Language Design and Implementation for the Domain of Coding Conventions

Boryana Goncharenko
University of Amsterdam, The Netherlands
boryana.goncharenko@gmail.com

Vadim Zaytsev
Raincode, Belgium
University of Amsterdam, The Netherlands
vadim@grammarware.net

Abstract

Coding conventions are lexical, syntactic or semantic restrictions enforced on top of a software language for the sake of consistency within the source base. Specifying coding conventions is currently an open problem in software language engineering, addressed in practice by resorting to natural language descriptions which complicate conformance verification. In this paper we present an endeavour to solve this problem for the case of CSS — a ubiquitous software language used for specifying appearance of hypertextual content separately from the content itself. The paper contains the results of domain analysis, a short report on an empirically obtained catalogue of 143 unique CSS coding conventions, the domain-specific ontology for the domain of detecting violations, the design of CssCoco, a language for expressing coding conventions of CSS, as well as a description of the tool we developed to detect violations of conventions specified in this DSL.

Categories and Subject Descriptors D.3.0 [*Programming languages*]: General

Keywords conventions; software language design

1. Introduction

Coding conventions have probably been used almost as long as programming languages. They can be viewed as a palliative on the way to design a proper language [62], or as linguistic constructs that cover shortcomings of the base language [79], or as guidelines for increased maintainability and knowledge propagation [72, 83]. For mainstream software languages for object-oriented programming and design there is substantial work on naming conventions [11, 12], calling conventions [7, 64], modelling conventions [51, 79]. We join this trend by contributing our findings about conventions used in CSS stylesheets.

There is a shortage of research on CSS — and not just in the SLE context. In the related work section below (§ 3.1) we seek, describe and classify all papers ever written about CSS, 41 in total. The shortage of CSS language ecosystem research alone could serve as a reason for our venture, and combined with the widespread use of this technology it becomes an indispensability.

After more detailed motivation for this project and establishing its relevance in § 2 and analysing whatever related work is available in § 3, we will proceed with § 4 explaining our domain analysis. Having obtained enough knowledge of the domain, we have performed top-down domain specific software language design, going through ontology as a domain model (§ 5); abstract syntax as an implementation model (§ 6); concrete syntax as an interface intended for language users; and developed a proof of concept (§ 7). In § 8 we will conclude the paper and discuss some afterthoughts.

Crucial information about these components and their implementation is shared in this paper, but the details are left to the supplement available as `http://dx.doi.org/10.6084/m9.figshare.3085831.v3`. The accompanying command line tool and the Sublime Text plugin were *excluded* from the artefact evaluation due to the conflict of interest (the last co-author co-chairing the AEC).

2. Motivation and Relevance

To establish whether dedicating any research effort to coding conventions in CSS is sensible, we formulate and substantiate three claims.

Claim 1: People use coding conventions.

Using plain internet search engines such as Google Search to show existence/prevalence is unreliable due to their personalising optimisation strategies. Search engines maintain personalised information bubbles [76] that display content to the users that they are likely to like and agree with. The scientifically well-founded and engineered Private WebSearch plugin by Saint-Jean et al. [76] does not seem

Copyright © ACM, 2016. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in Proceedings of the Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, SLE'16, Amsterdam, The Netherlands, http://doi.acm.org/10.1145/2997364.2997386.

Copyright © ACM . . .

to be available any more. According to the Private Search Engine List[1] comparative analysis, its reasonable contemporary substitute that aggregates results acquired anonymously from several engines based on ratings of Web of Trust, is Privatelee[2].

Searching for `"coding convention"` on Privatelee reportedly yields 87,338 hits. To gain more trust in these results, we have analysed the first 100 relevant links, skipping over duplicate results, general discussions and pages about conventions in data encoding, as well as a few out of scope false positives. As can be seen in the left column of Figure 1, there is a fair presence of many mainstream and some domain-specific languages. Each search result was checked manually and found to be either devoted to the importance of coding conventions, or — very occasionally — about the harmful effect existing conventions have on software. The raw data can be inspected in Appendix A.

Claim 2: There are coding conventions in CSS.

Since we now know which languages tend to be discussed in the context of coding conventions, we can investigate them further by collecting the number of search results of queries consisting of `"coding convention"` and the name of the language (or a list of related alternatives). We also add top 50 languages from the well-known TIOBE list. To establish a threshold of falsifiability, we apply the Potato Criterion: “potato” is not a software language but is a word occasionally used in examples, so we did the same search for “potato” and filtered out the languages that scored lower than its score of 997 pages found. As can be seen in the right column of Figure 1, SQL turns out to be the most popular focus of attention with 75,102 results (mostly because conventions of C#, PHP and other languages include guidelines for embedded SQL). Perl takes the second place, again with the help of its influential language design decisions (many guidelines include words like “perl-like” or “perl-style”). Fortran with 34,701 is on the third place, closely followed by CSS with 33,602 which leaves HTML, VB, Java, PHP, JavaScript, C++ and other languages far behind. The data with proof links is included in Appendix B.

Claim 3: Maintenance on pure CSS sheets is still being performed.

CSS 3 [14] added many new features to the language, but practitioners often rely on even more powerful extensions and alternatives (called “preprocessors” by an established misnomer tradition) such as SASS [13], LESS [80] and Stylus [39] that support variables and other well-sought functionalities. If most CSS is generated from higher level specifications, any conventions it might have, are irrelevant.

We use GitHub, which is currently with 10 million users and 24 million repositories the largest code host in the world [30]. Using BigQuery[3] to access GitHub’s public

[1] Private Search Engine, http://www.privatesearchengine.com

[2] Privatelee, https://privatelee.com

[3] Google BigQuery, http://bigquery.cloud.google.com

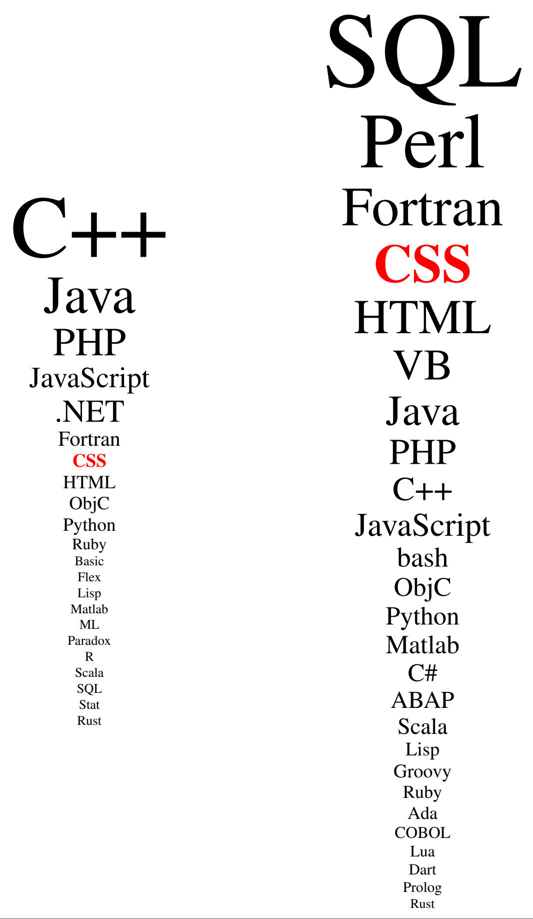


Figure 1. *On the left*, the word cloud of the first 100 related results for Privatelee search for coding conventions. Languages are understood broadly: “C++” includes C, “ObjC” includes both Swift and Objective C, “.NET” covers XAML. *On the right*, the word cloud of the number of hits reported by Privatelee for coding conventions per language, for all languages that scored higher than a *potato*.

dataset, we find 2,331,864 public repositories that have been updated in the period of January to April 2015. Disregarding 253,611 (10.9%) of them because those have become private or were deleted by the time of our investigation, and 41,274 (1.8%) more because those were too large to process without significant effort, we looked deeper in the remaining repositories for commits that did any maintenance on CSS. In total we analysed 2,282,788 commits, of which more than half (1,340,217, or 58.7%) involved only `.css` files and the remaining 41.3% included preprocessor maintenance (on `.scss`, `.sass`, `.less`, `.hss` or `.styl` files).

We conclude that coding conventions are an attractive topic for practitioners, who still perform significant maintenance activities on handcrafted CSS specifications and thus care how they look and to which conventions they conform. The existence of recommended coding conventions for most popularly used languages and active discussions around them establish that need as well.

3. Related Work

Given the context of the software language engineering conference, we omit any explicit links to the related body of knowledge on domain specific language design: there are many books and papers on that topic, and most of them are universally well known in the community. We do, however, feel the need to position our project and contributions in the context of two other research directions: CSS and coding conventions.

3.1 Cascading Style Sheets

CSS is found reliably useful by forward engineers of web content. However, it is surprisingly scarcely covered by existing research. We could not resist the temptation to refer to **all** published papers on the subject and then focus on the highlights. Appendix C contains the list of papers with extended bibliographical information such as DOI links.

To collect paper candidates, we used DBLP, which covers extensively all workshops, conferences, journals, books, preprint repositories and even encyclopaedias, collectively over three million papers. Its protection against crawling and automated data gathering is also not as strict as Google Scholar's. Using two search queries: "css"[4] and "cascading style sheets"[5] we found $144 + 14$ items. As a way to ensure there are no new unindexed papers left over, we did a direct search on Science Direct, Springer Open, ACM Digital Library and IEEE Xplore, which led to $97+1$ more candidates (all from Elsevier). After removing very few duplicates as well as substantial number of non-peer-reviewed preprints, books, book reviews and encyclopaedia entries, we read the abstracts and filtered out false positives about curvature scale space, clustering and scoring strategy, carbohydrate structure suite, etc. The remaining **41** papers were read and classified into following topics:

- **Application** of CSS (out of the SLE scope): general discussions [20, 48, 53, 82], case studies [4, 37].
- **Shortcomings** of CSS: harmful effect on indexing [34], language improvement with constraints [6], extending with Javascript [1].
- **"Preprocessors":** study of current use [56], making a new one [81], building DSLs on top [24, 74, 88].
- **Syntactic conformance:** two reports on the same project to classify errors in HTML and CSS [67, 68].
- **Refactoring:** removing redundant rules [35, 58], size reduction in general [10], clone detection [57], accessibility adaptation [98], personalisation [36, 85].
- **Analysis:** verification [26], defect prediction [8], optimisation [55], performance and energy consumption [77], complexity metrics [2], accessibility metrics [3], quality metrics [45].
- **Security and privacy:** data hiding [15, 95], engineering attacks [49], protecting browsers from them [41], fingerprinting [89].
- **IDE support**: new computation models [18], new engines [17, 52], libraries/plugins [43], generative tool support [44], change impact visualisation [71].

Mazinanian et al. [57] have recently investigated clones in CSS files and found a stunning 60% fraction of all CSS belonging to clones. However, some of this duplicate code is impossible to refactor since CSS 3 has no variables, functions and other methods commonly providing alternative means of model reuse. The same authors also go as far as to suggest removal of selectors that seem to be unused, from the deployed stylesheets [58]. The problem is easily solvable for the case of static hypertext when analysing HTML pages is enough to assess coverage of CSS selectors. For more typical, dynamic web pages, the usefulness of this advice heavily depends on the crawler that detects which selector configurations (in the form of DOM states) are reachable at runtime. Their prototype called Cilla is available at `http://github.com/saltlab/cilla`.

The state of the art in applying *static* source code analysis techniques to this problem is the work of Hague et al. [35], who formalised the problem of detecting unused CSS rules in terms of symbolic pushdown systems, taking most of the practical aspects into consideration, such as investigating jQuery calls together with the HTML itself. The solution prototype is available as a tool called TreePed at `http://bitbucket.org/TreePed/treeped`.

If we allow ourselves to be more conservative, we can rely on semantic stylesheet simplification; it tends to remove around 5% of declarations and refactor another 5% while providing universal guarantees of static semantic preservation independently of the coupled hypertext, as recently demonstrated by Bosch et al. [10]. Interestingly, Wu et al. [95] solve the opposite problem of introducing redundant selectors into existing CSS files in order to hide data in Epub books.

Park et al. [67] studied what kinds of errors undergraduate students made in a web development course. Even though they focused on both CSS and HTML, none of the commonly occurring errors they found were related to CSS in any way. Their taxonomy of errors [68] did include error types applicable to CSS, but mostly on a lexical level (missing delimiter, missing unit, wrong name, wrong mode, etc) with rare exceptions ("mistargeted style" for using properties incompatible with the selected node, overriding rules, etc) that were almost always resolved by students right away.

[4] http://dblp.uni-trier.de/search/publ?q=CSS$ -venue:CSS

[5] http://dblp.uni-trier.de/search/publ?q=cascading style sheets

In everyday web development life practitioners tend to use smell detection tools like CCS Lint [19], Codacy [16], CSS Nose [27] and W3C CSS Validator [94]. The latter concerns itself only with parse errors, CSS Lint can detect a list of predefined smells, Codacy is based on CSS Lint and is equivalent feature-wise, and CSS Nose is based on both CSS Lint and the W3C CSS Validator. Our work is partly motivated by extreme rigidity and lack of reconfigurability of these tools. There is a big gap between them and completely volatile tools such as SeeSS which assists users in identifying unintended visual changes by visualizing the impact of their CSS changes [52].

3.2 Coding Conventions

A project by Allamanis et al. [5] is among the most recent ones on coding conventions. They used natural language technology to detect conventions in source code — we have recently replicated their experiments on a different dataset with the same parameters, our conclusions supported the original findings [61]. For the project presented in this paper it means hope for future work on automatically inferring new conventions that are consistent with the codebase. One of the neighbour papers from this SLE showed how to successfully accomplish that for layout conventions [69].

There have been substantial advances in the field of *naming* conventions — that is, coding conventions concerning the names of variables and types. An example of such endeavours is the work of Butler et al. [11, 12] which classifies lemmas found in identifier names by parts of speech, eventually leading to better concept location. As shown by Linstead et al. [54], such naming conventions can be modelled with first-order Markov models and used for both classification and adherence verification with a reasonable degree of success. Going further down this road inevitably leads to topic modelling [31, 40, 75, 104]. Adjacent fields of research already have techniques that model users remarkably well [46, 97].

Another relatively active research area was formed around *calling* conventions [7] and produced modern software language-level techniques like staged allocation [64].

Perhaps in the future we should speak of *engineering conventions* or some other term for conventions of software language use, because the term "coding conventions" implies coding, but inexorably similar conventions are found in other areas such as software design [79] and model-driven engineering [51], as well as used to bind software artefacts of different kinds, such as source and metadata [84]. If that is achievable, we can connect to and profit from research on general conventions of collaborative work [66].

Coding conventions can be seen as a form of commitment to grammatical structure [47], and languages that express such commitments are not unheard of. They are usually specific to a technological space and come in forms of BNF dialects [99], metametamodels [63], database schema languages [38], etc. Specifically for coding conventions, there is no widespread consensus, even though some advocate viewing convention adherence as a metric that can show various degrees of success rather than a list of violations [83]. Such adherence is apparently not just structural, since it shows improvement if software processes become gamified [72].

The existing work on more flexible commitments to structure either still has a strong focus on parsing, the restrictions of which it tries to relax [87, 100], or on explicit modelling of uncertain aspects [22, 23, 86], or on consequences of flexibility on tool composition and pipelining [78, 101]. As already mentioned above, our work is different in the sense that we focus on *additional commitments* placed *on top of* another software language, with a property that they have no noticeable implications for parsing and execution but presumably strong relation to maintainability aspects. It remains to be seen if and how coding conventions can be implemented with language extensions — for now we are encapsulating them in an external DSL, convergence comes later.

4. Domain Analysis

In general, coding conventions is an umbrella term that comprises rules for whitespacing, comments, indentation, naming, syntax, code patterns, programming style, file organisation, etc. W3C, the primary organisation responsible for the specification of CSS, has not published any 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 coding conventions, we mined the 33,602 search results discovered in § 2. From each result only conventions that refer to plain CSS were taken into account, ignoring conventions related to "preprocessors" and the use of CSS classes from HTML. In cases when the result contained links to other style guides, those references were considered as results and analysed as well.

The search yielded quite a number of convention candidates that can be classified in the following groups:

Overgeneralisations: some statements positioned as conventions were in fact not conventions at all but rather high level guidelines lacking sufficient information to be applicable. Example: "*Do not use CSS hacks — try a different approach first*". These were omitted in our summary.

Contradictions: we often observed that conventions were explained in a natural language and exemplified with a code snippet. This is a common practice in software language documentation [103]. However, at times the natural language explanation contradicted the examples — in this case, we have chosen to assign higher priority to the examples.

Open interpretations: since code examples usually play the role of disambiguators, in their absence some conventions are open for interpretation. Example: "*Rules with more than 3 selectors are not allowed*". One interpretation could be forbidding multi-selectors with groups of more than three selectors (as in "`h1, h2, h3, h4 {color:red}`"). Another equally sensible interpretation is forbidding selectors with a combinator sequence of longer than three simple selectors (as in "`div table tr td {color:red}`"). The third interpretation is forbidding more than three type selectors or universal selectors within one sequence (as in, "`img[class~=a][src][alt] {border:0}`" which qualified as *one* simple selector in CSS 2.1 terminology [9] but is a *sequence* of four simple selectors in CSS 3 terminology [14], without any change in semantics). In such cases we have included all possible interpretations in our catalogue.

Underspecifications: in our paradigm, all guidelines in the form "you can do X under Y circumstances" imply that one does not do X usually, otherwise it would be pointless to specify when one can. In such cases we interpreted all implicit conventions explicitly. Example: *"You can put long values on multiple lines"* (implies that short values should be one-liners).

Style guides were large collections of CSS coding conventions used as more or less official guidelines in organisations and communities such as Mozilla [42], Google [29], GitHub [28], WordPress [93] and Drupal [21]. In total, 28 CSS style guides were discovered, containing 10–42 conventions for standalone guides or 5–10 conventions for those that were parts of larger style guides (covering also PHP, JS, HTML, etc). Since these style guides were written by professionals and had clear intentions driven by community needs or company interest, they provided the most comprehensive foundation for our collecting process — with the exception of convention candidates falling into the other four categories.

The total number of conventions we discovered was **471**. However, practitioners often share the same views and specify the same conventions in different style guides. As we found out, only one third of those conventions were unique. Thus, the result of the searches is **143** unique coding conventions appearing in CSS guidelines. The complete catalogue is huge and thus available online at `https://github.com/boryanagoncharenko/CssCoco/blob/master/analysis.md` or as Appendix G, but we do include some of the most popular conventions as examples below.

The conventions in our corpus are organised in groups depending on the exact type of constraints they impose. We have identified these three categories:

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

– Put one space between the colon and the value.

```
✓ .red {color: red;}
× .red {color:red;}
```

– One selector per line.

```
× img {border:0;} br {clear:left;}
  h1,h2 {text-align:center;}
```

– Put one space after the last selector.

```
✓ .red {color: red;}
× .red{color: red;}
```

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:

– HTML tags, class names and unquoted values should be lowercase.

```
✓ span.red {color: red;}
× SPAN.RED {COLOR: RED;}
```

– Put a ";" at the end of declarations.

```
✓ img {border:0;}
× img {border:0}
```

– Do not put quotes in `url()` declarations.

```
✓ body{background: url(recbg.jpg);}
× body{background: url("recbg.jpg");}
```

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 issues in a particular implementation. Examples are:

– Do not use ID selectors.

```
× p#first {font-weight: bold;}
✓ div > p:first-child {font-weight: bold;}
```

– Avoid qualifying ID and class names with type selectors.

```
✓  .red {color: red;}
× p.red {color: red;}
```

– When possible, use em instead of px.

```
✓ p {margin: 1em;}
× p {margin: 10px;}
```

As explained in the beginning of this section, our corpus consists of 143 entries written in this style:

Description: Disallow `@import`
Sources: CSS Lint (Nicholas C. Zakas, "Disallow import"), Real Deal ("CSS Naming Conventions and Coding Style"), Isobar ("Front-end Code Standards"), Code Guide (Mark Otto, "Code Guide")
Violations: For performance reasons, the usage of `@import` should be avoided. The following pattern is considered a violation: `@import url(foo.css);`
Actions: Find usage of `@import` statements

Analysis of this corpus led us to domain knowledge considered in more detail in subsequent sections, such as each convention being expressible by a combination of constraints, or using verbs `forbid` ("avoid", "do not") and `require` ("use", "prefer", "put") to formulate a convention. The three categories (layout, syntax, style) correspond to the data structures required to detect such conventions —

in terms of parsing in a broad sense, Ptr (parse tree), Cst (concrete syntax tree) and Ast (abstract syntax tree) [102]. Since we want to be able to detect all three, a parser was needed that delivers Ptrs; we used Gonzales [50]. Such consequences will be observed throughout § 5 – § 7).

5. Domain-Specific Ontology

Ontological analysis is a more or less established way of evaluating software notations [32, 60, 65, 70, 92]. It is based on the notion of an *ontology* as an explicit specification of a *conceptualisation*, which in turn is an abstract, simplified view of the world that is represented for some specific purpose [33]. An ontology 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 [59]. Based on the specificity of their constructs, ontologies can be top-level or domain-specific. Ontologies of the former type are highly general and provide the theoretical foundations for representation and modelling 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 [59].

The essence of ontological analysis can be explained in three steps: (1) designing a domain-specific ontology; (2) defining interpretation (notation to ontology) and representation (ontology to notation) mappings with the ontology as a reference point; (3) analysing the emerged anomalies and drawing conclusions about the quality of the notation. The anomalies can be of four kinds: *construct deficit*, when an ontological concept does not have a corresponding construct in the notation; *construct redundancy*, when a single ontological concept maps to more than one notational construct; *construct overload*, when a notational construct corresponds to more than one ontological concept; *construct excess*, when a concept in a notation does not map to any ontological concept [60].

Unlike traditional ontological analysis [32, 65, 92] requiring an already existing language and its use in practice, we apply it in an iterative *forward* software language engineering setup, as envisioned in the physics-of-notation methodology [60]. Thus, we create the ontology, map its concepts to abstract grammatical constructs and then give them concrete textual notation, falling into feedback loops whenever inconsistencies arise. The domain of the developed ontology is *detecting violations of CSS coding conventions* — hence, the designed ontology tries to capture only the concepts that exist when an agent searches a CSS stylesheet for violations of a given set of coding conventions.

The designed domain-specific ontology is based on the Bunge-Wand-Weber (BWW) top-level ontology [90], which is the leading ontology used for ontological analysis [60]. Our ontology uses the following high-level categories of the BWW ontology to describe the objects, concepts and entities in the specific domain: *Thing* (an elementary unit, composite or primitive), *Properties* (possessed by Things, can be intrinsic, emergent, hereditary or mutual), *State* (a vector with all Property values of a Thing), *Event* (a change of State), *Transformation* (mapping a set of States to a set of States), *History* (a trace of States that a Thing traverses), *Coupling* (two Things are Coupled if the existence of one affects the History of another), *Class* (a set of Things having a characteristic Property), *Subclass* (a set of Things within a Class having an additional Property), *System* (a set of Things that cannot be partitioned into two subsets without Couplings across them), *Composition* (all Things in a System), *Environment* (all Things outside the System that interact with its components). The graph of a system can be found on Figure 2.

The designed ontology was defined, as recommended by Wand and Weber, using a dictionary comprising definitions of entities in natural text, a BNF-like connection scheme, and additionally as a system diagram that demonstrates couplings (without using UML or ER as these modelling languages are subjects of ontological analysis themselves) [73, 91]. Following is a list with the main concepts discovered in the domain along with their descriptions. The used BWW concepts are in *italics* and the domain-specific concepts are 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. 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 code. Conventions are the building blocks of Style Guides. An individual Convention is a *composite thing* that contains a Context.

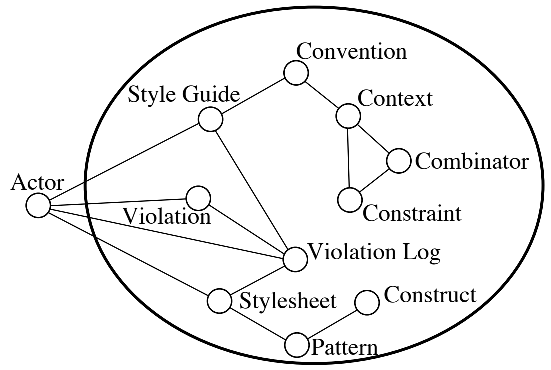
Intrinsic Property **Description** contains the reasoning behind the Convention.

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 **Constraint** or a **Constraint Combinator**. A violation is discovered when a Pattern in the current stylesheet fulfills all constraints specified by the Constraint or the Constraint Combinator.

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 Combinator** is an entity that connects logically Constraints or other Constraint Combinators. An



```
style_guide    ::= convention+ ;
convention     ::= context ;
context        ::= combinator | constraint ;
combinator     ::= (negation_combinator
                   | disjunction_combinator
                   | conjunction_combinator
                   | constraint)+ ;
violation_log  ::= violation* ;
stylesheet     ::= construct+ ;
pattern        ::= construct+ ;
```

Figure 2. *On the left*, a graph of the system: small circles represent BWW Things, lines show BWW Couplings, the big circle is BWW System. *On the right*, an abstract grammar linking the things together.

individual Constraint Combinator is a *composite thing* that comprises one or more logically related Constraints and/or Constraint Combinators.

Property **Number of Subjects** denotes the number of logically related Constraints and/or Combinators that are combined.

Property **Combinator Type** is the particular way the Constraints are combined.

Subclass **Negation Constraint Combinator** is a type of combinator that takes one Constraint or Combinator and returns the opposite Constraint or Combinator. An individual Negation Constraint Combinator is a *composite thing* that comprises one Constraint or Combinator.

Property **Number of Subjects** denotes the number of logically related Constraints and/or Combinators that are combined. In the case of the Negation Constraint Combinator, the Number of Subjects property is equal to one.

Property **Combinator Type** is the particular way the Constraints are combined. Specifically, this type of combinator negates the Constraint or Combinator it takes.

Subclass **Disjunction Constraint Combinator** is a type of combinator that takes two or more Constraints or Combinators and combines them using the OR logical operator. An individual Disjunction Constraint Combinator is a *composite thing* that comprises two or more subjects.

Property **Number of Subjects** denotes the number of logically related Constraints and/or Combinators that are combined.

Property **Combinator Type** is the particular way the Constraints are combined. Specifically, this type of combinator states that at least one of the Constraints it combines need to be fulfilled.

Subclass **Conjunction Constraint Combinator** is a type of combinator that takes two or more Constraints or Combinators and combines them using the AND logical operator. An individual Conjunction Constraint Combinator is a *composite thing* that comprises two or more subjects.

Property **Number of Subjects** denotes the number of logically related Constraints and/or Combinators that are combined.

Property **Combinator Type** is the particular way the Constraints are combined. Specifically, this type of combinator states that all of the Constraints it combines need to be fulfilled.

Class **Constraint** is a specific restriction that needs to be fulfilled. Constraints 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 that 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 that is being constrained.

Property **Requirement** denotes the particular limitation applied to the Subject. Specifically, the requirement is 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 that is being constrained.

Property **Requirement** denotes the particular limitation applied to the Subject. Specifically, the requirement is 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 that is being constrained.

Property **Requirement** denotes the particular limitation applied to the Subject. Specifically, the requirement is 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 that is being constrained.

Property **Requirement** denotes the particular limitation applied to the Subject. Specifically, the requirement is 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 that is being constrained.

Property **Requirement** denotes the particular limitation applied to the Subject. Specifically, the requirement is 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** 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 that matches the description of a Context. An instance of a Pattern is a composite thing built from one or many Constructs.

Property **Number of 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 of the Stylesheet is altered. 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 and a Violation is recorded in the Violation Log. The state of the Violation contains 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 not checked 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 not checked for compliance.

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

Most of the definitions in the ontology refer to concepts that appear in the coding conventions domain. When *not* viewed as a domain model, the ontology is certainly extensible: e.g., a Style Guide for us is just a collection of coherent conventions, but it can be assigned intrinsic properties such as authorship. The ontological concept of a Convention differs slightly from the intuitive one: 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 such as forbid (prohibit a pattern) and require (impose use limitations on a pattern). The meaning of a Convention is always expressed through the possible violations of that Convention. A Context aims at describing the whole violation pattern and consists of a single Constraint or a number of logically related Constraints (requirements to be fulfilled).

The grammar in Figure 2 illustrates that a Style Guide needs to contain one or more Conventions. A Convention consists of a Context, which in turn, comprises either a Combinator or a Constraint. Because a Context describes the whole pattern that is considered a violation, it can be expressed with a single Constraint or a combination of logically related Constraints. A Combinator is a recursive construct that can comprise Constraints or other Combinators. Different subclasses of Combinator have different constraints on the number of subjects they combine (one or two). A Violation Log 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 one or more Constructs.

Now we complement the composition model given in grammar form with an interaction model as a system graph in Figure 2. According to the theory of ontological models of information systems, 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 [90]. For example, a coupling exists between the Style Guide and the Convention things, because the existence of a Convention alters the state of the Style Guide. Each edge on the system graph is drawn following similar argumentation — we will spare the details here.

6. Language Syntax

For ontologically analysing already existing languages it is usual to construct two mappings: a *representation* matching the ontology and assigning notational elements to concepts and an *interpretation* doing the opposite. Since we were still in control of the language design, these could be developed bidirectionally and straightforwardly (cf. Table 1). We can avoid both *construct overload* and *excess* by construction. In fact, we only had to use one property indicating construct *deficit*: 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, so its support is left to the environment as well.

We have some seeming *redundancy*. For instance, the Conventions property of Style Guide is matched to Contexts property of Convention Set and the Conventions property of Context. However, this is required since the modelling grammar groups Conventions that share the same Ignored Constructs in a Context. Thus, a Style Guide in the modelling grammar does not possess conventions but a number of Contexts that, in turn, contain conventions. In this sense, the coexistence of properties Contexts and Conventions represents the concept of Conventions. Other cases of redundancy have similar motivation. Mapping a single ontological concept to a combination of modelling grammar constructs is an accepted approach and has been used in multiple studies [25].

After having designed the abstract syntax for our language, we have implemented it in Python as an `AstNode` and other classes forming a hierarchy below it, 80 classes in total. We do not include any description of this code, but it is available for inspection in raw form as http://github.com/boryanagoncharenko/CssCoco/blob/master/csscoco/lang/ast/ast.py or as a collection of annotated class diagrams in Appendix D. In general it is exactly what one may expect of the object-oriented design of an abstract syntax of a domain-specific language.

```
stylesheet ::= context* ;
context ::= Id ignore_clause? '{' convention* '}' ;
ignore_clause ::= 'ignore' (node_desc)+ (',' (node_desc)+)* ;
convention ::= 'forbid' pattern 'message' Str
            | 'find' pattern ('where' bexpr)?
             ('require'|'forbid') bexpr 'message' Str ;
pattern ::= node_decl (('in' | 'next-to') node_decl)*
         | fork ('in' node_decl)* ;
fork ::= '(' node_decl (',' node_decl)+ ')' ;
node_decl ::= (Id '=')? semantic_node ;
node_desc ::= 'unique'? node_type ('{' (bexpr|repeater) '}')?;
repeater ::= Int ',' Int? | (',')? Int ;
bexpr ::= '(' bexpr ')' | aexpr | type_expr | 'not' bexpr
       | bexpr bool_op bexpr ;
type_expr ::= aexpr operator='is' Id
           | node_desc+ ('before' | 'after') type_arg
           | node_desc+ 'between' type_arg 'and' type_arg ;
type_arg ::= Id | semantic_node ;
aexpr ::= ('-'|'+') aexpr | cexpr | element
       | aexpr cmp_op aexpr | aexpr match_op aexpr ;
bool_op ::= 'and' | 'or' ;
cmp_op ::= '<' | '>' | '<=' | '>=' | '==' | '!=' ;
match_op ::= 'in' | 'not in' | 'match' | 'not match' ;
element ::= Bool | Real | Int | Dec | Str | list ;
cexpr ::= cexpr '.' cexpr
       | Id ('(' (element | semantic_node ) ')')? ;
list ::= '[' list_el (',' list_el)* ']' ;
list_el ::= Int | Real | Str | semantic_node ;
node_type ::= '(' node_type ')' | 'not' node_type
           | node_type bool_op node_type | Id ;
```

Figure 3. A (simplified) concrete syntax grammar in EBNF — its operational ANTLR4 counterpart is located at http://github.com/boryanagoncharenko/CssCoco/blob/master/csscoco/lang/syntax/coco.g4 and documented as Appendix E.

Mapping abstract syntax of CssCoco to concrete textual syntax is done similarly, the shortened grammar of the result is presented on Figure 3, an annotated complete version available as Appendix E.

7. Validation

To study the feasibility of the designed domain-specific language, a *proof of concept* was developed. The implemented solution consists of two parts: a standalone Python 3.4 package and a plugin for Sublime Text 3 editor.

The first part of the designed solution is the CssCoco interpreter. It is implemented in almost 9000 lines of Python code. The source code is open and publicly available: http://github.com/boryanagoncharenko/CssCoco. The solution was also added to the Python Package Index (PyPi) repository https://pypi.python.org/pypi/csscoco where it received over 5000 downloads in the first month and was enjoying around a hundred downloads a day after that until PyPi shut down their statistics collection in February 2016. The package offers a `csscoco` command that takes as arguments a `.css` file and a `.coco` file and returns a list of the discovered violations. The package requires Node.js because it has a dependency on a previously existing CSS parser [50].

The second part of the proof of concept integrates the functionality implemented in the Python package into Sub-

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 Combinator	Not, Or, And Expression
Number of Subjects (Constraint Combinator)	Operand of Combinator Expressions
Negation Constraint Combinator	Not Expression
Disjunction Constraint Combinator	Or Expression
Conjunction Constraint Combinator	And Expression
Constraint	Comparison, Is, In, Match, Node Query Expressions
Subject (Constraint)	Operand of Constraint Expressions
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
Number of Constructs (Pattern)	Nodes (CSS Pattern)

Table 1. Representation/interpretation mapping between the domain-specific ontology and the abstract modelling grammar.

lime Text editor. The plugin uses the `csscoco` command to find violations in CSS files that are being edited in the text editor. The source code of the solution is also publicly available at a separate GitHub repository `http://github.com/boryanagoncharenko/Sublime-CssCoco`. The plugin offers a command that finds violations. Similarly to other linter tools, lines that contain violations are marked with a coloured border and a gist appearing 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 4 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. Installing the plugin allows users to edit `.coco` files, easily connect or disconnect them and hook the CssCoco linter on save file action. Several examples of how convention definitions look like, follow.

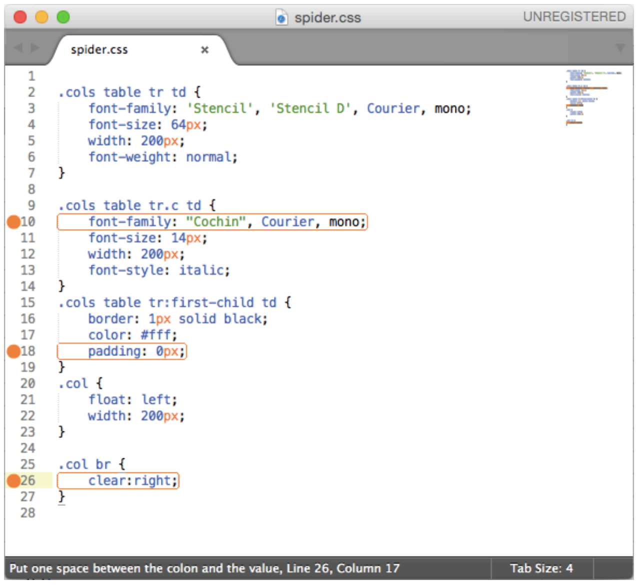
```
forbid import
message 'No import statements'
```

This is one of the simplest constructions, which expresses disallowing any `@import` statements. For more complex cases we need a find/require construct:

```
find c=colorname
require c.string match lowercase
message 'Colours should be lowercase'
```

Here we tell CssCoco to find all standard colour names in all places in a stylesheet where a colour name would be appropriate, and assert that its string value matches a preset regex for lowercase. The specification looks similar when several places need to be matched:

```
find r1=ruleset r2=ruleset
require newline{2} between r1 and r2
message 'Separate style definitions!'
```



```
 1
 2  .cols table tr td {
 3      font-family: 'Stencil', 'Stencil D', Courier, mono;
 4      font-size: 64px;
 5      width: 200px;
 6      font-weight: normal;
 7  }
 8
 9  .cols table tr.c td {
10      font-family: "Cochin", Courier, mono;
11      font-size: 14px;
12      width: 200px;
13      font-style: italic;
14  }
15  .cols table tr:first-child td {
16      border: 1px solid black;
17      color: #fff;
18      padding: 0px;
19  }
20  .col {
21      float: left;
22      width: 200px;
23  }
24
25  .col br {
26      clear:right;
27  }
28
```

```
Put one space between the colon and the value, Line 26, Column 17        Tab Size: 4
```

Figure 4. A development environment open with the CssCoco plugin activated and reporting three violations: double quotes instead of single quotes; using units of measurements with zero values; and the lack of space between the colon and the value. The message about the latter violation is displayed in the status bar because the cursor is on the offending line.

Expressions in conditions can be more complex. For instance, in the next convention we search for all declarations, then descend to their last child and compare its string representation with the expectation:

```
find d=declaration
require d.child(-1).string == ';'
message 'Missed final semicolon'
```

The proof of concept implementation successfully covers all conventions found in our corpus, except the following four kinds:

- *uniqueness constraints* (can be awkwardly emulated by matching two arbitrary elements and forbidding them to be equal);
- *ordering constraints* (conventions like "order selectors by type" are tightly coupled to the CSS grammar and requires convergence of the standard W3C grammar with the one used by the chosen parser);
- *context-dependent indentation constraints* (requires results from pretty-printing research on the language design level and maintaining special contextual information about the indentation levels on the implementation level);
- *vocabulary* (conventions like "do not abbreviate" require a natural language dictionary and a robust stemmer).

Implementing all these is also possible, yet goes well beyond a proof of concept and redirects the focus elsewhere. Thus, it was left for future work.

8. Conclusion and Discussion

We have thoroughly analysed the domain of coding conventions for CSS, adherence to them and detecting their violations. We have established the need for our research by performing search engine queries, analysing public repository commits and consuming all available relevant scientific literature. Then, we have modelled the domain with a domain-specific ontology, designed an abstract syntax by representing that ontology in a demonstrably clear and complete way, refined the outcome to a suitable textual concrete syntax capable of expressing domain constructs, and finally implemented the CssCoco language in a prototype tool. It was released as an open source project several months prior to submission of this paper. It works as a command line tool or integrates into Sublime Text, and illustrates that the suggested approach enables automatic detection of violations of user-specified conventions.

One of the threats to internal validity is selection bias: we confined ourselves to results provided by systems like Privatelee and GitHub; in § 2 we considered only CSS "preprocessors" known to us — this may skew the results especially for future replications. With respect to construct validity, we tried to avoid hypothesis guessing by stating that coding conventions in general and in CSS in particular are being *discussed* and not necessarily *desirable* — even though we have certain hypotheses based on surveys of programmers about code smells [96]. The obvious threat to external validity concerns the ability of the language as we have designed it, to express all coding conventions that CSS writers could ever come up with — this can only be addressed by iterative language design and is per definition future work. On a similar note, our tool detects refactoring *opportunities* [57] and we could investigate how to act on them, thus enforcing conventions.

There are some conceptual questions that can be asked based on our results but are left unanswered by them. For instance, the CssCoco language that we have designed and implemented, allows its users to specify custom rules for detecting coding convention violations — but is the need for having custom conventions inherent to CSS as a language or just to the current state of (im)maturity of the web developing community? Could the existing linters, perhaps with a few extensions, reach a point of widespread acceptance? Different existing language communities show drastically different behaviours, from Go that deploys a standard pretty-printer and declares it a sin to not use it; to C++ with a decades long unending holy war on the rightful place of the opening curly brace.

Software *language* engineers do not need to be convinced that creating a domain-specific language is a solid approach to solve a problem (any problem, really). However, the real consequences and tradeoffs of pursuing that path, each time need to be considered and communicated to software engineers and industrial clients. In our case, weighing the costs and profits of having a language to express conventions as opposed to formulating coding conventions as constraints, patterns or traversals on top of some general purpose metaprogramming facility, has been left out of scope for our project. Any reasonably advanced language workbench with the possibility of defining hierarchical algebraic data types and transforming or constraining them, could have been used here: Rascal, Spoofax, TXL, UML+OCL, EMF+IncQuery, srcML+XSLT, etc. This is an interesting implementational discussion seemingly strengthening the validation but ironically being of no consequence at all to the end users who care about the resulting tool's capability of being integrated into their existing workspaces.

This project, among other things, can be seen as an exercise in applying the methodology of *physics of notation* [60], in particular the ontological analysis [32, 65, 70, 90, 92], to the process of designing a domain-specific language with its domain explicitly encoded as an ontology and verifying that artefacts of other levels: the metamodel (abstract syntax) and the textual notation (concrete syntax) — are capable of representing the domain concepts clearly and completely, without deficit, redundancy, excess or overload of constructs. We see the endeavour as a contribution to the software language engineering discipline [47], and hope its degree of success or failure (determining which is left as an exercise to the readers) will help to shine more light on the methodology as a whole. The fact that by far the most cited paper ever produced within the SLE / LDTA / ATEM / WAGA community [60], is never used in day to day practice of software language engineering, is disconcerting.

Paper supplements containing datasets for claims from § 2, the complete list of prior related work on CSS with DOI links (the classification of them as well as the highlights were presented in § 3.1), the complete set of UML class diagrams with accompanying documentation, as well as the original non-shortened version of the grammar we have shown in Figure 3 with annotations per nonterminal, have been deposited as `http://dx.doi.org/10.6084/m9.figshare.3085831.v3`.

References

[1] C. F. Acebal, B. Bos, M. Rodríguez, and J. M. C. Lovelle. ALMcss: a Javascript Implementation of the CSS Template Layout Module. In *ACM Symposium on Document Engineering (DocEng)*, pages 23–32. ACM, 2012.

[2] A. Adewumi, S. Misra, and N. Ikhu-Omoregbe. Complexity Metrics for Cascading Style Sheets. In *12th International Conference on Computational Science and its Applications (ICCSA)*, volume 7336 of *LNCS*, pages 248–257. Springer, 2012.

[3] A. Aizpurua, M. Arrue, M. Vigo, and J. Abascal. Exploring Automatic CSS Accessibility Evaluation. In *Ninth International Conference on Web Engineering (ICWE)*, volume 5648 of *LNCS*, pages 16–29. Springer, 2009.

[4] K. Alabi. Generation, Documentation and Presentation of Mathematical Equations and Symbolic Scientific Expressions Using Pure HTML and CSS. In *16th International Conference on World Wide Web (WWW)*, pages 1321–1322. ACM, 2007.

[5] M. Allamanis, E. T. Barr, C. Bird, and C. A. Sutton. Learning Natural Coding Conventions. In *22nd Symposium on the Foundations of Software Engineering (FSE)*, pages 281–293. ACM, 2014.

[6] G. J. Badros, A. Borning, K. Marriott, and P. J. Stuckey. Constraint Cascading Style Sheets for the Web. In *12th Annual ACM Symposium on User Interface Software and Technology (UIST)*, pages 73–82. ACM, 1999.

[7] M. W. Bailey and J. W. Davidson. A Formal Model of Procedure Calling Conventions. In *Conference Record of the 22nd Symposium on Principles of Programming Languages*, pages 298–310. ACM, 1995.

[8] M. S. Bier and B. Diri. Defect Prediction for Cascading Style Sheets. *Applied Soft Computing*, 2016. In press, corrected proof, available online 30 May 2016.

[9] 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`.

[10] M. Bosch, P. Genevs, and N. Layada. Automated Refactoring for Size Reduction of CSS Style Sheets. In *ACM Symposium on Document Engineering (DocEng)*, pages 13–16. ACM, 2014.

[11] S. Butler. Mining Java Class Identifier Naming Conventions. In *34th International Conference on Software Engineering (ICSE)*, pages 1641–1643. IEEE, 2012.

[12] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Mining Java Class Naming Conventions. In *27th Conference on Software Maintenance (ICSM)*, pages 93–102. IEEE, 2011.

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

[14] T. Çelik, E. J. Etemad, D. Glazman, I. Hickson, P. Linss, and J. Williams. Cascading Style Sheets (CSS) Selectors Level 3. *W3C Recommendation*, Sept. 2011.

[15] Y. Chou and H. Liao. A Webpage Data Hiding Method by Using Tag and CSS Attribute Setting. In *Tenth International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP)*, pages 122–125. IEEE, 2014.

[16] Codacy. Patterns list. `https://www.codacy.com/patterns`.

[17] A. Cogliati, M. Pohja, and P. Vuorimaa. XHTML and CSS Components in an XML Browser. In *International Conference on Internet Computing (IC), Volume 2*, pages 563–572. CSREA Press, 2003.

[18] A. Cogliati and P. Vuorimaa. Optimized CSS Engine. In *Second International Conference on Web Information Systems and Technologies: Internet Technology / Web Interface and Applications (WEBIST)*, pages 206–213. INSTICC

Press, 2006.

[19] CSSLint. Rules. https://github.com/CSSLint/csslint/wiki/Rules.

[20] S. Culshaw, M. Leventhal, and M. Maloney. XML and CSS. *World Wide Web Journal*, 2(4):109–118, 1997.

[21] Drupal. CSS Coding Standards. https://www.drupal.org/node/1886770.

[22] R. Eramo, A. Pierantonio, and G. Rosa. Managing Uncertainty in Bidirectional Model Transformations. In *Eighth International Conference on Software Language Engineering (SLE)*, pages 49–58. ACM, 2015.

[23] M. Famelis, R. Salay, and M. Chechik. Partial Models: Towards Modeling and Reasoning with Uncertainty. In *34th International Conference on Software Engineering*, pages 573–583. IEEE, 2012.

[24] A. M. García, P. De Bra, G. H. L. Fletcher, and M. Pechenizkiy. A DSL Based on CSS for Hypertext Adaptation. In *25th Conference on Hypertext and Social Media (HT)*, pages 313–315. ACM, 2014.

[25] A. Gehlert and W. Esswein. Toward a Formal Research Framework for Ontological Analyses. *Advanced Engineering Informatics*, 21(2):119–131, 2007.

[26] P. Genevès, N. Layaïda, and V. Quint. On the Analysis of Cascading Style Sheets. In *21st World Wide Web Conference (WWW)*, pages 809–818. ACM, 2012.

[27] G. Gharachorlu. Code Smells in Cascading Style Sheets: An Empirical Study and a Predictive Model. Master's thesis, University of British Columbia, 2014.

[28] GitHub. Guidelines — Primer. http://primercss.io/guidelines/#css.

[29] E. Glaysher. HTML/CSS Style Guide. https://google-styleguide.googlecode.com/svn/trunk/htmlcssguide.xml.

[30] G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman. Lean GHTorrent: GitHub Data on Demand. In *11th Working Conference on Mining Software Repositories (MSR)*, pages 384–387. ACM, 2014.

[31] S. Grant and J. R. Cordy. Examining the Relationship Between Topic Model Similarity and Software Maintenance. In *Software Evolution Week: Conference on Software Maintenance, Reengineering, and Reverse Engineering*, pages 303–307. IEEE CS, 2014.

[32] P. Green and M. Rosemann. Integrated Process Modeling: an Ontological Evaluation. *Information Systems*, 25(2):73–87, 2000.

[33] T. R. Gruber. Toward Principles for the Design of Ontologies Used for Knowledge Sharing. *International Journal of Human-Computer Studies. Special Issue on the Role of Formal Ontology in the Information Technology*, 43(5-6):907–928, Dec. 1995.

[34] K. Gyllstrom, C. Eickhoff, A. P. de Vries, and M.-F. Moens. The Downside of Markup: Examining the Harmful Effects of CSS and Javascript on Indexing Today's Web. In *21st ACM International Conference on Conference on Information and Knowledge Management (CIKM)*, pages 1990–1994. ACM, 2012.

[35] M. Hague, A. W. Lin, and C.-H. L. Ong. Detecting Redundant CSS Rules in HTML5 Applications: a Tree Rewriting Approach. In *30th Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 1–19. ACM, 2015.

[36] S. Harper, S. Bechhofer, and D. Lunn. SADIe: Transcoding based on CSS. In *Eighth International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS)*, pages 259–260. ACM, 2006.

[37] M. Hevery and A. Abrons. Declarative Web-Applications without Server: Demonstration of How a Fully Functional Web-Application Can Be Built in an Hour with only HTML, CSS & Javascript Library. In *OOPSLA Companion*, pages 801–802. ACM, 2009.

[38] J.-M. Hick and J.-L. Hainaut. Database Application Evolution: A Transformational Approach. *Data & Knowledge Engineering*, 59(3):534–558, Dec. 2006.

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

[40] J. Hu, X. Sun, D. Lo, and B. Li. Modeling the Evolution of Development Topics Using Dynamic Topic Models. In *22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 3–12. IEEE, 2015.

[41] L. Huang, Z. Weinberg, C. Evans, and C. Jackson. Protecting Browsers from Cross-origin CSS Attacks. In *17th Conference on Computer and Communications Security (CCS)*, pages 619–629. ACM, 2010.

[42] D. Hyatt. Guidelines for Efficient CSS, 2000. https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Writing efficient CSS.

[43] R. D. Johansen, T. C. P. Britto, and C. A. Cusin. CSS Browser Selector Plus: A JavaScript Library to Support Cross-browser Responsive Design. In *Companion Volume of the 22nd International World Wide Web Conference (WWW)*, pages 27–30. ACM, 2013.

[44] M. Keller and M. Nussbaumer. Cascading Style Sheets: A Novel Approach Towards Productive Styling With Today's Standards. In *18th International Conference on World Wide Web (WWW)*, pages 1161–1162. ACM, 2009.

[45] M. Keller and M. Nussbaumer. CSS Code Quality: A Metric for Abstractness; Or Why Humans Beat Machines in CSS Coding. In *Seventh International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 116–121. IEEE Computer Society, 2010.

[46] D. Kelly, F. Daz, N. J. Belkin, and J. Allan. A User-Centered Approach to Evaluating Topic Models. In *26th European Conference on Information Retrieval Research: Advances in Information Retrieval (ECIR)*, volume 2997 of *LNCS*, pages 27–41. Springer, 2004.

[47] P. Klint, R. Lämmel, and C. Verhoef. Toward an Engineering Discipline for Grammarware. *ACM Transactions on Software Engineering Methodology (TOSEM)*, 14(3):331–380, 2005.

[48] J. K. Korpela. Lurching Toward Babel: HTML, CSS, and XML. *IEEE Computer*, 31(7):103–104, 1998.

[49] R. Kotcher, Y. Pei, P. Jumde, and C. Jackson. Cross-origin Pixel Stealing: Timing Attacks Using CSS Filters. In *SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1055–1062. ACM, 2013.

[50] S. Kryzhanovsky. Gonzales 1.0.7 — Fast CSS Parser, 2012. http://github.com/css/gonzales, MIT License.

[51] C. F. J. Lange, B. Du Bois, M. R. V. Chaudron, and S. Demeyer. An Experimental Investigation of UML Modeling Conventions. In *Ninth International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 4199 of *LNCS*, pages 27–41. Springer, 2006.

[52] H.-S. Liang, K.-H. Kuo, P.-W. Lee, Y.-C. Chan, Y.-C. Lin, and M. Y. Chen. SeeSS: Seeing What I Broke — Visualizing Change Impact of Cascading Style Sheets. In *26th annual ACM symposium on User Interface Software and Technology (UIST)*, pages 353–356, 2013.

[53] H. W. Lie and J. Saarela. Multipurpose Web Publishing Using HTML, XML, and CSS. *Communications of the ACM*, 42(10):95–101, 1999.

[54] E. Linstead, L. Hughes, C. V. Lopes, and P. Baldi. Capturing Java Naming Conventions with First-Order Markov Models. In *17th International Conference on Program Comprehension*, pages 313–314. IEEE CS, 2009.

[55] J. Marszalkowski, J. Mizgajski, D. Mokwa, and M. Drozdowski. Analysis and Solution of CSS-Sprite Packing Problem. *ACM Transactions on the Web*, 10(1):1, 2016.

[56] D. Mazinanian and N. Tsantalis. An Empirical Study on the Use of CSS Preprocessors. In *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 168–178. IEEE Computer Society, 2016.

[57] D. Mazinanian, N. Tsantalis, and A. Mesbah. Discovering Refactoring Opportunities in Cascading Style Sheets. In *22nd Symposium on the Foundations of Software Engineering (FSE)*, pages 496–506. ACM, 2014.

[58] A. Mesbah and S. Mirshokraie. Automated Analysis of CSS Rules to Support Style Maintenance. In *34th International Conference on Software Engineering (ICSE)*, pages 408–418. IEEE, 2012.

[59] S. K. Milton and B. Smith. Top-level Ontology: The Problem with Naturalism. *Formal Ontology in Information Systems*, pages 85–94, 2004.

[60] D. L. Moody. The Physics of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, 35(6):756–779, 2009.

[61] D. C. Moya. NATURALIZE: A Replication Study. Master's thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, Aug. 2015.

[62] J. V. Nickerson. Visual Conventions for System Design Using Ada 9X: Representing Asynchronous Transfer of Control. In *Conference Proceedings on TRI-Ada 1993*, pages 379–384. ACM, 1993.

[63] Object Management Group. *Meta-Object Facility (MOF™) Core Specification*, 2.5 edition, 2015. http://www.omg.org/spec/MOF/2.5.

[64] R. Olinsky, C. Lindig, and N. Ramsey. Staged Allocation: a Compositional Technique for Specifying and Implementing Procedure Calling Conventions. In *33rd Symposium on Principles of Programming Languages*, pages 409–421. ACM, 2006.

[65] A. L. Opdahl and B. Henderson-Sellers. Ontological Evaluation of the UML Using the Bunge–Wand–Weber Model. *Software and Systems Modeling*, 1(1):43–67, 2002.

[66] U. Pankoke-Babatz, K. Klckner, and P. Jeffrey. Norms and Conventions in Collaborative Systems. In *Eighth International Conference on Human-Computer Interaction (HCI). Volume 2: Communication, Cooperation, and Application Design (CCAD)*, pages 313–317. Lawrence Erlbaum, 1999.

[67] T. H. Park, B. Dorn, and A. Forte. An Analysis of HTML and CSS Syntax Errors in a Web Development Course. *ACM Transactions on Computing Education (TOCE)*, 15(1):4, 2015.

[68] T. H. Park, A. Saxena, S. Jagannath, S. Wiedenbeck, and A. Forte. Towards a Taxonomy of Errors in HTML and CSS. In *Ninth International Computing Education Research Conference (ICER)*, pages 75–82. ACM, 2013.

[69] T. Parr and J. Vinju. Towards a Universal Code Formatter through Machine Learning. In *Ninth International Conference on Software Language Engineering (SLE)*. ACM, 2016. In print. Pre-print at http://arxiv.org/abs/1606.08866v1.

[70] J. Parsons and Y. Wand. Using Objects for Systems Analysis. *Communications of the ACM*, 40(12):104–110, 1997.

[71] M. Pohja and P. Vuorimaa. CSS Layout Engine for Compound Documents. In *Third Latin American Web Congress (LA-Web)*, pages 148–156. IEEE Computer Society, 2005.

[72] C. R. Prause and M. Jarke. Gamification for Enforcing Coding Conventions. In *10th Joint Meeting of the 15th European Software Engineering Conference and the 23rd Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 649–660. ACM, 2015.

[73] M. Rosemann and P. Green. Developing a Meta Model for the Bunge–Wand–Weber Ontological Constructs. *Information Systems*, 27(2):75–91, 2002.

[74] H. Rouzati, L. Cruiz, and B. MacIntyre. Unified WebGL/CSS Scene-graph and Application to AR. In *18th International Conference on Web3D Technology*, page 210. ACM, 2013.

[75] A. Saeidi, J. Hage, R. Khadka, and S. Jansen. ITMViz: Interactive Topic Modeling for Source Code Analysis. In *IEEE 23rd International Conference on Program Comprehension (ICPC)*, pages 295–298. ACM, 2015.

[76] F. Saint-Jean, A. Johnson, D. Boneh, and J. Feigenbaum. Private web search. In *Workshop on Privacy in Electronic Society (WPES)*, pages 84–90. ACM, 2007.

[77] A. Sampson, C. Cascaval, L. Ceze, P. Montesinos, and D. S.

Gracia. Automatic Discovery of Performance and Energy Pitfalls in HTML and CSS. In *International Symposium on Workload Characterization (IISWC)*, pages 82–83. IEEE Computer Society, 2012.

[78] L. Sassaman, M. L. Patterson, and S. Bratus. A Patch for Postel's Robustness Principle. *IEEE Security and Privacy*, 10(2):87–91, Mar. 2012.

[79] A. Schroff and A. Teichrieb. Conventions for the Practical Use of UML. In *The UML — Technical Aspects and Applications*, pages 262–270. Physica-Verlag, 1997.

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

[81] M. Serrano. HSS: a Compiler for Cascading Style Sheets. In *12th International Conference on Principles and Practice of Declarative Programming (PPP)*, pages 109–118. ACM, 2010.

[82] D. Shea. CSS Zen Garden. In *31st International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH), Web Graphics*, page 18. ACM, 2004.

[83] M. Smit, B. Gergel, H. J. Hoover, and E. Stroulia. Code Convention Adherence in Evolving Software. In *27th Conference on Software Maintenance (ICSM)*, pages 504–507. IEEE, 2011.

[84] M. Song and E. Tilevich. Metadata Invariants: Checking and Inferring Metadata Coding Conventions. In *34th International Conference on Software Engineering (ICSE)*, pages 694–704. IEEE, 2012.

[85] H. Stormer. Personalized Websites for Mobile Devices using Dynamic Cascading Style Sheets. *IJWIS*, 1(2):83–88, 2005.

[86] G. Succi, F. Baruchelli, and M. Ronchetti. A Taxonomy for Identifying a Software Component for Uncertain and Partial Specifications. In *11th Symposium on Applied Computing (SAC)*, pages 570–579. ACM, 1996.

[87] F. Sur. Robust Matching in an Uncertain World. In *20th International Conference on Pattern Recognition (ICPR)*, pages 2350–2353. IEEE CS, 2010.

[88] J. Sutter, K. Sons, and P. Slusallek. A CSS Integration Model for Declarative 3D. In *20th International Conference on 3D Web Technology (Web3D)*, pages 209–217. ACM, 2015.

[89] N. Takei, T. Saito, K. Takasu, and T. Yamada. Web Browser Fingerprinting Using Only Cascading Style Sheets. In *10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA)*, pages 57–63. IEEE Computer Society, 2015.

[90] Y. Wand and R. Weber. An Ontological Model of an Information System. *IEEE Transactions on Software Engineering*, 16(11):1282–1292, 1990.

[91] Y. Wand and R. Weber. On the Deep Structure of Information Systems. *Information Systems Journal*, 5(3):203–223, 1995.

[92] R. Weber and Y. Zhang. An Analytical Evaluation of NIAM's grammar for Conceptual Schema Diagrams. *Information Systems Journal*, 6(2):147–170, 1996.

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

[94] World Wide Web Consortium. CSS Validation Service. http://jigsaw.w3.org/css-validator.

[95] D. Wu and H. Su. Information Hiding in EPUB Files by Rearranging the Contents of CSS Files. In *Ninth International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP)*, pages 80–83. IEEE, 2013.

[96] A. F. Yamashita and L. Moonen. Do Developers Care about Code Smells? An Exploratory Survey. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 242–251. IEEE, 2013.

[97] Z. Yang, A. Kotov, A. Mohan, and S. Lu. Parametric and Non-parametric User-aware Sentiment Topic Models. In *38th International SIGIR Conference on Research and Development in Information Retrieval*, pages 413–422. ACM, 2015.

[98] J. Zakraoui and W. L. Zagler. A Method for Generating CSS to Improve Web Accessibility for Old Users. In *13th International Conference on Computers Helping People with Special Needs (ICCHP), Part I*, volume 7382 of *LNCS*, pages 329–336. Springer, 2012.

[99] V. Zaytsev. BNF WAS HERE: What Have We Done About the Unnecessary Diversity of Notation for Syntactic Definitions. In *27th ACM Symposium on Applied Computing, Programming Languages Track (SAC/PL 2012)*, pages 1910–1915, 2012.

[100] V. Zaytsev. Formal Foundations for Semi-parsing. In *Software Evolution Week: Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 313–317. IEEE CS, 2014.

[101] V. Zaytsev. Taxonomy of Flexible Linguistic Commitments. In *Workshop on Flexible Model-Driven Engineering (FlexMDE)*, volume 1470 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2015.

[102] V. Zaytsev and A. H. Bagge. Parsing in a Broad Sense. In *17th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 8767 of *LNCS*, pages 50–67. Springer, 2014.

[103] V. Zaytsev and R. Lämmel. A Unified Format for Language Documents. In *Post-Third International Conference on Software Language Engineering (SLE 2010)*, volume 6563 of *LNCS*, pages 206–225. Springer, Jan. 2011.

[104] C. Zou and D. Hou. LDA Analyzer: A Tool for Exploring Topic Models. In *30th International Conference on Software Maintenance and Evolution (ICSME)*, pages 593–596. IEEE, 2014.