Received 19 January 2023, accepted 22 February 2023, date of publication 13 March 2023, date of current version 17 March 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3256886

RESEARCH ARTICLE

A Model-Driven Approach for the Management and Enforcement of Coding Conventions

ELDER RODRIGUES JR.[1] JOSÉ D'ABRUZZO PEREIRA[2], (Graduate Student Member, IEEE), AND LEONARDO MONTECCHI[1,3]

[1]Institute of Computing, University of Campinas, Campinas, São Paulo 13083-852, Brazil
[2]University of Coimbra, Centre for Informatics and Systems of the University of Coimbra, Department of Informatics Engineering, 3030-290 Coimbra, Portugal
[3]Department of Computer Science, Norwegian University of Science and Technology, 7034 Trondheim, Norway

Corresponding author: Leonardo Montecchi (leonardo.montecchi@ntnu.no)

This work was supported in part by the European Union's Horizon 2020 Research and Innovation Program through the Marie Sklodowska-Curie under Grant 823788 "ADVANCE," in part by the São Paulo Research Foundation (FAPESP) under Grant 2018/11129-8 and Grant 2019/06799-7, in part by Portuguese "Fundação para a Ciência e a Tecnologia" (FCT) under Grant 2020.04503.BD, and in part by the Foundation for Science and Technology (FCT), I.P./MCTES through National Funds (PIDDAC), within the scope of CISUC R&D Unit - UIDB/00326/2020 or project code UIDP/00326/2020.

ABSTRACT Coding conventions are a means to improve the reliability of software systems, and they are especially useful to avoid the introduction of known bugs or security flaws. However, coding rules typically come in the form of text written in natural language, which makes them hard to manage and to enforce. Following the model-driven engineering principles, in this paper we propose an approach for the management and enforcement of coding conventions using structured models. We define the Coding Conventions Specification Language (CCSL), a language to define coding rules as structured specifications, from which checkers are derived automatically by code generation. To evaluate our approach, we run a thorough experiment on 8 real open-source projects and 77 coding rules for the Java language, comparing the violations identified by our checkers with those reported by the PMD static analysis tool. The obtained results are promising and confirm the feasibility of the approach. The experiment also revealed that textual coding rules rarely document all the necessary information to write a reliable checker.

INDEX TERMS Coding standards, coding conventions, model-driven engineering, domain-specific languages, static analysis.

I. INTRODUCTION

Coding conventions [1], also termed as *coding standards*, are guidelines for software development that impose constraints on how to write source code in a certain programming language. Depending on their purpose, coding conventions may cover different aspects of software development, including file organization, indentation, comments, naming conventions, but also recommend programming practices and principles, architectural best practices, etc.

Besides recommendations that do not affect the software behavior (e.g., naming of variables), many rules are introduced to enforce 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 [2]). In general, the adherence to precise coding rules avoids introducing known bugs, and it is a fundamental practice for ensuring the reliability of complex software systems.

Coding conventions are not static artifacts; rather, they evolve over time, following the introduction of new language features or the discovery of new vulnerabilities. Some coding conventions may be specific to a single company [3] or application domain [4], while others may be published as formal standards. It has been argued that coding conventions, in their current shape, offer limited benefit because of the difficulties in actually enforcing and managing them [5]. 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

The associate editor coordinating the review of this manuscript and approving it for publication was Claudia Raibulet.

code examples. Therefore, they cannot be processed automatically, and implementing a reliable checker for a new rule is often a complex development effort in itself. In fact, tasks like understanding whether a tool can check a certain rule, or writing a checker for a new rule, must be done manually.

Model-Driven Engineering (MDE) [6] is centered around the idea that information in the software development process should be represented as *structured* and *machine-readable* models. These models should be precise enough to be used for the automated generation of lower-level artifacts (e.g., source code), thus increasing automation and reducing the possibility of human mistakes. This paper proposes an approach to manage and enforce coding conventions through structured, machine-readable models. The main benefit of this approach is that checkers for new or customized coding rules are automatically generated based on a high-level specification of the rule. To the best of our knowledge, little work has been done in this direction.

In more details, the contributions of this paper are the following: i) we introduce a MDE-based approach for managing coding conventions as structured specifications; ii) we define the Coding Conventions Specification Language (CCSL), a Domain-Specific Language (DSL) to specify coding rules for the Java language; iii) we realize the automated generation of checkers from CCSL specifications; and iv) we evaluate our approach against rules supported by a popular static analysis tool.

The initial idea of this work was proposed in [7]; that work is expanded here with i) a refined version of the CCSL metamodel, ii) the addition of transformations that actually generate checkers, and iii) a detailed experimental evaluation, in which we compare the violations identified by our generated checkers with those identified by an existing static analysis tool.

The rest of the paper is organized as follows. We introduce the necessary background and motivation in Section II, followed by the related work in Section III. In Section IV we present the overall idea of our proposal. We define our specification language, the CCSL, in Section V, while usage examples are given in Section VI. In Section VII we discuss the generation of checkers from CCSL specifications, and in Section VIII we briefly introduce the prototype implementation. The experimental evaluation is reported in Section IX, followed by a discussion on the obtained results in Section X. Finally, Section XI concludes the paper.

II. BACKGROUND AND MOTIVATION

A. CODING CONVENTIONS

As highlighted by Smit et al. in [8], the term *code conventions* or *coding conventions* is used as a broad umbrella term for different kind of rules applying to source code. Other terms like "coding standard", "coding rules", etc., are frequently used as well. 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 $\omega \in \mathcal{L}$. We use this term to emphasize that the string may be part of a larger source code base.

A *coding rule* is a restriction on the source code that is not imposed by the grammar of the programming language. It states the conditions under which a code portion ω must be considered *invalid* for the purpose of a software project. A coding rule represents therefore a restriction on the possible ways to program software. More formally, a coding rule specifies a function $f : \mathcal{L} \rightarrow \{\texttt{valid}, \texttt{invalid}\}$. Some rules only have a formatting purpose, e.g., naming of variables or placement of brackets, and do not alter the behavior of the software; we refer to them as *coding style rules* [9]. In this work we focus instead on rules that affect non-functional properties, like security or performance. Note that the border is somehow blurred: in some languages (e.g., Python) formatting can alter the semantics of the code; similarly, naming of methods and variables can affect the functioning of libraries and frameworks.[1]

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 a single project or company, e.g., see [10], 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++ [11] or the JPL Java Coding Standard [12]).

B. LIMITATIONS IN CURRENT PRACTICE

In current practice, a wealth of coding rules exists. For example, when Smit et al. interviewed 7 software engineers asking about the most important practices for software maintainability, their answers resulted in 71 different coding rules and different opinions on their priority [8]. Furthermore, many companies define their own coding conventions, which may differ among different teams or even for individual developers. Reasons include different programming languages, different project requirements, or simply a client imposing specific restrictions.

Even established collections of coding rules like the SEI CERT Coding Standards [13] are continuously evolving, following changes to the agreed best practices. These changes can be due, for example, to the discovery of new vulnerabilities or the introduction of new programming constructs. In fact, even the last minor update to the Common Weakness Enumeration (CWE)[2] involved the addition of 29 new vulnerabilities and 142 major changes to existing ones[3]; many

[1]https://pmd.github.io/latest/pmd_rules_java_errorprone.html#junitspelling (Accessed March 10, 2023)

[2]The CWE is a database of *weaknesses* of software. However, in most cases it also provides *coding rules* that should be followed to avoid introducing the weakness itself. Actually, many rules in existing coding standards refer to CWE entries for justification.

of those will lead to the definition of new or updated coding rules as a prevention.

Typically, coding rules are specified using the natural language. In some cases, they are complemented with code examples to demonstrate the problem being addressed. While the support of automated tools has improved in recent years (as discussed later in Section III), many limitations still exist. First, tool support is fragmented: each static analysis tool (SAT) checks a different set of rules, often for a specific programming language. Except for very well-established coding standards, verifying all the rules of a certain set requires the combined application of multiple tools, and rarely all the rules can be verified automatically at all. Most often, a tool implements some kind of *adaptation* of an ambiguous coding rule described in natural language. Even if the documentation provides some clarification, inspecting the code of the checker is sometimes the only reliable information source. The need to inspect the code means that it is often difficult to understand which rules a tool can check, or vice versa which tools are able to check a particular rule.

Tool support is especially challenging when customized rules need to be enforced. Johnson et al. interviewed 20 developers, and 17 of them complained that many tools are not trivial to configure, even to the point of being "so hard to configure, they prevent you from doing anything" [14]. The need for a simplified way to define customized checks has also been highlighted in by Sadowski et al. [15], as a way to improve the "crowdsourcing" of source code analysis. The authors mention that in an attempt to integrate the formerly known FindBugs (currently SpotBugs) at Google, only a small number of employees understood how to write new checkers, because of the kind and depth of knowledge needed for the task. Furthermore, the code implementing static analysis is often complex and may itself contain bugs, to the point that specialized debugging platforms are needed [16].

In this paper we provide a first step towards the specification of coding rules in a structured way, enabling the automated generation of checkers and other artifacts. Differently from tools that allow adding new rules by explicitly writing the checker code, we adopt a model-driven approach, by targeting a more abstract specification of rules and the automated generation of checkers.

III. RELATED WORK

The basic way to verify adherence to coding rules is to perform manual code review. This is, of course, a costly process. Over the years, tools to automate the verification of coding rules have emerged. Typically, they are based on *static code analysis*, which consists in analyzing the source code for common defects and known bug patterns, without executing the software itself.

One of the first tools targeting the Java language was FindBugs (now SpotBugs) [17], which was initially created to detect null pointer defects. It has then evolved with the support of additional rules, and it features a plugin module that can be used to write customized detectors. Similarly, QJ-Pro [18] checks conformance to a predefined set of formatting rules, misuses of the Java language, code structure, etc. Unfortunately, from the available documentation, it has not been possible to precisely determine which rules are supported by this tool. The development of QJ-Pro seems to have stopped several years ago.

Several other tools exist; a survey on static analysis techniques and tools can be found in [19]. While most tools provide some kind of extension mechanism, adding or modifying rules is typically a complex task, which requires low-level manipulation of the abstract syntax tree (AST) of the code under analysis. PMD [20] and CheckStyle [21] are two of the most configurable tools for Java. In CheckStyle, customized checks are defined using the APIs provided by the tool, which basically consists in implementing the visitor pattern on the AST using Java code.[4] PMD offers a similar possibility, while also allowing the definition of customized checkers through XPath queries [22] on a XML-based representation of the AST.[5] XPath is a query language for XML documents; besides being very verbose, this solution still operates on the syntax of the Java language, meaning that the developer has to explicitly take into account every possible syntactical variation that leads to a violation of the rule.

The SonarQube platform [23] has become increasingly popular in recent years, mainly due to its superior reporting capabilities and integration with build tools. It can be considered more as an aggregator, providing a standardized interface to different kinds of plugins. However, extensions need still to be provided as XPath queries or Java plugins,[6] thus requiring considerable development effort.

Starting from similar motivations as ours, Goncharenko et al. defined a DSL for specifying coding rules for CSS (Cascading Style Sheets), a simple language for web design [24]. Allamanis et al. introduced Naturalize, a tool based on Natural Language Processing (NLP), which can analyze a code base to first recognize naming and formatting conventions adopted in the project and then to identify possible violations [25]. Naturalize only addresses coding style rules, and there is no way to specify customized rules that address security, for example. There is however a growing trend in applying machine learning for static analysis. For example, Ochodek et al. [3] use algorithms based on decision trees to identify violations to coding style rules for Java.

Nunes et al. benchmarks different SATs with respect to their ability to identify vulnerabilities [26]. The results highlighted that the best solution depends on the deployment scenario and on the class of vulnerability being targeted, thus

[3]https://cwe.mitre.org/data/reports/diff_reports/v4.6_v4.7.html (Accessed March 10, 2023)

[4]https://checkstyle.sourceforge.io/writingchecks.html (Accessed March 10, 2023)

[5]https://pmd.github.io/pmd/pmd_userdocs_extending_writing_rules_intro.html (Accessed March 10, 2023)

[6]https://docs.sonarqube.org/latest/extend/adding-coding-rules/ (Accessed March 10, 2023)

confirming the need for specialized coding rules depending on the project or application domain. In [27], the same authors show that combining multiple SATs does not necessarily improve the results over using a single tool. Wu et al. focus on structuring the relations between rules and vulnerabilities across different repositories [28]. However, 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 code smells [29], [30], which however are only one of the reasons that drive the definition of coding conventions. 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 [31].

In this respect, the Knowledge Discovery Metamodel (KDM) [32] is of particular relevance. KDM is a metamodel, defined by the Object Management Group (OMG), for representing existing software: it considers the physical and logical elements of software at various levels of abstraction, and the relations between them. The primary purpose of KDM is to act as an interchange format for interoperability between tools. MoDisco [33] provides a concrete implementation of the metamodel, and it supports the extraction of KDM models from software. However, KDM has been thought for modeling an entire software project in its details, while our objective is to model coding conventions at a higher level of abstraction.

The QL language [34] 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 [34], while our objective is to define a DSL for the specification of coding conventions. While QL is very powerful and supports arbitrary queries on source code, it is also necessarily verbose; conversely, we aim at a concise specialized language for coding conventions. Finally, we are proposing a complete MDE workflow, in which our metamodel is the basis for deriving other more detailed artifacts, of which QL queries could be an example.

IV. THE PROPOSED METHODOLOGY

In this section, we describe our proposal for the management and enforcement of coding convention, which is summarized in Figure 1.

Instead of using the natural language, coding rules are specified using a DSL especially tailored to define such structured specifications; we call this language the Coding Conventions Specification Language (CCSL). Textual descriptions of existing coding rules are translated into specifications in such language, while new rules can be created directly as CCSL models.

Having coding rules defined as machine-readable specifications means that they can be processed automatically. For example, before adopting a certain coding convention, the set of rules can be analyzed to identify those that are conflicting or redundant, thus reducing the number of alerts and avoiding inconsistencies. While this kind of analysis is possible, we do

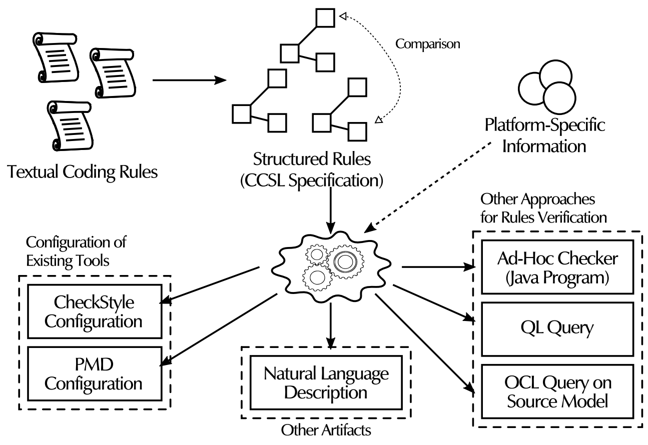

Textual Coding Rules
Comparison
Structured Rules (CCSL Specification)
Platform-Specific Information
Configuration of Existing Tools
CheckStyle Configuration
PMD Configuration
Natural Language Description
Other Artifacts
Other Approaches for Rules Verification
Ad-Hoc Checker (Java Program)
QL Query
OCL Query on Source Model

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

not focus on this aspect in the rest of the paper, and we leave it as a further research direction.

The main benefit of having CCSL specifications of coding rules is that they enable the automated generation of checkers, without the need to manually implement them or configure a SAT for the task. More in general, model transformations [35] can be applied to automatically derive different kinds of artifacts from CCSL specifications, for example configuration files for existing SATs.

As mentioned before, tools like CheckStyle or PMD can be configured to check custom rules. However, their configuration is complex and it requires deep knowledge of the tool. For example, in PMD custom rules are defined by writing XPath queries over the AST extracted from the source file, or by directly implementing a Java class that realizes the check. By defining a model transformation that generates such artifacts it is ensured that new coding rules that are specified with CCSL can be automatically checked, by deriving the proper configuration file for one of the existing tools.

However, this approach is limited by the capabilities and technical requirements of the target SAT. More in general, if coding rules are specified in a structured way, generators can be defined to verify them according to different strategies, as needed. This includes for example i) deriving source code to perform the verification programmatically, or ii) deriving queries in some specialized language, to be applied on an abstract representation of the code. In this paper we will derive checkers based on the Object Constraint Language (OCL) [36], a query language for models, published as an OMG standard. Deriving OCL queries is especially interesting when considering a MDE context and the capabilities of a platform like MoDisco [33], which can extract a structured model from the source code of an existing application.

Other kinds of artifacts could be derived from CCSL rules by automated generation, e.g., an explanation of the rule in natural language, or examples of source code portions that violate it. The element ''Platform-Specific Information'' in Figure 1 represents any additional information that may be required to generate a certain kind of artifact or to contextualize the rule to a given platform. For example, rule TSM00-J in [13] mentions "thread-safe methods". Properly verifying such rule requires knowledge of which methods are thread-safe in the target platform, or which language constructs make a method thread-safe (e.g., the *synchronized* keyword for Java). The simplest form to provide such information is a mapping between CCSL metaclasses and keywords of the target platform.

We demonstrate the feasibility of the approach by generating OCL-based checkers for Java source code. Details on the generation algorithm are given in Section VII. In the next section we introduce the CCSL.

V. CODING CONVENTIONS SPECIFICATION LANGUAGE

In this section we describe the metamodel of the CCSL language, which is used to provide structured specifications of coding rules. As mentioned above, in this paper we mainly focus on Java.

A. OVERVIEW

A CCSL specification of a coding rule describes the patterns that would *violate* the rule in the source code. That is, given a rule $f : \mathcal{L} \rightarrow \{\texttt{valid}, \texttt{invalid}\}$, 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) concepts of object-oriented programming; and iv) existing models of source code, in particular the aforementioned KDM, the MoDisco Java metamodel, and the Eclipse JDT DOM (an API to manipulate Java source code elements).

Differently from these languages, our metamodel is *not* a model of the source code. Instead, it is a model of *coding rules*, operating thus at a different level of abstraction. To understand the difference, note for example that the name of a method is not mandatory in our language, while it is clearly needed in a detailed model of source code. This is because we may need to specify rules that apply to any instance of the "Method" concept, independently of its name.

After identifying the constituting concepts, we defined the actual metamodel of our CCSL using the Ecore metamodeling language, which is part of the Eclipse Modeling Framework (EMF) [37]. The CCSL metamodel is organized in 6 main packages: *Core*, *NamedElements*, *DataTypes*, *Expressions*, *Statements*, and *Filters*, which are discussed in the following. The complete definition of the metamodel is available in the GitHub repository of the project [38].

B. CORE PACKAGE

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

An *AtomicRule* is defined by three properties:

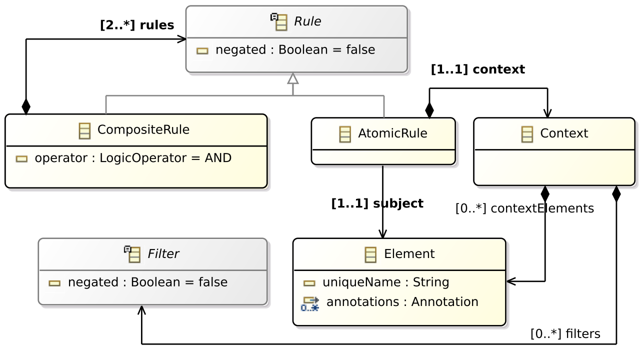


FIGURE 2. ***Core* package.**

Context. The *Context* describes the pattern to be searched in the source code, e.g., a class with name "Foo" that contains at least one method named "bar". The context of a rule must contain at least one *Element* and may contain a certain number of *Filter* instances.

Subject. The *subject* of a rule identifies the main element to which the rule applies, and it is always one of the elements defined in the *context*. In practice, the *subject* defines the element on which an alert is raised in case a violation is identified.

Filters. *Filters* are used to retain only elements of the context that fulfill specific conditions, e.g., classes whose name is matched by a regular expression. *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.

The *Element* metaclass (Figure 2) is the top of a hierarchy of metaclasses that represent different elements of the source code, e.g., classes, interfaces, methods, invocations, assignments, etc. The elements appearing in the rule's context (and their relations) specify the base pattern to be found in the code.

Complex rules can be specified as a *CompositeRule*, which is essentially a connector that combines multiple rules using Boolean logic operators.

C. *NamedElements* PACKAGE

The *NamedElements* package is shown in Figure 3. A *NamedElement* is an *Element* that has a name assigned by the programmer (e.g., variables, classes, methods, interfaces, etc.). The metaclasses in this package can be logically grouped in three categories: *Custom Types*, *Variables*, and *Methods*.

1) CUSTOM TYPES

This category includes the metaclasses representing custom types specified by the programmer (i.e., classes, interfaces, annotations, and enumerations). The *TypeDeclaration* metaclass is the top of the custom type hierarchy. The attribute "inheritance" defines whether a *TypeDeclaration* should be "final", "abstract", or "ANY", the latter meaning that it

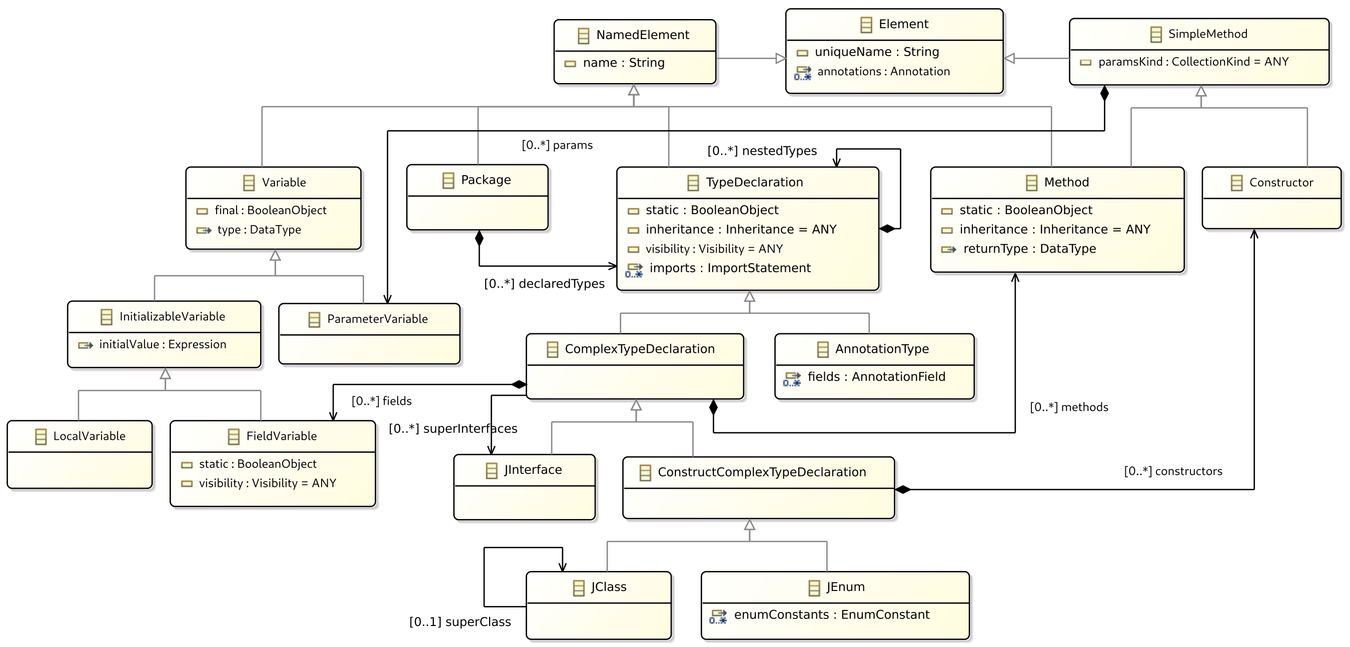


FIGURE 3. ***NamedElements* package.**

does not matter for the rule being specified. A *TypeDeclaration* can be a *ComplexTypeDeclaration* or an *AnnotationType*.

A *ComplexTypeDeclaration* represents custom types that can hold fields, variables, and methods, and that can implement/extend interfaces. An *AnnotationType* basically represents Java annotations. A *ComplexTypeDeclaration* can be a *JInterface* or a *ConstructComplexTypeDeclaration*. The latter represents complex types that can hold constructors, and it is extended by the *JClass* and *JEnum* metaclasses.

2) VARIABLES

The *Variable* metaclass is the superclass for all metaclasses representing variables. *InitializableVariable* represents a variable that can be initialized upon declaration, while the *ParameterVariable* metaclass represents a local variable corresponding to the parameter of a method. *InitializableVariable* is extendend by the *FieldVariable* and *LocalVariable*, corresponding to class attributes (fields) and local variables, respectively.

3) METHODS

The *SimpleMethod* metaclass abstracts concepts that are common to the *Method* and *Constructor* metaclasses. The attribute "params" defines the parameters that must exist in the method being specified. Detailing the name and kinds of the parameters is not mandatory: if no information is given on parameters, all the methods in the source code will be selected.

When the attribute "params" is specified, it means that we are looking for methods having those parameters in their signature. The semantics is that *at least* those parameters must exist in the method signature. To specify that a method must have *exactly* the given list of parameters, the "paramsKind" attribute should be set to "EXACT".

D. *DataTypes* PACKAGE

The *DataTypes* package is shown in Figure 4; its elements are used to define rules related to type specifications that can be found in a Java program. We consider two main kinds of *DataType*: *PrimitiveType* and *ObjectType*.

The *PrimitiveType* metaclass is extended by metaclasses representing Java primitive types (int, double, etc.); for simplicity, these metaclasses are not displayed in the figure. On the other hand, the *ObjectType* metaclass is extended by different metaclasses, each one representing a more specific kind of non-primitive type in Java.

The *ArrayType* metaclass represents the specification of an array type. The actual *type* of the elements of the array is given by a reference to another instance of *DataType*. The *TypeDeclaration* has been already introduced in the previous section, and it represents a custom type defined by the programmer (e.g., classes, interfaces, etc.).

The *ParameterizedType* metaclass represents the declaration of a type that is parameterized according to some other type, also known as generic types or generics in Java terminology. The actual type used as a parameter is specified by the *typeArguments* property. For example, `ArrayList<String>` is a *ParameterizedType* having a *JClass* with name "String" as its *typeArguments*.

The last two metaclasses, *TypeParameter* and *WildCardType*, are also related to generic types. *TypeParameter* represents the declaration of a generic type in methods or classes. For example, in the declaration of a method "`public <T> void foo(T var)`", the type `T` can be represented in

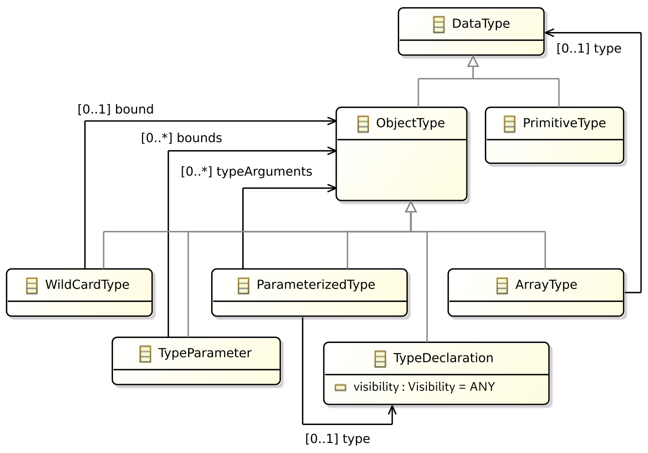


FIGURE 4. ***DataTypes* package.**

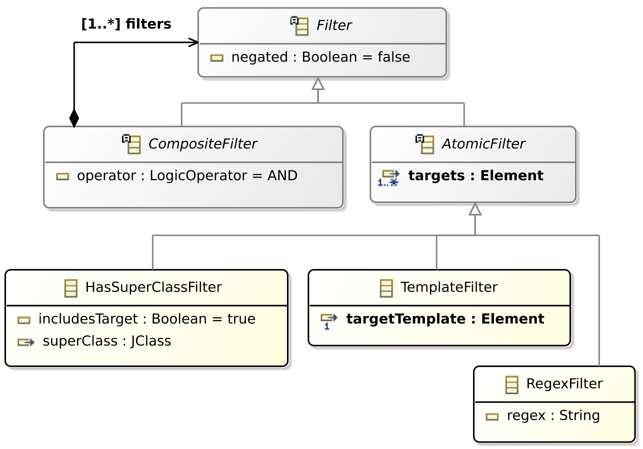


FIGURE 5. **Excerpt of the *Filters* package.**

CCSL as a an instance of *TypeParameter*. Finally, the *WildCardType* metaclass represents a wildcard of a type parameter, e.g., the question mark in `ArrayList<?>`.

E. EXPRESSIONS PACKAGE

This package contains language elements that return a value when evaluated, for example, method invocations, assignments, cast expressions, strings concatenation, etc. The root of the package is the *Expression* metaclass; like all the metaclasses that define a CCSL package, the *Expression* metaclass is a specialization of *Element*.

Most of the elements of the *Expressions* package can be directly mapped to expressions available in the Java language, and are thus self-explanatory (e.g., *CastExpression* represents a cast expression). The full list of classes in this package is available in the project repository [38]. However, a few of them deserve a more detailed description.

The *Invocation* metaclass is the common superclass of *MethodInvocation* and *ConstructorInvocation*. The generic concept of *Invocation* is useful when it is not necessary to specify whether a "normal" method or a constructor is invoked. Note that, contrary to intuition, the *Invocation* metaclass is not abstract. In fact, the concept of *Invocation* is supposed to be *concretely* used in CCSL specifications, and thus the metaclass is meant to be instantiated. The same applies to the other CCSL metaclasses that represent generic concepts, like the *Element* metaclass itself.

VariableAccess represents the access to the reference of a variable. For example, the variable declaration "`int b = c`" accesses the `c` variable to obtain its value. *DataTypeAccess* represents the access to a class (or enumeration). For example, the invocation "`ClassA.foo()`" accesses the *ClassA*, and then invokes its (static) method "`foo()`". In this context, CCSL considers `ClassA` an expression that returns the class itself.

ArrayInitializer represents the specification of an array of values, which is typically used in Java as alternative way to initialize an array. For example, the right-hand side of the assignment "`String[] array = {"a", "b", "c"}`" is represented in CCSL as an instance of the *ArrayInitializer* metaclass.

F. STATEMENTS PACKAGE

The *Statements* package includes metaclasses to represent commands that are executed in the source code. This also includes control flow instructions, i.e., "if", "while", "for", "try-catch", etc. As the other metaclasses at the top of a package, the *Statement* metaclass is an extension of *Element*.

Similarly to what has been discussed for the *Expressions* package, the CCSL metamodel includes a metaclass almost for all the statements that can appear in Java code. The complete list can be found in the documentation in the repository [38].

G. FILTERS PACKAGE

Filters can be used to identify specific elements within those selected by the context of the rule. Instances of filters are added to the *filters* attribute of the *Context* metaclass (refer to Figure 2).

The *Filter* metaclass is abstract, and it is the superclass of all the filters available in CCSL. To improve flexibility, filters adopt the idea of the Composite design pattern [39]: a *CompositeFilter* represents a list of filters combined by a Boolean operator, and *AtomicFilter* is an abstract metaclass that represents an entry-point to define new filters. A filter can also be *negated* or not.

Every *AtomicFilter* contains a list of elements to which the filter will be applied (*targets* attribute). Concrete filters are created by extending the *AtomicFilter* abstract metaclass. Figure 5 shows an excerpt of the *Filters* package, detailing the main structure of a filter and three of the filters available in the current version of CCSL. The full list of available filters, together with a brief description of their behavior, is reported in Table 1.

CCSL also includes a generic filter called *TemplateFilter*, designed to improve the flexibility of the metamodel. When none of the existing filters can specify the desired

TABLE 1. Filters that are currently available in the CCSL metamodel, with a brief explanation of their semantics. For each filter, the corresponding metaclass is `[Name]Filter`**.**

Name	Description
Count	Counts how many times a sample element *s* appears in a target element *obj*.
BlockFirstStatement	Checks if an element *e* appears as the first element in a target *Block* instance *b*.
BlockLastStatement	Checks if an element *e* appears as the last element in a target *Block* instance *b*.
HasSameReference	Checks if two elements are the same element in the source code.
HasSubclass	Checks recursively if a class *c* has a *sub* subclass.
HasSuperclass	Checks recursively if a class *c* has a *super* superclass.
ImplicitContainer	Checks recursively if an element *e* is contained by another element *c*, at any level of depth.
ImplicitContent	Checks recursively if an element *e* contains another element *c*, at any level of depth.
IsKindOf	Checks if a *DataType* or an *Expression* can be considered as being of type *t*. For example, an object of class “Foo” can be considered as being of type “Bar”, if class “Foo” extends class “Bar”.
IsTypeOf	Checks if a *DataType* or an *Expression* is exactly of type *t*.
Regex	Checks if the name of a *NamedElement* element matches a specific regular expression.
SameName	Checks if its target elements have the same name.
Template	Checks if the target element matches the provided template. See also the extended descripion in the text (Section V-G).

condition explicitly, the *TemplateFilter* can be configured with a “sample” of the kind of elements that should be selected. The filter then selects only the *target* elements that can be matched to the provided template.

VI. WRITING A CCSL SPECIFICATION

We provide now concrete usage examples of the CCSL metamodel. We use real rules from the SEI CERT Coding Standard [13] and from the PMD documentation [20], both for Java. Note that those rules are continuously evolving (which is one of the motivations behind this work); in this paper, we refer to version 6.21.0 of the PMD documentation.[7] Unfortunately, no version information is available for the SEI CERT Coding Standard.

As it is commonly done, we use a notation inspired to the UML Object Diagram [40] to display metamodel instances (i.e., CCSL specifications). We use colors to facilitate the interpretation of specifications: we indicate with *red* the root of the rule, with *green* its subject, with *yellow* the rest of the context, and with *cyan* the filters.

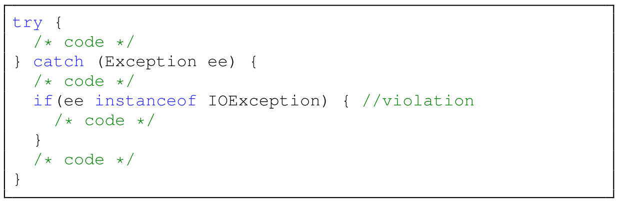
In the concrete implementation, CCSL specifications are Ecore models, which are by default stored in XMI (XML Metadata Interchange [41]) format, an XML-based format oriented towards automated processing. Our prototype implementation is further dicussed in Section VIII.

A. BASICS OF A CCSL SPECIFICATION

The main part of a rule is its *subject*, which identifies the *Element* to which it applies. While an *Element* may have various attributes (see Section V), in a typical specification, only a few of them will actually hold a value.

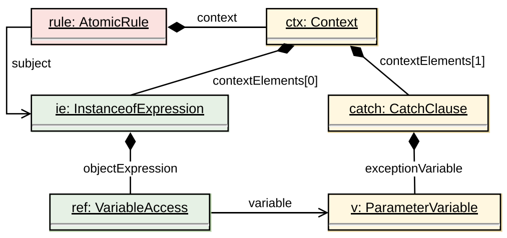
Consider the rule *AvoidInstanceOfChecksInCatchClause* from the PMD “Error Prone” ruleset:

[7]https://pmd.github.io/pmd-6.21.0/pmd_rules_java.html (Accessed March 10, 2023)



```
try {
  /* code */
} catch (Exception ee) {
  /* code */
  if(ee instanceof IOException) { //violation
    /* code */
  }
  /* code */
}
```

(a) Example of code that violates PMD rule *AvoidInstance-OfChecksInCatchClause*.



(b) CCSL specification of the rule.

FIGURE 6. *AvoidInstanceOfChecksInCatchClause* rule, from the PMD *Error Prone* ruleset. Example of violation (a), and the corresponding CCSL specification (b).

AvoidInstanceOfChecksInCatchClause.
“Each caught exception type should be handled in its own catch clause.”

Figure 6a illustrates a Java code that violates such rule: it catches a generic expression and then it checks the type of the exception with the *instanceof* operator. This means that the same `catch` block is being used to handle different kind of exceptions.

Following the checker implementation and examples provided by PMD itself, we consider a code portion invalid when an *instanceof* operator is applied to a *variable* declared as a parameter of a *catch* clause. Note that this is not exactly what is described in the textual description of the rule, as there are other ways to verify the type of an object at runtime (e.g., reflection).

The CCSL specification is given as an *AtomicRule*, whose *context* is composed of an *InstanceofExpression* and a *CatchClause* (see Figure 6b). The latter contains a *ParameterVariable v*, which represents the variable holding the exception being caught. The *subject* of the rule is the *InstanceofExpression*, where its left-hand side (*objectExpression*) is the access (*VariableAccess*) to the *ParameterVariable* declared in the *CatchClause*.

B. USING FILTERS

Similarly to rules, filters can have their own *context*. The context of a filter specifies elements that are used in the definition of the filter condition, but that are not part of the *context* of the rule itself.

Consider the rule MET09-J from SEI CERT [13]:

> 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."

Note that the actual coding rule is only the first sentence of the text, while the rest is an explication of the rationale. In fact, for the general case, determining whether the *hashCode* method actually returns the same value for all the equivalent objects is not feasible with static analysis, and it is actually an undecidable problem [42].

Figure 7 illustrates the specification of the above rule using CCSL. The element to be searched, which defines the *subject* of the rule, is a *JClass* that contains a method named "equals". However, only classes that define an "equals" method and do not define a "hashCode" method must be matched as violations. This can be achieved by applying a *TemplateFilter* on the *JClass* subject. The template in this case is a *JClass* that contains the "hashCode" method, and the filter is negated, meaning that it will exclude all the classes that do not match the template.

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 that make this rule ambiguous.

The first one concerns with the signatures of the *equals* and *hashCode* methods. The traditional signature of the *equals* method in Java is "`boolean equals(Object obj)`", but it is possible to overload it, for example as "`boolean equals(CustomClass obj)`". Whether the rule MET09-J should apply only to the original *equals* method or not is up to the interpretation of the reader. On the one hand, the problem addressed by the rule occurs on data structures from Java collections, which would call the first signature only. On the other hand, the rule does not specify the signature of the method. Actually, the original (verbatim) text of the rule mentions the "equals()" method, i.e., one without any parameters.

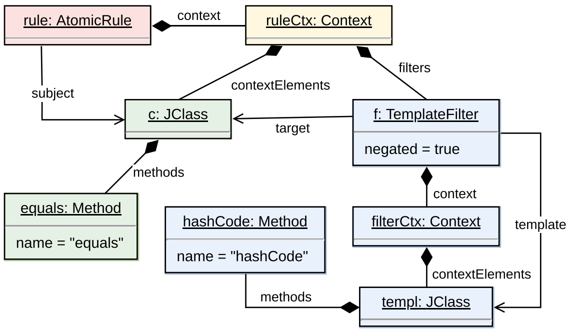


FIGURE 7. MET09-J CCSL specification.

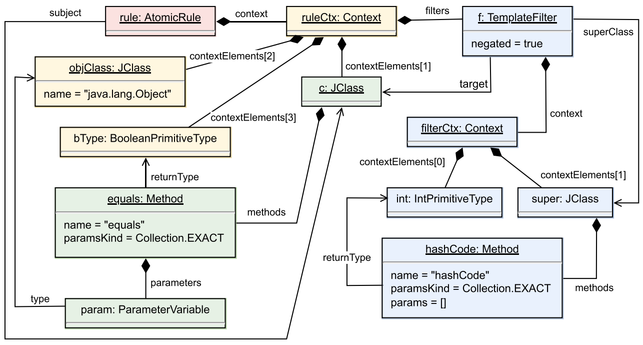


FIGURE 8. MET09-J refined CCSL specification.

Secondly, 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, but it is not clear if in this case a warning should be raised or not.

Figure 8 illustrates a revised specification of the MET09-J rule, considering these two aspects. This specification considers a violation if a class defines a method with signature "`boolean equals(Object obj)`", and neither the class itself nor its superclasses define a method with signature "`int hashCode()`". The filter being applied is the *HasSuperClassFilter*, which recursively checks if its target does *not* have (it is negated) a super class that defines the method "`int hashCode()`".

Which specification is the correct one is debatable, although most developers would agree that the second one is more accurate. Such ambiguity, however, highlights the importance of providing a structured specification of coding rules as opposed to textual ones.

The same rule exists in PMD with a more precise wording,[8] corresponding to the CCSL specification in Figure 8:

> *OverrideBothEqualsAndHashcode.*
> "Override both public boolean Object.equals(Object other), and public int Object.hashCode(), or override neither. Even if you are inheriting a hashCode() from a parent class, consider implementing hashCode and explicitly delegating to your superclass."

[8]https://pmd.github.io/pmd-6.21.0/pmd_rules_java_errorprone.html#overridebothequalsandhashcode (Accessed March 10, 2023)

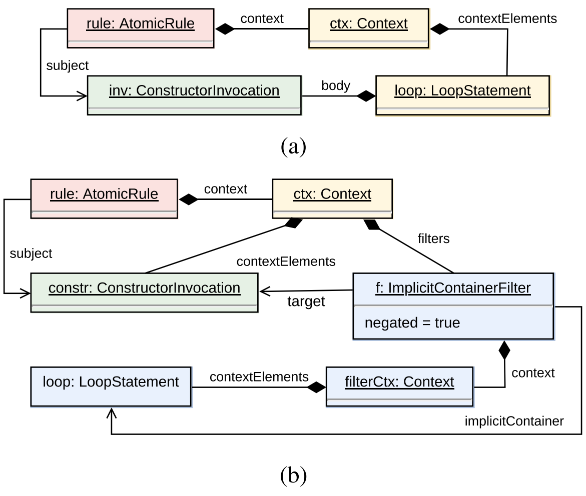


FIGURE 9. Two possible CCSL specifications of the *AvoidInstantiatingObjectsInLoops* PMD rule.

Note, however, that PMD itself does not provide any structured specification of this rule (not even an XPath query), but only an implementation of the checker as plain Java code.[9]

C. CONTAINMENT RELATIONS

An important feature of the CCSL is the possibility to specify complex relations between elements in the *context* of the rule. A particular case is the containment relation: when an element is contained by another element of the context, the derived checkers will look for an immediate containment relation between these objects. However, in some cases a weaker relation is needed.

Consider the PMD rule *AvoidInstantiatingObjectsInLoops* from the "Performance" coding convention:

> *AvoidInstantiatingObjectsInLoops.*
> "New objects created within loops should be checked to see if they can [be] created outside them and reused."

A possible CCSL specification of this rule is illustrated in Figure 9a, where a composition relation between a *LoopStatement* and a *ConstructorInvocation* is established. However, this specification is not accurate because it will recognize a violation only if a constructor call is *directly* contained by the loop and not, for example, by an *if* block that is in turn contained by the loop statement.

To specify that an element must be contained in another one at any level of depth, the *ImplicitContainerFilter* can be used. This filter specifies that, recursively, all the containers of its target should be compared with the sample element passed to the filter. Figure 9b illustrates the correct CCSL specification of the *AvoidInstantiatingObjectsInLoops* rule, using the *ImplicitContainerFilter*.

[9]https://github.com/pmd/pmd/blob/master/pmd-java/src/main/java/net/sourceforge/pmd/lang/java/rule/errorprone/OverrideBothEqualsAndHashcodeRule.java (Accessed March 10, 2023)

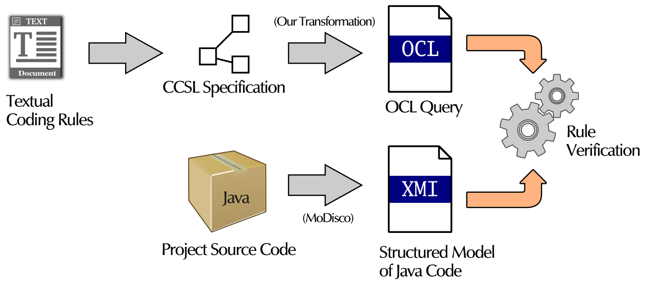


FIGURE 10. The workflow for the automated verification of rules specified with CCSL. OCL queries are automatically generated, and then applied to a structured model of the Java application.

VII. GENERATION OF CHECKERS FOR JAVA SOURCE CODE

This section is organized into three parts: we first discuss the approach adopted for the generation of checkers, we then detail the actual transformation algorithm, and finally we discuss an example based on a concrete CCSL specification.

A. OVERVIEW

The objective is to demonstrate the feasibility of our proposal, through a proof-of-concept tool able to automatically check CCSL specifications against Java code. Among all the possibilities highlighted in Figure 1, we decided to follow the OCL path: generating OCL queries that are then applied to a structured model of the Java source.

This choice was driven by both practical and strategical reasons. The first is that OCL has powerful querying contructs, as opposed to general-purpose programming languages. Generating an OCL query gives us greater flexibility for identifying specific elements in the source code. The second is the possibility to reuse existing MDE tools, in particular MoDisco [33] and the Eclipse OCL implementation [43]. Finally, we believe that this approach can simplify adapting the toolchain to other programming languages, because the generated checkers do not depend on the concrete syntax of the Java language.

The concrete workflow is illustrated in Figure 10. Verification of a CCSL rule involves three steps. First, an OCL query that identifies violations of the rule is automatically generated by model-to-text transformation. The generated query does not depend on the project to be checked: it can be generated once and applied multiple times on different projects. In the second step, a structured model of the project's source code is extracted using MoDisco. The model of the project extracted by MoDisco is also independent of the rule to be checked, and therefore it needs to be extracted only once. Finally, the OCL query is executed on the model of the target project.

B. TRANSFORMATION ALGORITHM

The generator of OCL queries from CCSL specifications has been developed using the Acceleo framework [44]. Acceleo is a tool to develop model-to-text transformations, which provides an implementation of the MOFM2T [45] specification defined by the OMG.

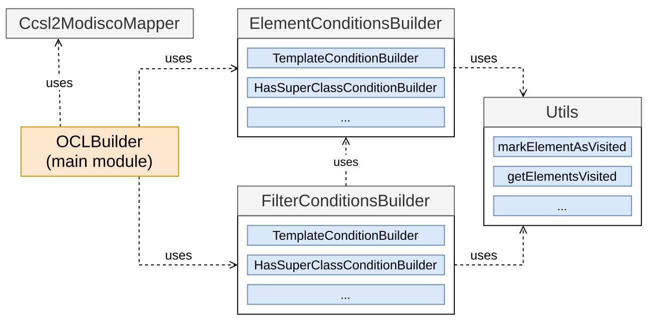


FIGURE 11. Architecture of the Acceleo transformation to generate OCL queries from CCSL specifications.

The Acceleo language is not a traditional object-oriented language. Instead, it is based on *modules* that expose *templates*. Templates, as the name implies, are blueprints of the text to be generated. Basically, they constitute of static text interleaved with control-flow instructions and access to the information contained in the source model (in this case, CCSL specifications).

Our transformation is organized in five main components (Figure 11): *OCLBuilder*, *CCSL2ModiscoMapper*, *ElementConditionsBuilder*, *FilterConditionsBuilder*, and *Utils*.

The main module and entry point of the transformation is the *OCLBuilder* module, which receives as input the CCSL specification for which the OCL query should be generated. The generation flow can be summarized in three steps, as follows.

1) MAPPING OF CCSL METACLASSES

Based on the kind of element used as *subject* of the CCSL specification, we identify the constructs that can represent it in the Java metamodel. Such information is obtained by accessing the *CCSL2ModiscoMapper* module, which stores the possible correspondencies. For example, the CCSL *LoopStatement* is mapped to multiple classes in the Java metamodel, corresponding to the `for`, `while`, or `do-while` statements.

With this information, we generate a skeleton of the OCL query to select all the source code elements that are instances of the mapped metaclasses, and we then proceed to the next step.

2) GENERATION OF SUBJECT CONDITIONS

In this stage, we generate all the OCL constraints for the conditions that must be satisfied by the *subject* of the rule. These constraints filter the instances that are selected by the base OCL skeleton, based on the properties and relations given in the *context* of the CCSL specification.

In this step, the module *ElementConditionsBuilder* works as a *Façade* [39], iterating on the elements related to the rule *subject*, and delegating the generation of OCL constraints for each CCSL *Element* to a specific (sub-)module. The iteration algorithm is based on a depth-first search on the graph of the CCSL specification, starting at the element pointed by the rule *subject*. OCL conditions are then recursively appended to the query while navigating the attributes and relations declared in the CCSL specification received as input.

When an element of the CCSL specification is visited, the following actions are performed. 1) We mark the element as visited to avoid infinite loops: if an element has already been visited, we do not generate its OCL condition again. 2) Using the `let` construct [36], we declare a new variable in the OCL query to hold a reference to the element being visited, in case we need to refer to it later in the rest of the query being generated. 3) We generate the conditions for the attributes of the visited element, and we add them to the OCL query. In this step, we also define which new nodes should be visited. In general, this includes all the elements referenced by any relation of the element, including its container if it exists.

The operations to get all the elements visited are provided by the *Utils* module. The same module also takes care of generating a unique name for each visited element, which is used for creating the variable in the `let` block in the second step.

3) GENERATION OF FILTERS CONDITIONS

In this stage, the transformation generates all the OCL constraints for all the filters that are specified in the CCSL rule, if any. The constraints are appended to the OCL query being generated, thus further reducing the number of instances that are selected by the final query. Each filter has its own strategy for generating the corresponding OCL code, since each one has its specific behavior.

The source code of the transformation is publicly available in the GitHub repository of the project [38].

C. RUNNING EXAMPLE

Figure 12 illustrates the OCL query generated from the specification of the *AvoidInstanceOfCheckInCatchClause* rule of Figure 6b. According to the algorithm described in the previous section, the query is generated as follows.

1) **Mapping of CCSL Metaclasses.** The subject of the rule is a CCSL *InstanceofExpression* element. In this case, it is directly mapped to the MoDisco *InstanceofExpression* metaclass (incidentally they have the same name). The base query skeleton then selects all the instances of the *InstanceofExpression* metaclass in the Java model (line 1).
2) **Generation of Subject Conditions for "*ie: InstanceofExpression*".** We now create the OCL conditions to satisfy the attributes and relations declared within the *subject*. We first generate a variable to hold the *InstanceofExpression* instance (`instanceOfExp_1`, line 2), and then we navigate the relation *objectExpression*, declared in the context, thus visiting the node "*ref: VariableAccess*".
3) **Generation of Subject Conditions for "*ref: VariableAccess*".** The visited element is referenced as

```
1   InstanceofExpression.allInstances()
2     ->select(instanceofExp_1: InstanceofExpression |
3       let varAccess_1: ASTNode =
4         instanceofExp_1.leftOperand->asOrderedSet()
5         ->closure(v: ASTNode |
6           if v.oclIsKindOf(ParenthesizedExpression) then
7           v.oclAsType(ParenthesizedExpression).expression
8           else v endif
9         )->last()
10      in varAccess_1 <> null
11      and varAccess_1.oclIsKindOf(SingleVariableAccess)
12      and let paramVar_1: ASTNode = varAccess_1
13          .oclAsType(SingleVariableAccess).variable
14      in paramVar_1 <> null and paramVar_1
15          .oclIsKindOf(SingleVariableDeclaration)
16      and let catchClause_1: OclAny =
17          paramVar_1.oclContainer()
18      in catchClause_1 <> null
19      and catchClause_1.oclIsKindOf(CatchClause)
20      and catchClause_1.oclAsType(CatchClause)
21          .exception = paramVar_1
22    )
```

FIGURE 12. The OCL query generated from the CCSL specification of the *AvoidInstanceofInCatchClause* rule (see Figure 6).

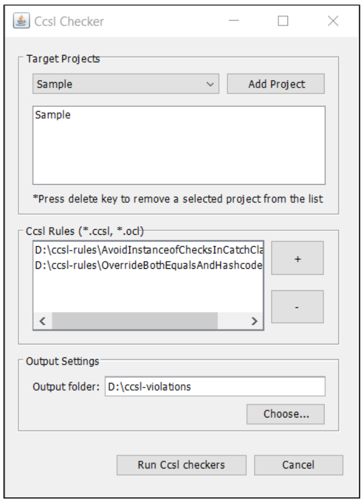
`varAccess_1` in the OCL query (line 3). Note that the query by default skips parentheses (lines 5–9). Hence, it does not matter if the `instanceof` expression is formatted as "`ee instanceof Type`" or instead, for example, as "`(ee) instanceof Type`".

In general, a CCSL *VariableAccess* element can be mapped to the *SingleVariableAccess*, *FieldAccess*, and *SuperFieldAccess* metaclasses in the Java model. However, because the *VariableAccess* is referencing a *ParameterVariable* in the CCSL specification (refer to Figure 6b), we know that only *SingleVariableAccess* is a valid match (line 11). We now visit the element "*v: ParameterVariable*", by navigating the *variable* relation.

4) **Generation of Subject Conditions for "*v: ParameterVariable*".** The *ParameterVariable* CCSL metaclass is mapped to the *SingleVariableDeclaration* metaclass in the Java model. This instance is referenced as `paramVar_1` (lines 12–13). We then create a type check condition to ensure the element is the expected one (lines 14–15), and we finally proceed to the next step by visiting the container of the *ParameterVariable* element (lines 16–17).

5) **Generation of Subject Conditions for "*catch: CatchClause*".** This is the last step for the processing of the *AvoidInstanceofInCatchClause* specification, as all the other CCSL elements have been visited already, and the rule does not contain filters.

The CCSL *CatchClause* metaclass is directly mapped to the *CatchClause* metaclass in the Java model. We generate a unique variable name to reference the *CatchClause* object (`catchClause_1`, line 18), and we also generate a type check condition, since the *oclContainer* returns an *OclAny* type (line 19). The last condition to be generated is to ensure that the `catchClause_1` and the `paramVar_1` elements are connected through the *exceptionVariable* relation as specified in Figure 6b (line 20–21).



Target Projects: Sample

Sample

*Press delete key to remove a selected project from the list

Ccsl Rules (*.ccsl, *.ocl)

D:\ccsl-rules\AvoidInstanceofChecksInCatchCla
D:\ccsl-rules\OverrideBothEqualsAndHashcode

Output Settings

Output folder: D:\ccsl-violations

FIGURE 13. Main window of the CCSL checker.

VIII. PROTOTYPE IMPLEMENTATION

The transformation algorithm described in the previous section is integrated into our prototype implementation of CCSL. This section describes its current status and our future plans; the source code is available on GitHub [38].

A. CCSL CHECKER PROTOTYPE

The current version of the CCSL Checker can verify one or more CCSL specifications against a Java project opened in the Eclipse workspace. The tool has been implemented as an Eclipse plugin because of its Eclipse-based dependencies, in particular EMF, MoDisco, Eclipse OCL, and Acceleo.

The tool accepts as input the coding rules defined as CCSL specifications or, alternatively, the generated OCL queries to be reused. CCSL specifications must be provided in XMI [41] format, which is the serialization format adopted by EMF. Figure 13 illustrates the main window of the prototype, where the user should select: i) the Java project(s) on which to run the rules; ii) the rules to be executed; and iii) the folder in which the resulting report will be saved.

When the user presses the "Run CCSL Checker" button, the prototype executes the following steps: i) it extracts the Java model of all selected projects; ii) it generates the OCL queries of selected rules; iii) it executes the OCL queries in each selected project; and iv) it generates one file for each rule, containing the identified violations with the file name and line number where they occurred.

B. TOWARDS A TEXTUAL NOTATION

One of the limitations of the current implementation is that CCSL rules are stored in XMI format. Although an expert EMF developer can easily read an XMI specification, the format is not intended to be directly manipulated by humans. For any metamodel defined with EMF, the framework provides a basic treeview editor that can be used to create model instances, ensuring that metamodel constraints are respected. While we used such editor to create the CCSL specifications used in this paper, it is not adequate for the casual user.

The EMF ecosystem features several tools to define a customized syntax for DSLs, either a graphical one (e.g., Sirius [46]) or a textual one (e.g., Xtext [47]). We are planning to use such facilities to provide a more user-friendly syntax to specify CCSL rules. We believe that a textual concrete syntax would have several advantages [48], for example, easier interaction with version control systems. Such developments are however improvements to the usability of the prototype, and not to its basic funcionality.

IX. EVALUATION

In this section we evaluate our proposal. The objective is to demonstrate the feasibility of the approach, that is, that *it is possible to automatically derive reasonably efficient checkers from coding rules specified using CCSL.*

A. METHODOLOGY

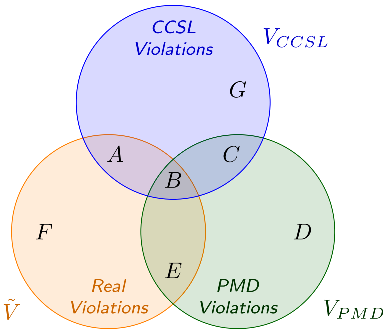
To evaluate the checkers generated from CCSL specifications, we compare their results with those of a popular SAT for Java, namely PMD [20]. The main reason for choosing PMD is its extensive documentation of the rules that are implemented. The evaluation approach can be summarized as follows: i) we selected a subset of the PMD rules for Java, ii) we specified them using CCSL, iii) we ran both the tools on multiple Java projects, and iv) we compared the results.

The results are compared by first classifying the individual violation reports (see Section IX-A1), then classifying rules based on how they are handled by the two tools in each individual project (see Section IX-A2), and then classifying rules according to the aggregated results on all the analyzed projects (see Section IX-A3).

Note that we do not aim to show that our tool is *better* than PMD at detecting violations, but instead that checkers generated with our method are comparable in performance to those of a widely-used, established, tool. PMD is being developed since at least 20 years ago (the first commit in its repository is from 2002), and it has been tested by a wide community on thousand of projects. Furthermore, it is known that different SATs often produce different results [26], [49], and judging the quality of alerts would require a deep analysis of each violation in the context of the analyzed project.

1) CLASSIFICATION OF VIOLATION REPORTS

We discuss the evaluation methodology with the aid of a Venn diagram (Figure 14). Given a certain coding rule applied on a certain source code, the set of *True Violations* ($\tilde{V}$) is the set of actually existing violations that should be reported by the tools. *CCSL Violations* (V_{CCSL}) is the set of violations reported by our CCSL Checker. Ideally, $V_{CCSL} \equiv \tilde{V}$, but in reality it may contain false positives and omit false negatives. Similarly, *PMD Violations* (V_{PMD}) is the set of violations reported by the PMD tool. With "false positive" we mean the event for which *the tool reports a nonexistent violation of the rule*. Note that this definition may be different from other interpretations; for example, a tool reporting an intentional, harmless violation of the rule is sometimes considered a false positive.



(a) Possible regions in which a violation report can be classified.

Class	Formula	Description
A	$(\tilde{V} \cap V_{CCSL}) \setminus V_{PMD}$	True positives reported by CCSL that are not reported by PMD (i.e., they are false negatives for PMD)
B	$\tilde{V} \cap V_{CCSL} \cap V_{PMD}$	True positives reported by both CCSL and PMD
C	$(V_{CCSL} \cap V_{PMD}) \setminus \tilde{V}$	False positives reported by both CCSL and PMD
D	$V_{PMD} \setminus (\tilde{V} \cup V_{CCSL})$	False positives reported by PMD only
E	$(\tilde{V} \cap V_{PMD}) \setminus V_{CCSL}$	True positives reported by PMD that are not reported by CCSL (i.e., they are false negatives for CCSL)
F	$\tilde{V} \setminus (V_{CCSL} \cup V_{PMD})$	False negatives for both CCSL and PMD (i.e., existing violation that are not reported by neither tool)
G	$V_{CCSL} \setminus (\tilde{V} \cup V_{PMD})$	False positives reported by CCSL only

(b) Definition of the different regions.

FIGURE 14. The classification of violation reports that will be used in our experiment.

As seen in the diagram (Figure 14a), the report of a violation can fall in one of seven distinct sets, depending on whether the violation actually exists, and on which tool raises it. A full description of these sets is given in Figure 14b.

2) CLASSIFICATION OF RULES

Based on the violations reported by the two tools *on a given Java project*, a certain rule can be classified in one of the following categories.

- **Exact.** The violations reported by CCSL and PMD are exactly the same, and at least one violation is reported by both tools. That is:

$$\big(V_{CCSL} = V_{PMD}\big) \wedge (V_{CCSL} \cup V_{PMD}) \neq \emptyset.$$

- **CCSL$^+$.** The CCSL Checker performed better than PMD. A rule is classified in this category when the following condition is met:

$$\big((G \cup E) = \emptyset \wedge (A \cup D) \neq \emptyset\big) \vee \big(V_{PMD} = \emptyset \wedge A \neq \emptyset\big).$$

  That is: i) there is no false positive reported by CCSL only and no true positive reported by PMD only ($G \cup E) = \emptyset$; and ii) there is at least one true positive reported by CCSL only or one false positive reported by PMD only, $(A \cup D) \neq \emptyset$; or alternatively iii) PMD did not report any violation, and CCSL reported at least one true positive (final part of the equation).

- **PMD$^+$.** PMD performed better than the CCSL checker. A rule is classified in this category when the following condition is met:

$$\big((A \cup D) = \emptyset \land (G \cup E) \neq \emptyset\big) \lor \big(V_{CCSL} = \emptyset \land E \neq \emptyset\big).$$

That is: i) there is no false positive reported by PMD only and there is no true positive reported by CCSL only, $(A \cup D) = \emptyset$; and ii) there is at least one true positive reported by PMD only or there is at least one false positive reported by CCSL only, $(G \cup E) \neq \emptyset$; or alternatively iii) CCSL did not report any violation and PMD reported at least one true positive (final part of the equation).

- **Partial.** A rule is classified as partial when there is only a partial overlap between the violations reported by the two tools and, therefore, no clear winner. More precisely, when CCSL reported at least one true positive that PMD did not report and vice versa, and both reported at least one violation. That is:

$$\big(V_{CCSL} \neq \emptyset \land A \neq \emptyset\big) \land \big(V_{PMD} \neq \emptyset \land E \neq \emptyset\big).$$

- **NoViolations.** Neither CCSL nor PMD did report any violation for the rule, that is:

$$V_{CCSL} = \emptyset \land V_{PMD} = \emptyset.$$

- **NoSpecification.** Independently of the violations reported by PMD, we classify a rule in this category when we could not provide a CCSL specification for it.

3) AGGREGATE CLASSIFICATION OF RULES

The above classification is established per project, i.e., a rule can be classified in different categories depending on the project under analysis. Actually, some rules have been classified as either CCSL$^+$ or PMD$^+$ depending on the project (see the Appendix). To obtain an aggregate view of the performance of our CCSL Checker with respect to PMD, we assign a final classification to each rule, based on the results obtained on the different projects.

Excluding the *NoSpecification* class, which is independent from the project, we establish the following aggregate classification for coding rules:

- *Same*: For all the analyzed projects the rule has been classified as either *Exact* or *NoViolations*.
- *Better*: For at least one of the analyzed projects the rule has been classified as *CCSL$^+$*, and as *Exact* or *NoViolations* for all the other projects.
- *Worse*: For at least one of the analyzed projects the rule has been classified as *PMD$^+$*, and as *Exact* or *NoViolations* for all the other projects.
- *Inconclusive*: All the remaining cases. When a rule is classified as inconclusive it means in some projects CCSL found violations that were not identified by PMD, and vice versa.

B. EXPERIMENT SETUP

As mentioned earlier, we based our analysis on version 6.21.0 of PMD, which was the latest one at the time we started our experiments. We selected a subset of the Java rules in the PMD documentation; in particular all those in the categories *Error Prone*, *Performance*, and *Multithreading*, yielding a total of 139 rules. We selected these three sets of rules because they address different characteristics of software, namely reliability, performance, and parallelism.

Then, we selected 8 real Java projects publicly available on GitHub, on which the rules shall be verified. The selected projects are listed in Table 2, including their names (linked to their GitHub repository), the version that we analyzed, and a short description. Three of those projects (*WebGoat*, *TeaStore*, and *WSVDBench*) were selected for a preliminary evaluation of this work, and then retained in the final experiments. They are mock implementations used for testing and benchmarking of web applications. The remaining 5 projects have been selected among the most "starred" projects on GitHub with a size less than 30MB, among the categories *Cryptography*, *Security*, and *Artificial Intelligence*. The size limit of 30MB has been applied to obtain an experiment of manageable size.

We first executed PMD on the 8 projects, configured to verify the previously selected rules, and recorded the results. Based on this, *we excluded from the rest of the experiment all the rules for which no violation was reported* in any of the 8 projects. This reduces the number of analyzed rules in a way fair to the experiment. Note that we only removed rules for which PMD did not find any violation, thus essentially removing rules that were going to be classified as *NoViolations* or *CCSL$^+$*. After this filtering, we retained 77 rules.

For each of these rules, we created a specification using CCSL, generated the OCL-based checker, and then executed it on the 8 projects. When we could not provide a CCSL specification of the rule, we classified it as *NoSpecification*. Then, we compared the results obtained with PMD and with CCSL, according to the methodology discussed in Section IX-A.

We note that, in this experiment, we did *not* know the ground truth; that is, we did not know *a priori* if a certain Java file contained violations of a given rule, nor where those violations were located. Still, when the two tools produced different results, we needed to understand if the additional violations being reported were true positives or not, to differentiate between the *CCSL$^+$*, *PMD$^+$*, and *Partial* classes. In those cases, we proceeded to a manual investigation of the reported violations, to understand if they had to be classified as true positive or false positive. This was not necessary for rules classified as *Exact*, *NoSpecification*, or *NoViolations*.

C. RESULTS

During the experiment, we were able to specify and generate checkers for 71 of the 77 selected rules (92.2%). The generated checkers have been executed on the 8 selected projects and the results compared with the output produced by PMD, as previously discussed. A summary of the obtained results is shown in Figure 15; for each rule, the figure shows the number of projects that resulted in a certain rule classification. For example, the rule *AvoidLiteralsInIfCondition* has been classified as *Exact* on 3 projects, as *CCSL*$^+$ on 2 projects, and as *PMD*$^+$ on the 2 remaining projects. Note that for many rules alerts were raised only on a subset of the selected projects.

The data used to draw the figure is available in the Appendix. The source of such data (raw tools output) is available on a separate GitHub repository that collects the data of the experiment [50], together with OCL queries generated from CCSL specifications.

Table 3 shows the same data, in this case organized per project. As before, note that a project typically contains violations only for a small subset of the analyzed rules. For each project, rules that did not generate violations are reported separately (*NoViolations* column), to focus the comparison of the two tools on relevant rules.

In all the projects, *for the majority of relevant rules we obtained equal or better results than PMD*. In fact, most rules were classified as *Exact* (47% on average) or *CCSL*$^+$ (14.4% on average). On the other hand, there are also cases for which we could not specify a rule using CCSL (6 rules out of 77), and cases where the PMD implementation has achieved better results than our approach (17.6% on average). For a small set of rules (1.5% on average) the two tools reported different violations.

While the results per project give an indication of the performance of the CCSL Checker in a real setting, they do not provide a clear view of the general behavior of the generated checkers. For example, a rule could behave as *CCSL*$^+$ only in one project, or the rules that are classified as *Exact* across different projects could, in fact, be always the same ones.

As explained in Section IX-A, the final classification step of our methodology classifies rules according to the behavior of the checkers across all the different project. Figure 16 shows the final aggregate results, in which each rule is classified as *Better*, *Same*, *Inconclusive*, *Worse*, or *NoSpecification* according to the results obtained across all the analyzed projects.

The results show that more than half of the rules are classified as *Same* (45.5%) or *Better* (14.3%), meaning that *almost 60% of the checkers generated from CCSL specifications performed equal or better than the corresponding PMD implementation*. We registered only 14 rules (18.2%) in which the PMD tool consistently performed better across the projects, and 6 rules (7.8%) that we could not specify with CCSL. The remaining 11 rules (14.3%) are classified as *Inconclusive*, meaning that the results were not consistent across the projects. However, this also means that, for all these 11 rules, the generated CCSL Checkers found more true positives or fewer false positives than PMD, at least in one project.

Further, we note that such aggregated classification abstracts away more complex results that also depend on the violations present in the projects. For example, rule *UnusedNullCheckInEquals* was classified as "Worse" even though it resulted in *PMD*$^+$ for only one project, because no violations were found in the other projects. Conversely, rule *EmptyCatchBlock* being classified as "Better" actually means that it resulted in *CCSL*$^+$ for 3 projects, and *Exact* for other 4 projects. Those detailed results are available in the Appendix.

D. THREATS TO VALIDITY

The experiment we performed is subject to some threats to validity, discussed in the following, together with the adopted mitigations.

As mentioned earlier, we do not know the ground truth for the analyzed projects. When the results between the two tools were different, we had to analyze whether the reported alerts were true positives or not. This is somehow subjective, due to the ambiguity in the textual description of rules. We mitigated this threat by using the PMD documentation as a reference and, when needed, by analyzing the implementation of the PMD checker. The two tasks of i) specifying the selected rules with CCSL and ii) analyzing the reported violations; were performed by two different authors separately. Doubts in the classification of rules were solved by discussion among all three authors.

For the same reason (lack of ground truth), the quality of our checkers was not analyzed in absolute terms, but only in comparison with PMD. This is of course a threat in case PMD results are of poor quality. However, PMD is one of the most popular SATs for Java, it has been under active development for more than 20 years, and it is used by hundreds of real projects. Furthermore, we applied the tools on real projects from GitHub; obtaining comparable results in such a real setting confirms the feasibility of the proposed approach.

In the experiment we only analyzed a limited set of rules. The analysis of results is extremely time consuming, especially when violation alerts differ between the tools. Typically, such cases involve analyzing corner cases, understanding *why* the two tools are reporting different results, verifying *what* the rule actually prescribes for that situation (if it is specified at all), and whether there are ambiguities in the rule description. We mitigated this threat by selecting multiple projects, so that different rules were activated in various of them and in different situations. Also, note that only a few rules present violations in a typical project, thus reducing the practical benefit of including additional rules to the experiment.

X. DISCUSSION

The checkers generated from CCSL specifications showed an accuracy comparable to that of the popular PMD tool. The results demonstrate that the proposed MDE approach is feasible and that it can generate checkers of good quality.

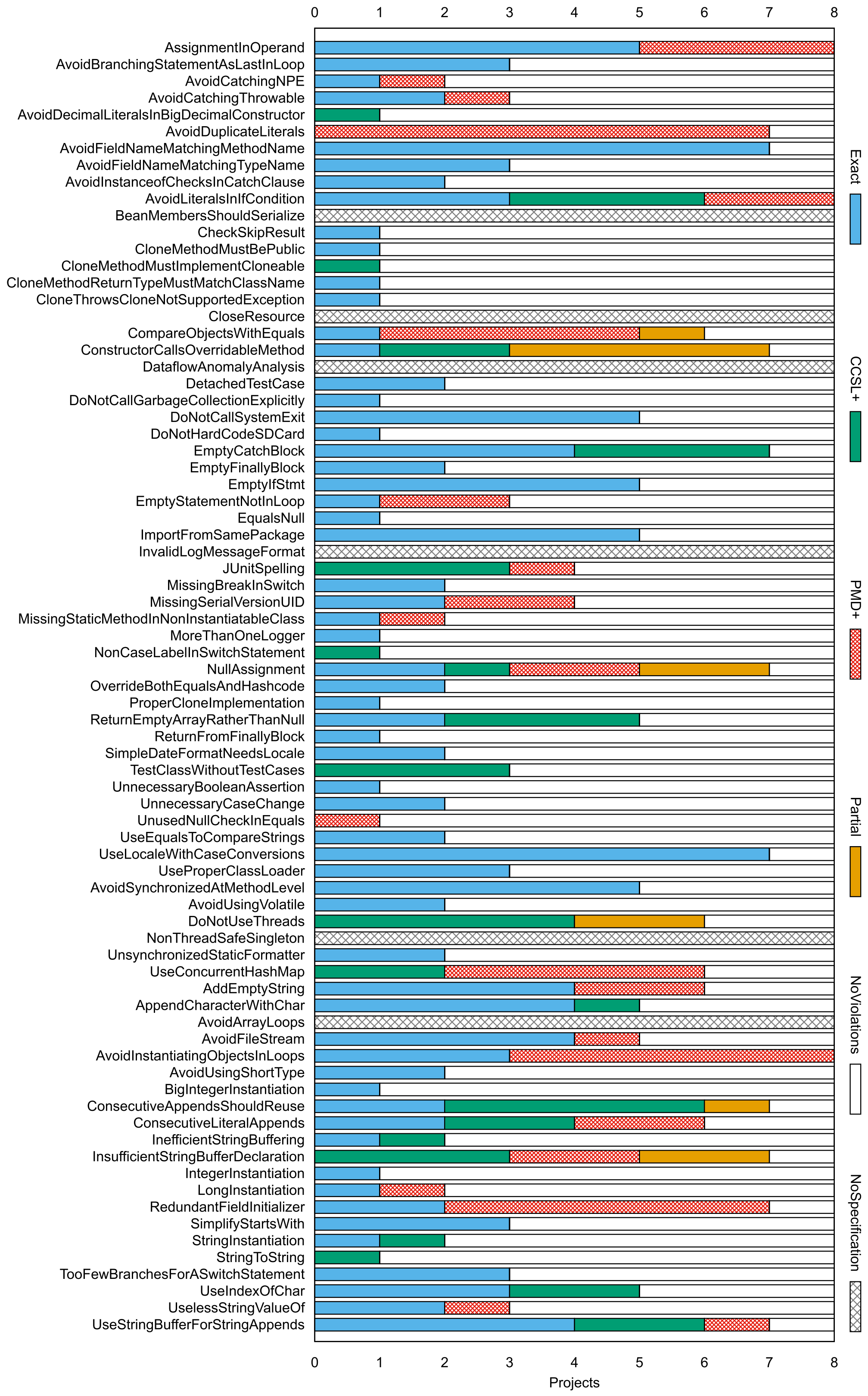


FIGURE 15. **Results of the experimental comparison between our CCSL checkers and PMD. Classification of rules across different projects.**

We note that our transformations are still part of a prototype, while on the other hand the PMD implementation of rules has been refined through more than 20 years of open source development.

TABLE 2. Software projects selected for the evaluation.

GitHub Repository	Commit	Description
cryptomator/cryptomator	a6e680e	Multi-platform transparent client-side encryption of files hosted in the cloud.
libgdx/gdx-ai	1e4973c	Artificial Intelligence framework for games based on libGDX.
microsoft/malmo	e79632b	Platform for Artificial Intelligence experimentation and research built on top of Minecraft.
DescartesResearch/TeaStore	5cab414	A micro-service reference and test application to be used in benchmarks and tests.
google/tink	bbc2f36	Multi-language, cross-platform, open source library that provides cryptographic APIs.
WebGoat/WebGoat	edd6b7d	A deliberately insecure web application maintained by OWASP designed to teach web application security lessons.
nmsa/wsvd-bench	95f08bb	Services that serve as workload of a benchmark for tools that detect SQL injection vulnerabilities in web services.
Netflix/zuul	b9f517c	Zuul is a gateway service that provides dynamic routing, monitoring, resiliency, security, and more.

TABLE 3. Results of the experimental comparison between our CCSL Checker and PMD. Classification of rules, organized per project. Please see Section IX-A for the adopted methodology.

	Rules Excluding *NoViolations*						*NoViol.*
Project	*CCSL*$^+$	*PMD*$^+$	*Exact*	*Partial*	*NoSpec.*	Total	
Cryptomator	4 [13%]	7 [23%]	11 [37%]	2 [7%]	6 [20%]	30	47
gdxAI	3 [13%]	6 [25%]	8 [33%]	1 [4%]	6 [25%]	24	53
Malmö	4 [10%]	9 [22%]	20 [49%]	2 [5%]	6 [15%]	41	36
TeaStore	7 [18%]	7 [18%]	20 [50%]	0	6 [15%]	40	37
Tink	8 [22%]	8 [22%]	14 [38%]	1 [3%]	6 [16%]	37	40
WebGoat	10 [21%]	5 [11%]	26 [55%]	0	6 [13%]	47	30
WSVDBench	1 [4%]	1 [4%]	17 [63%]	2 [7%]	6 [22%]	27	50
Zuul	7 [14%]	8 [16%]	26 [51%]	4 [8%]	6 [12%]	51	26

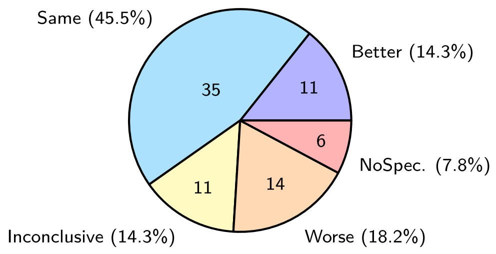


FIGURE 16. Results from the aggregate classification of rules. Please see Section IX-A for an explanation of the terms.

The main takeaways from this experiment reinforce our initial motivation for this work. Coding rules described in natural language seldom describe all the conditions that should be taken into account, so SAT developers need to be aware of possible special cases, which often lead to mistakes and omissions. In the analysis of mismatching reports, we encountered several cases caused by corner cases not considered in the rule description.

For example, rule *NullAssignment* says that one should not directly assign *null* to a variable, except in its initialization. In the implementation of this rule, PMD ignores variables marked as *final*, because their value cannot change after initialization. However, *it also ignores arrays that are marked as final*. In this case, the reference to the array itself cannot be changed, but the references to the individual elements of the array can still be set to *null*. This is what causes the rule to be classified as CCSL$^+$ for the gdxAI project. Note that the fact that the PMD implementation ignores variables marked as *final* was not mentioned in the rule documentation.

Similarly, the rule *ConstructorCallsOverridableMethod* mandates that a constructor should not invoke methods that can be overridden, because it "poses a risk of invoking methods on an incompletely constructed object". In the Tink project, the CCSL Checker reported a violation *in the constructor of an abstract class*; this violation was not reported by PMD. The constructor could not be invoked because the class is abstract; however, subclasses could still call the constructor with the *super* keyword. On the other hand, subclasses would also be in control of the overridden method, thus potentially reducing the risk of introducing defects. We decided to consider this case as a violation, because nothing about abstract classes was mentioned in the rule description.

Finally, it should be mentioned that some of the violations that were missed by our CCSL Checkers are due to a limitation of the underlying MoDisco tool that we use for the extraction of the model of the source code (see Figure 10). In fact, MoDisco is not aware of some constructs introduced in recent Java versions, and it simply discards such information from the model. In particular, this affected some violations that appeared inside lambda expressions [51]. Possibly, some of the rules classified as *Worse* (due to *PMD*$^+$ occurrences) would turn into *Same* if the model of the source code could be extracted correctly. In future versions of our tool we will investigate alternatives to MoDisco for the extraction of a model of source code. Projects like MLIR (Multi-Level Intermediate Representation) [52] look promising for this purpose.

XI. CONCLUSION

This paper proposed an approach for the management and enforcement of coding conventions based on model-driven engineering techniques. To the best of our knowledge, there

TABLE 4. Detailed results of the experimental comparison between our CCSL Checker and PMD.

	Cryptomator	gdxAI	Malmö	TeaStore	Tink	WebGoat	WSVDBench	Zuul	Rule Classification
Error Prone									
AssignmentInOperand	Exact	PMD+	PMD+	Exact	PMD+	Exact	Exact	Exact	Worse
AvoidBranchingStatementAsLastInLoop	∅	Exact	∅	∅	∅	Exact	∅	Exact	Same
AvoidCatchingNPE	∅	∅	∅	Exact	PMD+	∅	∅	∅	Worse
AvoidCatchingThrowable	∅	Exact	∅	∅	Exact	∅	∅	PMD+	Worse
AvoidDecimalLiteralsInBigDecimalConstructor	∅	∅	CCSL+	∅	∅	∅	∅	∅	Better
AvoidDuplicateLiterals	PMD+	PMD+	PMD+	PMD+	PMD+	PMD+	∅	PMD+	Worse
AvoidFieldNameMatchingMethodName	Exact	Exact	Exact	Exact	Exact	Exact	∅	Exact	Same
AvoidFieldNameMatchingTypeName	Exact	∅	∅	Exact	∅	Exact	∅	∅	Same
AvoidInstanceofChecksInCatchClause	∅	∅	∅	Exact	∅	∅	Exact	∅	Same
AvoidLiteralsInIfCondition	PMD+	CCSL+	Exact	PMD+	Exact	CCSL+	Exact	CCSL+	Inconclusive
BeanMembersShouldSerialize	—-	—-	—-	—-	—-	—-	—-	—-	—-
CheckSkipResult	∅	∅	∅	∅	Exact	∅	∅	∅	Same
CloneMethodMustBePublic	∅	∅	∅	∅	∅	∅	∅	Exact	Same
CloneMethodMustImplementCloneable	∅	∅	∅	∅	∅	∅	∅	CCSL+	Better
CloneMethodReturnTypeMustMatchClassName	∅	∅	∅	∅	∅	∅	∅	Exact	Same
CloneThrowsCloneNotSupportedException	∅	∅	∅	∅	∅	∅	∅	Exact	Same
CloseResource	—-	—-	—-	—-	—-	—-	—-	—-	—-
CompareObjectsWithEquals	∅	PMD+	PMD+	PMD+	Exact	PMD+	∅	Partial	Inconclusive
ConstructorCallsOverridableMethod	Partial	Partial	Partial	CCSL+	CCSL+	Exact	∅	Partial	Inconclusive
DataflowAnomalyAnalysis	—-	—-	—-	—-	—-	—-	—-	—-	—-
DetachedTestCase	∅	∅	∅	Exact	∅	Exact	∅	∅	Same
DoNotCallGarbageCollectionExplicitly	∅	∅	Exact	∅	∅	∅	∅	∅	Same
DoNotCallSystemExit	Exact	∅	Exact	∅	Exact	Exact	∅	Exact	Same
DoNotHardCodeSDCard	∅	∅	∅	∅	Exact	∅	∅	∅	Same
EmptyCatchBlock	Exact	∅	Exact	CCSL+	CCSL+	Exact	CCSL+	Exact	Better
EmptyFinallyBlock	∅	∅	∅	∅	∅	Exact	Exact	∅	Same
EmptyIfStmt	∅	Exact	Exact	∅	∅	Exact	Exact	Exact	Same
EmptyStatementNotInLoop	∅	∅	PMD+	∅	PMD+	Exact	∅	∅	Worse
EqualsNull	∅	∅	∅	Exact	∅	∅	∅	∅	Same
ImportFromSamePackage	Exact	Exact	∅	Exact	∅	Exact	Exact	∅	Same
InvalidLogMessageFormat	—-	—-	—-	—-	—-	—-	—-	—-	—-
JUnitSpelling	PMD+	∅	∅	CCSL+	∅	CCSL+	∅	CCSL+	Inconclusive
MissingBreakInSwitch	∅	∅	Exact	∅	∅	∅	∅	Exact	Same
MissingSerialVersionUID	PMD+	∅	∅	∅	∅	Exact	Exact	PMD+	Worse
MissingStaticMethodInNonInstantiatableClass	∅	∅	∅	∅	PMD+	Exact	∅	∅	Worse
MoreThanOneLogger	∅	∅	∅	∅	∅	∅	∅	Exact	Same
NonCaseLabelInSwitchStatement	∅	CCSL+	∅	∅	∅	∅	∅	∅	Better
NullAssignment	PMD+	CCSL+	PMD+	Exact	Partial	∅	Exact	Partial	Inconclusive
OverrideBothEqualsAndHashcode	∅	∅	∅	Exact	∅	∅	∅	Exact	Same
ProperCloneImplementation	∅	∅	∅	∅	∅	∅	∅	Exact	Same
ReturnEmptyArrayRatherThanNull	CCSL+	∅	Exact	∅	CCSL+	Exact	∅	CCSL+	Better
ReturnFromFinallyBlock	∅	∅	∅	∅	∅	∅	∅	Exact	Same
SimpleDateFormatNeedsLocale	∅	∅	∅	∅	∅	Exact	Exact	∅	Same
TestClassWithoutTestCases	∅	∅	∅	CCSL+	∅	CCSL+	∅	CCSL+	Better
UnnecessaryBooleanAssertion	∅	∅	∅	∅	∅	∅	∅	Exact	Same
UnnecessaryCaseChange	∅	∅	∅	∅	Exact	Exact	∅	∅	Same
UnusedNullCheckInEquals	∅	∅	PMD+	∅	∅	∅	∅	∅	Worse
UseEqualsToCompareStrings	∅	∅	Exact	∅	∅	Exact	∅	∅	Same
UseLocaleWithCaseConversions	Exact	∅	Exact	Exact	Exact	Exact	Exact	Exact	Same
UseProperClassLoader	∅	∅	Exact	∅	∅	Exact	∅	Exact	Same
Multithreading									
AvoidSynchronizedAtMethodLevel	Exact	∅	∅	Exact	Exact	∅	Exact	Exact	Same
AvoidUsingVolatile	Exact	∅	∅	∅	∅	∅	∅	Exact	Same
DoNotUseThreads	Partial	∅	CCSL+	CCSL+	CCSL+	CCSL+	∅	Partial	Inconclusive
NonThreadSafeSingleton	—-	—-	—-	—-	—-	—-	—-	—-	—-
UnsynchronizedStaticFormatter	∅	∅	∅	Exact	∅	∅	Exact	∅	Same
UseConcurrentHashMap	PMD+	∅	PMD+	CCSL+	PMD+	CCSL+	∅	PMD+	Inconclusive
Performance									
AddEmptyString	∅	∅	Exact	PMD+	Exact	PMD+	Exact	Exact	Worse
AppendCharacterWithChar	∅	∅	Exact	∅	CCSL+	Exact	Exact	Exact	Better
AvoidArrayLoops	—-	—-	—-	—-	—-	—-	—-	—-	—-
AvoidFileStream	∅	∅	Exact	PMD+	Exact	Exact	∅	Exact	Worse
AvoidInstantiatingObjectsInLoops	Exact	PMD+	PMD+	PMD+	PMD+	Exact	Exact	PMD+	Worse
AvoidUsingShortType	∅	Exact	∅	∅	∅	∅	∅	Exact	Same
BigIntegerInstantiation	∅	∅	Exact	∅	∅	∅	∅	∅	Same
ConsecutiveAppendsShouldReuse	CCSL+	∅	Exact	Exact	CCSL+	CCSL+	Partial	CCSL+	Inconclusive
ConsecutiveLiteralAppends	CCSL+	∅	Exact	∅	CCSL+	PMD+	PMD+	Exact	Inconclusive
InefficientStringBuffering	∅	∅	∅	Exact	∅	CCSL+	∅	∅	Better
InsufficientStringBufferDeclaration	CCSL+	∅	Partial	PMD+	CCSL+	PMD+	Partial	CCSL+	Inconclusive
IntegerInstantiation	∅	∅	Exact	∅	∅	∅	∅	∅	Same
LongInstantiation	∅	PMD+	∅	∅	∅	∅	Exact	∅	Worse
RedundantFieldInitializer	PMD+	PMD+	PMD+	Exact	PMD+	Exact	∅	PMD+	Worse
SimplifyStartsWith	∅	∅	∅	Exact	∅	Exact	∅	Exact	Same
StringInstantiation	∅	∅	Exact	∅	∅	CCSL+	∅	∅	Better
StringToString	∅	∅	∅	∅	∅	CCSL+	∅	∅	Better
TooFewBranchesForASwitchStatement	Exact	Exact	∅	Exact	∅	∅	∅	∅	Same
UseIndexOfChar	∅	∅	CCSL+	Exact	Exact	CCSL+	∅	Exact	Better
UselessStringValueOf	∅	∅	Exact	Exact	∅	∅	∅	PMD+	Worse
UseStringBufferForStringAppends	∅	Exact	CCSL+	CCSL+	Exact	Exact	Exact	PMD+	Inconclusive

—-: *NoSpecification* ∅: *NoViolations*

is little work in such direction. We defined a language, CCSL, that is used to specify coding rules as structured models, from which checkers are derived by automated transformations. One of the benefits of this approach is that checkers can be automatically generated from any rule that can be specified using our language.

We analyzed the effectiveness of the approach in a thorough experiment in which we applied 77 coding rules on 8 real open source projects written in Java. The experiment compared the violations reported by checkers generated from CCSL with those reported by the popular PMD tool. Overall, the results are promising and show the feasibility of the approach. In about 74% of the analyzed rules (57 out of 77), the checkers generated from CCSL specifications performed comparably or even better than PMD. Note that for many rules, the PMD checker consists of imperative code written in Java, and no high-level specification is available.

As future work, we plan to provide a simple textual notation to define CCSL specifications and integrate the checkers in the Eclipse IDE with inline notification of violations, thus providing a complete environment to developers. Furthermore, we plan to investigate the adaptation of the approach to other programming languages.

APPENDIX A

See Table 4.

REFERENCES

[1] M. Smit, B. Gergel, H. J. Hoover, and E. Stroulia, "Code convention adherence in evolving software," in *Proc. 27th IEEE Int. Conf. Softw. Maintenance (ICSM)*, Sep. 2011, pp. 504–507.

[2] Y. H. Tian, "String concatenation optimization on Java bytecode," in *Proc. Int. Conf. Softw. Eng. Res. Pract. (SERP)*, Las Vegas, NV, USA, vol. 2, 2006, pp. 945–951.

[3] M. Ochodek, R. Hebig, W. Meding, G. Frost, and M. Staron, "Recognizing lines of code violating company-specific coding guidelines using machine learning," *Empirical Softw. Eng.*, vol. 25, no. 1, pp. 220–265, Nov. 2019.

[4] R. Teixeira, E. Guerra, P. Lima, P. Meirelles, and F. Kon, "Does it make sense to have application-specific code conventions as a complementary approach to code annotations?" in *Proc. 3rd ACM SIGPLAN Int. Workshop Meta-Program. Techn. Reflection*, Boston, MA, USA, Nov. 2018.

[5] G. J. Holzmann, "The power of 10: Rules for developing safety-critical code," *Computer*, vol. 39, no. 6, pp. 95–99, Jun. 2006.

[6] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 25–31, Feb. 2006.

[7] E. Rodrigues and L. Montecchi, "Towards a structured specification of coding conventions," in *Proc. IEEE 24th Pacific Rim Int. Symp. Dependable Comput. (PRDC)*, Dec. 2019, pp. 168–177.

[8] M. Smit, B. Gergel, H. J. Hoover, and E. Stroulia, "Maintainability and source code conventions: An analysis of open source projects," Dept. Comput. Sci., Univ. Alberta, Edmonton, AB, Canada, Tech. Rep. TR11-06, Jun. 2011.

[9] N. Ogura, S. Matsumoto, H. Hata, and S. Kusumoto, "Bring your own coding style," in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Campobasso, Italy, Mar. 2018, pp. 527–531.

[10] G. J. Holzmann, "Mars code," *Commun. ACM*, vol. 57, no. 2, pp. 64–73, Feb. 2014.

[11] *Guidelines for the Use of the C++ Language in Critical Systems*, MISRA-C++:2008, MIRA Ltd., Warwickshire, U.K., Jun. 2008.

[12] "JPL java coding standard—JPL institutional coding standard for th java programming language," Jet Propuls. Lab., Tech. Rep., Mar. 2014.

[13] (2022). *SEI CERT Coding Standard*. [Online]. Available: https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards/

[14] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, San Francisco, CA, USA, May 2013, pp. 672–681.

[15] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at Google," *Commun. ACM*, vol. 61, no. 4, pp. 58–66, Mar. 2018.

[16] L. N. Q. Do, S. Kruger, P. Hill, K. Ali, and E. Bodden, "Debugging static analysis," *IEEE Trans. Softw. Eng.*, vol. 46, no. 7, pp. 697–709, Jul. 2020.

[17] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM Sigplan Notices*, vol. 39, no. 12, pp. 92–106, Dec. 2004.

[18] (2022). *QJ-Pro*. [Online]. Available: http://qjpro.sourceforge.net/

[19] A. Gosain and G. Sharma, "Static analysis: A survey of techniques and tools," in *Intelligent Computing and Applications*. New Delhi, India: Springer, 2015, pp. 581–591. [Online]. Available: https://link.springer.com/chapter/10.1007/978-81-322-2268-2_59

[20] (2022). *PMD*. [Online]. Available: https://pmd.github.io/

[21] (2022). *CheckStyle*. [Online]. Available: http://checkstyle.sourceforge.net/

[22] W3C, *XML Path Language (XPath) 3.1*, W3C Recommendation, World Wide Web Consortium, Cambridge, MA, USA, Mar. 2017.

[23] G. A. Campbell and P. P. Papapetrou, *SonarQube in Action*, 1st ed. Shelter Island, NY, USA: Manning, 2013.

[24] B. Goncharenko and V. Zaytsev, "Language design and implementation for the domain of coding conventions," in *Proc. ACM SIGPLAN Int. Conf. Softw. Lang. Eng.*, Oct. 2016, pp. 90–104.

[25] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2014, pp. 281–293.

[26] P. Nunes, I. Medeiros, J. C. Fonseca, N. Neves, M. Correia, and M. Vieira, "Benchmarking static analysis tools for web security," *IEEE Trans. Rel.*, vol. 67, no. 3, pp. 1159–1175, Sep. 2018.

[27] P. Nunes, I. Medeiros, J. Fonseca, N. Neves, M. Correia, and M. Vieira, "An empirical study on combining diverse static analysis tools for web security vulnerabilities based on development scenarios," *Computing*, vol. 101, no. 2, pp. 161–185, Feb. 2019.

[28] Y. Wu, R. A. Gandhi, and H. Siy, "Using semantic templates to study vulnerabilities recorded in large software repositories," in *Proc. ICSE Workshop Softw. Eng. Secure Syst.*, Cape Town, South Africa, May 2010, pp. 22–28.

[29] N. Moha, Y. G. Gueheneuc, L. Duchien, and A. F. L. Meur, "DECOR: A method for the specification and detection of code and design smells," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 20–36, Jan. 2010.

[30] H. Li and S. Thompson, "A domain-specific language for scripting refactorings in Erlang," in *Fundamental Approaches to Software Engineering (FASE)*. Berlin, Germany: Springer, 2012, pp. 501–515.

[31] C. Raibulet, F. A. Fontana, and M. Zanoni, "Model-driven reverse engineering approaches: A systematic literature review," *IEEE Access*, vol. 5, pp. 14516–14542, 2017.

[32] "Architecture-driven modernization: Knowledge discovery meta-model (KDM)," Object Manag. Group, Milford, MA, USA, Tech. Rep. formal/16-09-01, Version 1.4, Sep. 2016.

[33] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "MoDisco: A generic and extensible framework for model driven reverse engineering," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2010, pp. 173–174.

[34] P. Avgustinov, O. D. Moor, M. P. Jones, and M. Schäfer, "QL: Object-oriented queries on relational data," in *Proc. 30th Eur. Conf. Object-Oriented Program. (ECOOP)*, Dagstuhl, Germany, vol. 56, 2016, p. 2.

[35] N. Kahani, M. Bagherzadeh, R. James Cordy, J. Dingel, and D. Varró, "Survey and classification of model transformation tools," *Softw. Syst. Model.*, vol. 18, pp. 2361–2397, Aug. 2019.

[36] "Object constraint language," Object Manag. Group, Milford, MA, USA, Tech. Rep. formal/2014-02-03, Feb. 2014, Version 2.4.

[37] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, 2nd ed. Boston, MA, USA: Addison-Wesley, Dec. 2008.

[38] E. Rodrigues Jr., and L. Montecchi. (2022). *CCSL Metamodel*. [Online]. Available: https://github.com/Elderjr/Coding-Conventions-Specification-Language

[39] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Abstraction and reuse of object-oriented design," in *Proc. Eur. Conf. Object-Oriented Program.* Berlin, Germany: Springer, 1993, pp. 406–431. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-47910-4_21

[40] "OMG unified modeling language (OMG UML)," Object Manag. Group, Milford, MA, USA, Tech. Rep. formal/2017-12-06, Version 2.5.1, Dec. 2017.

[41] "XML metadata interchange," Object Manag. Group, Milford, MA, USA, Tech. Rep. formal/15-06-07, Jun. 2015, Version 2.5.1.

[42] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Trans. Amer. Math. Soc.*, vol. 74, no. 2, pp. 358–366, Feb. 1953.

[43] Eclipse Modeling Project. (2022). *Eclipse OCL (Object Constraint Language)*. [Online]. Available: https://projects.eclipse.org/projects/modeling.mdt.ocl

[44] (2022). *Acceleo*. [Online]. Available: https://www.eclipse.org/acceleo/
[45] "MOF model to text transformation language, v1.0," Object Manag. Group, Milford, MA, USA, Tech. Rep. formal/2008-01-16, Jan. 2008.
[46] F. Madiot and M. Paganelli, "Eclipse sirius demonstration," in *Proc. MoDELS Demo Poster Session Co-Located ACM/IEEE 18th Int. Conf. Model Driven Eng. Lang. Syst. (MoDELS)*, in CEUR Workshop Proceedings, vol. 1554, V. Kulkarni and O. Badreddin, Eds. Ottawa, ON, Canada, Sep. 2015, pp. 9–11. [Online]. Available: https://ceur-ws.org/Vol-1554/
[47] M. Eysholdt and H. Behrens, "Xtext: Implement your language faster than the quick and dirty way," in *Proc. ACM Int. Conf. Companion Object Oriented Program. Syst. Lang. Appl. Companion (SPLASH)*, Tahoe, NV, USA, 2010, pp. 307–309.
[48] M. Voelter, "Best practices for DSLs and model-driven development," *J. Object Technol.*, vol. 8, no. 6, pp. 1–24, 2009.
[49] A. Arusoaie, S. Ciobaca, V. Craciun, D. Gavrilut, and D. Lucanu, "A comparison of open-source static analysis tools for vulnerability detection in C/C++ code," in *Proc. 19th Int. Symp. Symbolic Numeric Algorithms Sci. Comput. (SYNASC)*, Timisoara, Romania, Sep. 2017.
[50] E. Rodrigues Jr., J. D'Abruzzo Pereira, and L. Montecchi. (2022). *CCSL vs. PMD Experiment*. [Online]. Available: https://github.com/Elderjr/ccsl-vs-pmd-experiment
[51] D. Mazinanian, A. Ketkar, N. Tsantalis, and D. Dig, "Understanding the use of lambda expressions in Java," *Proc. ACM Program. Lang.*, vol. 1, pp. 1–31, Oct. 2017.
[52] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim. (CGO)*, Seoul, South Korea, Feb. 2021, pp. 2–14.

ELDER RODRIGUES JR. received the master's degree in computer science from the State University of Campinas, Brazil, in 2020. His master's work was primarily focused on formalizing coding standards using model-driven engineering techniques. He is currently a Senior Java Developer leading a team with Inter, a digital bank in Brazil.

JOSÉ D'ABRUZZO PEREIRA (Graduate Student Member, IEEE) received the B.Sc. degree in computer science from the State University of Campinas (Unicamp), Brazil, and the dual M.Sc. degree in information technology and software engineering from the University of Coimbra (UC) and Carnegie Mellon University. He is currently pursuing the Ph.D. degree in information science and technology with UC. He is a member of the Software and System Engineering (SSE) Group, CISUC. His research interests include security and vulnerability detection, static code analysis, software project management, software quality, and self-adaptive systems.

LEONARDO MONTECCHI received the bachelor's and master's degrees from the University of Florence, Italy, in 2007 and 2010, respectively, and the Ph.D. degree in computer science, systems, and telecommunications from the University of Florence, in 2014. He was an Assistant Professor with the State University of Campinas, Brazil, from 2017 to 2021. He is currently an Associate Professor with the Norwegian University of Science and Technology, Trondheim, Norway. His expertise revolves around the modeling of complex systems, including formal models, probabilistic models, and model-driven engineering. His research interests include modeling as a support to the verification and validation of safety-critical and mission-critical systems. He regularly serves as a reviewer for international conferences and journals in the area of system and software reliability.