

# Designing Debuggers for Block Programming Languages

**Mokter Hossain**

Department of Computer Science  
The University of Alabama  
Tuscaloosa, AL, USA  
mokter@gmail.com

*Abstract* – Block Programming Languages (BPLs) have been gaining popularity among novice programmers of all ages, backgrounds, and interests [30]. Although BPLs provide constructive learning approaches that support learning by self-practicing, programmers who use BPLs often face difficulty when an anomaly or program bug emerges in their programs [5]. In this paper, we review the literature on debugging techniques used by novice-programmers. In particular, the paper focuses on understanding the barriers faced by those learning to program for the first time and the ways in which they debug their programs. We survey the existing debugger features provided in several visual BPLs that have been designed mainly for initial learners. After providing background information and the motivating context of this research, we analyze the six assigned core papers and discuss their contribution to the theme of debugging support for initial learners. A survey of existing related works from the literature is discussed in the context of several future research directions for feasible improvement in these emerging languages.

*Keywords:* Block Programming Language, debugger, collaborative debugging, novice-programmers

## 1. INTRODUCTION

Computer science educators, researchers, and programmers have been trying to motivate students of all ages, backgrounds, and interests toward learning how to program [30]. In 1967, Seymour Papert designed the Logo programming language that allowed graphics to be drawn by asking a “turtle” to move around on a drawing area [28]. With Logo, novice programmers were able to immediately see the output of their programs on the computer screen without having to complete a full, syntactically correct program [12]. Since then, computer educators have been spending their efforts in developing constructive programming languages that can reduce the syntactic burden on students and engage them in fruitful activities [12]. A number of leading U.S. universities and colleges have undertaken initiatives in developing first-time programmer friendly programming languages. Blocks Programming Languages are the product of such endeavors in the development of innovative programming environments for the prospective programmers [30]. A BPL allows a programmer to drag and drop a number of predefined blocks to develop very simple to moderately complex computer programs very easily. Scratch and App Inventor developed at the MIT Media Lab; and BYOB (now named as Snap!) developed at the University of California, Berkeley, are popular BPLs. These BPLs allow the programmers to program with a mouse where the programmatic constructs are represented as puzzle-like pieces that fit together only if they seem to be syntactically appropriate [24]. Due to the abstract nature and many attractive features, BPLs have become popular for novice programmers, as well as prospective computer science majors. Many universities and colleges have integrated one or more BPLs for their prospective computer science majors and minors.

User interfaces of the BPLs are designed based on the metaphor of visual programming bricks that are called program blocks. The blocks can be dragged and integrated with each other to form a complete program. A block contains one or more indentions to connect with another possible block. The existing BPL environments are integrated with implicit debugger, code converter, program sharing, remixing and many other interactive features. However, they have some limitations. For instance, the debugging systems in BPLs are non-existent or hidden from their users. Thus, while programming in a BPL, if someone faces a problem or does some kind of mistake in their program, they have little chance to get help from the language environment. Due to the lack of explicit code search option in the BPL, there is little chance to debug or trace a program. Programing in a BPL environment is easy, interesting, and self-motivating [30]. Naturally, it lends itself to interactive projects such as animation or interactive games that many students without any prior programming knowledge have been shown to gain self-confidence in their programming abilities [26]. However, completing interactive projects usually requires maintenance and debugging large numbers of interrelated blocks that may not be possible for them to manage. Paradoxically, the motivation that results from the ability to program interesting games become dissolved due to the difficult and frustrating debugging process [26].

A software bug is an anomaly or error, mistake, fault, or failure in a computer program that prevents the program from behaving as intended or causes the program to produce an unintended or incorrect result [11, 27, 35, 37]. *Debugging* is a methodical process of locating and fixing, or bypassing bugs or defects in computer programs [11, 22, 37]. To debug a program, first the problem is traced and isolated from the program, then the problem is fixed. Debugging tools, called *debuggers*, help a programmer identify coding errors (bugs) at various software development stages, starting from coding to user testing. A debugger is a natural component of programming and is often integrated into traditional development environments, such as Eclipse or VisualStudio. Debugging is also integrated with most software development process, whether used in commercial software or a personal application program [25]. Some programming languages include a capability to check for errors within the code editor. However, new programming languages that are focused on serving initial learners (e.g., Scratch, Snap) do not have integrated or comprehensive debuggers [26]. Furthermore, novice-programmers sometimes expect the programming environment to be able to interpret a construct in the way they intended and may be confused when a bug emerges in their program [5]. Thus, the process of debugging can be difficult and frustrating for novice programmers. The process becomes more challenging for distributed and collaborative programming environments because traditional debuggers do not offer specific features for collaboration or remotely shared usage [11].

In this research, we seek the opportunity of improving the debugging systems for BPLs. Thus, we review the related literature to collect initial program learners' programming and debugging strategies; to figure out the barriers they face; to learn the misconceptions they hold; and the debugging features they expect to be included in the future BPLs. In this paper, we frequently use the terms novice-programmers, non-programmers, and end-user-programmers that we collectively term as novice-programmers, or initial program learners. *Novice-programmers* are those who have some kind of working knowledge about programming, but do not have deep prior programming experience. *Non-programmers* are those who do not have active knowledge of programming. *End-user-programmers* are those who program in order to express some computational need within their domain of expertise. For instance, end-users may create a macro in Word or Excel to assist with their daily work activities. Novice programmers program in order to learn programming in a way that will deepen their expertise of programming, in general. The debugging strategies that novice programmers apply can help to understand how initial learners actually solve problems through expression of a program [16]. Studies have found that those who demonstrate good debugging skills are good programmers, but not all good programmers necessarily have strong debugging skills [1]. Novice-programmers are generally more skillful than non-programmers in terms of debugging [25]. Like novice programmers, end-user programmers usually do not have professional programming experience. Thus, research into novice and end-user programmers' debugging experience should be considered as pertinent to each other [15].

In the rest of this paper, an analysis of the core papers [2], [15], [16], [19], [23], and [25] is presented in Section 2 followed by their discussion in Section 3 with an overview of the state-of-the-art in Section 4. Finally, Section 5 concludes with future directions.

## 2. ANALYSIS OF CORE PAPERS

In this section we analyze the six assigned core papers. For each core paper, we describe its methods, empirical results, if any, and development, and scope of implication for our future work.

### A. *Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers*

Kelleher and Pausch [19] presented a taxonomy of languages and environments that they developed with a goal of making programming more accessible to the novice programmers of all ages. They also presented two informative tables including the language developers' summarized endeavors in order to make learning to program easier for novice programmers.

1) *Methods*: In organizing their taxonomy, the authors [19] started with a pivotal research question: *Why novices need to program?* Considering a variety of possible answers for this research question, such as: to pursue programming as a career path; to learn how to solve problems in a logical and structured way; to build customized software; to explore ideas in other subject areas; etc. the authors developed and enriched their taxonomy from the existing programming languages developed mainly for novice programmers.

2) *Development*: The authors developed their taxonomy (Table-1) from the existing programming languages and environments with a goal of lowering the barriers to novice programmers of all ages. In the proposed taxonomy they categorize the existing languages into two large groups:

- a. *Teaching Systems*: Languages that attempt to teach programming; and
- b. *Empowering Systems*: Languages that attempt to support the use of programming.

Languages in the Teaching Systems group are mostly simple programming tools that provide novice programmers with exposure to some of the fundamental aspects of the programming process. They categorically address two aspects: (1.1) Mechanics of Programming; and (1.2) Learning Support. The languages in the mechanics of programming address at least one of the following three features: (1.1.1) Expression of instructions (e.g., syntax); (1.1.2) Programs structure (e.g., programming style); and (1.1.3) Understanding Program execution. Whereas the languages in the Learning Support try to ease the process of learning to program by providing two concepts: (1.2.1) Social Learning; and (1.2.2) Motivation Context. The social learning concept motivates program learners to learn programming from each other and/or collaboratively.

Languages in the Empowering Systems group support users to build programs and applications for their own need. Thus, development of these languages focuses more on building applications rather than focusing on how well learners can translate their existing programming knowledge to the new languages. Systems of this category mainly examine ways of improving programming languages and alternative ways for developing end-user programs. Thus, the languages in this category address only one aspect: (2.1) Mechanics of Programming. However, systems in this category address one of the following three features for the program developers: (2.1.1) Code is too difficult – systems in this group examine creating programs either through demonstrating actions in the interface, or demonstrating conditions and actions or by specifying action. (2.1.2) Improve Languages –systems in this group are made to make the languages more understandable, improve interaction within the language, or to make better integration with the environment; and (2.1.3) Activities Enhanced by Programming – the systems in this group enhance activities by exploring programming in a particular domain such as in entertainment or education.

Table-1 : A Taxonomy of Programming Environments and Languages for Novice Programmers [19]

1. Teaching Systems											2. Empowering Systems									
1.1 Mechanics of Programming								1.2 Learning Support			2.1 Mechanics of Programming									
1.1.1 Expression of instructions				1.1.2 Structure		1.1.3 Execution		1.2.1 Social Learning		1.2.2 Motivation		2.1.1 Code is too Difficult		2.1.2 Improve Languages		2.1.3 Activities Enhanced by Programming				
Simplify Coding		Alternative to Typing																		
Simplify the Language	Prevent Syntax Errors	Construct Program Using Graphical/Physical Objects	Create Programs Using Interface Actions	Provide multiple methods for Creating	Designing New Programming Models	Making New Models Accessible	Tracking Program Execution	Make Programming Concrete	Models of Program Execution	Learn Program Side by Side	Networked Interaction	Providing a Motivating Context	Demonstrate Actions in the Interface	Demonstrate Conditions and Actions	Specify Actions	Make the language more understandable	Improve Interaction with Language	Integration with Environment	Entertainment	Education
BASIC	GNOME	Alice-2	LegoSheets	Leogo	Pascal, Smalltalk	Liveworld	Atari 2600 Basic	Karel, Josef	ToonTalk	AlgoBlock	MOOSE Crossing	Rocky's Boots, AlgoArene	Pygmalion	AgentSheet	Emile	COBOL, Logo, Alice 98	Alice 99, Flogo	Boxer, Visual Agent Talk	Bongo, Mindrover	SOLO, Starlogo

In this taxonomy (Table-1), the authors place the systems only once in their taxonomy, based on the particular system's primary intention to address. However, they also noticed that many of the systems have incorporated ideas drawn from the earlier systems. Thus, the authors presented two other informative tables: a table of system influences - presents the list of systems that influence the design of later systems for novice programmers; and a table of system attributes- shows the major and minor design influences that contributed in the development of corresponding system. The table of system influences shows that Logo, designed by Seymour Papert in 1967 [28], was the most influential programming language for the development of more than 37% of later languages. Table-2 briefly depicts the attributes of the systems included in this taxonomy.

3) *Implication*: An analysis of the informative tables reveals that support to understand programs through “debugging”; preventing syntax errors by providing “better syntax error messages”; and support communication through the “networked-shared manipulation” had not been implemented in about 90% of the languages discussed in this taxonomy, including the Logo. Thus, we consider the development of network-shared collaborative debuggers for Logo-like BPLs as an appropriate and timely research topic in this important and emerging area.

Table-2: System Attributes – shows the major design influences, including those that are not the primary contribution of the system [19]

		Simplify the Language	Prevent Syntax Errors	Construct Program Using Objects	Create Programs Using Interface Actions	Provide multiple methods for Creating	Designing New Programming Models	Making New Models Accessible	Tracking Program Execution	Make Programming Concrete	Models of Program Execution	Learn Program Side by Side	Networked Interaction	Providing a Motivating Context	Demonstrate Actions in the Interface	Demonstrate Conditions and Actions	Specify Actions	Make the language more understandable	Improve Interaction with Language	Integration with Environment	Entertainment	Education	Total number of systems that support the attribute
Style of Programming	procedural	5	1	7	3	1	1		1	3	2	2	1	2	3		1	2	3	4	3	3	48
	functional																		1				1
	object-based			1			1											1					5
	object-oriented	2		1			1	5					2	1				1			1	1	15
	event-based			5	1		2	1					2		1	3	3	2	9	3	4	1	37
	state-machine based						1																1
Programming Constructs	conditional	6	1	6	2	1	4	4	1	3	2		3	3	1	3	2	4	5	5	4	4	64
	count loop	1	1	5	2			2	2	2		1	1	1			1	1	2	1	1	1	21
	for loops	4				1	2	2	1	1			2	1	1		1	4		4	1	1	26
	while loops	5	1	1			2	4	1	3	1		1	2	1		1	3	1	3	1	2	33
	variables	6	1	4			3	2	1	1	1	1	2	2	1		2	3	2	4	1	4	44
	parameters	6	1	3		1	2	2	1	1	2	1	2	2	1		2	3	2	4	1	4	41
	procedures/ methods	6	1	5	1	1	4	4	1	3	2		2	2	3	3	1	4	4	5	1	4	57
	user-defined data types	5					2	2	1									3	1				15
	pre and post conditions	2																					2
Representation of Code	text	6	2	4	2	1	3	5		3	1		3	2	1		2	4	4	4	1	3	51
	pictures			8	3		1	1						1	2	3	1			1	4		25
	flowchart			2										1	1				3		2	1	10
	animations				1						1												2
	forms																						6
	finite state machine						1																1
Construction of Programs	physical objects			2								2								3			7
	typing code	6	2		1	1	3	5		3			3	2			1	4	1	4	1	4	41
	assembling graphical objects			7			1							1	2	3	1		7	1	4	1	28
	demonstrating actions				2	1					2		1		2	3				1			12
	selecting/ form filling		1	1	1	1					1		1	1			2	1	1	1		1	13
	assembling physical objects			2								2							3				7
Support to Understand Programs	back stories							3		2	2							1					8
	debugging															1							1
	physical interpretation			3	2		1			2	1	1		1			1	3	2		3	1	21
	liveness			1	2			1							1		1	1	4	2		1	14
	generated examples					1								1						1		1	4
	physical shape affordance			6											1				1		1		9
Preventing Syntax Errors	selection from valid options		1	2		1								1			1		3				9
	syntax directed editing		3															1				1	5
	dropping only in a valid location			3													1		2				6
	better syntax error messages	1																					1
	limit the domain		1	10	4	1	2	3		2		1	1	1			2	1	5		2	2	38
	select user-centered keywords			1		1	1	3		1			1					2	2				12
Designing Accessible Languages	remove unnecessary punctuations	2		1														1					4
	use natural language						1								1			2	1				5
	remove redundancy	4																1					5
	side by side			2	3							1											8
Support communication	networked-shared manipulation												1										1
	networked-shared results			1									2							1			4
	fun and motivating			10	4		1				1	1	2	3			2	3	4		5		37
Choice of Task	useful	2					2	2				1			3			3	7	4	2	3	29
	educational	6	3	11	4	1	2	5	1	3	2	2	3	3		3	3	2	4	1	3	4	68

*B. Non-programmers identifying functionality in unfamiliar code: Strategies and barriers.*

In this exploratory study, the authors, Gross and Kelleher [16] discussed the strategies that non-programmers employ in identifying functionality in unfamiliar code and the barriers that non-programmers face in using or customizing the codes for their own purpose.

1) *Methods*: Fourteen adult non-programmers voluntarily participated in the study exploratory. The authors randomly selected 15 Storytelling Alice programs from the Alice.org forum and summarized the programs by the following six properties:

- i) Size of the program- with a median of 437 by their line of codes (LOCs);
- ii) Amount of the program's executing code in its main method;
- iii) Existence of interactive events in the program;
- iv) Existence of ambiguous variable, object, method name;
- v) Existence of concurrent threads in the program; and
- vi) Use of dialog bubbles in the program's executing code.

Considering these six properties as the dimensions of guidance, the authors constructed four other Alice programs to use in their study. For each of these programs the authors composed a series of five tasks of varying complexity. Each task caused the participants to confront one or more of the above six dimensions and/or language constructs either by design or because of its associated program's design. The participants were asked to accomplish one or more of the two specific tasks: (i) bounding task – to mark the beginning and ending of the code responsible for specific functionality; and (ii) modification task- to make a very specific change to the code that affects the output functionality of the video or running program.

2) *Results*: The study revealed that finding target code in unfamiliar program codes is difficult for non-programmers. When searching for an action in unfamiliar programs, non-programmers search for an action by looking for key phrases such as: pointing, right, left, etc. Without finding such a phrase, they cannot resolve the search. While searching for a specific functionality in unfamiliar programs, they often verbally search for specific features in the source program or in the output. The authors termed such features as code landmark and output landmark, respectively.

3) *Development*: Based on the overall findings of the study the authors developed two interactive models of non-programmers approaches in finding target code in unfamiliar programs. These are: (1) Task Process Model; and (2) Landmark-Mapping Model. The task process model represents the typical task workflow when a subject attempts a task; and the landmark-mapping model suggests a model for how non-programmers organize and relate the landmarks they collect. Analyzing those models, the authors pointed out a number of common barriers and search strategies that non-programmers employ in finding target codes in unfamiliar programs. These are depicted in Table-3 and Table-4, respectively.

Table-3: Barriers non-programmers encounter when attempting to resolve a given task [16]

Barrier Name	# of Users Hitting Barrier	# of Tasks Observed in	Description
Object and action encoding	12/14	12/20	User description of observed output not in code text
Memory failure	7/14	8/20	User incorrectly remembers target output
Method interpretation	13/14	12/20	User misinterprets method invocations' clues
Lack of temporal reasoning	10/14	10/20	User does not use sequential execution of information
Temporal reasoning over use and ignoring constructs	13/14	12/20	User interprets sequential execution of code when inappropriate because of language construct
Magic code	7/14	15/20	User associates incorrect functionality with codes running concurrently

4) *Implication*: While this study presented non-programmers' experience on using the earlier Storytelling Alice programs, the model, strategies, barriers, and recommendations presented in this study could be applicable to the most common BPLs, such as in Scratch and Snap. Especially, the following design guidelines that the authors recommend for improving future programming environments to support non-programmers seem to be considerable for our research. These include:

- Connect code to observe output – by showing how the output changes when a line of code executes.
- Help users to reconstruct program execution flow – by highlighting lines of code executing at a certain point of time;
- Provide interaction to fully navigate code – by providing an interface for allowing the user to trace through the lines of code that executed; and
- Help using poorly constructed code – by building support into programming environments that help users to successfully navigate imperfect code.

Table-4: Search strategies non-programmers employ when attempting tasks [16]

Strategy name	# of users observed it	# of tasks observed in	% of all searches	Target generate	Description
Text and semantic	14	20	20%	No	User searches code for text similar to their search target description
Temporal	14	19	14%	No	User leverages timing to narrow search space
Comprehensive	14	17	7%	No	User locally focuses search around “known” code
Exhaustive	11	10	2%	No	User attempts to search entire code space
API	7	10	2%	Yes	User views objects’ interfaces to find search targets
Explorative	8	8	3%	Yes	User randomly explores interface for new information to use as search target
Context	9	8	1%	Yes	User identifies output immediately around target actions for new search targets
Uncategorized	-	-	51%	-	Users do not verbalize a strategy for all observed searches

### C. Debugging: A review of the literature from an educational perspective

McCauley et al. [25] reviewed a large volume of literature on learning and teaching of debugging computer programs, from the mid-1970s, and presented an overview of contemporary literature on these areas. They also investigated knowledge and strategies of both successful and unsuccessful debuggers, and presented some differences between novice and expert programmers in terms of their programming and debugging techniques.

1) *Methods*: The authors group the research papers according to their relevance in providing insight into one of the following four focal research questions:

- (1) Why do bugs occur?
- (2) What types of bugs occur?
- (3) What is the debugging process?
- (4) How can we improve the learning and teaching of debugging?

In order to investigate these research questions the authors conducted a survey on a significant body of the debugging literature that focuses on the identification and categorizations of some specific program bugs.

2) *Results*: The findings of the study did not yield a single answer to the first question: why do bugs occur? Rather, they seemed to debunk the most popular misconceptions that the language causes most program bugs. Most programming errors result from a chain of cognitive breakdowns that occur in skill, rules, or knowledge breakdown or caused due to other underlying problems. This study reported another contemporary study by Spohrer and Soloway [33] that revealed that there are at least seven causes of most program bugs. These are:

- i) Boundary problems – include off-by-one bugs (i.e., terminating a loop iteration too early or too late).
- ii) Plan dependency problems – describe misplaced code often relates to a nesting or condition construct.
- iii) Negation and whole part problems – involve misuse of logical constructs (e.g., using < when <= is needed).
- iv) Expectations and interpretation problems – misinterpretation of how certain quantities are calculated.
- v) Duplicate tail digit problems – involve dropping the final digit from a constant with duplicated tail digits.
- vi) Related knowledge interference problems – occur due to the similarity between some correct and incorrect knowledge.
- vii) Coincidental ordering problems – occur due to missing parentheses to override operator precedence.

The second question also reveals some notable findings such that “a few types of bugs account for a majority of the mistakes in students’ programs.” Another study [32] notes that the study identifies that the above seven causes of bugs fall into five different types of bugs. These are shown in Table-5.

The third question investigated the knowledge and strategy for both successful and unsuccessful, novice and expert debuggers to point out their debugging processes. Thus, the investigation is refined using the following three specific sub-questions:

- (1) What types of knowledge are aids in debugging?
- (2) What strategies are employed in debugging?
- (3) How do novices and experts differ?

Table-5. Types of non-construct-based bugs with their descriptions and causes [25]

Type of bug	Description	Cause
Zero is excluded	Zero is not considered as a valid amount	Boundary problem
Output fragment	Output statements are improperly placed in the code (e.g., use of if/else outside the block)	Plan-dependency problem
Off-by-one	Use of incorrect relational operator (e.g., use of <= rather than <)	Boundary problems
Wrong constant	Typo in typing constant (e.g., use of 4320 when wanted to use 43200)	Duplicate tail-digit problem
Wrong formula	Use of a wrong formula that is not appropriate to compute a the correct value	Expectations and interpretation problem

A number of notable findings are revealed for the first sub-question. Among these, an earlier study [10] found that good debuggers possess the following seven types of knowledge during the bug location phase of debugging.

- i) Knowledge of the intended program (program I/O, behavior, implementation);
- ii) Knowledge of the actual program (program I/O, behavior, implementation);
- iii) An understanding of the implementation of language;
- iv) General programming expertise;
- v) Knowledge of the application domain;
- vi) Knowledge of bugs; and
- vii) Knowledge of debugging methods.

Notable findings for the second sub-question are that expert programmers use a breadth-first approach when debugging. On the other hand, novice programmers take a depth-first approach, focusing on finding and fixing the error without regard to the overall program [36]. Novices take longer time to understand the new system [6]. However, notable results for the third sub-question seem to be unsurprising in many ways- such as: experts are typically faster and more effective debuggers than novices. Expert programmers possess a deeper, more sophisticated mental model than novices. Furthermore, novices are more likely to work on isolated pieces of code that sometime create new bugs as they attempt to fix existing bugs spontaneously [13].

Finally, findings of the fourth research question are more interesting. An earlier study [21] found that “debugging is a skill that does not immediately follow from the ability to write code.” Explicit debugging instruction is effective for novice programmers. They should be taught explicit debugging techniques to help them learn effective programming [21]. This has been echoed in findings of another study [8] that found that formal training in debugging helped novice programmers develop skills in diagnosing and removing defects from programs.

3) *Implications*: Many of the findings of this study, such as the causes of errors that novice programmers face and the strategies they use to locate and fix the program bugs seem to be useful and considerable for our future study in the development of debuggers for novice programmers. The types of non-construct-based bugs with their descriptions and causes are also useful.

#### D. End-user debugging strategies: A sensemaking perspective

In this paper the authors Grigoreanu et al. [15] report the results of an empirical study of end-user programmers’ debugging strategies about a spreadsheet’s correctness on a sensemaking perspective. The term *sensemaking* is used to describe how people make sense of the information around them related to a task and how they represent and encode that information to answer the task-specific questions [14].

1) *Methods*: In order to understand how end-user programmers solve debugging problems, the authors considered an identifiable subpopulation of spreadsheet users; and investigated the following research question:

*How do end-user programmers (both male and female) make sense of spreadsheets' correctness when debugging?* Ten undergraduate and graduate students (five male and five female) at a U.S. public university who were not majoring in computer science, but had experience in using formulas in Excel were selected to participate in this study. However, two most successful (one male, one female) and other two most unsuccessful (one male, one female) – a total of four participants' detail activity logs and think-aloud data were considered for the in-depth analysis in this study.

2) *Results*: The study reported a number of notable findings. These are in terms of the prevalence of information foraging during end-user debugging; two successful ways of traversing the sensemaking model and their potential ties to the literature on gender differences; sensemaking sequences leading to debugging progress; and sensemaking sequences with troublesome points in the debugging process.

3) *Development*: Using the empirical study results, the authors derived a sensemaking model for end-user debugging. Based on an existing sensemaking model for intelligence analysts, developed by Pirolli and Card [29], these authors derived a new model for end-user debuggers that contained the following three major interweaved sensemaking loops:

- i) Bug fixing loop- in which participants reason about the bugs and spreadsheet formulas/values;
- ii) Environment loop- in which the participants reason about the spreadsheet environment; and
- iii) Common sense loop – in which they reason about common-sense topics and/or the domain.

The authors used the model and its three major loops to shed light on the end-user debugging approaches and the problems that arose in the sensemaking central to debugging.

4) *Implication*: The general guidelines of this study have had some implications for the improvement of debugging tools to support end-user programmers. The empirical component of this study may help us to understand whether the new tool works or not and how that supports the sensemaking efforts of end-user programmers.

#### *E. Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior*

In this paper, the authors Ko and Myers [23] proposed a new debugging paradigm called *Interrogative Debugging (ID)*. In an ID paradigm a programmer can ask *why did* and *why didn't* type questions directly about a program's behavior and can view answers in terms of directly relevant runtime data. The authors then described the design strategies of such an interrogative debugger, *Whyline* - a Workspace that **H**elps **Y**ou **L**ink **I**nstructions to **N**umbers and **E**vents.

1) *Methods*: The *Whyline* is designed as a prototype ID interface for Alice. It visualizes answers of the following three questions:

- i) Would the *Whyline* be considered useful?
- ii) Would the *Whyline* reduce debugging time?
- iii) Would the *Whyline* help complete more tasks?

The authors examined the usability and effectiveness of the *Whyline* interface by comparing results of two different studies, categorically in terms of the above three questions. The two studies are conducted with two different groups of graduate students selected from a U.S. university. The first study, referred as the *Without study*, was conducted with four participants who did not use the *Whyline*; and the second study, referred as the *With study*, was conducted with five other participants who used the *Whyline*.

2) *Results*: The study revealed that in the *Without study*, participants tend to hypothesize and diagnose by inspecting and rewriting code. On the other hand, in the *With study*, they tend to hypothesize and diagnose by asking questions and analyzing the *Whyline*'s answer. By comparing six identical debugging scenarios for Alice programs from user tests in the *Without* and *With* studies, the authors found that the *Whyline* reduced debugging time by nearly a factor of 8- that enabled the participants to complete 40% more tasks. Thus, the *Whyline* significantly decreases debugging time for the Alice programmers.

3) *Implication*: It appears that the *Whyline* type ID has a great potential as a usable and effective debugging tool for Alice like programming environments such as for the BPLs such as Blockly, Scratch, and Snap. Especially, the *why didn't* questions may be crucial to programmers' perceptions of the utility of such debugging tools, if successfully implemented.



*F. A use-case for behavioral programming: An architecture in JavaScript and Blockly for interactive applications with cross-cutting scenarios*

Ashrov et al. [2] presented an approach for implementing Behavioral Programming (BP) in JavaScript and Blockly. The BP approach is an extension and generalization of scenario-based programming that enables software development in a natural and incremental way. A behavioral application consists of a number of threads or behaviors each of which represents an independent scenario that the system should or should not follow. These independent behavioral threads are interweaved at an application's run-time resources yield in integrated system behavior.

1) *Methods*: The implementation of BP in JavaScript and in Google Blockly includes the following three new types of blocks: *b-thread*, *b-Sync*, and *lastEvent*, as shown in Figure-1.



Figure-1. Three types of new blocks designed with JavaScript to implement BP in Blockly

In the *b-thread* block the *b-thread* logic is given. The *b-thread* body can use any Blockly block for implementing the desired processing. The parameter passed to the *b-thread* block is a list of values that enables the *b-thread* codes. The program structure generates multiple instances of the given *b-thread* logic, and passes one of the values from the list to each of the *b-thread* logic instances, to be used during instance execution. The *b-Sync* block is used inside a *b-thread* block for synchronization with other *b-threads* and for specifying requested, waited-for, and blocked events. The *lastEvent* block denotes a variable of the event that triggers the last most recent synchronization point; and can be accessed by *b-threads* that were resumed after waiting for a number of events to determine which of these events was actually triggered.

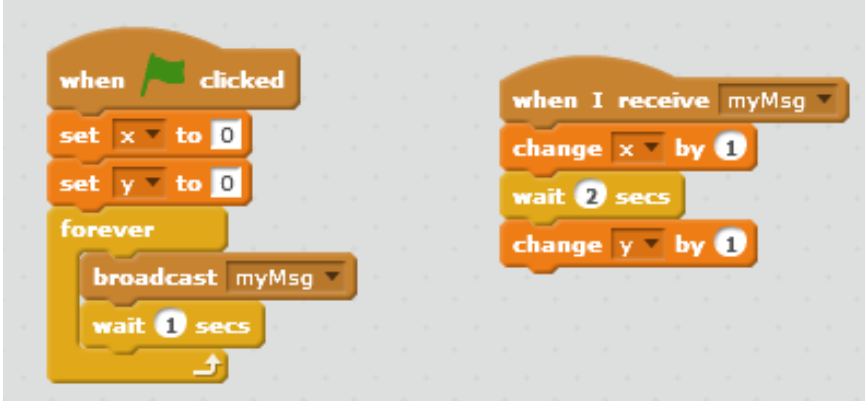
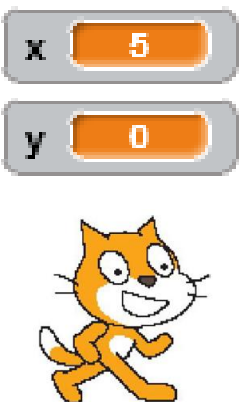
2) *Development*: The authors combined visual programming using Google Blockly with a single-threaded implementation of BP in JavaScript. They showed how a behavioral application could interface with the real-world applications in a way that was separated from the application logic. They also described an infrastructure for running behavioral programs on Android smartphones and demonstrated basic scenarios as well as sensors and actuators interfaces to smartphone functions.

3) *Implication*: At least two aspects of this development seem to be useful for our study. First, the state management process of large scenarios with BP could be used for managing the large program blocks in the BPLs. Second, the parallel programming process of BP could be used to manage continuous entities for interweaving single or multi-threaded scripts in the BPLs.

As the *b-threads* do not examine the data structure at run time to understand the evolving state of the program, the use of *b-thread* with BP could make managing large section of blocks simpler and easier to debug than programs written traditionally in a BPL. This can be illustrated with a less sophisticated example with the following Scratch scripts, shown in Table-6.

When this program is run, the first script broadcasts the message, *myMsg*, waits for 1 second, and then broadcasts the message again. Whenever this message is received by the second script, the variable *x* is incremented by 1, and after 2 seconds, the variable *y* is supposed to be incremented by 1. However, outputs of running and stopping these scripts show that when *myMsg* is broadcasted for the second time, the first execution of the second script is interrupted and is never resumed. Thus, *y* is never incremented. This issue has been pointed out in the Scratch user forum [4]; however, there is no suggestion how to solve it. As we see, If the *myMsg* is set to be broadcasted in  $\geq 2$  seconds time-limit, the program run expectedly when both of the variables *x* and *y* are increased properly.

Table-6. Some interweaving program scripts with their output

Scratch Scripts	Output
	

Ashrov et al. [2] suggest that using two b-threads, with one instance of each, the first performing a function similar to the above event broadcasting and processing, the event associated with the second message will not cause any effect, as at the synchronization point when it is triggered the processing b-thread will not be waiting for it, but instead will be waiting for a behavioral event signifying the completion of the wait time. If, on the other hand, the application starts another instance of the second b-thread class to catch such messages while other instances wait for the time-delay to pass, the event will be processed. In either case, the semantics will be well defined and the composite behavior will be readily predictable. Thus, using such BP events, variable sharing and interweaving problems can be resolved for single and multi-threaded programs written in BPLs.

### 3. DISCUSSION OF CORE PAPERS

The core papers were chosen based on their focus on the endeavor of making the programming languages and programming techniques attractive and available to the prospective programmers of all levels. However, these six papers can be categorized into three groups, according to their contribution in the perspective areas. Firstly, three of the core papers [19], [16], and [25] expressed their initiatives to *lowering the barriers to programming* in the newly developed and unfamiliar environments. In this regards, Kelleher and Pausch [19], developed a taxonomy of languages and environments with a goal of making programming more accessible to the novice programmers of all ages. Starting with all possible answers to their pivotal research question, *why novices need to program?*, these researchers developed the taxonomy from the existing about hundred programming languages and environments. Although these authors did not present the barriers that the novice programmers face while programming in the new languages, their goals were certainly intended to lowering the barriers to the novice programmers of all ages. In another paper [16] of this group, Gross and Kelleher [16] explicitly presented the barriers that non-programmers face in programming in the new languages. McCauley et al. [25] also investigated knowledge and strategies of both successful and unsuccessful debuggers.

Secondly, two of the core papers, [25] and [15], focused on the *debugging strategies* that the first-time programmers face. The first of these papers authored by McCauley et al. [25], also considered in previous group, reviewed a large volume of literature, from the mid-1970s, on learning and teaching of debugging computer programs, and presented the different debugging strategies that the novice and expert programmers imply in debugging their techniques. It categorically focused on the causes and types of program bugs; and process of debugging. Another paper of this group authored by Grigoreanu et al. [15] report the end-user programmers' debugging strategies about a spreadsheet's correctness on a sensemaking perspective. They categorically, investigated compared and contrasted the strategies that male and female programmers imply while debugging their programs.

Finally, the remaining two papers, [23] and [2] focused the design and development of effective debuggers for the novice programmers. The first paper of this group, authored by Ko and Myers [23] proposed a new type of interrogative debugging tool, called Whyline, that allowed the programmers to visually watch the effect of a line of code by asking *why did* or *why didn't* the desired outcome occurred in terms of the program's runtime data.

Comparing results of two empirical studies without and with using the Whyline debugger these authors showed that the Whyline type debugger can significantly decrease debugging time for the Alice programmers. Such findings of this paper imply that the Whyline type debugger has a great potential for the Alice like programming environments such as the Blockly, Scratch, and Snap. The last paper of this group, authored by Ashrov et al. [2] presented an approach for implementing Behavioral Programming (BP) in JavaScript and Blockly. These authors showed that the BP approach could interface with the real world applications in a way that was separated from the application logic. Their described infrastructure for running behavioral programs on Android smartphones implied that the state management process of large scenarios with BP could be used for managing the large program blocks in the BPLs. Moreover, the parallel programming process of BP could be used to manage continuous entities for interweaving single or multi-threaded scripts in the BPLs.

#### **4. OTHER RELATED WORK IN DEBUGGING SUPPORT FOR NOVICE PROGRAMMERS**

This section surveys the literature related to debugging features of programming languages that are especially designed for first-time programmers who are initial learners. Debugging is an important skill that continues to be both difficult for novice programmers to learn and challenging for computer science educators to teach. These challenges persist despite a wealth of important research on the subject dating back as far as the mid-1970s. Although the tools and languages that novices use for writing programs today are notably different from those available decades earlier, the basic problem-solving and pragmatic skills necessary to debug programs are still very similar. Hence, an understanding of the previous work on debugging can offer computer science educators insight into improving contemporary learning and teaching of debugging, and may suggest directions for future research into this important area.

Empirical research on debugging strategies by novice programmers began over a decade ago [15]. Initially, a series of sensemaking models began to appear in computer science literature, particularly in the area of Human Computer Interaction (HCI). Grigoreanu et al. [15] used these results to form a theory of how users make sense of a program based on its outputs, with a focus on participants' partial schemas developed through observation and hypothesis testing. Studies that investigated the needs of end-users during debugging revealed that much of what end-user programmers wanted to know during debugging was related to "why" and "why not" troubleshooting [23]. Past studies also discovered that males and females may use different strategies in program debugging and software development tasks related to problem solving. For example, females' self-efficacy on spreadsheet debugging is lower than male users in spreadsheet debugging [3, 15, 20, 31]. Some other empirical studies of end-user debugging have revealed that males outperform females. An explanation of this phenomenon might be that males have higher perceived self-efficacy (or confidence) than females in their ability to debug [3, 15, 20, 31]. This may be because opportunities for improved support for end-user debugging strategies for both genders are abundant, but the tools currently available to end-user debuggers may be especially deficient in supporting debugging strategies used by females [34].

Analyzing about a hundred programming languages that had been developed mainly for the first-time program learners, Kelleher and Pausch [19] developed a taxonomy with brief descriptions of all of the categories, and the systems within those categories. In order to make learning to program easier for novice programmers, these authors also presented two informative tables where they presented the list of systems that influence the design of later systems for novice programmers; and the major and minor system attributes that contributed the corresponding system's development, respectively. Gross and Kelleher [16] suggest to connect code to observe output by showing how the output changes when a line of code executes. They recommend that the best way to provide this feature is to offer support in the programming environment that enables users to correctly and quickly form mental mappings between the code and output. Both Gross and Kelleher [16] and the authors of Whyline [23], suggest the idea of allowing users to ask questions about the execution of a line, similar to the implementation of a Whyline-like debugging interface. Gross and Kelleher [16] developed Task Process and Landmark-Mapping models that describe how non-programmers approach finding code. These models can inform the design of new tools that will enable non-programmers to more quickly and accurately find the code that is responsible for functionality they would like to learn from or reuse.

To help end users catch errors from buggy programs early and reliably, Gragoreanu et al. [15] employed a novel approach for the design of debugging tools by focusing on supporting the end users' perceived debugging strategies. They first demonstrated the potential of a strategy-centric approach to tool design by presenting StratCel, an add-in for Excel. Second, they showed the benefits of this design approach: participants using StratCel found twice as many

bugs as participants using standard Excel, they fixed four times as many bugs, and within a small fraction of the time.

Carver and Risinger [6] conducted an experimental study to teach debugging strategies by providing explicit debugging instruction to a group of young students through a hierarchy of questions designed to facilitate bug location and correction. It has been claimed that their study of debugging instructions resulted in a significant improvement in novice programmers' debugging skills.

Katz and Anderson [18] studied novice debugging and observed two general search strategies: forward reasoning, where search stems from the actual, written code, and backward reasoning, where search starts from incorrect behavior of the program. Novice debugging research focuses on users who have a working knowledge of programming models (e.g., sequential execution) and program construction. Pirolli and Card [29] characterized intelligence analysts' sensemaking strategies in terms of a major loop and its subloops, reflecting the iterative nature of sensemaking. In addition to this interactive character, Grigoreanu et al. [15] introduced three major interweaved sensemaking loops that they termed as: bug fixing loop, environment loop, and common sense/domain loop.

Based on the results presented in their paper, Gross and Kelleher [16] implemented a tool that helps non-programmers to identify, extract, and reuse functionality from unfamiliar programs [15]. The tool uses a wizard-like interface to guide users through the code selection process. To help non-programmers identify the code responsible for target functionality, the tool correlates screenshots of program output with the code executing at that time, one of the primary activities that may assist in making mental mappings. By emphasizing the currently executing code, and providing affordances to locate the code in the program, the tool alleviates some program navigation difficulties. The tool asks a user to identify the beginning and ending lines of his or her target functionality, extracts it from the original program, and integrates it into the user’s program.

Vessey [36] observed that through the depth-first approach, expert programmers first familiarize themselves with a program before attempting to find the source of an error. Based on her observation and findings of the study, she developed a hierarchical “strategy path” that expert programmers imply while debugging programs. This is shown in Figure-2. Understanding the debugging strategies of expert programmers may also offer insight into suggested ways to build support into initial learning environments, such as block-based languages.

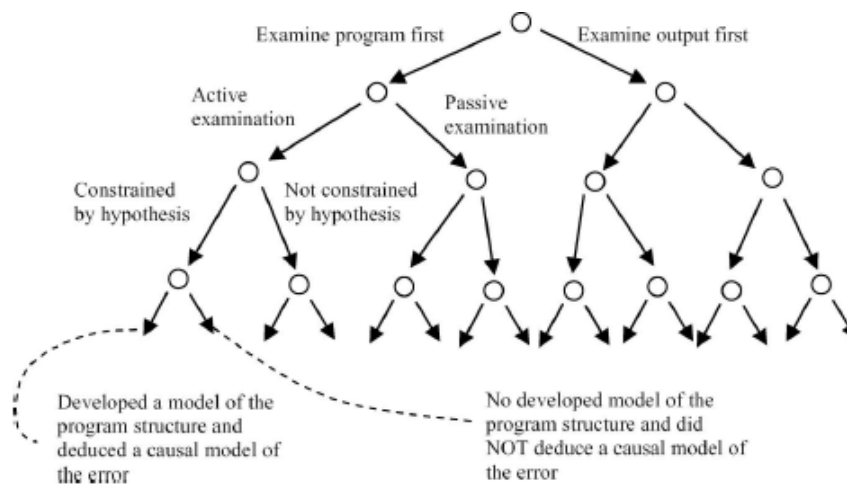


Figure-2. Vessey's possible debugging strategy paths for practicing programmers [36]

Carver and Risinger [6] provided another strategy similar to Vessey’s [36] that more closely resembles the overall troubleshooting framework developed by other contemporary researchers, such as Katz and Anderson [18]. Based on observation of the possible output, these authors develop a causal reasoning strategy for finding and fixing bugs when testing a program and finding it incorrect. This is shown in Figure-3.

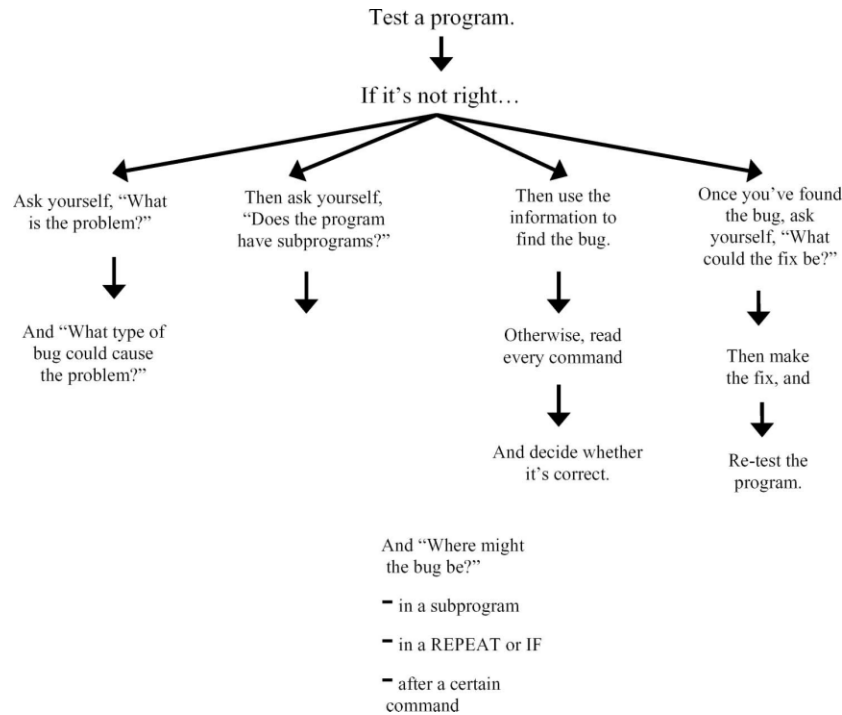


Figure-3. Carver and Risinger's [6] strategy for finding and fixing bugs

Vessey [36] found that experts used a breadth-first approach when debugging. Experts familiarize themselves with a program, gaining a systems view, before attempting to find the source of an error. On the other hand, novices take a depth-first approach, focusing on finding and fixing an error without regard to the overall program. Based on her study findings, Vessey [36] developed her strategy path using a hierarchy of goals that programmers use while debugging a program. The goals are: (1) determine the problem with the program (compare correct and incorrect output); (2) gain familiarity with the function and structure of the program; (3) explore program execution and/or control; (4) evaluate the program, leading to a statement of hypothesis of the error; and (5) repair the error.

In the current Blockly, Scratch, and Snap programming environments the basic mode of execution is run "all the time." These scripts take desired actions only when specific conditions are met, or constantly update the global state from a narrow viewpoint based on events that they observe. Ashrov et al. [2] urge that the breaking of an application into small independent behavior modules, facilitated by the behavioral programming design pattern, is especially helpful for visual programming languages like Snap or Scratch where the size of the screen limits the amount of visible information. Because long and complex code cannot be presented on one screen, programmers need to be able to break their applications into small independent pieces that can be understood and maintained in isolation.

Estler et al. [11] discussed the design and results of an empirical study that they conducted with an aim of identifying features that could enhance the effectiveness of collaborative debugging processes. Their study found collaboration as useful for effective debugging that improved programmers' overall debugging experience in collaborative settings. These results echoed findings of two other earlier studies that also found that collaborative programming improves design quality, reduces program defects and staffing risk, enhances technical skills, and makes programming enjoyable [7, 9]. Thus, the collaborative debugging process may be helpful for the first-time program learners for improving their debugging skills.

#### *Popularity, Programming Styles, and Limitations of BPLs*

The recent development of BPLs (e.g., Blockly, Scratch, and Snap) has received attention by K-12 teachers and novice programmers. Many universities and colleges have started using BPLs in courses either for their Freshman majors, or for a general non-majors course (e.g., at the University of Alabama, the CS 104 class uses BPLs as the programming environments, primarily to teach Math Education majors about programming). The popularity of BPLs suggests that their style of coding is more accessible than other standard programming languages, and perhaps

even more easily programmable than other visual programming languages. To achieve these goals, the developers of BPLs established some core design principles. For instance, the Scratch has been designed with three core design principles to make it more tinkerable, more meaningful, and more social than other programming environments. These have made the Scratch programming language as simple as the young children build some structures and stories while play with the Lego blocks; supported with diversified stories, games, animations, and simulations; easy to personalize the projects by creating graphics, importing photos, audio and video clips; and easy to program collaboratively by building, critiquing, and remixing one another's projects [30]. According to Kafai et al. [17], "programming is a form of expressing oneself and of participating in social networks and communities."; and "the development of creative and critical networks to share information and ideas stands as the model for students who wish to create a more collaborative and open society" [P. 9].

BPLs are often designed using a constructionist theory of learning that encourages people to share their own projects and to view and reuse projects created by other members in the global community [12]. With the current trend in social collaboration, many BPL programmers, especially younger students, look forward to sharing their programming skills, achievement, creation, and contribution through the web-based galleries. The recent versions of BPLs are moving to a browser-based delivery platform (e.g., Scratch, App Inventor, Snap!, and Blockly) and encouraging their programmers to upload, share, remix their programs into the language's online user forum. This is more advantageous for the young, novice, and freelance programmers who do not have enough opportunity to publish and share their programming skills in a printed medium or public forum. This can be observed by looking at the Scratch users' forum where there are more than 6.1 million projects, as of July 2014, shared online. Most of these programs are developed, or remixed collaboratively by non-professional programmers of all ages, especially, by young novice programmers. Thus, many of these shared programs are found as incomplete or with some sort of bugs.

Thus, the new research question arises: *What are the limitations in the existing BPLs and what is the next most essential improvement expected in the future versions of BPLs?* Our intention to search answers to these queries is to make BPLs accessible to more prospective programmers – in accordance with the designers of Scratch, Mitchel Resnick et al.'s motto, *Scratch: Programming for All* [30].

## 5. FUTURE DIRECTIONS

This section introduces future directions of research that are inspired by the current state-of-the-art presented in the previous section. The idea presented in this section is intended for further improvement in the debugging techniques and tools for the existing and future programming languages developed mainly for novice programmers. We identify and focus on an important prospective area for further research.

While the BPLs such as Scratch and Snap provide support to the first-time program learners for programmatic thinking and give them a good start in computer programming, there are some literal limitations of these languages. Two of such limitations are notable:

(1) *The BPLs are too much abstract:* Programming by blocks hides the standard program codes generated behind the blocks. Thus, the advanced and interested programmers do not have enough chance to see and edit the corresponding high-level language program codes. However, it is most likely that if these languages would have this option, some of the advanced programmers in these languages might switch to coding in other high-level programming languages. The recent version of Snap has some sort of this facility that a snap program can be exported (saved) as an .xml file format that can be imported (open) as snap programs, although most of the converted codes in .xml format are not understandable.

(2) *The BPLs do not have explicit debugger:* Programmers in a BPL can easily develop and run their programs without facing the intermediate steps between these. If someone includes some syntactically good but logically wrong block(s) he cannot debug or trace his program to trace out block by block output. Due to lack of explicit code search option sometimes, the existing help tips are not sufficient for him to figure out the exact program bugs. Although, most of the BPLs have their own and online community forum where expert participants upload their programs and give helping guidelines to others, sometimes those might not fulfill these demands.

### *Developing Collaborative Debuggers for BPLs*

The BPLs allow their programmers to construct programs using a drag-and-drop interface that removes the potential for syntax errors, although logic errors are still possible. They support the core programming constructs that novice

programmers need to learn, such as sequencing, iteration, and conditionals, as well as the creation of new customized blocks to support procedure abstraction. However, they cannot prevent many logical and runtime errors. They do not have adequate integrated debugging systems where a programmer can upload and place his incomplete or buggy programs to be debugged by other programmers. Many novice programmers leave their incomplete or buggy programs on the users' forum and may not look back to complete or fix their programs. Even if a program bug is fixed in a remixed or alternative version of an original program, the original programmer does not have enough chance to notice the fixes. However, there is no doubt that the BPLs encourage collaborative programming through program sharing. Studies have found that collaborative programming improves design quality, reduces defects and staffing risk, enhances technical skills, improves team communications, and makes programming significantly more enjoyable [7, 9]. Moreover, the BPLs do not have explicit support for searching program codes.

An analysis of the taxonomy developed by Kelleher and Pausch [19], with a goal of making programming more accessible to the novice, reveals that debugging is an important feature of programming languages, but that it is missing in almost all of the programming languages developed for novice programmers. This is a missed opportunity to explore the problem solving situations that may emerge when debugging a program, which could provide helpful insight to the novice programmer. Network supported communication and collaboration are also other important, but unimplemented, features that we want to consider for further research by developing a network-based collaborative debugger for the BPLs. Thus, the inclusion of new and innovative collaborative programming features can expand the BPLs to reach more potential novice programmers. In a collaborative debugging environment, programmers worldwide would be able to debug their programs collaboratively in synchronous and asynchronous modes.

A *collaborative debugger* is an integrated debugging system where programmers can debug each other's programs synchronously and asynchronously in a collaborative programming manner. Studies have found that collaborative programming improves design quality, reduces program defects and staffing risk, enhances technical skills, and makes programming enjoyable [7, 9]. Collaborative debugging also improves a programmer's debugging experience in collaborative settings [11]. Thus, the inclusion of collaborative debuggers into the BPLs has a scope of helping many novice programmers to get their uncompleted or buggy programs fixed by others.

In our future and extended research, we aim for developing such *collaborative debuggers* for BPLs where programmers worldwide would be able to upload and debug each other's programs collaboratively in synchronous and asynchronous modes. The initiator programmer will be able to upload and share a buggy program on a cloud-based collaborative debugger asking other programmers to help fix or explain the bug(s). Once another programmer fixes the bugs, the initiator will receive a notification as well as a copy of the updated version of the program. Thus, the development of collaborative debuggers seems to be an appropriate and timely improvement in the BPLs.

## ACKNOWLEDGEMENT

This material is based upon work supported by the National Science Foundation under Grant No. CE21 (CNS-1240944).

## REFERENCES

- [1] Ahmadzadeh, M., Elliman, D. and Higgins, C. 2005. An analysis of patterns of debugging among novice computer science students. In *ACM SIGCSE Bulletin*, vol. 37, no. 3, pp. 84-88.
- [2] Ashrov, A., Marron, A., Weiss, G. and Wiener, G. 2014. A use-case for behavioral programming: An architecture in JavaScript and Blockly for interactive applications with cross-cutting scenarios. *Science of Computer Programming*, In Press.
- [3] Beckwith, L., Kissinger, C., Burnett, M., Wiedenbeck, S., Lawrance, J., Blackwell, A. and Cook, C. 2006. Tinkering and gender in end-user programmers' debugging. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*. ACM, pp. 231-240.
- [4] Ben-Ari, M. and Maloney, J. 2012. Scratch project forum discussion. Retrieved July 28, 2014 from, <http://scratch.mit.edu/forums/viewtopic.php?id=8130>.
- [5] Bonar, J. and Soloway, E. 1985. Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human-Computer Interaction*, vol. 1, no. 2, pp. 133-161.
- [6] Carver, M. S. and Risinger, S. C. 1987. Improving children's debugging skills. In *Empirical studies of programmers: Second workshop*. Ablex Publishing Corp., pp. 147-171.
- [7] Chao, J. and Atli, G. 2006. Critical personality traits in successful pair programming. In *Agile Conference, 2006*. IEEE, vol. 5, pp.-93.
- [8] Chmiel, R. and Loui, M. C. 2004. Debugging: from novice to expert. In *ACM SIGCSE Bulletin*. vol. 36, no. 1, pp. 17-21.
- [9] Cockburn, A. and Williams, L. 2000. The costs and benefits of pair programming. *Extreme Programming Examined*, pp. 223-247.
- [10] Ducasse, M. and Emde, A. 1988. A review of automated debugging systems: knowledge, strategies and techniques. In *Proceedings of the 10th international conference on Software engineering*. IEEE Computer Society Press, pp. 162-171.
- [11] Estler, H. C., Nordio, M., Furia, C. A. and Meyer, B. 2013. Collaborative Debugging. In *Global Software Engineering (ICGSE), 2013 IEEE 8th International Conference on*. IEEE, pp. 110-119.
- [12] Federici, S. and Stern, L. 2011. A Constructionist Approach to Computer Science. In *Global Learn*, pp. 1352-1361.
- [13] Fix, V., Wiedenbeck, S. and Scholtz, J. 1993. Mental representations of programs by novices and experts. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*. ACM, pp. 74-79.
- [14] Furnas, G. W. and Russell, D. M. 2005. Making sense of sensemaking. In *CHI'05 extended abstracts on Human factors in computing systems*. ACM, pp. 2115-2116.
- [15] Grigoreanu, V., Burnett, M., Wiedenbeck, S., Cao, J., Rector, K. and Kwan, I. 2012. End-user debugging strategies: A sensemaking perspective. *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 19, no. 1, pp. 5.
- [16] Gross, P. and Kelleher, C. 2010. Non-programmers identifying functionality in unfamiliar code: strategies and barriers. *Journal of Visual Languages & Computing*, vol. 21, no. 5, pp. 263-276.
- [17] Kafai, Y. B. and Burke, Q. 2014. *Connected Code: Why Children Need to Learn Programming*. The MIT Press .
- [18] Katz, I. R. and Anderson, J. R. 1987. Debugging: An analysis of bug-location strategies. *Human-Computer Interaction*, vol. 3, no. 4, pp. 351-399.



- [19] Kelleher, C. and Pausch, R. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)*, vol. 37, no. 2, pp. 83-137.
- [20] Kelleher, C., Pausch, R. and Kiesler, S. 2007. Storytelling Alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI conference on Human factors in Computing Systems*. ACM, pp. 1455-1464.
- [21] Kessler, C. M. and Anderson, J. R. 1986. A model of novice debugging in LISP. In *Proceedings of the First Workshop on Empirical Studies of Programmers*, pp. 198-212.
- [22] Kidwell, P. A. 1998. Stalking the elusive computer bug. *Annals of the History of Computing*, IEEE, vol. 20, 4, pp. 5-9.
- [23] Ko, A. J. and Myers, B. A. 2004. Designing the Whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in Computing Systems*. pp. 151-158.
- [24] Malan, D. J. and Leitner, H. H. 2007. Scratch for budding computer scientists. *ACM SIGCSE Bulletin*, vol. 39, no. 1, pp. 223-227.
- [25] McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L. and Zander, C. 2008. Debugging: A review of the literature from an educational perspective. *Computer Science Education*, vol. 18, no. 2, pp. 67-92.
- [26] Meerbaum-Salant, O., Armoni, M. and Ben-Ari, M. 2011. Habits of programming in scratch. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*. ACM, pp. 168-172.
- [27] Moore, J. W. 1998. *Software Engineering Standards*. Wiley Online Library.
- [28] Papert, S. 1980. *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc.
- [29] Pirolli, P. and Card, S. 2005. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In *Proceedings of International Conference on Intelligence Analysis*. Mitre McLean, VA, pp. 2-4.
- [30] Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J. and Silverman, B. 2009. Scratch: Programming for All. *COMMUNICATIONS OF THE ACM*, vol. 52, no. 11, pp. 60-67.
- [31] Rosson, M. B., Sinha, H., Bhattacharya, M. and Zhao, D. 2008. Design planning by end-user web developers. *Journal of Visual Languages & Computing*, vol. 19, 4, pp. 468-484.
- [32] Spohrer, J. C. 1985. *Bug Catalogue II, III, IV*. Yale University, Department of Computer Science, New Haven, CT, USA.
- [33] Spohrer, J. G. and Soloway, E. 1986. Analyzing the high frequency bugs in novice programs. In *Empirical Studies of Programmers*, E. Soloway and S. Iyengar. Eds. Ablex. New York, pp. 230-251.
- [34] Subrahmaniyan, N., Beckwith, L., Grigoreanu, V., Burnett, M., Wiedenbeck, S., Narayanan, V., Bucht, K., Drummond, R. and Fern, X. 2008. Testing vs. code inspection vs. what else?: male and female end users' debugging strategies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 617-626.
- [35] Takahashi, J. M. 2012. Weighted code coverage tool, *U.S. Patent No. 8,291,384*. Washington, DC: U.S. Patent and Trademark Office.
- [36] Vessey, I. 1985. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, vol. 23, no. 5, pp. 459-494.
- [37] Zeller, A. 2009. *Why Programs Fail: A Guide to Systematic Debugging*. 2<sup>nd</sup> ed., Morgan Kaufmann.

## References

- [1] Ahmadzadeh, M., Elliman, D. and Higgins, C. 2005. An analysis of patterns of debugging among novice computer science students. In Anonymous *ACM SIGCSE Bulletin*. ACM, , 84-88.
- [2] Ashrov, A., Marron, A., Weiss, G. and Wiener, G. 2014. A use-case for behavioral programming: an architecture in JavaScript and Blockly for interactive applications with cross-cutting scenarios. *Science of Computer Programming*, .
- [3] Beckwith, L., Kissinger, C., Burnett, M., Wiedenbeck, S., Lawrance, J., Blackwell, A. and Cook, C. 2006. Tinkering and gender in end-user programmers' debugging. In Anonymous *Proceedings of the SIGCHI conference on Human Factors in computing systems*. ACM, , 231-240.
- [4] Ben-Ari, M. and Maloney, J. 2012. Scratch project forum discussion. , <http://scratch.mit.edu/forums/viewtopic.php?id=8130>.
- [5] Bonar, J. and Soloway, E. 1985. Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human-Computer Interaction*, 1, 2, 133-161.
- [6] Carver, M. S. and Risinger, S. C. 1987. Improving children's debugging skills. In Anonymous *Empirical studies of programmers: Second workshop*. Ablex Publishing Corp., , 147-171.
- [7] Chao, J. and Atli, G. 2006. Critical personality traits in successful pair programming. In Anonymous *Agile Conference, 2006*. IEEE, , 5 pp.-93.
- [8] Chmiel, R. and Loui, M. C. 2004. Debugging: from novice to expert. In Anonymous *ACM SIGCSE Bulletin*. ACM, , 17-21.
- [9] Cockburn, A. and Williams, L. 2000. The costs and benefits of pair programming. *Extreme programming examined*, , 223-247.
- [10] Ducasse, M. and Emde, A. 1988. A review of automated debugging systems: knowledge, strategies and techniques. In Anonymous *Proceedings of the 10th international conference on Software engineering*. IEEE Computer Society Press, , 162-171.
- [11] Estler, H. C., Nordio, M., Furia, C. A. and Meyer, B. 2013. Collaborative Debugging. In Anonymous *Global Software Engineering (ICGSE), 2013 IEEE 8th International Conference on*. IEEE, , 110-119.
- [12] Federici, S. and Stern, L. 2011. A Constructionist Approach to Computer Science. In Anonymous *Global Learn.* , 1352-1361.

- [13] Fix, V., Wiedenbeck, S. and Scholtz, J. 1993. Mental representations of programs by novices and experts. In Anonymous *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*. ACM, , 74-79.
- [14] Furnas, G. W. and Russell, D. M. 2005. Making sense of sensemaking. In Anonymous *CHI'05 extended abstracts on Human factors in computing systems*. ACM, , 2115-2116.
- [15] Grigoreanu, V., Burnett, M., Wiedenbeck, S., Cao, J., Rector, K. and Kwan, I. 2012. End-user debugging strategies: A sensemaking perspective. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 19, 1, 5.
- [16] Gross, P. and Kelleher, C. 2010. Non-programmers identifying functionality in unfamiliar code: strategies and barriers. *Journal of Visual Languages & Computing*, 21, 5, 263-276.
- [17] Kafai, Y. B. and Burke, Q. 2014. *Connected Code: Why Children Need to Learn Programming*. The MIT Press, .
- [18] Katz, I. R. and Anderson, J. R. 1987. Debugging: An analysis of bug-location strategies. *Hum. -Comput. Interact.*, 3, 4, 351-399.
- [19] Kelleher, C. and Pausch, R. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)*, 37, 2, 83-137.
- [20] Kelleher, C., Pausch, R. and Kiesler, S. 2007. Storytelling alice motivates middle school girls to learn computer programming. In Anonymous *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, , 1455-1464.
- [21] Kessler, C. M. and Anderson, J. R. 1986. A model of novice debugging in LISP. In Anonymous *Proceedings of the First Workshop on Empirical Studies of Programmers*. , 198-212.
- [22] Kidwell, P. A. 1998. Stalking the elusive computer bug. *Annals of the History of Computing*, IEEE, 20, 4, 5-9.
- [23] Ko, A. J. and Myers, B. A. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In Anonymous *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, , 151-158.
- [24] Malan, D. J. and Leitner, H. H. 2007. Scratch for budding computer scientists. *ACM SIGCSE Bulletin*, 39, 1, 223-227.
- [25] McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L. and Zander, C. 2008. Debugging: a review of the literature from an educational perspective. *Computer Science Education*, 18, 2, 67-92.

- [26] Meerbaum-Salant, O., Armoni, M. and Ben-Ari, M. 2011. Habits of programming in scratch. In Anonymous *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*. ACM, , 168-172.
- [27] Moore, J. W. 1998. *Software engineering standards*. Wiley Online Library, .
- [28] Papert, S. 1980. *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc., .
- [29] Pirolli, P. and Card, S. 2005. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In Anonymous *Proceedings of International Conference on Intelligence Analysis*. Mitre McLean, VA, , 2-4.
- [30] Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J. and Silverman, B. 2009. Scratch: programming for all. *Commun ACM*, 52, 11, 60-67.
- [31] Rosson, M. B., Sinha, H., Bhattacharya, M. and Zhao, D. 2008. Design planning by end-user web developers. *Journal of Visual Languages & Computing*, 19, 4, 468-484.
- [32] Spohrer, J. C. 1985. *Bug Catalogue II, III, IV*. Yale University, Department of Computer Science, .
- [33] Spohrer, J. G. and Soloway, E. 1986. Analyzing the high frequency bugs in novice programs.
- [34] Subrahmaniyan, N., Beckwith, L., Grigoreanu, V., Burnett, M., Wiedenbeck, S., Narayanan, V., Bucht, K., Drummond, R. and Fern, X. 2008. Testing vs. code inspection vs. what else?: male and female end users' debugging strategies. In Anonymous *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, , 617-626.
- [35] Takahashi, J. M. 2012. Weighted code coverage tool, .
- [36] Vessey, I. 1985. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23, 5, 459-494.
- [37] Zeller, A. 2009. *Why programs fail: a guide to systematic debugging*. Elsevier, .