A Toolchain for Metrics-based Comparison of COBOL and Migrated Java Systems

Jan Jelschen, Andreas Winter
Carl von Ossietzky Universität, Oldenburg, Germany
{jelschen,winter}@se.uni-oldenburg.de

Abstract

Migrating COBOL legacy systems to Java results in functional equivalent systems expressed in the new language, while the programming paradigm remains that of the old systems. The quality of translated code is therefore assumed to be inferior, if held to the standards of the target language's paradigm. This paper presents an integrated toolchain enabling metrics-based comparisons of original and translated systems to substantiate or refute this hypothesis, characterize the change in code quality, and gain insights for the improvement of translation tools.

1 Introduction

Migrating legacy systems written in old programming languages like COBOL to a more modern language like Java often becomes a necessity, e.g. due to required hard- or software platforms being phased out, or a lack of qualified personnel to maintain the system. Such migrations have to be supported by tools automating translation. However, changing between fundamentally different programming languages also incurs a paradigm shift: while COBOL is procedural, Java is designed to support object-orientation. Tools can map a COBOL program to a functional equivalent Java program, but usually cannot introduce an object-oriented design into a program conceived procedural (cf. Terekhov and Verhoef [7]). Therefore, the code quality of the translated system (e.g. in terms of maintainability) is expected to be lower than that of the original system, at least when evaluated under quality criteria of the target language's paradigm.

In this paper, a toolchain enabling evaluation of code quality metrics on COBOL and translated Java is presented, allowing direct comparisons of quality characteristics. It facilitates the quantification and characterization of quality differences, providing insights to enhance translation tools.

In Section 2, requirements for the toolchain and its composition are described. Section 3 discusses how to approach comparisons of COBOL and translated Java systems using the toolchain and appropriate metrics. The paper concludes with an outlook in Section 4.

2 Toolchain

The toolchain has to support three workflows: *translating COBOL to Java*, *evaluating metrics on COBOL*, and *evaluating metrics on Java*. In the latter case, metrics on Java systems which have *not* been translated from COBOL also have to be evaluable. This extends the scope and flexibility of the toolchain, particularly allowing to use other Java systems as a reference, when direct comparisons with COBOL systems is infeasible (this is further discussed in Sec. 3).

Apart from this, it has to be easy to *extend additional functionality*, e.g. visualization of metric results. An interesting experiment would be to compare different COBOL-to-Java-translators on the basis of metrics evaluated over their respective output. This requires the ability to smoothly *substitute different translator implementations* while leaving the rest of the toolchain unchanged.

Finally, metrics to be evaluated are not known in advance, depending on each specific experiment, necessitating *facile addition and modification of metrics*.

To facilitate the required flexibility, the toolchain is conceived in terms of *services*, abstracting from concrete tools.

2.1 Design and Implementation

Substantial parts of the infrastructure and individual tools have been developed and used in the context of the SOAMIG project [3]. As a basic data structure to hold representations of the processed systems, TGraphs have been chosen. Properties of TGraph instances can be examined using the graph query language GReQL [5], used here to express code metrics.

Data and control flows inside the toolchain, and services involved are depicted in Figure 1 as activity diagram, and explained in the following: *translation* subsumes services concerned with converting Java to COBOL, while *transformation* services provide the required data structures for *metrics* to be evaluated.

Translation. The actual COBOL-to-Java migration is realized by a series of services to *parse COBOL*, translate *COBOL to Java*, and *generate Java* source code. The implementation is provided through the "CoJaC" tools by *pro et con* [1].

The result of parsing COBOL is ouput as an XML file containing an abstract syntax tree (AST). Translating to or *parsing Java* results in a similar XML (parsing Java is, again, provided by a *pro et con* tool.). In Figure 1, these files are shown as datastores *COBOL-AST* and *Java-AST*, respectively.

Transformation. The XML output of the translation tools is TGraph-compatible, and is read into JGraLab's internal (in-memory) representation (depicted twice in the diagram as *XML2TG*).

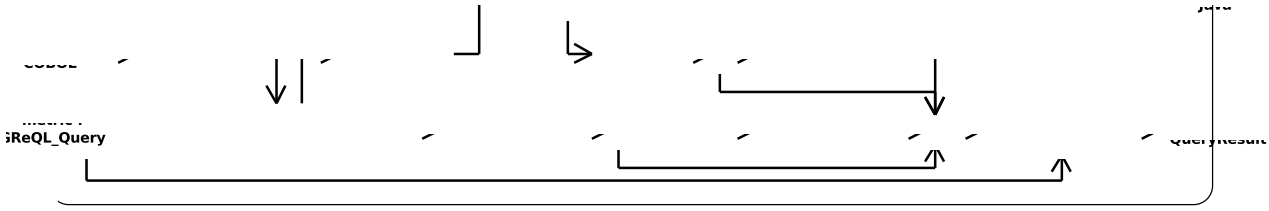

Figure 1: Data and control flow between services of the toolchain.

Metrics can be evaluated directly on the abstract syntax tree representations of either COBOL or Java code. However, many relevant metrics, like *cyclomatic complexity* [6], are more easily evaluated over the control flow graph (CFG) of a program. Therefore, two additional services (*Java AST2CFG* and *COBOL AST2CFG*) to convert Java and COBOL ASTs to CFGs have been implemented for the toolchain. The CFG-representation has the added benefit of being programming language-independent. This allows to express each metric in a single GReQL statement, whereas AST-based metrics have to be formulated over each languages' meta-model separately.

Metrics evaluation. Metrics are not "hard-wired" into the toolchain. Instead, they are provided as an input, expressed in GReQL. Any metric expressible in terms of abstract syntax trees or control flow graphs can be added without changing the infrastructure at all, making the approach very flexible in this regard.

Measures of complexity or structuredness can often be expressed over the CFG [2, pp. 167]. This makes this class of metrics language-independent, and thereby suitable for a direct comparison of original and translated software systems. In addition, the CFG-meta-model is very simple, with only two fundamental concepts: *basic blocks*, connected by *control flows*. Following is an example, showing the *cyclomatic complexity* metric as graph query over the CFG.

```
count(E{ControlFlow}) -
count(V{BasicBlock})  +
count(V{Procedure})   * 2
```

It sums up all basic blocks and all control flows, and subtracts the former from the latter. Then, the number of connected components in the graph times two is added to that (the detection of connected components is simplified here by the fact that there is an "auxiliary" *procedure* node for each of them).

The selection of metrics suitable for comparing COBOL and Java systems has not yet been completed. It is discussed in the following section.

3 Experiments

Due to lack of industrial scale COBOL systems, so far, only small code examples were tested, whose size severely limit the validity of evaluated metrics.

A challenge for further experiments will be to actually achieve comparability: On the one hand, language-independent metrics like *cyclomatic complexity* can be evaluated on both the original and the translated system. The informative value of such metrics – e.g. whether one can directly infer code quality from them – is disputable, though [2, p. 181].

On the other hand, it is desirable to measure the quality of the translated system with respect to the programming paradigm of the target language, i.e. evaluating object-orientation metrics on Java code. There are, however, no canonical COBOL-counterparts for such metrics, leaving open the question of a meaningful comparison.

One approach could be to establish mean values for object-oriented metrics to compare against, by running a large body of representative Java systems through the metric evaluation. While this would still not allow a direct comparison, the quality of a translated system could be rated to be of better, similar, or worse quality than other systems written in the same programming language.

4 Outlook

The toolchain presented here has been conceived with the additional purpose of serving as a case study for tool interoperability. Issues identified in this regard include overhead caused by *data format conversion* and the amount of *glue code* which needs to be written, as well as *platform-specific barriers*. These insights will guide ongoing work on service-based tool interoperability, finally aimed at producing a service-oriented, component-based interoperability framework for software evolution tools [4].

References

[1] U. Erdmenger and D. Uhlig. Ein Translator für die COBOL-Java-Migration. *Softwaretechnik-Trends*, 31(2), 2011.

[2] N. E. Fenton. *Software Metrics: A Rigorous Approach*. Chapman and Hall, 1991.

[3] A. Fuhr, A. Winter, et al. Model-Driven Software Migration - Process Model, Tool Support and Application. To appear in: A. Ionita, M. Litoiu, G. Lewis. *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments*, IGI Global, 2012.

[4] J. Jelschen and A. Winter. Towards a Catalogue of Software Evolution Services. *Softwaretechnik-Trends*, 31(2), 2011.

[5] B. Kullbach and A. Winter. Querying as an enabling technology in software reengineering. In *CSMR 1999*, pages 42–50. IEEE CS, 1999.

[6] T. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, (4):308–320, 1976.

[7] A. Terekhov and C. Verhoef. The realities of language conversions. *IEEE Software*, 17(6):111–124, 2000.