Journal of Network and Computer Applications 32 (2009) 1092–1105

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Language engineering techniques for the development of e-learning applications

Iván Martínez-Ortiz, José-Luis Sierra *, Baltasar Fernández-Manjón, Alfredo Fernández-Valmayor

Dpto. Ingeniería del Software e Inteligencia Artificial. Fac. Informática, Universidad Complutense de Madrid, C/ Profesor José García Santesmases, s/n. 28040 Madrid, Spain

ARTICLE INFO

Article history:
Received 19 June 2008
Received in revised form
29 January 2009
Accepted 26 February 2009

Keywords:
E-learning applications
Language engineering
Domain-specific languages
Authoring
Model checking
Rapid prototyping

ABSTRACT

In this paper we propose the use of language engineering techniques to improve and systematize the development of e-learning applications. E-learning specifications usually rely on domain-specific languages that describe different aspects of such final e-learning applications. This fact makes it natural to adopt well-established language engineering principles during the construction of these applications. These principles promote the specification of the structure and the runtime behavior of the domain-specific languages as the central part of the development process. This specification can be used to drive different activities: rapid prototyping, provision of authoring notations and tools, automatic model checking of properties, importation/exportation from/to standards, and deployment of running applications. This language engineering concept also promotes active collaboration between instructors (the users of the languages) and developers (the designers and implementers) throughout the development process. In this paper we describe this language-driven approach to the construction of e-learning applications and we illustrate all its aspects using a learning flow sequencing language as a case study.

1. Introduction

A common practice in e-learning is the use of languages to describe the different aspects of a learning scenario (e.g. content, activities, participants, etc). IMS standardization efforts are good examples of this trend (Friesen, 2005). Indeed, many of these efforts result in suitable languages for the description of specific aspects of an e-learning application. Among them it is possible to find languages for packaging learning contents (i.e. the IMS Content Packaging specification) (IMS, 2004), for describing the different products involved in an assessment process (i.e. the IMS Question & Test Interoperability specification) (IMS, 2006), for describing the profile of a particular learner (i.e. the IMS Learner Information Package specification) (IMS, 2005), for sequencing the activities in a learning flow (i.e. the IMS Simple Sequencing specification) (IMS, 2003b), or even for characterizing the different teaching methods arising in heterogeneous learning situations (i.e. the IMS Learning Design specification) (IMS, 2003a).

While these standardization efforts stress the use of languages to solve interoperability issues (i.e. as vehicles that can be used by heterogeneous platforms to exchange information), in our works we have promoted a complementary philosophy: using suitable languages to describe applications that are generated by automatically processing these descriptions (Sierra et al., 2006b). This philosophy is shared by the approaches to software development based on domain-specific languages (Deursen et al., 2000; Mernik et al., 2005). According to these approaches software development is conceived as a *language engineering* process, where suitable domain-specific languages are specified, implemented and maintained for each application domain, and where software applications are described using these languages instead of general-purpose programming ones. These approaches are specially well suited to domains where having efficient mechanisms to norm the interaction between domain experts and developers is a must. E-learning is a paradigmatic example of these domains, since many times the cost of providing the contents and fine-tuning the final applications exceeds by several orders of magnitude the initial development cost of the software infrastructures where the applications will finally be deployed. The adoption of a language-driven approach in e-learning results in a more rational distribution of responsibilities among the participants in the development process. Instructors will be in charge of producing and maintaining the final applications, while developers act as language engineers responsible for formalizing and maintaining the languages used by the instructors. Developers are also in charge of the software infrastructure associated with such languages (including the generators used to produce the final running applications).

* Corresponding author. Tel.: +34 913947548; fax: +34 913947547.
E-mail addresses: imartinez@fdi.ucm.es (I. Martínez-Ortiz), jlsierra@fdi.ucm.es (J.-L. Sierra), balta@fdi.ucm.es (B. Fernández-Manjón), valmayor@fdi.ucm.es (A. Fernández-Valmayor).

1084-8045/$ - see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jnca.2009.02.005

We have successfully tested these language-driven principles in the development of several e-learning systems and applications (Fernández-Manjón and Fernández-Valmayor, 1997; Moreno-Ger et al., 2007; Sierra et al., 2006c, 2007b, 2008c), where we have tested the importance of using domain-specific languages to orchestrate the collaboration needed between instructors and developers. In these experiences we have also realized the feasibility of a complementary approach to the interoperability-oriented standardization efforts when adopting a language-driven process model. Instead of seeking universal solutions, we consider that it is also interesting to take a different approach by formalizing the languages already used by instructors in their specific learning domains. Since these languages are domain specific and part of the instructors' experience, they are more understandable and easier to use for the instructors than the more generic ones. It does not mean that standardization issues are ignored. Indeed, these issues can be subsequently addressed by appropriate importation/exportation modules. However, by adopting a language engineering perspective, where developers are constantly providing suitable linguistic support according to the particular needs of the instructors, it is possible to promote an instructor-centered development process, which results in instructors' deeper involvement in the production and maintenance of the e-learning applications. In this work we will mainly illustrate this *bottom-up* approach, similar to the experiences reported by Sierra et al. (2006a), although the techniques will also be largely applicable in a *top-down* manner, based on the use, specialization and adaptation of pre-existing languages, such as the one proposed by Moreno-Ger et al. (2006).

The present paper exposes the marriage between e-learning and language engineering. For this purpose, it describes and illustrates the different activities promoted by a language-driven approach in e-learning, and how these activities are organized around sound specifications of domain-specific languages. These specifications start with the characterization of abstract *information models* for the languages, which are described in both a conceptual and a formalized way. The structural formalization of the languages allows for the subsequent formalization of their runtime behaviors in the form of suitable *operational semantics*. The resulting specifications are used to drive many other language engineering activities. Indeed, these specifications can be readily used to build running prototypes of the languages in a straightforward way, which can be used to refine the structure and the semantics of these languages. They can also be used to provide notations that are more user-friendly for the instructors (e.g. a graphical notation), by specifying a mapping between these notations and the abstract information models. The resulting languages facilitate the automatic checking of properties, which results in better authoring support for instructors. Since the usual e-learning specifications are also language based, it is possible to connect these specifications using appropriate language translations. Finally, the resulting high-level designs can be easily deployed using the well-known model-view-controller (MVC) pattern (Krasner and Pope, 1988), which is typically used for organizing many modern web-based applications.

The structure of the paper is as follows. Section 2 motivates the language-driven approach by comparing it with conventional development models. In Section 3 we introduce a case study that will be used throughout the paper for illustrative purposes. In Section 4 we focus on the structural and behavioral specification of domain-specific languages. Section 5 is devoted to analyzing the different activities enabled by this language-driven approach. The paper finishes by presenting some conclusions and lines for future work (Section 6).

2. Language-driven development of e-learning applications compared to conventional development approaches

In a conventional development process model instructors are requirement providers, while developers act as application implementers. By using conventional requirement acquisition techniques, developers interview instructors to determine which resources (i.e. contents, support tools, etc.) must be incorporated in the application, as well as how the final users (e.g. instructors, learners, etc.) interact with these contents. For example, as result of the requirement acquisition process to develop an e-learning course, instructors determine, among others: the learning contents and tools needed, the structure of the course/lessons and the transitions between the different parts of a lesson or the whole course. With all this information, and using general-purpose programming languages and technologies (e.g. Java, XML, etc.), developers implement the application; for example, they can provide a web-based implementation by using a suitable framework for the development of web-based applications, such as Apache Struts (Goodwill and Hightower, 2003). Then, the developed application is evaluated by instructors (perhaps with the help of end-users), who eventually can discover some aspects to be improved in the contents or in the interactions. Thus, instructors propose modifications and/or improvements in the application to the developers, who produce an enhanced application, starting a new evaluation (e.g. instructors could include new content, modify existing ones, as well as modify the learning flow which governs the transitions between the different parts of the course). This iterative behavior, characterized by the production/modification of applications, finishes when a satisfactory application has been obtained but it needs to be started again when the application needs to be updated (Fig. 1a). In fact, the process model described resembles the *Analysis Design Development Implementation and Evaluation* (ADDIE) methodology extensively used for the development of e-learning

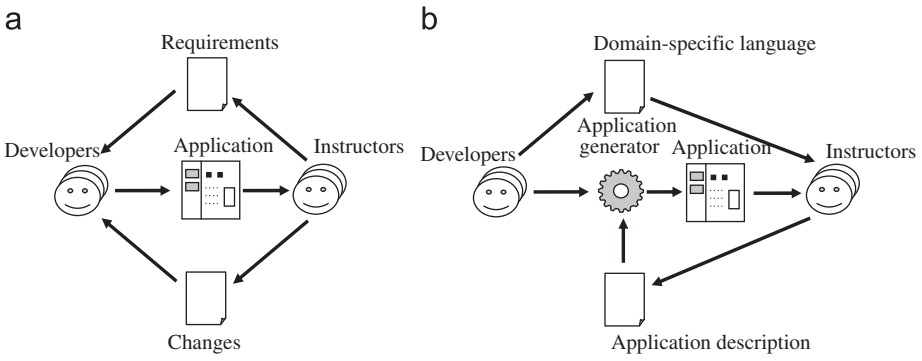

Fig. 1. Simplified views of (a) conventional development of e-Learning applications, and (b) language-driven development.

materials (Allen, 2006; Molenda, 2003). The main advantage of this conventional approach is its flexibility, because it avoids the expressive limitations that can be imposed by using a specific authoring tool (on the contrary, the process models explicitly involve a development team able to react to the needs and requirements of the instructors). However, the cost of producing and maintaining the applications can be prohibitively high (Sierra et al., 2006b). Indeed, a typical e-learning application can manage high volumes of contents and exhibit complex interactions with the different participants in the learning process (e.g. learners, instructors, assistants, etc.). Then, when instructors want to report an application change (e.g. error, new feature), many times the best way that they can do so is to document their proposed modifications, for example annotating them directly onto applications' screenshots (i.e. communication between instructors and developers could be orchestrated by descriptions based on natural language as well as by other kind of documents). These annotations are manually analyzed by developers to produce the adequate application (e.g. in Struts, developers should tweak XML deployment descriptors and Java code to achieve the desired modifications). Thus, the resulting production and maintenance loop is costly and sometimes inefficient due to the communication gap that exists between instructors and developers.

In turn, in a language-driven approach instructors use suitable domain-specific languages to produce and maintain the applications, while developers produce suitable implementations of such domain-specific languages. This conception of application development is possible because domain-specific languages can integrate primitive constructions, means of combination and means of abstraction which are close to the expertise and the needs of the instructors. Therefore, these languages are easier to use than other general-purpose programming languages for solving the specific problems posed by instructors (i.e. instructor-oriented languages are not general-purpose programming languages, but are languages for describing the different aspects of the teaching and learning processes). Thus, by adopting a language-driven approach the production and maintenance loop can be speeded up. Indeed, by using suitable domain-specific languages, instructors can describe the relevant aspects of an e-learning application (e.g. learning flows and contents) in a form which is machine processable. By doing so, applications can be automatically *generated* by using suitable *application generators* (compilers or interpreters) for these languages, which are also provided by developers. The overall approach is summarized in Fig. 1b.

Application generators can work, for instance, by automatically instantiating application frameworks (Sierra et al., 2008c). Thus, the language-driven approach does not imply that conventional programming languages and technologies (e.g. Java, XML, etc.) become unnecessary. On the contrary, these technologies also play a very important role since developers use them to provide the application generators for the domain-specific languages and the other software infrastructure required (e.g. suitable application frameworks). In addition, the design of domain-specific languages can make use of higher-level specification formalisms and language engineering techniques. Indeed, once the different aspects of a language are rigorously specified, implementing it is an ordinary (although possibly tedious) programming task. Therefore, next sections concentrate on the language engineering techniques involved in the design of domain-specific languages for e-learning, as well as on the different activities enabled by this language-driven approach.

3. An example

In order to illustrate the different aspects presented in this paper we will use a simple learning flow sequencing language. This language is a simplified version of a flow-driven educational modeling language included in ⟨e-LD⟩, a system developed at the Complutense University of Madrid (Martínez-Ortiz et al., 2007a). ⟨e-LD⟩ supports the authoring and playing of Units of Learning described with the aforementioned IMS Learning Design (IMS LD) specification (IMS, 2003a; Koper and Olivier, 2004).

IMS LD is a powerful activity and role-oriented educational modeling language developed at the Open University of the Netherlands as an evolution of the previous language EML (acronym for *Educational Modeling Language*) developed by the Professor Rob Koper's group (Koper and Manderveld, 2004). Since 2003 the IMS LD specification has been adopted by the IMS consortium. IMS LD incorporates a powerful *condition system* similar to a rule-based mechanism that makes it possible to design complex adaptive learning behaviors (Burgos et al., 2007b). However, during our experiences with IMS LD in developing adaptive game-based e-learning contents (Burgos et al., 2007a) we observed that many instructors found it easier to describe their designs using a flow-oriented notation. For this purpose, in ⟨e-LD⟩ we use a simple flow-driven domain-specific language, which is similar in spirit to the aforementioned IMS Simple Sequencing (IMS, 2003b), but with support for roles. Moreover, we significantly adopt language engineering principles in ⟨e-LD⟩ in order to keep the language adjusted to the needs of the instructors, while still maximizing its usability from an authoring point of view. We maintain an abstract *information model* for this language (or, using terminology taken from the model-driven development community, a *metamodel*—Stahl et al., 2006), and we provide several *bindings* (e.g. an XML-based one, a graphical one, etc.) in order to adapt the language to the aforementioned authoring needs. We also provide *export* and *import* mappings with IMS LD.

For sake of simplicity, in this paper we will ignore the aspects regarding roles, and we will only consider the sequencing part of the language in ⟨e-LD⟩. We will also avoid the description of learning objectives, environments, the description of activities, metadata, etc. The purpose is to work with a language useful enough for illustrative purposes, but still simple enough to fit in the limits of a single paper. Therefore, our example language will be able to describe how to sequence learning activities in a single-user learning flow. The language will support either single sequencing (i.e. the current step has a single next step), alternative sequencing (i.e. the next step depends on a condition), or multiple sequencing (i.e. there are several next possible steps to do). Activities will be *exposed* to the learner, and they will be *attended* by him/her. As a result, a set of *achievements* will be reached. Conditions themselves will be formulated as Boolean expressions on achievements.

In order to exemplify the kind of descriptions that can be done with this language, consider a simple example regarding a course on *Enology* (i.e. the science of wine). In this course the learner must first attend some classroom lectures. Then he/she must examine some background material (in particular, two papers written by *Joseph Birra* and *Mary Bebo*, two-fictitious—popular experts on the matter, and a web site about the art of wine tasting). After that, he/she must pass a knowledge test on the matter. Depending on the result, he/she can be compelled to re-examine the material, or otherwise he/she can proceed with the learning process. The rest of the learning process consists of the realization of a final-course work assignment, which must be graded positively before finishing the course. This learning flow is described in Fig. 2 using natural language.

4. Language design

The central step in the language-driven approach is to design the domain-specific language used to describe the key aspects of

the target applications. This design process must characterize the language both in a *structural dimension* (i.e. how the aspects addressed can be described using the language) and in a *behavioral dimension* (i.e. how the applications described behave). In this section we address these two design dimensions by emphasizing in the language engineering principles used. In Sierra et al. (2008a, 2008b) there are additional examples of applying this process into the context of a Socratic Tutoring System (Bork, 1985; Ibrahim, 1989).

4.1. Designing the language's structure

The design of the language will be addressed from an abstract point of view. Thus, the language itself will be characterized as an *information model*, instead of a concrete grammar. The rationale in doing so, which is also a common practice in e-learning specifications and standards, is to maximize the reusability of the language in different contexts.

Indeed, the language will be adapted to each particular context by providing a suitable *binding* to a concrete notation, as is also usual in the e-learning standardization practice. From a language engineering perspective, this structural design of the language is analogous to the definition of the *abstract syntax* of a conventional programming language, which lets language designers capture the features of the language that are essential for further specifying the meaning of their constructs (Friedman et al., 2001). Also, and as said before, the process can be understood as a meta-modeling activity in the context of model-driven engineering (i.e. as the design of a domain-specific modeling language—Stahl et al., 2006).

1. Perform the *Attend to lectures* activity
2. Perform the following activities
   - *Read Birra's Paper*
   - *Read Bebo's Paper*
   - *Visiting wine tasting website*
3. Perform the *Do Test* activity
4. If the achievement *test-passed* has not been achieved, go to step 2
5. Perform the *Do Work Assignment* activity
6. If *work-is-successful* has not been achieved, go to step 5

Fig. 2. An example of learning flow sequencing.

The information model can be initially provided by using standard data modeling techniques (e.g. entity-relationship diagrams, or UML class diagrams). In addition, it can be further formalized in terms of a first-order signature (i.e. a set of function and predicate symbols) with the aim of subsequently enabling the formal definition of the behavioral dimension.

In the case of our 〈e-LD〉 sequencing language, the abstract structure of a sequencing specification is a directed graph whose nodes correspond to learning activities, as well as to Boolean and flow operations. In Fig. 3 we sketch the information model of this language using an UML-like notation. This model characterizes a *learning flow sequencing specification* as a set of *commands*, which are further classified into *simple* and *branched* ones. Simple commands can only have a single *next* command in the specification, while branched ones can have two *next* commands. Simple commands include the exposition of a learning activity (*Activity*), atomic (*Achievement*) and composite (*And*, *Or*, *Not*) Boolean operations (they enable us to represent conditions in postfix notation), and a command for synchronizing several learning paths (*Join*). In turn, branched commands include a command for making alternative sequencing (*Alt*), and another one for performing multiple ones (*Fork*). Finally, the model includes a *Stop* command, which will be used to finish the sequencing. In Table 1 we summarize the predicates used to formalize descriptions that follow this information model. There is a predicate for each command. The first argument corresponds to a unique identifier for the command. In simple commands, the last argument refers to the identifier of the command in the next step. Finally, in Fig. 4 we exemplify the representation of the sequencing in Fig. 2 using this formalized abstract notation.

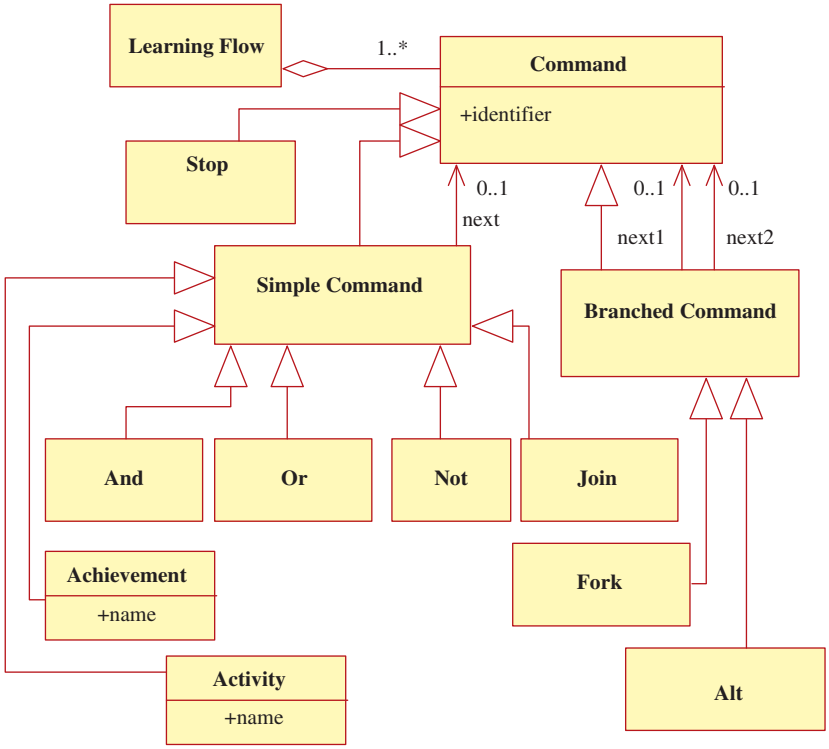

Fig. 3. Information model for the 〈e-LD〉 sequencing language.

Table 1
Predicates used to formalize the model of the ⟨e-LD⟩ sequencing language.

Predicate	Intended meaning
Activity (*i*,*a*,*j*)	Activity *a* must be exposed to the learner
Achievement (*i*,*a*,*j*)	*True* if the learner has reached achievement *a*. *False* otherwise
And (*i*,*j*)	True if the two last conditions tested were both *true*. *False* otherwise
Or (*i*,*j*)	*False* if the two last conditions tested were both *false*. *True* otherwise
Not (*i*,*j*)	*True* if the last condition tested was false. *False* otherwise
Alt (*i*,*j*,*k*)	If the last condition tested was *true*, the next step is command *j*. Otherwise it is command *k*
Fork (*i*,*j*,*k*)	The next step can be both command *j* and command *k*
Join (*i*,*j*)	Synchronize all the learning paths leading to it
Stop (*i*)	It stops the sequencing process

```
Activity (1,Attend to lectures,2)
Fork (2,3,4)
Activity (3,Read Birra's Paper,7)
Fork (4,5,6)
Activity (5,Read Bebo's Paper,7)
Activity (6,Visit wine tasting website,7)
Join (7,8)
Activity (8,Do Test,9)
Achievement (9,test-passed,10)
Alt (10,11,2)
Activity (11, Do Work Assignment, 12)
Achievement (12,work-is-successful,13)
Alt (13,14,11)
Stop (14)
```

Fig. 4. Abstract representation of the learning flow sequencing outlined in Fig. 2.

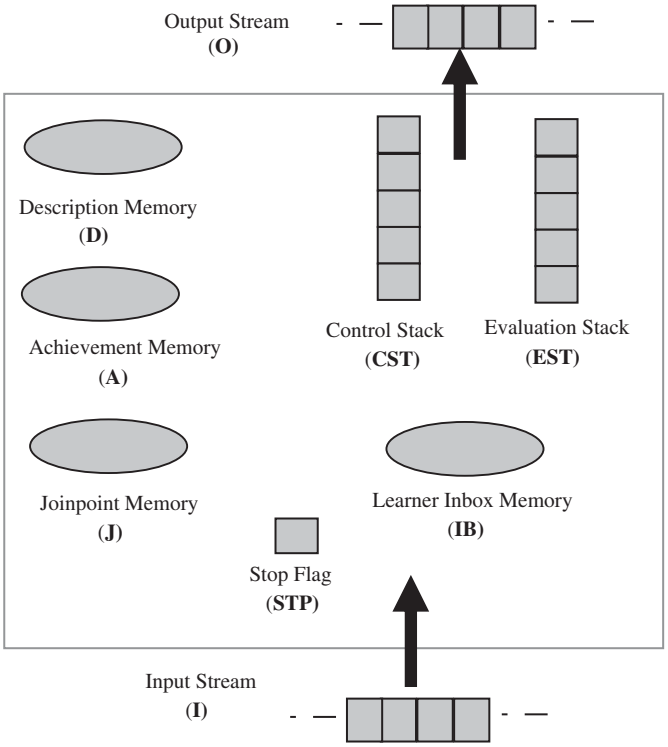

Fig. 5. Structure of the abstract machine for the ⟨e-LD⟩ sequencing language.

4.2. Specifying the language's behavior

Once the structure of the language has been defined, the second step in the language design is to characterize the runtime behavior of this language. This second step will be addressed by assigning suitable *operational semantics* to the language. Such semantics will model the runtime behavior as transitions between *computation states*. Therefore, in order to model the behavior, we need in turn to decide how to represent those computation states, and also how to characterize the transitions mentioned. These two aspects will be addressed in the following sections.

4.2.1. Characterizing computation states

The characterization of computation states is analogous to devising the architecture of an abstract machine for executing the descriptions in the domain-specific language. Since the language is specific, the resulting machine will also be so. Computation states (and hence the associated abstract machines) typically will include:

- The description in the language (which can be thought of as the program stored in the machine's memory).
- One or several control elements used to decide how the execution must proceed (e.g. information about the current instruction, or the current set of instructions in the case of multithreaded execution).
- A context (which can be thought of as the contents of the machine's registers and internal memory).
- Several streams communicating with other components (which can be thought of as buses connecting the CPU with input, output and other devices). These streams are very useful for isolating the behavioral details from the presentation/interaction/updating aspects.

This organization naturally leads to MVC-based architectures for the final implementations, as will be detailed below. Also, in order to facilitate language evolution, computation states can be characterized as indexed sets of components (i.e. sets of pairs with the form ⟨*c*,*C*⟩, where *c* is the component's name and *C* is the corresponding component value). It will facilitate the addition of new components to the states without affecting the specifications already available, with results similar to the use of *labels* in the transitions as proposed in works on the modularization of operational semantics specifications (Mosses, 2004). To facilitate the manipulation of this representation, we will introduce the following notational conventions:

- σ_c: It will be used to *obtain* the value of component c in state σ. Formally, $\sigma_c = C \Leftrightarrow \langle c, C\rangle \in \sigma$.
- $[c/C]$: It will used to *update* the value of component c in state σ. The new value will be C. Formally, $\sigma[c/C] = (\sigma - \langle c, \sigma_c\rangle) \cup \{\langle c, C\rangle\}$.
- $[c_1/C_1, c_2/C_2, \ldots, c_k/C_k]$: It will be a shortcut for $\sigma[c_1/C_1][c_2/C_2]\ldots[c_k/C_k]$.

Regarding our case study, in Fig. 5 we outline the structure of an abstract machine that can be used to execute learning flow sequencing descriptions. In this structure:

- The description will be stored in a *description memory* (abbreviated as **D**). This memory will consist of the sets of sequencing commands.
- The control elements will be a *control stack* (**CST**), a *Learner Inbox Memory* (**IB**) and a *stop flag* (**STP**). Each element in the control stack will be a pair ⟨*j*, *i*⟩ where *i* points to a command

in the description memory, and j points to the preceding command, also in the description memory (this last reference will be required during the execution of the *Join* commands). In turn, the learner inbox memory will store those activities that are currently exposed to the learner, along with the information needed to restart the execution once these activities are attended. Finally, the stop flag will indicate whether the execution has stopped or not.

- The context will be represented by an *Evaluation Stack* (**EST**), an *Achievement Memory* (**A**), and a *Joinpoint Memory* (**J**). The evaluation stack will be used to store the results of the conditions tested that will be required in subsequent tests. The achievement memory will store all the achievements established as a result of executing the activities. Finally, the *joinpoint* memory will store information about the *Join* commands executed. Its elements will be pairs of the form $\langle i, W\rangle$, with i being the identifier of a *Join* command and W a set of references to the preceding commands pending execution.
- Finally, the machine has an *Input* (**I**) and an *Output* (**O**) stream. The input stream will contain the result of executing the activities —i.e. **attended** (*a*,*As*) terms, with *a* being an activity's name and *As* a set of achievements. The output stream will contain commands to expose activities to the learner—i.e. **expose**(*a*) terms, with *a* the name of an activity.

Therefore the computation states managed in the semantics will be sets of the form

$$\{\langle \mathbf{D}, \_\rangle, \langle \mathbf{CST}, \_\rangle, \langle \mathbf{EST}, \_\rangle, \langle \mathbf{A}, \_\rangle, \langle \mathbf{J}, \_\rangle, \langle \mathbf{IB}, \_\rangle, \langle \mathbf{I}, \_\rangle, \langle \mathbf{O}, \_\rangle, \langle \mathbf{STP}, \_\rangle\} \quad (1)$$

where _ denotes an anonymous unique syntactic variable.

To facilitate the manipulation of the description, by **pred**(*D*,*i*) we will denote the set of identifiers of all the commands pointing to command *i* in description *D*. Finally, in order to facilitate the manipulation of the stacks and streams, we will abstract the specifications in the following typical operations:

- **empty**(*S*) for checking the emptiness of a stack, **top**(*S*) for consulting the stack's top, **pop**(*S*) for popping the stack's head, and **push**(*e*,*S*) for pushing element *e* into stack *S*.
- **in**(*S*) for reading from a stream, **out**(*e*,*S*) for writing element *e* in stream *S*, and **close**(*S*) for closing the stream.

4.2.2. Specifying the semantic rules

Once the structure of the computation states is set, the language's runtime behavior can be formalized. For this purpose we propose using the *structural style* of operational semantics (Plotkin, 2004; Mosses, 2006). In this style, semantics rules typically adopt the form

$$\frac{\Phi_o;;\ldots;;\Phi_k}{\Psi_o;;\ldots;;\Psi_n} \quad (2)$$

and therefore they resemble the inference rules of formal calculi in logic. The meaning of these rules is also straightforward: if sentences Φ_i in the premise hold, the sentences in conclusions Ψ_j will also hold. Some of these sentences will typically adopt the form of $\sigma_o \rightarrow \sigma_1$, meaning a transition from state σ_o to state σ_1. Other ones will involve additional tests and constraints on the possible states using typical logical and set theoretic operations. In addition, we use a *small-step* style of specifying the semantics (Mosses, 2006). This specification style concentrates on characterizing transitions between consecutive states, and will facilitate subsequent development steps, such as model checking and implementation.

In Fig. 6 we show the semantic rules for our example. These rules characterize the sequencing process as the iteration of two stages:

- The first stage executes all the sequencing steps required to expose the activities to the learner. For this purpose, commands must be successively executed until reaching an *Activity* one. In addition, *Fork* commands can cause several activities to be exposed. It requires tracking several execution paths instead of a single one, which is achieved by using the control stack. Indeed, this stage supposes executing commands until this stack is empty. The stage itself is governed by all the rules but the last one. The **activity-exposition** rule characterizes the execution of the *Activity* commands, and therefore the exposition of the activities to the learner: a suitable *exposed* command is written in the output stream, and information required to resume the execution when the activity has been attended is stored in the learner inbox memory. The **achievement-test**, **and-test**, **or-test** and **not-test** rules characterize how the different conditions are tested. For achievements, the result is *true* or *false* depending on where the achievement is or is not stored in the achievement memory. For composite conditions, the result is obtained by operating on the top values of the evaluation stack, which will correspond to the operands already evaluated, and such a result will replace the top values as is usual in the stack-based evaluation of expressions (Aho et al., 2007). The **alt-branching** and the **fork-branching** rules characterize the execution of the branching commands. While the next step to execute due to the *Alt* command depends on the top of the evaluation stack, the *Fork* command supposes pushing the two possible next steps in the control stack. The execution of a *Join* command is modeled using three different rules. The first one (**join-activation**) models the activation of a joinpoint: pointers to the *Join* command and to the pending commands are stored in the joinpoint memory. Also notice that the control stack is unchanged, to deal with the actual execution of the command. This execution supposes blocking the sequencing if there are more commands pending execution (**join-blocking** rule), or to restart it if such execution was provoked by the last one pending (**join-unblocking** rule). Finally, the execution of the *Stop* command supposes stopping the sequencing process (**stop** rule).
- The second stage, which is governed by the rule **activity-attendance**, supposes taking care of learner's interactions, which in this case study are reduced to attending activities. This stage comprises a step where the input stream is read to determine the activity attended and the achievements established as a result of such an action. These achievements are then used to update the achievement memory, and the continuation information, stored in the learner inbox memory, is used to restart the sequencing process by pushing this information in the control stack.

5. Enabled activities

As said before, once the design of the language is available, it can be used to organize several activities, which are facilitated by using well-established language engineering techniques. In particular, in this section we contemplate the addressing of rapid prototyping, the incorporation of author-oriented notations, the introduction of model-checking capabilities, the importation/exportation from/to standard specifications, and the deployment of the final applications.

$$\frac{\sigma_{\mathrm{STP}} = \mathbf{false} \;;;\; \mathbf{top}(\sigma_{\mathrm{CST}}) = \langle \_, i \rangle \;;;\; \mathbf{activity}(i,a,j) \in \sigma_{\mathrm{D}}}{\sigma \rightarrow \sigma\left[\mathbf{CST/pop}(\sigma_{\mathrm{CST}}), \mathbf{IB}/\sigma_{\mathrm{IB}} \cup \{\langle a, \langle i, j \rangle\rangle\}, \mathbf{O/out}(\mathbf{expose}(a), \sigma_{\mathrm{O}})\right]} \quad \textbf{activity-exposition}$$

$$\frac{\sigma_{\mathrm{STP}} = \mathbf{false} \;;;\; \mathbf{empty}(\sigma_{\mathrm{CST}}) = \mathbf{true} \;;;\; \langle \mathbf{attended}(a, As), I' \rangle = \mathbf{in}(\sigma_{\mathrm{I}}) \;;;\; \langle a, n \rangle \in \sigma_{\mathrm{IB}}}{\sigma \rightarrow \sigma\left[\mathbf{CST/push}(n, \sigma_{\mathrm{CST}}), \mathbf{I}/I', \mathbf{A}/\sigma_{\mathrm{A}} \cup As, \mathbf{IB}/\sigma_{\mathrm{IB}} - \{\langle a, n \rangle\}\right]} \quad \textbf{activity-attendance}$$

$$\frac{\sigma_{\mathrm{STP}} = \mathbf{false} \;;;\; \mathbf{top}(\sigma_{\mathrm{CST}}) = \langle \_, i \rangle \;;;\; \mathbf{achievement}(i,a,j) \in \sigma_{\mathrm{D}}}{\sigma \rightarrow \sigma[\mathbf{CST/push}(\langle i, j \rangle, \mathbf{pop}(\sigma_{\mathrm{CST}})), \mathbf{EST/push}(\begin{cases} \mathbf{true\ if}\ a \in \sigma_{\mathrm{A}} \\ \mathbf{false\ otherwise} \end{cases}, \sigma_{\mathrm{EST}})]} \quad \textbf{achievement-test}$$

$$\frac{\sigma_{\mathrm{STP}} = \mathbf{false} \;;;\; \mathbf{top}(\sigma_{\mathrm{CST}}) = \langle \_, i \rangle \;;;\; \mathbf{and}(i,j) \in \sigma_{\mathrm{D}} \;;;\; v_1 = \mathbf{top}(\sigma_{\mathrm{EST}}) \;;;\; v_o = \mathbf{top}(\mathbf{pop}(\sigma_{\mathrm{EST}}))}{\sigma \rightarrow \sigma\left[\mathbf{CST/push}(\langle i, j \rangle, \mathbf{pop}(\sigma_{\mathrm{CST}})), \mathbf{EST/push}(v_o \wedge v_1, \mathbf{pop}(\mathbf{pop}(\sigma_{\mathrm{EST}})))\right]} \quad \textbf{and-test}$$

$$\frac{\sigma_{\mathrm{STP}} = \mathbf{false} \;;;\; \mathbf{top}(\sigma_{\mathrm{CST}}) = \langle \_, i \rangle \;;;\; \mathbf{or}(i,j) \in \sigma_{\mathrm{D}} \;;;\; v_1 = \mathbf{top}(\sigma_{\mathrm{EST}}) \;;;\; v_o = \mathbf{top}(\mathbf{pop}(\sigma_{\mathrm{EST}}))}{\sigma \rightarrow \sigma\left[\mathbf{CST/push}(\langle i, j \rangle, \mathbf{pop}(\sigma_{\mathrm{CST}})), \mathbf{EST/push}(v_o \vee v_1, \mathbf{pop}(\mathbf{pop}(\sigma_{\mathrm{EST}})))\right]} \quad \textbf{or-test}$$

$$\frac{\sigma_{\mathrm{STP}} = \mathbf{false} \;;;\; \mathbf{top}(\sigma_{\mathrm{CST}}) = \langle \_, i \rangle \;;;\; \mathbf{not}(i,j) \in \sigma_{\mathrm{D}} \;;;\; v = \mathbf{top}(\sigma_{\mathrm{EST}})}{\sigma \rightarrow \sigma\left[\mathbf{CST/push}(\langle i, j \rangle, \mathbf{pop}(\sigma_{\mathrm{CST}})), \mathbf{EST/push}(\neg v, \mathbf{pop}(\sigma_{\mathrm{EST}}))\right]} \quad \textbf{not-test}$$

$$\frac{\sigma_{\mathrm{STP}} = \mathbf{false} \;;;\; \mathbf{top}(\sigma_{\mathrm{CST}}) = \langle \_, i \rangle \;;;\; \mathbf{alt}(i,j,k) \in \sigma_{\mathrm{D}}}{\sigma \rightarrow \sigma[\mathbf{CST/push}(\langle i, \begin{cases} j\ \mathbf{if}\ \mathbf{top}(\sigma_{\mathrm{EST}}) = \mathbf{true} \\ k\ \mathbf{otherwise} \end{cases} \rangle, \mathbf{pop}(\sigma_{\mathrm{CST}})), \mathbf{EST/pop}(\sigma_{\mathrm{EST}})]} \quad \textbf{alt-branching}$$

$$\frac{\sigma_{\mathrm{STP}} = \mathbf{false} \;;;\; \mathbf{top}(\sigma_{\mathrm{CST}}) = \langle \_, i \rangle \;;;\; \mathbf{fork}(i,j,k) \in \sigma_{\mathrm{D}}}{\sigma \rightarrow \sigma[\mathbf{CST/push}(\langle i, j \rangle, \mathbf{push}(\langle i, k \rangle, \mathbf{pop}(\sigma_{\mathrm{CST}})))]} \quad \textbf{fork-branching}$$

$$\frac{\sigma_{\mathrm{STP}} = \mathbf{false} \;;;\; \mathbf{top}(\sigma_{\mathrm{CST}}) = \langle \_, j \rangle \;;;\; \mathbf{join}(j,k) \in \sigma_{\mathrm{D}} \;;;\; \neg \exists W (\langle j, W \rangle \in \sigma_{\mathrm{J}})}{\sigma \rightarrow \sigma\left[\mathbf{J}/\sigma_{\mathrm{J}} \cup \{\langle j, \mathbf{pred}(\sigma_{\mathrm{D}}, j) \rangle\}\right]} \quad \textbf{join-activation}$$

$$\frac{\sigma_{\mathrm{STP}} = \mathbf{false} \;;;\; \mathbf{top}(\sigma_{\mathrm{CST}}) = \langle i, j \rangle \;;;\; \mathbf{join}(j,k) \in \sigma_{\mathrm{D}} \;;;\; \langle j, W \rangle \in \sigma_{\mathrm{J}} \;;;\; W - \{i\} \neq \varnothing}{\sigma \rightarrow \sigma\left[\mathbf{CST/pop}(\sigma_{\mathrm{CST}}), \mathbf{J}/(\sigma_{\mathrm{J}} - \{\langle j, W \rangle\}) \cup \{\langle j, W - \{i\} \rangle\}\right]} \quad \textbf{join-blocking}$$

$$\frac{\sigma_{\mathrm{STP}} = \mathbf{false} \;;;\; \mathbf{top}(\sigma_{\mathrm{CST}}) = \langle i, j \rangle \;;;\; \mathbf{join}(j,k) \in \sigma_{\mathrm{D}} \;;;\; \langle j, W \rangle \in \sigma_{\mathrm{J}} \;;;\; W - \{i\} = \varnothing}{\sigma \rightarrow \sigma\left[\mathbf{CST/push}(\langle j, k \rangle, \mathbf{pop}(\sigma_{\mathrm{CST}})), \mathbf{J}/\sigma_{\mathrm{J}} - \{\langle j, W \rangle\}\right]} \quad \textbf{join-unblocking}$$

$$\frac{\sigma_{\mathrm{STP}} = \mathbf{false} \;;;\; \mathbf{top}(\sigma_{\mathrm{CST}}) = \langle \_, i \rangle \;;;\; \mathbf{stop}(i) \in \sigma_{\mathrm{D}}}{\sigma \rightarrow \sigma[\mathbf{STP/true}, \mathbf{O/close}(\sigma_{\mathrm{O}})]} \quad \textbf{stop}$$

Fig. 6. Semantic rules for the 〈e-LD〉 sequencing language.

5.1. Rapid prototyping

The behavioral specification of the domain-specific language using structural operational semantics is amenable to supporting rapid prototyping in a straightforward way. The aim of rapid prototyping is to enhance the quality of the language specifications, since this quality will be decisive in the quality of the final applications. Indeed, if a good specification is provided, it will be possible to apply systematic techniques in order to derive reliable platforms, frameworks and/or parametric applications able to be customized using the languages. Therefore, we need to verify the correctness of the specification and, more important, to validate it according to the expectations of instructors and developers. For this purpose, rapid prototyping can be very helpful. In fact, before undertaking the implementation, developers can build a prototype of processor able to play descriptions which conform to the domain-specific language. In turn, the participation of instructors is also very valuable, as they can use these prototypes with real descriptions to decide on the suitability or unsuitability of the specification. As a consequence, errors can be detected and fixed in very early development stages. Therefore high-quality languages well adapted to the specific needs of relevant learning scenarios can be provided. This whole process also has advantages regarding production and maintenance of the final e-learning applications.

In our experiences we use the Prolog programming language (Sterling and Shapiro, 1994) because it is especially well suited for developing prototypes. The Prolog language facilitates the tasks involving the definition and manipulation of other languages. Indeed, it was designed with a primarily linguistic purpose, as a tool for natural language processing (Pereira and Warren, 1980). In addition, as described by Clément et al. (1986) and Sethi (1996), Prolog allows for the easy encoding of structural operational semantics. Another interesting feature is the fact that it is equipped with an extensible syntax, which can be extended by defining new operators. Finally, most Prolog implementations also incorporate very simple and elegant ways of doing concurrent processing based on co-routines. This feature will be very useful in isolating the purely behavioral aspects from those regarding interaction and presentation.

Therefore, Prolog sets an ideal framework to support rapid prototyping in our language-driven approach as it can speed up the construction of *definitional interpreters* for the domain-specific languages introduced during the language-driven development of e-learning applications. Definitional interpreters are very common in language engineering, and they are oriented to making the relevant behavioral features of the language explicit, but not to achieving efficiency or realistic deployment conditions (e.g. web-based execution). However, we will be able to implement it with very little effort in a few hours, and since it is systematically

obtained from the specification, we will also be able to update it easily if the language evolves or changes.

The high-level architecture of the definitional interpreters in our approach is induced by the typical nature of the operational semantics specification. This architecture, which is outlined in Fig. 7, includes:

- A *behavioral* layer that will encode the transition rules in the operational semantics specification. Following patterns similar to those discussed by Clément et al (1986) and Sethi (1996), it is possible to capture the language's transition relation with a `transition/1` predicate, whose argument is the encoding of a transition step $\sigma_0 \rightarrow \sigma_1$ (of a whole set of transition steps actually, since it is possible to use logical variables to represent state patterns, in the same way that syntactical variables are used in the semantic rules). Thus, there will be a one-to-one correspondence between semantics rules and Prolog clauses defining the `transition/1` predicate. Fig. 8a sketches the Prolog encoding of the **activity-exposition** semantic rule.
- An *interaction* layer that will model at a very basic level the interactions with the different participants in the learning process. The interaction layer includes a set of *interaction actions*, which deal with the presentation commands produced in the behavioral layers, collect the learner responses, and send them back to the behavioral layer. For this purpose, we represent interaction actions by an `interaction/3` predicate. The first argument of this predicate is the command itself, the second argument the current input, and the third argument the resulting input stream. In this way we need to provide a clause for each presentation command. In Fig. 8b we sketch the interaction action for the *expose* command, which is associated with the exposition of an activity to the learner.
- A *communication* layer that implements the stream operations used in the operational semantics. This layer promotes an explicit separation between the two previous ones. This separation is very relevant for developers. Indeed, final implementations will usually be web-based, and such implementations usually enforce such a separation between the client and the server side. Therefore, by also separating these aspects at the prototyping stages, developers will be able to test and refine the operational semantics in order to allow for appropriate interactive behavior. Regarding Prolog encoding, streams can be represented as open-ended lists. In addition, standard goal delaying Prolog mechanisms can be used in the provision of the operations.

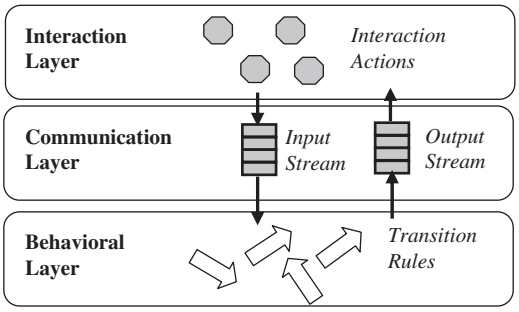

Fig. 7. High-level architecture of the prototypes for the definitional interpreters.

For more detail regarding rapid prototyping in the context of our language-driven approach see the work of Sierra et al. (2007a).

5.2. Provision of author-oriented notations

Author-oriented notations are very relevant to actively involving instructors in the whole development process. These notations can be textual or graphical. Also, it is possible to provide alternative notations for the same language, each one of them oriented to a different context or use. In particular:

- During prototyping the provision of a concrete syntax understandable to the instructors is mandatory to actively involving instructors during such an activity. Also, this syntax must be easy to implement in order to not increase the prototyping costs.
- A common practice in e-learning is to provide XML-based notations, which make possible the preparation of descriptions as documents marked with descriptive domain-specific markup languages. This conception of author-oriented notations based on descriptive markup is close to our previous work on a document-oriented approach to the production and maintenance of content-intensive and e-learning applications (Sierra et al., 2006b).

Finally, in order to facilitate the use of the languages by instructors, authoring scenarios encourage graphical notations (Martínez-Ortiz et al., 2007b).

- From the linguistic perspective promoted by the language-driven approach, the provision of these notations supposes:
- Defining suitable *concrete syntaxes*.
- Defining appropriate *translations* of these syntaxes to the language's abstract syntax.
- Implementing these translations in the context of suitable authoring tools.

a

```
transition (S0-> S1) :-
  component (S0, stp, false),
  component (S0, cst, CST),
  component(S0, d, D),
  top (CST, (_, I)),
  member(activity (I, A, J), D),
  %-----------------------
  pop (CST,CST1),
  updateComponent (S0, cst, CST1, SI1),
  component (S0, ib, IB),
  updateComponent (SI1, ib,
            [(a, (I,J))|IB], SI2),
  component (S0, o, O),
  out (O,expose(A), O1),
  updateComponent (SI2, o, O1, S1).
```

b

```
interaction(expose (A), In, NIn) :-
 showActivity (A),
 readAchievements (As),
 out (In,attended (A, As), NIn).
```

Fig. 8. Prolog encoding of (a) *activity-exposition* transition rule, and (b) the interaction action associated with the *expose* command. Definition of the predicates used in the right side of the clauses are omitted for the sake of conciseness.

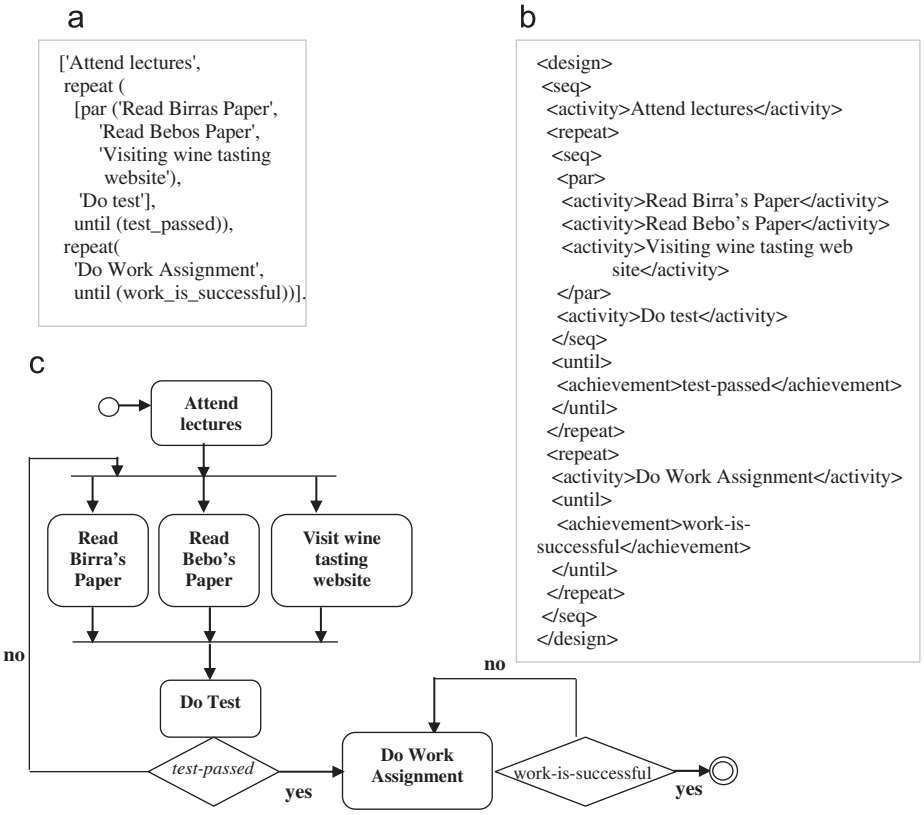

Fig. 9. Encoding the example of Fig. 2 using different concrete syntaxes: (a) a Prolog-based one; (b) an XML-based one; (c) a graphical one.

Also, according to well-established language engineering principles, it is very important to separate the specification of each of these aspects:

- At a purely descriptive level, concrete syntaxes (commonly known as *bindings* in e-learning standardization efforts) can be defined by using a plethora of grammar formalisms (Klint et al., 2005). In particular, prototyping-oriented notations can be defined as *regular tree grammars* (Comon et al., 2007) adopting a suitable form of the BNF notation. XML-based markup languages can be defined as a suitable schema language (e.g. DTDs or XML Schema) (Makoto et al., 2005). Graphical notations in turn can be defined using suitable visual grammar formalisms (Marriott et al., 1999).
- In turn, translation can be specified using standard mechanisms in language engineering. In particular, the *attribute grammar* paradigm (Paakki, 1995) is especially well suited to this purpose. This paradigm can also be applied to describing contextual constraints not directly describable with the formalism used to define the notation.
- Finally, implementation will vary with the notation and the application context. For instance, during prototyping, syntax can be embedded in Prolog using the aforementioned feature of user-defined operators. Considering the descriptions in the language as Prolog terms, translation is reduced to a term manipulation process, which in turn is reduced to a plane Prolog programming task. Regarding XML-based markup languages, they can be processed using some of the well-known XML processing frameworks (Birbeck et al., 2001). Also, in this case we have realized the feasibility of connecting event-oriented frameworks (e.g. SAX) with standard compiler-generation (e.g. JavaCC or Yacc-like) tools, following a pattern similar to that suggested by Stanchfield (2009) in relation to the ANTLR compiler-generation framework (Parr, 2007). The idea is to use SAX as a lexical scanning framework which can be connected to a translator automatically generated from a grammar-based specification. This grammar-based nature of the cited compiler construction environment contributes to decreasing the distance between implementation and specification and, therefore, to leveraging the overall development and maintenance costs (Sarasa-Cabezuelo et al., 2008). Finally, regarding graphical syntaxes, the meta-modeling support in the Eclipse framework, along with its associated edition facilities, constitutes a pragmatic and cost-effective solution to implementing them (Steinber et al., 2003).

Regarding our case study, in Fig. 9 we describe the example in Fig. 2 with three different alternative concrete syntaxes. In Fig. 9a, we use a prototyping-oriented notation. In Fig. 9b we use an XML-based one. Finally, in Fig. 9c we use a graphical oriented one, which is based on the UML's activity diagrams (Booch et al., 1998), and which also resembles the typical notations used in the description of workflows during the automation of business processes (Aalst and Kees, 2004). Moreover, the graphical notation used in Fig. 9c is a simplification of ⟨e-LD⟩'s actual graphical notation (Martínez-Ortiz et al., 2008a).

5.3. Model checking of properties

We consider that a good authoring support not only has to rely on a user-friendly notation and editing environment, it also provide authors with the automation of different tasks, such as the verification of properties in the authored descriptions. Some of

$$\frac{\sigma_{\mathrm{STP}} = \mathbf{false} \;;;\; \mathbf{top}(\sigma_{\mathrm{CST}}) = \langle \_, i \rangle \;;;\; \mathbf{activity}(i, a, j) \in \sigma_{\mathrm{D}}}{\sigma \rightarrow \sigma \left[\mathbf{CST/pop}(\sigma_{\mathrm{CST}}), \mathbf{IB}/\sigma_{\mathrm{IB}} \cup \{\langle a, \langle i, j \rangle \rangle\} \right]} \quad \textbf{activity-exposition}$$

$$\frac{\sigma_{\mathrm{STP}} = \mathbf{false} \;;;\; \mathbf{empty}(\sigma_{\mathrm{CST}}) = \mathbf{true} \;;;\; \langle a, n \rangle \in \sigma_{\mathrm{IB}} \;;;\; \mathbf{achievements}(a, As) \in \sigma_{\mathrm{D}} \;;;\; As' \subseteq As}{\sigma \rightarrow \sigma \left[\mathbf{CST/push}(n, \sigma_{\mathrm{CST}}), \mathbf{A}/\sigma_{\mathrm{A}} \cup As', \mathbf{IB}/\sigma_{\mathrm{IB}} - \{\langle a, n \rangle\} \right]} \quad \textbf{activity-attendance}$$

Fig. 10. Modification of the semantic rules for the 〈e-LD〉 sequencing language in order to abstract its operational semantics for model-checking purposes.

these properties are structural, and therefore they can be statically checked in a way similar to the syntax or to the contextual constraints of a conventional programming language. However, properties related to states during runtime still cannot be verified in such a way, since this verification requires considering the possible ways in which the resulting system or application can evolve from a particular state. *Model checking* (Clarke et al., 2000) is a technique that allows for this kind of verifications. In model checking, properties to be checked are described using a suitable *temporal logic* (Emerson, 1990). The goal of the checking process is then to establish where the application's state space is a *model* of (i.e. satisfies) the temporal properties, and, in the case of finding a violation, to document it in the form of a suitable execution trace.

In order to take advantage of model checking in the context of our language-driven approach, the following steps need to be addressed:

- To substitute interactive behavior by non-deterministic choice in the operational semantics. Indeed, since the goal is to explore the possible executions of the application, for each potential learner's interaction it is sufficient to try all the possible learner's responses.
- To design a suitable assertion language for describing the properties. Since the complexity of the checking process will depend on this language, a reasonable trade-off between expressivity and the impact on the complexity of the final checking process must also be taken into account. In particular, the CTL (*Computation-tree logic*) language, which we have adopted in some of our experiences, allows for a checking process whose time complexity is polynomial in the size of the state space (Clarke et al., 1986; Emerson, 1990). Yet this language is expressive enough to describe many interesting properties.
- To distinguish *relevant* states for property checking. Indeed, since the complexity of the checking process will also depend on the size of the state space, it will also be important to decrease the number of states by distinguishing this subset of relevant ones.
- To provide model-checking support. For some languages it will be possible to reuse pre-existing model checkers. For instance, in the context of the language-oriented development of educational adventure videogames described by Moreno-Ger et al. (2007), we have used the NuSMV model checker (Cimatti et al., 2000), which makes use of an efficient technique called *symbolic model checking* (Burch et al., 1992) to deal with huge state spaces (the results are reported by Moreno-Ger et al., 2009). Other works that use a similar approach are reported by Baldoni et al. (2007) and Baldoni and Marengo (2007). While this possibility leverages the overall cost of providing this support, it supposes facing the encoding of the operational semantics in the reused model-checker's specification language. It also hinders the implementation of some domain-specific optimizations during the searching of the state space. Finally, it also requires providing a way of presenting the instructors with counterexamples for properties which are violated, counterexamples which will be given in terms of the encoding instead of in terms of the original domain-specific language. Due to these shortcomings, for other languages the explicit provision of a dedicated model-checking engine could be more convenient.

```
DEF before (Fo, F1) = not E (not F1 U Fo)
before (expose (Do Work Assignment),
    expose (Do Test))
```

Fig. 11. Examples of definitions using the assertion language. First the *before* temporal operator is defined, with the intended meaning 'F1 must hold before Fo holds'. Then a property is formalized stating 'The *Do Test* activity must be exposed before the *Do Work Assignment* activity is exposed'.

By following the steps described above it is possible to build a framework supporting model checking for each particular domain-specific language. In order to effectively use this framework, instructors and developers must set it up for each particular project or family of projects by deciding the relevant properties to be checked. For this purpose, instructors informally state these properties, and developers formalize them using the assertion language. Once the properties are established, instructors take control. They author the descriptions, automatically detect potential problems as violations of the properties, examine the counterexample traces, and use those to solve the problems detected. The construction of the domain-specific model-checking frameworks themselves can be exemplified with our 〈e-LD〉 sequencing language:

- The substitution of interaction by non-deterministic choice supposes dropping the input and output streams from the computation states. In order to anticipate the possible learner's responses, one needs to add a predicate of the form **achievements**(*a*, *As*) to the abstract description, declaring the possible achievements *As* for each activity *a*. In Fig. 10 we show the modifications in the operational semantics required to achieve the substitution. These modifications only affect the rules **activity-exposition** and **activity-attendance**. The other rules remain unmodified.
- The assertion language is standard CTL with atomic propositions, ***expose***(*a*), where *a* is the name of an activity, ***achieved***(*a*), where *a* is the name of an achievement, and ***stopped***. The intended meanings of these propositions are the obvious ones. The assertion language also includes a basic abstraction mechanism, which allows for the definition of new operators. In Fig. 11 we show an example of the use of this language.
- Relevant states will be those produced by the new **activity-exposition** and **activity-attendance** rules, as well as by the **stop** rule, since those states are where changes in the truth value of the assertion language's basic propositions hold.

- Implementation is carried out using a dedicated model checker which is driven by the adjusted operational semantics specification. The inputs to this model checker are the learning flow sequencing specification, the declaration of the possible achievements for each activity, and the properties to be checked (expressed in the assertion language). Then it systematically explores the state space induced by the abstract semantics and the particular specification, it maintains the basic propositions which are true in each state, and it uses an on-the-fly version of the classical model-checking algorithm presented in Clarke et al. (1986) to check the properties. It also uses compression and hashing techniques to manage the states explored and to save space, and it focuses only on the relevant states. When it discovers a violation of a property, it produces an execution trace of the learning flow describing this violation. The current algorithm is able to deal with several million states in a typical modern personal computer, which is enough for most of the practical applications.

5.4. Exportation and importation

One of the main aims of standardization efforts in e-learning is to facilitate the interoperability of heterogeneous applications and systems, as previously stated. For this purpose, the different aspects of an e-learning system can be expressed using standard descriptive languages such as the IMS specifications mentioned at the beginning of this paper. Nevertheless, standard languages and notations can be very general for a specific problem domain, and they can also be more difficult to use from an authoring perspective. For this purpose, suitable *import* and *export* processes can be defined relating domain-specific notations and standard ones. Importation consists of producing specifications in the domain-specific language from specifications in a different (possibly standard) external language. On the other hand, exportation consists of the inverse step: producing specifications in the external language from specifications in the local one.

The language engineering methods applied during importation and exportation are similar to those used during the provision of authoring notations, since suitable translations between notations must be provided. Nevertheless, in this case one needs to face a lack of expressive power in the target notation. This problem arises when the target notation is not able to express all the features that can be described using the source one. Therefore, the results of the translation process will be:

- A (possibly partial) specification in the target notation.
- A (possibly empty) report of incidences discovered during translation.

We have followed this approach in the work reported by Sierra et al. (2007b), and Sierra and Fernández-Valmayor (2007). The first of these papers describes an agenda-based importation system for IMS Content Packaging. The second one reports how a flexible importation/exportation system is added to a system for building repositories of learning objects in specialized domains. Also, as reported by Martínez-Ortiz et al. (2008b) the approach is a core feature of our ⟨e-LD⟩ environment. In this context translation is conceived as a semi-automatic process, since instructors can use the incidence report to *refactor* the original specification in terms of the new expressive means. Also, due to the differences between the languages, the translation processes are not ensured to be complete, but only to produce *satisfactory* approximations to the original specifications. Finally, developers also play an active role during refactoring. On one hand, they can change or fine-tune the underlying software infrastructure to make possible the desired refactoring. On the other hand, they can implement evolutions of the domain-specific language to accommodate the expressive needs of the instructors not taken into account in the current version of such a language.

In order to exemplify the exportation and importation aspects, consider exportation from our ⟨e-LD⟩ sequencing language to IMS LD, and importation from IMS LD to this language:

- Exportation will produce a design with a single role that contains a method with a single act. This act will encode the sequencing specification. However, the process cannot be performed directly because of the impossibility of changing the state of an activity from *completed* to *uncompleted* in the IMS LD language. In order to approximate designs with loops, one needs to *unfold* these loops in a pre-established number of iterations. In ⟨e-LD⟩ a wizard helps the user with this unfolding process. The result is an IMS LD *approximation* of the original design. Therefore, specifications in the ⟨e-LD⟩ language can be understood as *templates* that can be used to generate an unlimited number of IMS LD designs. In addition, an incidence report will be generated describing those aspects that must be revised/completed (e.g. the activities cloned during the unfolding process, or the activities added to end the last iteration when the termination condition has failed). In Fig. 12a we illustrate this process. We sketch a simple ⟨e-LD⟩ sequencing specification (actually, a fragment of the example specification in Fig. 2), and we show the result of unfolding two iterations of the loop. After being revised by the instructors, this specification will be directly describable in IMS LD (Martínez-Ortiz et al., 2009).
- The importation process will produce a separate initial sequencing specification for each role.[1] In addition, the process will be hindered by the different sequencing paradigms, such as the one we indicated in Section 3. The incidence report will use conditions and notifications to annotate the initial sequencing specifications with the different aspects to be revised. This leads to a *learning design reengineering* approach, where instructors and developers can then work on refactoring the original design in an ⟨e-LD⟩ compliant one, which can be re-exported into an improved IMS LD design (Martínez-Ortiz et al., 2008b). The process is illustrated in Fig. 12b. This illustration includes a schematic representation of a level B IMS LD design, which actually captures a fragment of the example introduced in Section 3. The importation process takes the IMS LD level A part as an initial skeleton, and then uses the conditions to produce the incidence report. This report annotates the skeleton with aspects to be taken into account during refactoring. Notice how the refactoring process must also involve developers, since the learning flow sequencing depends on a real-valued property. Therefore, the actual implementation of the *Do Work* activity must be revised by the developers to produce an equivalent achievement instead. Meanwhile, developers could decide to extend the domain-specific linguistic support with mapping capabilities to facilitate this kind of fine-tuning.

5.5. Deployment

The deployment infrastructures that result from the language-driven approach are amenable to being architected following the MVC pattern. Semantic rules in the language's operational semantics are useful in order to structure the application's

[1] In the actual ⟨e-LD⟩ language these specifications are not separated, since the *real* language is able to deal with multiple roles.

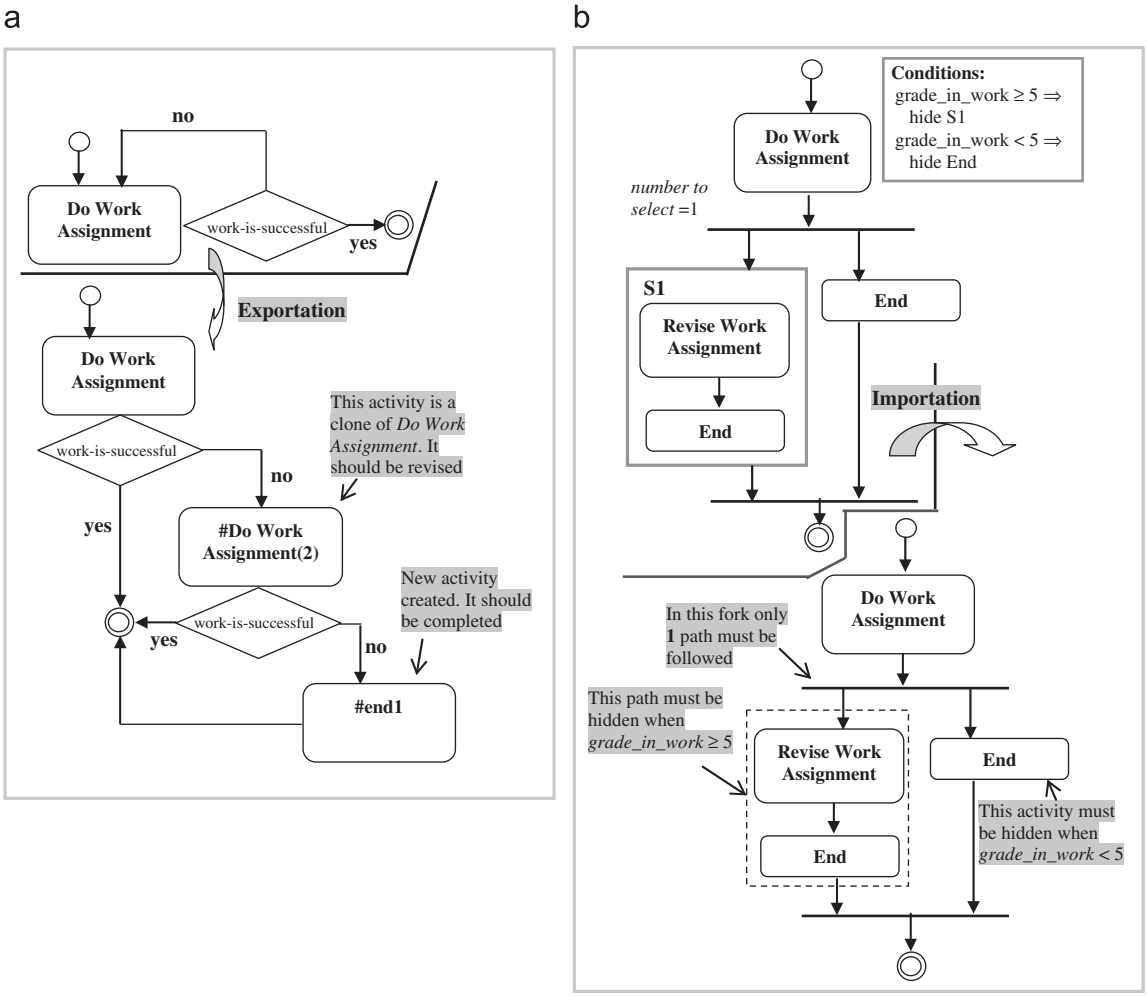

Fig. 12. (a) An example of the exportation process; (b) an example of the importation process.

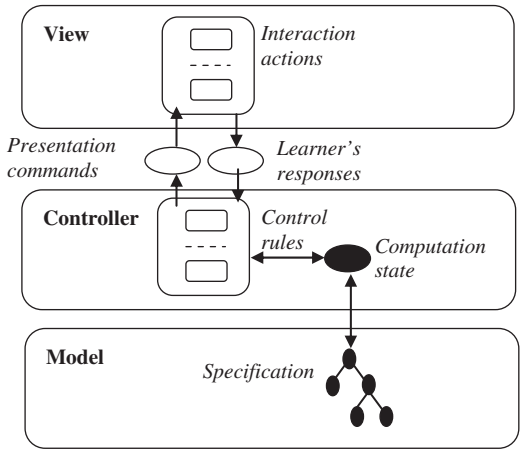

Fig. 13. MVC organization of the final deployment.

controller. The language's structural characterization is in turn useful to structuring the application's model. Finally, the structure of the different streams involved in the semantics is useful to identifying, for instance, the basic presentation commands and user actions. In Fig. 13 we sketch the resulting organization:

- The view contains suitable *interactions actions*, which are components governing the presentation of information to the learner, as well as the processing of the interaction results.
- The stream-based communication between the controller and the view is refined by function/method invocations. Input and output data is communicated using appropriate *learner responses* and *presentation commands*.
- The controller is configured with a suitable set of *control rules*, which are implementations of the semantic rules. The computation state is in turn kept as a globally available structure which in turn points to information elements in the model.
- Finally the model is an appropriate representation of a description in the domain-specific language.

For more details regarding implementation strategies in the context of the language-driven approach see the works of Sierra et al. (2008a, 2008b).

6. Conclusions and future work

In this paper we have presented how language engineering can be applied in order to improve and systematize the development of e-learning applications. The core of this approach is the structural and operational specification of a domain-specific language used to describe key aspects of the final learning application. This specification is governed by well-established language engineering methods, techniques and tools, which also

apply to several other enabled activities. In particular, we have illustrated how rapid prototyping, provision of author-oriented notations, verification of properties, connection with standards through exportation and importation processes, and deployment of the final running applications can be achieved, driven by the domain-specific language and its specification. As has been tested in several developments, and as we have discussed throughout the paper, this approach promotes an innovative way of collaboration between instructors and developers during the design and development of e-learning applications.

Currently we are enhancing and further systematizing the approach in the context of our 〈e-LD〉 environment. We are also experimenting with some aspects of the approach in different application domains. In particular, we are applying the ideas regarding model-checking in the development of multi-agent systems based on activity theory (Fuentes-Fernández et al., 2007). As a future work we want to provide better meta-linguistic support to help in the maintenance and evolution of the different linguistics specifications and by-products during the achievement of the language design and the other enabled activities.

Acknowledgements

The Spanish Committees of Science and Innovation and of Industry, Tourism and Commerce (Projects TIN2005-08788-C04-01, Flexo-TSI-020301-2008-19 and TIN2007-68125-C02-01) and the Regional Government/Complutense University of Madrid (research group 921340), as well as the Santander/UCM project PR34/07-15865 and the EU Alfa project CID (II-0511-A) have partially supported this work.

References

Aalst W, Kees H. Workflow management: models, methods, and systems. MA: MIT Press; 2004 320p.

Aho AV, Lam MS, Sethi R, Ullman JD. Compilers: principles, techniques and tools, 2nd ed. Boston, MA, USA: Addison-Wesley; 2007 1009p.

Allen CW. Overview and evolution of the ADDIE training system. Advances in Developing Human Resources 2006;8(4):430–41.

Baldoni M, Baroglio C, Brunkhorst I, Marengo E, Patti V. Reasoning-based curriculum sequencing and validation: integration in a service-oriented architecture. In: Duval E, Klamma R, Wolpers M, editors. Creating new learning experiences on a global scale. 2nd European conference on technology enhanced learning (EC TEL 2007); 2007 September 17–20, Crete, Greece. Berlin: Springer; 2007. p. 426–31.

Baldoni M, Marengo E. Curriculum model checking: declarative representation and verification of properties. In: Duval E, Klamma R, Wolpers M, editors. Creating new learning experiences on a global scale. 2nd European conference on technology enhanced learning (EC TEL 2007); 2007 September 17–20, Crete, Greece. Berlin: Springer; 2007. p. 432–7.

Birbeck M, Kay M, Livingstone S, Mohr SF, Pinnock J, Loesgen B, et al. Professional XML, 2nd ed. Birmingham: Wrox Press; 2001 1159p.

Booch G, Rumbaugh J, Jacobson I. The unified modeling language user guide. Reading, MA: Addison Wesley; 1998 482p.

Bork A. Personal computers for education. New York, NY, USA: Harper & Row Publishers, Inc.; 1985 179p.

Burch JR, Clarke EM, McMillan KL, Dill DL, Hwang LJ. Symbolic model checking: 10^20 states and beyond. Information and Computation 1992;98(2):142–70.

Burgos D, Moreno-Ger P, Sierra JL, Fernández-Manjón B, Koper R. Authoring game-based adaptive units of learning with IMS learning design and 〈e-Adventure〉. International Journal of Learning Technology 2007a;3(3):252–68.

Burgos D, Tattersall C, Koper R. Representing adaptive and adaptable units of learning. How to model personalized eLearning in IMS learning design. In: Fernández-Manjón B, Sánchez-Pérez JM, Gómez-Pulido JA, Vega-Rodríguez MA, Bravo-Rodríguez J, editors. Computers and education: E-learning-from theory to practice. Berlin: Springer; 2007b. p. 41–56.

Cimatti A, Clarker E, Giunchiglia F, Roveri M. NUSMV: a new symbolic model checker. International Journal on Software Tools for Technology Transfer 2000;2(4):410–25.

Clarke EM, Emerson EA, Sistla AP. Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems 1986;8(2):244–63.

Clarke EM, Grumberg O, Peled DA. Model checking. MA: MIT Press; 2000 314p.

Clément D, Despeyroux J, Despeyroux T, Hascoet L, Kahn G. Natural semantics on the computer. In: Fuchi K, Nivat M, editors. Proceedings of the France–Japan AI and CS symposium, Japan, 1986. p. 49–89.

Comon H, Dauchet M, Gilleron R, Jacquemanrd F, Lugiez D, Löding C, et al. Tree automata techniques and applications [Online]. 2007 October 17. 262p. Retrieved January 19, 2009 from: 〈http://tata.gforge.inria.fr/〉.

Deursen A, Klint P, Visser J. Domain-specific languages: an annotated bibliography. ACM SIGPLAN Notices 2000;35(6):26–36.

Emerson EA. Temporal and modal logic. In: Leeuwen JV, editor. Handbook of theoretical computer science, vol. B: formal models and semantics. MA: MIT Press; 1990. p. 995–1072.

Fernández-Manjón B, Fernández-Valmayor A. Improving world wide web educational uses promoting hypertext and standard general markup languages. Education and Information Technologies 1997;2(3):193–206.

Friedman D, Wand M, Hayes CT. Essentials of programming languages, 2nd ed. MA: MIT Press; 2001 389p.

Friesen N. Interoperability and learning objects: an overview of e-Learning standardization [Online]. Interdisciplinary Journal of Knowledge and Learning Objects 2005;1:23–31 Retrieved January 19, 2009 from 〈http://ijklo.org/Volume1/v1p023-031Friesen.pdf〉.

Fuentes-Fernández R, Gómez-Sanz J, Pavón J. Managing contradictions in multi-agent systems. IEICE Transactions on Information and Systems 2007;E90-D(8):1243–50.

Goodwill J, Hightower R. Professional jakarta struts. Indianapolis: Wiley; 2003 429p.

Ibrahim B. Software engineering techniques for CAL. Education and Computing 1989;5(4):215–22.

IMS. IMS learning design information model version 1.0 final specification [Online]. 2003a January 20. Retrieved January 19, 2009 from: 〈http://www.imsglobal.org/learningdesign/ldv1p0/imsld_infov1p0.html〉.

IMS. IMS Simple sequencing information and behavior model version 1.0 final specification [Online]. 2003b March 3. Retrieved January 19, 2009 from: 〈http://www.imsglobal.org/simplesequencing/ssv1p0/imsss_infov1p0.html〉.

IMS. IMS Content packaging information model version 1.1.4 final specification [Online]. 2004 October 4. Retrieved January 19, 2009 from: 〈http://www.imsglobal.org/content/packaging/cpv1p1p4/imscp_infov1p1p4.html〉.

IMS. IMS learner information package summary of changes version 1.0.1 final specification [Online]. 2005 January 5. Retrieved January 19, 2009 from: 〈http://www.imsglobal.org/profiles/lipv1p0p1/imslip_sumcv1p0p1.html〉.

IMS. IMS question and test interoperability assessment test, section, and item information model Version 2.1 public draft revision 2 specification [Online]. 2006 June 8. Retrieved January 19, 2009 from: 〈http://www.imsglobal.org/question/qtiv2p1pd2/imsqti_infov2p1pd2.html〉.

Klint P, Lämmel R, Verhoef C. Toward an engineering discipline for grammarware. ACM Transactions on Software Engineering and Methodology 2005;14(3):331–80.

Koper R, Manderveld J. Educational modelling language: modelling reusable, rich and personalised units of learning. British Journal of Educational Technology 2004;35(5):537–51.

Koper R, Olivier B. Representing the learning design of units of learning. Journal of Educational Technology & Society 2004;7(3):97–111.

Krasner GE, Pope TS. A description of the model-view-controller user interface paradigm in the smalltalk 80 system. Journal of Object Oriented Programming 1988;1(3):26–49.

Makoto M, Lee D, Mani M, Kawaguchi K. Taxonomy of XML schema languages using formal language theory. ACM Transactions on Internet Technology 2005;5(4):660–704.

Marriott K, Meyer B, Wittenburg KBA. Survey of visual language specification and recognition. In: Marriot K, Meyer B, editors. Visual language theory. Berlin: Springer; 1999. p. 5–85.

Martínez-Ortiz I, Moreno-Ger P, Sierra JL, Fernández-Manjón B. Supporting the authoring and operationalization of educational modelling languages [Online]. Journal of Universal Computer Science 2007a;13(7):938–47 Retrieved January 19, 2009 from: 〈http://dx.doi.org/10.3217/jucs-013-07-0938〉.

Martínez-Ortiz I, Moreno-Ger P, Sierra JL, Fernández-Manjón B. Educational modeling languages: a conceptual introduction and a high-level classification. In: Fernández-Manjón B, Sánchez-Pérez JM, Gómez-Pulido JA, Vega-Rodríguez MA, Bravo-Rodríguez J, editors. Computers and education: E-learning-from theory to practice. Berlin: Springer; 2007b. p. 27–40.

Martínez-Ortiz I, Moreno-Ger P, Sierra JL, Fernández-Manjón B. A flow-oriented visual language for learning designs. In: Li F, Zhao J, Shih TK, Lau R, Li Q, McLeod D, editors. Advances in web based learning-ICWL 2008. 7th International conference on web-based learning (ICWL 2008). 2008 August 20–22, Jinhua, China. Berlin: Springer; 2008a. p. 486–96.

Martínez-Ortiz I, Sierra JL, Fernández-Manjón B. Enhancing reusability of IMS LD units of learning: the e-LD approach. In: Proceedings of 8th IEEE international conference on advanced learning technologies (ICALT 2008). 2008 July1–5, Santander, Spain. Washington DC, USA: IEEE Computer Society; 2008b. p. 402–4.

Martínez-Ortiz I, Sierra JL, Fernández-Manjón B. Translating e-learning flow-oriented activity sequencing descriptions into rule-based designs. In: Proceedings of the 6th international conference on information technology: new generations (ITNG 2009). 2009 April 27–29, Las Vegas, USA. Washington DC, USA: IEEE Computer Society; Forthcoming 2009.

Mernik M, Heering J, Sloane AM. When and how to develop domain-specific languages. ACM Computing Surveys 2005;37(4):316–44.

Molenda M. In search of the elusive ADDIE model. Performance Improvement Journal 2003;42(5):34–6.

Moreno-Ger P, Martínez-Ortiz I, Sierra JL, Fernández-Manjón B. A descriptive markup approach to facilitate the production of e learning contents. In: Proceedings of the 6th international conference on advanced learning technologies (ICALT 2006). 2006 July 5–7, Kerkrade, The Netherlands. Washington DC, USA: IEEE Computer Society; 2006. p. 19–21.

Moreno-Ger P, Sierra JL, Martínez-Ortiz I, Fernández-Manjón B. A documental approach to adventure game development. Science of Computer Programming 2007;67(1):3–31.

Moreno-Ger P, Fuentes-Fernández R, Sierra JL, Fernández-Manjón B. Model-checking for adventure videogames. Information and Software Technology 2009;51(3):564–80.

Mosses PD. Modular structural operational semantics. Journal of Logic and Algebraic Programming 2004;60–61:195–228.

Mosses PD. Formal semantics of programming languages: an overview. Electronic Notes in Theoretical Computer Science 2006;148(1):41–73.

Paakki J. Attribute grammar paradigms—A high-level methodology in language implementation. ACM Computing Surveys 1995;27(2):196–255.

Parr T. The definitive ANTLR reference: building domain-specific languages. Pragmatic Bookshelf; 2007 376p.

Pereira FCN, Warren DHD. Definite clause grammars for language analysis-a survey of the formalism and a comparison with augmented transition networks. Artificial Intelligence 1980;13(3):231–78.

Plotkin GD. A structural approach to operational semantics. Journal of Logic and Algebraic Programming 2004;60–61:17–139.

Sarasa-Cabezuelo A, Navarro-Iborra A, Sierra JL, Fernández-Valmayor A. Building a syntax directed processing environment for XML documents by combining SAX and JavaCC. In: Proceedings of the 3rd international workshop on XML Data Management Tools & Techniques (XANTEC 2008) - DEXA 2008 workshops. 2008 September 1–5, Turin, Italy. Washington DC, USA: IEEE Computer Society; 2008. p. 256–60.

Sethi R. Programming languages: concepts and construct. Boston, MA, USA: Addison Wesley; 1996 640p.

Sierra JL, Fernández-Valmayor A, Guinea M. Exploiting author-designed domain-specific descriptive markup languages in the production of learning content. In: Proceedings of the 6th international conference on advanced learning technologies (ICALT 2006). 2006 July 5–7, Kerkrade, The Netherlands. Washington DC, USA: IEEE Computer Society; 2006a. p. 515–19.

Sierra JL, Fernández-Valmayor A, Fernández-Manjón B. A document-oriented paradigm for the construction of content-intensive applications. Computer Journal 2006b;49(5):562–84.

Sierra JL, Fernández-Valmayor A, Guinea M, Hernánz H. From research resources to virtual objects: process model and virtualization experiences. Journal of Educational Technology & Society 2006c;9(3):56–68.

Sierra JL, Fernández-Valmayor A, Fernández-Manjón B. How to prototype an educational modeling language. In: Proceedings of the IX International symposium on computers in education. 2007 November 14–16, Porto, Portugal, 2007a. p. 97–102.

Sierra JL, Fernández-Valmayor A. Universalizing chasqui repositories with a flexible importation/exportation system. In: Fernández-Manjón B, Sánchez-Pérez JM, Gómez-Pulido JA, Vega-Rodríguez MA, Bravo-Rodríguez J, editors. Computers and education: E-learning-from theory to practice. Berlin: Springer; 2007. p. 99–110.

Sierra JL, Moreno-Ger P, Martínez-Ortiz I, Fernández-Manjón B. A highly modular and extensible architecture for an integrated IMS-based authoring system: The 〈e-Aula〉 experience. Software-Practice & Experience 2007b; 37(4):441–61.

Sierra JL, Fernández-Manjón B, Fernández-Valmayor A. A language-driven approach for the design of interactive applications. Interacting with Computers 2008a;20(1):112–27.

Sierra JL, Fernández-Manjón B, Fernández-Valmayor A. Language-driven development of web-based learning applications. In: Leung H, Li F, Lau F, Li Q, editors. Advances in web based learning—ICWL 2007. 6th International conference on web-based learning, 2007 August 15–17, Edinburgh, United Kingdom. Berlin: Springer; 2008b. p. 520–31.

Sierra JL, Fernández-Valmayor A, Fernández-Manjón B. From documents to applications using markup languages. IEEE Software 2008c;25(2):68–76.

Stahl T, Voelter M, Czarnecki K. Model-driven software development, technology, engineering, management. Chichester, West Sussex, UK: Wiley; 2006 444p.

Stanchfield S. ANTXR: easy XML parsing, based on the ANLR parser generator [Online]. Retrieved January 19, 2009 from: 〈http://javadude.com/tools/antxr/index.html〉.

Steinber D, Budinsky F, Paternostro M, Merks E. EMF: eclipse modeling framework, 2nd ed. Boston, MA, USA: Addisson-Wesley; 2003 744p.

Sterling L, Shapiro E. The art of prolog, second edition: advanced programming techniques. MA, USA: MIT Press; 1994 549p.