Detecting Violations of CSS Code Conventions

Boryana Goncharenko
boryana.goncharenko@gmail.com

July 28, 2015, 29 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 anti- patterns. 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 reponsibility of the developers to ensure that their code complies to a given style guide. Typically, they need to read, 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 Masters 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 are restricted to define concepts and their relations that fall in 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 presented in [12]

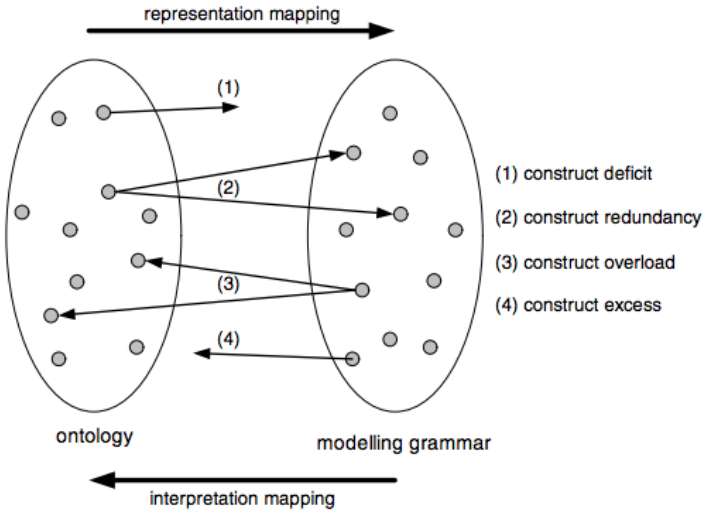
Ontological construct	**Explanation**
Thing	The elementary unit in the BWW ontological model. The real world is made up of things. A composite thing may be made up of other things (composite or primitive).
Properties	Things possess properties. A property is modeled via a function that maps the thing 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.
State	A vector of values for all property functions of a thing.
Event	A change of state of a thing. It is effected via a transformation.
Transformation	A mapping from a domain comprising states to a codomain comprising states.
History	The chronologically ordered states that a thing traverses.
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.

Class	A class is a set of things that can be defined via their possessing a characteristic property.
Kind	A kind is a set of things that can be defined only via their possessing two or more properties.
System	A set of things is a system if, for any bi-partitioning of the set, couplings exist among things in the two subsets.
System Composition	The things in the system are its composition.
System Environment	Things that are not in the system but interact with things in the system are called the environment of the system.

2.2 Ontological analysis

Ontological analysis is an established approach for evaluating the quality of software engineering notations [13]. 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 [14]. The underpinning of ontological analysis is that modeling grammars are incomplete if they are not able to represent what exists in reality [15]. 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 [14]



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. [13]

Chapter 3

Evaluating the Need for CSS Code Conventions

3.1 Research Method

Despite the new features added in the second [16] and third [17] 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 [18], LESS [19] and Stylus [20] 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.

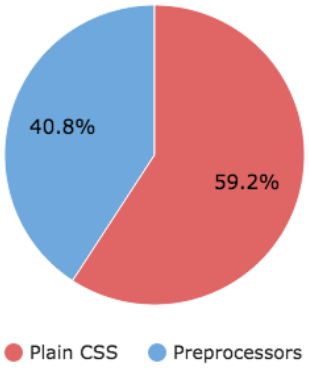
3.2 Results

A total of 1,589,713 commits to 1,311,654 public repositories have been analyzed. X of these were not processed due to errors. Typical errors contain repositories that have been turned private and... The number of commits that maintain any form of CSS is X. Figure 3.1 summarizes the findings.

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 [21]. 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.

Figure 3.1: Results



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 *Dont 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* implies that values should appear on one line. Such implicit conventions were registered as explicit conventions.

4.2 Results

As a result of the searches, a corpus of 165 unique CSS code conventions was accumulated. Sources of these conventions include CSS professionals, open-source communities and companies, such as Google, GitHub, Wordpress, Drupal. The following list presents a selected set of the discovered conventions:

- Avoid qualifying ID and class names with type selectors.
- Use RGBA only when opacity is needed.

- Use short hexadecimal values.
- CSS files must not include any @charset statements.
- Disallow @import.
- When possible, use em instead of pix.
- Avoid using z-indexes when possible.
- Require compatible vendor prefixes.
- Do not use !important.
- 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.
- 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.
- 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.
- A rule must not contain height and border, border-top, border-bottom, padding, padding-top, or padding-bottom.
- A rule must not contain width and border, border-left, border-right, padding, padding-left, or padding-right.
- Warning if a property is included in a rule twice and contains the same value.
- Forbid empty rules.
- A vendor-prefixed property must be followed by a standard property.
- Require fallback color. A color property with a RGBA(), HSL(), or HSLA() color without a preceding color property that has an older color format.
- A rule that has display: inline-block should not use float.
- A rule that has display: block should not use vertical-align.
- Use 4 spaces for indentation, no tabs.
- 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:

```
1 .myclass { }              /* violation */
2 .myclass { /* Comment */ } /* violation */
3 .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.

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.

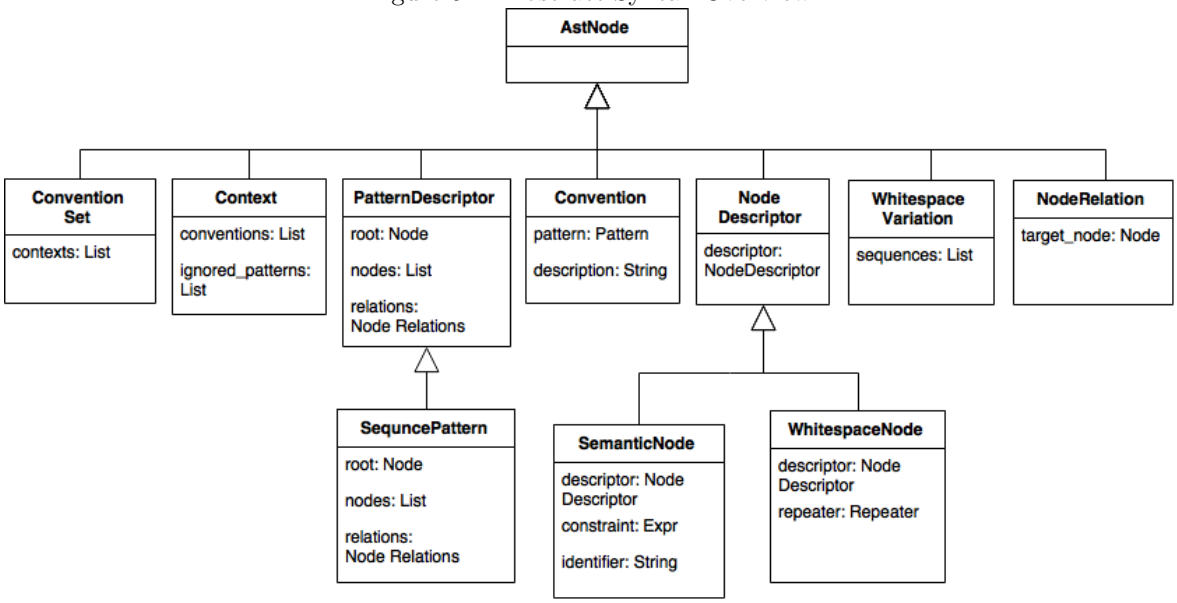
Conventions may use CSS-specific knowledge. For example, the rule *Use lowercase for properties; vendor-prefixed properties are exception* requires differentiating between standard and vendor-specific properties. While in this case the two types of properties can be easily distinguished, some conventions 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 Abstract syntax

This section describes the abstract syntax of the designed domain-specific language. The name of the language is CssCoco which is an short version of CSS Code Conventions. A prototype of the language is available at `https://github.com/boryanagoncharenko/CssCoco`. 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.

SequencePattern represents a special type pattern in which nodes are allowed to be only siblings. 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.

Node represents a description of a node used in a PatternDescriptor. Attribute `descriptor` contains information about the type of the described node. Attribute `constraint` is an expression that designates additional 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 `descriptor` contains information about the type of the described 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.

WhitespaceVariation represents a description of acceptable whitespace sequences. Whitespace variations are allowed to appear as operands of BeforeExpr, AfterExpr and BetweenExpr. Attribute `descriptor` contains information about the type of the described node. Attribute `constraint` is an expression that designates additional constraints applied to the node. Attribute `identifier` is a given string that can be used as a reference to the matched node.

NodeRelation represents a relation between two Nodes. Specializations of node relation are previous sibling, next sibling, parent and ancestor relations. Attribute `target_node` designates 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.

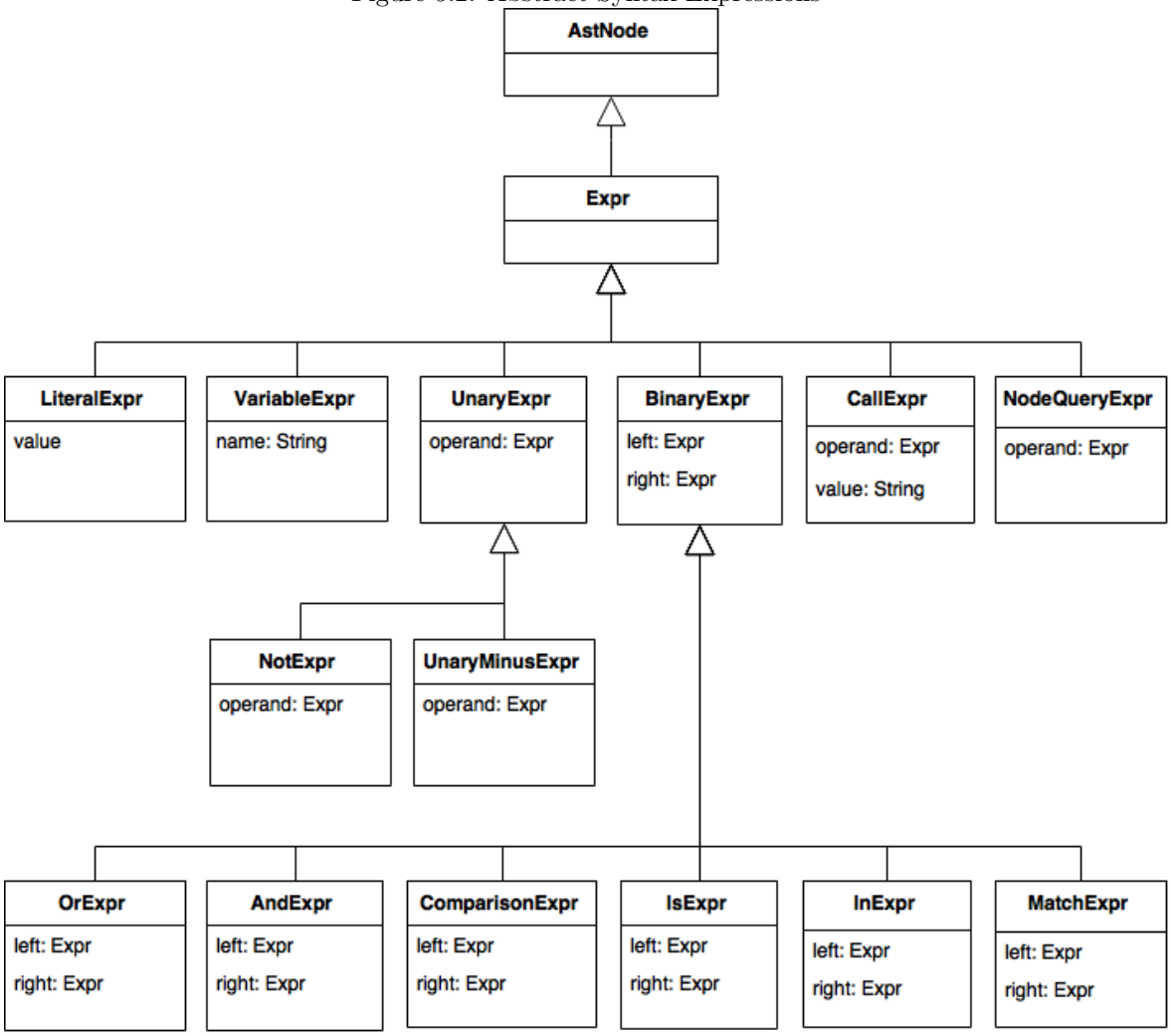
ComparisonExpr represents expression that compares two operands.

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.

Figure 5.2: Abstract Syntax Expressions



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.

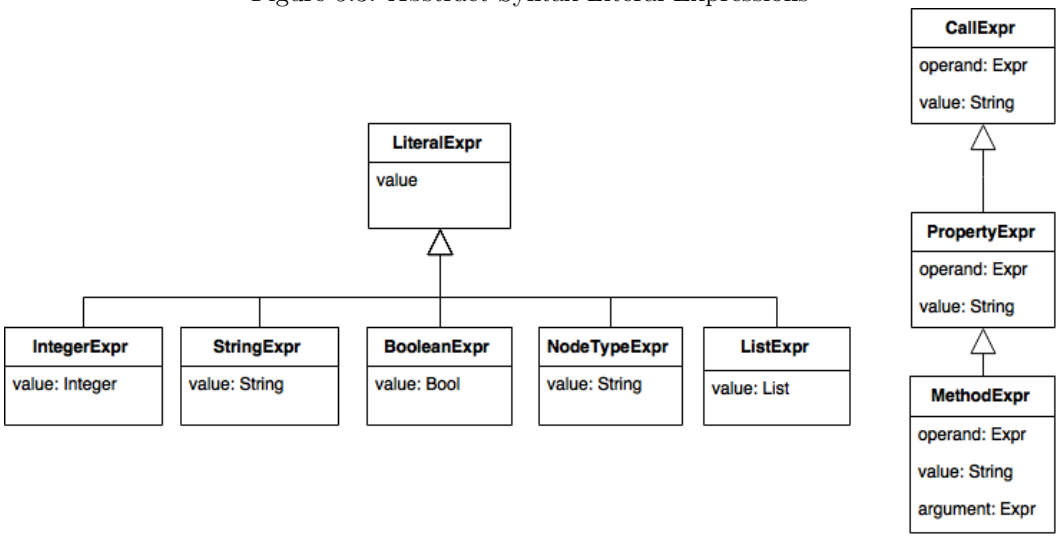
StringExpr represents expression containing a string value.

BooleanExpr represents expression containing a boolean value.

ListExpr represents expression containing a list value. The elements of the list are of type Literal-Expr.

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

Figure 5.3: Abstract Syntax Literal Expressions



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 5.4: Abstract Syntax Node Query Expressions

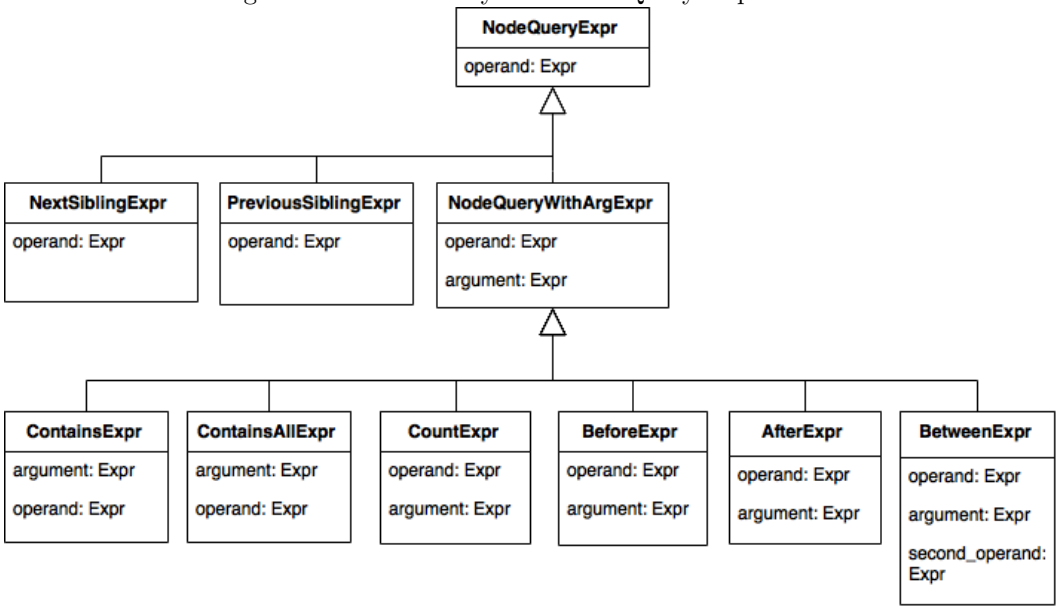


Figure 5.4 presents an overview of the Node Query Expressions. The following listing describes the subclasses in details:

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.

5.3 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. Such categories refer to different types of nodes and require ignoring certain patterns.

Abstract Syntax Mapping: ast.Context.

```
context : Identifier '{' convention* '}' ;
```

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. It aims at improving readability of conventions.

Abstract Syntax Mapping: ast.Convention.

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

pattern represents a pattern of nodes and their relations. It can describe a horizontal sequence of sibling nodes and or a vertical pattern of nested nodes. Also, it can describe pairs of elements.

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 ;
```

semantic_node represents a non-whitespace node. It describes the type of the node and its additional constraints.

Abstract Syntax Mapping: ast.Node.

```
semantic_node : node_type ('{' logic_expr '}')? ;
```

whitespace_variation represents a sequence of whitespace nodes. They differ from the rest of the nodes types because the user can specify how many times they need to be consecutively repeated.

Abstract Syntax Mapping: ast.WhitespaceVariation, ast.WhitespaceNode, ast.Repeater.

```
whitespace_variation : whitespace_node ('or' whitespace_node)* ;
whitespace_node : Identifier ('{' 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 of whitespace 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' type_=Identifier
          | whitespace_variation ('before' | 'after') type_operand
          | whitespace_variation '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 | Integer | String | list_ ;
```

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

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

```
call_expr : call_expr '.' call_expr
          | 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] ;
```

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

Abstract Syntax Mapping: ast.ListExpr.

```
list_ : '[' list_element (',' list_element)* ']' ;
list_element : Integer | 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
          ;
```

5.4 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 [22, 15, 13, 23, 24]. The particular 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.

5.4.1 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 [13]. 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 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 themselves subjects of ontological analysis 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 **Number of Conventions** indicates the size of 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** explains the meaning of the Convention in natural text.

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. Constraints are individual requirements that are used in a **Context** to form a description of a **Pattern**. There are different types of Constraints represented below as *kinds*.

Property **Subject** indicates the operand on which the Constraint operates. Different kinds of Constraints have different type of Subjects. Things that can be a subject are the Type, Textual Representation or CSS-specific knowledge of a Construct.

Kind **Existence Constraint** is a type of Constraint that requires existence of a Construct.

Property **Subject** indicates the operand on which the Constraint operates. Specifically, Subjects of the Existence Constraint are Constructs.

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

Property **Subject** indicates the operand on which the Constraint operates. Specifically, the Subjects of Comparison Constraints can be any other Constraint.

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

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

Property **Subject** indicates the operand on which the Constraint operates. Specifically, the Subjects of Type Constraints are Constructs.

Property **Type** denotes the type that should be met for the Constraint to be satisfied.

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

Property **Subject** indicates the operand on which the Constraint operates. Specifically, the Subjects of Textual Form Constraints are Constructs.

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

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

Property **Subject** indicates the operand on which the Constraint operates. Subjects of Set Membership Constraints are other Constraints.

Property **Set** denotes the set that the Subject should be present at for the constraint to be satisfied.

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 **Type** identifies the function of a Construct in the program. Examples are strings, attribute values etc.

Property **Textual Representation** is the string of a Construct that appears in the CSS program. Examples are tabs, ";", and ".myclass"

Property **CSS-specific knowledge** encapsulates properties of nodes specific to the CSS domain. For example, a CSS declaration node can possess knowledge whether it is vendor specific or not.

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 **Number of Constructs** denotes the size of 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 (of 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 }

TODO: include property and event of the Stylesheet!

The descriptions provided 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 the Backus-Naur Form (BNF) notation:

```
STYLE_GUIDE     ::= CONVENTIONS
CONVENTIONS     ::= CONVENTIONS CONVENTION | CONVENTION
CONVETION       ::= CONTEXT
CONTEXT         ::= CONSTRAINTS
CONSTRAINTS     ::= CONSTRAINTS CONSTRAINT | CONSTRAINT
CONSTRAINT      ::= COMPARISON_CONSTRAINT | TYPE_CONSTRAINT | TEXT_CONSTRAINT
                  | SET_CONSTRAINT | CONTEXT_CONSTRAINT
VIOLATION_LOG   ::= VIOLATIONS
VIOLATIONS      ::= VIOLATIONS VIOLATION | VIOLATION | EPSILON
STYLESHEET      ::= CONSTRUCTS
CONSTRUCTS      ::= CONSTRUCTS CONSTRUCT | CONSTRUCT
PATTERN         ::= CONSTRUCTS
```

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 (internal or external). 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 5.5.

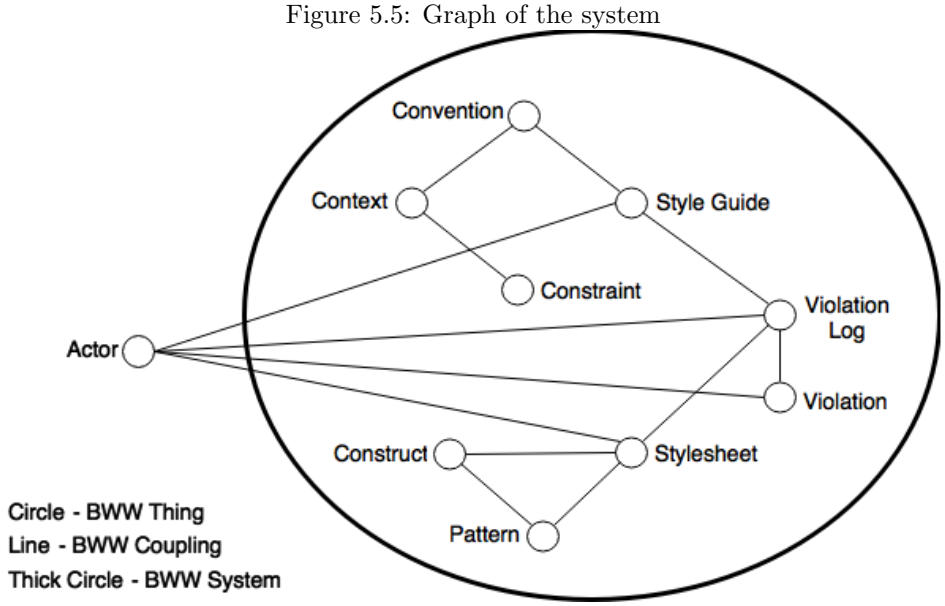
According to the theory, a coupling occurs when the existence of a given thing affects the history of another thing and, in turn, history is defined as the chronological ordered states that a thing traverses [11]. Thus, in the domain-specific ontology a coupling exists between the Style Guide and the 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. Also, an external for the system author can modify the Constructs and change the state of Stylesheet. Thus, couplings exist between the Actor and the Stylesheet, Violation Log and Style Guide.

Figure 5.5: Graph of the system



5.4.2 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. In the particular case, the language aims at expressing the existing code conventions. This means that only the part of ontology that describes a Style Guide should be included. Since the domain of the ontology refers to detecting violations of code conventions, it does not map to the language constructs only, but to the whole solution that comprises the language and its interpreter. Thus, the particular approach used to conduct the ontological analysis is comparing the designed domain-specific ontology to the whole designed system. We believe that such comparison provides a thorough analysis of the whole solution.

5.4.2.1 Representation Mapping

In this section the constructs of the domain-specific ontology are mapped to the constructs of the modeling grammar. Table 5.1. presents the correspondence between the two sets of constructs.

Table 5.1: Representation mapping

Ontological constructs	Grammar constructs
Style Guide	Convention Set
Number of 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)	Variable Expression, Call 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
Basic Value	Literal Expression
Violation Log	Violation Log
Number of Violations (Violation Log)	Violations (Violation Log)
Violation	Violation
Description (Violation)	Description (Violation)
Position in Stylesheet (Violation)	Position (Violation)
Stylesheet	Stylesheet Node
Checked (Stylesheet)	-
Construct	Css Node
Type (Construct)	Search Categories (Css Node)
Textual Representation (Construct)	String (Css Node)
CSS-specific Knowledge (Construct)	API (Css Node)
Pattern	Css Pattern
Number of Constructs (Pattern)	Nodes (Css Pattern)

Ontological Classes and Kinds

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.

REWAMP! The ontological concept of Constraint appears in the modeling grammar as an Expression (not Literal, Variable, Call expression?). Most of the kinds of Constraints are represented by a single subclass of Expression. For example, the Comparison Constraint maps to the Comparison Expression and the Type Constraint maps to the Is Expression. Similarly, the Textual Form Constraint is represented by the Match Expression and the Set Membership Constraint maps to the In Expression. However, the Existence Constraint maps to multiple constructs in the modeling grammar. The Existence Constraint appears in two forms in the solution: first, it appears when a Pattern Descriptor is defined. The structure of the pattern is expressed using a Node Descriptors that specifies the expected type of node and the any additional requirements the node needs to fulfill. In this sense, a Node Descriptor is a combination of a Constraints. However, the presence of a Node Descriptor denotes an Existence Constraint. It declares that a node must exist. To describe a more complex pattern, Node Relations are used to glue different Node Descriptors. For example, when two nodes have to be siblings, this is expressed through a sibling node relation. However, in certain cases the relations between Node Descriptors cannot be described. For example, when a convention imposes a constraint that a stylesheet must not contain more than 10 occurrences of a given property, it is not important whether the stylesheet is a parent of an ancestor of the properties. In such cases, the Node Query Expression is used.

The ontological constructs of Violation Log and Violation are mapped to the Violation Log and Violation classes, respectively. The notion Stylesheet is represented by the Stylesheet Node in the solution. Similarly, the Construct is mapped to Css Node and Pattern is mapped to Css Pattern.

The majority of the ontological classes have a direct mapping to a construct in the solution. For example, a Style Guide is mapped to a Convention Set, a Convention is mapped to Convention etc. The notion of Constraint is a particular limitation used in a Context and typically, it is applied to a single node. The corresponding construct in the system is Expression. Similarly, the specific kinds of Constraints are mapped to subclasses of the Expression. Specifically, a Comparison Constraint matches the notion of Comparison Expression. The Type Constraint is expressed through the Is Expression that verifies the type of a node. The Textual Form Constraint maps to the Match Expres-

sion, which allows using regular expressions on strings. The Set Membership Expression corresponds to the In Expression that checks whether the subject belongs in a given list of values. The Context Constraint matches the Node Query Expression, which retrieves node that have a given relation to the current node.

The concept of a Construct corresponds to a CssNode in the system and a Stylesheet maps to a StylesheetNode that refers to the root of the CSS tree. A Pattern is represented in the language by CssPattern. Note that Node Relation does not contribute to the ontological concept of a Pattern. A Node Relation is used to describe the relationship between nodes and it a part of the PatternDescriptor.

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 Number of 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 come 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.

!REVAMP The property Subject of class Constraint maps to Expression, just like the Constraint class itself. This mapping is explained by the fact that in the solution Expressions are recursively defined.

The ontological property Number of Violations of the Violation Log construct appears in the modeling grammar as property Violations of the Violation Log class. Similarly, 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 Type of class Construct corresponds to the Search Categories of Css Node, which allows a single node to be referred using a number of different categories. The Textual Representation property appears as a String property in Css node. The property CSS-specific knowledge of class Construct is mapped to the API of Css Node. Each Css Node in the system comprises a number of properties that are meaningful for its function in the CSS tree. The Number of Constructs property of Pattern is mapped to the Nodes property in Css Pattern class.

5.4.2.2 Interpretation Mapping

In this section the constructs of the designed system are mapped to the constructs of the domain-specific ontology. Table 5.2. presents the correspondence between the two sets of constructs.

Table 5.2: Interpretation mapping

Grammar constructs	Ontological constructs
Convention Set	Style Guide
Contexts (Convention Set)	partly Number of Conventions (Style Guide)
Context	Ignored Constructs (Context)
Conventions (Context)	partly Number of 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	Basic Value
Variable Expression	partly Subject (Constraint)
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	partly Subject (Constraint)
Node Query Expression	partly Existence Constraint
Css Node	Construct
Search Categories (Css Node)	Type (Construct)
String (CssNode)	Textual Representation (Construct)
API (CssNode)	CSS-specific knowledge (Construct)
Css Pattern	Pattern
Nodes (Css Pattern)	Number of Constructs (Pattern)
StylesheetNode	Stylesheet
Violations Log	Violations Log
Violations (Violations Log)	Number of Violations (Violation Log)
Violation	Violation
Description (Violation)	Description (Violation)
Position (Violation)	Position in Stylesheet (Violation)

Abstract Syntax

Most of the classes in the Abstract Syntax are mapped to a single ontological construct. For example, the Convention Set denotes the concept of Style Guide and the Convention class is mapped to the Convention ontological construct. The properties Contexts of Convention Set and Conventions of Context both represent the ontological property Number of Conventions of Style Guide. This mapping occurs because the abstract syntax puts together Conventions with identical ignored constructs. As a result, the Convention Set comprises Contexts, which, in turn, contain Conventions that share the same ignored constructs. This also, explains why the Context class maps to the ontological property of Ignored Constructs of Context. The classes Node Descriptor and Node Relation represent an Existence Constraint.

The Expression class maps to both Constraint and Subject of Constraint. By definition, the Subject is a property of Constraint which denotes the thing that the Constraint acts upon. Since Expressions are recursively constructed, the Expression class maps to the two ontological constructs. However, not every subclass of Expression can be represented as a Constraint. In fact, the Literal, Variable and Call Expressions can appear only as Subjects. For example, the Literal Expression with value 4 cannot be a Constraint. Rather it is used by a Constraint

Css Tree

Analysis

The majority of modeling constructs that belong the abstract syntax of the language map to a single ontological construct. For example, Convention Set corresponds to the concept of a Style Guide, Convention maps to the concept of Convention, and Expression matches a Constraint. Some of the language constructs, however, map partly to ontological concepts. For example, the Number of Conventions in a Convention Set can be extracted by the number of contexts and the number of conventions in each context. In this sense, both of the constructs partly support the ontological concept. Similarly, the notion of Context could be represented by a Pattern Descriptor or a Node

Descriptor. Logical expressions are used to glue other subtypes of Expressions in order to express a single ontological Context.

5.4.3 Ontological Evaluation of the System

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

5.4.3.1 Redundant Constructs

Construct redundancy is a type of anomaly in which more than one modeling constructs can represent a single ontological construct.

- Contexts and Conventions together represent Number of Conventions (Style Guide)

- Constraint is presented by a number of Expressions. However, these are the union of all types of Constraints. Additionally, the Not, And and Or expressions are used to form more complex Constraints.

- Existence Constraint is mapped to Node Descriptor, Node Relation and Node Query Expression Expressions.

- Value (Constraint) is mapped to Expression.

Candidate redundant constructs are the CallExpression, LiteralExpression, VariableExpression and UnaryExpression, since they all appear to map to the same ontological construct - Constraint. However, none of these constructs, taken individually, is capable of representing the concept of Constraint. It is their combination that expresses the concept. For this reason, the candidate constructs are not considered redundant. In fact, a number of researchers have compared a construct to a combination of constructs [14].

5.4.3.2 Construct Overload

Construct overload emerges when a single modeling construct can represent a number of ontological constructs. No candidates for construct overload have been found.

5.4.3.3 Construct Excess

Construct excess is a discrepancy in which a modeling construct does not have a mapping ontological construct. The system contains a number of classes and packages that do not have a corresponding ontological concepts, e.g. Type Checker. However, the class is used to ensure the correctness of the Style Guide. In this sense, it has a supporting function that contributes to an ontological concept.

5.4.3.4 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 can be added based on whether the IDE or text editor expose such knowledge. That said, the structure remains in the environment of the system and thus, the candidate is not considered a deficit.

Having the above consideration in mind, we believe the ontological analysis provides sufficient evidence to consider the designed system both ontologically clear and complete.

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 contribution of this thesis is threefold. First, the need for CSS conventions is evaluated and evidence that CSS is still handcrafted is provided. Second, a list of the existing code conventions and their analysis is presented. Third, a domain-specific language that is capable of expressing the gathered conventions is proposed. A prototype of language and its interpreter are implemented to enable automatic detection of violations.

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] 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.

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

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

[16] 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.

[17] 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/.

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

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

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

[21] 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.

[22] 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.

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

[24] 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.

[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.