Software & Systems Modeling
https://doi.org/10.1007/s10270-019-00729-w

REGULAR PAPER

Applying static code analysis for domain-specific languages

Iván Ruiz-Rube[1] · Tatiana Person[1] · Juan Manuel Dodero[1] · José Miguel Mota[1] · Javier Merchán Sánchez-Jara[2]

Received: 7 September 2018 / Revised: 4 February 2019 / Accepted: 21 March 2019

Abstract
The use of code quality control platforms for analysing source code is increasingly gaining attention in the developer community. These platforms are prepared to parse and check source code written in a variety of general-purpose programming languages. The emergence of domain-specific languages enables professionals from different areas to develop and describe problem solutions in their disciplines. Thus, source code quality analysis methods and tools can also be applied to software artefacts developed with a domain-specific language. To evaluate the quality of domain-specific language code, every software component required by the quality platform to parse and query the source code must be developed. This becomes a time-consuming and error-prone task, for which this paper describes a model-driven interoperability strategy that bridges the gap between the grammar formats of source code quality parsers and domain-specific text languages. This approach has been tested on the most widespread platforms for designing text-based languages and source code analysis. This interoperability approach has been evaluated on a number of specific contexts in different domain areas.

Keywords Text-based languages · Static analysis · Model-driven interoperability · Xtext · SonarQube

1 Introduction

Quality management comprises a set of activities to plan, control, assure and improve the quality of the organisations, products or services [31]. Increasing the quality of software artefacts can increase customer satisfaction and provide competitive advantages. Static program analysis is a helpful technique to verify software quality features. In this vein, the use of automatic tools to analyse source code enables to discover diverse code smells and potential errors. In combination with quality reference models and methods (e.g. SQALE [23]), static analysis tools are used to extract indicators of quality attributes from software artefacts, which quantifies and makes the amount of technical debt measurable [3].

Typically, quality inspection tools are prepared to parse and analyse the source code of general-purpose language (GPL) components. Inspection tools provide developers with software metrics of quality attributes, such as security, compatibility, portability, efficiency, maintainability, reliability, usability and functional suitability [19].

Domain-specific languages (DSL) are, in contrast to GPL, computer languages that are meant for experts in a number of diverse domains [11]. Professionals in many different areas can develop and describe problem solutions that take the shape of source code artefacts by means of a DSL that is especially defined for their own field and discipline.

However, can source code quality analysis tools be applied to software artefacts that are developed at the DSL code level? Providing facilities to evaluate the quality of the DSL source code artefacts by applying well-known static analysis techniques might be an additional factor that contributes to a successful adoption of DSLs [15].

Nevertheless, in spite of the facilities provided by the text-based DSL authoring tools for accelerating the development of language grammars, turning these into the proper format to be recognised by code quality platforms is still a time-consuming and error-prone task. It is also difficult to keep

Communicated by Prof. Tony Clark.

✉ Iván Ruiz-Rube
ivan.ruiz@uca.es

[1] Department of Computer Engineering, University of Cádiz, Puerto Real, Cádiz, Spain

[2] E-LECTRA Research Group, University of Salamanca, Salamanca, Spain

Published online: 06 April 2019

grammars consistent between the DSL and the quality analysis platform while evolving the language.

To tackle this issue, a model-driven interoperability strategy is proposed in this paper. The rest of this work is structured as follows. Section 2 introduces the foundations of this research and the related works. Section 3 describes the proposed solution to bridge the grammar formats to analyse text-based DSLs from within code quality platforms. In Sect. 4, some descriptive use scenarios of different text-based languages are presented to evaluate the solution, namely a language aimed at designing algorithms and a language for designing music sheets. Afterwards, a case study (Sect. 5) and usability test (Sect. 6) are provided for illustrating the potential of using code quality platforms for the above languages from the end user perspective. Finally, Sect. 7 discusses the results and draws the conclusions of this research.

2 Foundations and related works

The static analysis of source code is a useful technique to verify the quality of software artefacts. Static analysis can be used to estimate the technical debt [22]. In the software engineering field, this analogy claims that doing things quick and carelessly may cause future costs and effort to fix and redesign a software project. This risk can be reduced through refactoring, which can be applied to source code but also to UML models, requirement specifications and software architecture descriptions [30], among other artefacts. The debt metaphor is also suitable to describe the consequences of rash developments in a variety of disciplines, besides software production or maintenance, such as building architecture, musical projects, hardware design and so on. Thus, static analysis might become a useful quality analysis technique beyond source code.

2.1 Static analysis in domain-specific languages

There are examples of the application of static analysis to DSLs. For example, Mandal et al. [24] proposed a static analyser for three specific languages of real-world industrial systems control applications, namely IEC 61131-3, EDDL and RAPID. Their main objective is to ensure correctness of safety-critical software. In addition, Prähofer et al. [28] discussed the role of static code analysis for PLC programming and developed a specific tool for analysing IEC 61131-3 programs.

Besides, Infrastructure as Code (IaC) using system configuration languages, such as Puppet, is extensively used for data centre provisioning. In this vein, Sharma et al. [36] analysed a large amount of GitHub repositories containing Puppet code and, as a result, they proposed a catalog of implementation and design configuration smells. Besides, Shambaugh et al. [35] have created a verification tool to implement complete and scalable determinacy analysis for Puppet.

Static analysis is also used for business processes models, such as BPMN or BPEL, by validating well-formedness constraints and deriving static approximations of behavioural properties with data-flow analysis [34] or by generating verifiable Petri net-based models [14].

The use of interactive quality platforms for analysing source code, such as SonarQube,[1] Codacy[2] or SQuORE,[3] is increasingly gaining attention in the developers' community [39]. SonarQube is particularly prevalent, partly due to its flexibility to parse a great number of GPLs and its open source nature. SonarQube generates reports about code duplication, coding standards, potential bugs and diverse software metrics that enable the user to draw quality evolution graphs, as well as to compute the technical debt due to code quality.

The issue is that quality platforms, such as SonarQube, are initially intended to check programs written in general-purpose languages, such as Java, C++ or Python, among others. None of the existing quality platforms for analysing source code are applied to software artefacts that are developed at the DSL code level.

2.2 Developing text-based DSLs

Two main options arise when developing a DSL, namely language exploitation and language invention [25]. On the one hand, an existing GPL, in combination with an application library, can be a useful tool to describe domain-specific issues. On the other hand, DSL inventions can be designed with no manifest commonalities with existing languages. Additionally, DSLs can be classified according to their concrete syntax (e.g. text-based or visual).

In recent years, various tools have arisen to easily develop external DSLs, both visually and textually. Many of these tools fall into the model-driven software engineering approach, which promotes model design, development and transformation to conduct the software process life cycle [7].

The number of existing domain-specific text-based languages has significantly grown, especially those created with Xtext. At the time of writing, there are more than 5000 grammar files based on Xtext[4] in GitHub.[5] JetBrains MPS,[6] Rascal Metaprogramming Language,[7] MontiCore Language Work-

[1] https://www.sonarqube.org.

[2] https://www.codacy.com.

[3] http://www.squoring.com.

[4] http://www.eclipse.org/Xtext.

[5] https://github.com.

[6] https://www.jetbrains.com/mps.

[7] http://www.rascal-mpl.org.

bench[8] and the Spoofax Language Workbench[9] are other examples of tools for designing text-based DSLs.

Xtext [5] is part of the Eclipse Modeling Project, which gathers most of the libraries, frameworks and tools to deal with the design and development of external DSLs. This framework is the most widespread environment to develop programming languages and text-based DSLs by means of a dedicated language metamodel. The languages generated with this framework can be deployed to end users as part of different environments, which can be enriched with code completion, syntax colouring, navigation and other features.

Extended Backus–Naur Form (EBNF) [18] is a meta-language that specifies the syntax of a linear sequence of symbols. It designs both the logical structure of the language (abstract syntax) and its textual representation (concrete syntax). It is used to formally express context-free grammar of computer programming languages. In the particular case of languages made with Xtext, their grammars are designed by using their own specific language, which is very similar to EBNF, to describe the concrete syntax of the new language and how it is mapped to an in-memory semantic model. Xtext uses the well-known ANTLR parser [27], which implements an LL top-down parse algorithm for a subset of context-free languages. The grammar language is composed of terminal rules, parser rules, data type rules, hidden terminal symbols (to guide the parser in case of ambiguities) and syntactic predicates. All of the terminal, parser and data type rules use EBNF expressions. For example, the Xtext code necessary to design a DSL to describe directed graphs is shown in Listing 1. Besides the initial declarations for the grammar, several non-terminal rules are included to represent the graph structure. A graph is composed of one or more (operator +) nodes and zero or more (operator *) edges. A node is declared by including only an identifier, and a edge is declared by referencing two cross-references to some of the previous nodes. Finally, an identifier is any number of letters, underscores and numbers ('0'..'9'), but does not start with a number.

Listing 1 Xtext grammar for the Graph DSL

```
grammar org.example.GraphDSL
generate graphdsl http://example.org/GraphDSL
Graph:
  {Graph}
  'nodes' (nodes+=Node)+ 'end'
  'edges' (edges+=Edge)* 'end';
Node:
  name=ID;
Edge:
  source=[Node] '->' target=[Node];
terminal ID:
    ('a'..'z'|'A'..'Z'|'_')
    ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
```

From this grammar, Xtext can generate an editor for typing and formatting code such as the Listing 2.

Listing 2 Use example of the Graph DSL

```
nodes
  A
  B
  C
end
edges
  A -> B
  B -> C
end
```

Additionally, it is possible to infer grammars for DSLs by examining the positive and negative samples of sentences of an unknown language. In this way, non-programmer domain experts can write their own DSL by simply providing examples of their DSL programs. A survey of the algorithms used to support this grammatical inference process is explained in [37].

2.3 Issues with source code quality parsers

Language users often appreciate informative feedback as they type. In this vein, Xtext supports automatic validations to assure the validity of the documents according to the grammar and scope designed for the language. The development of custom constraints and validations is also possible in the Xtext framework. However, it does not provide facilities that are readily available in regular code quality platforms, such as historical data or dashboard building. Other languages and frameworks that support validation, such as the Object Constraint Language (OCL) or the Epsilon Validation Language (EVL), also lack these visual capabilities.

SonarQube includes a mechanism for parsing new languages and including new rules to check programs either written with the built-in languages or with new ones. To evaluate the quality of DSL source code built with Xtext or to compute code metrics, we must develop all of the software components required by SonarQube to parse and query the Abstract Syntax Tree (AST) of the source code parts. However, the development of language parsers in SonarQube is quite different from the usual model-driven practices.

The SonarQube extension mechanism follows a language exploitation approach [25]; that is, instead of writing a grammar by using EBNF expressions, SonarQube parsers are implemented as Java classes that use a specific library called SonarSource Language Recogniser[10] (SSLR). SSLR is a library that provides everything required to create lexers and parsers for analysing a piece of source code. This library is based on the Parsing Expression Grammar (PEG) formalism [10], hence providing an API (through a *LexerfulGrammarBuilder* class) to define the set of terminal and non-terminal

[8] http://www.monticore.de.

[9] http://www.metaborg.org.

[10] http://docs.sonarqube.org/display/DEV/SSLR.

symbols, the parsing expressions and the root rule (starting expression). Unlike context-free grammars, PEGs enable to unambiguously express the language syntax , which can be parseable in linear time [29].

Listing 3 shows a snippet of the main Java class required by SonarQube to parse DSL graphs as defined in Listings 1 and 2. This fragment contains only the Java class definition that implements the SonarQube grammar. Two specific classes have to be added to define the textual and symbolic keywords. Additionally, several Java classes which are required to implement the SonarQube plug-in architecture are also needed.

As can be seen, the grammar format for designing DSL editors with Xtext diverges from the grammar format used by SonarQube. Therefore, to accomplish the recognition of our DSLs by the quality platform is time-consuming and error-prone, especially when it comes to maintaining the consistency of grammars while evolving the language. Afterwards, several AST visitors should be implemented to analyse quality rules or compute measures.

3 Model-driven interoperability strategy

In addition to code generation, the model-driven software engineering approach empowers other applications, such as systems interoperability. With this approach, it is possible to bridge the gap between the Xtext and SonarQube grammar formats, as described in prior works [33].

Xtext grammar specification is declarative, whereas SonarQube grammars must be written as Java code. Figure 1 shows the strategy for bridging the grammar formats of both tools. First, a Text-to-Model (T2M) syntactic transformation should be carried out to obtain a model (conforms to the Xtext metamodel) from the DSL's grammar file. Second, a Model-to-Model (M2M) process transforms the Xtext grammar model elements to those of a Java model, according to the Sonar grammar. Lastly, a Model-To-Text (M2T) process serialises the final model from the previous step with the syntax required by SonarQube.

There are several alternatives to perform the M2M process, such as the by-example approach [21], which is similar to the query-by-example and programming-by-example techniques. Other approaches are concerned with the concrete syntax details by pairing productions of the source and target grammar [4]. However, the most common procedure to define the model transformation process begins with, essentially, defining references between elements of each abstract syntax by means of a formal language.

The Xtext grammars are mapped to both structural and behavioural artefacts in SonarQube. Firstly, a Xtext grammar file corresponds to a Java class extending *com.sonar.sslr.api.Grammar*. This class reflects the language structure and must contain, per each Xtext rule, one reference attribute to an object implementing the *com.sonar.sslr.api.Rule* interface. All the atomic and non-terminal symbols of the Xtext grammars are mirrored to Java *enum*-typed constants, which implement the *TokenType* or *GrammarRuleKey* interfaces. Besides, the Builder design pattern [12] is used to create and assemble the rules, by means of the *LexerfulGrammarBuilder* class methods. In this case, the *rule*, *is* and *sequence* methods must be used to define the grammar rules. The *isOneOfThem* method is invoked for referencing keywords, whereas rule calls and cross-references are simply managed by accessing class fields of the target grammar. The alternative paths in the grammars require invocations to the *firstOf* method. Finally, assignments and groups correspond to invocations to the *zeroOrMore* or *optional* methods, depending on their cardinality. Figure 2 depicts the abstract mapping between elements of both grammars.

This general interoperability strategy has been simplified for its implementation. It is not necessary to implement the first T2M process from scratch because Xtext already provides an injector to support it. To do this, the user has to configure a specific property in the Modeling Workflow Engine (.mwe) file that comes with every Xtext project. Consequently, when the user runs the process represented by that workflow, an XMI version of the grammar description is automatically generated. For the sake of simplicity, the second and third transformation processes have been implemented as a single step (highlighted with a thicker line in Fig. 1). This prevents the need to use a model transformation engine, such as ATL [20], and a subsequent Java serialiser. Therefore, an Acceleo[11] module and a set of code templates were developed to generate the Java source code components required by SonarQube to parse a new language. In addition, all the boilerplate code required by SonarQube is automatically generated. In this approach, the transformation step is key. Thus, the non-functional aspects must be considered to ensure the quality of the transformations [2,38].

Two Eclipse IDE plug-ins have been developed and made available at the *Xtext2Sonar* website.[12] The plug-ins extend the command options available in the contextual menu associated with the .xtext grammar files. The source code of Java projects will be automatically generated by launching the *Xtext2Sonar/Generate SonarQube Plugin* command. The selected Java projects have to be packaged with Apache Maven[13] to obtain the .jar files. Eventually, they can be deposited in any existing SonarQube installation to provide support for the new language.

One of the benefits of using a model-driven approach is to be able to automatically generate source code. In our case, the

[11] http://www.eclipse.org/acceleo.

[12] https://github.com/TatyPerson/Xtext2Sonar.

[13] https://maven.apache.org.

Listing 3 Java Sonar grammar for the Graph DSL

```
import com.sonar.sslr.api.GenericTokenType.IDENTIFIER;

public enum GraphDSLImpl implements GrammarRuleKey {
  GRAPH,NODE,EDGE;
  private LexerfulgrammarBuilder lexer;

  public static Grammar create(GraphDSLConfiguration conf) {
    lexer = LexerfulGrammarBuilder.create();
    generateRules();
    lexer.setRootRule(GRAPH);
    return lexer.buildWithMemoizationOfMatchesForAllRules();
  }

  private static void generateRules() {
    lexer.rule(ID).is(lexer.isOneOfThem(IDENTIFIER, IDENTIFIER));
    lexer.rule(GRAPH).is(b.sequence(lexer.isOneOfThem(GraphDSLKeyword.NODES,
        GraphDSLKeyword.NODES),
      lexer.oneOrMore(NODE),
      lexer.isOneOfThem(GraphDSLKeyword.END, GraphDSLKeyword.END),
      lexer.isOneOfThem(GraphDSLKeyword.EDGES, GraphDSLKeyword.EDGES),
      lexer.zeroOrMore(EDGE),
      lexer.isOneOfThem(GraphDSLKeyword.END, GraphDSLKeyword.END)));
      lexer.rule(NODE).is(ID);
      lexer.rule(EDGE).is(b.sequence(IDENTIFIER,
      lexer.isOneOfThem(GraphDSLPunctuator.ARROW, GraphDSLPunctuator.ARROW),
        IDENTIFIER));
    }
}
```

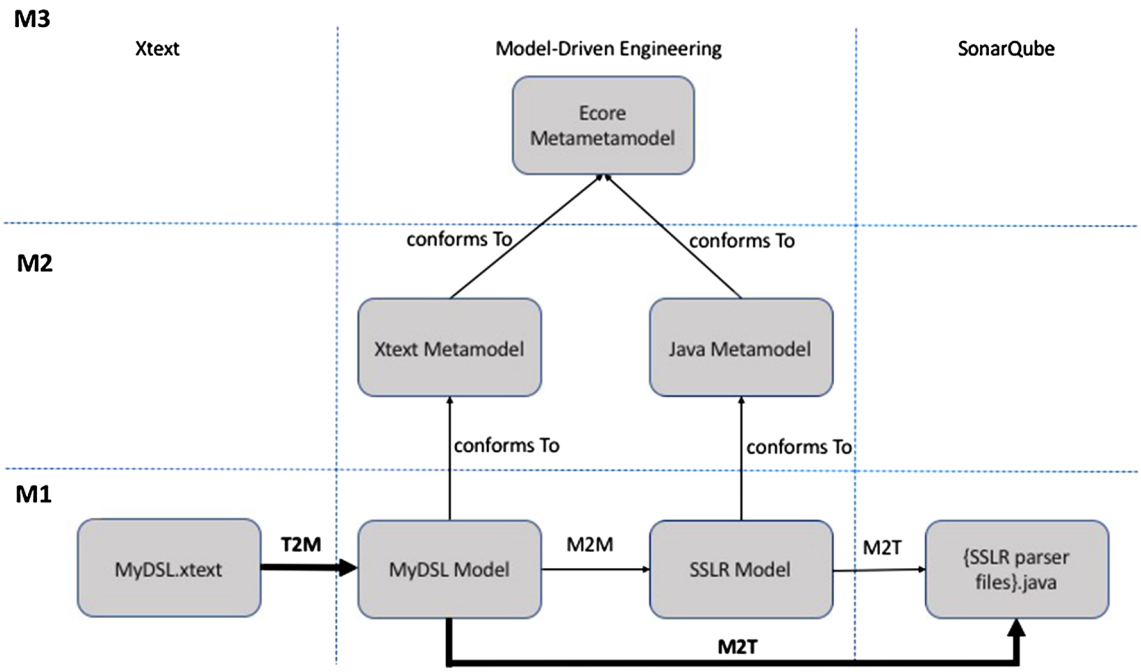

Fig. 1 Model-driven interoperability strategy (based on the global schema defined in [7])

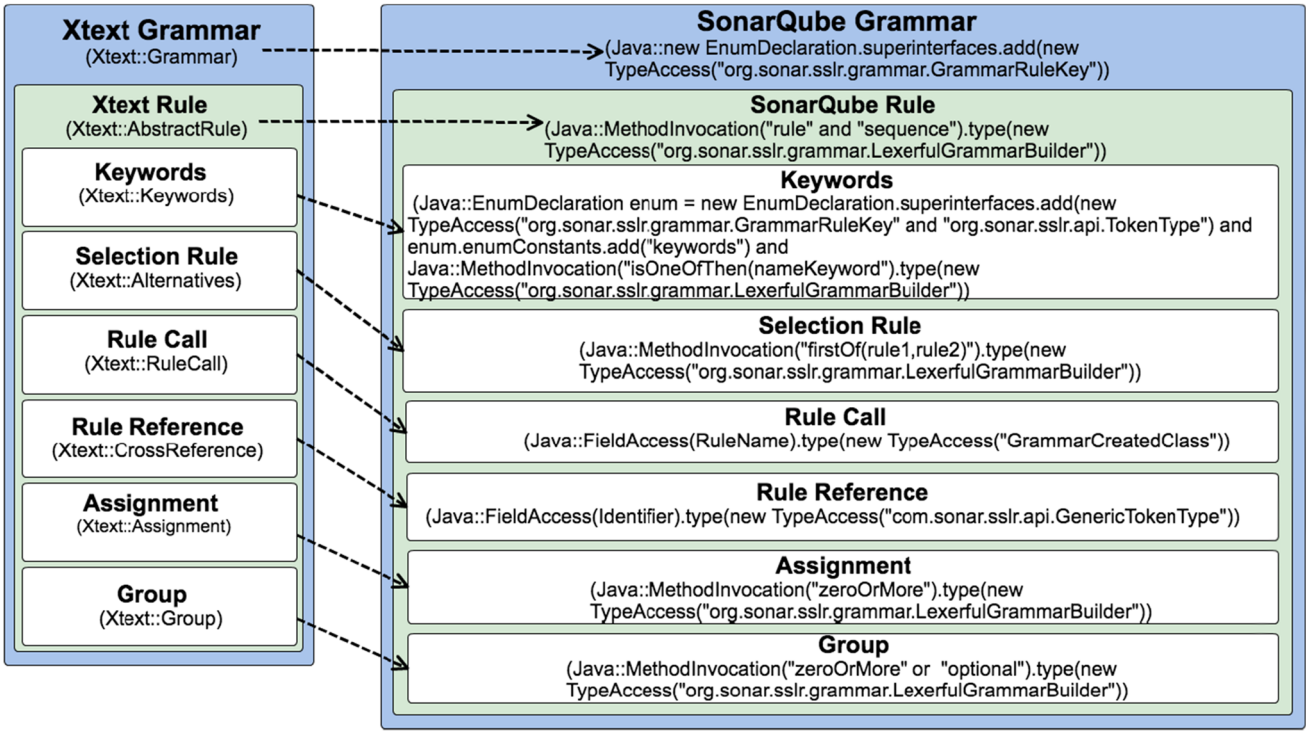

Fig. 2 Mapping between grammars elements

Java code required by SonarQube for parsing new languages is generated. Changes to the language definition are automatically propagated to the SonarQube parser, thus avoiding to have to manually adjust the Java parser. It prevents DSL designers from oversight mistakes or bad programming practices and is a noticeable saving of programming effort.

Besides, end users will be able to parse and analyse their text-based models, based on the new DSL, in the same manner that they can do it with other built-in languages. Meanwhile, the DSL users will be able to generate chart reports, define alerts, compute derived metrics and access the historical data, among other facilities.

Some basic metrics are provided by default in SonarQube for the new languages, such as the number of files and the number of lines of code; and some rules, such as line length checking. However, because those metrics cannot be directly transposed from one language to another, a further development phase is necessary to define the quality rules and the domain-specific metrics.

All of the information about the installation and configuration processes is available on the website accompanying the released plug-ins.

4 Use scenarios

This paper proposes a strategy to bridge the grammar formats between the most widespread platforms for designing text-based DSLs and analysing source code quality. A supporting tool is also provided as two Eclipse plug-ins. To ensure their validity, we applied one of the evaluation methods described by Hevner [17], which deploys a number of descriptive use scenarios for different knowledge fields. In this case, the use scenarios are related to the analysis of computer algorithms and the checking of sheet music. For each of these scenarios, source lines of code (SLOC) and cyclomatic complexity (CC) are measured to estimate the size and complexity savings of the model-driven approach. To ensure reproducibility, all of the artefacts are automatically generated to support the selected use scenarios that are available on the *Xtext2Sonar* website.

4.1 Analysing computer algorithms

The application of this case is suitable for both DSLs and GPLs, as long as they have been developed with Xtext. In this vein, the interoperability strategy was applied to Vary,[14] which is a computing environment for typing and running algorithms written in pseudocode notation. This tool (see Fig. 3) is aimed at learners of computer programming courses and computational scientists who need to easily write and run algorithms, while taking advantage of modern development environment features.

[14] http://tatyperson.github.io/Vary/.

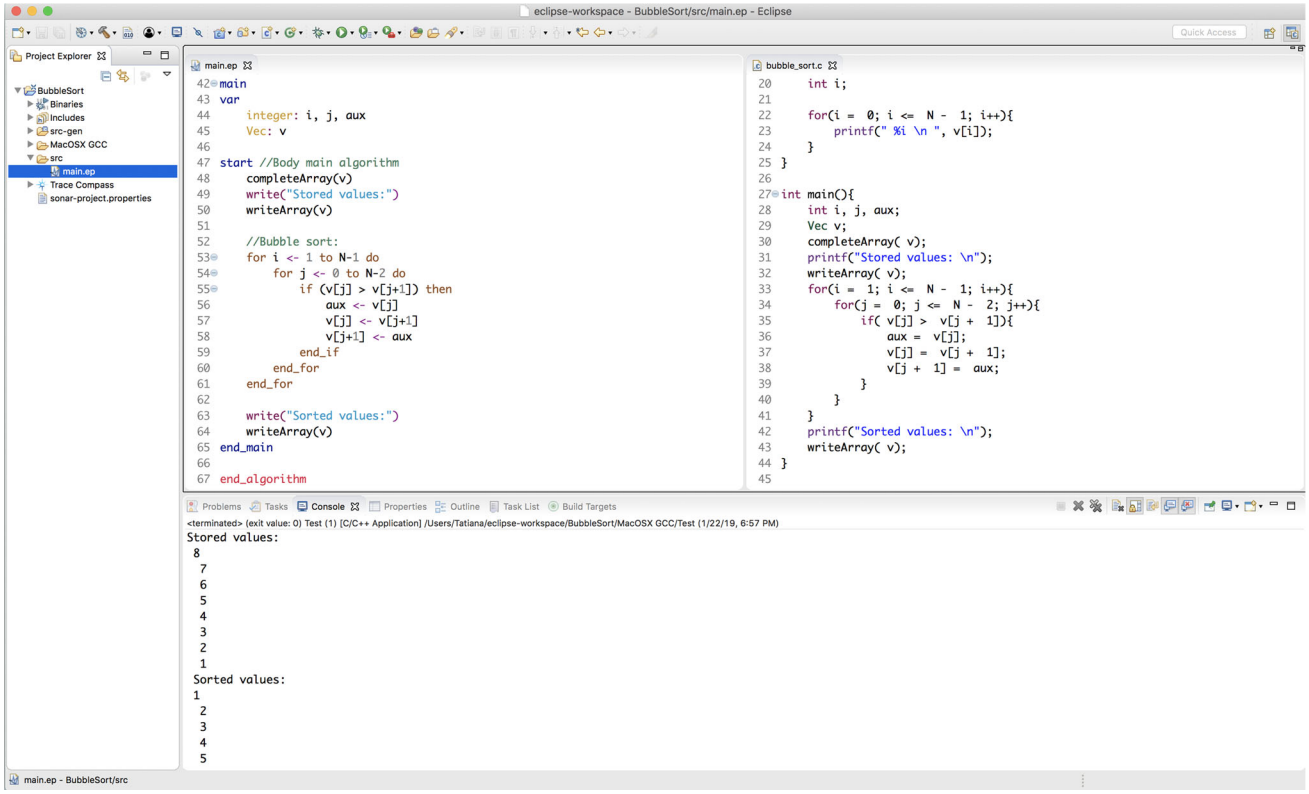

Fig. 3 Vary IDE

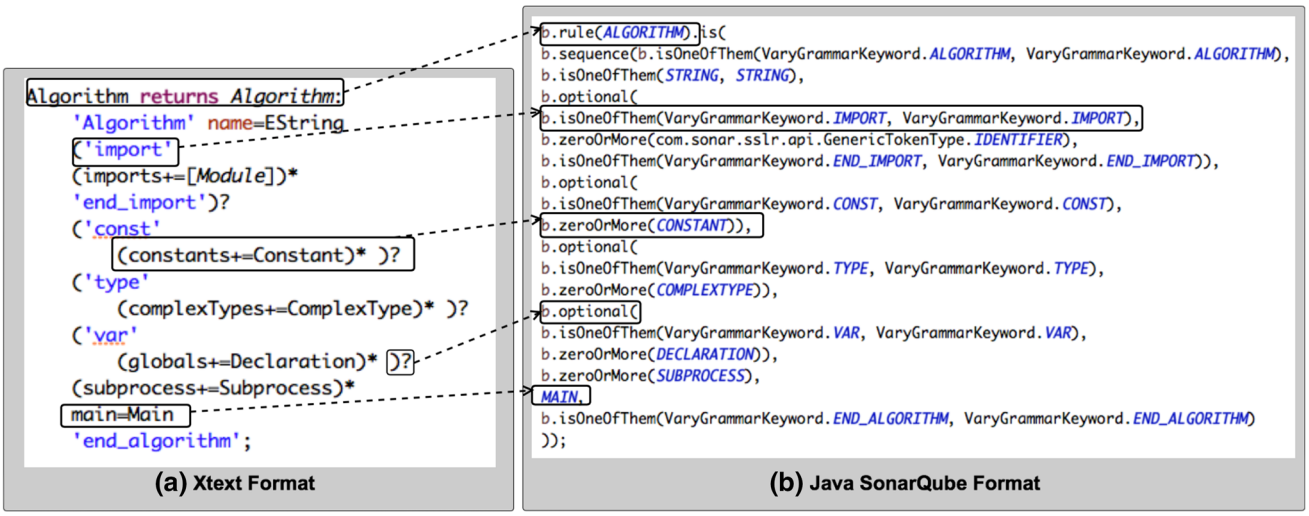

Fig. 4 Correspondence between grammar code elements of Vary DSL

Analysing algorithms in pseudocode notation is also possible with SonarQube. It provides users with the basic metrics (e.g. lines of code, percentage of commented code, etc.) and quality checks according to several guideline rules, such as the abuse of global variables, an excessive number of lines of code in a module, or whether the program subroutines are well documented, among others. A snippet of the Xtext grammar of the Vary language and its equivalent for SonarQube, along with their relationships, is shown in Fig. 4.

In this case, all of the artefacts required by Sonar for parsing the Vary language were initially developed from scratch. In fact, the motivation of this paper arose from the lessons that were learned during the development of Vary. *Xtext2Sonar* was subsequently applied for comparative purposes and to prove its feasibility.

Table 1 Source lines of code (SLOC) and cyclomatic complexity (CC) of the SonarQube artefacts required for Vary

Artefact	SLOC	CC
VaryKeyword.java	115	5
VaryPunctuator.java	64	3
VaryImpl.java	1076	2
VaryLexer.java	85	3
VaryToolkit.java	15	2
VaryParser.java	33	4

Table 2 Source lines of code (SLOC) and cyclomatic complexity (CC) of the SonarQube components required for LilyPond DSL

Artefact	SLOC	CC
LilyPondKeyword.java	92	5
LilyPondPunctuator.java	66	3
LilyPondImpl.java	763	2
LilyPondLexer.java	85	3
LilyPondToolkit.java	15	2
LilyPondParser.java	33	4

From a grammar file containing a total of 95 rules in 418 SLOC, *Xtext2Sonar* generated all of the Java components, mainly featuring a main class containing 942 SLOC. Table 1 presents the distribution of size and complexity of the generated code. In this case, *Xtext2Sonar* would have saved a programming effort of nearly 1300 lines of Java code.

4.2 Analysing sheet music

LilyPond[15] is a tool for musicians who want to produce sheet music [26]. This tool processes text input, which contains all information about the content of the scores and can be easily read by humans or by another program. LilyPond provides a dedicated DSL to type chord notes and combines them with melody and lyrics (see Fig. 5). Moreover, the DSL enables users to spend less time tweaking the output because LilyPond automatically generates the graphical output format and determines by itself the spaces, break lines and pages to obtain a proper layout.

Integrating the computer language supported by this tool into SonarQube might enables musicians to gain a deeper understanding of certain aspects of the musical pieces and their evolution over time. To this end, *Xtext2Sonar* was applied for automatically generating the artefacts required to analyse music compositions in SonarQube. Figure 6 shows a snippet of the Xtext grammar for the LilyPond DSL and its equivalent for SonarQube, along with their relationships.

From a grammar file containing a total of 82 rules in 156 SLOC, *Xtext2Sonar* is able to generate all the infrastructure required by SonarQube, mainly featuring a main Java class containing 676 SLOC. Table 2 presents the distribution of size and complexity of the code required to analyse the text-based models created with the DSL. As reflected in the table, this has involved a considerable saving of programming effort, nearly 1000 SLOC.

With the LilyPond plug-in for SonarQube, the users can analyse several quality aspects of the music sheets. The main goal is to encourage the users to follow best practice when creating sheet music. Following this aim, a set of metrics have been defined to ensure the readability of the musical score, such as checking the proper definition of the title and composer of the work. Additionally, other metrics have been added to deal with the number of lines by checking the correct use of loop structures instead of repeated individual statements. Figure 7 shows a screenshot of the SonarQube tool while analysing a given music sheet.

4.3 Analysing text-based languages of other domains

Besides the previous scenarios, the *Xtext2Sonar* approach has been successfully applied for the generation of SonarQube plug-ins to analyse DSLs of additional cases. A summary of such evaluation cases follows, which are also available on the website accompanying the tool.

- Sculptor[16] is an open source productivity tool to automatically generate Java applications by using a dedicated DSL to describe the domain layer in object-oriented systems.
- TANGO Controls[17] is a software toolkit for connecting things together and building control systems by using a specific language.
- Eclipse SmartHome[18] is a framework to build smart home and ambient-assisted living solutions. It relies on the use of DSLs for the different kinds of artefacts.

5 A case study on assessing learning of algorithm design

The above section presented the benefits of our approach from the DSL designer's or developer's perspective. This section focuses on the DSL end user by presenting a case study conducted to ensure the applicability of the metrics generated for the Vary language.

[15] http://lilypond.org/.

[16] http://sculptorgenerator.org.

[17] http://www.tango-controls.org.

[18] http://www.eclipse.org/smarthome.

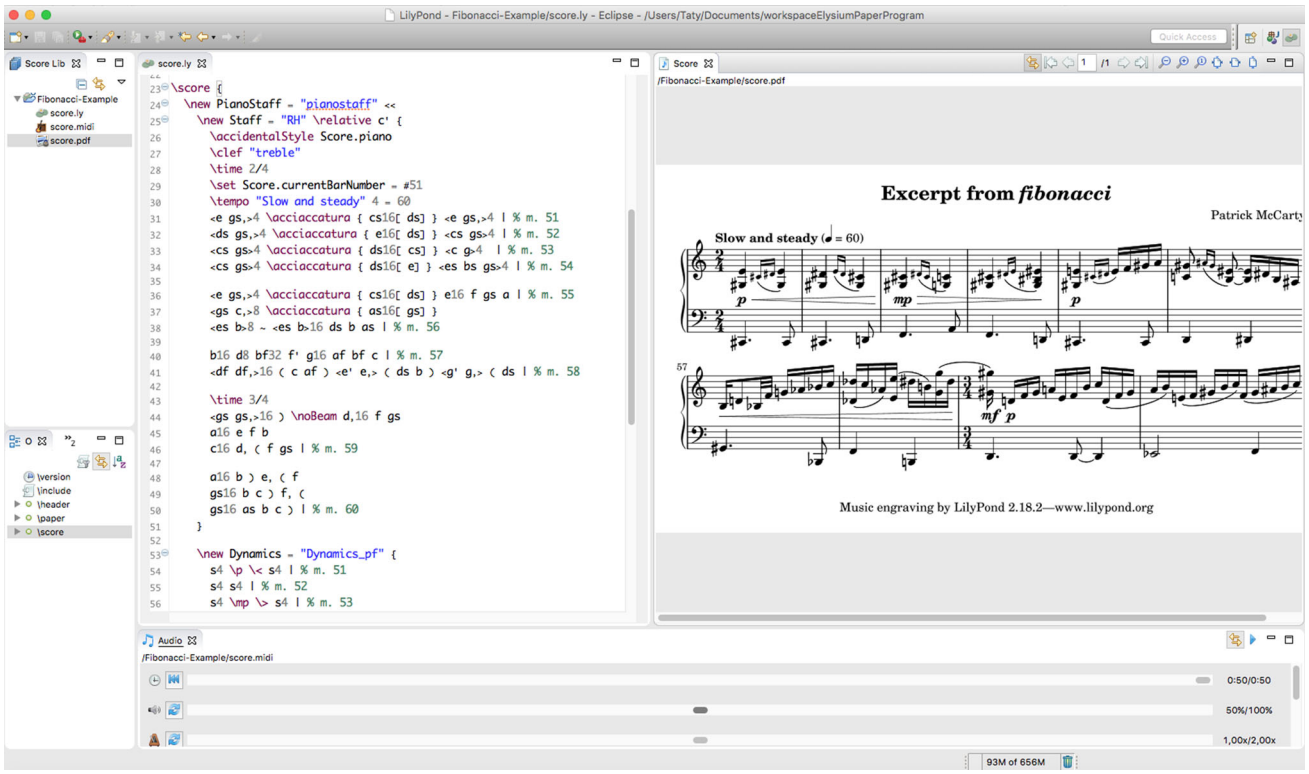

Fig. 5 LilyPond IDE

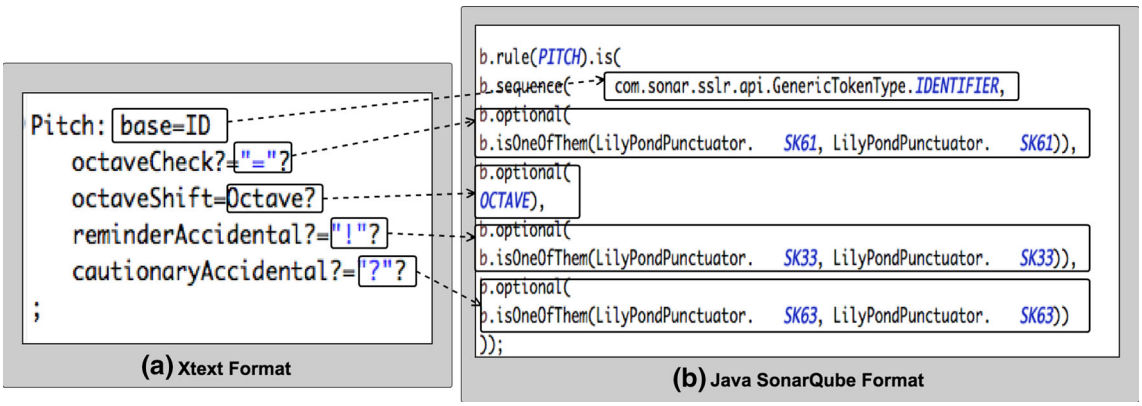

Fig. 6 Correspondences between grammar code elements of LilyPond DSL

5.1 Case study design

The main objective of this case study is to explore how the automatic computation of metrics for analysing the quality of the algorithms written with the Vary tool can help a teacher to assess their students' programming assignments. The study provides insights and generates ideas for new research.

The study was conducted in the context of the subject *Introduction to Programming* in the Degree in Computer Engineering at the University of Cadiz. In this case, a group of students were asked to develop an algorithm in pseudocode to solve a given problem. The problem consists in summing the odd numbers between 1 and a number provided by the user.

Concerning the frame of reference, this case study is related to the sustainable assessment issue in learning environments. According to Boud [6], sustainable assessment aims at the students' development of life-long learning evaluation skills and gives the students confidence to progress in learning throughout life, without increasing the workload of the academic staff [9].

The research questions are as follows: (i) Can SonarQube metrics provide support for the automatic assessment of assignments consisting in writing algorithms? And (ii)

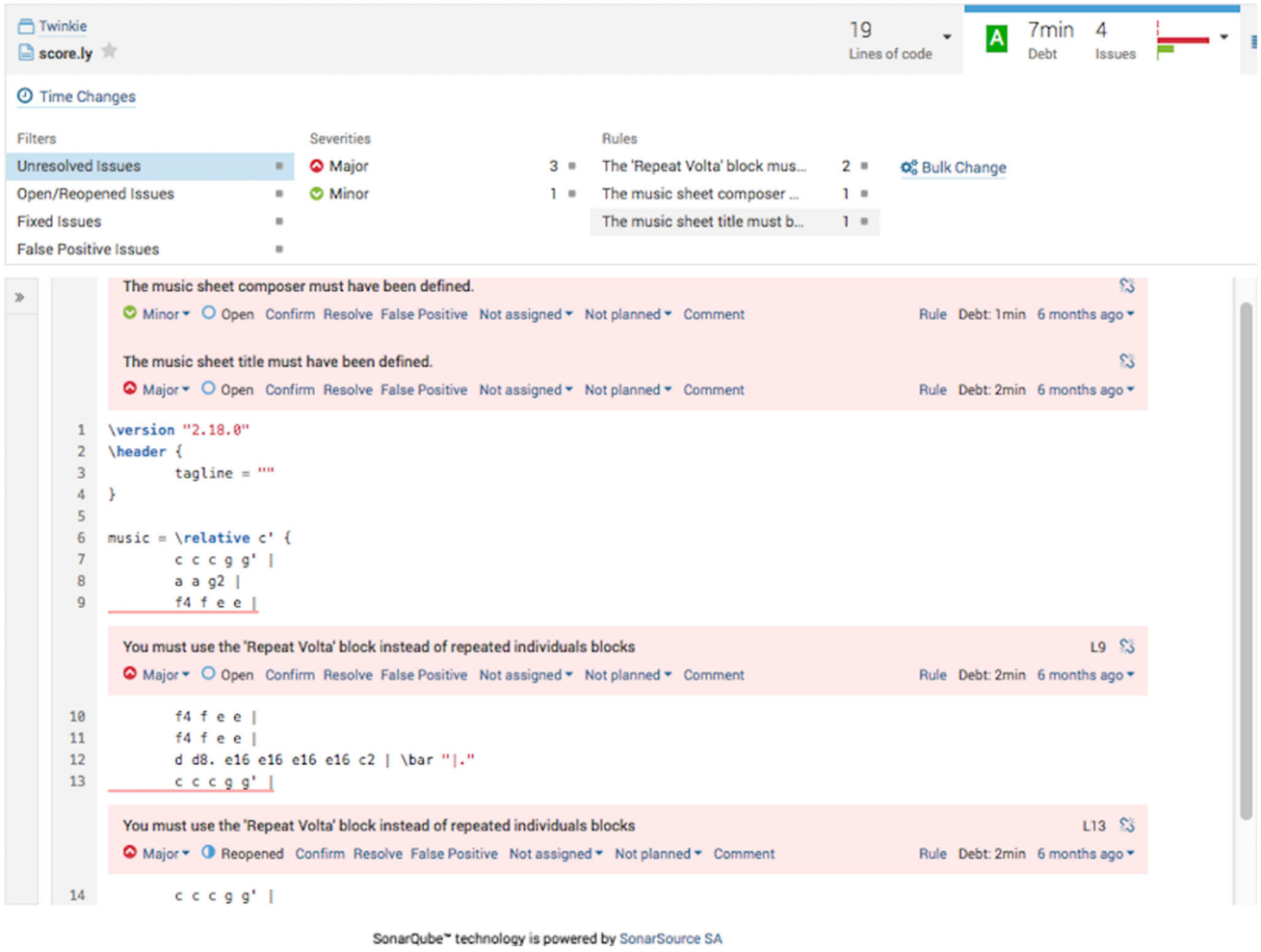

Fig. 7 SonarQube showing non-conformities of a music sheet

are there any correlations between the students' grades and the automatic measures of the students' code? The general hypothesis claims that assessing students performance during the learning of algorithm design is possible by applying static analysis techniques to check quality metrics in pseudocode files.

The data collection was performed without interacting with the subjects during the data collection (i.e. an indirect method). Every mark indicated by the teacher for each student assignment was stored in a datasheet document. Furthermore, all of the files submitted to the learning management store, namely Moodle, for teacher review were subsequently analysed with SonarQube and then stored in the same datasheet.

5.2 Data collection

The marks that were manually assigned by the teacher are broken down by the attributes (weighted) that follow.

Table 3 Maximum thresholds and weights expected for the maintainability metrics

Metric	Maximum threshold	Weight (%)
Cyclomatic complexity (CC)	3	5
Source lines of code (SLOC)	18	5
Percentage of duplicated code (DC)	20	5
Number of quality rules violated (QR)	1	20

- *Correctness*: The algorithm is well programmed and can be properly compiled (30%).
- *Validity*: The results of the algorithm are expected (35%).
- *Maintainability*: The algorithm code is easy to read and maintain (35%).

With regard to SonarQube, in addition to the common code metrics such as CC, SLOC and the percentage of duplicated code (DC), a set of quality rules (QR) were developed to analyse the quality of the algorithms:

- *Algorithm with no documentation.*
- *Variable name too long.*
- *Variable name is too short and maybe is meaningless.*
- *Use of a input sentence (read) without using an output sentence (write).*
- *Too many global variables.*
- *Code lines that are too long.*
- *Procedure or function with no documentation.*
- *Lines with bad indentation.*

Furthermore, an interview with the lecturer of this subject was conducted to define a mapping between her assessment criteria, the metrics and SonarQube's quality rules. Because correctness and validity are out of scope of the SonarQube capabilities, maintainability was the attribute that was assessed by computing a combination of the CC, SLOC, DC and QR values. In this case, the assessment of these metrics was computed by using some linear adjustment functions that provided the mark according to the maximum thresholds (Table 3) defined by the teacher for this assignment. Additionally, a set of weights were set for each metric.

5.3 Analysis and reporting of collected data

A total of 31 student's assignments were manually reviewed by the teacher and later automatically checked with SonarQube. Some analyses were performed with quantitative methods, mainly through analysis of correlation and descriptive statistics, such as scatter plots. To mention some examples, the average mark for maintainability attribute is 2.02 out of 3.5 with a standard deviation of 1.04, whereas the one measured by SonarQube with the weights and thresholds above is 1.91 out of 3.5 with a standard deviation of 0.55. The collected data also reveal a low/medium correlation ($r = 0.362$, $p = 0.045$) between the maintainability degree estimated by the teacher and the value computed with support of the SonarQube metrics and its quality rules. Meanwhile, the correlation between the final mark assigned by the teacher and the final mark obtained with the support of SonarQube was significant ($r = 0.9673$; $p < 0.001$), as shown in Fig. 8.

The first research question has validated how SonarQube can be extended with custom metrics for assessing quality features of algorithms, thus providing the basis for the automatic assessment. This procedure may be applied to the assessment of a large number of students, achieving a significant saving in the effort of reviewing.

With regard to the second question, the fact that the correlation level is not very high does not mean that this approach is not feasible. In contrast, it shows that formulas should be continuously improved on a regular basis. In addition, having some cases with high marks of maintainability measured by SonarQube but with not so high marks assigned by the teacher may be a warning that the tool is measuring other criteria that were not actually taken into account during the manual reviewing, or vice versa. It also raises another question: Are the weight and threshold values used in the computations according with the teachers' mental conceptualisation during the (perhaps less precise) assessment?

In this case, SonarQube's capabilities are used to systematise and automate part of the grading process. For example, the amount of duplicated code that can be tolerated may vary among different teachers or even, for the same teacher, among different kinds of problems. The perception of the teacher with regard to each metric or quality check affects the universalization of the assessment tools, but at least assures the accurate application of the same criteria for all the students. Furthermore, in a recent study [8] about the mistakes that stu-

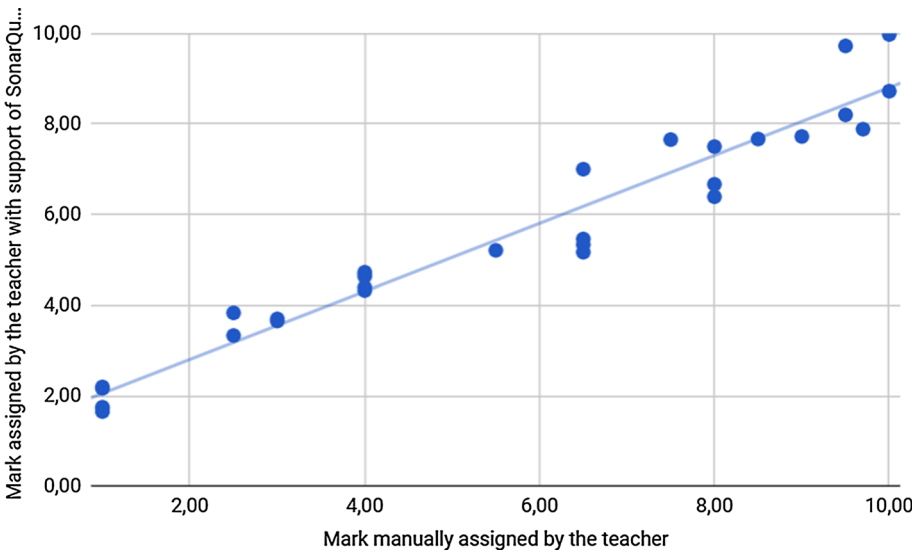

Fig. 8 Correlation between teacher-assigned marks and SonarQube-computed ones

dents make while learning to program Java and whether the educators could make an accurate estimate of which mistakes were most common, the authors found that the educators' estimates do not agree with one another or with the students' data.

This case study has been presented how the metrics computed by SonarQube can be used to automatically grade students. Furthermore, the SonarQube visualisation provides teachers with additional, useful feedback, such as what mistakes students make more frequently, the students' task that do not fulfil certain quality rules, and so on. Because of the web nature of the quality platform, the students will be able to easily check their own mistakes on their assignments.

6 A usability study of static code analysis in music sheet composition

This section also focuses on the DSL end users by developing a usability test. This test is aimed at checking the applicability of SonarQube metrics to assess the quality of music sheets using the Lilypond DSL.

6.1 Usability study design

ISO 9241 [1] defines usability as the extent to which a product can be used by specified users to achieve specified goals effectively, efficiently and satisfactorily in a specified context of use. The usability test herein aims at finding out if automatic metrics computation for analysing the quality of Lilypond music sheets is suitable for musicians as the DSL end users. The test has been defined and executed by following the guidelines provided by Rubin et al. [32].

To obtain significant results, the study was conducted on a set of experts in the music domain, from musicologists to performers. All of the experts were previously screened to ensure they are accustomed to computer-aided sheet music design tools, particularly Lilypond.

The following scenario was prescribed. First, the respondents had to directly observe a given sheet music[19] in PDF format and they were asked to find issues (i.e. errors and bad practice) on the sheet. Afterwards, the experts had to access the SonarQube instance,[20] open the file "score.ly" containing the source of the previous music sheet, and then visualise the list of evidence automatically found in the sheet. Finally, the musicians were asked whether the syntactic error or bad practice warnings issued by the tool corresponded to those observed with the naked eye.

A series of quality rules for the music domain were integrated in SonarQube, including among others:

- There are no consecutive silent beats.
- When a note is held in the previous beat, a precautionary alteration must be included.
- Check that the tempo has been defined at the beginning.
- Check that the time change has been made correctly.
- The music sheet title must be defined.
- The music sheet composer must be defined.
- A page step cannot exist in a repeat volta instruction.
- Lines of code should not be too long.

These rules are based on a set of best practices related to style, musical logic, interpretation indications and bibliographic issues for assessing music sheets [13]. All such errors and bad practices were detected only once on the provided sheet file, except for "there are no consecutive silent beats," which was found four times.

The musicians involved in this study were initially introduced to the overall objectives of the source code quality platform, such as SonarQube, and its application to analyse music sheets (as this research states). Once the experts had visually observed the sheet music generated from LilyPond and after checking the evidence that was automatically returned by SonarQube, a survey was conducted.

6.2 Data compilation

Two questionnaires were designed for the survey with Google Forms. Pretest questionnaires were used to determine the initial state of users' opinions and knowledge, before doing the observations.[21] Posttest questionnaires were used to compile the users' insights after the observations.[22] Alongside these questionnaires, a consent and revocation form was created to guarantee the privacy of the personal identification data that were compiled.

The pretest form was used to collect general data, such as professional activity or academic degree, along with more specific information, such as knowledge of digital music notation, use of software for composing music sheets and the procedures or techniques used to detect and correct the mistakes, whereas in the posttest form, a series of questions were included to discover whether the participants found the errors or bad practice in the music sheet provided, whether they were able to visualise and understand the quality evidence shown in SonarQube, whether this evidence corresponded with the manually observed issues and their opinion about including this type of automatic tool on a regular basis.

[19] https://goo.gl/ju1zwd.

[20] https://goo.gl/xsPzgY.

[21] https://goo.gl/17pGq9.

[22] https://goo.gl/dkZB3j.

Table 4 Usability evaluation results

Dimension	Learnability (%)	Efficiency (%)	Satisfaction (%)	Utility (%)
Professional activity of the participants				
Teacher	85.7	71.4	83.33	7.6
Researcher	85.7	57.1	100	7.1
Musicologist	75.0	75.0	100	8.0
Student	100	50.0	100	8.0
Average	85.0	65.0	95.0	7.6
Chi-squared	0.880	0.875	0.582	0.607
Experience with music sheet composition software				
Non-experienced	50.0	50.00	75.0	6.5
Experienced	93.8	68.8	100.00	7.8
Average	85.0	65.0	95.0	7.6
Chi-squared	0.028	0.482	0.040	0.396
Highest academic degree achieved by the participants				
Graduate	77.8	55.6	88.9	7.7
Postgraduate	100.00	85.7	100	8.4
Doctorate	75.0	50.0	100.00	5.8
Average	85.0	65.0	95.0	7.6
Chi-squared	0.383	0.355	0.526	0.652

6.3 Data analysis and findings

To analyse the collected data, they had to be standardised to properly categorise the specific responses of each expert. The data were then analysed according to different dimensions:

- *The whole sample*: A total of 20 musicians participated in the study.
- *Experience with music sheet composition software*: Most of the participants (16) were experienced with this kind of tool, whereas very few (4) did not have previous experience.
- *Main professional activity of the participants*: researchers (7), teachers (7), musicologists (4) and postgraduate music students (2).
- *Highest academic degree achieved by the participants*: graduates (9), postgraduates (7) and doctorates (4).

Usability attributes such as learnability, efficiency and satisfaction and the perceived utility have been used in this study. Table 4 shows the obtained results. The main insights for each of the questions posed in the posttest are presented below.

- *Learnability: Have you been able to access to SonarQube and visualise the errors and bad practices issued by the tool for the sheet music made with LilyPond?* A total of 93.8% of the participants gave us a positive response, decreasing until 50% for users without previous experience with music production software. The results showed a statistically significant difference (0.028).
- *Efficiency: Do you think that the errors and warnings issued by SonarQube correspond to the ones you found when you observed the sheet music in PDF?* More than 70% of the teachers and the musicologists answered affirmatively. However, this value drops to 57.14% and 50.00% in the case of the researchers and the students.
- *Satisfaction: Would you consider the inclusion of this type of tool to analyse the quality of sheet music?* Most of the participants (100%) deemed that the use of this kind of tool was interesting. Predictably, this value falls to 75% for non-experienced users. The results showed a statistically significant difference (0.040).
- *Utility: How useful would you consider this tool?* The results obtained in this rating have been quite promising, revealing a score of 7.6 out of 10 with a standard deviation of 2.08. Remarkably, only 50% of the musicologists considered that the errors provided by SonarQube correspond to those they found when directly observing the sheet music. However, they rated the tool with 8 out 10, which gives us an insight into the real potential that this approach may offer.

The analysis possibilities of the SonarQube platform are not merely limited to the completeness of the bibliographic metadata that describes the score. In addition to the metrics and rules we included in the tool, the experts identified some bad practices on the scores. Some of the found issues were (i) the inclusion of measures group which must be played twice, paced between page breaks, hindering reading during interpretation; (ii) the inclusion of alterations that systemat-

ically affect a group of altered notes throughout the text not positioned as part of the key signature and (iii) the absence of cautionary alterations that allow noticing the cessation of the effect of sharps or flats past their zone of influence. All of these rules could be easily integrated in the tool by writing XPath rules directly through the web interface.

This study has shown how the computation of metrics for analysing the quality of music sheets can be automated by means of source code quality platforms. In light of the obtained results, additional applications are possible. Using SonarQube, for instance, musicians can further check the quality of their compositions, music teachers can assess students, researchers can analyse the differences and commonalities in the mistakes or bad practices made during the music writing process, and so on.

7 Discussion and conclusions

The use of DSLs is increasing among domain experts in different sectors. In this context, frameworks for developing DSLs are able to reduce development times. Although DSL toolkits such as Xtext usually provide a low-level API to implement validation rules within simple scripts, they do not include any kind of support for generating reports and alerts, computing derived metrics, drawing charts, presenting historical data and so on. These features are already provided by code quality continuous inspection platforms, such as SonarQube. However, code quality platforms do not enable their users to automate code inspections for their specific domain languages. Consequently, our proposal intends to reuse that software infrastructure by providing a tool that automatically builds language recognisers for the widespread SonarQube source quality platform.

The *Xtext2Sonar* tool was developed by following a model-driven interoperability strategy to transform Xtext grammar files into Java plug-ins for the SonarQube code quality platform. Nevertheless, this approach does not completely automate the entire process because practitioners (i.e. the users of the DSLs) are required to propose the specific metrics and rules according to their areas of expertise to subsequently integrate them in SonarQube. Such metrics and rules must be written using Java via a SonarQube plugin or adding XPath rules directly through the SonarQube web interface.

Our proposal uses Xtext-based languages as main input, so that we have the same advantages and drawbacks of Xtext regarding the grammars expressivity. All the valid LL* grammars developed with Xtext are suitable to be transformed into SonarQube grammars. However, since Xtext uses the ANTLR parser, which is based on a LL algorithm, left recursive grammars are not allowed. As a consequence, a left-factoring of the grammars are always required before using Xtext and our tool.

A demo instance of SonarQube is provided[23] to analyse artefacts generated with some example domain languages, which are aimed to design algorithms, compose music sheets, connect smart home systems and IoT gadgets and so on.

The solution described in this paper provides some benefits. From the DSL end user perspective, it encourages domain experts to use the whole infrastructure and all of the analytical capabilities provided by the quality tool, namely SonarQube, to check their own text-based artefacts developed with their DSLs. This may also contribute to the successful adoption of DSLs. This benefit is shown both in a case study and a usability test, which explored how the automatic computation of SonarQube metrics for certain languages can provide support for computer programming teachers and musician experts. In the former, some of the metrics provided by SonarQube were used to support teachers during the students' grading processes, whereas in the latter, several usability attributes of the tool, such as learnability, satisfaction and utility for analysing quality of music sheets, were assessed by musician experts.

From the DSL developer's or maintainer's perspective, the model-driven automation drastically reduces the effort required to develop the components to support the new DSLs in SonarQube. This significant reduction of the effort invested was illustrated by measuring the number of lines of code automatically generated and their complexity. However, the measurement of the saved implementation effort could be questioned because of the model-based transformation and the likely generation of boilerplate code. Nevertheless, this threat was tackled minimising the source code generated by reusing as much code as possible and applying design principles.

Additionally, it is necessary to analyse threats according to the construct validity, internal validity, external validity and reliability. To maximise the internal and construct ones, we maintained a detailed protocol both for the case study and the usability test. They were also reviewed by peer researchers and performed a thorough process of discussion and analysis of the metrics. On the one hand, both the teachers and researchers agreed on a shared definition of maintainability according to the ISO 25000 standard and looked for the metrics that better fit to teacher's expectations during the assessment activity. On the other hand, a researcher on critical edition of musical texts provided a set of common best practices for assessing music sheets.

With the aim of assuring the reliability of the study, the datasheet documents that were used for the analysis are available on the *Xtext2Sonar* website, along with a SonarQube running instance configured with the metrics developed to

[23] http://vedilsanalytics.uca.es/sonarqube/.

check the algorithms and the music sheets. Nonetheless, it is not possible to generalise these findings due to the limited size of the samples and the fact that metrics are DSL-specific. Hence, further experimentation and analysis with broader samples are required to evaluate to what extent the findings are of relevance for other cases.

The developed software (the set of Eclipse plug-ins) is necessarily coupled to both Xtext and Sonar by implementation reasons. However, because of its model-driven approach, the method could be easily transferred to other frameworks, as long as the DSL parser was based on EBNF grammars and the quality tool on EBNF or PEGs. However, the latter cannot be completely affirmed for quality platforms due to the lack of alternative open source platforms with a similar purpose. In our future work, we plan to provide SonarQube-like quality platforms with facilities to analyse visual domain-specific languages, such as Blockly [16].

Acknowledgements This work has been developed in the VISAIGLE project, funded by the Spanish National Research Agency (AEI) with ERDF funds under grant ref. TIN2017-85797-R.

References

1. Abran, A., Khelifi, A., Suryn, W., Seffah, A.: Usability meanings and interpretations in iso standards. Softw. Qual. J. **11**(4), 325–338 (2003)
2. Ameller, D., Franch, X.J.: Dealing with non-functional requirements in model-driven development. In: 2010 18th IEEE international requirements engineering conference, pp. 189–198 (2010)
3. Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P.: The financial aspect of managing technical debt: a systematic literature review. Inf. Softw. Technol. **64**, 52–73 (2015)
4. Besova, G., Steenken, D., Wehrheim, H.: Grammar-based model transformations: definition, execution, and quality properties. Comput. Lang. Syst. Struct. **43**, 116–138 (2015)
5. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing Ltd, Birmingham (2013)
6. Boud, D.: Sustainable assessment: rethinking assessment for the learning society. Stud. Contin. Educ. **22**(2), 151–167 (2000)
7. Brambilla, M., Cabot, J., Wimmer, M.: Model-driven software engineering in practice. Synth. Lect. Softw. Eng. **1**(1), 1–182 (2012)
8. Brown, N.C.C., Altadmri, A.: Novice java programming mistakes: large-scale data vs. educator beliefs. ACM Trans. Comput. Educ. (TOCE) **17**(2), 7:1–7:21 (2017). https://doi.org/10.1145/2994154
9. Davies, S.: Effective assessment in a digital age. URL: http://www.jisc.ac.uk/media/documents/programmes/elearning/digiassass_eada.pdf (2010). Accessed 15 Oct 2013
10. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '04, pp. 111–122. ACM, New York, NY, USA (2004). https://doi.org/10.1145/964001.964011
11. Fowler, M.: Domain-Specific Languages. Pearson Education, London (2010)
12. Gamma, E.: Design Patterns: Elements of Reusable Object-Oriented Software. Pearson Education India, Bangalore (1995)
13. Gould, E.: Behind Bars: The Definitive Guide to Music Notation. Faber Music, London (2011)
14. Heinze, T.S., Amme, W., Moser, S.: Static analysis and process model transformation for an advanced business process to petri net mapping. Softw. Pract. Exp. **48**(1), 161–195 (2018)
15. Hermans, F., Pinzger, M., Deursen, A.: Domain-specific languages in practice: a user study on the success factors. In: Model Driven Engineering Languages and Systems: 12th International Conference, pp. 423–437. Springer, Berlin (2009)
16. Hermans, F., Stolee, K.T., Hoepelman, D.: Smells in block-based programming languages. In: 2016 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2016, Cambridge, United Kingdom, September 4–8, 2016, pp. 68–72 (2016)
17. Hevner, A., Chatterjee, S.: Design Research in Information Systems: Theory and Practice, vol. 22. Springer, Berlin (2010)
18. ISO/IEC: 14977: Information technology—Syntactic metalanguage—Extended BNF. Standard, International Organization for Standardization (1996)
19. ISO/IEC: 25010: Systems and software engineering—systems and software quality requirements and evaluation (SQuaRE)—System and software quality models. Tech. rep., International Organization for Standardization (2010)
20. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Sci. Comput. Program. **72**(1), 31–39 (2008)
21. Kappel, G., Langer, P., Retschitzegger, W., Schwinger, W., Wimmer, M.: Model transformation by-example: a survey of the first wave. In: Düsterhöft, A., Klettke, M., Schewe, K.-D. (eds.) Conceptual Modelling and Its Theoretical Foundations, pp. 197–215. Springer (2012)
22. Kruchten, P., Nord, R.L., Ozkaya, I.: Technical debt: from metaphor to theory and practice. IEEE Softw. **29**(6), 18–21 (2012)
23. Letouzey, J.L.: The SQALE method for evaluating technical debt. In: Managing Technical Debt (MTD), 2012 Third International Workshop on, pp. 31–36. IEEE (2012)
24. Mandal, A., Mohan, D., Jetley, R., Nair, S., D'Souza, M.: A generic static analysis framework for domain-specific languages. In: 2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA), vol. 1, pp. 27–34 (2018). https://doi.org/10.1109/ETFA.2018.8502576
25. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. **37**(4), 316–344 (2005)
26. Nienhuys, H.W., Nieuwenhuizen, J.: Lilypond, a system for automated music engraving. In: Proceedings of the XIV Colloquium on Musical Informatics (XIV CIM 2003), vol. 1, pp. 167–172 (2003)
27. Parr, T., Fisher, K.: LL (*): the foundation of the ANTLR parser generator. In: ACM SIGPLAN Notices, vol. 46, pp. 425–436. ACM (2011)
28. Prähofer, H., Angerer, F., Ramler, R., Lacheiner, H., Grillenberger, F.: Opportunities and challenges of static code analysis of iec 61131-3 programs. In: Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies Factory Automation (ETFA 2012), pp. 1–8 (2012). https://doi.org/10.1109/ETFA.2012.6489535
29. Redziejowski, R.R.: From ebnf to peg. Fundam. Inform. **128**(1–2), 177–191 (2013)
30. Rochimah, S., Arifiani, S., Insanittaqwa, V.F.: Non-source code refactoring: a systematic literature review. Int. J. Softw. Eng. Appl. **9**(6), 197–214 (2015)
31. Rose, K.: Project Quality Management: Why, What and How, 2nd edn. J. Ross Publishing, USA (2005)
32. Rubin, J., Chisnell, D.: Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests. Wiley, Hoboken (2008)
33. Ruiz-Rube, I., Person, T., Dodero, J.M.: Static analysis of textual models. In: Jornadas de Ingeniería del Software y Bases de Datos (JISBD) (2016)

34. Saad, C., Bauer, B.: Data-flow based model analysis and its applications. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.) Model-Driven Engineering Languages and Systems, pp. 707–723. Springer, Berlin, Heidelberg (2013)
35. Shambaugh, R., Weiss, A., Guha, A.: Rehearsal: a configuration verification tool for puppet. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '16, pp. 416–430. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2908080.2908083
36. Sharma, T., Fragkoulis, M., Spinellis, D.: Does your configuration code smell? In: Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16, pp. 189–200. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2901739.2901761
37. Stevenson, A., Cordy, J.R.: A survey of grammatical inference in software engineering. Sci. Comput. Program. **96**, 444–459 (2014)
38. Syriani, E., Gray, J.: Challenges for addressing quality factors in model transformation. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, pp. 929–937 (2012)
39. Tomas, P., Escalona, M., Mejias, M.: Open source tools for measuring the internal quality of java software products. A survey. Comput. Stand. Interfaces **36**(1), 244–255 (2013)

Iván Ruiz-Rube is an assistant professor at University of Cadiz, Spain. He received his Master degree in Software Engineering and Technology from the University of Seville and his PhD from the University of Cadiz. His fields of research are technology-enhanced learning and software process improvement. He has published several papers in these fields. Previously, he has worked as a software engineer for consulting companies such as Everis Spain S.L. and Sadiel S.A.

Tatiana Person has a Master degree in Software Engineering from the University of Cádiz. She has worked as a researcher in one R&D project with national funding. She is currently working as an assistant lecturer in the Department of Computer Engineering of the University of Cádiz. She is doing her PhD on how to bring data analytics techniques and mobile software development closer to noncoders.

Juan Manuel Dodero received his Master degree in Computer Science from the Polytechnic University of Madrid and his PhD from the Carlos III University of Madrid. He is currently an associate professor of the University of Cadiz (Spain). He has also been an associate lecturer of the Carlos III University of Madrid and worked as a R&D engineer for Intelligent Software Components S.A. His main research interests are web science and engineering and technology-enhanced learning, fields in which he has co-authored numerous contributions in journals and research conferences.

José Miguel Mota received the Master degree in computer science with the Universitat Oberta de Catalunya, Spain. He is currently an Associate Lecturer with the University of Cádiz, Spain. He is also a Ph.D. Candidate Researcher in technology-enhanced learning. His current research interests include mobile learning, augmented reality and learning analytics. He has published several papers in these fields.

Javier Merchán Sánchez-Jara has a Library Science and Information Science degree, and a Master degree in Textual Heritage and Digital Humanities by the University of Salamanca (USAL), and a Ph.D in Information in the Knowledge Society. He is a member of E-Lectra (Reading, Digital Editing, Transfer and Evaluation of Scientific Information) research group, member of the IEMyRHD (Institute for Medieval and Renaissance Studies and Digital Humanities) and the (Observatory of Bibliometrics and Scientific Research) OBIC at the University of Salamanca. Currently, he is researching on markup standards for music information.