Applying static code analysis for domain-specific languages

Iván Ruiz-Rube, Tatiana Person, Juan Manuel Dodero, José Miguel Mota

Department of Computer Engineering
University of Cádiz
Spain

Javier Merchán Sánchez-Jara

E-LECTRA Research Group
University of Salamanca
Spain

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. Consequently, methods and tools for analysing source code quality can also be applied to software artifacts 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.

1 Introduction

Quality management comprises a set of activities to plan, control, assure and improve the quality of the organisations, products or services [25]. Increasing the quality of software artifacts 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 [20]), static analysis tools are used to extract indicators of quality attributes from software artifacts, 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 non-functional, quality attributes, such as security, compatibility, portability, efficiency, maintainability, reliability and usability [16].

In contrast to GPL, Domain-Specific Languages (DSL) are computer languages that are meant for experts in a number of diverse domains [10]. Professionals in many different areas can develop and describe problem solutions that take the shape of source code artifacts by means of a DSL that is especially defined for their own field and discipline.

However, can the tools for analysing source code quality be applied to software artifacts that are developed at the DSL code level? Refactoring techniques can be applied to UML models, requirement specifications, and software architecture descriptions [24], among other artifacts. Thus, facilities to evaluate the quality of the DSL source code artifacts by applying well-known static analysis techniques might be an additional factor that contributes to a successful adoption of DSLs [12].

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 the 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 research context and issues. Section 3 describes the proposed solution to bridge the grammar formats to analyse text-based DSLs from within the code quality platforms. In Section 4, some descriptive use scenarios of different text-based languages are presented to evaluate the solution. Section 5 presents the design and result of a case study conducted for a language aimed at designing algorithms. A usability test related with a language for designing music sheets is also included in Section 6. Finally, Section 7 discusses the results and draws the conclusions of this research.

2 Research context and related work

Two main options arise when developing a DSL, namely language exploitation and language invention [21]. On the one hand, an existing GPL, in combination with an application library, can be a useful tool to describe domain-specific issues. This is the language exploitation approach taken by source code quality platforms, such as SonarQube, which provides a Java extension library as a piggyback DSL to recognise additional source languages to be analysed. On the other hand, domain-specific notations are DSL inventions that are designed with no manifest commonalities with existing languages.

DSLs can be classified from several perspectives, mainly according to their concrete syntax (e.g., text-based or visual) and their way of implementation, such as internal (embedded into a host general purpose language) or external (independent of any other language). 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, transformation and is used to conduct the software process life-cycle [7].

2.1 Issues with text-based DSL frameworks

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 5,000 grammar files based on Xtext[1] in Github[2]. JetBrains MPS[3], MontiCore Language Workbench[4], Rascal Metaprogramming Language[5] and the Spoofax Language Workbench[6] 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 probably 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) [15] 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 [23], 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.

[1] http://www.eclipse.org/Xtext
[2] https://github.com
[3] https://www.jetbrains.com/mps
[4] http://www.monticore.de
[5] http://www.rascal-mpl.org
[6] http://www.metaborg.org

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 [28].

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

2.2 Issues with source code quality parsers

The use of interactive quality platforms for analysing source code, such as SonarQube[7], Codacy[8] or SQuORE[9], is increasingly gaining attention in the developer community [30]. SonarQube is particularly prevalent, partly due to its flexibility to parse several GPL (currently, more than 20) 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 technical debt analogy [19] in the software engineering field implies that doing things quickly and dirtily may cause additional costs and effort in the future to fix and redesign software projects. This

[7]https://www.sonarqube.org
[8]https://www.codacy.com
[9]http://www.squoring.com

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));
    }
}
```

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. The issue is that quality platforms, such as SonarQube, are, in principle, intended to check programs written in a general-purpose programming language.

SonarQube includes an extension 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 [21]; 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 Java library that provides everything required to create lexers and parsers for analysing a piece of source code.

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.

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

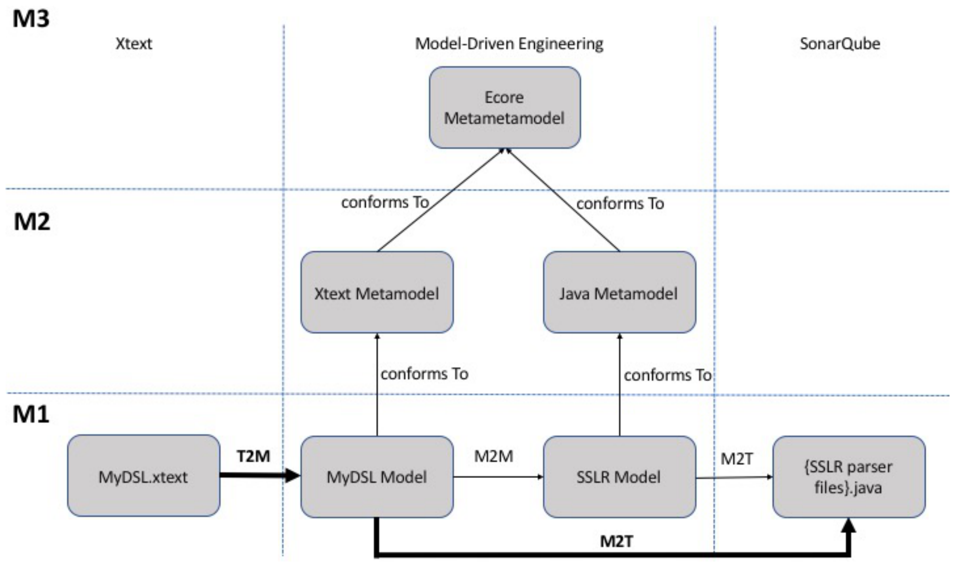


Figure 1: Model-Driven Interoperability strategy (based on the global schema defined in [7])

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.

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 DLS'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. Finally, 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 [18], 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. 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 Figure 1). This prevents the need to use a model transformation engine, such as ATL [17], 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 [29, 2].

From the user's perspective, 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

[11]http://www.eclipse.org/acceleo
[12]https://github.com/TatyPerson/Xtext2Sonar

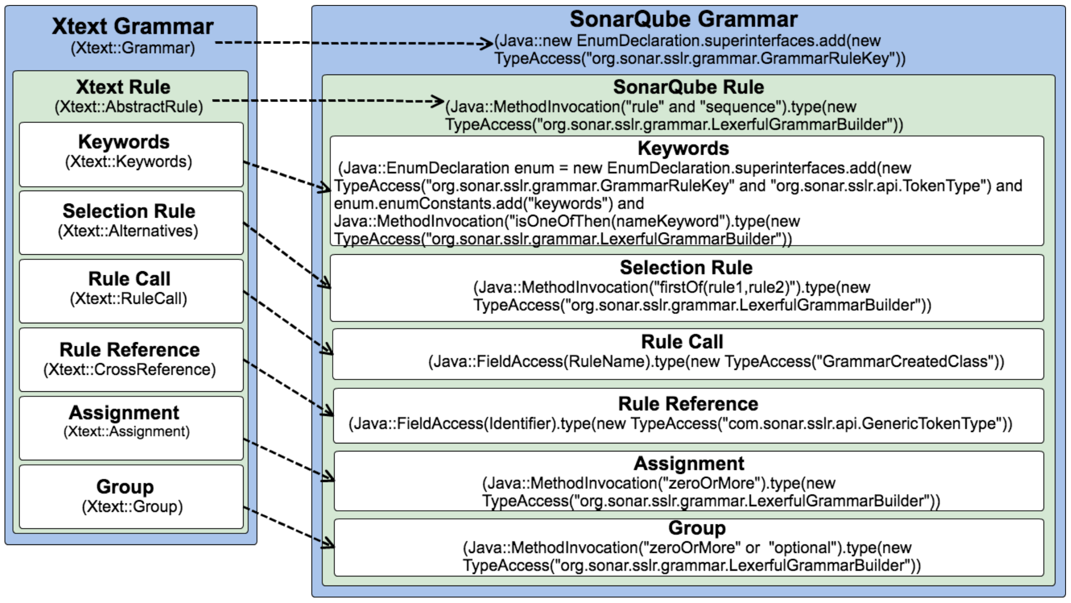

Xtext Grammar
(Xtext::Grammar)
Xtext Rule
(Xtext::AbstractRule)
Keywords
(Xtext::Keywords)
Selection Rule
(Xtext::Alternatives)
Rule Call
(Xtext::RuleCall)
Rule Reference
(Xtext::CrossReference)
Assignment
(Xtext::Assignment)
Group
(Xtext::Group)
SonarQube Grammar
(Java::new EnumDeclaration.superinterfaces.add(new TypeAccess("org.sonar.sslr.grammar.GrammarRuleKey"))
SonarQube Rule
(Java::MethodInvocation("rule" and "sequence").type(new TypeAccess("org.sonar.sslr.grammar.LexerfulGrammarBuilder"))
Keywords
(Java::EnumDeclaration enum = new EnumDeclaration.superinterfaces.add(new TypeAccess("org.sonar.sslr.grammar.GrammarRuleKey" and "org.sonar.sslr.api.TokenType") and enum.enumConstants.add("keywords") and Java::MethodInvocation("isOneOfThen(nameKeyword").type(new TypeAccess("org.sonar.sslr.grammar.LexerfulGrammarBuilder"))
Selection Rule
(Java::MethodInvocation("firstOf(rule1,rule2)").type(new TypeAccess("org.sonar.sslr.grammar.LexerfulGrammarBuilder"))
Rule Call
(Java::FieldAccess(RuleName).type(new TypeAccess("GrammarCreatedClass"))
Rule Reference
(Java::FieldAccess(Identifier).type(new TypeAccess("com.sonar.sslr.api.GenericTokenType"))
Assignment
(Java::MethodInvocation("zeroOrMore").type(new TypeAccess("org.sonar.sslr.grammar.LexerfulGrammarBuilder"))
Group
(Java::MethodInvocation("zeroOrMore" or "optional").type(new TypeAccess("org.sonar.sslr.grammar.LexerfulGrammarBuilder"))

Figure 2: Mapping between grammars elements

menu associated to the .xtext grammar files (see Figure 3). The additional command can be used to generate the source code of Java projects. 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.

The users will then be able to parse and analyse their text-based models based on the new DSL in the same manner that they do with any other built-in language. Meanwhile, the DSL users will be able to generate reports with charts, to define alerts, to compute derived metrics, to access to the historical data, and so on.

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 an interoperability 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 [14], which deploys a number of descriptive use scenarios for different knowledge fields. For each scenario, 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 Figure 4)

[13] https://maven.apache.org
[14] http://tatyperson.github.io/Vary/

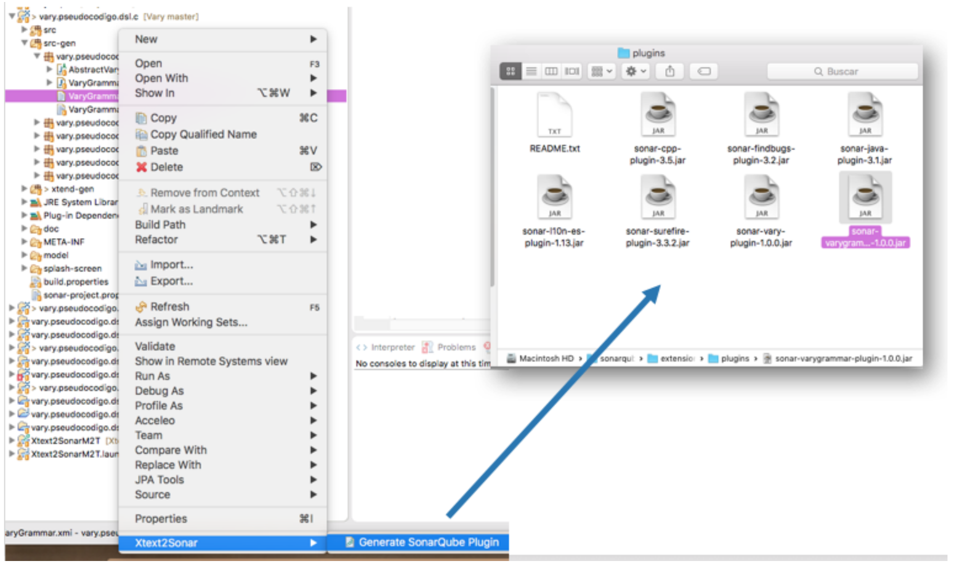


Figure 3: Xtext2Sonar contextual menu

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

Artifact	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

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.

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 Figure 5.

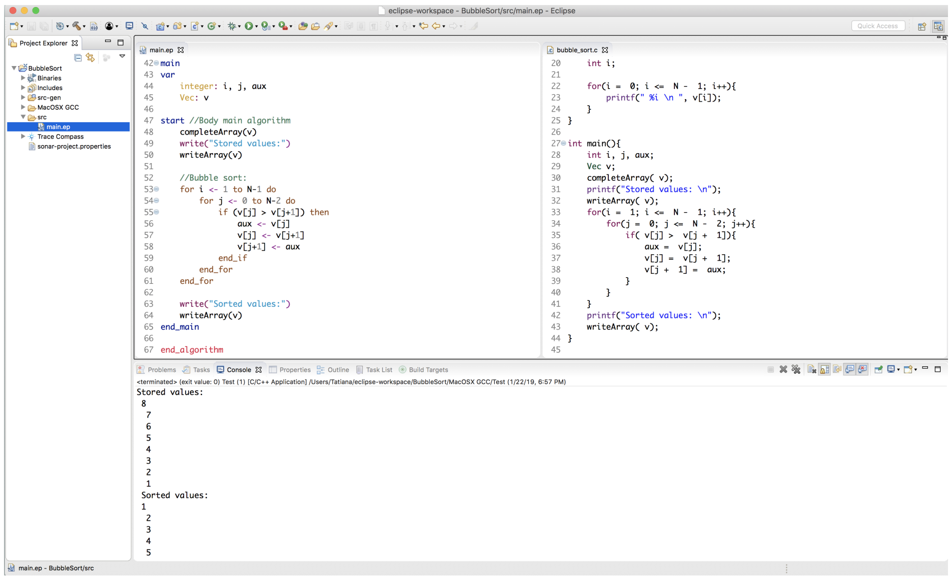
In this case, all of the artifacts required by Sonar for parsing the Vary language were initially developed from scratch. In fact, the motivation of this paper arose from the lessons that were learned during the development of Vary. *Xtext2Sonar* was subsequently applied for comparative purposes and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

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 1,300 lines of Java code.

4.2 Analysing sheet music

LilyPond[15] is a tool for musicians who want to produce sheet music [22]. 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

[15] http://lilypond.org/

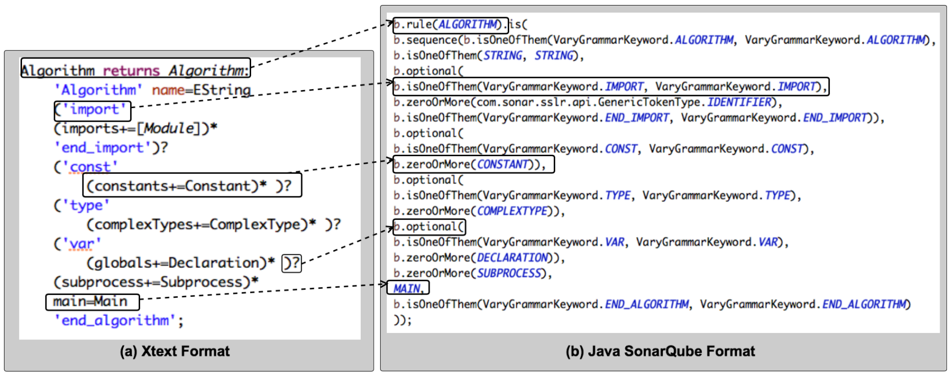


```
main
var
    integer: i, j, aux
    Vec: v

start //Body main algorithm
    completeArray(v)
    write("Stored values:")
    writeArray(v)

    //Bubble sort:
    for i <- 1 to N-1 do
        for j <- 0 to N-2 do
            if (v[j] > v[j+1]) then
                aux <- v[j]
                v[j] <- v[j+1]
                v[j+1] <- aux
            end_if
        end_for
    end_for

    write("Sorted values:")
    writeArray(v)
end_main

end_algorithm
```

```
    int i;

    for(i =  0; i <=  N -  1; i++){
        printf(" %i \n ", v[i]);
    }
}

int main(){
    int i, j, aux;
    Vec v;
    completeArray( v);
    printf("Stored values: \n");
    writeArray( v);
    for(i =  1; i <=  N -  1; i++){
        for(j =  0; j <=  N -  2; j++){
            if( v[j] >  v[j +  1]){
                aux =  v[j];
                v[j] =  v[j +  1];
                v[j +  1] =  aux;
            }
        }
    }
    printf("Sorted values: \n");
    writeArray( v);
}
```

```
<terminated> (exit value: 0) Test (1) [C/C++ Application] /Users/Tatiana/eclipse-workspace/BubbleSort/MacOSX GCC/Test (1/22/19, 6:57 PM)
Stored values:
 8
  7
  6
  5
  4
  3
  2
  1
 Sorted values:
 1
  2
  3
  4
  5
```

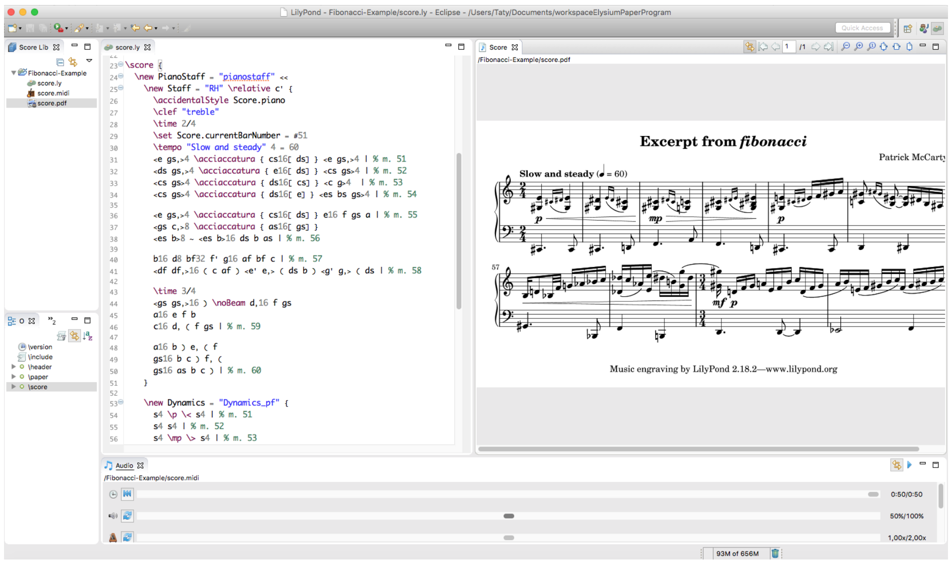
Figure 4: Vary IDE



```
Algorithm returns Algorithm:
    'Algorithm' name=EString
    ('import'
    (imports+=[Module])*
    'end_import')?
    ('const'
        (constants+=Constant)* )?
    ('type'
        (complexTypes+=ComplexType)* )?
    ('var'
        (globals+=Declaration)* )?
    (subprocess+=Subprocess)*
    main=Main
    'end_algorithm';
```

```
b.rule(ALGORITHM).is(
b.sequence(b.isOneOfThem(VaryGrammarKeyword.ALGORITHM, VaryGrammarKeyword.ALGORITHM),
b.isOneOfThem(STRING, STRING),
b.optional(
b.isOneOfThem(VaryGrammarKeyword.IMPORT, VaryGrammarKeyword.IMPORT),
b.zeroOrMore(com.sonar.sslr.api.GenericTokenType.IDENTIFIER),
b.isOneOfThem(VaryGrammarKeyword.END_IMPORT, VaryGrammarKeyword.END_IMPORT)),
b.optional(
b.isOneOfThem(VaryGrammarKeyword.CONST, VaryGrammarKeyword.CONST),
b.zeroOrMore(CONSTANT)),
b.optional(
b.isOneOfThem(VaryGrammarKeyword.TYPE, VaryGrammarKeyword.TYPE),
b.zeroOrMore(COMPLEXTYPE)),
b.optional(
b.isOneOfThem(VaryGrammarKeyword.VAR, VaryGrammarKeyword.VAR),
b.zeroOrMore(DECLARATION)),
b.zeroOrMore(SUBPROCESS),
MAIN,
b.isOneOfThem(VaryGrammarKeyword.END_ALGORITHM, VaryGrammarKeyword.END_ALGORITHM)
));
```

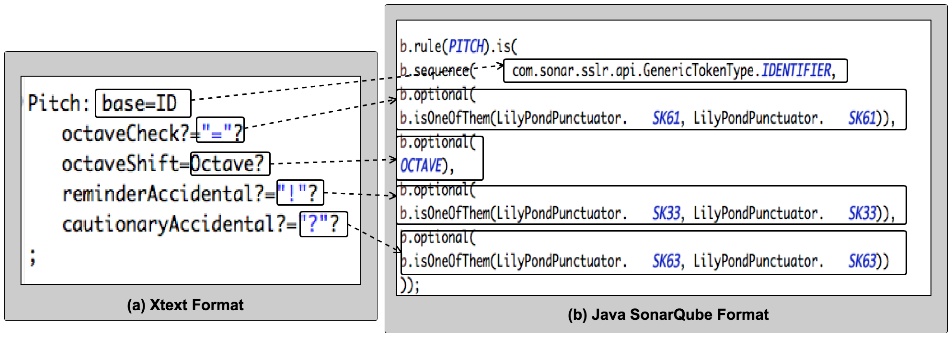
Figure 5: Correspondence between grammar code elements of Vary DSL


LilyPond - Fibonacci-Example/score.ly - Eclipse - /Users/Taty/Documents/workspaceElysiumPaperProgram
Score Lib
Fibonacci-Example
score.ly
score.midi
score.pdf
\version
\include
\header
\paper
\score
Score
/Fibonacci-Example/score.pdf
Excerpt from fibonacci
Patrick McCarty
Slow and steady (♩ = 60)
Music engraving by LilyPond 2.18.2—www.lilypond.org
Audio
/Fibonacci-Example/score.midi
0:50/0:50
50%/100%
1,00x/2,00x
93M of 656M

score.ly

```
\score {
  \new PianoStaff = "pianostaff" <<
    \new Staff = "RH" \relative c' {
      \accidentalStyle Score.piano
      \clef "treble"
      \time 2/4
      \set Score.currentBarNumber = #51
      \tempo "Slow and steady" 4 = 60
      <e gs,>4 \acciaccatura { cs16[ ds] } <e gs,>4 | % m. 51
      <ds gs,>4 \acciaccatura { e16[ ds] } <cs gs>4 | % m. 52
      <cs gs>4 \acciaccatura { ds16[ cs] } <c g>4  | % m. 53
      <cs gs>4 \acciaccatura { ds16[ e] } <es bs gs>4 | % m. 54

      <e gs,>4 \acciaccatura { cs16[ ds] } e16 f gs a | % m. 55
      <gs c,>8 \acciaccatura { as16[ gs] }
      <es b>8 ~ <es b>16 ds b as | % m. 56

      b16 d8 bf32 f' g16 af bf c | % m. 57
      <df df,>16 ( c af ) <e' e,> ( ds b ) <g' g,> ( ds | % m. 58

      \time 3/4
      <gs gs,>16 ) \noBeam d,16 f gs
      a16 e f b
      c16 d, ( f gs | % m. 59

      a16 b ) e, ( f
      gs16 b c ) f, (
      gs16 as b c ) | % m. 60
    }

    \new Dynamics = "Dynamics_pf" {
      s4 \p \< s4 | % m. 51
      s4 s4 | % m. 52
      s4 \mp \> s4 | % m. 53
```

Figure 6: LilyPond IDE



```
Pitch: base=ID
    octaveCheck?="="?
    octaveShift=Octave?
    reminderAccidental?="!"?
    cautionaryAccidental?="?"?
;
```

```
b.rule(PITCH).is(
b.sequence(
com.sonar.sslr.api.GenericTokenType.IDENTIFIER,
b.optional(
b.isOneOfThem(LilyPondPunctuator.    SK61, LilyPondPunctuator.    SK61)),
b.optional(
OCTAVE),
b.optional(
b.isOneOfThem(LilyPondPunctuator.    SK33, LilyPondPunctuator.    SK33)),
b.optional(
b.isOneOfThem(LilyPondPunctuator.    SK63, LilyPondPunctuator.    SK63))
));
```

Figure 7: Correspondences between grammar code elements of LilyPond DSL

melody and lyrics (see Figure 6). 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.

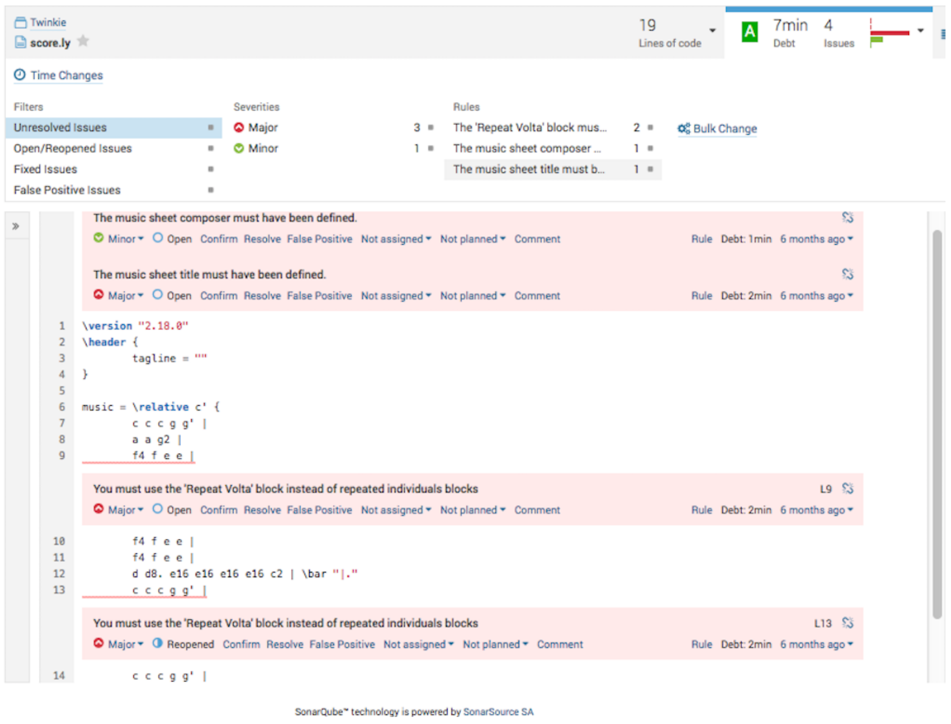
Integrating the computer language supported by this tool into SonarQube might enable musicians to gain a deeper understanding of certain aspects of the musical pieces and their evolution over time. To this end, *Xtext2Sonar* was applied for automatically generating the artifacts required to analyse music compositions in SonarQube. Figure 7 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 1,000 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 8 shows a screenshot of SonarQube while analysing a given music sheet.

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

Artifact	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



The music sheet composer must have been defined.
Minor ▾ Open Confirm Resolve False Positive Not assigned ▾ Not planned ▾ Comment Rule Debt: 1min 6 months ago ▾

The music sheet title must have been defined.
Major ▾ Open Confirm Resolve False Positive Not assigned ▾ Not planned ▾ Comment Rule Debt: 2min 6 months ago ▾

```
1   \version "2.18.0"
2   \header {
3           tagline = ""
4   }
5
6   music = \relative c' {
7           c c c g g' |
8           a a g2 |
9           f4 f e e |
```

You must use the 'Repeat Volta' block instead of repeated individuals blocks L9
Major ▾ Open Confirm Resolve False Positive Not assigned ▾ Not planned ▾ Comment Rule Debt: 2min 6 months ago ▾

```
10          f4 f e e |
11          f4 f e e |
12          d d8. e16 e16 e16 e16 c2 | \bar "|."
13          c c c g g' |
```

You must use the 'Repeat Volta' block instead of repeated individuals blocks L13
Major ▾ Reopened Confirm Resolve False Positive Not assigned ▾ Not planned ▾ Comment Rule Debt: 2min 6 months ago ▾

```
14          c c c g g' |
```

SonarQube™ technology is powered by SonarSource SA

Figure 8: SonarQube showing non-conformities of a 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. This framework relies on the use of DSLs for the different kinds of artifacts.

5 A case study on assessing learning of algorithm design

To ensure the applicability of the metrics generated for the Vary language the following case study was conducted. It has been accomplished by following the steps described in the guidelines for conducting and reporting case study research in software engineering [27].

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. This study enables to discover insights and generate ideas for new research.

The study was conducted in the context of the subject *Introduction to Programming* in the Degree in Computer Engineering at University of Cádiz. 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 with 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) 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.

- *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%).

[16]http://sculptorgenerator.org
[17]http://www.tango-controls.org
[18]http://www.eclipse.org/smarthome

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%

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 a output sentence (write).*
- *Too many global variables.*
- *Code lines that are too long.*
- *Procedure or function with no documentation.*
- *Lines with bad indentation.*

Furthermore, a 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 reveals 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 ($r = 0{,}9673$; $p = 7{,}734e\text{-}19$), as shown in Figure 9.

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

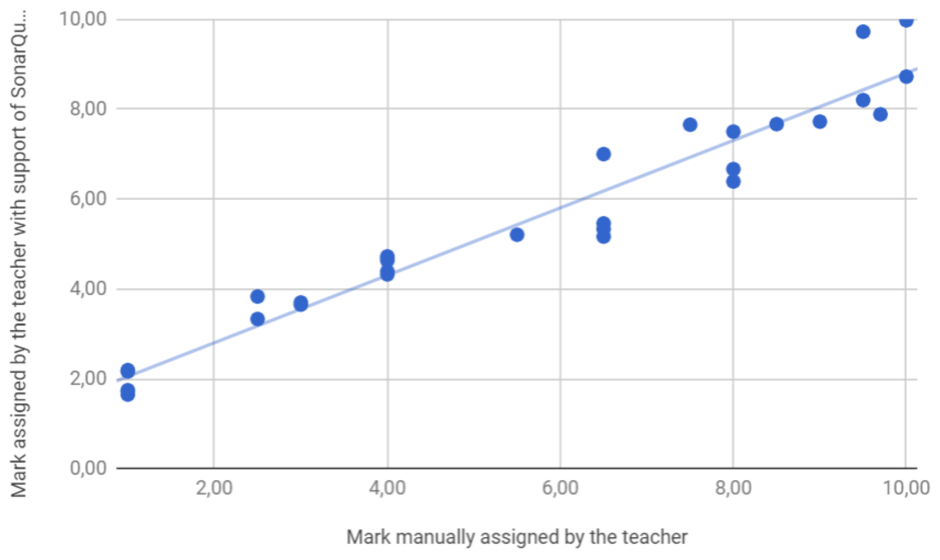


Figure 9: Correlation between teacher-assigned marks and SonarQube-computed ones

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 systematize 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 students 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

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 satisfactory in a specified context of use. To check the applicability of SonarQube's metrics to assess the quality of music sheets, a usability evaluation with experts was conducted.

6.1 Usability study design

This usability test aims to find if the automatic computation of metrics for analysing the quality of music sheets written with the Lilypond notation is suitable for musicians. This test was defined and performed by following the guidelines provided in [26].

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 to 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 [11]. 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. Pre-test questionnaires were used to determine the initial state of users' opinions and knowledge, before doing the observations[21]. Post-test 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 pre-test 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 post-test 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.

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:

[19] https://goo.gl/ju1zwd
[20] https://goo.gl/xsPzgY
[21] https://goo.gl/17pGq9
[22] https://goo.gl/dkZB3j

- *The whole sample*: a total of 14 musicians participated in the study.
- *Experience with music sheet composition software*: most of the participants (12) were experienced with this kind of tool, whereas very few (2) did not have previous experience.
- *Professional activity of the participants*: researchers (7), teachers (6), performers (6), musicologists (4), a music librarian (1) and a music therapist (1).
- *Highest academic degree achieved by the participants*: graduates (5), postgraduates (5) and doctorates (3).

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 post-test are presented below.

Table 4: Usability evaluation results

Dimension	Participants	Learnability	Efficiency	Satisfaction	Utility
Professional activity of the participants					
Librarian	1	0,00%	0,00%	0,00%	3,00
Therapist	1	100,00%	100,00%	100,00%	9,00
Teacher	6	83,33%	66,67%	83,33%	7,17
Performer	6	83,33%	66,67%	83,33%	7,83
Researcher	7	85,71%	57,14%	85,71%	7,43
Musicologist	4	75,00%	50,00%	100,00%	9,00
Experience with music sheet composition software					
Experienced	12	91,67%	75,00%	100,00%	8,17
Non-experienced	2	50,00%	50,00%	50,00%	5,50
Highest academic degree achieved by the participants					
Graduate	6	83,33%	83,33%	83,33%	8,09
Postgraduate	5	100,00%	60,00%	100,00%	9,00
Doctorate	3	66,67%	66,67%	100,00%	6,67
All	14	85,71%	71,43%	92,86%	7,79

- *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 85,71% of the participants gave us a positive response, decreasing until 50% for users without previous experience with music production software.
- *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?* A total of 71,4% of the sample answered affirmatively. However this value drops to 57,14% and 50,00% in the case of researchers and musicologists.
- *Satisfaction: Would you consider the inclusion of this type of tool to analyse the quality of sheet music?* Most of the participants (92,86%) deemed that the use of this kind of tool was interesting. Predictably, this value falls to 50% for non-experienced users.
- *Utility: "How useful would you consider this tool?"* The results obtained in this rating have been quite promising, revealing a score of 7.79 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 observed the sheet music. However, they rated the tool with 9 out 10, which gives us an insight into the real potential that this approach may offer.

  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 domain-specific languages 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.

A demo instance of SonarQube is provided[23] to analyse artifacts 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.

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 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.

It would not be difficult to cope with additional DSL development frameworks because of the use of EBNF input models. However, this cannot be affirmed for quality platforms due to the lack of

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

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 [13].

References

[1] Abran, A., Khelifi, A., Suryn, W., Seffah, A.: Usability meanings and interpretations in iso standards. Software Quality Journal **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. Information and Software Technology **64**, 52–73 (2015)

[4] Besova, G., Steenken, D., Wehrheim, H.: Grammar-based model transformations: Definition, execution, and quality properties. Computer Languages, Systems & Structures **43**, 116 – 138 (2015)

[5] Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing Ltd (2013)

[6] Boud, D.: Sustainable assessment: rethinking assessment for the learning society. Studies in continuing education **22**(2), 151–167 (2000)

[7] Brambilla, M., Cabot, J., Wimmer, M.: Model-driven software engineering in practice. Synthesis Lectures on Software Engineering **1**(1), 1–182 (2012)

[8] Brown, N.C.C., Altadmri, A.: Novice java programming mistakes: Large-scale data vs. educator beliefs. ACM Transactions on Computing Education (TOCE) **17**(2), 7:1–7:21 (2017). DOI 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.[15 Oct 2013] (2010)

[10] Fowler, M.: Domain-specific languages. Pearson Education (2010)

[11] Gould, E.: Behind bars: The definitive guide to music notation. Faber Music (2011)

[12] 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)

[13] 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)

[14] Hevner, A., Chatterjee, S.: Design research in information systems: theory and practice, vol. 22. Springer Science & Business Media (2010)

[15] ISO/IEC: 14977 - Information technology – Syntactic metalanguage – Extended BNF. Standard, International Organization for Standardization (1996)

[16] 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)

[17] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of computer programming **72**(1), 31–39 (2008)

[18] Kappel, G., Langer, P., Retschitzegger, W., Schwinger, W., Wimmer, M.: Model transformation by-example: a survey of the first wave. In: Conceptual Modelling and Its Theoretical Foundations, pp. 197–215. Springer (2012)

[19] Kruchten, P., Nord, R.L., Ozkaya, I.: Technical debt: From metaphor to theory and practice. IEEE Software **29**(6), 18–21 (2012)

[20] 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)

[21] Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys **37**(4), 316–344 (2005)

[22] 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)

[23] Parr, T., Fisher, K.: LL (*): the foundation of the ANTLR parser generator. In: ACM SIGPLAN Notices, vol. 46, pp. 425–436. ACM (2011)

[24] Rochimah, S., Arifiani, S., Insanittaqwa, V.F.: Non-source code refactoring: A systematic literature review. International Journal of Software Engineering and Its Applications **9**(6), 197–214 (2015)

[25] Rose, K.: Project quality management: why, what and how. J. Ross Pub. (2005)

[26] Rubin, J., Chisnell, D.: Handbook of usability testing: howto plan, design, and conduct effective tests. John Wiley & Sons (2008)

[27] Runeson, P., Höst, M.: Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering **14**(2), 131–164 (2009). DOI 10.1007/s10664-008-9102-8

[28] Stevenson, A., Cordy, J.R.: A survey of grammatical inference in software engineering. Science of Computer Programming **96**, 444–459 (2014)

[29] 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)

[30] Tomas, P., Escalona, M., Mejias, M.: Open source tools for measuring the internal quality of java software products. a survey. Computer Standards & Interfaces **36**(1), 244 – 255 (2013)