Informatics in Education, 2020, Vol. 19, No. 1, 15–32

DOI: 10.15388/infedu.2020.02

An Approach to Evaluate the Complexity of Block-Based Software Product

Ilenia FRONZA[1], Luis CORRAL[2], Claus PAHL[1]
[1]*Free University of Bozen/Bolzano, Piazza Domenicani 3, 39100 Bolzano, Italy*
[2]*ITESM Campus Queretaro, Epigmenio Gonzalez 500, Queretaro, Mexico*
e-mail: ilenia.fronza@unibz.it, lrcorralv@tec.mx, claus.pahl@unibz.it

Received: July 2019

Abstract. Computer programming skills have been growing as a professional competence also to unqualified end-users who need to develop software in their professional practice. Quality evaluation models of end-user-developed products are still scarce. In this paper, we propose a metric that leverages "When", a condition typically found in block-based software development frameworks. We evaluated 80 Scratch projects collecting a metric related to the presence of the *When* condition and investigated common traits and differentiation with other metrics already proposed in the literature. We found that, in an evaluation with respect to the conditionals found in Scratch projects, *When* delivers a distinct and complementary approach to software complexity in products developed using block-oriented software development tools.

Keywords: software metrics, block-based programming languages, Scratch, complexity, When, End-User Software Engineering, EUSE, software quality.

1. Introduction

The acquisition of computer programming skills has been growing as a professional competence not only for specialists in Software Engineering or Computer Science but also to unqualified[1] end-users who often need to develop simple or complex software systems as part of their professional practice. The introduction of fully graphical, block-based software development tools like LabView or Simulink, or the inclusion of simple development tools like embedding Visual Basic in Office documents, allow users with little or no experience in software development to enable themselves as software programmers and deliver functional software products for enhanced productivity. Moreover, the features offered by tools like Scratch, Lego Mindstorms, App Inventor or Thunkable

[1] People who do not have either a degree in Computer Science/Software Engineering or extensive experience in software development.

enable people of all ages to the creation of fully functional software tools, starting in the early stages of their education.

The promotion of Computational Thinking principles in K-12 education provides a strategic framework for the development of competencies that foster the approach and orientation towards software development even though the student has not or will not pursue a Computer Science or Software Engineering career (Wing, 2014)(Fronza and Zanon, 2015). Enabling everybody to develop software can be seen as a very positive trait and expands the productive capacity of a professional (Fronza *et al.*, 2016); however, one of the main disadvantages is the overall low quality of a product that has not been developed professionally. Even if the errors may not be catastrophic, if in certain domains the effects are brought to a production environment, they can be relevant (Burnett, 2009). For example, software resources configured by end-users to monitor non-critical medical conditions can cause unnecessary pain or discomfort for users who rely on them (Orrick, 2006).

One of the reasons for the overall low quality of end-user-produced software is that most of the end-users lack of specific training in Software Engineering (Scheubrein, 2003). This situation poses in Software Engineering research the question of finding strategies to evaluate the quality of this kind of software product considering external quality aspects that go beyond the functionality of satisfaction perception of the software product in use (ISO 25010).

In this paper, we propose a new metric for assessing the complexity of the software products created by trainee developers. This assessment approach takes advantage of characteristics typically found in block-oriented software development frameworks like Scratch or App Inventor. We evaluated 80 Scratch projects collecting a metric related to the "When" event listener count, and we investigated the value delivered by this count, its common traits and differentiation with respect to other metrics already proposed in the available literature. We found that, in an evaluation with respect to other characteristics found in Scratch projects, "When" delivers a distinct and complementary approach to software complexity (that is, Cyclomatic Complexity) in products developed using block-oriented software development tools, shedding light on the way that trainees implement their knowledge in the form of a more complex software product.

The paper is structured as follows. Section 2 reviews background literature, related work, and relevant items for end-user software engineering. Section 3 introduces *When*, a metric to complement the structural/complexity metrics. Section 4 describes the case study that has been executed to investigate the correlation between the proposed metric and the existing complexity metrics. Section 5 reports the results obtained, and Section 6 discusses the findings of this study. Finally, Section 7 concludes the present work and provides directions for further research.

2. Background and Related Work

Block-Based Programming Languages (BBPL) have become a vital tool for an initial approach to software development both for students and professionals. Statistics pro-

vided by relevant BBPLs platforms count users and products in millions: as of 2019, Scratch repository hosts over 37 million projects created by over 35 million registered users[2]; App Inventor counts over 8.3 million users who have created over 34 million mobile apps[3].

The need to assess learners' software products has led to the onset of research into the analysis of code written in BBPLs, in particular with the goal of assessing Computational Thinking learning (Fronza and Pahl, 2018) by assessing the development of CT concepts, practices, and perspectives (Brennan and Resnick 2012) (Fronza *et al.,* 2017). Nevertheless, there are not many methods to analyze BBPL projects, and most of them have been designed for Scratch. Of note is the project called Dr. Scratch, which analyses a Scratch project to assign a Computational Thinking (CT) score and detects bad programming habits or potential errors (Moreno-León *et al.*, 2015). Another project called Ninja Code Village (Ota *et al.,* 2016) automatically assesses CT concepts in Scratch.

One research challenge into this area is the definition of the appropriate set of metrics that need to be extracted for quality assessment. As shown in Table 1, some effort has been spent on mapping the metrics that are used in professional programming to the BBPLs environment (Hermans and Aivaloglou, 2016). S. Grover (2017) described several difficulties novice programmers exhibit in introductory setting, such as assigning meaningful names to variables. J. Waite (2017) explored code smells in BBPLs. The same goal was pursued by F. Hermans and E. Aivaloglou (2016) for the specific case of Scratch.

Research and practitioner literature on Software Engineering offers different approaches to evaluate the complexity of a software product. Examples of standard metrics used in a professional setting are Cyclomatic Complexity and the Halstead suite of metrics; however, they use source code characteristics like conditions, decisions, and operators that are hard to replicate in a BBPL context. Moreover, it is necessary to un-

Table 1
Quality metrics for BBPLs

Metric	Definition	Ref.
Names	Percentage of components that have not been renamed	(Waite, 2017)
Superfluous stuff	Code blocks left lying around	(Waite, 2017)
Duplication	Similar computations or events occur in multiple places in the program (i.e., it could be implemented more elegantly, for example by using a loop)	(Waite, 2017), (Hermans and Aivaloglou, 2016)
Long method	A group of blocks grows very large, which implies lack of decomposition and design	(Waite, 2017) (Hermans and Aivaloglou, 2016)
Variables	Variables have a meaningful name	(Grover, 2017)

[2] `https://scratch.mit.edu/statistics/`

[3] `http://ai2.appinventor.mit.edu/stats`

derstand the conditions intrinsic to BBPL that can as well provide additional insights for the understanding of the complexity of a piece of software.

An approach to understanding the effectiveness of the educational process of trainee developers is to understand the complexity of the products they create. The term *software complexity* has been defined in several ways by different people. Basili (1980) defined complexity as a measure of resources expended by a system while interacting with a piece of software to perform a given task. The interacting system can be a computer or a programmer. In the first case, complexity describes the execution time and amount of storage needed to perform the computation. In the latter case, complexity is defined in terms of the difficulty of performing tasks such as coding, debugging, testing, or modifying the software (Kearney *et al.,*1986).

Software complexity metrics provide a quantified expression of the inherent characteristics of software (Kevrekidis *et al.*, 2009), such as reliability (Lew *et al.*, 1988), maintainability (Kevrekidis *et al.*, 2009) and numerous other quality factors of software systems. Increased software complexity means that maintaining and modifying will take longer, will cost more, and will result in more errors.

Although software complexity cannot be eliminated, it can be controlled with the use of a meaningful complexity metric to provide continuous feedback throughout a software project to help control the development process. For this reason, with the increased usage and sophistication of software applications, many programmers are looking at ways of minimizing the complexity associated with software and thereby reduce the maintenance costs associated with them. Therefore, many software complexity metrics have been proposed over the time (De Silva *et al.*, 2012).

Kaur and Verna (2016) provided a review of the various complexity metrics that can be retrieved in the existing literature. Among these metrics, Cyclomatic Complexity (CC) and Halstead's metrics have been widely used in Software Engineering to estimate maintenance effort and guide software testing, by identifying complicated and hard to maintain modules (Kafura and Reddy, 1987) (Moreno-León *et al.*, 2016).

The *Cyclomatic Complexity* (CC) metric was introduced by Thomas J. McCabe in 1976 to measure the maximum number of linearly independent paths through a control flow graph (McCabe, 1976), and is considered as an indicator of testability and maintainability of a program (Ammar *et al.*, 2001). *Halstead's metrics* identify specific properties of a program that can be measured and the relationships between them to assess software complexity (Halstead, 1977). These metrics have been compared in the last decades, with reported consistency between them (Henry *et al.*, 1981) (Moreno-León *et al.*, 2016). Finally, a simple yet expressive metric is *Lines of Code* (LOC), as it evaluates the size of a software product in terms of the number of executable lines of code, excluding comments and blank lines.

To understand the importance of the evaluation of the software product in an industrial or productive context, international standards like ISO/IEC 25010 define *Quality in use* as "the degree to which a product or system can be used by specific users to meet their needs to achieve specific goals with effectiveness, efficiency, freedom from risk and satisfaction in specific contexts of use". For End-User Software Engineering (EUSE), this means a perspective of quality from a standpoint in which the point of

view is of the user in the role of "end-user", as this dimension of quality is observable only when the final product is used in real-world execution conditions. Quality in use represents the point of view of the final customer concerning the quality of the product. Moreover, the subjective appreciation of the end-user influences the evaluation of the quality in use. ISO/IEC 25010 defines two dimensions of quality: in the context of EUSE, internal and external quality characteristics may define a view of "quality" from a standpoint in which the point of view is of the user in her/his role of "developer", making this perspective of high interest for our analysis. As ISO 25010 sheds light on strategies to evaluate relevant quality characteristics of the software product, it opens the door for proposing measuring strategies as it does not recommend how to track quality attributes concretely. No additional literature resources were found to lay the foundation on how to relate standardized software quality attributes to the context of BBPLs.

3. *When*: a Proposed Metric to Complement the Structural/Complexity Metrics

Evaluating the accomplishment of the educational process of a trainee requires a quantitative way to observe and assess the produced results. Understanding BBPL-originated software products implies the challenge of evaluating the quality of source artifacts without having traditional source code. The source artifacts are blocks that are sorted and matched to follow a flow, execute a sentence, or evaluate a condition; therefore, we need to take an approach that differs from traditional source code metrics. Structurally, as proposed by Cyclomatic Complexity or Halstead's metrics, we can evaluate the presence of conditions, decisions, or operators. However, it is necessary to be open to additional characteristics, intrinsic to BBPLs, that can shed additional light on the quality of the final software piece.

3.1. *Events*

In a common language definition, one can understand an event as "something that happens". This definition holds for traditional source code programming languages and BBPLs. Events are situations that occur during the flow of execution of a piece of software, and that can be caught or detected by a specific block of code called *Event Listener* to eventually drive or impact the flow of execution of the program.

3.2. *Event Listeners*

A relevant source artifact present in BBPLs is the *event listener*. Event listeners are lines of code that are placed to detect if a particular event has happened. Typically, an action or event handler is associated with the event, to indicate the action that shall be triggered in the case that the event is detected.

Depending on the programming language used, event listeners can be easily embedded into software code. Taking an example in HTML and JavaScript, we can see how web browser or web document events allow the JavaScript language to implicitly register different event handlers on elements in an HTML document. The event listener `onClick` awaits the user's click on the button to trigger the execution of the method "`execute_something()`".

```
<button onClick="execute_something()">Click here</button>
```

In the following example in the Java language, let **`myButton`** be an arbitrary button visible in the user interface. The method **`setOnClickListener`** registers an event interface to the button, while the **`OnClickListener`** method actually "listens" (awaits) for the button to be clicked. The lines of code enclosed by the method **`onClick`** handle the event, that is, define what shall be executed when is **`myButton`** pressed.

```
myButton.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
    }
});
```

In BBPLs, where lines of code are not evident, event listeners are represented in the form of blocks whose function is to "listen" or await for a specific execution circumstance to happen, and once such circumstance occurs, detect it so that a decision is made and an action is executed. Control flow palettes typically expose several events for the developer to leverage, and those blocks can be dragged and dropped always showing an enclosing structure to embed the series of instructions that are to be executed should the event happens during the execution of the program. Examples of events that are typically caught are *"When a Button is Clicked"* (Fig. 1), *"When a Sprite is Dragged", or "When a Sprite is touching a border"*, etc.

Event listeners can be roughly compared with conditions: conditions are logically evaluated according to an input and depending on the result (i.e., a TRUE or FALSE Boolean value), a decision is made, and an execution flow is followed. Similarly, event listeners do not necessarily calculate and evaluate a Boolean value but catch the presence of a certain execution circumstance and consequently trigger an action or impact an execution flow.

Fig. 1. Scratch event listener.

3.3. *Event Listeners*

We propose to leverage the presence of relevant event listeners to define an additional indicator of complexity specifically thought for BBPLs. Although event listeners are common in web and mobile programming, little research has been done to analyze the presence of event listeners to relate them to a software metric. For example, in the context of web programming, Watanabe *et al.* (2015) counted the number of JavaScript mouse event listeners to calculate a metric that identifies how focus navigation has been implemented on the web.

The metric proposed by this work is called "When" and represents the total number of *When* event listener blocks present in the collection of source blocks of the software piece. In Scratch, a When block is a graphic representation of an event listener that waits for an individual situation to occur to trigger an action. Those events can be relevant to GUI interaction (e.g., "When the user hits a key") or to interaction between elements of the process flow ("When a sprite touches a border").

We believe that, in the context of BBPLs, the count of When blocks might provide an indication of the increasing difficulty in modifying and understanding the code (i.e., complexity), since the presence of an event listener implies the necessity of evaluating a condition within the execution context, in a similar manner in which internal source blocks or source code conditions are evaluated and thus decisions are made. Traditional source code metrics like Cyclomatic Complexity or Halstead's metrics are not sensitive to event-listening and may leave out relevant complexity conditions given by the presence of event listener instructions. Furthermore, focusing our study on BBPLs positions our analysis in a development environment in which we believe that a direct way to associate functionality to user interface elements is via event handlers.

4. Case Study

The goal of our case study is isolating the independent value that the evaluation of event listening blocks may deliver to the complete understanding of the complexity of a BBPL-generated software piece. Therefore, we can formulate our hypothesis as follows:

> *Hypothesis:* The value delivered by the evaluation of the code block "When" provides an additional insight that does correlate to complexity and size metrics.

To acquire the numerical data to confirm or reject our hypothesis, we retrieved a collection of Scratch projects and ran several evaluations where we calculated *When* and other metrics to relate the presence of *When* blocks to the structural/complexity metric.

4.1. *Data Extraction*

For data extraction, we used Hairball, an open-source, static analysis tool to extract metrics from Scratch projects (Boe *et al.*, 2013). The tool is programmed in Python and provides plug-ins to perform different types of analysis[4]. In our case, we have used the *metrics* and *blocks* plug-ins to extract the following metrics:

- Total number of blocks (using *blocks.BlockCounts*);
- *Cyclomatic complexity* (using *metrics.CyclomaticComplexity*) of a Scratch project, which equals the number of decision points in the code plus one. Decision points are: *if %s then %s, if %s then %s else %s, repeat until, wait until, %s and %s, %s or %s*;
- *Halstead's Vocabulary* and *Length* (using *metrics.Halstead*) calculation is based on the number of distinct operators (n_2), the number of distinct operands (n_1), the total number of operators (N_2) and the total number of operands (N_1) in a program (Moreno-León *et al.*, 2016) (Ruan *et al.*, 2017). For example, in the case of the code snippets shown in Fig. 2 we have that: $n_1 = 4$, $N_1 = 4$, $n_2 = 3$, $N_2 = 4$. Operands: *when %s do, set %s to, %s – %s, get %s;* Operators: *tick, global time (*2), 1*. Based on these numbers, following the guidelines of (Moreno-León *et al.*, 2016), we take into account only Vocabulary and Length, which are calculated as follows: Vocabulary: $n = n_1 + n_2$; Length: $N = N_1 + N_2$.
- When (using *blocks.BlockCounts*): total number of "When" blocks (e.g., when a button is clicked).

4.2. *Study Sample*

The study sample consists of 80 randomly selected Scratch projects. Part of these projects have been retrieved online from a public Scratch source code repository[5]. Another part of the sample consists of projects that were collected by the authors when teaching Scratch in elementary and middle schools. The type and purpose of those projects are very diverse; some of them are simple animations, while others imply intensive user interaction and complex execution flows.

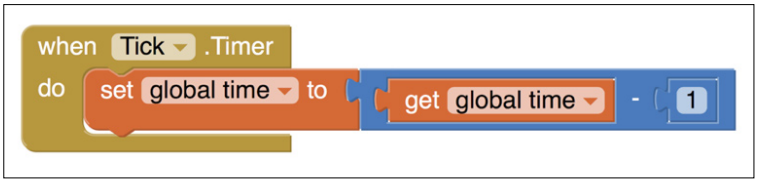

Fig. 2. App Inventor code snippet.

[4] https://github.com/ucsb-cs-education/hairball

[5] https://github.com/LLK/Scratch_1.4/tree/master/Projects

5. Results

Table 2 shows the descriptive statistics of the analyzed projects for the different metrics listed in Section 4.1.

Table 3 shows the correlations between each of the considered metrics. The values settling around 1, as well as the obtained p-values show that the proposed metric *When* has a positive, significant, and robust correlation with Cyclomatic Complexity, Length, and Vocabulary. The correlation indicates that the proposed metric *When* is in line with other, classic software engineering complexity metrics. As reported in the existing literature (Moreno-León *et al.*, 2016), Halstead's metrics and Cyclomatic Complexity also have a positive, significant strong correlation; the same happens between Halstead's Vocabulary and Length.

Fig. 3 shows the scatter plot of the metrics When and Cyclomatic Complexity, together with the best fitting line. The coefficient of determination is $r^2 = .81$ (p-value < 0.05), which indicates that, in a project, 81% of the variance of When metric can be predicted from Cyclomatic Complexity.

Fig. 4, which shows the scatter plot of the metrics When and Halstead's Vocabulary with the best fitting line, depicts a similar situation. In this case $r^2 = .60$ (p-value < 0.05), which states a lower accuracy of the linear model than the prior one. The scatter plot of the metrics When and Halstead's Length with the best fitting line ($r^2 = .73$, p-value < 0.05) shown in Fig. 5 illustrates a similar behavior.

The results detailed so far show that more complex projects (i.e., having higher CC, Vocabulary, and Length) also have higher values of When metric. However, we

Table 2

Metrics extracted from the Scratch projects in the study sample: descriptive statistics

	Min	1st qu.	Median	Mean	3rd qu.	Max
Total number of blocks	9.00	30.75	64.00	145.60	129.00	1566.00
Cyclomatic complexity	1.00	4.00	11.00	29.44	21.50	389.00
Halstead length	13.00	64.75	116.50	270.30	254.50	2805.00
Halstead vocabulary	11.00	30.50	46.00	60.31	75.25	340.00
When	1.00	4.00	8.00	22.90	21.00	177.00

Table 3

Correlation between metrics

	Cyclomatic Complexity	Halstead Length	Halstead Vocabulary	When
Cyclomatic Complexity	1	0.89**	0.64**	0.90**
Halstead Length		1	0.84**	0.86**
Halstead Vocabulary			1	0.78**
When				1

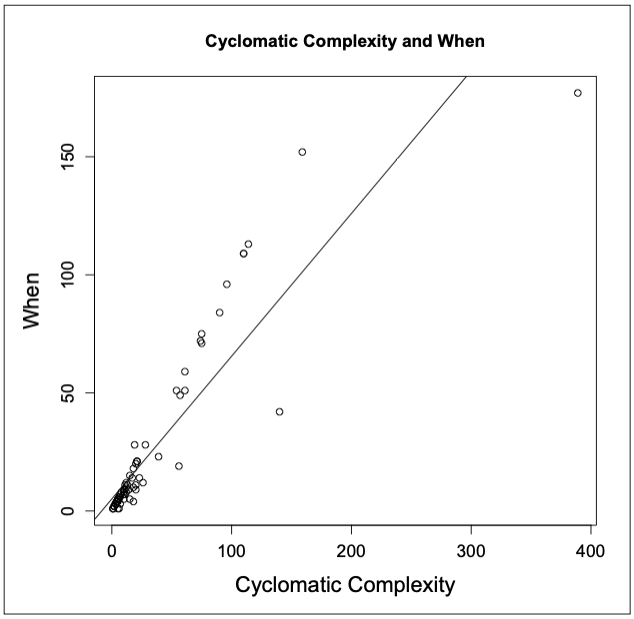

Fig. 3. Scatter plot of the metrics *When* and Cyclomatic Complexity, with best-fitting line.

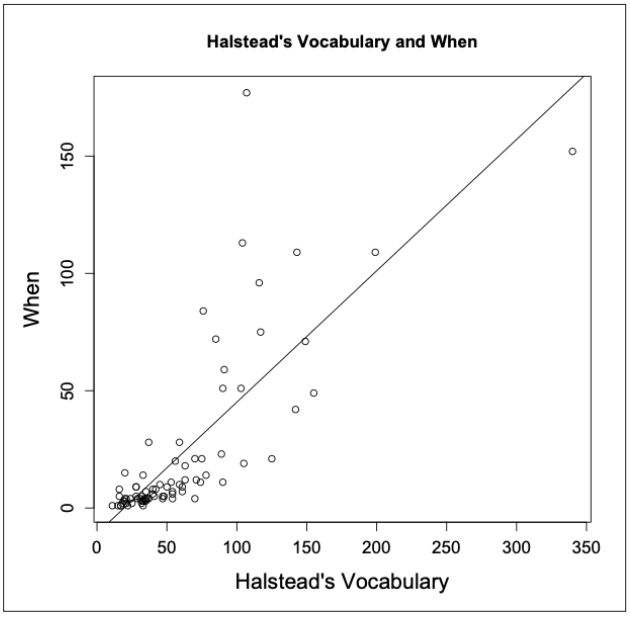

Fig. 4. Scatter plot of the metrics *When* and Halstead's Vocabulary, with best-fitting line.

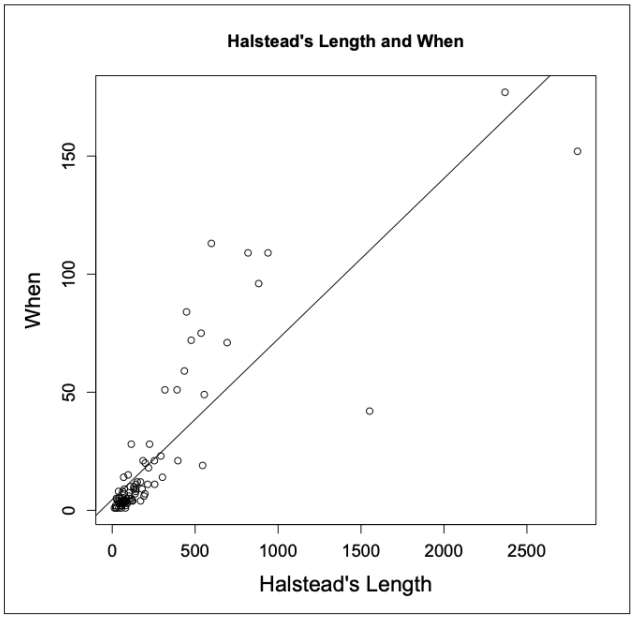

Fig. 5. Scatter plot of the metrics *When* and Halstead's Length, with best-fitting line.

are interested in focusing on projects having low complexity: the goal of the When metric, indeed, is to provide an easier indication of complexity, which might be especially useful for novices. For this reason, we are interested in checking if the same results hold for projects having lower complexity. As a rule of thumb[6], CC should always be lower than 20; thus, we selected only the 57 projects in our data set having CC $\leq$ 20, and we repeated the above-presented analysis. Table 4 and Table 5 show the descriptive statistics and the correlations between each of the extracted metrics, respectively.

Table 4

Metrics extracted from the Scratch projects in the study sample: descriptive statistics (when CC <= 20)

	Min	1st qu.	Median	Mean	3rd qu.	Max
Total number of blocks	9.00	26.00	40.00	49.47	70.00	138.00
Cyclomatic complexity	1.00	4.00	6.00	7.98	12.00	20.00
Halstead length	13.00	56.00	80.00	95.75	130.00	256.00
Halstead vocabulary	11.00	24.00	34.00	37.61	48.00	90.00
When	1.00	3.00	5.00	6.37	9.00	28.00

[6] `https://www.c-sharpcorner.com/article/3-tips-to-reduce-cyclomatic-complexity-in-c-sharp/`

Table 5
Correlation between metrics (when CC <= 20)

	Cyclomatic Complexity	Halstead Length	Halstead Vocabulary	When
Cyclomatic Complexity	1	0.69**	0.66**	0.80**
Halstead Length		1	0.90**	0.56**
Halstead Vocabulary			1	0.45**
When				1

The analysis of the projects having CC ≤ 20 reveals that the proposed metric *When* still has a positive, strong and significant correlation with Cyclomatic Complexity, while the correlation with Length and Vocabulary is lower respect to the previous data set (Table 2). The scatter plot in Fig. 6 indicates that, in a project, 64% of the variance of *When* metric can be predicted from CC because the coefficient of determination is $r^2 = .64$ (p-value < 0.05). This value is lower than before (i.e., when considering the entire data set), but still suggests that *When* can be considered as an indicator of complexity also for those projects having lower complexity.

The situation changes in Fig. 7 and Fig. 8, where the coefficients of determination are $r^2 = 0.30$ (p-value < 0.05) and $r^2 = .20$ (p-value < 0.05), respectively. This suggests that *When* cannot be predicted from Vocabulary or Length.

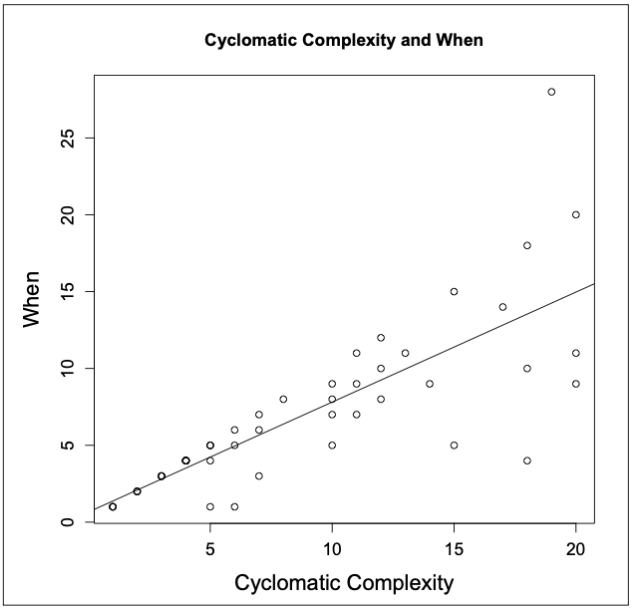

Fig. 6. Scatter plot of the metrics When and Cyclomatic Complexity, with best-fitting line (projects having CC <= 20).

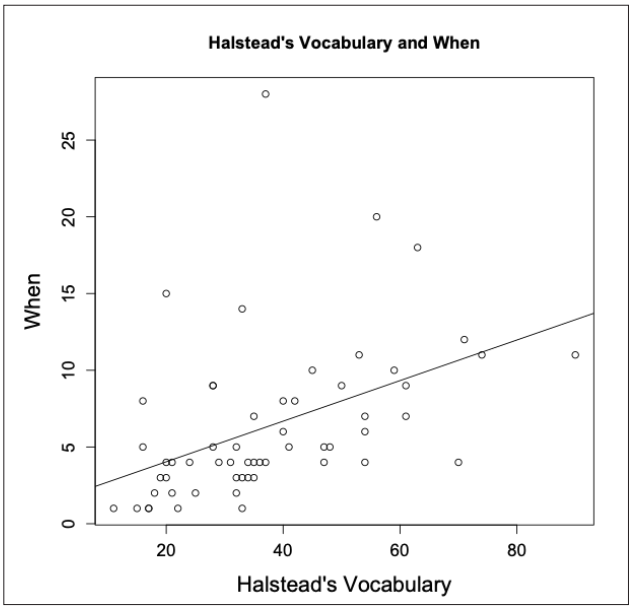

Fig. 7. Scatter plot of the metrics *When* and Halstead's Vocabulary, with best-fitting line (projects having CC <= 20).

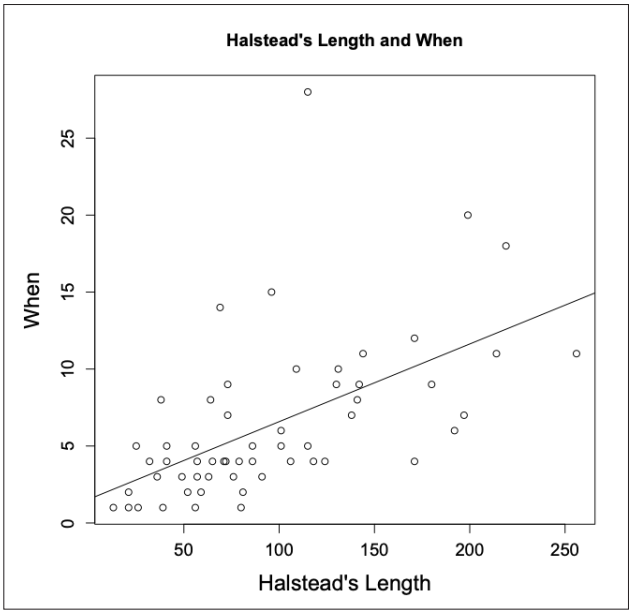

Fig. 8. Scatter plot of the metrics When and Halstead's Length, with best-fitting line (projects having CC <= 20).

6. Discussion

The quantitative analysis shows a positive, strong and significant correlation between the *When* count with respect to Cyclomatic Complexity, which grants significant level of reliability to predict one metric using the other. The analysis showed as well that Halstead metrics *Vocabulary* and *Length* cannot be associated (that is, predicted) by the presence and count of *When* event listeners.

Those assertions can be analyzed in a twofold way. On the one hand, the Cyclomatic Complexity is a metric that indicates a level of elaboration that is observable through the number of branches or possible ways that the flow of execution may take to reach the end of a piece of code. On the other hand, the analyzed Halstead metrics, Vocabulary and Length, are instead indicators of the size and operands of a computer program. The two approaches provide very distinct insights in terms of Software Engineering metrics.

As an event listener, *When* associates its behavior to the evaluation of "something that happens" and triggers the execution of a set of instructions. With this approach, it is easier to understand the stronger association it has with the Cyclomatic Complexity, that denotes the unfolding of different branches based on conditionals, and its loose tie with metrics that relate to size.

Our hypothesis claims that "the value delivered by the evaluation of the code block *When* provides an additional insight that does correlate to complexity and size metrics". Our data analysis confirms that the When count metric can be an efficient aid to predict Cyclomatic Complexity. It is worthy of discussion that *When* can be calculated more straightforwardly: the Cyclomatic Complexity metric is calculated following a formula that evaluates the decision points of the code and its eventual branching. The *When* metric, instead, is calculated more directly, as it only associates the count of *When* event listeners present in the analyzed set of blocks.

Moreover, in the context of the user that develops software using BBPL, assuming a beginner, non-expert profile, it is expected that the inexperienced developer prefers a linear approach to the solution of problems, and that the presence of traditional conditions such as *if* shall be lower (Venables, 2009), posing an additional challenge to calculate Cyclomatic Complexity. Instead, event listeners are needed even in very simple Scratch projects to trigger actions, including the execution of the project itself (typically with the block "When the flag is clicked"). The absence of conditions challenges to evaluate the complexity of a project using conditional-based metrics like Cyclomatic Complexity since it will return systematically lower values on that metric.

This opens the opportunity that, from a didactic and pedagogical point of view, evaluating *When* event listeners can shed light on understanding the complexity of projects developed by beginners or novice developers that do not have enough experience to structure or draw complex execution flows, yet can associate actions to events caught in the user interface or by the interaction of components within. In this way, instructors can leverage the calculation of this metric to understand how their educational strategies have an impact on a particular quality characteristic (i.e., complexity) of the software product developed by trainees.

7. Open Items and Conclusions

BBPL-originated programs can be complex even without the presence of multiple or nested conditions. Without source code, the evaluation of the complexity of such projects can be a hard and inefficient task. The presence of event listeners, difficult GUI blocks, and other BBPL-intrinsic conditions pose an additional challenge in understanding, developing, and maintaining pieces of software. Event listeners are found to be a relevant source of complexity for software pieces, however, they remain non-visible to traditional source code metrics like Cyclomatic Complexity or Halstead's metrics as they look for the presence of a particular code structure and take event listeners as a simple source code line, even though such line evaluates a condition and makes a decision.

Further analysis shall be done to determine if *When* can be useful to analyze the complexity of projects developed by beginners who do structure complex execution flows and that would eventually produce very linear projects with low Cyclomatic Complexity but more complex in terms of the events that are listened and handled. To this end, we recommend extending the analysis performed in this work associating to each project a metric that describes the level of expertise of the developer. In this way, it can be studied, discussed, and eventually confirmed if the presence of When blocks can relate effectively to evaluate the complexity of BBPL-based programs originated by novice developers.

As a means of future validation or reproducibility of this study, we recommend replicating the analysis proposed in this work using an alternative BBPL like App Inventor. App Inventor, like Scratch, is a BBPL framework that allows the development of mobile applications for the Android Operating System. App Inventor projects can as well be exported and eventually analyzed in Cyclomatic Complexity and event listener count (that is, When), to understand if the association found in Scratch projects holds as well in other BBPLs.

In this paper, we proposed an approach to evaluate the complexity of software products developed by non-experts. We leveraged the presence of standard event listener block in BBPLs like Scratch, to propose a software metric called *When*, whose purpose is to shed light on the complexity of BBPL-originated programs taking as fundamental characteristic the number of event listeners present in the program. We evaluated the count of *When* event listeners and the impact of such blocks in evaluating the complexity of the complete software piece. A case study run in 80 Scratch projects showed a significant correlation between the *When* count with respect to Cyclomatic Complexity, in particular in structurally complex projects. Projects that return a high Cyclomatic Complexity metric deliver as well a high *When* metric, with the consideration that calculating *When* is more straightforward as the metric only associates a block count.

Evaluating software metrics in traditional source-code based computer programs can be a challenging yet insightful task, that permits us to understand a more precise panorama on how the software is developed, how logic is constructed and how com-

plex the software product is. Extending this possibility to BBPLs is a precious resource since source code is not accessible, yet the logic is openly structured. Having a primary means to understand better the complexity of BBPL-based pieces of software opens as well the doors to have a better understanding on how non-expert profiles develop software, how the acquisition of software development competencies are growth, and how end-users create proficiency and attainment in the capacity of developing software autonomously.

References

Ammar, H., Nikzadeh, T., Dugan, J. (2001). Risk assessment of software-system specifications. *IEEE transactions on reliability*, 50, 171–183.

Basili, V. (1980). Qualitative Software Complexity Models: A Summary. In *Tutorial on Models and Methods for Software Management and Engineering*, IEEE Computer Society Press, Los Alamitos.

Boe, B., Hill, C., Len, M., Dreschler, G., Conrad, P., Franklin, D. (2013). Hairball: Lint-inspired Static Analysis of Scratch Projects. *SIGCSE Technical Symposium*. ACM.

Brennan, K., Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 Annual Meeting of the American Educational Research Association* (AERA'12). 1–25.

Burnett, M. (2009). What is end-user software engineering and why does it matter? *International Symposium on End User Development*. Springer.

De Silva, D., Kodagoda, N., Perera, H. (2012). Applicability of Three Complexity Metrics. *The International Conference on Advances in ICT for Emerging Regions – ICTer*, (p. 82–88).

Fronza, I., Zanon, P. (2015). Introduction of computational thinking in a hotel management school [Introduzione del Computational Thinking in un istituto alberghiero]. *Mondo Digitale*, 14 (58), pp. 28–34.

Fronza, I., El Ioini, N., Corral, L. (2016). Blending mobile programming and liberal education in a social-economic high school. *Proceedings – International Conference on Mobile Software Engineering and Systems, MOBILESoft 2016*, pp. 123–126.

Fronza, I., El Ioini, N., Corral, L. (2017). Teaching Computational Thinking Using Agile Software Engineering Methods: A Framework for Middle Schools. *ACM Trans. Comput. Educ.* 17, 4, Article 19.

Fronza, I., Pahl, C. (2018). Envisioning a computational thinking assessment tool. *CEUR Workshop Proceedings*, 2190.

Grover, S. (2017). Tackling novice learners naive conceptions in introductory programming. *Hello World.*

Halstead, M. (1977). *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc.

Henry, S., Kafura, D., Harris, K. (1981). On the relationships among three software metrics. *ACM SIGMETRICS Performance Evaluation Review*. Vol. 10. No. 1. ACM.

Hermans, F., Aivaloglou, E. (2016). Do code smells hamper novice programming? A controlled experiment on Scratch. *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on* (p. 1–10). IEEE.

Kafura, D., Reddy, G. (1987). The use of software complexity metrics in software maintenance. *IEEE Transactions on Software Engineering,* 3, 335–343.

Kaur, H., Verma, G. (2016). Software Complexity Measurement: A Critical Review. *International Journal of Engineering and Applied Computer Science (IJEACS)*, 12–16.

Kearney, J., Sedlmeyer, R., Thompson, W., Gray, M., Adler, W. (1986). *Software Complexity Measurement. Communications of the ACM*, 29, 1044–1050.

Kevrekidis, K., Albers, S., Sonnemans, P., Stollman, G. (2009). Software complexity and testing effectiveness: An empirical study. *2009 Annual Reliability and Maintainability Symposium*. Fort Worth, TX, US: IEEE.

Lew, K., Dillon, T., Forward, K. (1988). Software complexity and its impact on software reliability. IE*EE Transactions on Software Engineering*, 14(11), 1645–1655.

McCabe, T. (1976). A Complexity Measure. *IEEE Transaction on Software Engineering*, 6, 308–320.

Moreno-León, J., Robles, G., & Román-González, M. (2015). Dr. scratch: Automatic analysis of scratch projects to assess and foster computational thinking. *RED-Revista de Educacion a Distanci*a, 1–23.

Moreno-León, J., Robles, G., & Román-González, M. (2016). Comparing computational thinking development assessment scores with software complexity metrics. *Global Engineering Education Conference (EDUCON)* (p. 1040–1045). IEEE.

Orrick E. (2006). Position Paper, *Second Workshop on End-User Software Engineering, in conjunction with the ACM Conference on Human Factors in Computing*, Montreal, Quebec.

Ota, G., Morimoto, Y., Kato, H. (2016). Ninja code village for Scratch: Function samples/function analyser and automatic assessment of computational thinking concepts. *Visual Languages and Human-Centric Computing (VL/HCC)*, 2016 IEEE Symposium on (p. 238–239). IEEE.

Ruan, L., Patton, E., Tissenbaum, M. (2017). Evaluations of programming complexity in app inventor. *Siucheung KONG The Education University of Hong Kong*. Hong-Kong.

Scheubrein, R. (2003). Elements of end-user software engineering. *INFORMS Transactions on Education,* 4, 1, 37–47.

Venables, A., Tan, G., Lister, R. (2009). A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of the fifth international workshop on Computing education research workshop (ICER '09)*. ACM, New York, NY, USA, 117–128.

Waite, J. (2017). Do we pass on best practice when we teach block-based programming to primary school pupils? *Hello World*.

Watanabe, W., Dias, A., Fortes, R. (2015). Fona: Quantitative metric to measure focus navigation on rich internet applications. *ACM Trans. Web* 9, 4, Article 20 (September 2015), 28 pages.

Wing, J.M. (2014). *Computational thinking benefits society*. Retrieved October 18, 2019. `http://socialissues.cs.toronto.edu`

Xie, B., Abelson, H. (2016). Skill progression in MIT app inventor. *Visual Languages and Human-Centric Computing (VL/HCC), 2016 IEEE Symposium on* (p. 213–217). IEEE.

Xie, B., Shabir, I., Abelson, H. (2015). *Measuring the programmatic sophistication of app inventor projects grouped by functionality*. Retrieved October 18, 2019. `http://web.mit.edu/bxie/www`

I. Fronza is an assistant professor in software engineering at the Free University of Bozen-Bolzano, Italy. She received a M.Sc. degree in Mathematics from the University of Trento, Italy, and a PhD in Computer Science from Free University of Bozen-Bolzano. Her research interests lie in the software engineering field, specifically on software engineering training and education. This endeavour shall provide a better understanding, innovative techniques and tools for teaching software engineering, technology enhanced learning, and product assessment (also in non-conventional programming languages). Ilenia Fronza is guiding the Software Engineering Training Education research group, which aims at proposing educational techniques and tools to improve software development in production and educational ecosystems. Over the years, she has engaged a large number of students and educators in various projects.

L. Corral completed his Ph.D. at the Free University of Bozen-Bolzano, Italy, and his Master of Computer Science at the Autonomous University of Guadalajara, Mexico after a Bachelor of Science in Computer Systems Engineering at the Technological Institute of Queretaro, Mexico. Luis Corral has strong industrial experience. Through his career, he has held positions as Software Engineer in General Electric Aviation, leading globalized Quality Assurance processes for certifiable airborne software. He has a full commitment with education, training and development, serving as research fellow of the Faculty of Computer Science of the Free University of Bozen-Bolzano, Italy. Currently, he lectures Computer Science in the School of Information Technology and Electronics of ITESM, Campus Queretaro, and leads the Technical Education Programs at GE Infrastructure Queretaro. He is member of the Mexican National Research System, in the area of Engineering. His areas of interest are computational thinking, software quality assurance, mobile software engineering and energy aware mobile systems.

C. Pahl is a professor of computer science and vice-dean of research at the Free University of Bozen-Bolzano, Italy. His research interests include software engineering in service and cloud computing, specifically migration, architecture specification, dynamic quality, performance engineering, and scalability. Software engineering has been a continuous, cross-cutting concern. He received a Ph.D. in computing from the University of Dortmund and has held academic positions in Germany, Ireland, Denmark and Italy.