Generation of Document Type Exercises for Automated Assessment

José Paulo Leal
CRACS – INESC-TEC, Porto, Portugal
Department of Computer Science, Faculty of Science, University of Porto, Portugal

Ricardo Queirós
CRACS – INESC-TEC, Porto, Portugal
uniMAD, ESMAD/P.PORTO, Portugal

Marco Primo
Faculty of Sciences, University of Porto, Portugal

Abstract

This paper describes ongoing research to develop a system to automatically generate exercises on document type validation. It aims to support multiple text-based document formalisms, currently including JSON and XML. Validation of JSON documents uses JSON Schema and validation of XML uses both XML Schema and DTD. The exercise generator receives as input a document type and produces two sets of documents: valid and invalid instances. Document types written by students must validate the former and invalidate the latter. Exercises produced by this generator can be automatically accessed in a state-of-the-art assessment system. This paper details the proposed approach and describes the design of the system currently being implemented.

2012 ACM Subject Classification Applied computing → Computer-assisted instruction; Information systems → Information storage systems; Information systems → Web data description languages; Information systems → Extensible Markup Language (XML)

Keywords and phrases exercise generation, automated assessment, document type assessment

Digital Object Identifier 10.4230/OASIcs.SLATE.2022.4

Funding This paper is based on the work done within the Automatic Assessment Of Computing Exercises project supported by the European Union's Erasmus Plus programme (agreement no. 72020-1-ES01-KA226-VET-096004).

1 Introduction

The motivation for this research comes from the JuezLTI, a project that aims at the integration of automated assessment in Learning Management Systems (LMS) using the Learning Tool Interoperability (LTI) specification. Different assessment domains are in development as part of this project, ranging from programming languages to database query languages, including languages for serialization formalisms such as XML and JSON. This paper focuses on the automated assessment of document type definitions using languages such as JSON Schema, XML Schema, and Document Type Definition (DTD).

To support a wide range of domains JuezLTI follows a simple but effective assessment strategy. Exercises are assessed using a set of test cases, each including an input and an expected output. In the case of document type definitions, the inputs are instance documents, each with an expected validation result ("output"), either valid or invalid.

One of the drawbacks of this assessment strategy is the toil of producing good exercises. Each exercise should have a solution and a comprehensive set of tests covering all corner cases. Small and incomplete test sets may accept wrong solutions and preclude the use of automated feedback. The proposed approach is to generate test sets from solutions - document type definitions in this case. For example, from a JSON schema definitions two sets are generated: one with instances, valid JSON documents according to the definition; and another with non-instances, invalid JSON documents according to the definition. Although instance generators are available for document type definition languages, the authors are not aware of non-instance generators. Moreover, both instances and non-instances must be generated for the purpose of automated assessment: with documents as small as possible, covering all corner cases, and providing explanations for what is being tested to be used in automated feedback.

The following sections provide background on this research. Section 2 introduces serialization formalisms, document type definitions, and validation exercises. Section 3 reviews the concepts of automated assessment. Section 4 describes the proposed approach. Section 5 summarizes the current results and describes future work.

2 Serialization formalisms

Text-based document formalisms such as XML and JSON play an important role in software development. They are both human and machine-readable and thus are widely used in tasks that require data serialization. These formalisms prescribe a set of well-formedness rules that documents must follow. For example, in XML documents close tags must end the previous open tag, and attribute values must be delimited by either single or double quotes; in JSON documents property name-value pairs must be separated using commas, and property names must be delimited with double-quotes. This kind of basic rule allows the creation of a wide range of documents for different domains. Nevertheless, on a certain domain, or on specific software, only very specific types of documents are allowed.

```
{
    "name": "Jill Doe",
    "height": 173,
    "hobbies": [
        "skating",
        "reading"
    ]
}
```

(a) Instance describing a person.

```
{
    "type": "object",
    "properties": {
        "name":   { "type": "string" },
        "height": { "type": "number" },
        "hobbies": {
           "type": "array",
           "items": { "type": "string" }
        }
    }
}
```

(b) Schema to validate persons.

Figure 1 Valid JSON instance according to JSON Schema.

Document type definitions describe documents beyond well-formedness rules. The most popular document type definition languages, such as JSON Schema and XML Schema, describe the structure of documents. Documents can be seen as trees and document types in these languages specify the relationships among neighbouring nodes (parent-children and siblings nodes), and the basic types of leaf nodes. For example, Figure 1a presents a JSON document with information on a person, her name, height, and a list of hobbies; and Figure 1b presents a JSON Schema document that accepts an object with properties “name” (a string), “height” (a number) and “hobbies” (an array of strings).

Document types are instrumental to validate document instances. A validator is a program that verifies if a given document is valid according to a given document type. They are typically used when documents are generated or before processing them, to check if they are according to the specification. Validators can also be used to access document types produced as exercises.

To solve an exercise on document types a student must produce a schema in a certain language such as JSON Schema. The exercise statement describes the kind of document and may give examples of a valid document, such as that on the Figure 1a. The schema produced by the student must be equivalent to a reference solution. This assessment can be easily automated using two sets of instances: one with valid documents and another with invalid ones. Using a validator and the schema produced by the student, the documents in the first set must be all valid and those on the second set must be all invalid.

This approach to schemata assessment is comparable to what the traditional black-box model used on programming exercises assessment. In this model, the student's program is executed with a set of test cases. Each test provides an input file that is fed to the program and the obtained output is compared with the test's output file. If they all coincide then the program is considered correct. Due to the complexity of programming languages, test cases can be produced automatically from the solution just for simple cases, using programming properties [2]. But this strategy can be systematically used for document schemata. This paper presents an approach to generate both instances and non-instances of a given schema. Two formalisms are supported - JSON and XML. For the former is used JSON Schema, and for the latter, both XML Schema and DTD are used.

3 Automated assessment

Automated assessment is a widely used approach in many domains, particularly in computer programming [4]. In this context, special tools called automatic judge systems are responsible for grading programming assignments, by comparing the obtained output with the expected output [3]. Most of these tools grade a submission according to a set of rules, following a black-box approach and produce an evaluation report. The validation process has two phases: static analysis, which tests the consistency of the source code; and dynamic analysis, which includes running the program with each test case loaded with the problem and comparing its output to the expected output.

In order to automate the assessment process, programming exercises are assembled in packages with their resources described specialized metadata [5]. Most of the existent metadata formats support only assessment of blank-sheet coding questions. However, the different phases of a student's learning path may demand distinct types of exercises (e.g., bug fix and block sorting) to foster new competencies such as debugging programs and understanding unknown source code or, otherwise, to break the routine and keep engagement. YAPExIL [5] is format to describe programming exercises supporting different types of activities.

Property-based testing can also be used for assessment [2]. Instead of fixed test cases, test data are generated randomly from properties of a functions by a test script. There are several advantages in this approach: 1) it is easier to conduct more tests covering the scope of all possible inputs (thus find more mistakes), 2) in case of failure, shrinking heuristics can be used to automatically simplify failing cases.

A domain closer to this research is web services assessment. Petrova-Anonova [6] proposed an automatic generator of test data for XML Schema-based testing of Web Services. The tool automatically extracts an XML Schema from WSDL or WS-BPEL documents and generates both correct and incorrect XML instances needed for web service interactions. This way, the tool can be used both for testing web services at the functional and robustness levels. Other contributions were made to generate XML documents with data both valid and invalid depending on the type of testing performed. Other works [7, 1] focus on the generation of random values for the invocation parameters providing automatic derivation of data instances from WSDL descriptions.

4 Test case generation

The cornerstone of the proposed approach is the generation of instances and non-instances of a given schema. These documents, the given schema and an exercise statement are then assembled in a programming exercise in the YAPExIL format. Although this strategy is used both for JSON and XML, (non)-instance generation is based on JSON and JSON Schema. XML type definition languages are converted to JSON Schema for (non)-instance generation, and the generated JSON documents are then converted to XML.

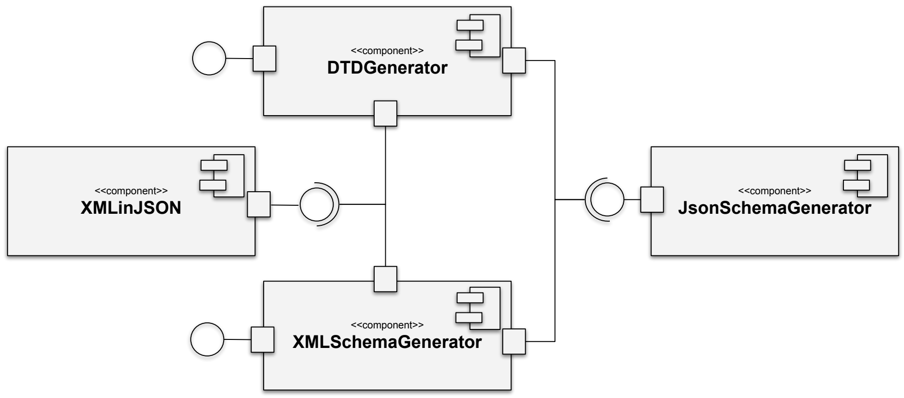

Figure 2 System architecture.

Figure 2 depicts a UML component diagram of the generator's architecture. It has four main components, three of which are generators for document type definition languages. These components have similar interfaces, with methods for generating both instances and non-instances, and validating instances against definitions. The core component is the `JsonShemaGenerator`. The XML type definition languages generators, namely for DTD and XMLSchema, use the previous generator and convert documents between XML into JSON using the `XMLinJSON` component.

The following subsections detail the generation of JSON documents from JSON schema, performed by the `JsonSchemaGenerator`, and conversion between XML and JSON, performed by the `XMLinJSON` component.

4.1 Document generation from JSON Schema

The `JsonSchemaGenerator` component was designed having in mind the creation of test sets for exercises on document type definitions. It generates both instances and non-instances - documents that are invalid according to a given type definition. These two sets of documents are generated having in mind they will be used for automated assessment and feedback. Hence, it produces instances of different sizes, covering different kinds of possible mistakes, and also short hints that can be used as feedback.

To ensure the creation of sets of instances as large as needed, the instance generator produces either an infinite cyclical sequence of instances or an empty set. For example, a schema that validates booleans produces the infinite sequence alternating `true` and `false`; and a schema that validates no instance (such as the schema false) produces the empty set. With this approach, the generation can be tailored to test sets of any size. Instances are generated with increasing size, starting with the smallest possible instances. For example, in a schema of an array, the first generated instance has the smallest size allowed, zero if a minimum size is not defined. This approach improves the quality of the feedback when instances are presented to students to explain the error since smaller examples are easier to understand.

An important feature of the instance generator is the ability to generate non-instances. Two kinds of non-instances are generated: with a different type and of the same type but with invalid constraints. For example, in the case of a schema for an even number, a non-instance is either be a non-number (an instance generated by any type different from the type number) or an odd number (a number with an invalid constraint). The sequence of generated non-instances is varied, as well as incremental in size. In the same example, the first generated non-instance is an odd number (1) - followed by an empty list, an empty object, an empty string, a boolean, etc. The subsequent instances include larger numbers, longer strings, lists with more elements, and objects with more properties.

4.2 Conversion between JSON and XML

The `JsonSchemaGenerator` described in the previous subsection is the core of the document type generator. To handle XML document types these need to be converted into JSON Schema, and JSON (non-)instances converted to back XML. For that purpose, a representation of XML in JSON - or XIJ - was developed, implemented by the `XMLinJSON` component.

The goal of XIJ is to support all XML features in JSON, rather than to produce concise documents. It supports mixed content, i.e. text mixed with annotations, namespaces, and all types of XML nodes, such as comments and processing instructions. The module developed for XIJ includes a parser and a serializer. The overall strategy to convert generate XML instances and non-instances for definitions is as follows. Document type definitions in DTD or XML Schema are converted in JSON Schema definitions. This JSON Schema validates a document in XIJ equivalent to an XML document in the original definition. Then, the JSON Schema generator described in the previous section is used to create sets of (non)-instances, and these are finally converted to XML. However, the creation of a JSON Schema equivalent to a DTD or an XML Schema posed a few challenges.

The features of type definition languages are not equivalent. The most obvious difference relates to basic types: DTDs have only text (`#PCDATA` or `CDATA`), XML Schema has a comprehensive library, and JSON schema has JSON basic types plus integer. A not-so-obvious difference is how repetitions are handled: XML Schema provides fine-grain control over repetitions, DTDs control with regular expression operations, and JSON Schema cannot control repetitions. In fact, JSON Schema uses regular expressions but only for strings, either as values or as property names.

JSON schema had to be extended to make it compatible with DTD and XML Schema. Fortunately, the required extensions are within the scope of JSON Schema itself [8]. The creation of a library of basic types compatible with that XML Schema is straightforward by referencing an external schema document containing those definitions. The support for repetition control was obtained by introducing the `minOccurs` and `maxOccurs` properties in schema definitions, with a syntax and semantics equivalent to the attributes with the same name in XML Schema; these new properties are only effective when schemata are used to validate array items.

5 Current status and future work

The goal of project JuezLTI is to integrate several assessment domains in Learning Management Systems (LMS). One of these assessment domains supports the automated assessment of exercises on type definitions to validate JSON and XML documents. The research presented in this paper proposes an approach to automatically generate exercises from solutions; that is, from type definitions in JSON Schema, for JSON documents, or DTD and XML Schema for XML documents.

The distinctive feature of this approach is the generation of both valid and invalid instances to check students attempts. The core of the system is a document generator for JSON Schema definitions. To handle XML documents, DTDs and XSDs are converted to JSON Schema that validates documents in XIJ, a representation of XML in JSON. The generated documents in this representation are then serialized to XML.

The advantage of the proposed approach is twofold. Firstly, it saves time since it generates test cases directly from solutions. Secondly, the generated tests thoroughly cover all corner cases, unlike those produced manually, resulting in a more effective assessment.

The proposed approach is a work in progress. The core JSON generator is already developed and tested, as well as the representation of XML in JSON. The DTD converter is almost completed and the XML Schema converter is still in the design stage. In the next stage, the set of instances will be packaged as exercises using the YAPeXIL format and stored in a learning object repository.

The results obtained so far with the JSON version are very promising. The generator produces a large number of varied instances of incremental sizes, adapted to the intended use. To validate the proposed system, students will solve generated exercises as part of a Learning Management System activity.

References

1 Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Andrea Polini. Ws-taxi: A wsdl-based testing tool for web services. In *2009 International Conference on Software Testing Verification and Validation*, pages 326–335. IEEE, 2009.

2 Clara Benac Earle, Lars-Åke Fredlund, and John Hughes. Automatic grading of programming exercises using property-based testing. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, pages 47–52, 2016.

3 Katerina Georgouli and Pedro Guerreiro. Incorporating an automatic judge into blended learning programming activities. In *International Conference on Web-Based Learning*, pages 81–90. Springer, 2010.

4 José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. Automated assessment in computer science education: A state-of-the-art review. *ACM Transactions on Computing Education (TOCE)*, 2022.

5 José Carlos Paiva, Ricardo Queirós, José Paulo Leal, and Jakub Swacha. Yet another programming exercises interoperability language (short paper). In *9th Symposium on Languages, Applications and Technologies (SLATE 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

6 Dessislava Petrova-Antonova, Kunka Kuncheva, and Sylvia Ilieva. Automatic generation of test data for xml schema-based testing of web services. In *2015 10th International Joint Conference on Software Technologies (ICSOFT)*, volume 1, pages 1–8. IEEE, 2015.

7 Harry M Sneed and Shihong Huang. The design and use of wsdl-test: a tool for testing web services. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(5):297–314, 2007.

8 Austin Wright, Henry Andrews, and G Dennis. Json schema: A media type for describing json documents. In *IETF, Internet-Draft draft*. IETF - Internet Engineering Task Force, 2020.