Towards a Structured Specification of Coding Conventions

Elder Rodrigues Jr. and Leonardo Montecchi

Instituto de Computação
Universidade Estadual de Campinas
Campinas, SP, Brazil
elder.junior@students.ic.unicamp.br, leonardo@ic.unicamp.br

***Abstract*—Coding conventions are a means to improve the reliability of software systems. They can be established for many reasons, ranging from improving the readability of code to avoiding the introduction of security flaws. However, coding conventions often come in the form of textual documents in natural language, which makes them hard to manage and to enforce. Following model-driven engineering principles, in this paper we propose an approach and language for specifying coding conventions using structured models. We ran a feasibility study, in which we applied our language for specifying 215 coding rules from two popular rulesets. The obtained results are promising and suggest that the proposed approach is feasible. However, they also highlight that many challenges still need to be overcome. We conclude with an overview on the ongoing work for generating automated checkers from such models, and we discuss directions for an objective evaluation of the methodology.**

***Keywords*—** Coding standards, coding conventions, model-driven engineering, domain-specific languages, static analysis.

I. Introduction

Coding conventions [22], also termed as *coding standards*, are sets of guidelines for software development that recommend a certain programming style or specific practices, or impose constraints. Depending on their purpose, coding conventions may cover different aspects of software development, including file organization, indentation, comments, declarations, naming conventions, programming practices, programming principles, architectural best practices, etc.

Besides purely "cosmetic" recommendations, the adherence to precise coding rules is a fundamental practice for enforcing non-functional properties like security or performance. For example, attackers often exploit known vulnerabilities introduced by poor usage of programming constructs or system calls. Similarly, performance bottlenecks can be avoided by preferring certain programming constructs instead of others (e.g., see [27]). Coding conventions are not static artifacts; rather, they evolve over time following the introduction of new language features or the discovery of new vulnerabilities.

It has been argued that existing coding conventions — in their current shape — offer limited benefit, because of the difficulties in actually enforcing and managing them [11]. In fact, like many other artifacts in the development process, coding conventions mostly come in the form of textual documents written in natural language, possibly complemented with code examples. Thus, they cannot be processed automatically, and tasks like the following ones must be done manually: i) check similarity between rules, ii) identify conflicting rules, iii) understand if a tool is able to check a certain rule, iv) configure a tool to check a certain rule, etc.

Following the principles behind Model-Driven Engineering (MDE) [20], all the artifacts in the software development process, thus including coding conventions, should be represented as *structured models*, to increase the degree of automation, improve integration, and reduce the possibility of human mistakes. However, to the best of our knowledge, there is little work on this topic in the literature.

In this paper we investigate the possibility of specifying coding conventions through structured, machine-readable, models. More in details, we provide the following contributions in this direction: i) we introduce a MDE-based approach for managing coding conventions as structured specifications; ii) we define a domain-specific language that realizes such approach for the Java programming language; iii) we run a feasibility study, in which we use the defined language to specify existing coding conventions for the Java language; and iv) we discuss our roadmap for a comprehensive evaluation of the proposed approach.

The rest of the paper is organized as follows. In Section II we introduce the necessary background and motivation, while in Section III we discuss the related work. In Section IV we present the overall approach, which is centered around a domain-specific language for specifying coding conventions. We define this language in Section V, by discussing its metamodel (i.e., abstract syntax), while usage examples are given in Section VI. The feasibility study and its results are presented in Section VII, while a discussion of the limitations and how we plan to overcome them is reported in Section VIII. Finally, conclusions are drawn in Section IX.

II. Background and Motivation

A. Coding Conventions

Frequently, even in the scientific literature and in tool manuals, the terms "coding convention", "coding standard", "coding rules", etc., are used interchangeably. To avoid ambiguity, we give here a brief definition of these terms for the context of this paper.

A (programming) language $\mathcal{L}$ is a subset of all the possible strings over a certain alphabet A, that is, $\mathcal{L} \subseteq A^*$. We use the term *code portion* to refer to any string that is admissible according to the language, i.e., any $\omega \in \mathcal{L}$.

A *coding rule* is a restriction on the possible ways to program software. It states the conditions under which a code portion ω must be considered invalid for the purpose of a software project. More formally, a coding rule is a function $f\colon \mathcal{L} \rightarrow \{\texttt{valid}, \texttt{invalid}\}$. Some rules only concern formatting aspects, e.g., naming of variables or placement of brackets. Enforcing such rules does not require altering the behavior of the software. We call this particular class of rules *coding style rules*.

A *coding convention* is a set of coding rules, usually having a specific purpose, e.g., improving security or performance. Many coding conventions are created for the purpose of a single project or company, and they never reach the public domain. Conversely, we consider a coding convention to be a *coding standard* when it is widely recognized in its reference community, or when it is actually published as a technical standard (e.g., MISRA C++ [14] or the JPL Java Coding Standard [13]).

B. Limitations in Current Practice

In current practice, a wealth of coding rules exists. For example, in the study in [23], an interview among 7 software engineers about the most important practices for software maintainability resulted in 71 different coding rules, and different opinions on their priority. Even established collections of coding recommendations, like the SEI CERT Coding Standards [24] or the MITRE Common Weakness Enumeration (CWE) [26], are continuously evolving, following the discovery of new vulnerabilities, the introduction of new features of programming languages, or simply changes to the agreed best practices. For example, the latest update to CWE involved 137 new vulnerabilities and 304 major changes[1].

Furthermore, many companies define their own coding conventions, which may be different among different teams or even for individual developers. This can happen, for example, because of different programming languages, of different project requirements, or simply because a certain client imposes its specific restrictions.

Typically, coding rules are specified using the natural language. Sometimes they are complemented with code examples, to demonstrate the problem being addressed and how enforcing the rule would remove it. The author of [11], scientist at NASA/JPL, argued that the benefit of existing coding conventions is often small, even for critical applications. The main reason for such lack of effectiveness was attributed to the lack of comprehensive tool-based (i.e., automated) compliance checks. Although more than ten years have passed, the practice in the management of coding conventions has not significantly improved. Probably, we can still say that the most evident pattern among different coding conventions is that "each new document tends to be longer than the one before it" [11].

While the support of automated tool has improved in recent years (as discussed later in Section III), it can not be said that coding standards are supported by comprehensive automated compliance checks, except for very specific set of rules. Tool support is fragmented: each static analysis tool checks a different set of rules, often for a specific programming language. Verifying all the rules of a certain coding convention needs the combined application of multiple tools, and rarely all the rules can be automatically checked. This is especially true when customized coding rules need to be enforced. To complicate this scenario, it is often difficult to understand which rules a tool is capable to check (or vice versa).

Providing a structured specification of coding standards would open up several benefits. For example, industrial standards could include such structured specification of the imposed coding rules, and tool developers could expose the set of rules that their tool is capable of checking. Most importantly, a machine-readable description of coding rules would enable the automated generation of checkers for such rules. Such possibility would also improve automation, for example enabling the integration of coding standards in "continuous" development practices [21].

We note that the problem of finding a structured way to describe coding conventions is starting to attract attention in the scientific literature. Starting from similar motivations, the work in [9] defines a domain-specific language to specify coding rules for CSS[2], a simple language for web design.

III. Related Work

Static analysis consists in searching source code for common defects and known bug patterns, without executing the software itself. Several tools exist for this purpose. Among the most popular ones, FindBugs (now SpotBugs) [12] is a tool to find bugs in Java programs, originally created to detect null pointer defects. It features a plugin module that can be used to write customized detectors for additional bug patterns. Similarly, QJ Pro [4] checks conformance to a predefined set of formatting rules, misuses of the Java language, code structure, etc. Unfortunately, from the available documentation we were not able to precisely determine which rules are supported by this tool.

Several other tools for static code analysis exist. However, few of them support the definition of customized extensions. PMD [3] and CheckStyle [2] are two notable exceptions. Both of them can analyze code for compliance with either predefined rules, or rules created by users using scripts or configuration languages. A study comparing FindBugs, QJ Pro, and PMD can be found in [29], while a more general survey on static analysis techniques and tools can be found in [10].

For customization, both PMD and CheckStyle use a configuration language based on XML. However, it should be noted that the kind of rules that can be described with such languages is limited by the features that are actually implemented in the tool. They also allow users to write custom checks using Java code or XPath queries, which however produce very verbose and complex specifications that in the end are themselves prone to defects. In this paper, we define a generic tool-agnostic language to specify coding conventions, from which low-level tool-specific configurations can be derived by model transformation.

[1] https://cwe.mitre.org/data/reports/diff_reports/v3.1_v3.2.html

[2] Cascading Style Sheets, see https://www.w3.org/Style/CSS/

As mentioned earlier, the authors of [9] define an approach to provide machine-readable specifications of coding rules for CSS. Another interesting approach has been introduced in [5] and implemented in Naturalize, a tool based on Natural Language Processing (NLP), which analyzes a code base to first recognize naming and formatting conventions adopted in the project, and then identify possible violations. However, these approaches only address formatting and naming issues (i.e., coding style), and there is no way to specify customized coding rules that address security aspects, for example. The authors of [30] focus on structuring the relations between rules and vulnerabilities across different repositories, but they do not provide a structured specification of the rules themselves.

Other works in the literature focused on modeling different aspects of source code. Some of them focused on the formalization of so-called code smells (e.g., [15]), which however are only one of the reasons that drive the definition of coding conventions. More in general, most of these works are related to the specification of rules for the reverse engineering of software. A survey of MDE techniques for reverse engineering can be found in [18].

In this respect, the Knowledge Discovery Metamodel (KDM) [17], defined by the Object Management Group (OMG), is of particular relevance. KDM is a metamodel for representing existing software: it considers the physical and logical elements of software at various levels of abstraction, as well as their relations. The primary purpose of this metamodel is to enable a common interchange format for interoperability between tools, as a vendor-neutral format. MoDisco (Model Discovery) [7] provides a concrete implementation of the metamodel, and it supports the extraction of KDM models from software. However, the objective of KDM is to model an entire software project in its details, while our objective is to model coding conventions at a higher level of abstraction.

The QL language [6] is a query language that has been mostly applied to the specification of queries on source code. QL is considered a general-purpose query language [6], while our objective is to define a domain-specific language for the specification of coding conventions. Also, because it supports arbitrary queries on source code, QL is necessarily quite verbose, while we aim at a concise specialized language. Finally, we are proposing a complete MDE workflow, in which our metamodel is the basis for deriving other artifacts (see Figure 1 later). QL queries could be an example of derived artifacts, as discussed in the next section.

IV. The Proposed Approach and its Challenges

In this section we describe the general workflow we envision for the management of coding convention, as well as the challenges that need to be overcome to concretely realize it.

A. The Workflow

Our workflow for the management of coding conventions as structured, machine-readable, specifications is depicted in Figure 1. The workflow is centered around a domain-specific language that is used to provide such structured specifications. We call this language the Coding Conventions Specification Language (CCSL). Textual version of existing coding rules are translated into specifications in such language, while new rules can be created directly as CCSL models.

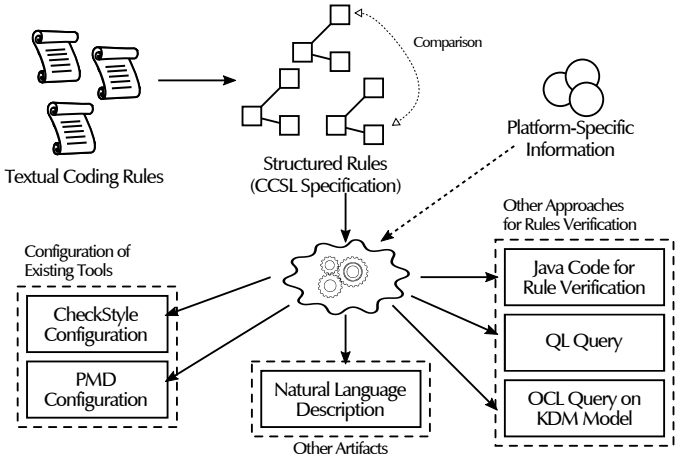

Figure 1: The proposed workflow for a the management of coding conventions.

Once the machine-readable specification of coding rules is available, they can be analyzed, for example to identify conflicting rules or equivalent ones. Then, from such specifications, model-transformations can be applied to automatically derive different artifacts. We identify three classes of artifacts that can be derived from CCSL models.

First, it is possible to derive configuration files for existing static analysis tools. As mentioned before, tools like CheckStyle or PMD can be configured to check custom rules. The process is however quite complex, and it requires a deep knowledge of the tool. For example, in PMD this is done by writing XPath queries over the Abstract Syntax Tree (AST) extracted from the source file, or by directly implementing a Java class that realizes the check. By defining a model transformation algorithm it is ensured that new coding rules that can be defined by a CCSL specification can be automatically checked, by deriving the proper configuration file for one of the existing tools.

Second, there may be situations in which existing static analysis tools are not used, e.g., because no tool is actually able to check a certain set of rules, or because of other technical or organizational reasons. In this case, the specified coding rules can be verified by other means, e.g. by deriving: i) source code or scripts to perform the verification programmatically; ii) QL queries; or iii) OCL[3] queries, in case a structured model of the software under analysis is available. This last option is especially interesting when considering a MDE context and the capabilities of a platform like MoDisco [7], which is able to extract a KDM model from the source code of an existing application.

In principle, other kinds of artifacts could be derived. For example, an explanation of the rule in natural language, or examples of source code portions that violate it.

Since we aim to define CCSL as a high-level language, platform-specific information may be needed to concretely

[3]The Object Constraint Languge (OCL) is a query language for models, published as an OMG standard. See https://www.omg.org/spec/OCL/.

derive low-level artifacts (e.g., tool configuration) for a specific platform. For example, consider a rule that mentions "thread-safe methods", as rule TSM00-J in [24]. Properly verifying such rule requires knowledge of which methods are thread-safe in a certain platform, or which language constructs make a method thread-safe (e.g., the Java *synchronized* keyword). This is reflected by the element "Platform-Specific Information" in Figure 1. The simplest form of such information is a mapping between elements of the CCSL (i.e., metaclasses) and keywords of the target programming language or platform.

B. Main Challenges

To realize the workflow described in the previous section, several technical and research challenges need to be overcome. We highlight the most important ones in the following.

1) Vocabulary: The first challenge consists in the vocabulary to be considered, since different programming languages may use different terms to represent the same concept. To minimize this problem, in this paper we focus on the Java language only. Extension to other programming languages will be investigated as future work.

2) Abstraction Level: A model is an abstraction of reality, in the sense that it cannot represent all aspects of reality [8]. When trying to define a structured representation of coding rules, we are trying to define *models* of such rules. Such models are expressed in CCSL, whose abstract syntax is defined by a *metamodel*. Finding the appropriate abstraction level of the metamodel is challenging, especially because the notion of preciseness of a model is not an absolute notion [8].

While a lower abstraction level allows more coding conventions to be specified, it results in a complex and verbose metamodel, resembling the abstract syntax of the programming language itself, and thus going in the opposite direction with respect to a domain-specific language. On the other hand, a higher abstraction level simplifies the specification of coding rules for the user, but it makes the derivation of low-level artifacts more difficult.

3) Rules Verification: Even if a coding rule can be specified using CCSL, this does not guarantee that it may be easily verified using some static analysis tool or program code. In fact, some rules may be hard or impossible to be verified using static analysis only.

4) Ambiguity of Natural Language: Coding conventions are normally described using the natural language. Although they are usually accompanied with examples of complying and non-complying code (e.g., see [24]), a certain degree of ambiguity still remains. Even when it is possible to provide a structured specification of the coding rule, it is challenging to determine whether such specification really represents what the original rule intended. This aspect is especially challenging for the evaluation of the metamodel, as discussed later in Section VII.

V. Coding Conventions Specification Language

Working towards the realization of the workflow in Figure 1, In this section we describe the metamodel of the CCSL language, which is used to provide structured specifications of coding rules.

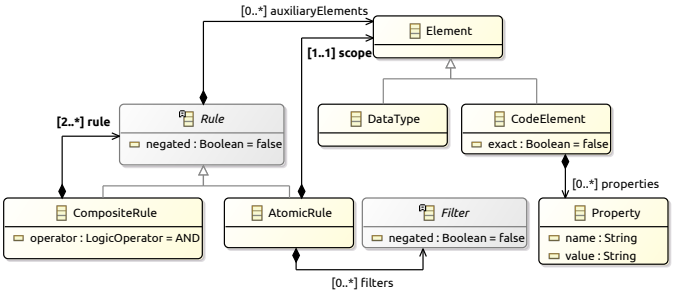

Figure 2: *Core* Package.

In this paper we decided to focus on Java since i) it is a very popular language, widely used in the industry, and ii) it is covered by many coding conventions. We decided however to group the Java-specific aspects in a separate package, to leave the metamodel open for extensions.

A. Overview

In our approach, the structured specification of a rule specifies patterns that would *violate* the rule in the code. That is, given a rule $f\colon \mathcal{L} \to \{\texttt{valid}, \texttt{invalid}\}$, our objective is to give a specification of the subset of the programming language $\mathcal{L}_f \subseteq \mathcal{L}$ such that $f(\omega) = \{\texttt{invalid}\} \iff \omega \in \mathcal{L}_f$.

We identified the core concepts that need to be included in the language by analyzing multiple sources, including: i) existing coding conventions, in particular those for the Java language; ii) existing query languages; iii) main concepts of object-oriented programming; and iv) existing source code models, in particular the aforementioned KDM.

Once the concepts have been identified, we defined the actual metamodel of our CCSL language using the Ecore metamodeling language, which is part of the Eclipse Modeling Framework (EMF) [25]. The main elements of the metamodel are described in the next sections, grouped by package. Its complete definition is available on GitHub [19].

B. Core Package

This package contains the core concepts of the metamodel, which are illustrated in Figure 2 in EMF notation. In our metamodel, a coding rule is represented by the *Rule* metaclass, which can be either atomic or composite.

An *AtomicRule* is composed of:

Scope A scope identifies the programming language element to which the rule applies. The rule is applicable only when the element specified by the scope actually exists in the source code.

Filters Filters are used to select only those elements that fulfill specific conditions. *Filter* is an abstract metaclass, and it is extended by several concrete filters. A filter can be *negated*, which means that only elements *not* fulfilling the filter are selected.

Complex rules can be specified as a *CompositeRule*, which is essentially a connector to combine multiple rules using Boolean logic operators (and, or, if-then, etc).

As an example consider a simple rule for Java that forbids any public field except constants. In this case, the *scope* is

any field that has the "public" property, while a *filter* would be applied to exclude those fields that also have the "static" and "final" properties (i.e., constants).

Sometimes it is necessary to include information that is related to the scope but that is not actually part of the scope, for example the return type of a method. For this reason, every rule may contain a set of elements (*auxiliaryElements*).

Figure 2 also illustrates the *Element* metaclass, which is the top metaclass in the hierarchy. Currently, the metamodel considers two kinds of elements: *DataType* and *CodeElement*. The *DataType* represents a datatype (e.g., a primitive type) of the programming language. On the other hand, the *CodeElement* metaclass represents an element of the source code being analyzed (i.e., variables, methods, invocations, etc.). As the Ecore language supports multiple inheritance a metaclass can be subclass of both *CodeElement* and *DataType*.

Each *CodeElement* has a set of *Property*. The *Property* metaclass represents a generic property of the element, characterized by its name and possibly a value. In this context, with the term "property" we mean characteristics that source code elements may exhibit (e.g., modifiers of a class).

As an example, consider a variable declared as private and final in the Java language. In our metamodel it would be represented as an instance of the *Variable* metaclass, with two instances of the *Property* metaclass with their "name" attribute set as "private" and "final", respectively.

C. NamedElements Package

A *NamedElement* is a *CodeElement* that can be identified by its name. We distinguish four kinds of *NamedElement*: i) *ComplexType*, which represents a custom datatype, e.g., a Class (*ComplexType* is also a subclass of *DataType*, which was omitted from the diagram for simplicity) ii) *Variable*, iii) *Method*, and iv) *Namespace*.

The *NamedElement* hierarchy is illustrated in Figure 3. Due to space limitations the full hierarchy was omitted; full details of the metamodel are available on GitHub [19].

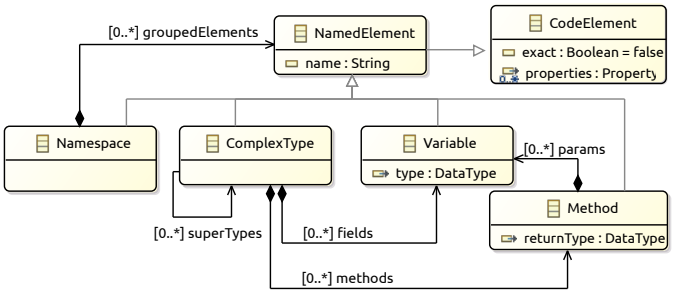

Figure 3: *NamedElements* Package.

D. Statements Package

The next key concept in the metamodel is the *Statement* metaclass and its hierarchy, as illustrated in Figure 4. The full hierarchy was omitted due to space limitations; full details of the metamodel are available at [19].

A *Statement* represents a command to be executed. The metamodel supports the following kinds of statements as specializations of the *Statement* metaclass.

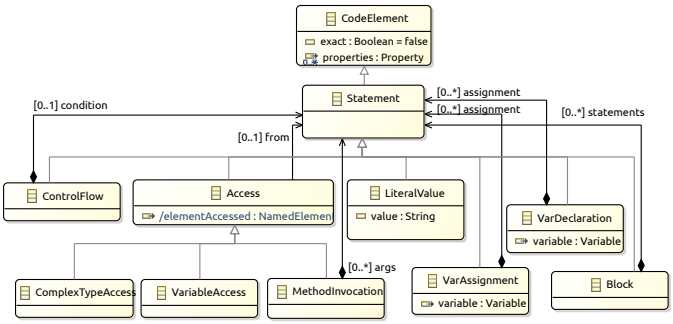

Figure 4: *Statements* Package

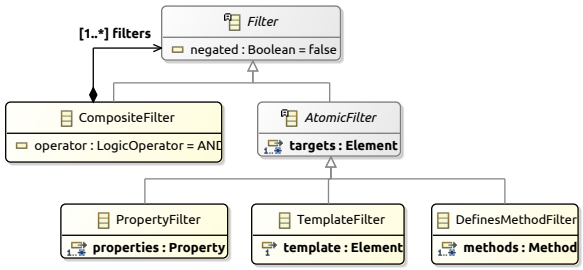

Figure 5: *Filters* Package

The *Access* statement represents the access to the reference of a variable, ComplexType, or method. A *Block* represents a block (sequence) of statements, while *ControlFlow* represents a control flow construct like the if-then and while statements.

Finally, *Expression* and *LiteralValue* represent an expression and a literal value, respectively, while *VarDeclaration* and *VarAssignment* represent variable declaration and assignment, respectively.

E. Filters Package

The *Filter* metaclass represents filters that are used to identify specific elements within the scope given by the *element* attribute of the *Rule* (see Figure 2).

The structure of a filter is depicted in Figure 5. To improve flexibility, filters adopt the main idea of the Composite design pattern, where the *CompositeFilter* represents a list of filters combined by a logic operator (and, if-then, etc.), and *AtomicFilter* is an abstract metaclass which represents an entry-point to define new filters. Every filter can also be *negated* or not.

Every *AtomicFilter* contains a list of elements to which the filter will be applied (*target* attribute). Concrete filters for different purposes are created by extending the *AtomicFilter* abstract metaclass (see Figure 5). For the sake of brevity we describe here only the following three filters, which will be used in the rest of the paper:

1) *PropertyFilter*. It is used to filter an element according to its properties. That is, it checks whether an element contains a certain list of properties.
2) *DefinesMethodFilter.* It filters a *ComplexType* according to the methods that it defines. An element is selected by the filter if the specified methods are defined in the *ComplexType* element or in one of its ancestors (i.e., super types).
3) *TemplateFilter*. This filters checks if the target element can be matched to the provided template, which is

another instance of the *Element* metaclass. This is the most generic filter and it was designed to improve the flexibility of the metamodel. In fact, in case none of the available filters can be used to explicitly specify the desired condition, the *TemplateFilter* can be used to provide a "sample" instance of an *Element* that describes the kind of elements that should be selected.

Examples of application of filters are presented later in Section VI.

F. Java Package

We decided to keep the organization of the CCSL metamodel generic, to facilitate its extension in the future. For this reason, we separate elements that represent specific aspects of the Java language in this package (Figure 6).

For example, the generic *ComplexType* metaclass does not support that classes can be nested (inner classes) and that classes can implement interfaces. Specialized metaclasses have been added, as extensions of the *ComplexType* metaclass. Similarly, the *Method* metaclass has been extended to represent that in Java methods can throw exceptions.

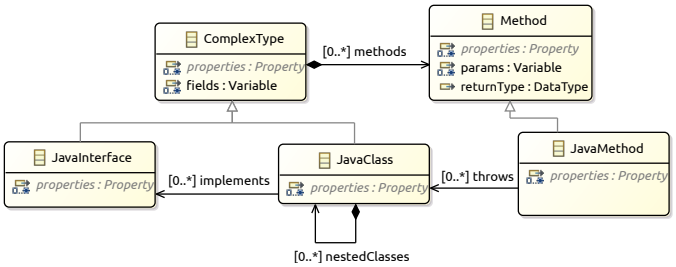

Figure 6: *Java* Package.

VI. CCSL Usage Example

To better explain how CCSL is actually used, in the following we provide concrete examples of application of the metamodel. A broader study has been executed and it is reported in Section VII.

The metamodel presented above defines the abstract syntax of CCSL. For practical reasons, we use a textual concrete syntax, which has several advantages [28], including easier interaction with version control systems. In particular, we present our examples using a syntax based on the Human-Usable Textual Notation (HUTN) [16], an OMG standard for storing models in a human-understandable format.

We have separated this section in two subsections. The first introduces the basic idea behind the CCSL using toy examples, while the second demonstrates its real application to the specification of a real rule of the SEI CERT Coding Standard [24].

A. Writing a CCSL Specification

As previously mentioned, the main part of a rule is its scope, which identifies the kind of elements to which it applies. It should be noted that, while an *Element* may in general have many different attributes, most of them have a minimum multiplicity of zero. This means that only the attributes that are needed for expressing the rule need to be specified, and all the other may be ignored.

As an example, consider the specification illustrated in Figure 7. In this example a code portion is considered invalid if it contains a *ComplexType* named as "Foo" that define a method named as "bar". Applying this rule to a Java source code means that any class or interface with the name set to "Foo" and that defines a method named as "bar" would not be allowed. It is worth noting that this specification applies to any *ComplexType* that define *bar* method and not only those that contains the *bar* method only.

```
1 AtomicRule {
2   scope: ComplexType {
3     name: "Foo"
4     methods: [Method {
5       name: "bar"
6     }]
7   }
8 }
```

Figure 7: Basic example of a CCSL specification.

B. MET09-J Rule

Once the basic concepts about CCSL have been introduced, consider now rule MET09-J of the SEI CERT Coding Standard for Java [24]:

"**MET09-J:** ***Classes that define an equals() method must also define a hashCode() method.*** *[. . .] The equals() method is used to determine logical equivalence between object instances. Consequently, the hashCode() method must return the same value for all equivalent objects. Failure to follow this contract is a common source of defects."*

The part in bold is the actual title of the rule, while the rest of the text is its explanation. Note that, in general, determining if the *hashCode* method actually returns the same value for all equivalent objects is not feasible with static analysis, and it is in general a hard problem. Therefore, we do not include it in the specification of the rule.

Figure 8a illustrates the specification of the above rule using CCSL. Since the rule is not composed of sub-rules, it is only necessary to create an instance of the *AtomicRule* metaclass (line 1). The element to be searched, which defines the scope of the rule, is a *JavaClass* that contains a method named "equals" (lines 2–6). However, only classes that define an "equals" method and do not define a "hashCode" method must be matched. This can be achieved by applying a filter: the *TemplateFilter* (lines 7–17) receives as target the *JavaClass* referenced by letter "c" and checks whether it is *not* possible (it is negated, line 8) to match it against the template (a *JavaClass* that contains the "hashCode" method, lines 10–16).

Rule MET09-J is also a good example of how rules defined in natural language may be ambiguous and thus be interpreted in different ways. There are at least two aspects of this rule that are ambiguous:

1) It is not clear which signatures of the *equals* method and *hashCode* method that are affected by the rule. In fact, it is possible to define multiple methods having the same

name, using polymorphism. The traditional signature of the equals method in Java is *boolean equals(Object obj)*. However, it is possible to overload the equals method by defining a slight variation: *boolean equals(CustomClass obj)*. Whether the rule MET09-J should apply only to the original *equals* method or not is open to interpretation. Furthermore, note that the original (verbatim) text of the rule actually mentions the "equals()" method, i.e., one without parameters.

2) The *hashCode* method could have been defined in a superclass rather than in the same class that defines the *equals* method. This would also prevent introducing the bug mentioned by the rule. However, by looking at the text it is not clear whether such implementation should raise a warning or not.

Figure 8b illustrates an alternative specification of the MET09-J rule, which considers the following interpretation. If a certain class defines a *boolean equals(Object obj)* method, then the *int hashCode()* method should be provided in that class or in one of its superclasses, except for the *Object* class.

In order to reference primitive types and the *Object* class it is necessary to create instances of the *PrimitiveType* and the *JavaClass* metaclasses in the *auxiliaryElements* of the rule (lines 2–10). The scope of the rule is still a *JavaClass*, but now the *equals* method is specified as a method which returns a *boolean* and contains exactly one parameter of type Object (lines 12–17). Finally, the filter being applied in the scope is the *DefinesMethodFilter* (see Section V-E). Such a filter will search for a implementation of the *hashCode* method for all the hierarchy (except for Object class) of the given class (lines 19–28). Note that the specification of the *hashCode* method has the attribute *exact* set to true (line 23). This means that only implementations of *hashCode* method that contains exactly zero parameters will be considered.

In this case, whether specification (a) or (b) is the correct one is debatable. This however highlights the importance of providing a structured semi-formal specification, to avoid ambiguities of this kind.

```
1  AtomicRule {
2    scope: c as JavaClass {
3      methods: [Method {
4        name: "equals"
5      }]
6    },
7    filters: [ TemplateFilter{
8      negated: true,
9      targets: [c],
10     template: JavaClass {
11       methods: [Method {
12         name: "hashCode"
13       }]
14     }
15   }]
16 }
```

(a) Initial specification.

```
1  AtomicRule {
2    auxiliaryElements:[
3      boolean as PrimitiveType{name: "boolean"},
4      int as PrimitiveType{name: "int"},
5      objClass as JavaClass {
6        name: "java.lang.Object"
7      }
8    ]
9    scope: c as JavaClass {
10     methods: [Method {
11       exact: true,
12       name: "equals",
13       returnType: boolean,
14       params: [Variable {type: objClass}]
15     }]
16   },
17   filters: [ DefinesMethod{
18     negated: true,
19     targets: [c],
20     methods: [Method {
21       exact: true,
22       name: "hashCode",
23       returnType: int,
24       params: []
25     }]
26   }]
27 }
```

(b) Alternative specification considering that "hashCode" may be implemented by ancestors.

Figure 8: Two possible CCSL specification of rule MET09-J from SEI CERT [24].

VII. Feasibility Study

In this section we report on a feasibility study we performed as initial evaluation of the proposed approach. The objective of this study was to understand i) whether the proposed approach is feasible, and ii) to what extent the current version of the metamodel is able to specify existing coding rules.

A. Study Description

To run the study we selected rules from two popular coding conventions for the Java language. Each rule in the selected subsets was manually analyzed, and a specification using CCSL was attempted.

Recalling the definition given in Section II, a coding rule is a function $f\colon \mathcal{L} \to \{\texttt{valid}, \texttt{invalid}\}$. Therefore, being able to provide a specification of a certain rule f means that the specification correctly identifies a subset of the language $\mathcal{L}_f \subseteq \mathcal{L}$ such that $f(\omega) = \{\texttt{invalid}\} \iff \omega \in \mathcal{L}_f$. That is, being able to provide a CCSL specification that identifies all (and only) the possible source code portions that violate the rule.

Whether a correct specification can be provided or not is somehow subjective, as it depends on the skill of the modeler, and on the interpretation of the original rule written in the natural language (see Section IV-B).

To decide if a rule was correctly specified we used the following two heuristics:

- *Have all the concepts mentioned in the original rule been included in the CCSL specification?*
- *Does the CCSL specification contain enough information to verify the rule automatically?*

Otherwise, we consider that we were not able to provide a specification of the rule using our language.

B. Selected Coding Rules

For this evaluation we selected two popular but quite different set of rules: one is derived from checks implemented in a static analysis tool, while the other contains general principles

Table I: Number of individual rules that were successfully specified using CCSL (*Specified/Yes* column), and those for which a specification was not possible (*Specified/No* column).

Source	Section	Specified		Total
		Yes	*No*	
PMD	Error Prone	73 (75%)	24 (25%)	97
	Multithreading	8 (80%)	2 (20%)	10
	Performance	24 (80%)	6 (20%)	30
	SubTotal	**105 (77%)**	**32 (23%)**	**137**
SEI CERT	Characters and Strings	0	5 (100%)	5
	Declarations and Initialization	0	3 (100%)	3
	Exceptional Behavior	4 (40%)	6 (60%)	10
	Expressions	1 (17%)	5 (83%)	6
	Methods	7 (54%)	6 (46%)	13
	Numeric Types and Operations	3 (25%)	9 (75%)	12
	Object Orientation	5 (42%)	7 (58%)	12
	Thread APIs	5 (83%)	1 (17%)	6
	Thread Pools	0	5 (100%)	5
	Visibility and Atomicity	0	6 (100%)	6
	SubTotal	**25 (32%)**	**53 (68%)**	**78**
Total		**130 (60%)**	**85 (40%)**	**215**

for improving security, for which a checker is not necessarily available.

1) PMD Rules: The PMD tool [3] mentioned before is a static code analyzer that provides a wide set of predefined coding rules, organized according to different topics. We selected three groups of rules for our evaluation: *Error Prone*, *Multithreading*, and *Performance*. All of these rules are described with natural language, and then implemented in the tool by means of an XPath query or by explicitly coding a Java method that performs the check. Due to the characteristics of the tool, it is not possible to implement rules spanning the whole source tree, or rules requiring complex analysis. Therefore, coding rules contained in this set are in general relatively simple.

2) SEI CERT Coding Conventions: The Software Engineering Institute (SEI) of Carnegie Mellon University maintains extensive coding conventions focused on security, for popular languages such as Java and C++ [24]. Rules in this collection are harder to model, as they address different aspects of programming, and their descriptions span different levels of detail. For example, some rules go into the details of the unpacking of compressed files (IDS04-J), while other ones simply give the generic recommendation to "perform proper cleanup at program termination" (FIO14-J).

We based our feasibility study on the version for Java of such rules. The "SEI CERT Oracle Coding Conventions for Java" are organized in different subsets (e.g., *Expressions*, *Methods*, etc.). We excluded some subsets from our evaluations, as they are clearly outside the scope of the metamodel, for example the *Runtime Environment* subset, which includes recommendations for the configuration of the runtime platform (e.g., ENV04-J: "Do not disable bytecode verification"). The selected subsets are listed in Table I.

C. Results

During the experiment we analyzed a total of 215 individual coding rules. That is, we tried to devise the CCSL specification of 215 different rules, similarly to what was done for the MET09-J rule in Section VI-B. Results of the analysis are reported in Table I. The CCSL specifications given for the analyzed rules are also available on GitHub [19].

We believe that the results confirm the feasibility of the approach, as we were able to specify more than half (60%) of the considered rules. However, they also highlight limitations of the current version of the language, and they indicate that the kind of rules that is considered has a great impact on the chance of a successful specification.

In fact, we were able to specify most of the rules provided by PMD (77%) using our approach. This was somehow expected, as such rules have already an implementation that is capable of checking them, meaning, at least, that their verification can be automated. It should be noted however that with PMD a new checker must be manually written for each new rule and that some checkers are just Java code, i.e., they do not have a high-level specification. Rules provided by PMD for which it was not possible yet to provide a specification (23%) are due to the lack of specific primitives in the current version of the metamodel. For example, the analysis highlighted that more complex concepts related to the execution flow are needed.

Conversely, we can provide a CCSL specification only for a small portion of the rules in the SEI CERT group. Most of those rules are in fact either too generic or so specific that it would be necessary to develop specific checkers to verify them. Increasing the abstraction level of the metamodel may improve these results, although this may make the generation of artifacts from the metamodel harder. At the same time, we highlight that for many of the SEI CERT rules no automated checker is available anyways. Considering this aspect, being able to provide a structured, machine-readable, specification of almost one third of them as a first attempt is still an encouraging result.

Finally, it should be noted that those results represent a feasibility study based on the current status of our work. Refinements of the metamodel may support the specification of some rules that have not been specified. The construction of MDE frameworks is recognized to be an iterative process, in which the first step is building a deep understanding of a small part of the reference domain [28]. A deeper discussion on our plans for a comprehensive evaluation of the proposed approach are reported in the next section.

VIII. Discussion and Next Steps

A. Threats to Validity

As initial investigation on this topic, the work presented in this paper is subject to some threats to validity, mainly originating from the challenges described in Section IV-B. We discuss here the most important ones, together with our plans for mitigating them.

1) Internal Validity: The most important threat concerning internal validity is the correlation existing between the definition of the metamodel and its application for the specification of coding rules. In fact, both activities were performed by the same persons. The second threat is the subjectiveness

in judging whether a certain CCSL specification correctly represents the original rule written in natural language. As discussed in Section IV-B this threat is difficult to mitigate, given the ambiguity of the natural language.

A possible way to mitigate this problem is to perform an extensive evaluation with experts, or in general with external participants. Two kinds of evaluation can be performed: i) provide them with CCSL specifications and ask them to assess whether the specifications correctly represent the corresponding textual rule, or ii) provide them with a set of textual rules and ask them to devise an equivalent CCSL specification. The second experiment would be ideal, but it is considerably more difficult to realize, because of the high demands posed on the participants, in terms of required time and skills.

A more systematic way to mitigate the subjectiveness problems is to actually generate checkers from the modeled rules, run them on real source code, and then compare the results with those obtained with existing static analysis tools. A path to the realization of this evaluation is discussed in the next section.

2) External Validity: The workflow presented in this paper (Figure 1) is intended to be applicable for expressing coding conventions for multiple programming languages. However, in this paper we defined a domain-specific language (CCSL) that is limited to the Java language. This was in part mitigated by selecting two popular and comprehensive coding standards, whose rules address different aspects of programming. Investigating whether the generalization of the approach to other programming languages is feasible is part of our future work.

B. Transformations

A systematic way to mitigate the threats discussed before is to implement transformations that generate automated checkers for CCSL rules. This would allow us to ask more objective questions for deciding whether a rule has been specified correctly or not, e.g.:

1) *Was it possible to derive automated checkers from the CCSL specification?*
2) *Does the checker derived from the CCSL specification provide the same results as existing tools that implement the same rule? If not, is it more accurate or less accurate?*

We provide here a concrete overview of the approach we are adopting to implement the generation of such checkers. The main approach is illustrated in Figure 9 and consists of two steps: i) extraction of a structured model of the application from the Java source code (this can be done by the MoDisco tool [7]); and ii) generation of OCL queries from CCSL specifications, to be applied on such models of the application.

As an example, consider the previous mentioned MET09-J rule (see Section VI-B). When executing the transformations with MET09-J rule as input, the OCL query illustrated in Figure 10 is generated automatically by a model-to-text transformation. The general structure of the transformation, and of the generated OCL, can be summarized in three steps:

1) *Identification of the scope*. Here it is identified which metaclasses from the structured Java metamodel can represent the scope metaclass from CCSL specification.

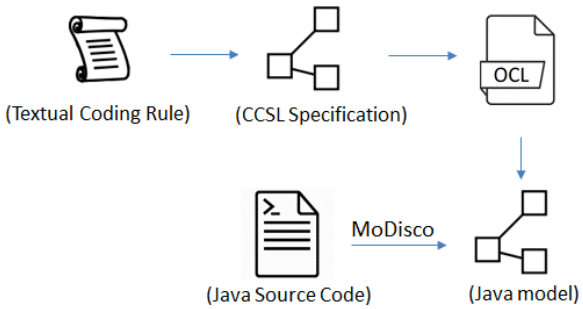

Figure 9: The workflow for the automated verification of rules specified with CCSL. OCL queries are automatically generated, and then applied to a structured model of the Java application.

   Once the target metaclasses have been identified, the skeleton of an OCL query to select all the instantiations that meet the specification of the given scope is generated (line 1). For example, in the MET09-J rule the target metaclass from Java metamodel is the *ClassDeclaration* metaclass.
2) *Scope conditions*. In this stage all the OCL constraints for all the conditions that must be satisfied by the scope are generated. When processing the specification of MET09-J rule, this step generates a set of conditions where only Java classes that have an *equals* method are selected (lines 3–28).
3) *Filters conditions*. In this stage the transformation generates all the OCL constraints for all the filters that are specified in the CCSL rule. When processing the specification of MET09-J rule, the transformation generates a condition where only Java classes that do not contain a hashCode method are selected (lines 29–46).

We have implemented the transformation using Acceleo [1]. The source code that implements the currently available transformations, as well as the metamodel are avaliable in the GitHub repository [19].

IX. Concluding Remarks

In this paper we proposed an approach to provide structured specifications of coding conventions, by applying model-driven engineering techniques. To the best of our knowledge, there is little work in such direction. We defined a language, CCSL, that can be used to specify coding rules in a structured way, and we applied it to two large sets of existing coding conventions.

We analyzed a total of 215 individual rules from these two coding conventions, with the objective to understand to what extent they could be specified using the proposed approach. Results are promising, but also show that the focus of rules and the way they are written have a fundamental impact. Overall, it was possible to represent 60% of the considered coding rules, which can be considered satisfactory for a first investigation.

We discussed the limitations of the study, and how they can be mitigated. In particular, we outlined the approach that we are using to implement a comprehensive transformation algorithm that generates OCL queries for Java models from CCSL specifications. Once the complete transformation will

```
-- Identification of the scope
ClassDeclaration.allInstances()→select(c1 |
-- Scope Conditions
  c1.bodyDeclarations→exists(m2 |
    m2.oclIsKindOf(MethodDeclaration) and m2.oclAsType(MethodDeclaration).name = 'equals' and
    m2.oclAsType(MethodDeclaration).returnType.type.name = 'boolean' and
    m2.oclAsType(MethodDeclaration).parameters→exists(v3 | v3.type.type.oclAsSet()→exists(c4 |
      c4.oclIsKindOf(ClassDeclaration) and
      c4.oclAsType(ClassDeclaration).package→asOrderedSet()→closure(package)→reverse()→iterate(
        p: Package; fullName: String = '' | fullName.concat(p.name).concat('.')
      ).concat(c4.oclAsType(ClassDeclaration).name) = 'java.lang.Object' and
    m2.oclAsType(MethodDeclaration).parameters→size() = 1 and
-- Filter Conditions
  not (
    let c1InheritanceClasses: Set(ClassDeclaration) = c1→closure(c: ClassDeclaration |
      if(not c.superClass.oclIsUndefined()) then  c.superClass.type else Set(ClassDeclaration){} endif
    ) in c1InheritanceClasses→exists(c: ClassDeclaration | c.bodyDeclarations→exists(m5 |
      m5.oclIsKindOf(MethodDeclaration) and m5.oclAsType(MethodDeclaration).name = 'hashCode' and
      m5.oclAsType(MethodDeclaration).returnType.type.name = 'int' and
      m5.oclAsType(MethodDeclaration).parameters→size() = 0
  )))
))))
```

Figure 10: OCL query corresponding to the CCSL specification of Figure 8b, generated by automated transformation.

be available, it will be possible to compare the results of our framework with one of the existing static analysis tools and thus perform a more objective evaluation.

As future work, we will work on refining and extending the metamodel, aiming to cover a wider range of rules. Another interesting direction we plan to investigate is to derive guidelines for the definition of coding conventions, that is, understanding which characteristics make them suitable for a structured specification and automated verification.

Acknowledgment

This work has been supported by the São Paulo Research Foundation (FAPESP) with grant #2018/11129-8. This work has been developed in the context of the H2020-MSCA-RISE-2018 "ADVANCE" project (grant 823788).

References

[1] Acceleo. https://www.eclipse.org/acceleo/, (Accessed October 24, 2019)
[2] CheckStyle, http://checkstyle.sourceforge.net/ (Accessed October 24, 2019)
[3] PMD, https://pmd.github.io/ (Accessed October 24, 2019)
[4] QJ Pro, http://qjpro.sourceforge.net/ (Accessed October 24, 2019)
[5] Allamanis, M., Barr, E.T., Bird, C., Sutton, C.: Learning natural coding conventions. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 281–293. ACM (2014)
[6] Avgustinov, P., de Moor, O., Jones, M.P., Schäfer, M.: QL: Object-oriented Queries on Relational Data. In: 30th European Conference on Object-Oriented Programming (ECOOP 2016). vol. 56, pp. 2:1–2:25. Dagstuhl, Germany (2016)
[7] Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: MoDisco: a generic and extensible framework for model driven reverse engineering. In: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10). pp. 173–174 (2010)
[8] Bézivin, J.: On the unification power of models. Software and Systems Modeling (4), 171–188 (2005)
[9] Goncharenko, B., Zaytsev, V.: Language design and implementation for the domain of coding conventions. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2016). pp. 90–104 (2016)
[10] Gosain, A., Sharma, G.: Static analysis: A survey of techniques and tools. In: Intelligent Computing and Applications. pp. 581–591 (2015)
[11] Holzmann, G.J.: The power of 10: rules for developing safety-critical code. IEEE Computer **39**(6), 95–99 (2006)
[12] Hovemeyer, D., Pugh, W.: Finding bugs is easy. ACM SIGPLAN Notices **39**(12), 92–106 (December 2004)
[13] Jet Propulsion Laboratory (JPL): JPL Java Coding Standard – JPL Institutional Coding Standard for the Java Programming Language. Version 1.0 (March 2014)
[14] MIRA Limited: Guidelines for the use of the C++ language in critical systems. MISRA-C++:2008 (June 2008)
[15] Moha, N., Gueheneuc, Y.G., Duchien, L., Le Meur, A.F.: DECOR: A Method for the Specification and Detection of Code and Design Smells. IEEE Transactions on Software Engineering **36**(1), 20–36 (2010)
[16] Object Management Group: Human-Usable Textual Notation (HUTN) Specification. Version 1.0, formal/04-08-01 (August 2004)
[17] Object Management Group: Architecture-Driven Modernization: Knowledge Discovery Meta-Model (KDM). Version 1.4, formal/16-09-01 (September 2016)
[18] Raibulet, C., Fontana, F.A., Zanoni, M.: Model-driven reverse engineering approaches: A systematic literature review. IEEE Access **5**, 14516–14542 (2017)
[19] Rodrigues Jr., E., Montecchi, L.: CCSL Metamodel, https://github.com/Elderjr/Coding-Conventions-Specification-Language (Accessed October 24, 2019)
[20] Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. IEEE Computer **39**(2), 25–31 (2006)
[21] Shahin, M., Ali Babar, M., Zhu, L.: Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. IEEE Access **5**, 3909–3943 (2017)
[22] Smit, M., Gergel, B., Hoover, H.J., Stroulia, E.: Code convention adherence in evolving software. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM). pp. 504–507 (2011)
[23] Smit, M., Gergel, B., Hoover, H.J., Stroulia, E.: Maintainability and source code conventions: An analysis of open source projects. Tech. Rep. TR11-06, University of Alberta, Department of Computing Science (June 2011)
[24] Software Engineering Institute – Carnegie Mellon University: SEI CERT Coding Standard, https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards/ (Accessed October 24, 2019)
[25] Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework (2nd Edition). Addison-Wesley Professional (December 2008)
[26] The MITRE Corporation: Common Weakness Enumeration (CWE). https://cwe.mitre.org/ (2019)
[27] Tian, Y.H.: String concatenation optimization on java bytecode. In: Proceedings of the International Conference on Software Engineering Research and Practice, SERP 2006, Volume 2. pp. 945–951. Las Vegas, Nevada, USA (2006)
[28] Voelter, M.: Best Practices for DSLs and Model-Driven Development. Journal of Object Technology **8**(6) (2009)
[29] Wagner, S., Jürjens, J., Koller, C., Trischberger, P.: Comparing bug finding tools with reviews and tests. In: IFIP International Conference on Testing of Communicating Systems. pp. 40–55 (2005)
[30] Wu, Y., Gandhi, R.A., Siy, H.: Using semantic templates to study vulnerabilities recorded in large software repositories. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems (SESS 2010). pp. 22–28. ACM, Cape Town, South Africa (2010)