism*, 2014.
- [65] Smith, D. C., A. Cypher, and J. Spohrer, "KidSim: Programming Agents Without a Programming Language," *Communications of the ACM*, vol. 37, pp. 54-68, 1994.
- [66] Spohrer, J. G. and E. Soloway, "Analyzing the high frequency bugs in novice programs," in *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, 1986, pp. 230-251.
- [67] Tajeddin, Z. and J. Kamali, "Typology of scaffolding in teacher discourse: Large data-based evidence from second language classrooms," *International Journal of Applied Linguistics*, vol. 30, pp. 329-343, 2020.
- [68] Tanimoto, S. L., "Towards a theory of progressive operators for live visual programming environments," in *Proceedings of the 1990 IEEE Workshop on Visual Languages*, 1990, pp. 80-85.
- [69] Telles, M. and Y. Hsieh, *The Science of Debugging*. Scottsdale: Coriolis Group Books, Scottsdale AZ, USA, 2001.
- [70] Voštinár, P., "Creating mobile apps for teaching," in *INTED2018 Proceedings*, 2018, pp. 811-816.
- [71] Watt, S., "Syntonicity and the Psychology of Programming," in *Proceedings of the Tenth Annual Meeting of the Psychology of Programming Interest Group*, Milton Keenes, UK, 1998, pp. 75-86.
- [72] Yang, D., Z. Yang, and M. U. Bers, "The efficacy of a computer science curriculum for early childhood: evidence from a randomized controlled trial in K-2 classrooms," *Computer Science Education*, pp. 1-21, 2023.
- [73] Yaune, J., S. R. Bartholomew, and P. Rich, "A systematic review of "Hour of Code" research," *Computer Science Education*, pp. 1-33, 2022.
- [74] Zha, S., Y. Jin, P. Moore, and J. Gaston, "Hopscotch into Coding: Introducing Pre-Service Teachers Computational Thinking," *TechTrends*, vol. 64, pp. 17-28, 2020.