DOI: 10.2298/CSIS110505002S

Building XML-Driven Application Generators with Compiler Construction Tools

Antonio Sarasa-Cabezuelo[1], Bryan Temprado-Battad[1], Daniel Rodríguez-Cerezo[1], José-Luis Sierra[1]

[1] Computer Science School,
Complutense University of Madrid
Calle Profesor José García Santesmases, s/n
28040 Madrid, Spain
{asarasa, bryan, drcerezo, jlsierra}@fdi.ucm.es

Abstract. This paper describes how to use conventional compiler construction tools, and parser generators in particular, to build XML-driven application generators. In our approach, the document interface is provided by a standard stream-oriented XML processing framework (e.g., SAX or StAX). This framework is used to program a generic, customizable XML scanner that transforms documents into streams of suitable tokens (opening and closing tags, character data, etc.). The next step is to characterize the syntactic structure of these streams in terms of generation-specific context-free grammars. By adding suitable semantic attributes and semantic actions to these grammars, developers obtain generation-oriented translation schemes: high-level specifications of the generation tasks. These specifications are then turned into working application generators by using standard parser generation technology. We illustrate the approach with <e-Subway>, an XML-driven generator of shortest-route search applications in subway networks.

Keywords: Application Generators, Compiler Construction Tools, XML Processing, Software Development Approach

1. Introduction

Application generators and generative approaches to software development are keystone technologies in enhancing productivity and ensuring the quality of final software artifacts [5][7][9]. In application generators, XML is frequently chosen as a basic encoding format for input specifications [6]. Thus, having cost-effective and efficient methods for processing XML documents is mandatory in these scenarios. For this purpose, architects of application generators have a wide range of XML-processing technologies available, ranging from task-specific (e.g., XSLT) to general-purpose ones (e.g., SAX or DOM). General-purpose XML processing frameworks (i.e., SAX, DOM, StAX, etc) [15] are particularly relevant for very specific or complex processing tasks not easily accomplished with pre-existing task-specific technology.

However, general-purpose processing frameworks are largely data-centric: they see XML documents as chunks of data. By contrast, the intrinsic nature of descriptive markup and XML is fundamentally language-oriented: to design an XML format for a particular type of document is equivalent to devising a suitable domain-specific markup language. It immediately raises an obvious question: if XML documents are structured with (formal) markup languages, why not use conventional language-processing techniques to support the processing of these documents?

The answer to this question depends on the complexity of the markup language and the processing tasks. For simple XML documents (e.g., a sequence of logs with a description and a timestamp) and simple processing tasks (e.g., producing an HTML table with the logs), the effort of designing and implementing the processing component as if it were a sort of compiler, using methods and techniques specific to the compiler construction field, may be excessive. However, for more complex documents (e.g., QTI documents describing assessments in an e-Learning system [11]) and more complex processing tasks (e.g., configuring assessment systems with the QTI documents), this effort can pay off. Actually, the latter constitute the kind of scenarios faced by developers of application generators.

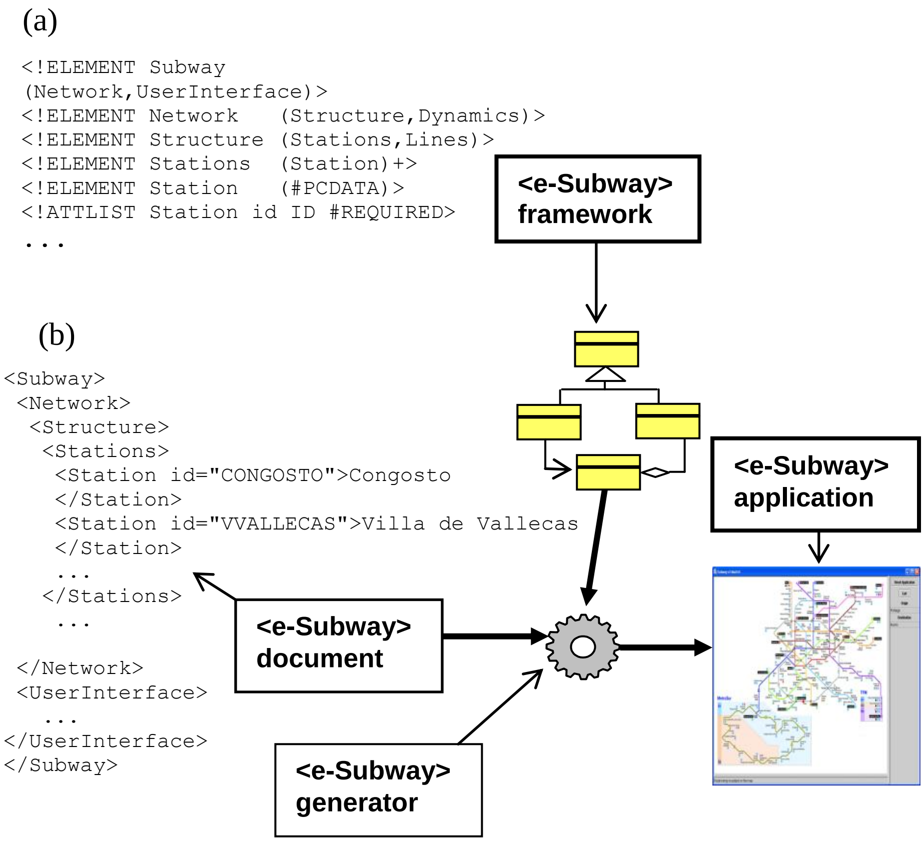
An attractive feature of the language-oriented approach is that the design and implementation of language processors (and, in particular, of translators) is mature enough to support a wide range of tools able to produce reliable and efficient implementations from high-level specifications. Of those tools, the most widely known are parser generators (i.e., YACC-like tools) [1]. These tools accept translation schemes, i.e., context-free grammars annotated with the semantic actions that actually perform the processing, as input, and produce working translators as output. Thus, by using one of these tools, it is possible to drastically reduce the development effort compared to a handcrafted implementation.

This paper shows how it is possible to build sophisticated XML processing environments by combining parser generators with general-purpose stream-oriented XML processing frameworks. For this purpose, it develops a general method that can be used with a great variety of parser generation environments or underlying XML processing frameworks. The result is a systematic approach to the language-oriented development of complex syntax-directed XML processing components, which is especially well-suited to the development of XML-driven application generators.

The rest of the paper is organized as follows: section 2 introduces <e-Subway>, the system that will be used for illustrative purposes. Section 3 outlines the approach and illustrates it with <e-Subway>. Section 4 presents some work related to ours. Finally, section 5 presents some conclusions and lines of future work.

2. Case study

The system <e-Subway> is an XML-based system for the construction of shortest-route search applications in subway networks. This system was already used as a case study in some of our previous experiences concerning the generation of applications from structured documents [38][39]. <e-Subway> integrates:



(a)

```
<!ELEMENT Subway
(Network,UserInterface)>
<!ELEMENT Network   (Structure,Dynamics)>
<!ELEMENT Structure (Stations,Lines)>
<!ELEMENT Stations  (Station)+>
<!ELEMENT Station   (#PCDATA)>
<!ATTLIST Station id ID #REQUIRED>
...
```

(b)

```
<Subway>
 <Network>
  <Structure>
   <Stations>
    <Station id="CONGOSTO">Congosto
    </Station>
    <Station id="VVALLECAS">Villa de Vallecas
    </Station>
    ...
   </Stations>
    ...

 </Network>
 <UserInterface>
   ...
</UserInterface>
</Subway>
```

Figure 1. (a) Excerpt of the <e-Subway> DTD; (b) The <e-Subway> generation process

- An XML-compliant markup language for structuring documents that describe the different aspects of route searching applications (e.g., stations, lines, connections and other aspects of the subway network, as well as selected aspects of the final application's user interface). In Fig. 1a, we outline a fragment of the DTD for this language.
- A domain-specific object-oriented framework. Applications in <e-Subway> are instantiations of this framework.

– A *generator*. This component processes documents that describe <e-Subway> applications and produces the documented applications as instantiations of the <e-Subway> framework (Fig. 1b) (i.e., it does not actually generate code, but produces in-memory instances –objects– of the <e-Subway> framework's classes, and establishes appropriate links between these instances).

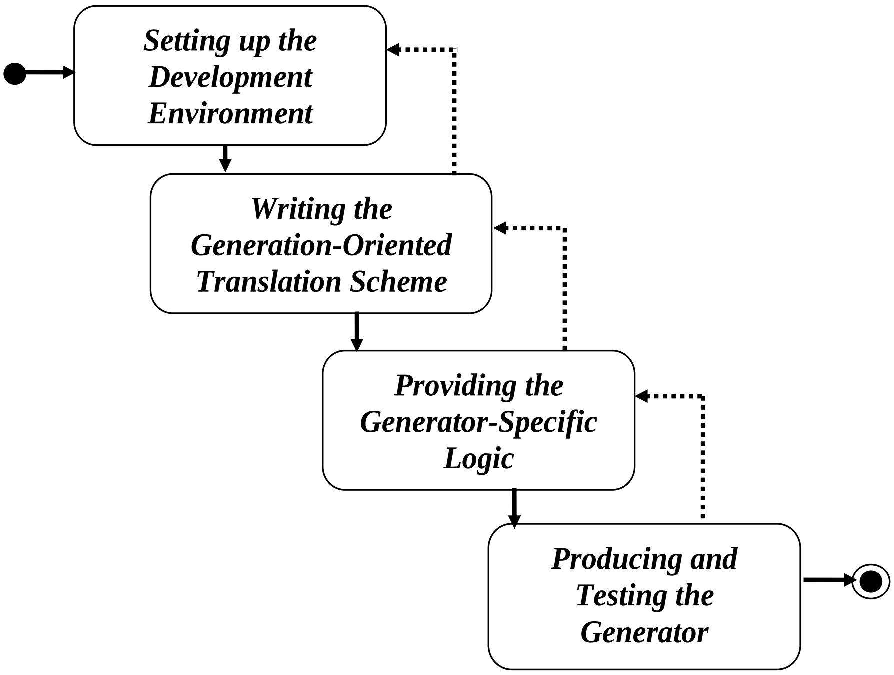

Setting up the Development Environment
Writing the Generation-Oriented Translation Scheme
Providing the Generator-Specific Logic
Producing and Testing the Generator

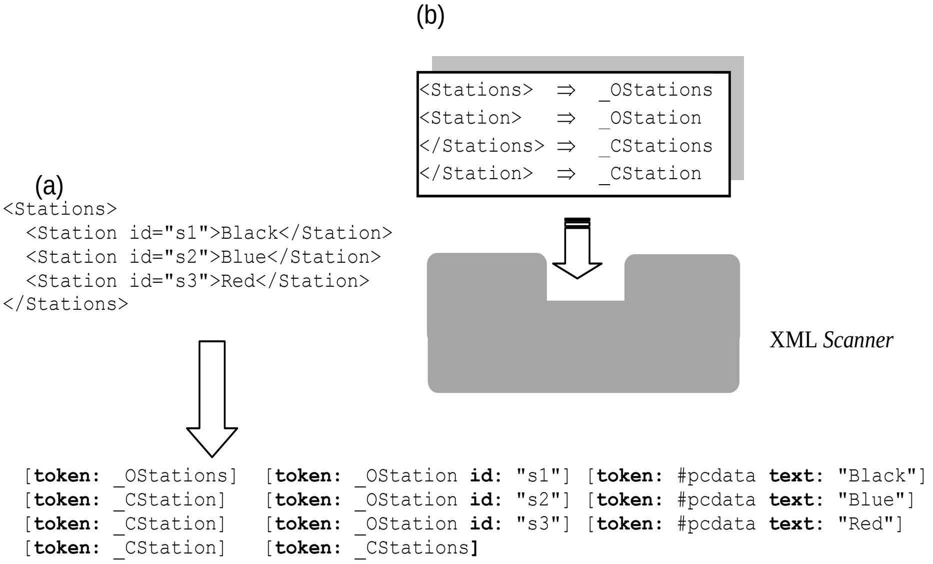
Figure 2. Activities and sequencing of activities in the development approach

3. The development approach

Fig. 2 summarizes the approach to developing XML-driven application generators with conventional parser generation tools, focusing on the main activities and on the sequencing of these activities (the backwards transitions allow an iterative/incremental production process). Notice that this workflow largely mirrors that which is usually followed by any compiler developer. Indeed, he/she must provide a suitable grammar for the source language, add semantic actions to this grammar to yield a translation scheme, generate the translator either by hand or by using a suitable generation tool, etc. This parallelism makes the language-oriented nature of the proposal described in this paper apparent. Nevertheless, it is important to notice that the goal is not to provide a full translator from scratch, but instead to put an additional language processing layer on top of an existing stream-oriented XML processing framework. In particular, the processor will operate on XML information elements (e.g., represented in the form of SAX events) instead of individual characters. As a result, it will lead to the organizing of the application-specific logic attached to a general processing framework into two well-differentiated tiers: one that operates as a syntax-directed translator, and another that provides services to this translator. The following subsections analyze each activity in this workflow.

3.1. Setting up the development environment

This activity integrates a parser generation tool with a general-purpose XML processing framework. This activity will be performed only sporadically, since the same development environment can be used in the development of many different application generators.



```
<Stations>  ⇒  _OStations
<Station>   ⇒  _OStation
</Stations> ⇒  _CStations
</Station>  ⇒  _CStation
```

```
<Stations>
  <Station id="s1">Black</Station>
  <Station id="s2">Blue</Station>
  <Station id="s3">Red</Station>
</Stations>
```

```
[token: _OStations]  [token: _OStation id: "s1"] [token: #pcdata text: "Black"]
[token: _CStation]   [token: _OStation id: "s2"] [token: #pcdata text: "Blue"]
[token: _CStation]   [token: _OStation id: "s3"] [token: #pcdata text: "Red"]
[token: _CStation]   [token: _CStations]
```

Figure 3. (a) Example of tokenization; (b) Customization of an XML Scanner

As previously stated, a parser generation tool produces translators for formal languages from high-level specifications. These translators are driven by parsers that operate on streams of tokens provided by lexical analyzers. Thus, the key idea behind integration is to see XML documents as streams of tokens. Integration itself is focused on the logical structure: streams of tokens are produced by remapping the data structures provided by the general-purposes processing frameworks, instead of by directly operating on the actual XML files. The integration distinguishes four different lexical categories, or kinds of tokens:

- *Character data tokens*, which correspond to fragments of textual content in the processed documents.
- *Opening* and *closing* tags.
- The *end of document*.

In addition to its lexical category, each token can include additional lexical information in the form of lexical attributes:

- *Character data tokens* have the actual textual content associated with them.
- *Opening* tags have the element attributes specified in the tag, as well as namespace information, associated with them.

Fig. 3a shows an example of tokenization.

Based on these considerations, integration provides a generic and customizable XML Scanner by using the selected XML processing framework. This component can be generic, since it is only needed to indicate how to map opening and closing tags into lexical categories (e.g., by using a table, as suggested in Fig. 3b). Also, this kind of integration can be successfully carried out by using a stream-oriented framework such as SAX or StAX. Indeed, the action of the XML Scanner can be conceived of as the transformation of a stream of documental information items into a stream of tokens, as expected by the generated translators.

Concerning the technical details, since generated translators are push components (i.e., they take control, requesting tokens from the scanners when required), integration is particularly straightforward with a pull XML processing framework (e.g., StAX), since these frameworks provide each next information item on demand. On the other hand, integration with a push framework (e.g., SAX) requires inverting control (e.g., using a producer-consumer multithreaded solution). In our previous papers [33] and [34], we give examples of the two kinds of integration.

Finally, it is important to highlight the difference between the XML Scanner proposed in this section and the scanner of a conventional language processor. Indeed, the XML Scanner proposed in our approach is built on top of a full-flagged stream-oriented XML processing framework, able to support features that are common to any XML-based markup language (e.g., support for different character sets and encodings, comment recognition, entity and namespace management, etc.). On the other hand, the scanner of a conventional language processor usually works on text files or stream of characters. Therefore, although it could be possible to provide a conventional scanner for tokenizing a particular type of XML documents, it would have to deal with the aforementioned features to be fully XML-compliant. The complexity of exiting XML parsers teaches us that it is not exactly an easy task. It makes the difference between our proposal and the conventional development of a language processor apparent: if we develop a language processor for a particular type of XML documents following the standard patterns explained in any university-level compiler construction course (see for instance [1]), we will probably get a program able to process input text files with an XML syntax-like, but not a program able to deal with the features common to all XML applications (e.g., the ability to split a huge XML document in several files and to assemble these fragments using the XML entity mechanism, to deal with different character sets, to deal with namespaces, etc.).

3.2. Writing the Generation-Oriented Translation Scheme

This is the central activity of our development approach. Its purposes are to:

– Write a suitable *generation-specific* grammar that gives structure to the stream of tokens provided by the XML Scanner*.*
– Annotate this grammar with code (*semantic actions*) to describe the generation task. The result is the syntax-directed, *generation-oriented translation scheme* produced by this activity.

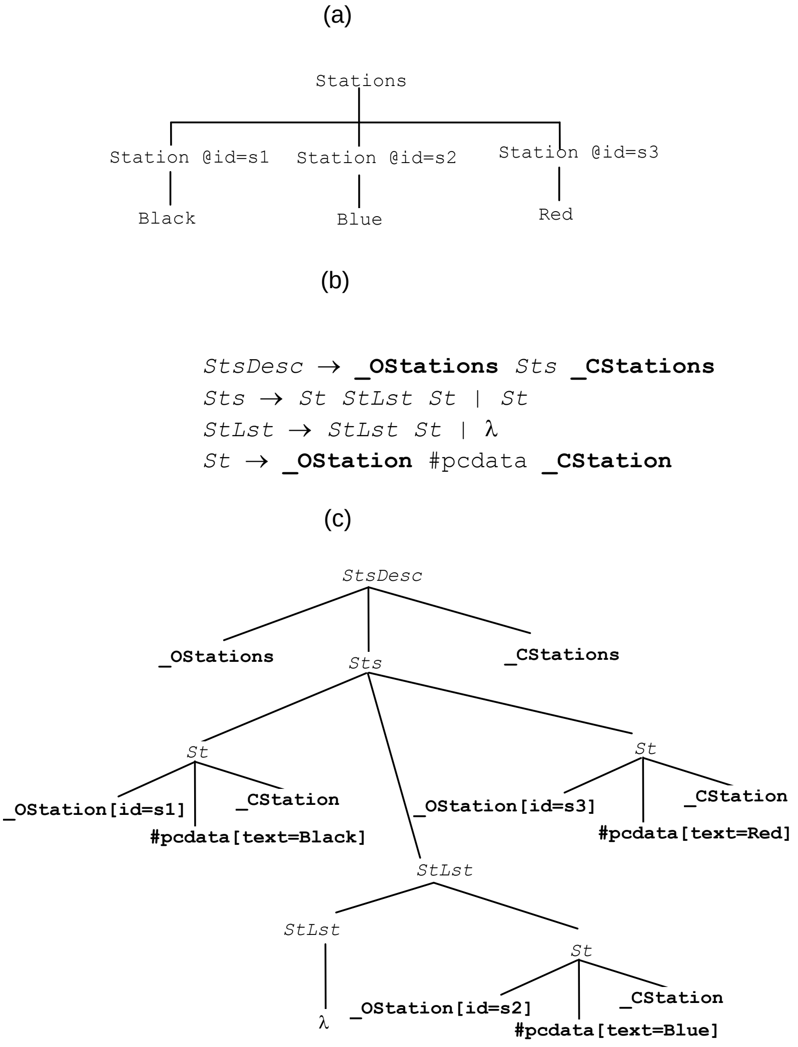
It is important not to confuse the generation-specific grammar with the document grammar (e.g., a DTD or an XML Schema) used to describe the markup language. The generation-specific grammar of this activity addresses a key aspect of the processing: to give a suitable structure to the stream of tokens in order to facilitate application generation. Indeed, this aspect must be addressed by any general-purpose XML processing solution. For instance, it is implicit in the code that deals with the children of an element node in a DOM-based processing application, in the callback methods and the state variables of a SAX event handler, or in the set of mutually recursive procedures of a StAX-based application. The main difference (and advantage) of our approach is that this structuring aspect is explicitly described at a very high abstraction level, as a context-free grammar, instead of being hand-coded in a final implementation. The structure imposed on a stream of tokens by a generation-specific grammar takes the form of a parse tree. Fig. 4b shows an example. As this example makes apparent, the parse tree is finer-grained than the usual document tree, where the element contents lack any structure outside a uniform sequence of nodes (compare Fig 4a with Fig 4c).

The conceptual processing model behind a generation-oriented translation scheme is to perform a traversal of the parse tree, executing semantic actions at significant points in this traversal. In addition, semantic actions can store and consult information in the nodes of the parse tree (typically this information is organized as an assignment of values to semantic attributes), as well as in global variables.

The exact nature of the traversal is determined by the kind of translators generated by the parser generation tool:

– *Top-down translators*, such as those generated by JavaCC and ANTLR, traverse the parse tree in *preorder* (i.e., the translator visits each node before visiting its children). The significant points are, for each node, when: (i) the translator enters the node, (ii) the translator enters a child, (iii) the translator has left a child, and (iv) the translator exits the node.

– *Bottom-up translators*, such as those generated by YACC-like tools (e.g., CUP), traverse the parse tree in *postorder* (i.e., for each node, the translator first visits the node's children and then the node itself). There is a significant point each time the translator exits a node.



(a)

(b)

StsDesc → **_OStations** *Sts* **_CStations**
Sts → *St StLst St* | *St*
StLst → *StLst St* | λ
St → **_OStation** #pcdata **_CStation**

(c)

Figure 4. (a) Document tree for the document in Fig. 3a; (b) (Part of) a generation-specific grammar; (c) Parse tree for the document in Fig. 3a according to this grammar.

It is worthwhile to note that, while it is useful to have this model in mind when writing generation-oriented translation schemes, it is only a conceptual model. In practice, the parse tree is never built, the traversal is implicitly performed during parsing, and the semantic actions are executed in a suitable order. Also, the semantic attributes are only available as parameters of recursive procedures (e.g., in recursive descent translators generated by JavaCC or ANTLR) or stored in the records of a semantic stack (e.g., in YACC-generated bottom-up translators). This behavior is a fundamental feature when dealing with huge documents (like those required by the generation of data-intensive applications) or with documents made available asynchronously in an XML stream (as required by on-line generators, which incrementally generate applications as they process their descriptions). It also constrains the kind of specifications that can be done. For instance, top-down translators do not work with *left-recursive* grammars, which are useful for characterizing left-associative structures. Also, although bottom-up translators are able to deal with left-recursion in a very efficient way, it is substantially more difficult to deal with *inherited* information (i.e., information that flows from parent to child or from sibling to sibling) than in top-down translators [1].

```
...
StsDesc → _OStations Sts _CStations {
   $$.stations = $2.stations
}
Sts → St StLst St {
   ops.addFirstStation($2.stations,$1.id,$1.name);
   ops.addLastStation($2.stations,$3.id,$3.name);
   $$.stations = $2.stations;
}
Sts → St {
   $$.stations = ops.makeStList();
   ops.addFirstStation($$.stations,$1.id,$1.name);
   ops.addLastStation($$.stations,$1.id,$1.name);
}
StLst → StLst St {
   $$.stations = addStation($1.stations,$2.id,$2.name);
}
StLst → λ {
   $$.stations = ops.makeStList();
}
St → _OStation #pcdata _CStation {
   $$.id = $1.id;
   $$.name = $2.text;
}
...
```

Figure 5. Excerpt of a translation scheme for a fragment of the <e-Subway> markup language

Knowing the traversal carried out by the translator makes it possible to place the semantic actions in the syntax rules of the generation-specific grammar. The specification formalism must also provide a way of referring to the semantic attributes (e.g., placing them as parameters of the syntax symbols, as in JavaCC, or using pseudovariables, as in YACC-like tools). Fig. 5 depicts a fragment of the syntax-directed translation scheme for a bottom-up translation model of the <e-Subway> generator using a YACC-like notation (in particular, it uses YACC-like pseudovariables: $$ to refer to the semantic record of a rule's head, $i to refer to the semantic record of the i-esime body's symbol). The translation scheme builds an in-memory representation of the stations in a line, following the typical generation pattern of populating a suitable semantic model [9].

Finally, it is interesting to remark that, for the sake of generalization, we have kept our approach simple enough to fit in the different parser generation tools available. For this reason, more advanced capabilities have been explicitly omitted, although they might facilitate some advanced processing tasks. For instance, one of these advanced capabilities could be the interplay between syntax and semantics, supported by tools like ANTLR [30], and which, for instance, would allow us to make parsing dependent on predicates concerning certain semantic attributes. Still, some clever behavior can be achieved without introducing these advanced features by setting the XML Scanner to produce different tokens for different occurrences of the same element type, depending of the values of some of their XML attributes.

3.3. Providing Generator-Specific Logic

The semantic actions in the translation scheme will typically use other, more conventional machinery that must also be provided to produce a fully functional application generator. This machinery constitutes the so-called *generator-specific logic*.

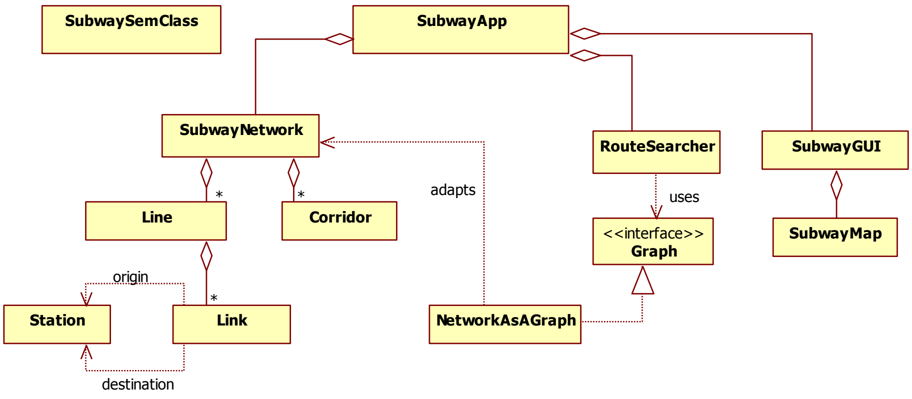


Figure 6. Main components of the <e-Subway> framework

For instance, in <e-Subway>, this generator-specific logic is formed by the <e-Subway> *framework*, which constitutes the aforementioned *semantic model* in this scenario [9]. Thus, and as indicated in section 2, the resulting generator does not generate actual code, but instantiates classes in the

<e-Subway> *framework* and links the resulting objects in appropriate ways (using the terminology introduced in [9], it *populates* the <e-Subway> *framework*, as we indicate below). Fig. 6 depicts the main components of the <e-Subway> framework.

In this way, the approach promotes a clear separation between the language-oriented processing of the XML documents and the conventional software that supports this processing. This separation can be further emphasized by providing a suitable façade for the generator-specific logic, with operations that will be invoked from the translation scheme (it indeed follows the embedment helper pattern described in [9]). The `ops` global variable in Fig. 5 illustrates this practice (in the actual <e-Subway> generator, the variable refers to an instance of such a façade, which is represented by `SubwaySemClass` in Fig. 6).

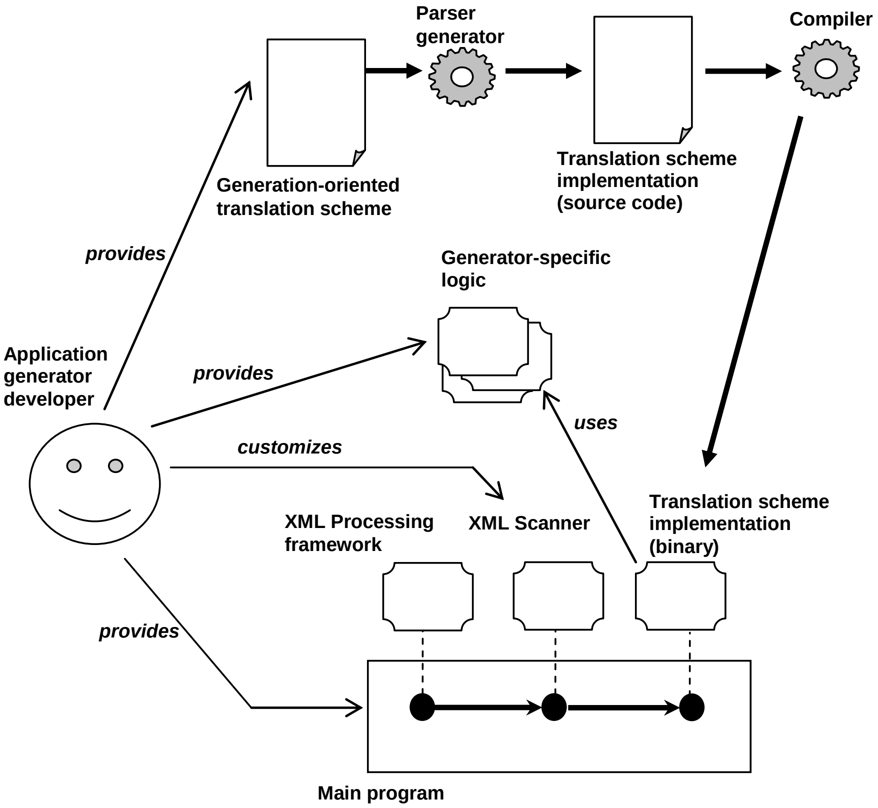

Parser generator
Compiler
Generation-oriented translation scheme
Translation scheme implementation (source code)
provides
Generator-specific logic
Application generator developer
provides
uses
customizes
Translation scheme implementation (binary)
XML Processing framework
XML Scanner
provides
Main program

Figure 7. The production process of XML-driven application generators in the development approach

3.4. Producing and Testing the Generator

Once the translation scheme and the application-specific logic are available, it is possible to get the working generator automatically by using the parser generation tool. The production process is detailed in Fig. 7. Indeed:

- The translation scheme is used as input to the parser generation tool in order to obtain the implementation of a translator written in the target language of the parser generation tool (e.g., Java for JavaCC or CUP). Notice that this way, the parser generation tool becomes a kind of meta-generator [6] in our proposal.
- In turn, this implementation can be turned onto a working binary component by using a compiler for such a target language (e.g., a Java compiler, assuming JavaCC or CUP was used).
- The customized XML Scanner must also be provided. Usually it can involve writing the mapping table (see section 3.1) using a customization file, or directly writing this table in the target programming language (e.g., Java).
- Finally, the developer must provide a small main program gluing all this together. This program will properly connect all the components required to constitute the generation pipeline. This pipeline will be made of: (i) a standard XML processing framework able to turn XML documents into information elements (e.g., represented by SAX events) suitable for the XML Scanner, (ii) the customized XML Scanner used to turn these elements into tokens accepted by the translator generated, and (iii) the translator itself, which makes use of the generator-specific logic.

The resulting generator can be tested in order to resolve possible defects and/or malfunctions. This activity therefore completes the development process.

4. Related Work

In this section we compare our work to conventional XML processing approaches (subsection 4.1), to other approaches to language-driven XML processing (subsection 4.2), and to approaches to XML processing based on attribute grammars (subsection 4.3)

4.1. Conventional XML processing approaches

As indicated in section 1, conventional approaches to XML processing range from task-specific ones (e.g., XSLT [41] for document transformation or XQuery [43] for expressing queries to XML structured documents) to general-purpose frameworks (e.g., tree-oriented ones, like DOM [19], or stream-oriented ones, like SAX [3][21], StAX [21] or XML-Pull [21]; see also [15] for a survey of this kind of general-purpose XML processing frameworks). While both types of these traditional approaches (task-specific and general purpose ones) share a data-centric orientation (i.e., these approaches see XML documents as chunks of data instead of sentences in a formal language), task-specific approaches tend to be of a higher level and of a more declarative nature than general-purpose ones. Indeed, task-specific approaches promote the use of domain-specific languages specifically tailored to the task at hand (e.g., transformation specifications in XSLT, FLWOR expressions in XQuery), while general-purpose processing frameworks are usually expressed in a general-purpose programming language (e.g., Java) and their use demands programming skills in this kind of general-purpose programming languages. As a consequence, task-specific approaches are usually more usable than general-purpose ones. However, the applicability of task-specific approaches is reduced to concrete processing tasks; for other tasks, either another task-specific approach or a general-purpose one will need to be used.

The language-oriented approach presented in this paper tries to bring together the best of the two aforementioned XML processing worlds (task-specific and general-purpose ones). Indeed, it clearly splits the processing task into two well-differentiated layers: (i) a *linguistic* layer, explicitly governed by an underlying formal grammar, which deals with the syntax-directed processing of the stream of basic components in an XML document, and (ii) an additional *specific logic* layer, which is understood as a set of additional services required by the linguistic layer. While the second layer must be provided by using general-purpose programming languages, the first layer can rely on domain-specific languages to describe syntax-directed language processing tasks, like those provided by the parser generation tools alluded to in this paper. As a consequence, the advantages of the approach from the development and maintenance perspective become apparent. On one hand, the linguistic layer can be expressed in domain-specific, high-level and largely declarative ways, using translation schemes, which can contribute to facilitating its conception, development and maintenance. On the other hand, since the approach does not constrain the nature of the specific logic layer, it is as general as any of the aforementioned general-purpose approaches. However, as a disadvantage, developers must face an increment in complexity due to the explicit organization of processing applications in these two well-differentiated layers. Of course, and as indicated in section 1, whether this complexity pays out or not will depend on the nature of the XML-based markup language: the more complex the language is, the more convenient the adoption of this proposal will be. Indeed, the non-trivial complexity of the markup languages that can arise in the domain of application generators makes this approach very convenient for this domain.

Our proposal can also be compared to traditional approaches from the point of view of efficiency, although, concerning the domain of application generators, where the documents involved will usually be small, this factor is less critical than ease of development and maintenance. Still, since our approach is intrinsically stream-oriented, it can usually give performances comparable to pure stream-oriented approaches based, for instance, on SAX or StAX. Indeed, another advantage to our approach, which is a direct consequence of using syntax-directed translation specifications built on top of underlying context-free grammars, arises: *when we develop XML processing applications we can think of trees, but the final applications will be executed as stream-oriented ones*. Therefore, the approach can achieve (and even beat) the usability of tree-oriented processing solutions, as well as the efficiency of stream-oriented ones.

Since it promotes a generative strategy to derive the actual implementation of the linguistic layer from a high-level specification based on the input language of a parser generation tool, our proposal has some points in common with XML *data binding* proposals [20]. A typical data-binding framework incorporates generators that are able to generate an application-specific representation by processing the *document grammar* (i.e., DTD or XML Schema) for the application's document type. As with the other conventional approaches mentioned, this representation is typically *data-centric*, as it consists of a set of application-specific classes, which are instantiated during parsing. Nevertheless, data-binding proposals are not exempt from disadvantages. Indeed, these proposals are tightly coupled with the document grammar, which is turned into application-specific classes using a more or less rigid set of pre-established rules. Although the proposals usually support *binding specifications*, which let developers modulate the classes generated and the *bindings* for the documents, the transformational capabilities of these specifications are usually limited to simple mapping facilities for elements and attributes. While these capabilities are sufficient for simple data-oriented XML applications, they fail when facing complex and/or mixed-element content models arising in non-trivial XML-based markup languages (such as those used in the domain of application generators). Our proposal, in turn, makes it possible to base the processing on generation-specific grammars, which are specific, not only to each language, but also to the processing task at hand.

4.2. Language-driven processing of XML documents

The conception of applications that process XML (or, more generally speaking, structured) documents as a sort of *compiler* or *translator* for a computer language has a long tradition in the document engineering context, such that it is highlighted, for instance, in [16]. Indeed, as it made apparent in [15], the internals of general-purpose XML processing frameworks can be explained from the point of view of conventional computer language processing workflows. However, as discussed in section 1, the application-specific processing of the documents usually operates on the data structures representing the documents provided by these frameworks. As a consequence, this application-specific processing is usually viewed as the processing of conventional data structures (e.g., traversing DOM trees, responding to SAX events, …) and the connection with language processing methods, techniques and tools is definitively missed. In order to restage this connection, some proposals (which are typically used for educational purposes) suggest undertaking the development of XML-based applications by building a parser for each particular XML-based markup language with the help of a parse-generation tool (see, for instance, [29], pages 351-352). As we indicated in section 3.1, this straightforward approach, however, supposes that we ignore general features common to any XML processing application (e.g., entity processing, comment recognition, namespace support, etc.). In this paper, we have shown how it is possible to use conventional parse generation tools in combination with standard XML processing frameworks to achieve the benefits of both approaches: on one hand, using standard and well-proven general-purpose XML processing frameworks to take advantage of general-purpose features common to any XML application, and, on the other hand, being able to organize application-specific processing in linguistic terms, as promoted by parse-generation tools.

The idea of parser generators have inspired several proposals for the construction of XML processing applications (e.g., ANTXR [40], which is built on top of the ANTLR parser generator tool, and RelaxNGCC [27], an extension of the RelaxNG [42] schema language for the specification of translation schemes). While these proposals usually rely on specialized tools supporting dedicated specification languages, in this paper we have shown how it is possible (and reasonable) to use conventional and well-proven parser generation tools without requiring dedicated languages for the description of the translation schemes. As indicated above, this fact is confirmed in our previous works [33][34], where we have shown how it is possible to build sophisticated XML processing environments by combining parser generators (JavaCC [14] and CUP [2]) with general-purpose stream-oriented XML processing frameworks (SAX and StAX).

4.3. XML Processing and Attribute Grammars

Although the tendency in formal models for processing XML documents is to emphasize tree automata and related formalisms [36], there are several works on using *attribute grammars*, a well-known formalism for describing the syntax and semantics of context-free languages [12][28], for the language-oriented implementation of XML processing tasks. Many of these works are typically focused on amalgamating attribute grammar concepts with the EBNF syntax that usually underlies an XML DTD, and which is reflected in unranked tree representations for the XML documents. The approach adopted in [31] to cope with EBNF is to decouple semantic rules and productions. Indeed, their semantic rules are associated in terms of parent-child relationships, instead of being associated with productions. This problem was addressed early by the work reported in [8] regarding a transformation system for structured documents supporting different document models (e.g., SGML, LaTEX, etc.). In [24][25], this kind of extended attribute grammar is used for querying structured documents, and it constrains the type of semantic expressions allowed to regular expressions in the alphabets of attribute occurrences. In the work described in [17][18], which reports on an application in the domain of information retrieval, documents are represented using abstract attribute grammars, where each non-terminal corresponds to an element type. In this work, a set of pre-established rules is used to derive such grammars from the DTDs, using a similar approach to that described in [16] (see [18] for an explicit enumeration of these rules). In [13], l-attributed grammars defined from EBNF syntaxes are used to support the efficient processing of XML streams. Similarly, in the works reported in [23][26] the unranked nature of the XML document trees is managed by promoting binary encodings of these document trees. Finally, in [32][35] we describe XLOP (XML *Language-oriented Processing*), an attribute grammar–based front-end to the proposal described in this paper. Indeed, by using encoding patterns similar to those described in [4] to implement attribute grammars by using conventional compiler construction tools, XLOP is able to turn the attribute grammar-based specifications of XML processing tasks into translation schemes for the CUP parser generation tool.

Our work in XLOP makes the relationships between the proposal described in this paper and attribute grammar-based approaches to XML processing apparent. Indeed, since the designer who writes an attribute grammar does not need to specify the evaluation order for the semantic equations, attribute grammars are of a higher level than translation schemes, where designers must make the execution order of the semantic actions explicit. However, many times it can burden the applicability of the approach, since average developers, who do not necessarily have deep knowledge of specialized formal semantic specification techniques, usually find it hard to work with non-standard computation models [9], like the dependency-driven one that underlies attribute grammars. For this purpose, the plain use of parser generation tools presented in this paper can provide an intermediate approach that can be more easily accepted by developers of XML processing applications. Also, sometimes parser generation tools can lead to more efficient / more straightforward implementations than those directly generated from attribute grammars. In addition, the use of patterns like the one described in [4] can enable hybrid approaches: indeed, it is possible to start with an attribute grammar-based specification, to encode it as a translation scheme using the patterns given in [4], and then to evolve it into a more efficient / more conventional implementation. These ideas have been partially applied in [34] by including dependency-driven translation capabilities in the application of the approach to the CUP + STaX marriage. Finally, based on our experiences, we have realized that one of the key aspects of the approach described in this paper is to perform the explicit provision of (plain BNF) context-free grammars (e.g., the generation-specific grammar) instead of relying on direct EBNF counterparts to the DTDs / document schemas as in [13][24][25], on pre-established rules to convert (EBNF-based) document grammars into BNF grammars as in [17][18], on the explicit decoupling of syntax and semantics as in [31], or on pre-established encodings of the document trees as in [23][26].

5. Conclusions and future work

In this paper, we have proposed a metalinguistic conception of the development of XML-driven application generators. According to this approach, these generators are treated as a sort of language processor. This treatment allows us to use compiler construction tools, and in particular parser generators, as adequate tools to orchestrate the development. It enables the automatic production of application generators from high-level specifications based on generation-oriented translation schemes. In addition, these application generators can be smoothly integrated with general-purpose standard XML-processing frameworks by using a generic and customizable XML Scanner. The approach facilitates the development and maintenance of application generators driven by complex XML-based markup languages, as well as by huge data-intensive XML documents and/or by documents that are provided asynchronously in an XML data stream.

Currently we are working on more flexible configuration mechanisms for the XML Scanner. We are also investigating mechanisms to improve the efficiency of the final generators. We are also planning to test the approach on the development of other application generators in the e-Learning domain, such as was reported in [22][37], as well as in the domain of multi-agent systems [10].

Acknowledgements. Thanks are due to project grants TIN2010-21288-C02-01 and Santander-UCM GR 42/10, group reference 962022. Also, Daniel Rodriguez-Cerezo was supported by the Spanish University Teacher Training Program (EDU/3445/2011).

References

1. Aho A.V., Lam M.S., Sethi R., Ullman J.D.; Compilers: principles, techniques and tools (2nd edition). Addison-Wesley. (2006)
2. Appel, A.W.: Modern Compiler Implementation in Java. Cambridge University Press. (1997)
3. Brownell, D. SAX2. O'Reilly. (2002)
4. Cerezo, D., Sarasa, A., Sierra, J.L. Implementing Attribute Grammars Using Conventional Compiler Construction Tools. 3rd Workshop on Advances in Programming Languages (WAPL'11), Szczezin, Poland. (2011)
5. Cleaveland, J.C.: Building Application Generators. IEEE Software, Vol. 5, No. 4, 25-33. (1988)
6. Cleaveland, J.C.: Program Generators with XML and Java. Prentice Hall. (2001)
7. Czarnecki, K.: Generative Programming: Methods, tools and Applications. Addison-Wesley. (2000)

8. Feng, A., Wakayama, T.: SIMON: A Grammar-based Transformation System for Structured Documents. Electronic Publishing, Vol. 6, No. 4, 361-372. (1993)
9. Fowler, M.: Domain Specific Languages. Addison-Wesley. (2010)
10. Fuentes-Fernández R., Gómez-Sanz J., Pavón J.: Requirements Elicitation and Analysis of Multiagent Systems Using Activity Theory. IEEE Transactions on Systems, Man, and Cybernetics, Part A, Vol. 39, No. 2, 282-298. (2009)
11. IMS. IMS Question and Test Interoperability 2.1. www.imsglobal.org/question/
12. Knuth, D.E: Semantics of Context-free Languages. Mathematical System Theory Vol. 2, No. 2, 127-145. (1968)
13. Koch, C., Scherzinger, S.: Attribute Grammars for Scalable Query Processing on XML Streams. The VLDB Journal, Vol. 16, No. 3, 317-342. (2007)
14. Kodaganallur, V.: Incorporating Language Processing into Java Applications: A JavaCC Tutorial. IEEE Software, Vol. 21, No. 4, 70-77. (2004)
15. Lam, T.C., Ding, J.J., Liu, J.C.: XML Document Parsing: Operational and Performance Characteristics. IEEE Computer, Vol. 41, No. 9, 30-37. (2008)
16. Leite-Ramalho, J.C.: Anotação Estrutural de Documentos e sua Semântica -- Especificação da Sintaxe, Semântica e Estilo para Documentos. Ph.D. Thesis. Braga, Portugal. (2000)
17. Lopes-Gançarski, A. L., Doucet, A., Rangel-Henriques, P.: Grammar-based Interactive System to Retrieve Information from XML Documents. IEE Proceedings-Software, Vol. 153, No. 2, 51-60. (2006)
18. Lopes-Gançarski, A., Rangel-Henriques, P.: Information Retrieval from Structured Documents Represented by Attribute Grammars. International Conference on Information Systems Modeling, Rep. Cheque. (2002)
19. Marini, J.: Document Object Model: Processing Structured Documents. McGraw-Hill. (2002)
20. McLaughlin, B. Java & XML Data Binding. O'Reilly. (2002)
21. McLaughlin, B. Java & XML. O'Reilly. (2006)
22. Moreno-Ger P, Sierra J.L., Martínez-Ortiz I., Fernández-Manjón B.: A Documental Approach to Adventure Game Development. Science of Computer Programming, Vol. 67, No. 1, 3-31. (2007)
23. Nakano, K.: An Implementation Scheme for XML Transformation Languages Through Derivation of Stream Processors. Second Asian Symposium of Programming Languages and Systems (APLAS'04), Taipei, Taiwan. (2004).
24. Neven, F.: Attribute Grammars for Unranked Trees as a Query Language for Structured Documents. Journal of Computer and System Sciences, Vol. 70, No. 2, 221-257. (2005)
25. Neven, F.: Extensions of Attribute Grammars for Structured Document Queries. 7th International Workshop on Database Programming Languages (DBLP'99), Kinloch Rannoch, Scotland, UK. (1999)
26. Nishimura, S., Nakano, K.: XML Stream Transformer Generation through Program Composition and Dependency Analysis. Science of Computer Programming, Vol. 54, No. 2-3, 257-290. (2005)
27. Okajima,D.: RelaxNGCC -- Bridging the Gap Between Schemas and Programs. XML.com, 8. (2002)
28. Paakki, J.: Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation. ACM Computer Surveys, Vol. 27, No. 2, 196-255. (1995)
29. Parr, T.: Language Implementation Patterns. Pragmatic Bookshelf. (2010)
30. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Bookshelf. (2007).

31. Psaila, G., Crespi-Reghizzi, S.: Adding Semantics to XML. 2nd International Workshop on Attribute Grammars and their Applications (WAGA'99), Amsterdam, The Netherlands. (1999)
32. Sarasa, A., Martinez-Aviles, A., Sierra, J.L., Fernández-Valmayor, A.: A Generative Approach to the Construction of Application-Specific XML Processing Components. 35th Euromicro Conference on Software Engineering and Advanced Applications, Patras, Greece. (2009)
33. Sarasa, A., Navarro, I., Sierra, J.L, Fernández-Valmayor, A.: Building a Syntax Directed Processing Environment for XML Documents by Combining SAX and JavaCC. 3rd Int. Workshop on XML Data Management Tools & Techniques. DEXA'08. September 1-5, Turin, Italy. (2008)
34. Sarasa, A., Temprado, B., Martínez, A., Sierra, J.L., Fernández-Valmayor, A.: Building an Enhanced Syntax-Directed Processing Environment for XML Documents by Combining StAX and CUP. Fourth Int. Workshop on Flexible Database and Information Systems. DEXA'09. August 31 – September 4, Linz, Austria. (2009)
35. Sarasa, A., Temprado-Battad, B., Sierra, J.L, Fernández-Valmayor, A.: XML Language-Oriented Processing with XLOP. 5th International Symposium on Web and Mobile Information Services, Bradford, UK. (2009)
36. Schwentick, T.: Automata for XML - A Survey. Journal of Computer and System Sciences, Vol. 73, No. 3, 289-315. (2007)
37. Sierra, J.L, Fernández-Valmayor, A., Fernández-Manjón, B.: From Documents to Applications Using Markup Languages, IEEE Software, Vol. 25, No. 2, 68-76. (2008)
38. Sierra, J.L., Fernández-Valmayor, A., Fernández-Manjón, B., Navarro, A.: ADDS--A Document-Oriented Approach for Application Development, Journal of Universal Computer Science, Vol. 10, No. 9, 1302-1324. (2004)
39. Sierra, J.L., Fernández-Valmayor, A., Fernández-Manjón, B.: A Document-Oriented Paradigm for the Construction of Content-Intensive Applications. The Computer Journal, Vol. 49, No. 5, 562-584. (2006)
40. Stanchfield, S. ANTXR: Easy XML Parsing, based on The ANLR Parser Generator. javadude.com/tools/antxr/index.html (current October 2011)
41. Tidwell, D.: XSLT, 2nd Edition. O'Reilly. (2008)
42. Vlist, E. Relax NG. O'Relly. (2003)
43. Wallmsley, P. XQuery, O'Reilly. (2007)

Antonio Sarasa-Cabezuelo is a full-time Lecturer in the Computer Science School at Complutense University of Madrid, Spain (UCM). His research is focused on the language-oriented development of XML-processing applications, and on the development of applications in the fields of digital humanities and e-Learning. He was one of the developers of the Agrega project on digital repositories (a pioneer project in this field in Spain). He is a member of the research group ILSA (Implementation of Language-Driven Software and Applications: http://ilsa.fdi.ucm.es). He has participated in several research projects in the fields of software language engineering, digital humanities and e-learning, and he has published over 50 research papers in national and international conferences.

Bryan Temprado-Battad is a PhD. Student in the Computer Science School at UCM and a member of the ILSA Research Group. His research is focused on language oriented development and on attribute grammars applications, being one of the principal contributors to the development of the XLOP System. The results of his works have been published in several research papers in international journals and conferences.

Daniel Rodríguez-Cerezo is a PhD student in the Computer Science School at UCM, and a member of ILSA. His research is focused on the use of several e-Learning techniques (simulations, interactive prototyping tools, recommendation systems for learning object repositories, etc.) to improve teaching and learning of the Software Language Engineering discipline. Besides, he is interested in the development and improvement of software language engineering techniques.

José-Luis Sierra is an Associate Professor at the UCM's Computer Science School, where he leads the ILSA Research Group. His research is focused on the development and practical uses of computer language description tools and on the language-oriented development of interactive and web applications in the fields of digital humanities and e-Learning. Prof. Sierra has leaded and participated in several research projects in the fields of digital humanities, e-learning and software language engineering, the results of which have been published in over 100 research papers in international journals, conferences and book chapters. He serves regularly as reviewer / PC Member for several international reputed journals and conferences.

Received: May 05, 2011; Accepted: March 21, 2012.