Detecting Violations of CSS Code Conventions

Boryana Goncharenko
boryana.goncharenko@gmail.com

August 1, 2015, 33 pages

Supervisor: Vadim Zaytsev
Host organisation: University of Amsterdam

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Contents

Abstract

Code conventions are used to preserve code base consistency and express preference of a particular programming style. Often, code conventions are expressed in natural language and it is a responsibility of the developers to read, understand and apply them. Typically, developers need to ensure that their code complies to a given style guide manually. There are a number of tools that can automatically detect violations of conventions, however, current solutions remain rigid, laborious or with limited scope.

This thesis aims at answering three research questions. First, it evaluates the need for CSS conventions based on whether CSS is still handcrafted. Second, it discovers existing code conventions. Third, it designs a domain-specific language that is capable of expressing existing CSS code conventions. The thesis presents a specification of the designed domain-specific language and an implementation of its interpreter that detects violations automatically.

Chapter 1

Introduction

Code conventions put constraints on how code should be written in the context of a project, organization or programming language. Style guides can comprise conventions that refer to whitespacing, indentation, code layout, preference of syntactic structures, code patterns and antipatterns. They are mainly used to achieve code consistency, which in turn improves the readability, understandability and maintainability of the code [1, 2, 3].

Style guides are often designed in an ad hoc manner. Coding conventions typically live in documents that contain a description of each rule in natural language accompanied by code examples. This is the case with the style guidelines of Mozilla [4], Google [5], GitHub [6], WordPress [7] and Drupal [8]. It is the responsibility of the developers to ensure that their code complies to a given style guide. Typically, they need to read and understand the conventions and then apply them manually. Such an approach introduces a number of issues. First, using natural language can make guidelines incorrect, ambiguous, implicit or too general. Second, the fact that developers apply conventions manually increases the chances of introducing violations involuntarily. There are a number of tools that can automatically detect violations of conventions, however, current solutions are often hard to customize or are limited to one type of violations, e.g. only whitespacing.

The core idea behind the project is to provide a solution that lets developers express an arbitrary set of coding conventions and detect their violations automatically. Writing conventions in an executable form could assist authors in detecting incorrect, ambiguous or inconsistent guidelines. Automatic detection of violations could minimize the effort required by developers to write code that complies to the guidelines. To meet the constraints of a Master's project, the implementation is limited to the domain of Cascading Style Sheets (CSS). The project requires determining the need for CSS code conventions in organizations, collecting and analyzing available style guides, and providing a way to express conventions. Specifically, the project attempts to answer the following set of questions:

Research Question 1 Do developers still maintain plain CSS?

Research Question 2 What code conventions for CSS exist?

Research Question 3 How to express existing CSS code conventions?

The thesis is organized as follows. Chapter 2 provides information about previous studies and defines concepts and terms used throughout the thesis. Chapter 3 presents the research approach used to determine whether CSS is handcrafted and analysis of the gathered results. Chapter 4 contains the research method used to discover existing code conventions and the results of the research. The design and validation of the DSL are presented in Chapter 5. Chapter 6 concludes the thesis.

Chapter 2

Background

2.1 Bunge-Wand-Weber ontology

A **conceptualization** is an abstract, simplified view of the world that is represented for some purpose [9]. It consists of the concepts that are assumed to exist in some area of interest and their relationships [9]. An **ontology** is an explicit specification of a conceptualization [9]. It describes what is fundamental in the totality of what exists and it defines the most general categories to which we need to refer in constructing a description of reality [10].

Researchers distinguish between two kinds of ontologies: top-level and domain- specific [10]. Ontologies of the former type are highly general and provide the theoretical foundations for representation and modeling of systems. Ontologies of the latter type define concepts and their relations only for a particular domain. A domain- specific ontology is based on a specific top-level ontology if it uses the categories defined by the high level ontology [10].

The Bunge-Wand-Weber (BWW) ontology [11] is a high-level ontology used in the representation model developed by Wand and Weber [12]. Table 2.1 presents a selected set of the ontological constructs in the BWW ontology.

Table 2.1: Selected ontological constructs in the BWW representation model

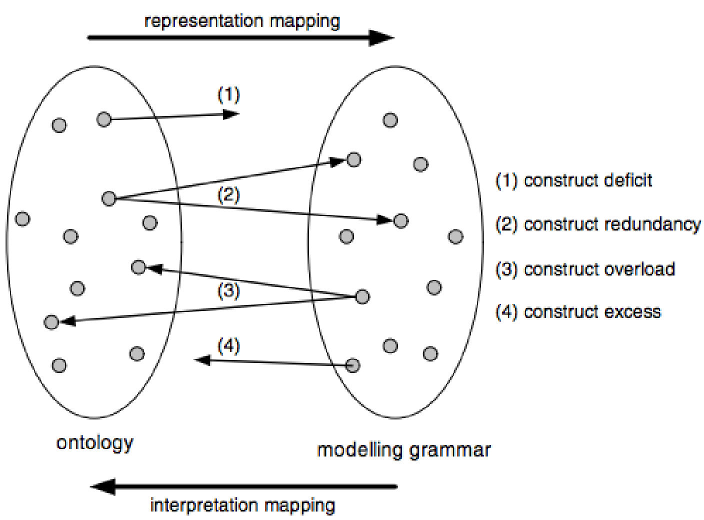
Ontological construct	**Explanation**
Thing	The elementary unit in the BWW ontological model. The real world is made up of things. A composite thing may be made up of other things (composite or primitive). [12]
Properties	Things possess properties. A property is modeled via a function that maps the thing into some value. A property of a composite thing that belongs to a component thing is called an hereditary property. Otherwise it is called an emergent property. A property that is inherently a property of an individual thing is called an intrinsic property. A property that is meaningful only in the context of two or mode things is called a mutual or relational property. [12]
State	A vector of values for all property functions of a thing. [12]
Event	A change of state of a thing. It is effected via a transformation. [12]
Transformation	A mapping from a domain comprising states to a codomain comprising states. [12]
History	The chronologically ordered states that a thing traverses. [13]
Coupling	A thing acts on another thing if its existence affects the history of the other thing. The two things are said to be coupled or interact. [12]

Class	A class is a set of things that can be defined via their possessing a characteristic property. [13]
Subclass	A set of things that can be defined via their possessing the set of properties in a class plus an additional set of properties. [13]
System	A set of things is a system if, for any bi-partitioning of the set, couplings exist among things in the two subsets. [12]
System Composition	The things in the system are its composition. [12]
System Environment	Things that are not in the system but interact with things in the system are called the environment of the system. [12]

2.2 Ontological analysis

Ontological analysis is an established approach for evaluating the quality of software engineering notations [14]. It consists of a two way comparison between a set of modeling grammar constructs and a set of ontological constructs. The **interpretation mapping** compares the notation with the ontology and the **representation mapping** compares the ontology with the notation [15]. The underpinning of ontological analysis is that modeling grammars are incomplete if they are not able to represent what exists in reality [16]. Furthermore, the analysis requires one-to-one mapping between the modeling grammar and the ontological constructs. Any deviation from such correspondence leads to a discrepancy (Figure 2.1).

Figure 2.1: Ontological Analysis [15]



Construct deficit occurs when an ontological construct does not have a corresponding construct in the modeling grammar. **Construct redundancy** is observed when a single ontological construct maps to more than one modeling grammar construct. **Construct overload** appears when a modeling grammar construct corresponds to more than one ontological construct. **Construct excess** emerges when a modeling grammar construct does not map to any ontological construct. [14]

Chapter 3

Evaluating the Need for CSS Code Conventions

3.1 Research Method

Despite the new features added in the second [17] and third [18] versions of CSS, the language has obvious limitations, for example, lack of variables. A number of preprocessors have evolved to tackle the downsides of CSS. Solutions such as SASS [19], LESS [20] and Stylus [21] offer enhanced or even different syntax and translate it to CSS. Preprocessors are not only ubiquitously recommended, but also widely adopted in practice. The presence of such solutions poses the question whether conventions for CSS are required at all. If nowadays CSS is generated and not maintained, the need for CSS conventions is substituted with need for preprocessor conventions.

To determine whether CSS is still handcrafted, all commits to open source repositories hosted on GitHub for the period Jan-Apr 2015 were analyzed. To differentiate between plain CSS and preprocessor code, the extensions of all files in the commits were inspected. In case the commit contains a file with extension `.scss`, `.sass`, `.less` or `.styl`, it is considered preprocessor maintenance. In case the commit contains files with the `.css` extension and no preprocessor extensions, it is considered maintenance of plain CSS. Since the main objective of the search is finding maintained files, only files that have been modified are taken into consideration. Files that have been added are excluded from the results, since developers often add third-party CSS libraries to their repositories.

3.2 Results

To find repositories that contain a commit in the time internal January - April 2015, GitHub's BigData API was used. As a result, a total of 2,331,864 public repositories were found and analyzed. While in the majority of the cases the repositories were analyzed sucessfully, the designed approach failed to retrieve information from 15% of these repositories. Specifically, 253,611 repositories were made private, thus, excluding the possibility to access their commit information. Additionally, 91,114 repositories contain commits of extremely large size that cannot be handled due to memory constraints. As a result, 85.2% of the repositories were analyzes successfully, 10.9% are no longer in the public space, and 3.9% have an extreme size (Figure 3.1).

From all 1,987,139 repositories that were analyzed, 2,261,941 commits that maintain any form of CSS were discovered. 1,328,163 of these commits maintain plain CSS, while the rest of the commits contain at least one preprocessor file. In other words, 58.7% of all CSS-related commits are still maintenance of plain CSS (Figure 3.2).

Figure 3.1: Number of analyzed repositories

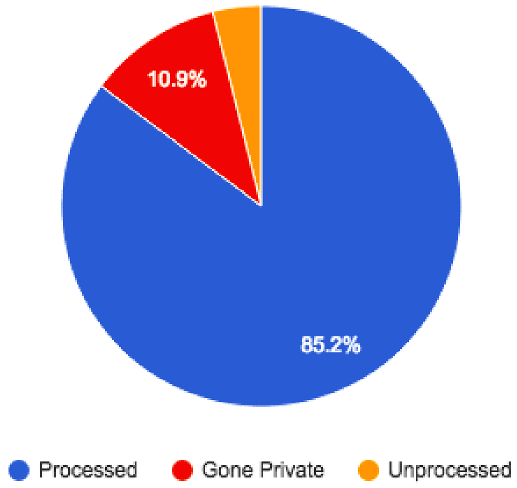
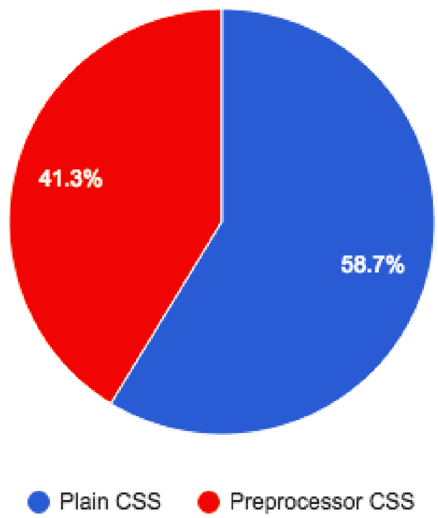


Figure 3.2: Number of analyzed commits



3.3 Analysis

There are a number of limitations that need to be considered before interpreting the results of the conducted research. Firstly, the search is conducted on a single hosting platform - GitHub. That said, currently GitHub reports having over 10 million users and 24 million repositories [6], which makes it the largest code host in the world [22]. Secondly, the search is narrowed to the publicly available repositories. Thus, it lies on the premise that there is not a significant difference between the public and private repositories hosted on the platform. Thirdly, the search detects only the four most popular preprocessor extensions and omits other preprocessors. It is possible that a number of custom preprocessors are used in practice. However, it is assumed that the number of such commits would not increase the number of total CSS commits to the extend at which the percentage of plain CSS commits is significantly diminished.

Having the above limitations in mind, the search provides evidence to conclude that despite the popularity of preprocessors, plain CSS is still handcrafted on GitHub in the beginning of 2015.

Chapter 4

Discovering Existing CSS Code Conventions

4.1 Research Method

The primary organization responsible for the specification of CSS has not published an official CSS style guide. As a result, the CSS community has produced a pool of coding conventions, best practices, guidelines and recommendations.

To discover existing CSS code conventions, two searches with the keywords `CSS code conventions` were made using the search engines `http://duckduckgo.com` and `http://google.com`. The first 100 results of each search were analyzed. From each result only conventions that refer to plain CSS are taken into account and conventions related to preprocessors are ignored. In case the result contains links to other style guides, these references are considered as results and analyzed separately.

While searching for conventions, a number of issues were discovered. First, some of the conventions do not provide sufficient information to be applied in practice. Such an example is the convention *Do not use CSS hacks - try a different approach first* when the style guide does not define the meaning of CSS hacks. Such overgeneralized conventions were omitted from the results.

Another part of the discovered conventions introduce a discrepancy between their description in natural language and the provided code example. An instance of such contradiction is when the convention *Nothing but declarations should be indented* is followed by a code snippet illustrating that rules in media queries should also be indented. In such cases the convention is interpreted as described by the code example.

When conventions are not supported with code examples, their description could remain open for interpretation. For example, *Rules with more than 4 selectors are not allowed* could be seen as forbidding multi-selectors with more than four selectors, or disallowing selectors with more than four simple-selectors. All possible interpretations of ambiguous conventions were registered as separate conventions.

There are conventions that are not explicitly stated, but could only be inferred by the other rules. For example, the convention *You can put long values on multiple lines* states that a long value is allowed to appear on several lines. However, this rule implies the presence of another convention, that is not explicitly stated. It implies that short values should appear on one line and only lengthy values are allowed to appear on multiple. Such implicit conventions were registered as explicit conventions.

4.2 Results

As a result of the searches, 28 CSS style guides were discovered. Sources of these conventions include CSS professionals, open-source communities and companies, such as Google, GitHub, Wordpress, Drupal. Most of the discovered CSS convention sets were parts of bigger style guides and thus they contain a small number of conventions 5-10. Standalone CSS style guides contain around 10-42 conventions. The total number of conventions in the discovered style guides is 471. However, style guides often share the same conventions and even refer to one another. Because of this overlapping, only one third of the 471 conventions are unique.

Thus, the result of the searches is 151 unique code conventions appearing in 28 CSS style guides. Some of the most popular conventions are listed below:

- Put a ; at the end of declarations.
- Do not put quotes in URL declarations.
- Use short hex values.
- Use the shorthand margin property.
- Do not use units after 0 values.
- Use a leading zero for decimal values.
- Avoid qualifying ID and class names with type selectors.
- Use short hexadecimal values.
- When possible, use em instead of pix.
- Avoid using z-indexes when possible.
- Require compatible vendor prefixes.
- Do not use id selectors.
- Id and class names should be lowercase.
- All values except the contents of strings should be lowercase.
- HTML tags should be lowercase.
- Use single quotes in charsets.
- Use single quotes in attribute selectors.
- Put one space between the colon and the value of a declaration.
- Put one space between the last selector and the block.
- One selector per line.
- Forbid empty rules.
- A vendor-prefixed property must be followed by a standard property.
- No trailing spaces.

The full list of all discovered conventions along with their sources and explanation of their meaning is available in CssCoco GitHub repository.

Chapter 5

Expressing CSS Code Conventions

5.1 Analysis of conventions corpus

Code conventions is an umbrella term that comprises rules for whitespacing, comments, indentation, naming, syntax, code patterns, programming style, file organization etc. To gain an overview of the type of conventions used in the CSS domain, all conventions in the corpus are organized in groups depending on the type of constraints they impose. The following three categories were defined (sublists provide examples of conventions that fall in each category):

Layout category contains rules that constrain the overall layout of the code. It includes conventions related to whitespace, indentation and comments. Examples include:

- Use four tabs for indentation.
- Put one blank line between rulesets.
- Disallow spaces at the end of the line.

Syntax Preference category comprises conventions that express preference of a particular syntax. Note that rules in this category do not aim at ensuring CSS validity, but choose between syntactic alternatives. For example, both single and double quote strings are valid in CSS and a convention may narrow down the choice of the developer to single quotes. Examples include:

- Use lowercase for id and class names.
- Require a semicolon at the end of the last declaration.
- Use strings with single quotes.

Programming Style category consists of conventions that put constraints on how CSS constructs are used to achieve a certain goal. They specify preferred code patterns or anti-patterns. Conventions in this group are used mainly to improve maintenance and performance, or to avoid a bug in a particular implementation. Examples are:

- Do not use the universal selector.
- Avoid using !important.
- A vendor-prefixed property must be followed by a standard property.

Conventions in each of the groups were analyzed and their violations were made explicit. While the violations of most of the conventions are obvious, some of them require knowledge about the possible valid CSS syntax. For example, conventions such as *Avoid id selectors* directly describe their violations - id selectors. That said, the convention *Use single quotes in URLs* has two violations: URLs with single quotes and URL without quotes.

After the violations of each convention were made explicit, the specific actions needed to detect these violations were determined. Currently, the detection of violations is performed by developers manually or with the partial help of tools. To perform such checks, developers need to understand different concepts, e.g. the concept of a rule, HTML element, IDs, etc and perform certain actions, such as find a structure, evaluate a constraint etc. The analysis tries to grasp the specific concepts and actions used to find violations. To illustrate the process, the analysis of one convention is included. Analysis of all conventions in the corpus is available at CssCoco GitHub repository.

Convention: Disallow empty rules.

Author: CSS lint

Violations: Presence of rulesets that do not contain declarations. In case at least one declaration is present, the ruleset does not violate the convention. Examples include:

```
.myclass { }             /* violation */
.myclass { /* Comment */ } /* violation */
.myclass { color: green; } /* not violation */
```

Actions: Recognize rulesets and declarations. Determine whether a ruleset does not contain any declarations.

Analysis: The convention aims at getting rid of one type of refactoring leftovers - rulesets without declarations. Removing empty rulesets reduces the total size of CSS that needs to be processed by the browser. One possible approach for discovering violations of the convention at hand is to search the stylesheet for rulesets and then check whether each ruleset contains a declaration. To perform this search successfully, developers need to understand the concept of a ruleset and a declaration, i.e. they need to be able to recognize these two CSS structures. Further, developers need to determine relations between structures, particularly, whether a ruleset contains a declaration.

After all conventions were analyzed, the specific actions and concepts were used to extract requirements and draw conclusions about the needed functionality. First, every convention can be represented as a combination of constraints, regardless of the way it is expressed. There are two major constructs used to convey conventions in natural language: forbid and require. Conventions that forbid describe directly their violations. For example, the convention *Disallow @import* specifies that import statements are violations. Conventions that use the latter construct describe a pattern and once the pattern is found the constraint is evaluated. In case the constraint is not met, a violation is discovered. For example, the convention *class names should be lowercase* requires finding class nodes and then checking whether they are lowercase.

As the three categories of conventions imply, conventions can reference nodes from the abstract syntax tree, concrete syntax tree and parse tree of CSS. For example, the convention *A rule must not contain width and padding declarations* accesses concepts that are present in the abstract syntax tree. Similarly, the rule *Put a semicolon at the end of the last declaration* refers to nodes that are omitted by the abstract syntax tree and are present in the concrete syntax tree. All whitespacing and indentation conventions target nodes that are relevant only in the parse tree of CSS.

Usually, the patterns described in conventions target one of all described nodes. For example, when the pattern refers to rulesets in media queries, the target node is the ruleset and the media query is part of the context. Similarly, when a pattern describes a ruleset that contains a float declaration, the target node is the ruleset and the float declaration is only a constraint. Certain conventions can have more than one target nodes, e.g. when two declarations is a ruleset need to be compared.

Conventions refer to nodes using their type or function in the CSS program. For example, in the snippet `[class=‘‘test’’]` the node test can be selected 1) because it is of type string and 2) because it is an attribute value. Similarly, a node with value `#ffffff` may be selected because it is a hexadecimal value or because it represents a color.

Conventions may use CSS-specific knowledge. For example, the rule *Use lowercase for properties; vendor-prefixed properties are exception* requires differentiating between standard and vendor-specific properties. While in this case the two types of properties can be easily distinguished, some conventions require information that cannot be obtained using the CSS code. Consider the convention *Order vendor-prefixed values by their version; newer versions of vendor values should appear after old ones.* To detect violations for this convention the release dates of the properties need to be available for comparison.

Conventions rarely target a single node. Typically, they refer to a number of nodes organized in a pattern. For example, the convention *A ruleset must not contain display and float declarations* requires searching for two specific declaration that appear under the same parent node. The nodes in the pattern do not have to be immediate relatives. In fact, they can be scattered across the tree. For example, the rule *Do not use more than 5 @font- face declarations* requires searching for specific nodes that appear anywhere in the tree.

5.2 CssCoco DSL

5.2.1 Syntax overview

To express the conventions in the corpus, the domain-specific language CssCoco is proposed. It is a declarative language that has two main constructs: conventions and contexts. Conventions express the specific rules that has to be enforced on the code and contexts describe the CSS nodes that need to be ignored while searching for violations.

The language constructs that define conventions try to resemble the way conventions are expressed in natural text. There is a construct that describes directly what is disallowed. For example, the convention *Do not use import statements* is expressed in the following way:

```
forbid import
message 'Do not use import statements.'
```

The keyword *forbid* is followed by a description of the node that is disallowed. In the current convention, we only need to state its type - import. Each convention requires a message clause. The string after the *message* keyword will be displayed to the user, when a violation is found.

As domain analysis indicates, conventions are often expressed as a pattern that, if found, needs to meet given constraints. In CssCoco syntax such conventions are defined using the `find ... require ...` construct. For example, the convention *All class names should be lowercase* is described as follows:

```
find c=class
require c.name match lowercase
message 'All id and class names should be lowercase'
```

The find clause in the above rule specifies the pattern that needs to be found and the require clause states the constraint that should be applied to the discovered nodes. Note that to refer to a matched node in the require clause, the node should be assigned as identifier, `c` in the above example.

Conventions can put more constraints on a node description in the pattern. For example, the convention *Use a leading zero for decimal values* requires finding all nodes of type number that have a numeric value in the interval [-1, 1]. Such constraints are expressed as curly brackets immediately after the type of the node.

```
find n=number{num-value < 1 and num-value > -1}
require n.string match '^0.*'
message 'Use a leading zero for decimal values'
```

Conventions can describe patterns that consist of more than one node. For example, the rule *Use single quotes in charsets* can be expressed in the following way:

```
find s=string in charset
require s.has-single-quotes
message 'Use single quotes in charsets'
```

The pattern contains the description of two nodes: first, a node that is of type string and a node of type charset. The `in` keyword description of the pattern indicates that the string node is nested in the charset node. In this way, the pattern will match only the strings that appear in a charset.

In CssCoco conventions are grouped in contexts that specify what nodes should be ignored when searching for patterns. For example, often when rules refer to newlines they completely disregard indentation. The convention *Every declaration must be on a new line* requires a newline to be present immediately before the declaration. However, when declarations are indented their immediate previous sibling is an indentation node. To handle such cases, the language uses contexts that explicitly describe the ignored nodes.

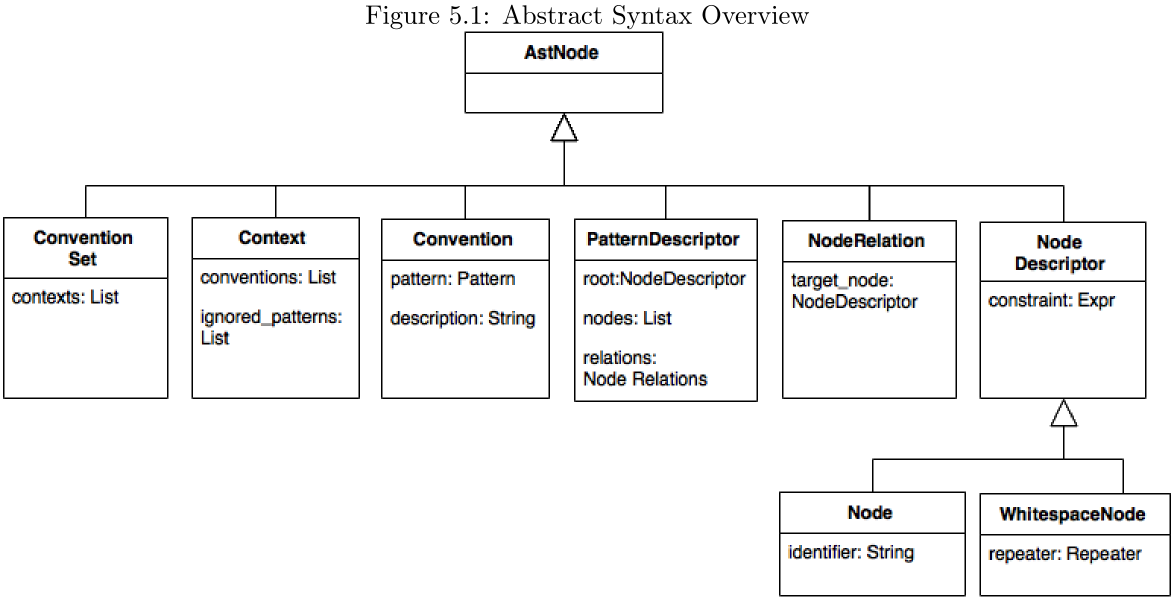
```
Whitespace
ignore indent
{
    find d=declaration
    require newline before d
    message 'Put every declaration on a new line'
}
```

Contexts have a user-defined name and an optional ignore clause, The `ignore` keyword is followed by a description of the nodes that need to disregarded.

5.2.2 Abstract syntax

This section describes the abstract syntax of the designed domain-specific language. An overview of the abstract syntax is presented in Figure 5.1, followed by detailed views of each of the subclasses.

Figure 5.1: Abstract Syntax Overview



ConventionSet represents a style guide. It comprises a number of conventions that form coherent guidelines. Attribute `contexts` is a list of Contexts that contain conventions.

Context represents a group of conventions that belong to the same semantic group (e.g. whitespacing, syntax preference, programming style). Attribute `conventions` is a list of Contexts that contain conventions. Attribute `ignored_patterns` is a list of Patterns that are ignored while searching for the target pattern. For example, while searching for violations of semantic conventions, the whitespacing and indentation nodes are ignored.

Convention represents a rule that enforces specific constraints. Attribute `pattern` is the pattern that the convention targets. Attribute `description` is the description of the convention in natural text. This description is displayed to the user when a violation of the convention is discovered.

PatternDescriptor represents a description of a node or a combination of related nodes that given convention constraints. Attribute `root` is the top node described in the pattern. Attribute `nodes` is a collection of all nodes described by the pattern. Attribute `relations` is a collection of relationships between the nodes used in the pattern.

NodeDescriptor is an abstract class that contains a description of a Css Node. Attribute `constraint` is an expression that designates constraints applied to the node. Attribute `identifier` is a given string that can be used as a reference to the matched node.

Node represents a description of a node used in a PatternDescriptor. Attribute `constraint` is an expression that designates constraints applied to the node. Attribute `identifier` is a given string that can be used as a reference to the matched node.

WhitespaceNode represents a description of a whitespace node that references space, newline, indentation symbols. Attribute `constraint` is an expression that designates constraints applied to the node. Attribute `repeater` is an optional constraint that specific the number of times a whitespace node can appear consecutively. Repeaters are useful to express conventions that do not specify exact quantities of whitespace symbols. For example, the convention "put at least one blank line between rules" sets a lower limit of the number of blank lines, but not an upper limit.

NodeRelation represents a relation between two Nodes. Attribute `target_node` designates a description of the Node targeted by the relation.

A detailed view of the expressions in the abstract syntax of CssCoco is presented in Figure 5.2. Following is a description of the subclasses of Expression:

LiteralExpr represents an expression containing a literal value. Attribute `value` is the value of the literal expression.

VariableExpr represents a reference to a matched node. Attribute `name` is the identifier used to reference the node.

UnaryExpr represents expressions with a single operand. Attribute `operand` is operand of the expression.

NotExpr represents logical negation expression.

UnaryMinusExpr represents unary minus expression.

BinaryExpr represents expressions with a two operands.

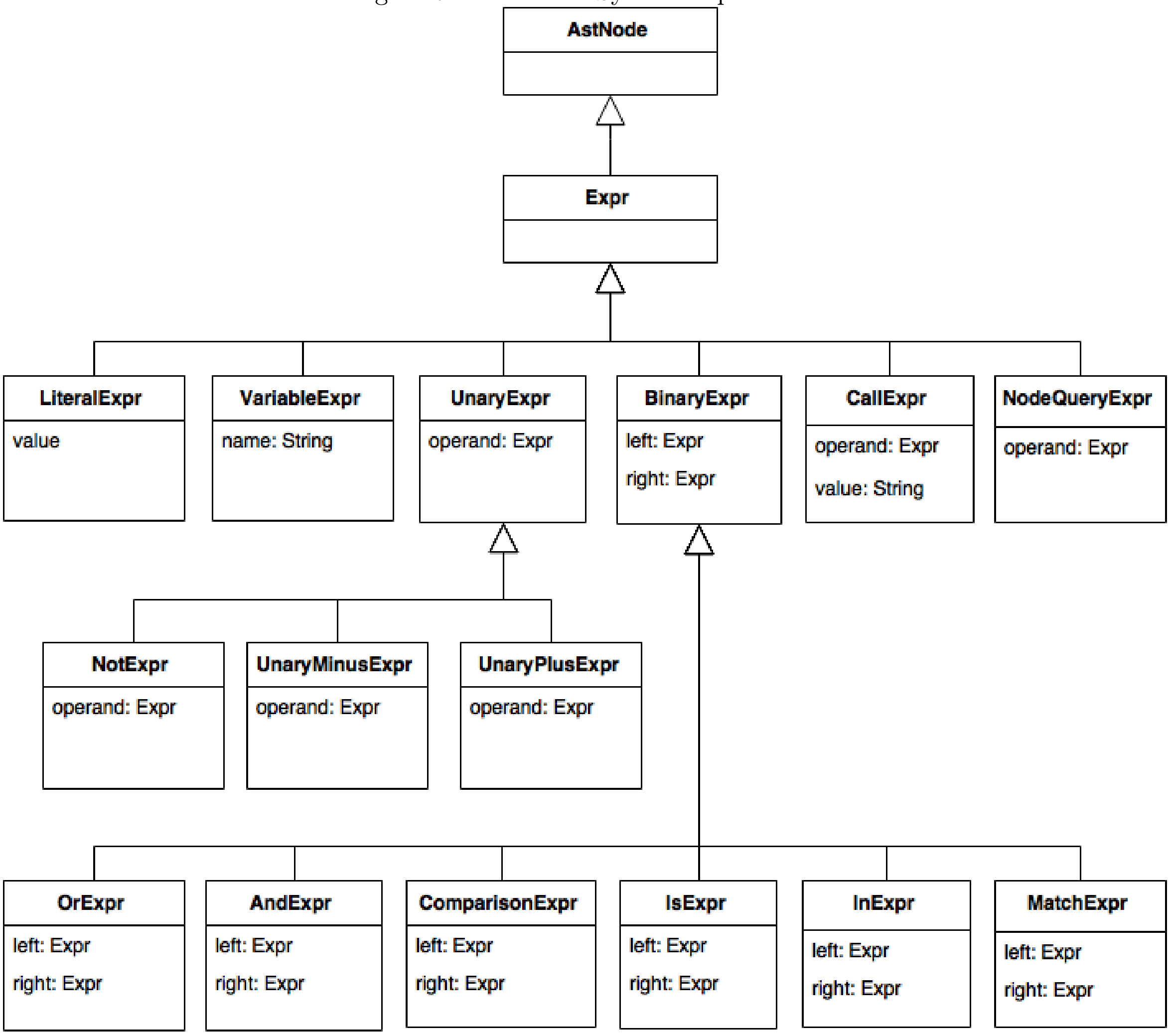
Attributes `left` and `right` represent the first and second operands, respectively.

OrExpr represents logical disjunction expression.

AndExpr represents logical conjunction expression.

ComparisonExpr represents expression that compares two operands.

Figure 5.2: Abstract Syntax Expressions



IsExpr represents expression that checks whether the first operand is of the given type, specified by the second operand.

InExpr represents expression that checks whether the first operand is present in a list of values, specified by the second operand.

MatchExpr represents expression that checks whether the first operand matches a regular expression, specified by the second operand.

CallExpr represents expression that invokes a API property or method of the operand. Attribute `operand` is the operand of the expression. Attribute `value` is the name of the API property or method that is invoked.

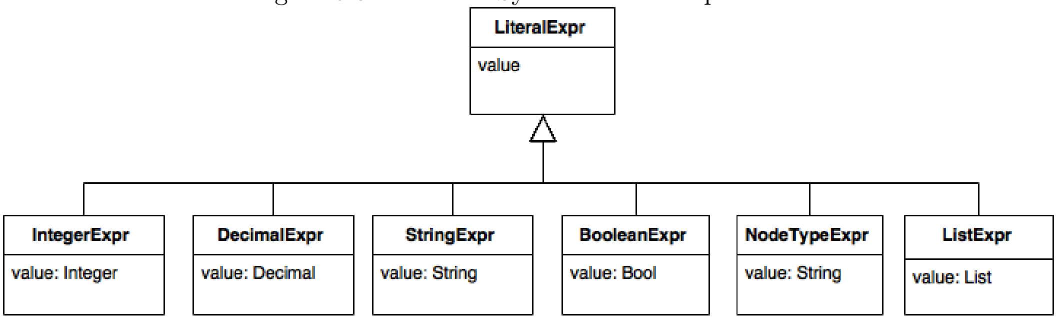
NodeQueryExpr represents expression that queries node context. Attribute `operand` is the node used as a reference point for the query.

Figure 5.3 presents a detailed view of the literal expressions used in the abstract syntax of CssCoco. Following is a listing of the classes.

IntegerExpr represents expression containing a integer value.

DecimalExpr represents expression containing a decimal value.

Figure 5.3: Abstract Syntax Literal Expressions



StringExpr represents expression containing a string value.

BooleanExpr represents expression containing a boolean value.

ListExpr represents expression containing a list value. The elements of the list are of type LiteralExpr.

NodeTypeExpr represents expression containing a string value that describes node type.

Figure 5.4: Abstract Syntax Literal Expressions

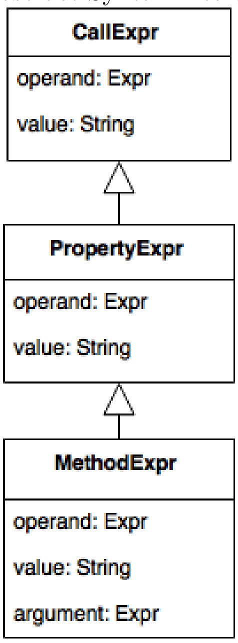


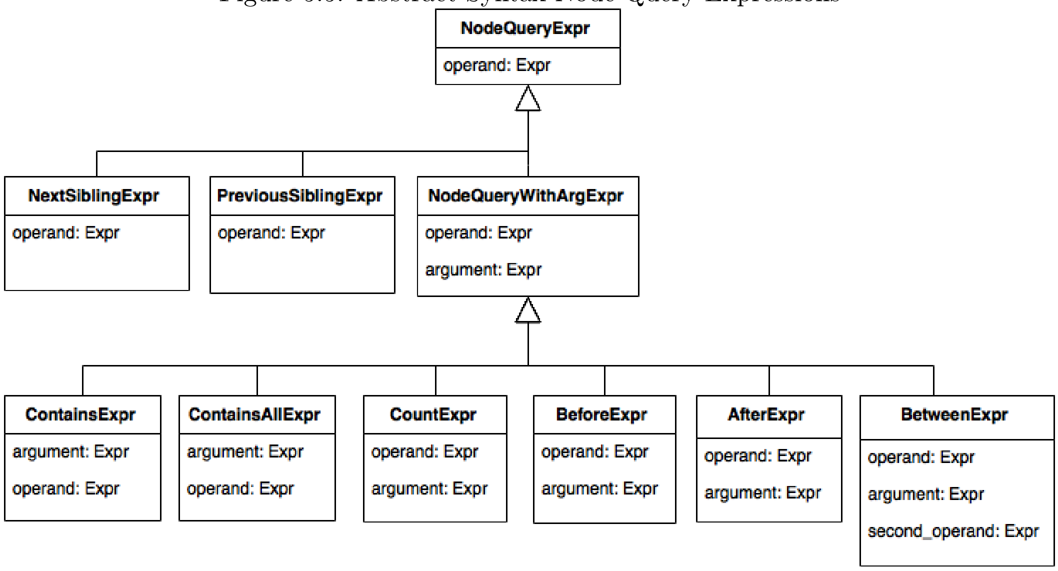
Figure 5.4 presents a detailed view of the call expressions used in the abstract syntax of CssCoco. Following is a listing of the classes.

PropertyExpr represents an expression that returns the value of a property of the operand node. Attribute `operand` represents the node targeted by the expression. Attribute `value` holds the name of the property that is accessed.

MethodExpr represents an expression that returns invokes a method of the operand node. Attribute `argument` represents argument passed to the invoked method.

Figure 5.5 presents an overview of the Node Query Expressions. The following listing describes the subclasses in details:

Figure 5.5: Abstract Syntax Node Query Expressions



NextSiblingExpr represents expression that returns the following sibling of the operand node.

PreviousSiblingExpr represents expression that returns the previous sibling of the operand node.

NodeQueryWithArgExpr represents expression that queries node context and uses additional constraints for the query. Attribute `argument` represents the additional constraints used by the query.

ContainsExpr represents an expression that checks whether the operand node contains a node that matches given constraints.

ContainsAllExpr represents an expression that checks whether the operand node contains nodes that match given constraints.

CountExpr represents an expression that counts the number of ancestor nodes of the operand that match a given constraint.

BeforeExpr represents an expression that checks whether a given whitespace variation appears before the operand node.

AfterExpr represents an expression that checks whether a given whitespace variation appears after the operand node.

BetweenExpr represents an expression that checks whether a given whitespace variation appears between the two operand nodes.

5.2.3 Concrete syntax

This section contains the concrete syntax of the designed DSL. Below are presented the grammar rules accompanied by the mapping to the abstract syntax of the language.

stylesheet represents a style guide.

Abstract Syntax Mapping: ast.ConventionSet.

```
stylesheet : context* ;
```

context represents a group of logically related conventions. A single style guide can comprise a number of conventions that enforce various constraints, e.g. whitespacing, syntax preference, program style. Contexts group conventions that ignore the same nodes while searching for their violations.

Abstract Syntax Mapping: ast.Context.

```
context : Identifier ignore_clause? '{' convention* '}' ;
ignore_clause : 'ignore' (node_descriptor)+ (',' (node_descriptor)+)* ;
```

convention represents a single rule in the style guide. Conventions are typically expressed by directly stating what is disallowed or describing a condition that if met, requires additional constraints. The former way of expressing conventions are represented by the `forbid` conventions. The latter approach uses the structure `find ... require ...`. To break down complex disallowing conventions, the structure `find ... forbid ...` has been introduced. This aims at improving readability of conventions. Additionally, the find conventions have a where clause which applies constraints for matching nodes. It is used to expression matching constraints that span over multiple nodes and therefore cannot be present in any of the node descriptors.

Abstract Syntax Mapping: ast.Convention.

```
convention : 'forbid' pattern 'message' String
           | 'find' pattern ('where' logic_expr)? ('require'|'forbid') logic_expr 'message' String
           ;
```

pattern represents a pattern of nodes and their relations. For example, it can describe a horizontal sequence of sibling nodes, a vertical pattern of nested nodes, or pairs of elements with a common parent.

Abstract Syntax Mapping: ast.PatternDescriptor.

```
pattern : node_declaration (('in'|'next-to') node_declaration)*
        | fork ('in' node_declaration)*
        ;
fork : '(' node_declaration (',' node_declaration)+ ')' ;
node_declaration : (Identifier '=')? semantic_node ;
```

node_descriptor represents a description of a node. It describes the type of the node and its additional constraints.

Abstract Syntax Mapping: ast.NodeDescriptor.

```
node_descriptor : 'unique'? node_type ('{' (logic_expr|repeater) '}')? ;
repeater : Integer ',' Integer? | (',')? Integer ;
```

logic_expr represents expressions that perform logic operations and glue arithmetic and type expressions.

Abstract Syntax Mapping: ast.NotExpr, ast.AndExpr, ast.OrExpr and all arithmetic_expression and type_expression mappings.

```
logic_expr : '(' logic_expr ')'
           | 'not' logic_expr
           | logic_expr 'and' logic_expr
           | logic_expr 'or' logic_expr
           | type_expr
           | arithmetic_expr
           ;
```

type_expr represents expressions that ensure node type and perform node queries on nodes. They are located in a separate parser rule because they interpret Identifiers as node type expressions instead of a API calls.

Abstract Syntax Mapping: ast.IsExpr, ast.BeforeExpr, ast.AfterExpr, ast.BetweenExpr.

```
type_expr : arithmetic_expr operator='is' Identifier
          | node_descriptor+ ('before' | 'after') type_operand
          | node_descriptor+ 'between' type_operand 'and' type_operand
          ;
type_operand : Identifier | semantic_node ;
```

arithmetic_expr represents arithmetic, comparison, set membership and regex expressions. These are located in a separate parser rule because they interpret identifiers as API calls instead of node type expressions.

Abstract Syntax Mapping: ast.UnaryMinus, ast.UnaryPlus, ast.LessThan, ast.LessThanOrEq, ast.GreaterThan, ast.GreaterThanOrEq, ast.Equal, ast.NotEqual, ast.InExpr, ast.MatchExpr, ast.LiteralExpr.

```
arithmetic_expr : ('-'|'+') arithmetic_expr
                | arithmetic_expr ('<'|'>'|'<='|'>='|'=='|'!=') arithmetic_expr
                | arithmetic_expr ('in'|'not in'|'match'|'not match') arithmetic_expr
                | call_expression
                | element
                ;
element : Boolean | Decimal | Integer | String | list_ ;
```

call_expression represents an API call expression and also node query expression.

Abstract Syntax Mapping: ast.CallExpr and ast.NodeQueryExpr.

```
call_expression : call_expression '.' call_expression
                | Identifier ('(' (element | semantic_node ) ')')?
                ;
```

Boolean: represents Boolean literal expression.

Abstract Syntax Mapping: ast.BooleanExpr.

```
Boolean : 'true' | 'True' | 'false' | 'False' ;
```

String: represents String literal expression.

Abstract Syntax Mapping: ast.StringExpr.

```
String : "'" (EscapeSequence | ~['])*? "'" ;
EscapeSequence : "\\" "'" ;
```

Integer: represents Integer literal expression.

Abstract Syntax Mapping: ast.IntegerExpr.

```
Integer : (ZeroDigit | NonZeroDigit Digit*) ;
Digit : ZeroDigit | NonZeroDigit ;
NonZeroDigit : [1-9] ;
ZeroDigit : [0] ;
```

Decimal: represents Decimal literal expression.

Abstract Syntax Mapping: ast.DecimalExpr.

```
Decimal : ( NonZeroDigit Digit* | ZeroDigit? ) '.' Digit+ ;
Digit : ZeroDigit | NonZeroDigit ;
NonZeroDigit : [1-9] ;
ZeroDigit : [0] ;
```

list and **list_element** represent the List literal expression.

Abstract Syntax Mapping: ast.ListExpr.

```
list_ : '[' list_element (',' list_element)* ']' ;
list_element : Integer | Decimal | String | semantic_node ;
Letter : [a-zA-Z] ;
Identifier : (Letter)(Letter|Digit|'_'|'-')* ;
```

type_expression represents the NodeType literal expression.

Abstract Syntax Mapping: ast.NodeType.

```
node_type : '(' node_type ')'
          | 'not' node_type
          | node_type 'and' node_type
          | node_type 'or' node_type
          | Identifier
          ;
```

5.2.4 Proof of Concept

To study the feasibility of the designed language, a proof of concept was developed. The implemented solution consists of two parts: a standalone Python package and a plug-in for Sublime Text editor.

The first part of the designed proof of concept comprises the CssCoco interpreter. The implementation is done in Python and currently contains 13 000 lines of code. The source code is available at https://github.com/boryanagoncharenko/CssCoco. The solution was also added to the Python Package Index (Pypi) repository and for less than month it has accumulated over 3000 downloads. Once installed, the package offers a `csscoco` command that takes as parameters a css and a coco file and returns a list of the discovered violations.

The current implementation of the CssCoco interpreter is available only for Python 3.4. Additionally, the proof of concept requires nodejs. This dependency is added because, the only CSS parser that produces CSS parse trees with the required level of details is implemented in nodejs.

The second part of the proof of concept brings the functionality implemented in the Python package to Sublime Text editor. The plug-in uses the `csscoco` command to find violations in CSS files that are being edited in the text editor. Currently, the plug-in is implemented for Sublime Text 3. The source code of the solution is publicly available at https://github.com/boryanagoncharenko/Sublime-CssCoco.

The plug-in offers a command that find and visualizes violations and also. Similarly to other linter tools, rows that contain violations are marked with a color border and a gist appears at the side bar. When the cursor is positioned on a line that contains a violation, the error message is displayed in the status bar. For example, on Figure 5.6 the cursor is placed on line 26 and the status bar indicates that there should be one space between the colon and the value of the declaration.

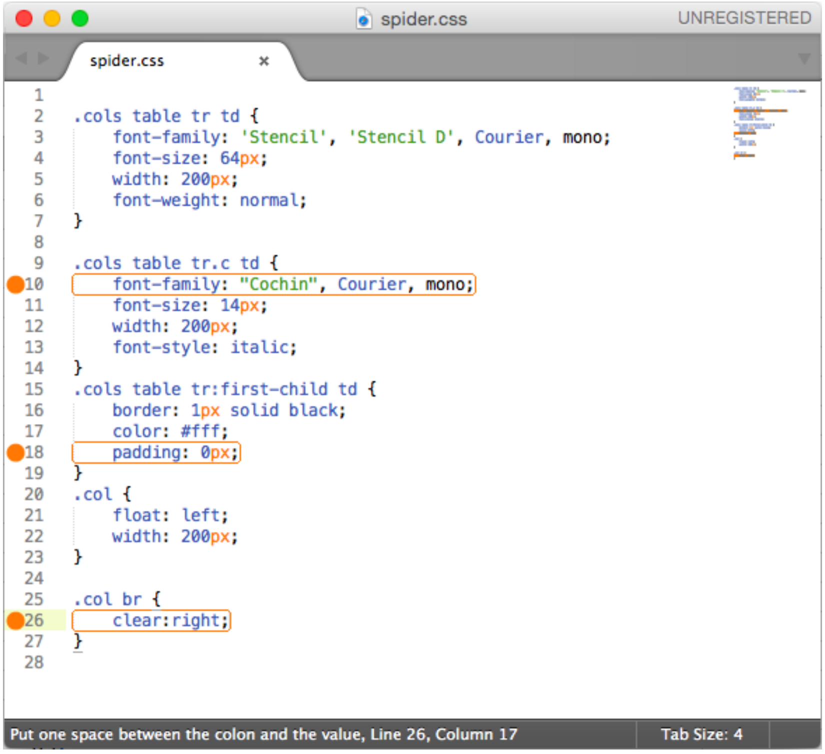
The proof of concept leaves some of the features of the language not implemented. Specifically, is does not include the following:

- Unique construct. A small portion of the conventions require the unique construct. The feature is not included in the current version and is left for future implementation.
- Ordering rules. To check for violations of conventions that specify ordering, requires efficient implementation especially for large CSS files. The feature is currently not supported because of time constraints.
- Indentation rules. Conventions that refer to indentation specify relative indentation. For example, when a convention states that the contents of a ruleset should be indented, this implies that they should be indented once compared to the beginning of the ruleset. Detecting violations of such conventions is laborious and is left for future implementation.

5.3 Validation

The method chosen for validating the designed domain-specific language is ontological analysis, since it is a widely accepted way for evaluating software notations [23, 16, 14, 24, 13]. The particular

Figure 5.6: CssCoco Sublime Text Plug-in



approach used for conducting ontological analysis consists of several steps. First, a domain- specific ontology is designed. Second, the ontology is used as a reference point for the interpretation and representation mappings. Third, emerged anomalies are analyzed and conclusion about the quality of the notation is made.

5.3.1 Ontology design

The first stage of validation requires designing a domain-specific ontology. The specific domain of the developed ontology is limited to detecting violations of CSS code conventions. In other words, the designed ontology tries to capture only the concepts and their relations, that exist when an agent searches a CSS program for violations of given set of code conventions.

The designed domain-specific ontology is based on the BWW top-level ontology [11], i.e. it uses the high-level categories of the BWW ontology to describe the objects, concepts and entities in the specific domain. The rationale behind the decision to use BWW ontology is that it has been the leading ontology used for ontological analysis [14]. The main ontological constructs used in the BWW ontology are listed in section 2.1.

The designed ontology is presented using several approaches. As recommended by Wand and Weber, the ontology is presented using a dictionary comprising definitions of entities in natural text and, second, using Backus-Naur Form (BNF) notation [12, 25]. Additionally, a system diagram is included to provide a better view of the couplings between the different entities. The ontology is intentionally not presented using Unified Modeling Language or Entity-Relationship diagrams. These modeling languages are subjects of ontological analysis themselves and therefore are not suitable for expressing an ontology.

Following is a list with the main concepts discovered in the domain along with their descriptions. The used BWW concepts are written in *italics* and the domain-specific concepts are written in **bold**.

Class **Style Guide** describes the coding practices adopted in the context of a single project, organization, community or language. An individual Style Guide is a *composite thing* built of

Conventions and their relations. Conventions in a Style Guide are interpreted together to form a coherent set of guidelines.

Property **Conventions** refers to the conventions contained in the Style Guide.

Class **Convention** is a specific rule that imposes constraints on the CSS program. It is the building block of Style Guides. An individual Convention is a *composite thing* that contains a Context.

Intrinsic Property **Description** contains the reasoning behind the Convention.

Hereditary Property **Ignored Constructs** denotes the description of constructs that should be ignored while searching for the Convention's Context. It is inherited by the Context thing that builds a Convention.

Class **Context** is a description of a Pattern that the Convention forbids. An individual Context is a *composite thing* that comprises a number of logically related Constraints. When a Pattern in the current Stylesheet fulfills all Constraints in the Context, a Violation is discovered.

Property **Ignored Constructs** are descriptions of Patterns that need to be disregarded while searching for the current Context. In fact, the property denotes a collection of Contexts.

Class **Constraint** is a specific restriction that needs to be fulfilled. They are used in a **Context** to build a description of a **Pattern**. Constraints are individual requirements that are imposed on subjects. Based on the value of the requirement, there are different types of Constraints represented below as *subclasses*.

Property **Subject** indicates the thing is being constrained.

Property **Requirement** denotes the particular limitation applied to the Subject.

Subclass **Existence Constraint** is a type of Constraint that requires existence of the subject.

Property **Subject** indicates the thing is being constrained.

Property **Requirement** denotes the particular limitation applied to the Subject. Specifically, that the Subject must exist.

Subclass **Comparison Constraint** is a type of Constraint that compares the subject to another value.

Property **Subject** indicates the thing is being constrained.

Property **Requirement** denotes the particular limitation applied to the Subject. Specifically, that the Subject must be related to the Value in a given way.

Property **Value** denotes the value that is used for the comparison.

Subclass **Type Constraint** is a type of Constraint that checks whether the subject is of a given type.

Property **Subject** indicates the thing is being constrained.

Property **Requirement** denotes the particular limitation applied to the Subject. Specifically, that the Subject must be of the given type.

Property **Value** denotes the type that the subject should meet to satisfy the constraint.

Subclass **Textual Form Constraint** is a type of Constraint that imposes restrictions on the textual representation of the subject.

Property **Subject** indicates the thing is being constrained.

Property **Requirement** denotes the particular limitation applied to the Subject. Specifically, that the Subject must be equal to the given Value.

Property **Value** denotes the textual form that the Subject should meet for the constraint to be satisfied.

Subclass **Set Membership Constraint** is a type of Constraint that requires the subject to be a member of a set.

Property **Subject** indicates the thing is being constrained.

Property **Requirement** denotes the particular limitation applied to the Subject. Specifically, that the Subject must be a member of the Value.

Property **Value** denotes the set that the subject should be present at for the constraint to be satisfied.

Class **Literal Value** is a thing that represents a constant value. It includes numbers, strings, boolean values etc.

Property **Value** denotes the specific value possessed by the literal.

Class **Violation Log** is the final product of a violations search. An individual Violation Log is a composite thing that contains Violations.

Property **Number of Violations** indicates the size of the Violation Log.

Class **Violation** A Violation occurs when a Pattern that matches the Context of a Convention is found.

Property **Description** explains in natural text what causes the Violation. Typically, the Description is extracted from the Convention that the Violation breaks.

Property **Position in Stylesheet** indicates the location of the Pattern that violates the Convention in the Stylesheet.

Class **Stylesheet** is the CSS code that needs to be checked for compliance with the Style Guide. An instance of Stylesheet is a composite thing that comprises a number of Constructs.

Property **Checked** indicates whether a Stylesheet has been checked for compliance to a given Style Guide.

Class **Construct** is a part of the Stylesheet. It can refer to nodes in the CSS abstract syntax tree, concrete syntax tree and parse tree. Examples include whitespacing, indentation, comments, colons, delimiters, rulesets, declarations, etc.

Property **Property** encapsulates properties of nodes specific to the CSS domain. For example, the type and the string representation of the node are its properties. Similarly, specific CSS Nodes can expose properties that are tightly coupled to the CSS domain, such as release date or vendor name of a CSS property.

Class **Pattern** is a particular part of the CSS program that matches the description of a Context. An instance of a Pattern is a composite thing built from one or many Constructs and Relations between them.

Property **Constructs** denotes the constructs that are contained in the Pattern.

Event **Search for Violations in Stylesheet** occurs when the developer completes the search for violations in a Stylesheet, a Violation Log is created and the state. When the search is completed, the Stylesheet is considered checked for compliance to the Style Guide.

New State **Violation Log** { Violations = value }

New State **Stylesheet** { Checked = True }

Event **Context (Convention) Discovered** occurs when the Context of a convention is discovered, a Violation is recorded in the Violation Log. The state of the Violation contain its description and position in Stylesheet.

New State **Violation** { Description = value, Position in Stylesheet = value }

Event **Stylesheet modified** occurs when the Constructs in the Stylesheet are modified. The state of the Stylesheet is changed to unchecked for compliance.

New State **Stylesheet** { Checked = False }

Event **Style Guide modified** occurs when any of the parts of a Style Guide are modified. This event changes the state of the Stylesheet to unchecked for compliance.

New State **Stylesheet** { Checked = False }

Most of the definitions in the ontology refer to simple concepts that appear in the code conventions domain. For example the concept of a Style Guide refers to a collection of coherent conventions. A Style Guide on its own does not have any emergent or intrinsic properties and, thus, it is defined through the conventions it comprises. Note that in reality a Style Guide may contain a number of intrinsic properties, for example a Style Guide may have an author and contributors. Such properties, however, are not considered part of the specific domain, and thus lie outside the scope of the ontology.

A Convention is defined as the building block of a Style Guide. However, the ontological concept of a Convention is slightly different than what is used as a convention in the domain analysis section. For example, domain analysis indicates that there are two types of conventions: forbid and require. The former type of conventions describes directly their violations. The latter type describes a pattern that, when found, additional requirements must be met and in case they are not met, a violation is discovered. For example, the violation *Do not use id selectors* tells that id selectors are violations. The convention *Strings should be with single quotes* states that strings must have single quotes, and a violation is discovered if the found string does not. Convention types are not related to the meaning of conventions but to the way they are expressed. A convention with the same meaning could be expressed using both structures: *Strings should be with single quotes* and *Forbid strings with double quotes*. Since ontological concepts are concerned with the meaning of things and have to be independent of the language used to express them, the ontology does not possess subclasses of Convention.

A Context is a description of things that are disallowed by a Convention, i.e. it states explicitly what Patterns violate a Convention. In this sense, the meaning of a Convention is always expressed through the possible violations of that Convention. A Context is built from a number of Constraints. While a Context aims at describing a whole violation pattern, a Constraint refers to a single specific requirement. A Constraint always has a subject and a requirement. The subject is the object that is being constraint and the requirement refers to the specific limitation that is applied to the subject. Based on the value of the requirement property, there are different types of Constraints. For example, a Comparison Constraint requires the subject to be equal or greater than a given value and the Type Constraint require the subject to be of a particular type. The Existence Constraint requires the subject to exist.

Since Conventions are used to constraint CSS code, subjects are typically Constructs or their properties. A Construct denotes a concrete part of the Stylesheet, e.g. newline, semicolon, declaration, ruleset etc. Each Construct exposes a number of properties that are tightly coupled to the function of the Construct in the stylesheet. For example, a property construct has properties that indicate whether it is vendor-specific and its release date.

Patterns are composed of Constructs. The Constructs in the Pattern do not need to form a coherent valid Stylesheet and they do not have to be adjacent or directly related. In fact, they could be scattered across the whole Stylesheet. Patterns denote the concrete Constructs that match the description provided by the Context of a Convention. In this sense, they are the specific instances of Violations.

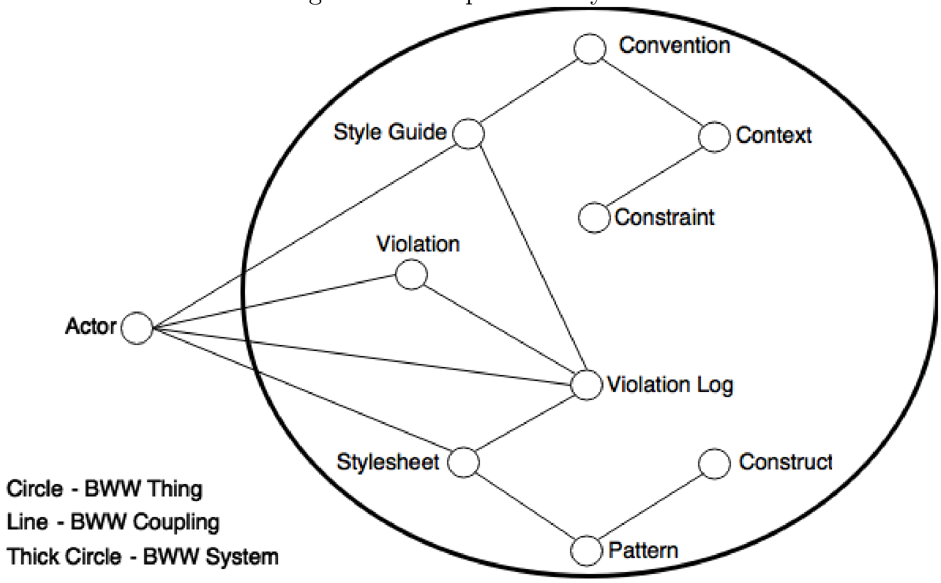
Definitions of the ontological concepts in the listing above often state that an instance of a class is a composite thing that consists of other things. To provide a better understanding of the way composite things are constructed, the same concepts are also expressed using BNF notation:

```
STYLE_GUIDE    ::= CONVENTION+
CONVETION      ::= CONTEXT
CONTEXT        ::= CONSTRAINT+
CONSTRAINT     ::= EXISTENCE_CONSTRAINT | COMPARISON_CONSTRAINT | TYPE_CONSTRAINT
                 | TEXT_CONSTRAINT | SET_CONSTRAINT
VIOLATION_LOG  ::= VIOLATIONS*
STYLESHEET     ::= CONSTRUCT+
PATTERN        ::= CONSTRUCT+
```

The grammar above illustrates that a Style Guide needs to contain one or more Conventions. Similarly, a Context requires at least one Constraint in order to exist. A Violation Log, however, could exist without any Violations in the cases when a Stylesheet is checked for conformance to a Style Guide and no violations are discovered. Both Stylesheet and Pattern are defined through 1 or more Constructs. Note that the concept of Stylesheet does not map to a CSS file, but to the whole CSS code that needs to be processed, regardless of type of CSS. This is why a Stylesheet requires at least one Construct in order to exist.

While the above grammar presents the composition of things, it does not illustrate how things interact with each other. To provide a better understanding of the dynamics between things defined in the ontology, a graph of the system is presented in Figure 5.7.

Figure 5.7: Graph of the system



According to the theory, a coupling occurs when the existence of a given thing affects the history of another thing and, in turn, history is defined as the chronological ordered states that a thing traverses [11]. Thus, in the domain-specific ontology a coupling exists between the Style Guide and the Convention things, because the existence of a Convention alters the state of the Style Guide. Similarly, a Context changes the state of a Convention and a Constraint affects the state of a Context.

There are also couplings between Construct, Pattern and Stylesheet things. Both Stylesheet and Pattern are composed of Constructs, and thus affected by their existence. Since a Pattern is a specific occurrence of a combination of Constructs, it is also coupled to Stylesheet.

A Violation is coupled to a Violation Log, since the presence of a new a Violation alters the state of the log. Further, a Violation Log contains information about the violations of a particular Style Guide that occur in a specific Stylesheet. In this sense, a Violation Log is a function of a Style Guide and a Stylesheet and it is coupled to both things.

There are a number of external events that can change the state of the system. An actor can initiate search for violations, which affects the state of the Violation Log and the Stylesheet. If during the search an actor discovers a violation, the state of the Violation is altered. Also, an actor can modify the Stylesheet and the Style Guide. Thus, couplings exist between the Actor and the Stylesheet, Violation, Violation Log and Style Guide.

5.3.2 Ontological analysis

The ontological analysis draws comparison between the designed domain-specific ontology and the domain-specific language. It consists of two mappings: representation and interpretation. The former mapping matches the ontology to the language and the latter - the language to the ontology. Typically ontological analysis is used to compare the abstract syntax of the language constructs to the concepts of an ontology. However, the designed ontology contains concepts that fall outside the scope of the abstract syntax. For example, the notion of Violation Log has a corresponding class in the system. However, that class denotes the product of the violations search and is not part of the abstract syntax. The particular approach chosen to conduct ontological analysis is to compare the abstract syntax of the language to the ontology. Additionally, all remaining ontological constructs are mapped to their representation in the whole system. For example, the ontological concept of a Violation Log is matched with the Violation Log class in the analysis module.

5.3.2.1 Representation and Interpretation Mappings

This subsection contains the two mappings between the modeling grammar and domain-specific ontology. Table 5.2 presents the representation mapping. The left-hand side of the table contains the ontological constructs in the order of definition and the right-hand side comprises the corresponding modeling grammar construct. Table 5.4 presents the interpretation mapping. The left-hand side contains classes from the abstract syntax, followed by classes that have a representation in the ontology and the right-hand side contains the corresponding ontological construct.

Ontological Classes and Subclasses

The ontological concept of Style Guide is represented in the modeling grammar as a Convention Set. Similarly, the notion of Convention maps to the Convention class and the concept of Context maps to the Pattern Descriptor class.

The ontological concept of Constraint appears in the modeling grammar as a number of Expressions. These include Expressions used in mappings for the subclasses of Constraint: Comparison Expression, Is Expression, Match Expression, In Expression and Node Query Expression. Additionally, the Constraint is mapped to And, Not and Or Expressions. These constructs cannot serve as Constraints themselves but are used to form more complex Constraints.

The mappings for most of the Constraint subclasses are straight forward. For example, the Comparison Constraint is represented as a Comparison Expression and the Type Constraint appears as an Is Expression. Similarly, the Textual Form Constraint is a Match Expression and the Set Membership Constraint maps to the In Expression. That said, the Existence Constraint require further attention. As defined in the ontology, the Existence Constraint requires the subject to exist. In the modeling grammar this concept is indirectly presented using multiple structures. Nodes in a pattern are specified using two ways: either through using Node Descriptor and Node Relations, or through Node Query Expression. The modeling grammar describes the target nodes from the pattern and their parent nodes using the former approach, and nodes that are nested in the target nodes - using the latter. For example, when a the pattern that needs to be matched is a ruleset that contains a z-index property and appears in a media query, the existence of the media query and the ruleset (and their relationship) is described using Node Descriptors and a Node Relation. The existence of the property contained in the ruleset is denoted using a Node Query Expression.

The ontological constructs of Violation Log and Violation are mapped to the Violation Log and Violation classes, respectively. The notion of a Stylesheet is represented by the Stylesheet Node in the solution and a Pattern is mapped to Css Pattern. The concept of a Construct is represented in the abstract syntax tree as a Variable Expression. The reason behind this mapping is that in the modeling grammar Variable Expression denote only Constructs.

Ontological Properties

The majority of the ontological properties are mapped to a single construct in the solution. For example, the Description property of Convention class in the ontology is directly mapped to the Description property of the Convention class in the modeling grammar. However, some of the property mappings are not that obvious. For example, the ontological property Conventions of class Style Guide is represented by a combination of properties in the solution: Contexts of Convention Set and Conventions of Context. Also, the notion of Ignored Constructs maps to the Context class in the solution. The reason for these mappings comes from the fact that the solution groups together conventions with identical ignored constructs. As stated in the ontology description, each convention specifies a set of constructs that need to be ignored while searching for its violations. For example, while evaluating the constraints of conventions related to newlines, the indentation constructs are typically ignored. Because the ignored constructs are similar for most of the conventions, the modeling grammar groups conventions that use the same ignored constructs into contexts. Thus, the Convention Set contains Contexts and a Context specifies the ignored constructs of all convention that it contains.

The Subject and the Value of a Constraint are mapped to the operands of Expression. The subclasses of Constraint are based on the different values of the Requirement property. Similarly, the Requirement of Constraint is expressed through child classes of Expression.

The ontological property Violations of the Violation Log construct appears in the modeling grammar as property Violations of the Violation Log class. The Violation class exposes Description and Position properties that map to the ontological properties Description and Position in Stylesheet. The Checked property of Stylesheet does not have a representation in the system. The property Property of Construct is denoted in the abstract syntax using a Call Expression.

5.3.3 Ontological Evaluation of the System

The primary purpose of the representation and interpretation mappings between the ontology and the modeling grammar is discovering discrepancies between the two entities. The four types of ontological anomalies are considered in the following subsections.

5.3.3.1 Redundant Constructs

Construct redundancy is a type of anomaly in which more than one modeling construct can represent a single ontological construct. The representation mapping illustrates that there are three candidates for this discrepancy.

First, the Conventions property of Style Guide is matched to Contexts property of Convention Set and the Conventions property of Context. Such mapping is required since the modeling grammar groups Conventions that share the same Ignored Constructs in a Context. Thus, a Style Guide in the modeling grammar does not possess conventions but a number of Contexts that, in turn, contain conventions. In this sense, the properties Contexts and Conventions together represent the concept of Conventions.

Note that redundancy occurs when the two grammar constructs can independently represent the same ontological concept. In the specific case, however, none of the constructs taken individually can express the property. It is their combination that represents the concept. Mapping a single ontological concept to a combination of modeling grammar constructs is an accepted approach and has been done in multiple studies [15]. Thus, the two constructs are not considered redundant.

Second, the Comparison, Is, In, Match, Node Query, And, Or and Not Expressions map to the concept of Constraint. As defined in the ontology, a Constraint is a class that has multiple subclasses depending on the specific requirement they enforce. In this sense, Expressions that map to any subclasses of Constraint, also map the Constraint class. These are the Comparison, Is, In, Match and Node Query Expressions. The remaining part of Expressions are And, Or and Not. These Expressions are used to combine Constraints into complex Constraints. For example, a Context may contain a number of Constraints that need to be put together using the And Expression. However, the And Expression itself cannot be a Constraint - it is used to build Constraints. In this sense, none of the three Expressions can independently represent a Constraint. Their combination with the Expressions mapped to the subclasses of Constraint expresses the concept of Constraint. Thus, the expressions are not considered redundant.

Third, the Existence Constraint is mapped to the Node Descriptor, Node Relation and Node Query Expressions. Existence Constraint is a type of Constraint that requires the Subject to exist. For example, the convention *Use a fallback property for RGBA and HSLA values* requires the fallback declaration to have the same property as the matched declaration and the value of the fallback declaration to be different than RGBA or HSLA. However, before these, the convention requires the fallback declaration to exist.

In the modeling grammar, when a Pattern Descriptor refers to multiple nodes, the presence of a Node Descriptor denotes the existence of CSS Node and the Node Relation specifies how the Nodes are connected in the Pattern Descriptor. As described in the domain analysis section, each Pattern Descriptor has one of more targets. These are the specific nodes that need to be iterated. While the Node Descriptors and Node Relations describe the target nodes in the Pattern and their parent nodes, the Node Query Expression is used to describe parts of the pattern that are nested in the targets.

As with the previous two cases, the ontological concept cannot be represented by Node Descriptors and Node Relations or by Node Query Expressions taken individually. Their combined use allows expressing patterns. For this reason, the constructs are not considered redundant.

5.3.3.2 Construct Overload

Construct overload emerges when a single modeling construct can represent a number of ontological constructs. In the interpretation and representation mappings every language construct is assigned a single ontological construct. In this sense, there are not candidates for overload discrepancy.

5.3.3.3 Construct Excess

Construct excess is a discrepancy in which a modeling construct does not have a mapping to an ontological construct. Typically, ontological analysis is performed through comparing the abstract syntax of the domain-specific language to the ontology. In the interpretation mapping all constructs that belong to the abstract syntax are mapped to an ontological construct. Thus, there are not candidates for construct excess from the abstract syntax.

5.3.3.4 Construct Deficit

Construct deficit appears when an ontological construct does not have a corresponding modeling structure. A candidate for such discrepancy is the property Checked of class Stylesheet as it appears without a matching construct in the system. However, maintaining the status of a Stylesheet is considered outside the scope of the system. Such functionality is highly dependent on the specific information offered by the IDE or text editor. Support for this property remains in the environment of the system. In this sense, the candidate is not considered an occurrence of deficit.

5.3.3.5 Results

In this section are presented the interpretation and representation mappings between the modeling grammar and the designed domain-specific ontology. Further, all types of discrepancies are analyzed and argumentation is provided why they should not be considered anomalies. Having the analysis in mind, it can be concluded that the ontological analysis provides sufficient evidence to consider the designed system both ontologically clear and complete.

Ontological constructs	Grammar constructs
Style Guide	Convention Set
Conventions (Style Guide)	Contexts (Convention Set), Conventions (Context)
Convention	Convention
Description (Convention)	Description (Convention)
Context	Pattern Descriptor
Ignored Constructs (Context)	Context
Constraint	Comparison, Is, In, Match, Node Query, Not, And, Or Expressions
Subject (Constraint)	Operand of Expression
Requirement (Constraint)	Subclasses of Expression
Value (Constraint Subclasses)	Second Operand of Binary Expression
Existence Constraint	Node Descriptor, Node Relation, Node Query Expression
Comparison Constraint	Comparison Expression
Type Constraint	Is Expression
Textual Form Constraint	Match Expression
Set Membership Constraint	In Expression
Literal Value	Literal Expression
Value (Literal Value)	Value (Literal Expression)
Violation Log	Violation Log
Violations (Violation Log)	Violations (Violation Log)
Violation	Violation
Description (Violation)	Description (Violation)
Position in Stylesheet (Violation)	Position (Violation)
Stylesheet	Stylesheet Node
Checked (Stylesheet)	-
Construct	Variable Expression
Property (Construct)	Call Expression
Pattern	Css Pattern
Constructs (Pattern)	Nodes (Css Pattern)

Table 5.2: Representation mapping

Grammar constructs	Ontological constructs
Convention Set	Style Guide
Contexts (Convention Set)	partly Conventions (Style Guide)
Context	Ignored Constructs (Context)
Conventions (Context)	partly Conventions (Style Guide)
Convention	Convention
Description property (Convention)	Description (Convention)
Pattern Descriptor	Context
Node Descriptor	partly Existence Constraint
Node Relation	partly Existence Constraint
Literal Expression	Literal Value
Variable Expression	Construct
Not Expression	partly Constraint
And Expression	partly Constraint
Or Expression	partly Constraint
Comparison Expression	Comparison Constraint
Match Expression	Textual Form Constraint
In Expression	Set Membership Constraint
Is Expression	Type Constraint
Call Expression	Property (Construct)
Node Query Expression	partly Existence Constraint
Css Pattern	Pattern
Nodes (Css Pattern)	Constructs (Pattern)
StylesheetNode	Stylesheet
Violations Log	Violations Log
Violations (Violations Log)	Violations (Violation Log)
Violation	Violation
Description (Violation)	Description (Violation)
Position (Violation)	Position in Stylesheet (Violation)

Table 5.4: Interpretation mapping

Chapter 6

Conclusion

The variability of CSS conventions used in practice cannot be handled by existing tools. Thus, developers often need to make sure their code complies to a given style guide manually. The thesis offers a solution to this problem. Its contribution is threefold:

1. First, the need for CSS conventions is evaluated through analyzing a total of 1,589,713 commits to 1,311,654 public repositories. Results indicate that 60% of all commits that maintain any form of CSS, still maintain plain CSS. The gathered data provides evidence to conclude that despite of the popularity of preprocessors, CSS is still handcrafted on GitHub in the beginning of 2015.

2. Second, to discover existing CSS code conventions two search engines were used and the first 100 results of each search were analyzed. As a result, 28 style guides containing 471 conventions were discovered. Analysis indicates that 151 of the conventions are unique. A list containing the description of conventions, their sources and detailed analysis is presented.

3. Third, a domain-specific language that is capable of expressing the gathered conventions is proposed. A proof of concept consisting of two parts is developed: a standalone Python package and a plug-in for Sublime Text editor. The implementation illustrates that the suggested approach enables automatic detection of violations of CssCoco conventions. The designed language is validated using ontological analysis. A domain-specific ontology, based on Bunge-Wand-Weber top level ontology, is defined and used as a reference in the ontological analysis. The conducted analysis of ontological discrepancies indicates that the language is both ontologically clear and complete.

Bibliography

[1] R. P. L. Buse and W. R. Weimer, "Learning a Metric for Code Readability," *IEEE Transactions on Software Engineering*, vol. 36, pp. 546–558, July 2010.

[2] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," *Software Engineering, IEEE Transactions on*, vol. 36, no. 4, pp. 546–558, 2010.

[3] T. Tenny, "Program readability: Procedures versus comments," *Software Engineering, IEEE Transactions on*, vol. 14, no. 9, pp. 1271–1279, 1988.

[4] D. Hyatt, "Guidelines for efficient CSS," 2000. https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Writing_efficient_CSS.

[5] E. Glaysher, "HTML/CSS Style Guide." https://google-styleguide.googlecode.com/svn/trunk/htmlcssguide.xml#General_Style_Rules.

[6] "GitHub." https://github.com/about/press.

[7] Wordpress, "CSS Coding Standards." https://make.wordpress.org/core/handbook/coding-standards/css/.

[8] Drupal, "CSS coding standards." https://www.drupal.org/node/1886770.

[9] T. R. Gruber, "Toward principles for the design of ontologies used for knowledge sharing," *International journal of human-computer studies*, vol. 43, no. 5, pp. 907–928, 1995.

[10] S. K. Milton and B. Smith, "Top-level ontology: The problem with naturalism," in *Formal ontology in information systems*, pp. 85–94, 2004.

[11] Y. Wand and R. Weber, "An ontological model of an information system," *Software Engineering, IEEE Transactions on*, vol. 16, no. 11, pp. 1282–1292, 1990.

[12] Y. Wand and R. Weber, "On the deep structure of information systems," *Information Systems Journal*, vol. 5, no. 3, pp. 203–223, 1995.

[13] R. Weber and Y. Zhang, "An analytical evaluation of niam's grammar for conceptual schema diagrams," *Information Systems Journal*, vol. 6, no. 2, pp. 147–170, 1996.

[14] D. L. Moody, "The "physics" of notations: toward a scientific basis for constructing visual notations in software engineering," *Software Engineering, IEEE Transactions on*, vol. 35, no. 6, pp. 756–779, 2009.

[15] A. Gehlert and W. Esswein, "Toward a formal research framework for ontological analyses," *Advanced Engineering Informatics*, vol. 21, no. 2, pp. 119–131, 2007.

[16] P. Green and M. Rosemann, "Integrated process modeling: an ontological evaluation," *Information systems*, vol. 25, no. 2, pp. 73–87, 2000.

[17] B. Bos, T. Çelik, I. Hickson, and H. W. Lie, "Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification," *W3C Recommendation*, June 2011. http://www.w3.org/TR/2011/REC-CSS2-20110607.

[18] E. J. Etemad, "Cascading Style Sheets (CSS) Snapshot 2010," *W3C Working Group Note*, May 2011. http://www.w3.org/TR/2011/NOTE-css-2010-20110512/.

[19] H. Catlin, N. Weizenbaum, and C. Eppstein, "SASS: Syntactically Awesome Style Sheets," 2006. http://sass-lang.com.

[20] A. Sellier, J. Schlinkert, L. Page, M. Bointon, M. Jurčovičová, M. Dean, and M. Mikhailov, "Less," 2009. http://lesscss.org.

[21] T. J. Holowaychuk, "Stylus," 2015. https://learnboost.github.io/stylus.

[22] G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman, "Lean ghtorrent: Github data on demand," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 384–387, ACM, 2014.

[23] A. L. Opdahl and B. Henderson-Sellers, "Ontological evaluation of the uml using the bunge–wand–weber model," *Software and systems modeling*, vol. 1, no. 1, pp. 43–67, 2002.

[24] J. Parsons and Y. Wand, "Using objects for systems analysis," *Communications of the ACM*, vol. 40, no. 12, pp. 104–110, 1997.

[25] M. Rosemann and P. Green, "Developing a meta model for the bunge–wand–weber ontological constructs," *Information systems*, vol. 27, no. 2, pp. 75–91, 2002.