Baltic J. Modern Computing, Vol. 7 (2019), No. 4, 454-474
https://doi.org/10.22364/bjmc.2019.7.4.01

Complexity Metrics for Sassy Cascading Style Sheets

John Gichuki NDIA[1,2], Geoffrey Muchiri MUKETHA[2],
Kelvin Kabeti OMIENO[3]

[1] School of Computing and Informatics, Masinde Muliro University of Science and Technology, Kenya
[2] School of Computing and Information Technology, Murang'a University of Technology, Kenya
[3] School of Computing and Information Technology, Kaimosi Friends University College, Kenya

ndia.john@mut.ac.ke, gmuchiri@mut.ac.ke, komieno@kafuco.ac.ke

Abstract. Many front-end web developers are nowadays increasingly using sassy cascading stylesheets (SCSS) instead of the regular cascading style sheets (CSS). Despite its increased demand, SCSS has inherent complexity which arises from its features such as the use of nesting, inheritance, variables, operators, and functions. In addition, SCSS complexity, like all other software, continually increases with age. High complexity is undesirable because it leads to software that is difficult to understand, modify and test. Although there has been some metrics proposed to measure stylesheets complexity, these were defined in the context of regular CSS, and cannot be used to measure SCSS due to differences in their syntax. This paper proposes four metrics for measuring the complexity of SCSS code. The metrics have been used to calculate the complexity of three code snippets and three real-world projects and were found to be intuitional. The metrics were also evaluated using the Kaner framework and satisfied all the evaluation questions, indicating that they are sufficiently practical as required in the industry. In addition, the metrics were evaluated using Weyuker's properties, and results show that all the four metrics satisfied seven out of the nine properties, implying that they are theoretically sound.

Keywords: SCSS, rule blocks, complexity, complexity metrics, theoretical validation

1. Introduction

Cascading style sheets (CSS) language is a fundamental W3C standard that handles the presentation of the web documents written in Hypertext markup language (HTML), Extensible HTML (XHTML), and any Extensible Markup Language (XML) document to bring about aesthetically pleasing and user-friendly interfaces (Adewumi et al., 2012). In recent years, researchers and the industry have adopted the use of CSS to the extent that it has now become an integral part of web-based applications that separates structure from presentation (Adewumi et al., 2012; Geneves et al.,2012; Punt et al., 2016). However, to enable faster and maintainable development of CSS code, developers are shifting to the use of CSS pre-processors such as Sass, Less, Stylus, CSS-crush, Myth and Rework. CSS pre-processor is a program that processes or converts pre-processors code into CSS. According to Mazinanian and Tsantalis (2016), 54% of web developers

are now using CSS preprocessors and among these preprocessors, 92% of these developers prefer to use either SASS or LESS.

SASS pre-processor will be the focus of this study because it is increasingly being adopted by developers when compared to LESS pre-processor. Besides, governments such as the United States have recommended its use because it provides resources such as frameworks, libraries, tutorials and a comprehensive style guide as support (Mazinanian and Tsantalis, 2016). SASS pre-processor supports two syntaxes, Sassy CSS (SCSS) which uses the .scss extension and indented syntax which uses the .sass extension. SCSS is the newer of the two syntaxes and the most popular among front web developers for the following reasons: 1) It is a superset of CSS making migration to SCSS a lot easier, 2)It is easy to use the existing stylesheets and incorporate SASS features, 3) It is also more expressive meaning its more logically grouped, for example, one can compress several lines of codes in SASS into just fewer lines in SCSS (Cederholm, 2013). Fig. 1 shows a family tree of SASS pre-processor.

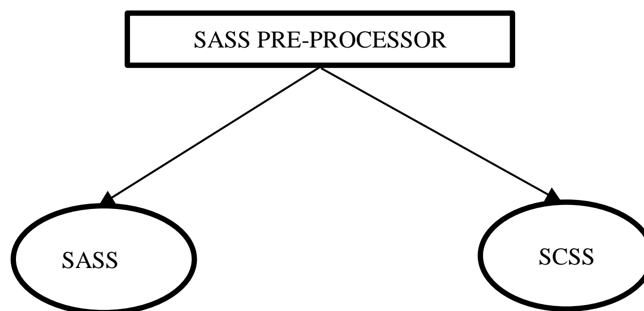

Fig. 1. SASS pre-processor syntaxes

SCSS has inherent complexity, due to the continuous growth of code size from features such as nesting, mixins, variables, inheritance, functions, operators, and control directives that are otherwise lacking in regular CSS. In addition, Web developers take a substantial amount of time to learn the SCSS language and about 46% of front web developers still prefer the use of regular CSS because they feel it has simple syntax (Mazinanian and Tsantalis, 2016). Software complexity leads to less reliable, understandable and maintainable software (Mesbah and Mirshokraie, 2012; Ghosheh et al., 2008; Muketha et al., 2010; Adewumi et al., 2012; Ogheneovo, 2014). The use of software complexity metrics has been recognized in software engineering as a way of controlling the complexity of software. According to Muketha et al., (2010), Parthasarathy and Anbazhagan (2006), software metrics inform on the success and failure level of software and the areas to improve the software.

In the style sheets domain, there is very little research regarding their complexity, mainly because it’s relatively a new area (Punt et al., 2016; Mesbah and Mirshokraie, 2012). The only software metrics proposed in style sheets domain are the six metrics defined by Adewumi et al., (2012), to measure the complexity of regular CSS. Although these metrics are promising, they cannot be used to measure SCSS complexity because of its unique features. Therefore, there is a need to define complexity metrics for SCSS.

2. The Structure and Complexity of SCSS

This section presents a detailed description of the structural features of SCSS and then attempts to relate these to SCSS complexity.

2.1. SCSS structure

The basic building component of an SCSS is a rule block. A rule block is made up of a selector and one or more attributes (Adewumi et al., 2012). The selector points to the HTML element to be styled while attributes specify the style on the element. An attribute is also known as the property name and can have one or more values. SCSS has other blocks such as mixin blocks (comprising of a @mixin directive with opening and closing braces), function blocks (comprising of @function directive with opening and closing braces), control directives block (comprising of control directive i.e. @if, @each, @for, @elseif with opening and closing braces), and media blocks (it comprises of @media with opening and closing braces). In this paper, all the various kind of blocks are referred to as SCSS blocks. An SCSS block is defined as any block that consists of a selector or @rule directive, opening brace, set of attributes and/or directives and a closing brace.

Sassy CSS is a style sheet language whose aim is to determine how the web pages are presented. In contrast, the aim of conventional programming languages such as Java, C++, etc. is to automate processes. Basically, SCSS is used to describe data while regular programming languages modify data. There are several differences between SCSS and regular programming languages. Table 1 presents the differences between SCSS and other structured and object-oriented software.

Table 1. Comparison between software programs and SCSS programs

Criteria	Software program	SCSS code
Modularized by	Modules/classes	SCSS block e.g. rule block, function directive block, mixin block, etc.
Coordinating module	Main program, class, module or method to coordinate all others	None
Program statements	Simple statements e.g. assignment.	Attributes and rule directives.
Control-flow statements	Sequence, branch, loop, and calls	Branch, loops, and calls
Data types	Variables/constants	Variables
Data definition	Each language defines its own data types	SCSS relies on SASS Pre-processor data types
Programming scope	Programs for performing calculations e.g. finding the sum of two numbers	Programs for formatting the presentation of web pages. e.g. assigning font size 12 to a paragraph

A simple alert rule block is shown in Fig. 2 with three regular attributes, i.e. padding, font-size, and text-align. Padding has been used to generate a space of 15px around the content of an element while font-size sets the size of text as 1.2em. Finally, text-align centers the content of the element where the alert class is implemented.

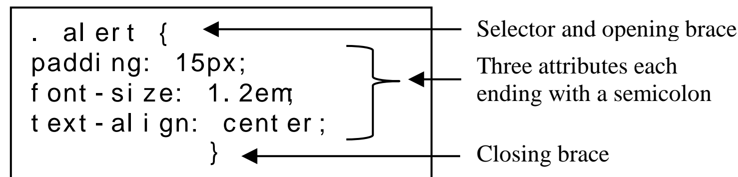

Fig. 2. An alert rule block

An illustration of multiple blocks is shown in Fig. 3. The figure has one mixin block which can be called in various places of the code. It also has five rule blocks where the three of them are nested. Fig. 3 also demonstrates the use of variables and selector inheritance.

The formal definition of an SCSS block $SCSS_B$ is

$SCSS_B = <A, D>$

An SCSS block ($SCSS_B$) is a 2-tuple <A, D>, where A is the set of attributes, and D is the set of directives such as mixin directives, control directive, function directive, and media directives.

2.2. SCSS Complexity

Several researchers have defined software complexity as the extent to which the software is difficult to understand (Harrison et al., 1999; Muketha et al., 2010). In order to manage the complexity of software, studies have shown that the factors responsible for it should be identified before defining metrics. The complexity determinant factors in software are size and length of the software (Muketha et al., 2010; Adewumi et al., 2012; Misra and Cafer, 2012; Khan et al., 2016), control flows (McCabe, 1976; Cardoso, 2007; Muketha et al., 2010; Misra and Cafer, 2012), use of operators (Halstead, 1977; Misra and Cafer, 2012), use of function calls (Shao and Wang, 2003; Misra and Cafer, 2012), use of variables (Misra and Cafer, 2012; Kushwaha and Misra, 2006), nesting (Piwowarski, 1982; Li, 1987; Chhillar and Bhasin, 2011; Frain,2013), inheritance (Chawla and Nath, 2013; Gill and Sikka, 2011; Chung and Lee, 1992; Misra et al., 2011), and coupling (Stevens et al., 1974; Chidamber and Kemerer,1994; Li and Henry, 1993; Abreu et al., 1996).

In stylesheets domain, the factors that contribute to its complexity are the size of CSS, rule block structures varieties, rule block reuse, cohesion and number of attributes (Adewumi et al., 2012). However, these factors are only limited to regular CSS and it's not indicated which process was used to identify them.

In the software engineering field, there are several software metrics proposed to measure and control software complexity. In the domain of stylesheets, there are few complexity metrics defined In Adewumi et al. (2012), proposed some metrics for regular CSS which influenced this study, for example, Rule length (RL) and Number of Rule Blocks (NORB), which are an adaptation of the Lines of Code (LOC). However, these metrics based on their definition, cannot be directly applied to SCSS code. RL considers a rule to be any of the following; a selector plus an opening brace, attributes that end with a semicolon, and a closing brace. This leaves out other rules in SCSS which are

executable such as @extend, @include and declaration of variables. The other metric which is NORB doesn't cover other blocks available in SCSS such as mixin block, function blocks and control directive block. Therefore, these existing metrics are limited because they fail to show the actual size of the SCSS code. This implies that the metrics don't give enough information the SCSS designers require, for example when to redesign a large SCSS block.

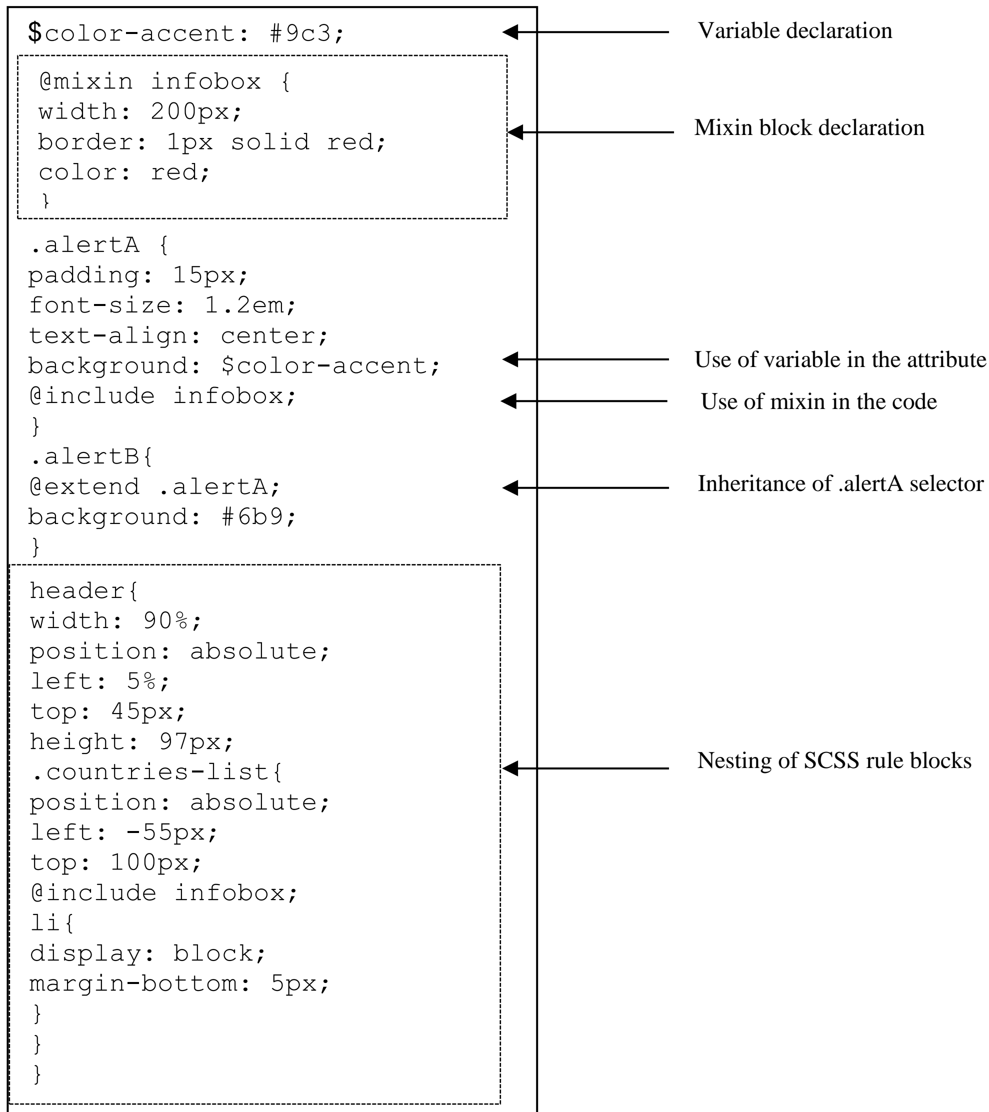

Fig. 3. SCSS code with multiple blocks

The Number of attributes Defined per Rule Block (NADRB) metric informed this paper and its essence is to compute the complexity of a rule block by counting the number of all attributes and divide by the number of rule blocks. This implies that the higher the average number of attributes per rule block, the more complex the CSS code. This metric is limited because it doesn't, for example, consider the use of control flows and function calls in rule blocks, meaning the SCSS designers can't tell when to redesign a very complex SCSS block.

The class inheritance factor (CIF) metric motivated the definition of a new metric for inheritance in SCSS code. The CIF metric computes the ratio of the sum of all ancestors for all classes divided by the maximum possible inheritance for the system. The inheritance is strictly one-way, meaning if class X extends class Y, then class Y cannot extend class X. Therefore, the maximum inheritance level for a system is 0 + 1 +……+ (n-1) (Mayer and Hall, 1999). The CIF metric is promising in comparison to method inheritance factor (MIF) and attribute inheritance factor (AIF) because in OOP it's the classes that are extended and not methods or attributes. However, the usefulness of CIF is yet to be established because it has not been validated.

The metric defined for SCSS coupling in this paper extends the coupling between objects (CBO) metric which is used in the object-oriented domain. CBO is a count of the number of classes that are coupled to a certain class. This metric is really promising, but it requires some adaptation so that it can be used for SCSS measurement.

3. Metrics Definition

The proposed metrics are derived from existing CSS metrics and other software metrics through the process of modification. This study followed the Entity-Attribute-Metric model in the definition of metrics for SCSS (Fenton and Pfleeger, 1997). In this paper the interesting attributes identified to be measured from SCSS program include;

i. Cognitive complexity of SCSS blocks
ii. Nesting level for SCSS code
iii. Selector Inheritance level for SCSS code
iv. Coupling level of SCSS code

The metrics identified to measure each of the attributes are;

Average Block Cognitive Complexity for SCSS ($ABCC_{SCSS}$)

The metric $ABCC_{SCSS}$ extends Number of Attributes Defined per Rule Block (NADRB) and is used to compute the complexity of a rule block in regular CSS. NADRB metric calculates complexity by determining the average number of attributes defined in the rule blocks. The proposed $ABCC_{SCSS}$ metric will consider other factors beyond the number of attributes, such as @rule and directives, operators, function calls, and variables.

The following are the factors identified that contribute to the SCSS block complexity:

i. Number of regular attributes (NRA): According to Adewumi et al., (2012), the more the number of attributes in a rule block the more complex it becomes.

ii. Number of operators (NO): Researchers have recognized the number of operators as a factor that contributes to the complexity of code (Misra and Cafer, 2012; Halstead, 1977)

iii. Use of control directives: The control directives contribute to the complexity of code as supported by various studies (Muketha et al., 2010; Misra and Cafer, 2012; McCabe, 1976). In rule blocks, the use of control directives is assigned weights as shown in Table 2. The weights are adopted from Törn et al. (1999) who proposed a value of 1.3 for a branch and 1.5 for a loop. The number of branch statements (NB) and the number of looping statements (NL) are considered.

iv. Number of function calls (NFC) is also supported by studies as an aspect that contributes to code complexity (Misra and Cafer, 2012; Shao and Wang, 2003). An attribute with a function call is assigned a weight of 1.3 like a branch, this weighting is informed in consistence with the way Misra and Cafer (2012) assigned selection/branch statements and function calls with the same weight.

v. Number of mixin calls (NMC): The @include statement simply calls a certain declared mixin in the code. This increases complexity because what is being called is in a different place in the code. @include directive rule is weighted at 1.3 the same as the function calls.

vi. Number of extend directives (NE): This rule directive inherits a selector, meaning that code complexity increases when it's implemented. @extend directive rule is weighted at 1.3, just like function calls because some code in a different place is being referred.

Table 2. Weights for basic control structures

Type of directive	Statements	Cognitive weight
Use of @rule directives	@include and @extend	1.3
Branch	@if , @else if , if ( ) and function calls, mixin calls, use of extends	1.3
Loop	@for, @while, and @each	1.5

To calculate $ABCC_{SCSS}$, the complexity of each SCSS block is computed herein referred to as Block Cognitive Complexity (BCC). The sum of complexity of all SCSS blocks is computed and is represented by the Total Block Cognitive Complexity metric (TBCC). TBCC is then divided by the number of all SCSS blocks (NOBL). NOBL is a simple size metric that counts all the blocks used in SCSS.

i. BCC = NRA + NO + (NB*1.3) + (NL * 1.5) + (NFC * 1.3) + (NMC * 1.3) + (NE *1.3)

ii. $TBCC=\sum_{i=1}^{n} BCC_i$

Where *n* is the total number of SCSS blocks

iii. $ABCC_{SCSS}$ = TBCC / NOBL

Nesting Factor for SCSS (NF_{SCSS}): Nesting refers to the enclosing of constructs such as if, while, for and each inside other constructs. Nesting increases program complexity (Li, 1987). SCSS allows nesting of CSS rules inside each other instead of repeating selectors in a separate declaration (Cederholm, 2013). According to Frain (2013), the nesting of rules should be kept as shallow as possible otherwise, it reduces the maintainability of the code. This means the higher the nesting level the more complex a program.

Regular CSS doesn't have nesting feature, therefore nesting concept in SCSS is borrowed from structured programming languages and object-oriented programming (OOP) languages. However, nesting in SCSS has an extra component as compared to other languages. In the regular programming languages when defining metrics only nesting depth is usually considered, while in SCSS we should consider nesting depth and nesting breadth. Fig. 3 demonstrates nesting depth where we have *countries-list* rule block inside *header* rule block and *li* rule block inside *countries-list* rule block.

Nesting breadth refers to having several independent rule blocks inside a single rule block. For example, in Fig. 4 the *countries-list* rule block and the *li* rule block are two independent rule blocks inside the *header* rule block. The two blocks *countries-list* and *li* rule blocks have no relationship with each other, only that they share the features of the *header* rule block. However, the nesting breadth is not considered with the control directives of SCSS, since all the nested blocks have a relationship with each other.

```
header{
width: 90%;
position: absolute;
left: 5%;
top: 45px;
height: 97px;
.countries-list{
position: absolute;
left: -55px;
top: 100px;
@include infobox;
}
li{
display: block;
margin-bottom: 5px;
}
}
```

Fig. 4. SCSS code with nesting breadth

In the computation of the nesting depth, a metric value of 1 will be assigned to the first level, a value of 2 to the second level, a value of 3 to the third level and so on (Chhilar and Bhasin, 2011). A nesting depth of 3 means we have three levels of nesting, meaning the depth cognitive complexity (DCC) value is 3+2+1=6 and if it's a nesting depth of 5 then DCC value will be 5+4+3+2+1=15. The calculation of nesting breadth

simply counts the number of SCSS blocks inside a single SCSS block. Therefore, if there are two independent rule blocks in a single block, then the complexity is assigned as 2.

Therefore, the proposed metric NF_{SCSS} is meant to compute the nesting level by considering the total depth nesting depth (TDNL) and the total breadth nesting level (TBNL) of all SCSS blocks.

i. DCC $=\sum_{i=0}^{m-1}(m-i)$
   Where m is the nesting depth
ii. TDNL$=\sum_{k=1}^{n} DCCk$
   Where n = number of SCSS blocks
iii. TBNL = number of independent blocks in different single rule blocks
iv. NF_{SCSS} = TDNL * TBNL

Selector Use Inheritance Level (SUIL): This metric measure complexity brought about by inheriting selectors in SCSS. Though there is a form of inheritance in the regular CSS, it doesn't allow inheritance of selectors. The inheritance concept in SCSS is borrowed from the object-oriented software. Therefore, the SUIL metric for SCSS is motivated by the class inheritance factor (CIF) metric of the OOP domain.

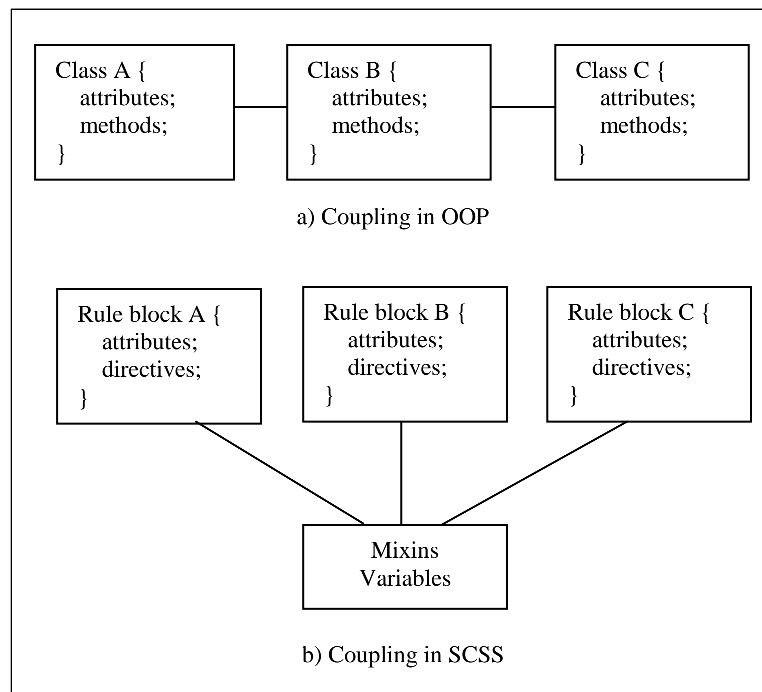

Fig. 5. Difference in coupling between OOP and SCSS

The proposed SUIL modifies the CIF metric and is calculated by taking the sum of all inherited selectors which is divided by the total number of all selectors.

$$SUIL = \sum_{i=1}^{n} NSI \,/\, \sum_{i=1}^{n} NS$$

where NSI is the Number of all selector inheritance instances and NS is the Number of all selectors in the program and n is the number of SCSS blocks.

Coupling Level for SCSS (CL_{SCSS}) metric

Coupling is the measure of the strength of association established by a connection from one class to another (Stevens et al., 1974; Chidamber and Kemerer, 1994). In OOP, coupling occurs when methods of one class use methods or variables of another class. In SCSS, coupling occurs when rule blocks share mixins and variables. The more the rule blocks sharing the same mixin or variable, the higher the coupling level. Fig. 5a demonstrates coupling in OOP, where Class B methods and variables can be accessed by both Class A and Class C. In Fig. 5b, the mixins, and variables are global data which are shared by Rule block A, Rule block B and Rule block C.

A need for a new metric for measuring coupling level in SCSS arises. The CL_{SCSS} metric is proposed and it's computed by summing the number of all declared mixins (NDM) with the number of all declared variables (NDV) which is then divided by the summation of all the number of mixin calls (NMC) and total number of all variable instances (NVI) in the program.

$$CL_{SCSS} = (NDM+NDV) \,/\, \left(\sum_{i=1}^{n} NMC + \sum_{i=1}^{n} NVI \right)$$

where n is the number of SCSS blocks in the program

4. Computing Metrics Values for SCSS Code

Three code snippets and three real-world projects have been identified for demonstrating how the proposed metrics are to be computed. These are presented in the subsequent sections.

4.1. Computing the Metrics Values of Three Code Snippets

As an initial step, three code snippets are analyzed using the proposed metrics with the aim of ascertaining whether the metric values are intuitional. These code snippets are presented in Appendix 1. Snippet 1 has 14 SCSS blocks, 3 mixins declarations, 2 variables declaration, 5 mixin instances, 3 variable instances, 1 extend directive, 5 operators, 9 selectors, 1 for statement and nesting feature is used. Snippet 2 has 14 SCSS blocks, no mixins declared, I variable declaration, 2 variable instances, 1 function call, 3 extend directives, 6 operators, 10 selectors, and nesting feature is implemented. Snippet 3 has 9 SCSS blocks, 1 mixin declaration, 2 variable declarations, 2 mixin instances, 3 variable instances, 1 extend directive, no operators, no control directives,8 selectors and nesting feature is implemented. These results are presented in Table 3.

Table 3. Summary of metric values for the code snippets

Metrics	Snippet 1 values	Snippet 2 values	Snippet 3 values
$\mathbf{ABCC_{SCSS}}$	1.90	1.99	1.76
$\mathbf{NF_{SCSS}}$	1.0	12.0	3.0
SUIL	0.11	0.30	0.13
$\mathbf{CL_{SCSS}}$	0.63	0.50	0.60

4.2. Computing the Metrics Values of a Real Project

In order to fully establish the intuition of the proposed metrics, SCSS code from three real-world projects was obtained by using google advanced search feature. Using this feature, project files with .scss extension were identified and downloaded from the Web. These files are located in the following website links:

1. http://happy-shala.com/sass/
2. http://www.greatjewishmusic.com/Midifiles/Rosh-Hashana/sass/
3. http://www.mce.ie/public/js-webshim/dev/shims/styles/scss/

The main file considered for analysis in the first website link is called style.scss. This file depends on two other files, namely, the mixins.scss and the vars.scss. These three files were downloaded and analyzed together. The main file considered for analysis in the second website link is called style.scss. The file also depends on two other files, namely, mixins.scss and vars.scss. The three files were downloaded and analyzed together. The main file considered for analysis in the third and final website link is called shim.scss. This file, together with its two dependent files, api-shim.scss and extends.scss were downloaded for analysis. Metrics results obtained after analysis of these three websites are presented in Table 4.

Table 4. Summary of metric values for the real-world projects

Metrics	happy-shala.com	greatjewishmusic.com	mce.ie
$\mathbf{ABCC_{SCSS}}$	2.58	2.17	2.9
$\mathbf{NF_{SCSS}}$	6960	8019	3034
SUIL	0	0	0.03
$\mathbf{CL_{SCSS}}$	0.31	0.27	2.33

5. Theoretical Validation Results

Theoretical validation of metrics is an important step in the definition of new metrics because it shows that the metrics have a sound mathematical foundation. Therefore, the proposed metrics have been validated using Weyukers properties and Kaner framework. Weyuker's properties have been used by several researchers to evaluate their proposed software metrics and they agree to the fact that it's a necessary framework and that for a measure to be valid it must satisfy most of its properties (Cherniavsky and Smith,1991; Abreu and Carapuca,1994; Chidamber and Kemerer,1994; Gursaran,2001; Sharma et al., 2006; Muketha et al., 2010; Baski and Misra, 2011). The Kaner framework has been used by a number of researchers (Adewumi et al., 2012; Baski and Misra, 2011), and has been applied in this paper for practical evaluation of the proposed metrics. The summarised Weyuker's results are presented in Table 5.

5.1. Validation with Weyukers properties

Property 1: (∃P) (∃Q) (|P| ≠ |Q|) where P and Q are two different SCSS blocks. This property is satisfied when there exist SCSS blocks P and Q such that |P| is not equal to |Q|. Therefore, if we can't find two SCSS blocks of different complexity, then all SCSS blocks have the same complexity value. All the metrics proposed $ABCC_{SCSS}$, NF_{SCSS}, SUIL, and CL_{SCSS}, return different complexity value for any two SCSS blocks that are not identical and therefore they satisfied this property.

Property 2: Let c be a non-negative number. Then there are finitely many SCSS blocks of complexity c. This property asserts that if an SCSS block changes then its complexity changes. When the number of attributes is changed, complexity values change for the $ABCC_{SCSS}$. In addition, when the number of extend rule directives changes then SUIL value change, and when the number of include statements and variables change then CL_{SCSS} metric value changes. In addition, NF_{SCSS} metric value changes when you reduce or increase nested SCSS blocks, meaning it also satisfies this property.

Property 3: There can exist distinct SCSS blocks P and Q where |P| = |Q|. This property affirms that two different SCSS blocks can have the same metric value, this is to say that two SCSS blocks have the same level of complexity. This property was satisfied with all the proposed metrics.

Property 4: (∃P) (∃Q)(P ≡ Q &|P| ≠ |Q|)

There can be two SCSS blocks P and Q whose external features look the same, however, due to different internal structure |P| is not equal to |Q|. This property asserts that two SCSS blocks with the same number of attributes and directives could return different metric values. This property is satisfied by $ABCC_{SCSS}$, SUIL, and CL_{SCSS}. The NF_{SCSS} metric values could change even in the circumstances where the number of nested rules is the same. Therefore, NF_{SCSS} satisfies this property.

Property 5: (∃P) (∃Q) (|P| ≤ |P; Q| & (|Q| ≤ |P; Q|)

This property asserts that if we concatenate two SCSS blocks P and Q, the new metric value must be greater than or equal to the individual rule block. All the proposed metrics return numeric values meaning that they satisfy this property.

Property 6: (∃P) (∃Q) (∃R) (|P| =|Q| and |P; R| ≠ |Q; R|)

This property implies that if two SCSS blocks have same metric value (P and Q), it doesn't necessarily mean that when each of the SCSS blocks is concatenated with similar SCSS block R, the resulting metric values are the same. All the proposed metrics have physical components meaning that they return fixed values. Therefore they don't satisfy this property.

Property 7: If you have two SCSS blocks P and Q which have the same number of attributes in a permuted order, then |P| is not equal to |Q|.

This property implies that the order of similar attributes affects their complexity. Therefore, if two rule blocks have the same number of attributes but differ in the ordering, it's not necessary that they have the same complexity level. In the case where the SCSS blocks length is constant and you only change the permutation of the order of statements then all the proposed metrics will retain the same level of complexity. Therefore all the metrics defined didn't satisfy this property.

Property 8: If P is a renaming of Q, then |P| = |Q|

Where you have two SCSS blocks P and Q differing only in their selector names, then |P| is equal to |Q|. The metric values for all the proposed metrics are either size measures, complexity measures or coupling measures and they all return numeric values. Therefore, all proposed metrics satisfied this property.

Property 9: $(\exists P)\ (\exists Q)\ (|P| + |Q| < (\ |P; Q|)$

This property asserts that there exist two SCSS blocks P and Q, where the complexity metric value of the two SCSS blocks when summed up is less than when the rule blocks are interacting. The interaction between rule blocks and the growth of rule blocks over time adds to the complexity of rule blocks. The growth of blocks complexity happens when new attributes are added or even when a new SCSS block is added to the existing SCSS block, meaning that the new metric value is equal to or greater than the sum of the two original rule blocks. All the metrics $ABCC_{SCSS}$, NF_{SCSS}, SUIL, and CL_{SCSS} satisfied this property.

Table 5: Summary of validation of SCSS metrics with Weyuker's properties

Property	**$ABCC_{SCSS}$**	**NF_{SCSS}**	**SUIL**	**CL_{SCSS}**
1	✓	✓	✓	✓
2	✓	✓	✓	✓
3	✓	✓	✓	✓
4	✓	✓	✓	✓
5	✓	✓	✓	✓
6	×	×	×	×
7	×	×	×	×
8	✓	✓	✓	✓
9	✓	✓	✓	✓

5.2. Practical Evaluation with Kaner Framework

Kaner framework is used to prove the practical utility of the proposed metrics. Therefore, the aim of implementing Kaner framework is to find out if the metrics defined make any sense and to enable the designers to see how the metrics can be used for experimental purposes, thus proving their practicality (Misra and Adewumi, 2018). According to Kaner (2004), the following eleven questions should be addressed for purposes of evaluation of software metrics.

i. *What is the purpose of this measure?*

The purpose of the measure must be clear so as consider it as a valid measure. Therefore, the purpose of this measure is to evaluate the complexity of sassy cascading style sheets (SCSS).

ii. *What is the scope of this measure?*

The measure used should have a specific area it acts on. The proposed metrics will be used by front web developers in web-based projects, particularly those who style the web-documents.

iii. *What attribute are we trying to measure?*

The attribute to measure will be maintainability through its sub-attributes; understandability, modifiability, and testability.

iv. *What is the natural scale of the attribute we are trying to measure?*

The proposed metrics will measure understandability, modifiability, and testability and they can all be measured on an ordinal scale

v. *What is the natural variability of the attribute?*

The quality attributes are subjective in nature, meaning that different SCSS developers can rate the understandability, modifiability, and testability of the same code differently.

vi. *Metrics definition*

The metrics must be clearly defined and in this study, the metrics have been defined in section 3.

vii. *What is the metric and what measuring instrument do we use to perform the measurement?*

There are four proposed metrics; $ABCC_{SCSS}$, NF_{SCSS}, SUIL and CL_{SCSS} and they have been computed manually. In addition, a static metrics tool will be developed to measure the metrics.

viii. *What is the natural scale for this metric?*

The natural scale for all the metrics defined fall in the ratio scale

ix. *What is the natural variability of readings from this instrument?*

When we manually compute the metrics there is no subjectivity to it, meaning that there is no variability. For the metrics tool, the software will be tested to ensure no bugs that would lead to erroneous metric values.

x. *What is the relationship of the attribute to the metric value?*

The maintainability of SCSS is directly related to the proposed complexity metrics. This means we can tell the understandability, modifiability, and testability of SCSS by using the proposed metrics.

xi. *What are the natural and foreseeable side effects of using this instrument?*

Since the static metrics tool will be thoroughly tested, then there will be no negative effects after the implementation of the tool.

6. Discussion

Results based on the three code snippets show that the new metrics are intuitional. The $ABCC_{SCSS}$ metric value for snippet 2 at 1.99 is higher than the metric value for snippet 1 though they have the same number of SCSS blocks. This is reasonable because snippet 2 has more attributes, control directives and extend directives than snippet 1. The NF_{SCSS} metric value for snippet 2 at 12.0 is higher than all others, which makes sense because it has more nested blocks. The SUIL metric value for snippet 2 is the highest at 0.30 snippets, this means that it has many extend directives implemented in relation to the number of selectors in the snippet. The final metric CL_{SCSS} value is highest in snippet 1 at 0.63, this is reasonable because the mixins and variables are more extensively shared in snippet 1, as compared to other snippets.

Results based on the three real-world projects show that the metrics are intuitional, as shown by the different metrics values computed. The metrics values are an indicator of the different levels of complexity of the SCSS code in those projects. For example, in happy-shala.com, the $ABCC_{SCSS}$ metric value is 2.58 and is higher than that of greatjewishmusic.com which is 2.17, but lower than that of mce.ie website value of 2.9. The NF_{SCSS} metric value is highest for greatjewishmusic.com at 8019, followed by happy-shala.com at 6960 and mce.ie reports the lowest value at 3034. The SUIL metric value for the mce.ie website is 0.03 and zero (0) for both the happy-shala.com and greatjewishmusic.com websites. This means that the inheritance feature is implemented only in the mce.ie website. The last metric CL_{SCSS} value is highest in mce.ie at 2.33, followed by happy-shala.com at 0.31 and greatjewishmusic.com reported the lowest value of 0.27, meaning that it has a lot of sharing of mixins and variables.

In the case of validation with Weyukers properties, results show that all the metrics satisfied seven out of nine properties. This makes the measures reasonable though they all didn't satisfy property 6 and 7, and this is because they assign fixed weights to the attributes and the rule directives. In addition, interactions in SCSS don't add any extra external complexity and the permutation of statements don't add any complexity.

Results from Kaner framework show that all the four metrics satisfy its eleven evaluation requirements. This implies that the proposed metrics are useful to practically evaluate SCSS code complexity.

7. Conclusion and Future Work

This paper proposes four metrics for measuring the complexity of SCSS code. The metrics were used to compute the complexities of three code snippets and three real-life world projects. Values obtained from the code snippets and the real-life projects show that the metrics are intuitional. It was established that the more complex files returned higher complexity metric values than the less complex files. For example, while computing metrics from real-world projects, the mce.ie website returned higher $ABCC_{SCSS}$ values than other websites due to the fact that it had higher average block complexity. It was also established that the greatjewishmusic.com website returned

higher NF_{SCSS} values due to the fact that its rule blocks are more nested. Similarly, the mce.ie website returned higher SUIL and CL_{SCSS} values due to the fact that it has implemented the inheritance feature and coupling respectively. High values of each of these metrics imply that it will be difficult to understand, modify and test the code. Front web developers should, therefore, be concerned whenever metrics values tend to go high, as these could affect their design decisions.

Validation results of the proposed metrics using Weyukers properties showed that all the metrics satisfied most of its properties, meaning that all the metrics are theoretically sound. The study further evaluated the proposed metrics with Kaner framework and all metrics proved their practicality from the theoretical point of view. Therefore, the new metrics are structurally good and can be used together to show the full picture of the SCSS complexity.

In the future, empirical validation of the proposed complexity metrics will be carried out using real-world projects. Another future work is to develop a metrics tool for SCSS so as automate the computation of these metrics.

References

Abreu, F. B., Carapuca, R. (1994). Candidate Metrics for Object-Oriented Software within a Taxonomy Framework. *Journal of System Software*, Vol. 26, pp. 87–96.

Abreu, M., Abreu, F. B. (1996). Evaluating the impact of Object-Oriented Design on Software Quality. *Proceedings of 3rd International Software Metrics Symp.* Berlin.

Adewumi, A., Misra, S., Ikhu-Omoregbe, N. (2012). Complexity Metrics for Cascading Style Sheets. In B. Murgante (Ed.), *Lecture Notes in Computer Science* (Vol. 7336, pp. 248-257). Springer.

Baski, D., Misra, S. (2011). Metrics Suite for Maintainability of XML Web-Services. *IET Software*, 5(3), 320-341.

Brilliant, S.S., Knight, J.C. (1999). Empirical research in software engineering, *ACM SIGSOFT Soft. Eng. Notes,* 24, (3), pp. 45–52.

Cardoso, J. (2007). Complexity Analysis of BPEL Web Processes. In: Software Process: Improvement and Practice , Wiley Online Library.

Cederholm, D. (2013). A BOOK APART: Sass for Web Designers. (M. Brown, E. Kissane, J. Bolton, and T. Lee, Eds.) New York, USA: Jeffrey Zeldman.

Chawla, S., Nath, R. (2013). Evaluating Inheritance and Coupling Metrics. *International Journal of Engineering Trends and Technology (IJETT), 4*(7), 2903-2908.

Cherniavsky, J., Smith, C.(1991). On Weyukers Axioms for Software Complexity Measures. *IEEE Transaction on Software Engineering*, Vol. 17, No. 6, pp. 636–638.

Chhilar, U., Bhasin, S. (2011). A new complexity weighted composite complexity measure for object-oriented systems. *International journal of information and communication technology research*, Vol 1, No. 3.

Chidamber, S.R., Kemerer, C.F.(1994). A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pp. 476–493.

Chung, C., Lee, M. (1992). Inheritance based Object-Oriented Software Metrics. *IEEE Region 10 Conference.* Melbourne, Australia.

Fenton, N. E., Pfleeger, S. L. (1997). Software metrics: a rigorous and practical approach. PWS Pub.

Frain, B. (2013). Sass and Compass for designers. Packt Publishing Ltd.

Geneves, P., Layaida, N., Quint, V. (2012, April). On the analysis of cascading style sheets. *In Proceedings of the 21st international conference on World Wide Web* (pp. 809-818). ACM.

Ghosheh, E., Black, S., Qaddour, J. (2008). Design metrics for web application maintainability measurement. *IEEE/ACS International Conference on Computer Systems and Applications* (pp. 778-784). Doha: IEEE.

Gill, N. S., Sikka, S. (2011). Correlating Dimensions of Inheritance Hierarchy with Complexity and Reuse. *International Journal on Computer Science and Engineering (IJCSE), 3*(9), 3250-3253.

Gursaran, G.R. (2001). On the Applicability of Weyuker Property Nine to Object-Oriented Structural Inheritance Complexity Metrics. *IEEE Transaction on Software Engineering*, Vol. 27, No. 4, pp. 361–364.

Halstead, M. H. (1977). Elements of Software Science (Operating and programming systems series). Elsevier Science Inc., New York, NY.

Harrison R., Counsell S., Nithi, R., An Evaluation of the MOOD set of Object-Oriented Software Metrics, *IEEE Transactions on Software Engineering*, vol. 24 no. 6, 1999, p. 491-496

Shao Jingqiu, Wang Yingxu (2003, April). A new measure of software complexity based on cognitive weights. *Can. J. Elect. Comput. Eng*., Vol. 28, No. 2.

Kaner, C. (2004). Software engineering metrics: What do they measure and how do we know?. In *in METRICS 2004. IEEE CS*.

Khan, A. A., Mahmood, A., Amralla, M. S., Mirza, T. H. (2016, January). Comparison of Software Complexity Metrics. *International Journal of Computing and Network Technology, 4*(1), 19-26.

Kushwaha, D. S., Misra, A. K. (2006, September). Improved Cognitive Information Complexity Measure: A Metric that establishes Program Comprehension Effort. *SIGSOFT Software Engineering Notes, 31*(5), 1-7.

Li, E. Y. (1987). A measure of program nesting complexity. National Computer Conference, (pp. 531-538). San Luis Obispo, California.

Li, W., and Henry, S. (1993). Object-oriented metrics that predict maintainability. *The Journal of Systems and Software, 23*(2), 111-122.

Mayer, T., Hall, T. (1999, July). Measuring OO Systems: a critical analysis of the MOOD metrics. In Technology of object-oriented languages and systems, 1999, proceedings of (pp. 108-117). IEEE.

Mazinanian, D., Tsantalis, N (2016, March). An empirical study on the use of CSS preprocessors. *In 2016 IEEE 23rd international conference on Software Analysis, Evolution, and Reengineering (SANER)* (pp. 168-178). IEEE.

McCabe, T. J. (1976, December). A Complexity Measure. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, SE-2(4), 308-320.

Mesbah, A., Mirshokraie, S. (2012, June). Automated analysis of CSS rules to support style maintenance. *In Proceedings of the 34th International Conference on Software Engineering* (pp. 408-418). IEEE Press.

Misra, S., Adewumi, A. (2018). Object-Oriented Cognitive Complexity Measures: An Analysis. *In Intelligent systems: Concepts, Methodologies, Tools, and Applications* (pp. 1324-1347). IGI Global

Misra, S., Cafer, F. (2012, November). Estimating Quality of JavaScript. *The International Arab Journal of Information Technology*, 9(6), 535-543.

Misra, S., Akman, I., Koyuncu, M. (2011, June). An inheritance complexity metric for object-oriented code: A cognitive approach. *Indian Academy of Sciences, 36*(3), 317-337.

Muketha, G. M., Ghani, A. A. A., Selamat, M. H., Atan, R. (2010). Complexity Metrics for Executable Business Processes. *Information Technology Journal*, 9: 1317-1326.

Ogheneovo, E. E. (2014, December). On the Relationship between Software Complexity and Maintenance Costs. *Journal of Computer and Communications*, 2, 1-16.

Parthasarathy, S., Anbazhagan, N. (2006). Analyzing the software quality metrics for object-oriented technology. *Inform. Technol. J*, 5, 1053-1057.

Piwowarski, P. (1982). A Nesting Level Complexity Measure. ACM SIGPLAN Notices. 17(9), 44-50.

Punt, L., Visscher, S., Zaytsev, V. (2016, October). The A? B* A Pattern: Undoing Style in CSS and Refactoring Opportunities it Presents. *In 2016 IEEE International Conference on Software Maintenance and Evolution* (ICSME)(pp. 67-77). IEEE.

Sharma, N., Joshi, P., Joshi, R.K. (2006). Applicability of Weyuker's Property 9 to Object-Oriented Metrics. *IEEE Transaction on Software Engineering*, Vol. 32, No. 3, pp. 209–211.

Stevens, W. P., Myers, G. J., Constantine, L. L. (1974). Structured design. *IBM Systems Journal, 13*(2), 115-139.

Törn, A., Andersson, T., Enholm, K. (1999). A complexity metrics model for software. *South African Computer Journal* 24: 40-48.

Appendix: Code Snippets

Code Snippet 1

```
@mixin Raleway-SemiBold {
    font-family: 'Raleway-SemiBold';
}
@mixin Raleway-Medium {
    font-family: 'Raleway-Medium';
}
@mixin PlayfairDisplay-Regular {
    font-family: 'PlayfairDisplay-Regular';
}
$color1: #f4f4f4;
$color2: #000;

p {
font-size: 5px + (6px * 2);
font-color: $color1;
@include PlayfairDisplay-Regular;
}
span{
    width: 60px;
    height: 45px;
    position: absolute;
    @include Raleway-Medium;
 }
@for $i from 1 through 4 {
.p#{$i} { padding-left : $i * 10px; }
}
@function remy ($pxsize)  {
        @return ($pxsize/16) + rem;
}

h1 {
font-size: remy(32);
```

```
font-color: $color2
}
h2{
@extend p;
font-color: $color2
}
h3 {
@include Raleway-Medium;
}
h4 {
@include Raleway-Medium;
}
h5 {
@include Raleway- SemiBold;
}

@media (min-width: 768px) {
    .modal-dialog {
        position: relative;
        top: 15%;
    }
}
```

Code Snippet 2

```
$colortest: 1;
span{
    width: 60px;
    height: 45px;
}
p {
font-size: 5px + (6px * 2);
color:#ff0000;
@extend span;
@if $colortest >1 {
text-color: blue;
    @if $colortest == 1 {
      text-color: white;
        }
    }
}
@function remy ($pxsize)  {
@return ($pxsize/16) + rem;
}
h1 {
font-size: remy(32);
@extend span;
}
h2{
@extend p;
}
```

```
#country-toggle{
    width: 60px;
    height: 45px;
    span:nth-child(1) {
       top: 41px;
       }
    span:nth-child(2) {
       top: 49px;
       }
   }
     .dropdown-menu{
           li{
           padding: 10px .7em;
                 &:last-child{
       margin:0;
       }
                 }
             }
    p{
      font-size: 5px
       color:#ff0000;
    }
```

Code Snippet 3

```
$color1: #04f5f7;
$color2: #000111;
@mixin Raleway-SemiBold {
   font-family: 'Raleway-SemiBold';
}
.js-offcanvas {
    color: $color1;
    background: $color2;
    ul {
        padding-left: 0;
        margin-bottom: 0;
        li {
            display: block;
            border-bottom: 1px solid
            font-size: 1.6rem;
}
        }
}
#get-in-touch {
   . plst {
        width: 50%;
        margin: 2rem auto;
       @include Raleway-SemiBold;
    }
}
```

```
p{
       font-size: 5px
       color:#ff0000;
    }

h1 {
       font-color:$color2;
       @include Raleway-SemiBold;
}
h2{

       @extend p;
}
```

Received November 17, 2018, revised August 2, 2019, accepted October 14, 2019