# Towards a Generic DSL for Automated Marking Systems

Fritz Solms(✉) and Vreda Pieterse

Department of Computer Science, University of Pretoria, Pretoria, South Africa
fritz@solms.co.za, vpieterse@cs.up.ac.za

**Abstract.** The automated static and dynamic assessment of programs makes it practical to increase the learning opportunities of large student classes through the regular assessment of programming assignments. Automatic assessments are traditionally specified in tool-specific languages which are closely linked to the functionality and implementation of a particular tool. This paper considers existing specification languages for assessments and proposes a generic and extensible domain-specific language for the specification of programming assignment assessments.

**Keywords:** DSL · Syntax · Automated assessment · Software testing

## 1 Introduction

In undergraduate programming courses it is particularly important to request students to regularly complete programming assignments so that they can be graded and be given meaningful feedback. At institutions such as the University of Pretoria, many of these undergraduate courses have many hundreds of students and manual assessment by teaching assistants is resource intensive, time-consuming, tedious and bound to be inconsistent. Consequently there has been a quest to develop automated grading systems and to evolve them so as to provide more meaningful assessments and improved feedback to students [1,14–16].

The computer science department at the University of Pretoria uses an in-house developed automatic marking system, *Fitchfork*, to mark *C* and *C++* programming assignments for first-year computer science students [14]. The instructor specifies a set of test cases beforehand. Students then upload their source code and any supporting artefacts onto a Linux server where the source is unpacked into a sandbox where it is compiled and executed. The output of a students' program is matched against the expected output for each given test case.

One critical aspect of such systems is the assessment specification which the instructors must develop. The specification includes information on how the programming assignment will be processed (e.g. compiled and executed), how it will be assessed, how marks are aggregated and the feedback that will be given to the students. It is vital that the specification of such assessment should be intuitive, efficient and not prone to errors.


S. Gruner (Ed.): SACLA 2016, CCIS 642, pp. 59–66, 2016.
DOI: 10.1007/978-3-319-47680-3_6

It has been found that informally specified languages tend to become excessively complex and error prone as systems evolve so as to perform more complex assessments [17]. For this reason, researchers have been exploring the possibility of developing more robust specification languages for assessments, which can be formally verified and for which one can easily develop supporting tools.

Domain-specific languages (DSLs) are commonly introduced to provide simple, consistent domain-centric languages which are easy to use and easy to process [8]. The availability of DSL work-benches [9] simplifies the task of developing and verifying a domain-specific language and of enriching the language with tools such as language-aware editors, and transformation tools.

We are currently developing an *assessment-specification language* (ASL) for the specification of a generic and extensible assessment process. What differentiates our language from the languages developed for other tools is (a) a semi-formal specification of the semantics using a metamodel. This model can be transformed into an ontology for verifying language qualities. Other features are (b) the ability to support different textual and diagrammatic concrete syntaxes; (c) extensive tool support for generating syntax-aware textual and diagrammatic editors, model validation, model transformation and code generation, and (d) the language is tool-independent so that its scope is constrained to the specification of what is required to be done, not how a particular tool would perform the assessment. The final feature is (e) one can write an adapter layer which aggregates the assessment across different assessment tools, i.e. by transforming aspects of an assessment specification which can be handled by a particular tool to the specification that can be processed by that tool.

In principle, the use of such an ASL will enable memoranda which were specified for one tool to transported to another tool, contributing to simplifying the sharing of assessments across platforms and tools.

Section 2 discusses a few existing automatic assessment systems in terms of their contribution to the types of assessments that should be specifiable using the ASL we propose in this paper. We also refer to other authors who have proposed DSLs similar to the ASL we propose. Section 3 lists the objects that may be used by an automatic assessment tool—these constitute the semantic scope for the proposed ASL. We also discuss its quality requirements. In Sect. 4 we describe how we developed our ASL. We justify the tools we used, show the abstract syntax we developed using a UML class diagram and discuss our design decisions. Methods to transform this abstract syntax to concrete syntaxes are briefly mentioned. In the final two sections, we summarise what we have achieved, highlight the benefits of having our ASL and discuss future work to improve this ASL and promote its use.

## 2 Related Work

Wilcox provides a survey of the testing strategies used to grade programming assignments [19]. He discusses (a) *textual output comparison*, (b) *output analysis* which performs further processing of the output to, for example, assess whether

the output is internally consistent, (c) *stream control* which is used to interleave input and output in order to drive a particular program flow, (d) *testing* against an API in a way similar to unit testing, (e) *source code analysis*, (f) *issue detection* used to observe the issues encountered during the assessment process and the occurrence of compilation or execution issues (e.g. non-termination).

A tool for automatically analysing and assessing the programming style of C++ programs was implemented by Ala-Mutka *et al.* [3]. They claim that its use improved the quality of their coursework and that students learned to pay better attention to their coding practices. Ponženel *et al.* [15] acknowledge that the addition of white-box testing (i.e. structural evaluation) to the predominant black-box testing (i.e. functional testing) applied by most systems is essential for the pedagogically sound and fair evaluation of student programs. For example, the AutoLEP system [18] combines static analysis with dynamic testing when evaluating student programs. Static analysis includes syntactic and structural checking. Similarly, eGrader [2] uses JUnit for dynamic analysis and a static evaluation based on a graph representation of the program.

Fonte *et al.* [7] illustrate the need to allow the identification of partially correct programs with semantic errors. Fitchfork [14] achieves this by comparing the output of a program with the known output that a program would produce if it contained an anticipated semantic error. The detection of such expected wrong output enables us to give the student feedback about the semantic error in the program.

Lately there has been a move toward developing DSLs to describe assessments [7,17]. Fonte *et al.* [7] propose a DSL they call OSSL which supports the semantic specification of expected program output. They use extension modules to specify the integration between the Oto grading system and external tools such as a compiler, and JUnit. Insa and Silva [12] developed an assessment Java library with abstraction methods for verifying the properties of code and a DSL built on top of it for assessment templates.

The manual specification of assessments for automatic assessment tools is likely to be tedious. The specification can be simplified when using an ASL. The reduced complexity will probably contribute to improving the quality of the assignments that are specified in this manner.

## 3 Requirements for the Proposed DSL

This section discusses the requirements of the generic ASL we intend to specify. Firstly, we identify the semantic scope of the ASL in terms of the essential elements one should be able to specify when using the proposed ASL. We then discuss the critical quality requirements for the ASL.

### 3.1 Semantic Scope

The ASL needs to be an open language whose scope can be extended with add-on modules. The language core should contain the essential elements needed by

all assessment tools. We have identified that such core should include: (a) the specification of process steps and the dependencies between process steps, (b) the concept of an assessment which can be extended with specific assessment types, (c) the basic infrastructure for specifying mark allocation and aggregation and (d) the infrastructure to identify error scenarios in the assessed code in order to give the students insightful feedback on ways of improving their programs.

Many existing assessment tools assess the output of program execution. To accommodate these, we decided that this type of assessment should be included in the core language specification. Therefore the ASL should have means to specify simple text output assessments which can be used to assess the output of a program's execution. This type of assessment can be employed to assess the output of other kinds of processing steps, for example the compilation process. It can even be utilised to perform a static assessment of the code by assessing the output of a file search evaluating the presence of some constructs in the code.

### 3.2 Quality Requirements for the ASL

The ASL can only be expected to be widely adopted if it meets standard usability requirements such as learnability, efficiency, effectiveness, reliability and satisfaction. Dumas and Reddish [6] emphasise that the people who use a product should be able to accomplish their tasks quickly and easily. These must accommodate users who may have different levels of technical skills varying language backgrounds. It is important for users to be able to extend the language in order to specify more specialised processing, assessment and mark aggregation requirements. Assessment specifications must be verifiable against the semantic rules of the language. The ASL should be portable across platforms (e.g. operating systems) as well as across assessment systems. It is expected that one can transform the subset of an assessment to a tool-specific assessment specification for a tool which can be used for that aspect of the assessment. The language must be published as an open public standard so that it is accessible and usable by different assessment tool developers and the users of these tools.

## 4 The Domain-Specific Language

Domain-specific languages can be developed in a variety of technologies. One of the options is that of using the technology support specified by OMG's *Model-Driven Architecture* standard. The advantages of using these standards are that (a) the standard is reasonably mature and it evolves in a controlled way, (b) there are multiple concrete tool implementations for the standard and (c) there are extensive auxiliary tools such as transformation tools and tools for generating language editors [9].

In particular, we separated the abstract syntax (introducing the semantics) from the concrete syntax (used by instructors to specify assessments). This separation facilitates (a) concrete syntax-independent verification of a specification against the semantics of the language, (b) the development of different concrete

syntaxes for users with different levels of technical skills and different home languages and (c) the transformation of an assessment specified in any of the concrete syntaxes to an abstract representation which is independent of the concrete syntax used, hence allowing the uniform processing of assessment specifications across different concrete syntaxes.

### 4.1 Abstract Syntax

Here we introduce the ASL. It is specified using *Ecore* which is an implementation of *EMOF* provided by the *Eclipse* foundation [9].

Figure 1 shows a diagram of the abstract syntax of the core language. The language allows for the specification of a process of multiple processing steps. The order in which the steps are to be executed is specified only indirectly through the dependencies between steps. This simplifies the specification of assessment processes, makes them more maintainable and allows for the concurrent execution of steps which do not have dependencies on one another.

Each processing step optionally specifies a command which is executed as well as zero or more assessments. The commands for example may be to extract an archive, compile the source code, execute the program with specified test data sets, or to execute a unit test. The resources (memory, time/CPU, networking, etc.) which a command can consume may be constrained via one or more resource constraints.

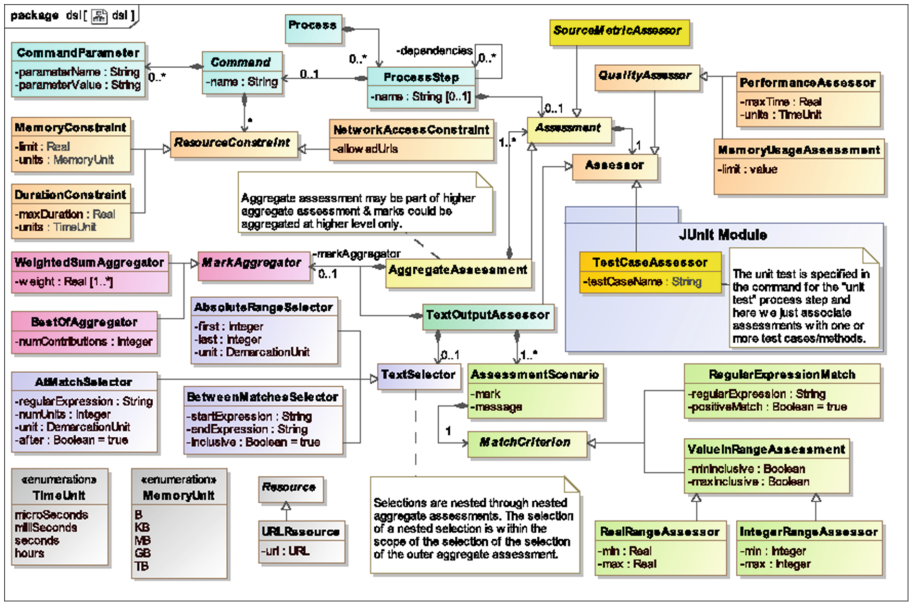


**Fig. 1.** The abstract syntax of the DSL for programming assessments.

The central concept of the language is the abstract concept of an assessment. Assessments can be recursively aggregated into aggregate assessments using different ways to aggregate the marks of the assessment components by selecting appropriate mark aggregators. The language is extensible, allowing additional assessors to be specified in extension modules. Figure 1 illustrates how a *JUnit* extension module is added to the language.

The core language includes the specification of *TextOutputAssessor*s. It is crucial for students to receive insightful feedback on the errors they make. For this reason, *TextOutputAssessor*s allow for the assessment of different output scenarios resembling several variations of correct, partially correct and incorrect solutions, each with their own mark and their own feedback message. Scenarios are identified by matching the output text selection to a particular output pattern. The feedback message associated with a scenario is meant to give the students insights into solution deficiencies and improvement options. To further enrich the assessment, the language allows for multiple scenarios to contribute toward the mark for the assessment. Though one can simply use a "BestOfAggregator" to select the scenario with the highest mark, one can also specify more complex aggregators which reward or penalise certain aspects of a solution, i.e. the same mark aggregators used to aggregate marks across assessments can be used to aggregate the marks accumulated across scenario assessments. The default for aggregating across aggregate assessments is simple-sum aggregation. The default for scenario aggregation is to select the best scenario mark.

### 4.2 Concrete Syntaxes

A domain-specific language allows for the specification of multiple concrete syntaxes. It is largely the specification of different concrete syntaxes of the language that determines the language usability characteristics discussed in Sect. 3.2. A significant amount of work has been done to design DSLs guided by usability metrics [4,5]. The rigorous development of a concrete syntax guided by usability metrics is work which is currently in progress. This will enable us to illustrate the abstract language we have developed so far with a simple English-based syntax.

An example text syntax can be developed in *EMFText* [11] which is a tool which gets as input the syntax specification as a mapping between concrete syntax and abstract syntax elements in a BNF-like notation. *EMFText* can be used to generate a syntax-aware editor and to do the mapping between an assessment specification specified in the concrete syntax and its representation in the abstract syntax.

## 5 Summary

The ASL was designed while keeping in mind the quality requirements for the language. In particular, the language supports the specification of extension modules within which it is possible to provide the semantics required to specify different types of processing commands, assessments and marks aggregation algorithms.

When new elements are being developed, our generic ASL assessment specification must be mapped onto tool-specific assessment specifications for such element to ensure the portability of the language across assessment tools.

Since this is an ecore-based DSL, a wide variety of declarative and imperative model-to-model [10,13] and model-to-text [9] transformation tools are available. Furthermore, transformation can also be specified implicitly by specifying the tool language as a concrete syntax for our ASL. In cases where the assessment specification requires concepts not covered by our ASL, the ASL needs to be extended. Such an extension should only be required to increase the scope of the language, not for technical reasons such as to allow mapping to a tool-specific assessment specification. Any technical enrichment should be made during the mapping transformation.

We have generated a language-aware editor which verifies a concrete assessment specification against the language rules. Further qualities of the language can be specified as static constraints against the language metamodel. This can be done using the *Object Constraint Language* and the *Eclipse OCL libraries* [9].

The usability of the language will be determined by the development of concrete syntaxes for the language and will therefore be the subject of future work.

## 6 Conclusions and Future Work

We have introduced an extensible domain-specific language for the specification of program assessments. The specification of such a language as a domain-specific language has the advantages of being able to specify a variety of concrete syntaxes for different user groups and of having a rich tool set available for generating language-aware editors, assessment validators and transformations for transporting onto tool-specific assessment formats. The focus of our work will now shift to specifying a concrete syntax based on usability guidelines and on assessing such languages by measuring their usability metrics and by performing in-field user testing. We will then specify transformations onto tool-specific assessment specification formats, which will include that of our in-house developed system. Different aspects of assessment (e.g. assessment of source code and dynamic metrics) are expected to be covered and specified in extension modules.

## References

1. Ahoniemi, T., Reinikainen, T.: Aloha: a grading tool for semi-automatic assessment of mass programming courses. In: Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006, Baltic Sea 2006, pp. 139–140. ACM, New York (2006)
2. Al Shamsi, F., Elnagar, A.: An intelligent assessment tool for students' Java submissions in introductory programming courses. J. Intell. Learn. Syst. Appl. **4**(1), 59–69 (2012)
3. Ala-Mutka, K., Uimonen, T., Järvinen, H.M., Knight, L.: Supporting students in C++ programming courses with automatic program style assessment. J. Inf. Technol. Educ. **3**, 245–262 (2004)

4. Albuquerque, D., Cafeo, B., Garcia, A., Barbosa, S., Abrahão, S., Ribeiro, A.: Quantifying usability of domain-specific languages: an empirical study on software maintenance. J. Syst. Softw. **101**, 245–259 (2015)
5. Bariic, A., Amaral, V., Goulão, M.: Usability evaluation of domain-specific languages. In: Eighth International Conference on the Quality of Information and Communications Technology (QUATIC), pp. 342–347, September 2012
6. Dumas, J.S., Redish, J.C.: A Practical Guide to Usability Testing. Intellect Bks, Portland (1999)
7. Fonte, D., da Cruz, D.C., Gançarski, A.L., Henriques, P.R.: A flexible dynamic system for automatic grading of programming exercises. In: 2nd Symposium on Languages, Applications and Technologies, SLATE 2013, Porto, Portugal, pp. 129–144, June 2013
8. Fowler, M.: Domain Specific Languages, 1st edn. Addison-Wesley Professional, Boston (2010)
9. Gronback, R.C.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit, 1st edn. Addison-Wesley Professional, Boston (2009)
10. Guduric, P., Puder, A., Todtenhoefer, R.: A comparison between relational and operational QVT mappings. In: Sixth International Conference on Information Technology: New Generations, ITNG 2009, pp. 266–271, April 2009
11. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Model-based language engineering with EMFText. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2011. LNCS, vol. 7680, pp. 322–345. Springer, Heidelberg (2013)
12. Insa, D., Silva, J.: Semi-automatic assessment of unrestrained Java code: a library, a DSL, and a workbench to assess exams and exercises. In: Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2015, pp. 39–44. ACM, New York (2015)
13. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Sci. Comput. Program. **72**(1–2), 31–39 (2008). http://dx.doi.org/10.1016/j.scico.2007.08.002
14. Pieterse, V.: Automated assessment of programming assignments. In: Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research, CSERC 2013, pp. 4:45–4:56. Open Universiteit, Heerlen, Open Univ., Heerlen, The Netherlands (2013). http://0-dl.acm.org.innopac.up.ac.za/citation.cfm?id=2541917.2541921
15. Poženel, M., Fürst, L., Mahnič, V.: Introduction of the automated assessment of homework assignments in a university-level programming course. In: 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), pp. 761–766, May 2015
16. Tremblay, G., Guérin, F., Pons, A., Salah, A.: Oto, a generic and extensible tool for marking programming assignments. Softw. Pract. Exper. **38**(3), 307–333 (2008)
17. Tremblay, G., Lessard, P.: A marking language for the Oto assignment marking tool. In: Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education, ITiCSE 2011, pp. 148–152. ACM, New York (2011). http://0-doi.acm.org.innopac.up.ac.za/10.1145/1999747.1999791
18. Wang, T., Su, X., Ma, P., Wang, Y., Wang, K.: Ability-training-oriented automated assessment in introductory programming course. Comput. Educ. **56**(1), 220–226 (2011)
19. Wilcox, C.: Testing strategies for the automated grading of student programs. In: Proceedings of the 47th ACM Technical Symposium on Computing Science Education, SIGCSE 2016, pp. 437–442. ACM, New York (2016)