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
to be available any more. According to the Private Search Engine List\textsuperscript{1} comparative analysis, its reasonable contemporary substitute that aggregates results acquired anonymously from several engines based on ratings of Web of Trust, is Privatelee\textsuperscript{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\textsuperscript{3} to access GitHub’s public 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, .less, or .sta 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

---

\textsuperscript{1}Private Search Engine, http://www.privatesearchengine.com
\textsuperscript{2}Privatelee, https://privatelee.com
\textsuperscript{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.
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” and “cascading style sheets” 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].

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 (“mistrargeted style” for using properties incompatible with the selected node, overriding rules, etc) that were almost always resolved by students right away.
In everyday web development life practitioners tend to use smell detection tools like CSS 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 Linstedt 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\}\)
for this reason, 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\}\)
for this reason, 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 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;\}
- Avoid qualifying ID and class names with type selectors.
  - .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:

<table>
<thead>
<tr>
<th>Description</th>
<th>Disallow @import</th>
</tr>
</thead>
<tbody>
<tr>
<td>Violations</td>
<td>For performance reasons, the usage of @import should be avoided. The following pattern is considered a violation: @import url(foo.css);</td>
</tr>
<tr>
<td>Actions</td>
<td>Find usage of @import statements</td>
</tr>
</tbody>
</table>

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
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.

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 a member of the 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 whitespaces, 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 does not possess conventions but a number of Contexts that, in turn, contain conventions. In this sense, the co-existence 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/syntax/coco.g4 and documented as Appendix E.

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

<table>
<thead>
<tr>
<th>Ontological constructs</th>
<th>Grammar constructs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Style Guide</td>
<td>Convention Set</td>
</tr>
<tr>
<td>Conventions (Style Guide)</td>
<td>Contexts (Convention Set), Conventions (Context)</td>
</tr>
<tr>
<td>Convention</td>
<td>Convention</td>
</tr>
<tr>
<td>Description (Convention)</td>
<td>Description (Convention)</td>
</tr>
<tr>
<td>Context</td>
<td>Pattern Descriptor</td>
</tr>
<tr>
<td>Ignored Constructs (Context)</td>
<td>Context</td>
</tr>
<tr>
<td>Constraint Combinator</td>
<td>Not, Or, And Expression</td>
</tr>
<tr>
<td>Number of Subjects (Constraint Combinator)</td>
<td>Operand of Combinator Expressions</td>
</tr>
<tr>
<td>Negation Constraint Combinator</td>
<td>Not Expression</td>
</tr>
<tr>
<td>Disjunction Constraint Combinator</td>
<td>Or Expression</td>
</tr>
<tr>
<td>Conjunction Constraint Combinator</td>
<td>And Expression</td>
</tr>
<tr>
<td>Constraint</td>
<td>Comparison, Is, In, Match, Node Query Expressions</td>
</tr>
<tr>
<td>Subject (Constraint)</td>
<td>Operand of Constraint Expressions</td>
</tr>
<tr>
<td>Value (Constraint Subclasses)</td>
<td>Second Operand of Binary Expression</td>
</tr>
<tr>
<td>Existence Constraint</td>
<td>Node Descriptor, Node Relation, Node Query Expression</td>
</tr>
<tr>
<td>Comparison Constraint</td>
<td>Comparison Expression</td>
</tr>
<tr>
<td>Type Constraint</td>
<td>Is Expression</td>
</tr>
<tr>
<td>Textual Form Constraint</td>
<td>Match Expression</td>
</tr>
<tr>
<td>Set Membership Constraint</td>
<td>In Expression</td>
</tr>
<tr>
<td>Literal Value</td>
<td>Literal Expression</td>
</tr>
<tr>
<td>Value (Literal Value)</td>
<td>Value (Literal Expression)</td>
</tr>
<tr>
<td>Violation Log</td>
<td>Violation Log</td>
</tr>
<tr>
<td>Violations (Violation Log)</td>
<td>Violations (Violation Log)</td>
</tr>
<tr>
<td>Violation</td>
<td>Violation</td>
</tr>
<tr>
<td>Description (Violation)</td>
<td>Description (Violation)</td>
</tr>
<tr>
<td>Position in Stylesheet (Violation)</td>
<td>Position (Violation)</td>
</tr>
<tr>
<td>Stylesheet</td>
<td>Stylesheet Node</td>
</tr>
<tr>
<td>Checked (Stylesheet)</td>
<td>—</td>
</tr>
<tr>
<td>Construct</td>
<td>Variable Expression</td>
</tr>
<tr>
<td>Property (Construct)</td>
<td>Call Expression</td>
</tr>
<tr>
<td>Pattern</td>
<td>CSS Pattern</td>
</tr>
<tr>
<td>Number of Constructs (Pattern)</td>
<td>Nodes (CSS Pattern)</td>
</tr>
</tbody>
</table>

Table 1. Representation/interpretation mapping between the domain-specific ontology and the abstract modelling grammar.
Figure 4. A development environment open with the Css-Coco 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:

```javascript
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 “pre-processors” 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 \textit{physics of notation} \cite{60}, in particular the ontological analysis \cite{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 the textural 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 \cite{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 \cite{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 \url{http://dx.doi.org/10.6084/m9.figshare.3085831.v3}.

References

\begin{thebibliography}{99}
\bibitem{16} Codacy. Patterns list. \url{https://www.codacy.com/patterns}.


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


