Grammar-Driven Development of JSON Processing Applications

Antonio Sarasa-Cabezuelo, José-Luis Sierra
Fac. Informática. Universidad Complutense de Madrid. 28040 Madrid (Spain)
{asarasa,jlsierra}@fdi.ucm.es

Abstract—**This paper describes how to use conventional parser generation tools for the development of JSON processing applications. According to the resulting grammar-driven development approach, JSON processing applications are architected as syntax-directed translators. Thus, the core part of these components can be described in terms of translation schemata and can be automatically generated by using suitable parser generators. It makes it possible to specify critical parts of the application (those interfacing with JSON documents) by using high-level, grammar-oriented descriptions, as well as to promote the separation of JSON processing concerns from other application-specific aspects. In consequence, the production and maintenance of JSON processing applications is facilitated (especially for applications involving JSON documents with intricate nested structures, as well as for applications in which JSON formats are exposed to frequent changes and evolutions in their surface structures). This paper illustrates the approach with JSON-P as the generic JSON processing framework, with ANTLR as the parser generation tool, and with a case study concerning the development of a player for simple man-machine dialogs shaped in terms of JSON documents.**

Keywords—**JSON, Grammar-Driven Development, Translation Schemata, Parser Generator, ANTLR, JSON-P**

I. Introduction

JSON (JavaScript Object Notation) [15][34] is a data exchange format based on a subset of the JavaScript programming language that in recent years has achieved enormous relevance in industry. Indeed, many times JSON results in a more natural mechanism for representing data structures than other alternative formats (e.g., XML [25], which is more suitable for representing hierarchical data). In fact, JSON makes it possible to use collections of name-value pairs and ordered sequences of values, which mirrors the typical data included in mainstream programming languages structures (records, objects or hash tables for collections of name-value pairs, and arrays or lists for ordered sequences of values). This JSON feature makes it natural to map JSON documents to data structures in a target programming language. Still, since JSON is based on text encoding, it is independent from particular programming languages and binary formats; indeed, it can be inspected and, with some effort, interpreted by humans, which facilitates development, debugging and system interconnection tasks. Finally, JSON has also found an important application area as a storage format in non-relational database systems [12][24].

As any other data interchange enabling technology, the success of any development based on JSON relies on finding suitable ways of processing JSON documents in the resulting applications. For this purpose, multiple technologies for processing JSON documents have been proposed, which can be classified into two broad categories:

- *Specific processing technologies*. With these artifacts, it is possible to carry out specific-purpose processing tasks (e.g, querying and document transformation). Examples of these proposals are [5][10] [18]. While these technologies are easy to use, due to their specific and task-oriented nature, the main drawback of this task-specific approach is the need to find suitable specific technologies for each particular processing task.
- *Generic processing technologies*. These technologies make it possible to achieve any processing task. They are provided by libraries and frameworks for JSON manipulation embedded as part of a general-purpose programming language. Examples of these technologies include those that perform marshalling and unmarshalling between JSON documents and data structures [23], and frameworks for parsing JSON documents [4][11][13][14][16][19] [22][29]. In addition, although these technologies can be used to address any processing task, they are substantially more difficult than specific technologies, resulting in higher development and maintenance efforts.

Regardless of their scope of applicability, the aforementioned processing approaches are *data-oriented* in nature, in the sense of conceiving JSON documents as mere data containers, and JSON processing as the mapping of this data into data structures in the host languages. However, since JSON is a formal language, an alternative, language-oriented, approach is possible. This approach will be focused on computer language processing aspects instead of a data marshaling / un-marshaling perspective. In particular, it will be possible to characterize types of JSON documents as *formal grammars*, and then to orchestrate the processing of these documents according to a syntax-directed processing model. Indeed, the characterization of JSON documents as formal grammars is consistent with schema languages like JSON Schema [17]. Thus, the proposed language-oriented (or, more specifically, *grammar-oriented*) approach goes a step further, by conceiving processing tasks of JSON documents being carried out by syntax-directed language processors operating on these

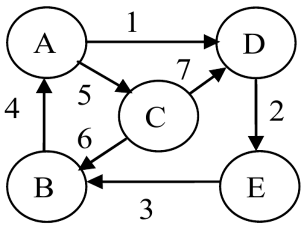
JSON documents. In turn, these language processors can be developed by using dedicated compiler construction tools (parser generators like JavaCC [21], ANTLR [27] or CUP [3], in particular). This approach exhibits the advantages of the variety and stability of these tools, the high level of abstraction to specify the processing (indeed, the approach brings the advantages of task-specific strategies to general-purpose processing settings), greater simplicity in application maintenance, and naturalness for addressing efficient stream-based processing.

This paper describes this grammar-oriented approach to the development of JSON processing applications. The rest of the paper is structured as follows: Section II provides a short introduction to JSON. Section III outlines the grammar-oriented approach. Section IV shows how the approach can be actually implemented by combining a concrete JSON processing framework (JSON-P) with a concrete parser generation tool (ANTLR). Section V illustrates how the approach can be applied to concrete scenarios with the development of a JSON-based application for playing human-computer dialogs. Finally, Section VI provides some conclusions and lines of future work.

II. JSON

As indicated earlier, JSON is a lightweight text-based notation for encoding data structures. Thus, this notation rules how to encode data structures as text entities, known as JSON *documents*. For this purpose, JSON distinguishes among the following kind of data:

(a)



(b)

```
{"nodes":["A","B","C","D","E"],
 "arcs":[
  {"source":"A","target":"D","label":"1"},
  {"source":"A","target":"C","label":"5"},
  {"source":"B","target":"A","label":"4"},
  {"source":"C","target":"B","label":"6"},
  {"source":"C","target":"D","label":"7"},
  {"source":"D","target":"E","label":"2"},
  {"source":"E","target":"B","label":"3"}]}
```

Figure 1. (a) A labelled directed graph, (b) a JSON encoding of the graph in (a).

- *Basic data:* (double precision floating-point) numbers, strings (double-quoted sequences of Unicode characters, with standard scape conventions), Booleans (true and false), and the `null` value.
- *Compound data: arrays* (ordered sequences of comma-separated values, delimited by `[` and `]`), and *objects* (unordered, comma-separated, collections of key-value pairs delimited by `{` and `}`; each key-value pair is in the form *key: value,* where *key* is, in turn, a string).

Using these somewhat simple conventions, JSON makes it possible to represent data structures of arbitrary complexity (it is very similar to what happens with s-expressions in LISP [1], or with XML markup). This flexibility, together with its seamless integration with JavaScript, explains the successful adoption of JSON as an enabling technology for web development, where, for instance, it has become a de facto standard for data exchange in RESTFul service-oriented architectures [30].

Figure 1 illustrates the use of JSON to represent a labeled directed graph with a JSON document. The encoding conventions followed should be apparent from the JSON document itself. It reveals another important feature of JSON: since it is a text-based format, with a little effort it can become understandable to developers. Thus, it facilitates making a good amount of system internals accessible both for humans and machines in terms of JSON documents.

III. The Grammar-Driven approach to the development of JSON applications

In addition to the textual encoding of data structures, JSON documents can be conceived as sentences in a formal language. Indeed, when JSON is used to encode a particular kind of data structure (e.g., labeled directed graphs, as in Figure 1), it is possible to distinguish a subset of JSON documents that meaningfully represents instances of such a data structure. This subset of documents, in turn, can be thought of as defining another formal language: the language of the JSON documents allowable in the particular application domain. Thus, it is possible to apply to JSON similar principles to those used in other analogous fields, like XML (i.e., distinction between *well-formed* and *valid* documents, and characterization of document types with formal grammars). In particular, the use of formal grammars to describe JSON documents in a given application domain acquires full meaning. This paradigmatic bias (i.e., going from a data-oriented perspective to a linguistic, grammar-oriented one) leads to the grammar-driven approach presented in this paper.

The grammar-driven approach can be derived by first considering the structure of a standard syntax-directed translator (Figure 2a). This structure comprises two basic components:

- The *scanner* that is in charge of tokenizing input sentences.
- The *translator,* a *parser* augmented with semantic actions that, when acting on the token sequence produced by the scanner, is able: (i) to recognize this sequence of tokens as belonging to the input language, or otherwise to reject it as invalid, (ii) to arrange it according to its underlying syntactic structure, and, (iii) to process it by firing the semantic actions.

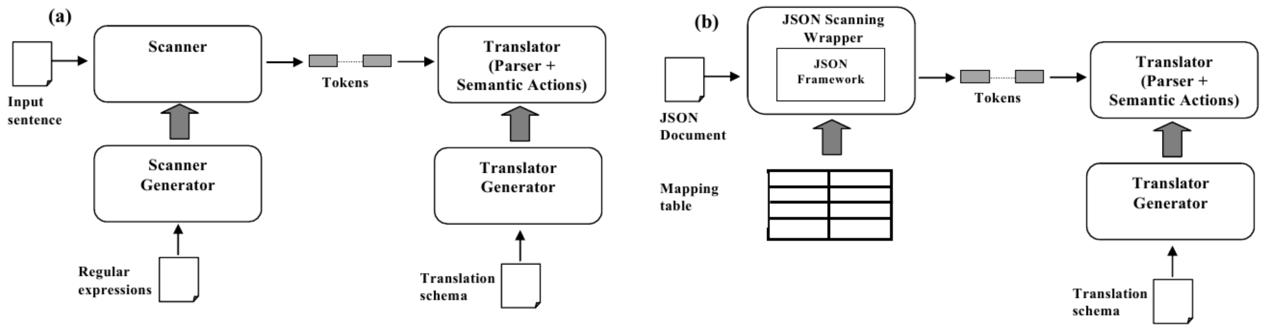


Figure 2. (a) Structure of a Syntax-Directed Translator and the automationaccomplish JSONof its development; (b) modification of the structure despicted in (a) to processing

As is widely acknowledged by the programming language community, this organization results in a processing model especially suited for stream processing, which, under reasonable assumptions, is able to behave in an extremely efficient way [2]. In addition, both the scanner and the translator can be automatically generated from high-level specifications (regular expression-based ones concerning the scanner; *translation schemata* –i.e., context-free grammars augmented with semantic actions, concerning the translator) [2]. Indeed, generation tools like JavaCC, ANTLR or CUP greatly facilitate this development task.

The next step is to adapt classic syntax-directed organization to JSON processing. For this purpose, the scanner in Figure 2a can be replaced by a new component: the *JSON scanning wrapper* (Figure 2b). When operating on JSON documents, this component will map the logical structure of these documents into sequences of tokens, as expected by a syntax-directed translator. It is important to notice that the provision of this component does not rely on the programming of a new generic JSON processor. On the contrary, this component can be meaningfully piggybacked on an existing JSON processing framework (like JSON-simple [19] or JSON-P [16]).

Once this replacement is accomplished, the rest of the organization remains unchanged, as evidenced by Figure 2b. In particular, it is still possible to specify processing (this time of JSON documents) by using high-level translation schemata, and to automatically turn these specifications into efficient implementations by using parser generation tools. Therefore, tools like JavaCC, ANTLR and CUP take a new and unpredicted role, as tools for developing efficient, stream-oriented, JSON processing applications.

Thus, notice that this grammar-driven development approach makes it possible to make up grammar-driven production environments for JSON processing applications by combining a suitable parser generation tool with a general purpose JSON processing framework. The adaptation between the two components will be performed by means of a JSON *scanning wrapper*, which will be dependent on the particular parser generation and processing framework. Beyond this specific component, the approach is nicely independent of the particular parser generator and processing framework chosen.

Concerning the use of this kind of grammar-oriented environments in the actual development of JSON applications, it involves:

- Customizing the JSON *scanning wrapper* to tokenize the logical structure of the JSON documents involved in the application. It can be readily done by providing a *mapping table* associating a distinct token with: (i) each key in each object, (ii) the object opening and closing marks (i.e., { and }), (iii) each possible basic value in the document (i.e., *number, string, true* and *false*). The other structure in the document (e.g., ordered sequences in lists) can be characterized in purely grammatical terms. For instance, Figure 3 despicts the mapping table for the example of labelled graphs in section II.

JSON Element	Token Kind
"nodes"	NODES
"arcs"	ARCS
"source"	SOURCE
"target"	TARGET
"label"	LABEL
{	OO
}	CO
string	STRING

Figure 3. Mapping table for documents like those of Figure 1.

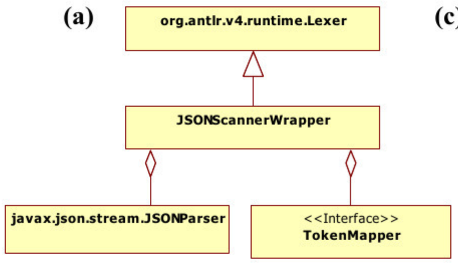
- Characterizing the grammatical structure of the source JSON documents. It can be done by using standard BNF or EBNF notation, augmented with some facilities for describing the structure of objects.

(a)
graph → {nodes: **nodes**, arcs: **arcs**}
nodes → (STRING)*
arcs → (**arc**)*
arc → {source: STRING, target: STRING, label: STRING}

(b)
arc → source: STRING, target: STRING, label: STRING |
source: STRING, label: STRING, target: STRING |
target: STRING, source: STRING, label: STRING |
target: STRING, label: STRING, source: STRING |
label: STRING, source: STRING, target: STRING |
label STRING, target: STRING, source: STRING

Figure 4. (a) Structure of graph-description JSON documents, (b) description of the structure of an arc object using standard EBNF notation.

(a)



(b)

```
public interface TokenMapper {
  int typeForKeyName(String keyName);
  int typeForObjectOpening();
  int typeForObjectClosing();
  int typeForStringValue(String data);
  int typeForNumericValue(String value);
  int typeForTrueValue();
  int typeForFalseValue();
  int typeForNullValue();
}
```

(c)

```
public Token nextToken() {
 try{
  if (parser.hasNext()) {
   JsonParser.Event eventCode=parser.next();
   while (parser.hasNext()&&
          eventCode!=JsonParser.Event.KEY_NAME &&
          eventCode!=JsonParser.Event.START_OBJECT &&
          eventCode!=JsonParser.Event.END_OBJECT &&
          eventCode!=JsonParser.Event.VALUE_NULL &&
          eventCode!=JsonParser.Event.VALUE_STRING &&
          eventCode!=JsonParser.Event.VALUE_NUMBER &&
          eventCode!=JsonParser.Event.VALUE_FALSE &&
          eventCode!=JsonParser.Event.VALUE_TRUE)
          eventCode=parser.next();
   if (eventCode == JsonParser.Event.KEY_NAME) {
       String name=parser.getString();
       JsonLocation location= parser.getLocation();
       return newToken(
                tokenmapper.typeForKeyName(name),
                name, location);}
   else if (eventCode == JsonParser.Event.VALUE_STRING) {
        String data = parser.getString();
        JsonLocation location = parser.getLocation();
        return newToken(
                 tokenmapper.typeForStringValue(data),
                 data, location);
    }
    else ...
```

Figure 5. (a) `JSONScannerWrapper` and its relationships with JSON-P and ANTLR, (b) the `TokenMapper` interface, (c) excerpt of `nextToken in JSONScannerWrapper`

In particular, we propose to describe objects by expressions in the form $\{k_1\, m_1\!: V_1, \ldots, k_n\, m_n\!: V_n\}$, where each k_i is a distinct key name, each V_i is a EBNF expression characterizing the structure of the allowable values for k_i, and each m_i is a modifier controlling the ocurrence of key-value pairs of the kind k_i:v in actual documents. In this expression, the order of appearance of the key-value pairs does not matter. In addition, key-value pairs in the form k_i: v must occur (i) exactly one time if m_i is omitted, (ii) zero or one time if m_i is set to `?` , (iii) zero or more times if m_i is set to `*`, and (iv) one or more times if m_i is set to `+`. For instance, Figure 4a characterizes the grammatical structure of the documents involved in the graph example of the previous section using standard EBNF augmented with this convention.

- Encoding the grammatical structure in the parser generation tool. While in principle it could be possible to translate such a structure to pure EBNF (and thus, to pure BNF) notation(s), the lack of order of key-value pairs in objects can make this direct approach cumbersome, since it could involve enumerating all the possible permutations of key-value sequences (see Figure 4b). Thus, it is possible to use additional semantic facilities in the generator to facilitate such an encoding (e.g., validating semantic actions, semantic predicates …)
- Augmenting the grammar with semantic contexts and semantic actions in order to characterize the processing task as a syntax-directed translation process. The result is a translation scheme, which will be dependent on the parser generator adopted.
- Providing the additional machinery necessary to complete the processing application. Depending on the kind of application, it could include data visualization facilities, database support, a domain model to be instantiated as result of processing the JSON document, etc. In any case, it is interesting to provide a suitable façade in terms of which of the semantic actions in the translation scheme can be written. This façade will be called a *semantic module.*
- Generating the JSON processing component from its specification as a translation scheme. For this purpose, the parser generator is used.
- Gluing it all together in a suitable main program able to launch the application itself.

It is worthwhile to notice that, as a consequence of this grammar-oriented approach, applications are split into two well-differentiated layers:

- A *linguistic layer,* which is declaratively described as a translation scheme expressed in the specification language of the parser generator.
- An *application logic layer,* which is given in terms of conventional software components interfaced by the semantic module.

It leads to an interesting division of labor among developers specialized in JSON processing using formal grammars, and more conventional developers specialized in the development of more conventional application / business logics. The linguistic layer takes care of the orchestration of conventional application logic components, each of which can largely be provided in isolation from the others. In turn, this orchestration is directed by the grammatical structure that underlies the

JSON documents, and it can be described and maintained at a high level, using declarative, grammar-based, specifications, instead of being expressed in a more conventional general-purpose programming language.

IV. Grammar-Driven development with JSON-P and ANTLR

In this section we show how to enable the grammar-driven approach by combining JSON-P and ANTLR:

- JSON-P (Java API for JSON Processing) is a general-purpose JSON processing framework for Java [16]. It defines an API for mapping JSON documents into tree-like representations (the equivalent to DOM in the XML world), and another API to process JSON documents in a streaming fashion.
- ANTLR [27] is a multi-language parser generation tool, which is able to generate recursive descent parsers that combine many of the more recent parsing tendencies: the use of prediction automata for unlimited lookahead (achieved by the LL(*) parsing method), the use of semantic predicates, and the use of backtracking and tabulation to mimic *packrat* parsing [8] (see [28] for a more in-depth description of ANTLR internals). This combination of parsing technologies, together with their support for multiple implementation languages (among them, Java) makes this tool one of the more widely used worldwide.

Concerning JSON-P, this combination uses its facilities for JSON streaming processing. In particular, JSON-P provides a *pull* API similar to StAX in the XML world, which is especially well suited for its combination with ANTLR-generated parsers, since it can naturally work as a scanner for such a parser. In this way, the JSON *Scanning Wrapper* in this combination (see Figure 5a):

- Encloses a `JSONParser` instance (i.e., an instance of the *pull* streaming processing artifact provided by JSON-P)
- Can be customized by an instance of a suitable `TokenMapper` implementation, which actually characterizes the mapping tables (Figure 5b)
- Extends the ANTLR `Lexer` class. In particular, it implements the `nextToken` method to return ANTLR `CommonToken` instances representing the tokens associated with the JSON logical elements. Figure 5c shows an excerpt of this method in our combination.

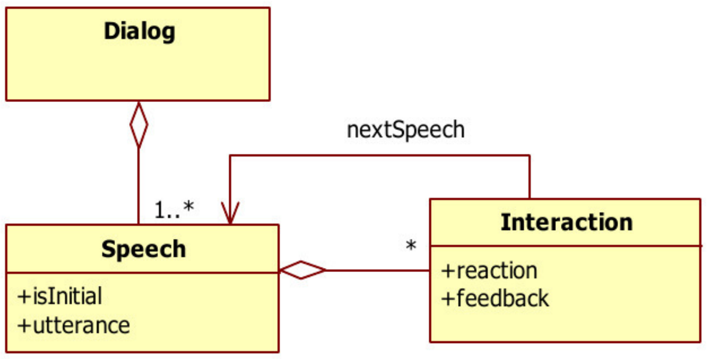
```
arc locals [boolean sourceOn=false,
            boolean targetOn=false,
            boolean labelOn=false]:
  OC ( ({$sourceOn}? SOURCE STRING {$sourceOn=true;}) |
       ({$targetOn}? TARGET STRING {$targetOn=true;}) |
     ({$labelOn}? LABEL STRING {$labelOn=true;}) )* CC
   {$sourceOn && $targetOn && $labelOn}? ;
```

Figure 6. ANTLR encoding of the rule for `arc` in Figure 4a.

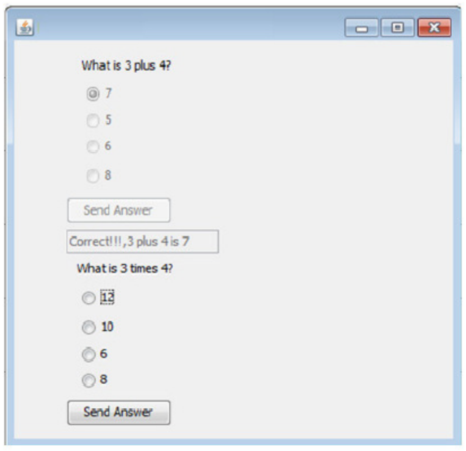
In this way, in the resulting environment:

- The customization of the JSON *Scanning Wrapper* involves: (i) providing a suitable implementation of the `TokenMapper` interface (each method in this interface determines how to map relevant JSON elements into types for ANTLR tokens), and (ii) specializing `JSONScannerWrapper` to use such an implementation as a customization table.
- The encoding of the grammatical structure can take advantage of ANLTR validating semantic predicates to simplify the description of object expressions in the augmented EBNF notation. Indeed, $\{k_1\ m_1\!: V_1, \ldots, k_n\ m_n\!: V_n\}$ is represented by $OC\ ((vp_1\ K_1\ V_1\ a_1) \mid \ldots \mid (vp_n\ K_n\ V_n\ a_n))^*\ CC\ v_f$ where: (i) K_i is the type of token corresponding to k_i, (ii) a_i is a semantic predicate registering the number of times that k_i has occurred, (iii) vp_i is a semantic predicate that validates whether K_i can occur, (iv) v_f is a semantic predicate validating whether all the mandatory key-value pairs have appeared, and (v) OC and CC are respectively the object opening and closing tokens. Figure 6 pro-

(a)



(b)


What is 3 plus 4?

7
5
6
8

Send Answer

Correct!!,3 plus 4 is 7

What is 3 times 4?

12
10
6
8

Send Answer

Figure 7. (a) Dialog semantic model;(b) Snapshot for the Dialog player.

vides an example concerning the EBNF structure provided in Figure 4a.

- Then, additional semantic context and semantic actions can be added to the resulting ANLTR grammar to configure the specification of the JSON processing component as a translation scheme. Once this translation scheme is processed with the ANTLR tool to yield the Java implementation, the corresponding *Lexer* class must be replaced by the customization of the JSON Processing Wrapper, in order to make the parser actually operate on the logical structure of the JSON documents.

V. Case Study: Grammar-Driven Development of a JSON-customizable Interactive Application

This section describes how we have applied the grammar-driven development model defined in the previous sections to the implementation of an interactive application oriented to play dialogs between the user and the computer[1]. These dialogs can be described using JSON documents. Subsection V.A describes how the interactive application behaves and how the dialog documents are structured. Then, subsection V.B details the grammar-driven development of this application.

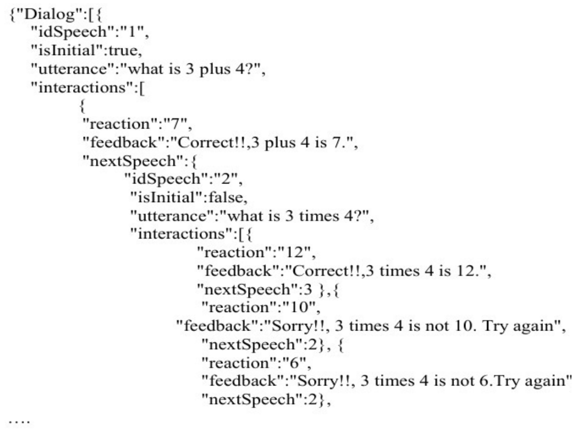
A. *The dialog player*

The dialogs played by our application are based on the *Socratic Tutorials* developed by Prof. Alfred Bork's team during the eighties of the past century [6]. They obey the semantic model in Figure 7a. Thus, when playing a dialog (Figure 7b):

- The application proffers a *speech*. It is a chunk of text that can be read by the user.
- Then it displays a repertory of possible *interactions* and lets the user select one of them.
- When the user selects the interaction, the application gives him/her an associated *feedback.*
- Finally, it either continues with other speeches or ends the execution.

The application can be customized with dialogs described as JSON documents. Indeed, these documents are a possible concrete syntax for the semantic model depicted in Figure 7a. In this way:

- Dialogs are represented by objects with a "`Dialog`" key. The value of this key is a sequence of speeches.
- Each speech is, in turn, represented by an object that includes: (i) a mandatory "`idSpeech`" key, whose value identifies the speech in the JSON document (this value will be used for referencing the speech from other places in the document), (ii) an optional "`isInitial`" key, whose value, when `true`, indicates the speech starts the dialog, (iii) a mandatory "`utterance`" field that includes the text to be proffered by the machine, and (iv) an optional "`interactions`" field containing a sequence with all the possible interactions (if a speech without interactions is reached, the player ends the execution).
- Finally, interactions are represented by objects including: (i) a mandatory "`reaction`" key representing the actual text of the interaction, (ii) a mandatory "`feedback`" key representing the text of the feedback, and (iii) a mandatory "`nextSpeech`" representing the speech that continues the dialog; its value can be either another speech description, or the *id* of other speech in the dialog.



```
{"Dialog":[{
   "idSpeech":"1",
   "isInitial":true,
   "utterance":"what is 3 plus 4?",
   "interactions":[
          {
          "reaction":"7",
          "feedback":"Correct!!,3 plus 4 is 7.",
          "nextSpeech":{
                "idSpeech":"2",
                 "isInitial":false,
                 "utterance":"what is 3 times 4?",
                 "interactions":[{
                         "reaction":"12",
                         "feedback":"Correct!!,3 times 4 is 12.",
                         "nextSpeech":3 },{
                          "reaction":"10",
                      "feedback":"Sorry!!, 3 times 4 is not 10. Try again",
                          "nextSpeech":2}, {
                          "reaction":"6",
                          "feedback":"Sorry!!, 3 times 4 is not 6.Try again"
                          "nextSpeech":2},
....
```

Figure 8. Excerpt of a Dialog Description JSON Document.

Figure 8 shows an excerpt of JSON document describing a dialog.

B. *Grammar-Driven Development*

The operation of the dialog player described in the previous subsection is as follows:

- It processes the input JSON document in order to instantiate the semantic model in Figure 7a.
- Then it plays the dialog by a direct interpretation of the semantic model instance.

While the interpretation stage is straightforward once the semantic model has been instantiated, the instantiation process is considerably more cumbersome, due in part to the changing and evolving nature of the concrete JSON encoding. Thus, the player can be meaningfully architected according to the grammar-driven approach as follows:

- The instantiation of the semantic model is developed in grammatical terms, using JSON-P and ANTLR.
- The semantic module is implemented as a façade class providing instantiation operations as methods, as well as a couple of tables required during the instantiation process: one table mapping *ids* into `Speech` instances, and another table mapping ids into `Interation` instances whose next speeches are those associated to such *ids.*
- The semantic model itself, along with the player shell, constitute the application-specific logic.

1It is actually a simplification of the real system in order to fit the space constraints of the paper. The actual system is closer to that described in [33]

(a)

```
public class DialogTokenMapper implements TokenMapper {
  public int typeForKeyName(String keyName) {
  if (keyName.equals("idSpeech"))
     return DialogParser.IDSPEECH;
  if (keyName.equals("isInitial"))
     return DialogParser.IS_INITIAL;
  ...
  }
  ...
 }
```

(b)

```
public class DialogLexer extends JSONScannerWrapper {
  public DialogLexer(CharStream in) {
       super(in,new DialogTokenMapper());
  }
}
```

(c)

```
grammar Dialog;
@parser::header {import java.util.*;}
@parser::members {private DialogSemM sem =
                                  new DialogSemM();}
dialog returns [Dialog d]:
  {List<Speech> speeches = sem.newSList();}
  (s=speech {speeches.add($s.speechvalue);} )+
  {$d = sem.mkDialog(speeches);} ;

speech returns [Speech speechvalue]
  locals[boolean id=false, boolean isInitial = false,
         boolean ut=false, boolean inter=false]:
  OO (
    ({!$id}? IDSPEECH  sid=STRING {$id=true;} )        |
    ( {!$isInitial}? IS_INITIAL bool {$isInitial=true;} ) |
    ( {!$ut}? UTTERANCE sut=STRING {$ut=true;})           |
    ( {!$inter}? INTERACTIONS interactions {$inter=true;} )
  )* CO
  {$id && $ut}?
  {$speechvalue = sem.newSpeech($sid.text, $isInitial?$bool.b:false,
                    $sut.text, ($inter)? $interactions.inters: null); } ;

interactions returns [List<Interaction> inters] :
   {$inters = sem.newIList(); }
   (i=interaction {$inters.add($i.inter);})* ;

interaction returns [Interaction inter]
   locals[boolean reaction=false, boolean feedback=false,
          boolean ns = false]:
   {Speech sp=null;}
   OO (
       ({!$reaction}? REACTION r=STRING {$reaction=true;}) |
       ({!$feedback}? FEEDBACK f=STRING {$feedback=true;}) |
       ({!$ns}? NEXT_SPEECH (s = speech {sp=$s.speechvalue;}|
                             ref=STRING
                             {sp = sem.getSpeech($ref.text);})
        {$ns=true;})
   )* CO
   {$reaction && $feedback && $ns}?
   {$inter = sem.newInteraction($r.text,$f.text,sp);
    if (sp == null)
       sem.toBePatched($ref.text,$inter);} ;
bool returns [boolean b] : TRUE {$b=true;} | FALSE {$b=false;};
   // Lexical rules will be not used, but they are necessary
   // for completing the ANTLR grammar
IDSPEECH : 'IDSPEECH';
IS_INITIAL : 'ISINITIAL';
...
```

(d)

```
public class DialogSemM {
  private Map<String,Speech> speeches;
  private Map<String,List<Interaction>> patchMap;

  public DialogSemM() {
    speeches = new HashMap<>();
    patchMap = new HashMap<>();
 }

  public Dialog mkDialog(List<Speech> speeches) {
   return new Dialog(speeches);
  }
  public List<Speech> newSList() {
    return new LinkedList<Speech>();
  }
  public List<Interaction> newIList() {
    return new LinkedList<Interaction>();
  }
  public Speech newSpeech(String id, boolean isInitial,
                          String ut,List<Interaction> is)
    Speech sp = new Speech(isInitial,ut,is);
    speeches.put(id,sp);
    List<Interaction> pl = patchMap.get(id);
    if (pl != null) {
      for(Interaction i: pl)
          i.putNextSpeech(sp);
      patchMap.put(id,null);
    }
    return sp;
  }
  public Speech getSpeech(String sid) {
   return speeches.get(sid);
  }
  public Interaction newInteraction(String r, String f,
                                    Speech ns) {
     return new Interaction(r,f,ns);
  }
  public void toBePatched(String si,Interaction i) {
    List<Interaction> pl = patchMap.get(i);
    if (pl == null) {
     pl = new LinkedList<Interaction>();
     patchMap.put(si,pl);
    }
    pl.add(i);
  }
}
```

Figure 9. (a) Implementation of the mapping table for the Dialog case-study; (b) specialization of the JSONScannerWrapper ; (c) ANTLR grammar for the processing of JSON Dialog Documents; (d) semantic module.

- The main gluing program performs the instantiation, and then activates the player with the resulting semantic model instance.

Thus, the organization is very similar to that of applications built using DSL construction frameworks such as Eclipse XText [7] (in this case, input descriptions are encoded in JSON instead on a domain-specific syntax, however).

Concerning development details, Figure 9a shows an excerpt of the token mapping table. As indicated in Section IV, it involves implementing the `TokenMapper` interface, as made apparent in Figure 9a. Notice that, in this implementation, actual token codes are taken from the `DialogParser` class. This will be the parser class generated by ANTLR. Therefore, token names must be kept consistent throughout this mapping table and the subsequent ANTLR grammar.

Once the mapping table is available, it is possible to customize the JSON Scanner Wrapper. As indicated in Section IV it involves to subclass `JSONScannerWrapper` in order to install an instance of the mapping table provided (Figure 9b). The name given to this subclass must be consistent with the name of the lexer to be generated by ANTLR.

Next step, the most relevant one, is to characterize the syntactic structure of the JSON documents, then to encode this structure as an ANTLR grammar following the patterns given in Section IV, and finally to augment this grammar with suitable semantic actions. Figure 9c shows the resulting ANTLR translation scheme.

Then the semantic class that implements the semantic module can be provided (Figure 9d). In this class, in addition to creating a new speech, the `newSpeech` method back-patches all the interactions referring to such a speech, which is consistent with the usage of the operations in the ANTLR grammar.

Next step is to generate all the parsing code from the ANTLR grammar, and to replace the `DialogLexer` generated by that shown in Figure 9b. Finally, the application-specific logic and the main launching program must be provided, which constitutes a routine programming task.

VI. Conclusions And Future Work

In this paper, we have shown how to combine generic, stream-oriented, JSON processing frameworks with parser generators in order to facilitate the development of JSON processing applications. The resulting approach is aware of the grammatical nature of JSON documents and enables the specification of JSON processing tasks at a higher and more declarative level than that provided by general-purpose programming languages. Contrary to proposals like [9], formal grammars in our proposal operate on the logical structure of JSON documents instead of on the raw text of these documents. In this sense, our proposal is aligned with our previous works in XML processing [31][32], in which we proposed similar grammar-driven models for processing XML using grammars and parser generators.

Currently we are working on the implementation of an environment for providing more assistance to our grammar-driven development process model. We also are planning to use attribute grammars [20][26] as specification mechanisms of JSON processing tasks, paralleling our previous work in the XML world [31]. Finally, and although our firsts tests with developers are satisfactory, we plan to carry out a more systematic comparative study of our approach with more conventional approaches to JSON processing.

Acknowledgment

This work was partially supported by the project grant TIN2010-21288-C02-01.

References

[1] Abelson, H., Sussman, G.J. (1993). *Structure and Interpretation of Computer Programs*. MIT Press

[2] Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D. Compilers: principles, techniques and tools (2nd ed.). Addison-Wesley. 2007

[3] Appel, A.W. Modern Compiler Implementation in Java (2002). Cambridge University Press

[4] Berg, J. (2012). Utvärdering av bibliotek för generering och "parsning" av JSON. Degree Dissertation. KTH

[5] Beyer, K. S., Ercegovac, V., Gemulla, R., Balmin, A., Eltabakh, M., Kanne, C. C., ... & Shekita, E. J. (2011). Jaql: A scripting language for large scale semistructured data analysis. In Proceedings of 37th VLDB Conference.

[6] Bork, A. 1985. Personal Computers for Education. New York, NY, USA: Harper & Row Publishers, Inc.

[7] Eysholdt, M., Behrens, H (2010). Xtext: implement your language faster than the quick and dirty way. ACM international conference on Object Oriented Programming Systems Languages and Applications Companion (SPLASH '10), 307-309.

[8] Ford, B (2002). Packrat Parsing : Simple, Powerful, Lazy, Linear Time, Functional Pearl. 17th ACM SIGPLAN international conference on Functional programming, pp. 36-47.

[9] Gerasika, A (2011). How to convert JSON to XML using ANTLR. http://www.gerixsoft.com/blog/xslt/json2xml2 (last access: April 11, 2013)

[10] Goessner, S (2007). JSONPath – XPath for JSON. http://goessner.net/articles/JsonPath/ (last access : April 11, 2013)

[11] Gson. Google-gson - A Java library to convert JSON to Java objects and vice-versa. https://code.google.com/p/google-gson/ (last access: April 11, 2013)

[12] Han, J., Haihong, E., Le, G., & Du, J. (2011, October). Survey on NoSQL database. 6th international conference on Pervasive computing and applications (ICPCA'11), pp. 363-366.

[13] iJSON. https://pypi.python.org/pypi/ijson/ (last access: April 11, 2013)

[14] Jackson. http://jackson.codehaus.org/ (last access: April 11, 2013)

[15] JSON. http://www.json.org/ (last access: April 11, 2013)

[16] JSON-P. Java API for JSON Processing (JSON-P). http://json-processing-spec.java.net/ (last access: April 11, 2013)

[17] JSON-Schema. http://json-schema.org/ (last access: April 11, 2013)

[18] JSONSelect . http://jsonselect.org/ (last access : April 11, 2013)

[19] Json-simple. Json-simple – A simple Java Toolkit for JSON. https://code.google.com/p/json-simple/ (last access: April 11, 2013)

[20] Knuth, D. E. Semantics of Context-free Languages. Mathematical System Theory 2(2), 127–145. 1968.

[21] Kodaganallur, V (2004). Incorporating language processing into Java applications: a JavaCC tutorial. IEEE Software 21(4), 70-77.

[22] Litjson. http://lbv.github.io/litjson/ (last access : April 11, 2013)

[23] Maeda, K. (2012). Performance evaluation of object serialization libraries in XML, JSON and binary formats. 2nd Conference on Digital Information and Communication Technology and its Applications (DICTAP), pp. 177-182.

[24] Membrey, P., Plugge, E., & Hawkins, T. (2010). The definitive guide to MongoDB: the noSQL database for cloud and desktop computing. Apress.

[25] Nurseitov, N., Paulson, M., Reynolds, R., & Izurieta, C. (2009). Comparison of JSON and XML data interchange formats: A case study. Computer Applications in Industry and Engineering (CAINE), 157-162.

[26] Paakki, J. Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation. ACM Computing Surveys, 27, 2, 196-255. 1995

[27] Parr, T (2007). The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Bookshelf.

[28] Parr, T., Fisher, K (2011). LL(*): the Foundation of the ANTLR Parser Generator. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11), pp. 425-436.

[29] Rapidjson. Rapidjson - A fast JSON parser/generator for C++ with both SAX/DOM style API. https://code.go ogle.com/ p/rapidjson/ (last access: April 11, 2013)

[30] Richardson, L., & Ruby, S. (2007). RESTful Web Services. O'Reilly.

[31] Sarasa-Cabezuelo, A., Sierra, J.L. (2013). The grammatical approach: A syntax-directed declarative specification method for XML processing tasks. Comp. Stand. & Interfaces 35(1), 114-131

[32] Sarasa-Cabezuelo, A., Temprado-Battad, B., Rodríguez-Cerezo, D., Sierra, J. L. (2012). Building XML-driven application generators with compiler construction tools. Computer Science and Information Systems, 9(2), 485-504.

[33] Sierra, J.L., Fernández-Valmayor, A., Fernández-Manjón, B (2008). From Documents to Applications Using Markup Languages. IEEE Software, 25(2), 68-76

[34] Zakas, Z. N (2012). Professional JavaScript for Web Developers 3rd Edition. Wrox Press.