A Language-Driven Approach for the Design of Interactive Applications

José-Luis Sierra, Baltasar Fernández-Manjón, Alfredo Fernández-Valmayor

PII: S0953-5438(07)00076-8
DOI: 10.1016/j.intcom.2007.09.001
Reference: INTCOM 1621

To appear in: *Interacting with Computers*

Received Date: 18 April 2007
Revised Date: 1 August 2007
Accepted Date: 2 September 2007

Please cite this article as: Sierra, J., Fernández-Manjón, B., Fernández-Valmayor, A., A Language-Driven Approach for the Design of Interactive Applications, *Interacting with Computers* (2007), doi: 10.1016/j.intcom.2007.09.001

Accepted in Interacting with computers

A Language-Driven Approach for the Design of Interactive Applications

José-Luis Sierra*, Baltasar Fernández-Manjón, Alfredo Fernández-Valmayor

Dpto. Ingeniería del Software e Inteligencia Artificial. Fac. Informática. Universidad Complutense. 28040, Madrid. Spain.
{jlsierra,balta,valmayor}@fdi.ucm.es
Tel. +34913947548. Fax. +34913947547

Abstract. In this paper we propose a language-driven approach for the high-level design of interactive applications architected according to the model-view-controller pattern. The approach is especially well-suited for applications that incorporate contents with sophisticated structures, and whose interactive behavior is driven by these structures. In our approach we characterize the structure of the contents stored in the applications' models with suitable domain-specific languages. Then we characterize the interactive behavior of these applications by assigning suitable operational semantics to these languages. The resulting designs are amenable to support rapid prototyping, exploration and early discovery of application features, systematic implementation using standard web-based technologies, and rational collaboration processes between domain-experts and developers during production and maintenance. We exemplify the approach in the e-learning domain with a system for the production of Socratic tutors.

Keywords: language-driven development, domain-specific language, model-view-controller, e-learning, document-oriented approach

1. Introduction

Interactive applications structured according to the model-view-controller (MVC) design pattern (Krasner and Pope, 1988) have attracted great interest in the last few years. Indeed, this architecture is typically used in organizing modern web-based applications (Draheim et al, 2003). Some of these applications can exhibit a high degree of complexity, both in those aspects regarding the structure of the contents as well as in those aspects regarding interactions with the final users. An extreme example can be found in many web-based e-learning applications (Collier, 2002), which integrate bodies of contents structured according to sophisticated and evolving standards (Friesen, 2005) and which rely on complex interaction strategies represented by elaborated pedagogical designs (Koper and Tattersall, 2005).

In recent years we have been heavily involved in the development of several web-based e-learning applications for various purposes. In all these cases we have extensively used the MVC architecture for structuring the final applications. One of the main lessons learned from these experiences (some of which will be briefly presented in this paper) is the importance of adopting a well-principled and rigorous approach to modeling the structural and behavioral aspects of these applications in the very first stages of development. For this purpose, we have found it natural to adopt a *language-centered approach*. In this approach, we start by identifying the *language* that characterizes the contents' structure relevant for enabling the desired interactive behavior. Thus, modeling such behavior is naturally addressed by assigning suitable operational semantics to such a language. This linguistic approach has two important features; first, it naturally leads to rational process models that determine the collaboration between domain-experts and developers during the design, construction and maintenance of this kind of applications (Sierra et al, 2005a; Sierra et al, 2006b); and second, it enables an implementation approach where main elements in the defined language align with main elements in the MVC architecture.

The structure of the paper is as follows. In section 2 we present some preliminaries. Section 3 gives an overview of the approach. Section 4 deals with the structural aspects of the design. Section 5 addresses the specification of behavior. Section 6 provides some implementation guidelines. Section 7 discusses some related work. Finally, section 8 presents the conclusions and some lines of future work.

2. Preliminaries

In this section we give a short introduction to language-driven software development (section 2.1). We also outline an example of application, an educational tutoring system, which we will use to illustrate the different aspects of our approach (section 2.2). Finally we describe the schema for MVC applications that will serve as reference throughout the paper (section 2.3).

2.1. Language-Driven Software Development

Language-driven approaches conceive software development as the design, implementation and maintenance of *domain-specific languages* specially tailored for each application domain (Clark et al, 2004; Mauw et al, 2004). A domain-specific language is a computer language oriented to a particular problem domain (Mernik et al, 2005; van Deursen et al, 2000).

* Corresponding author

Since they incorporate primitives, means of combination and means of abstraction restricted to and closely related with their associated problem domains, these languages are easier to use than other general-purpose languages for solving the specific problems for which they are designed. Indeed, the ultimate goal of a domain-specific language is to let experts in the domain understand and use the language. In this sense, such a language can play the role of an authoring tool. Therefore, language-driven development approaches facilitate the collaboration between developers and domain-experts, since they promote a clear separation of roles between those different stakeholders: developers produce languages, while experts in the application domain use these languages to produce and maintain the applications. In the words of Harold Abelson and Gerald J. Sussman (1996): "… seen from this perspective, the technology for coping with large-scale computer systems merges with the technology for building new computer languages, and computer science itself becomes no more (and no less) than the discipline of constructing appropriate descriptive languages". As stated earlier, in the present work we adopt this approach for the design of MVC applications.

Domain-specific languages are rather common in the software industry. Some examples of widely used domain-specific languages are HTML, for preparing web pages, SQL for querying relational databases, or JavaCC for compiler construction. The concept should be familiar to Unix / Linux users. Indeed, Unix-like systems include many of those languages (e.g. *awk* for string processing, *troff* for document formatting, *make* for controlling the generation of executables and other non-source files, etc), which can be combined by users in a smooth way to address tasks that are more complex. This kind of languages is also very common in e-learning. In the end, the different IMS specifications of the several aspects of an e-learning system (Friesen, 2005) are actually reduced to multiple domain-specific XML-based descriptive markup languages. In our opinion, this is what makes the application of suitable language-driven approaches to such a domain meaningful.

2.2. An Example Application

In order to illustrate our approach we will use a very simplified version of a Socratic Tutoring System, which is based on the works performed by Prof. Alfred Bork's team during the eighties (Bork, 1985; Ibrahim, 1989). Although the pedagogical adequacy of tutoring systems as mechanisms for supporting sophisticated learning processes has been seriously questioned, today there is a very active community working in this field, as well as relevant initiatives (e.g. visit icampus.mit.edu/XTutor/). Moreover, Socratic Tutors may be considered as an extension of some aspects of widely-used content sequencing proposals, such as the IMS Simple Sequencing proposal adopted by SCORM, Shareable Content Object Reference Model specification (ADL-SCORM, 2003; visit also www.imsglobal.org/simplesequencing). However, our reason for choosing this kind of systems for exemplifying our approach is not so much pedagogical as technological, since the goal of our work is not to criticize or to defend a particular learning approach, but to provide guidelines that can be effectively used to produce and maintain applications. For this purpose, we need a language simple enough to be fully addressed in this paper, and this type of tutoring system will let us do so.

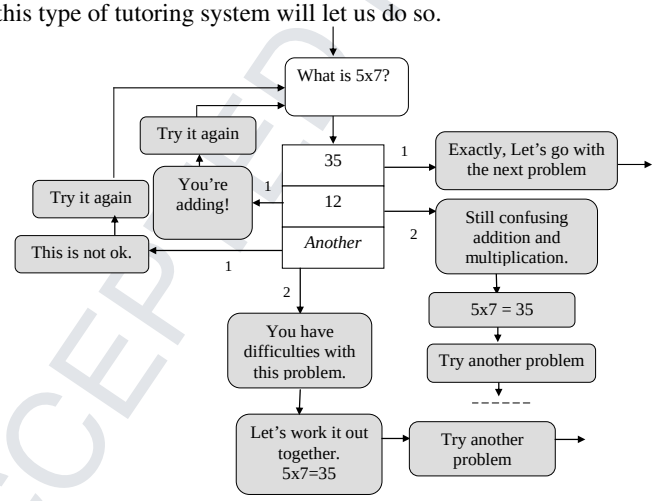

Fig. 1. Graphic representation of a tutorial fragment for a Socratic tutoring system -example adapted from Bork (1985)-.

Our example system runs tutorials where a problem is posed to the learner and through a master-disciple dialog a solution is built. The responses of the learners are analyzed by the system, which provides them with some feedback and determines the next step to undertake in the learning process. In order to adapt these feedbacks to the learning path followed, the system records the number of times that the learner gives a particular answer to a question. In general, feedback depends on the history of previous answers but in the simplest cases it can exclusively depend on the values of these counters, as assumed in the simplification used in this paper.

In Fig. 1 we show a simple example of this kind of organization for the tutorials. The white rounded box introduces a *question point*: it is associated with a question proffered by the system, and it is followed by a set of possible learner's responses (the compartmented white box). The other rounded boxes (the shadowed ones) are associated with general speeches proffered by the system. The learning flow is represented by directed arcs. An arc with its starting point in a possible response is labeled with a value, which indicates the value of the corresponding counter required to traverse it (e.g. in Fig. 1, the first time that the learner gives the response *12* to the *what is 5x7?* question, the system traverses the corresponding arc labeled by *1*, and the second time that labeled by *2*).

2.3. A Reference Architecture

In Fig. 2 we sketch a very high-level organization for an MVC application, which will be adopted as a reference in the rest of the paper. In this organization we distinguish:

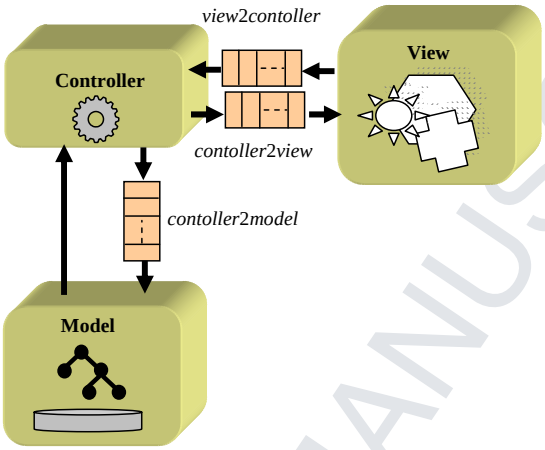

Fig. 2. High-level organization of an MVC architecture.

- The *model*, which contains the contents integrated in the application together with their structure. In our application example, this model will be a representation of a Socratic tutorial of the kind described above.
- The *view*, which presents the information to the user, and which captures his/her interactions. In our application example, the view will present the learner with the different problems, questions and feedback offered by the system. Moreover, it will also acquire the learner's responses.
- The *controller*, which reacts to the user's interactions by querying and/or updating the model, and by updating the view. In our example, the controller will take care of the interaction behavior informally described in the previous point.

Notice that in contrast to the more usual presentation of MVC we use streams to obtain a more abstract architecture. We explicitly distinguish input and output streams that communicate the controller with the view. The *view2controller* input stream will port suitable representations of the user's interactions, while the controller will write appropriate commands to govern the view's update in the *controller2view* output stream. Therefore, these two streams are a convenient way of abstracting the view for the rest of the application. Regarding the model, we will suppose that the controller has complete access to its structure, but not to the actual contents. In order to update the real contents, the controller must write appropriate update commands in the *controller2model* output stream. This lets us ignore the actual nature of the contents.

3. An Overview of the Approach

Our language-driven approach is based on our previous experiences using domain-specific descriptive markup languages (Coombs et al, 1987) in order to involve domain-experts in the development and maintenance of interactive applications. Among these experiences, which are the basis for our work on the so-called *document-oriented* approach to the production and maintenance of content-intensive applications (Sierra et al, 2005a; Sierra et al, 2006b), we can highlight the following:

- *Lire en Français*, an educational hypermedia for language learning (Fernández-Valmayor et al, 1999). The main pedagogical aim of this application was to help students to develop the abilities needed to understand texts written in a foreign language close to their mother's tongue. We used domain-specific SGML-based markup languages to regulate the collaboration between linguists, who provided the texts and the linguistic structures associated with them, and

developers, who used those descriptions to re-generate the application. The application was formerly deployed on a CD, but subsequently it was ported to a web environment, where an MVC architecture was also adopted.

- *Chasqui*, an MVC web-based system for the construction of repositories of digital learning objects associated with academic museums (Sierra et al, 2006a). Learning objects in *Chasqui* can own documental resources structured with domain-specific markup languages. In this experience we put special emphasis on involving domain experts, archaeologists, ethnologists and historians in the design of such languages from the beginning.

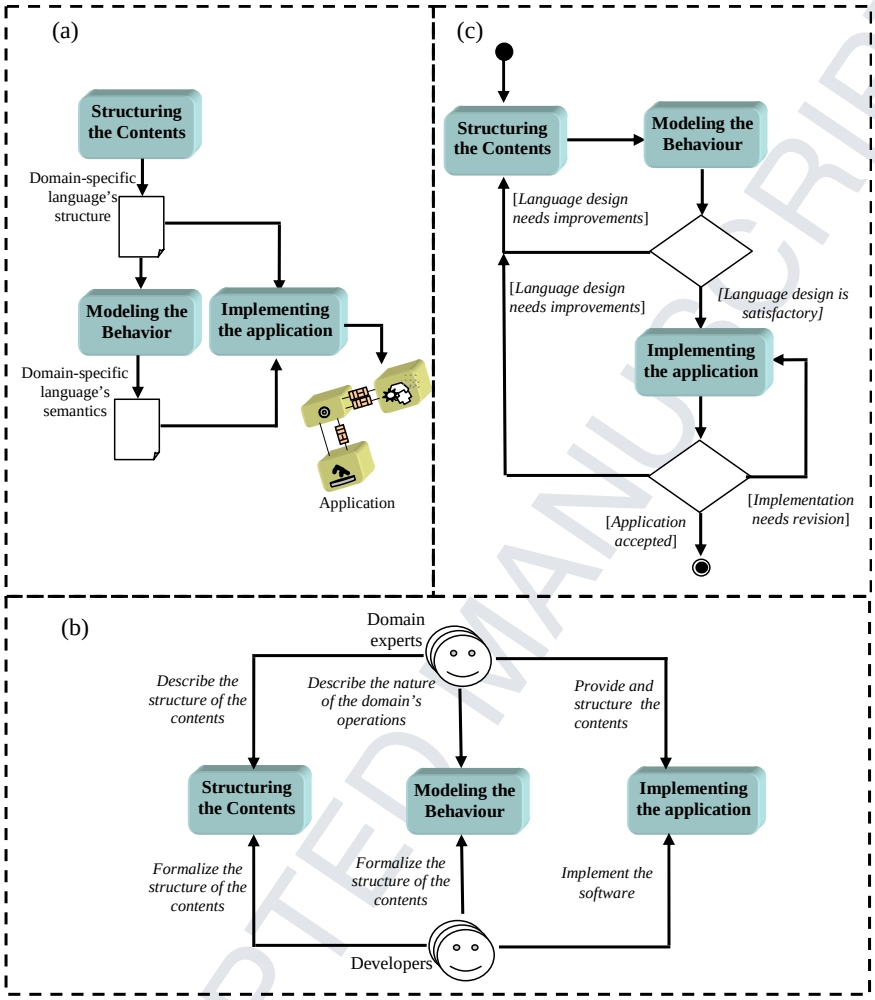

Fig. 3.Three views of the language-driven approach: (a) activities and products; (b) participants and their roles in the activities; (c) sequencing of the activities.

- The <e-Aula> platform, a highly customizable research e-learning platform (Sierra et al, 2007c). This platform, which is also doted with an MVC architecture, is fully compliant with the e-learning specifications proposed by the IMS consortium (visit www.imsglobal.org). Our main aim in constructing this platform was to facilitate its customization on different application profiles. This customization can involve static, moderately interactive, and highly interactive contents. Lecturers, domain experts in different subject areas such as computer science, medicine and linguistics, structured these contents as marked documents, using suitable domain-specific markup languages.
- <e-QTI> (Martínez-Ortiz et al, 2006a), a reusable and extensible assessment engine based on the XML binding defined by the IMS QTI (Question & Test Interoperability) e-learning specification (visit www.imsglobal.org/question/). Formerly <e-QTI> was conceived as a service in <e-Aula>, although later we deployed it as a standalone MVC-architected web-based application.
- <e-Game> / <e-Adventure>, a markup language for documents describing educational adventure videogames that we have formulated in the context of the <e-Aula> platform (Moreno-Ger et al, 2007). Indeed, the language that we have used in the present work is also inspired by the language used to structure conversations in <e-Adventure>.

In all these fields we found the use of domain-specific languages oriented to structuring the application contents particularly useful in reducing the average time of the maintenance iterations. Indeed, these languages can be directly used by domain-experts to directly structure the application contents. The resulting documents can be subsequently translated onto suitable abstract representations, which can be automatically processed to generate or re-generate the corresponding application model.

In Fig. 3 we sketch our approach in order to highlight its main aspects. Following a similar method to the one adopted in the presentation of our aforementioned document-oriented approach, we include a view of *activities and products*, a view of *activity sequencing*, and a view of *participants and roles*.

- In the *activities and products* view (Fig. 3a) we abstract the approach in terms of three activities and their resulting products. As mentioned before, the core of the approach is to provide a domain-specific language for structuring the application contents. In order to do so, we distinguish between the activity of *structuring the contents*, whose purpose is to characterize the *domain-specific language's structure*, and the activity of *modeling the behavior*, whose aim, through the semantics of this language, is to make the application's interactive behavior explicit. Finally, we introduce an activity for *implementing the application*, where the final *application* is produced according to the MVC architecture. In this paper we will mainly focus on the first two activities, although we also will give some guidelines regarding implementation.
- In the *participants and roles* view (Fig. 3b) we introduce the two main participants in the development process, and the roles they play in each activity. These participants are *domain experts*, who are responsible for describing the application domain, providing the contents and structuring them using the domain-specific language in order to enable the subsequent interactive behavior, and *developers*, who are responsible for formalizing the abstract syntax and semantics of the language and implementing the application's software by mapping those syntactic and semantic structures into the MVC architecture.
- In the *sequencing* view (Fig.3c) we outline how the activities are actually sequenced during the development of the applications. Notice that there are three different development loops. On one hand, when the language (its structure and its semantics) has been provided, it can be evaluated in order to decide on its adequacy. For this purpose, techniques such as rapid prototyping can be used. As we will see below, rapid prototyping can be facilitated by the explicit formalization of the language's semantics. On the other hand, during implementation the resulting application is evaluated. In this process some aspects to improve may be discovered, and therefore new implementation iterations should be carried out. It should be noticed that many of these iterations will imply only the use of the domain-specific language to complete and to refine the application's contents. Indeed, it can be also extended to maintenance stages, where, in the limit, maintenance will be carried out by domain-experts using the cited language. Still, some iteration may imply fixing some bugs discovered in the application's software. Finally, it is also possible to discover some features in the contents that are not addressable with the currently available language. Therefore, the design of the language itself must be revised in order to extend and/or to adapt it.

Next sections detail the different aspects of the approach, focusing in the nature of its activities.

4. Structuring the Contents

The first step in our approach is to design a language that reflects the structure of the application's contents that is relevant for user-system interactions. From the perspective of the MVC architecture, it will be equivalent to deciding the high-level structure of the information stored in the application's model. It is important to notice that this language, as happens with descriptive markup languages, will not be intended to represent the *actual contents*, but only to represent their structure. Therefore, this language will include mechanisms to refer to the actual contents. This language will be very relevant for specifying the high-level application's behavior. Indeed, in our approach we will specify the application's behavior by adding suitable operational semantics to such a language, which is the counterpart of determining the controller's behavior in this approach. This language will also play an important role during the *externalization* of the contents. In application domains such as e-learning, this externalization is very important in order to involve domain-experts in the production and maintenance of the application's contents. In fact, this language can be synthesized in a domain-specific descriptive markup language for the contents, as described in the previous section.

In the case of the Socratic tutors, the design of this language will be biased to include those features that are relevant from an interaction point of view. This means that other features (e.g. presentation-oriented ones) will usually be omitted. Furthermore, the design will be focused on characterizing the abstract syntax of the language, as this syntax will serve as the basis for subsequent behavioral specification. The abstract structure of the Socratic dialog of the section 2 is a weighted graph whose nodes contain different types of information. As in our previous wok in educational adventure videogames (Moreno-Ger et al, 2007), where as said before we faced a similar modeling scenario regarding conversations, a relational-oriented representation of this structure is well-suited to design needs. Indeed, modeling the behavior of the application will demand frequent queries to different components of the model, and it is simpler to specify these components as separate and distinct information items. Indeed, the representation can be conceived of as a set of such information items, which are in turn represented as ground terms on a suitable signature.

Information item	Intended meaning
start(*rs*)	The learning process starts with the speech *rs*.
speech(*s*,*n*)	The speech *s* is followed by *n*. In this item, *n* can be another speech, a question point, or the end of the learning process. In addition, *s* is a unique identifier.
question(*q*)	A question point identified with the unique identifier *q*.
feedback(*q*,*a*,*n*,*s*)	A feedback for the answer *a* (a unique identifier) collected in a question point *q*. Also, *n* is the number of times that the answer has been collected. The feedback itself will start with speech *s*.
end(*e*)	An ending point for the learning process. Here, *e* is a unique identifier.

Fig. 4. Information items for the language of the Socratic tutoring system.

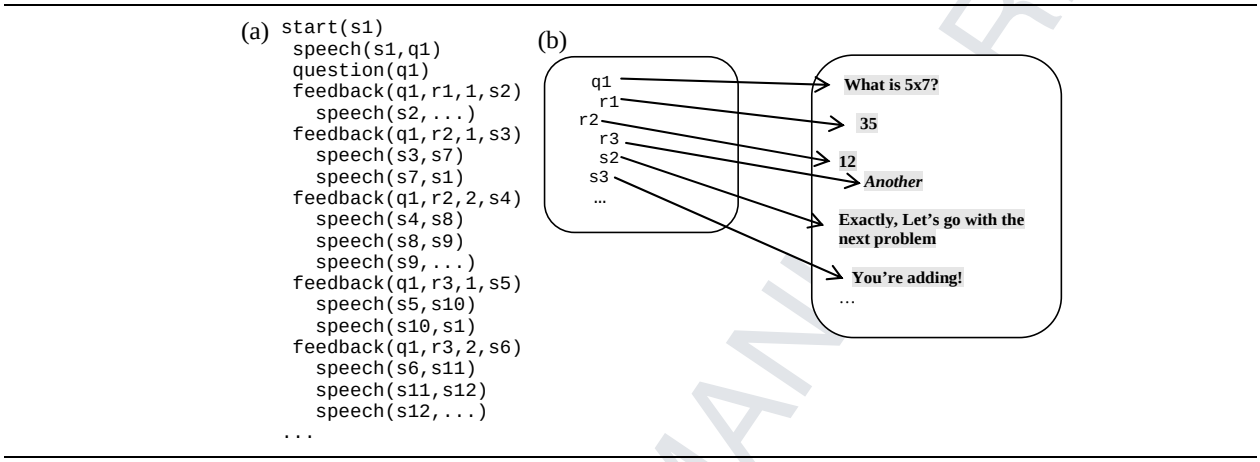

Fig. 5.(a) Representation of the structure of the fragment of tutorial sketched in Fig. 1; (b) actual contents are not included in the representation, but they are replaced by appropriate identifiers.

In Fig. 4 we propose such a relational-oriented representation. In Fig. 5a we illustrate this representation with the tutorial fragment represented in Fig. 1. Only the structure of the contents, but not the actual contents, is represented. Indeed, in the representation, such contents are replaced by identifiers referring to them, as suggested in Fig. 5b.

5. Modeling the Behavior

Once the language for structuring the contents is available, the behavior of the application can be modeled by assigning suitable operational semantics to such a language. For this purpose we propose to use the structural style of operational semantics (Mosses, 2006; Plotkin, 1981). This approach leads to reasonably understandable specifications, which can also be easily prototyped in order to check the adequacy of the subsequent implementation with very little additional effort (Clément et al, 1986). According to the approach, the operational semantics of a language is characterized by modelling the behaviour of an abstract machine that executes this language as a *formal calculus* made up of *inference rules* like those depicted in Fig. 6. The reading of such rules is the usual one: when all the elements in the premise hold, the elements in the conclusion hold. Empty premises can be omitted, and the resulting rules are used to introduce *axioms* into the calculus. Most of the interesting calculi will usually consist of an infinite number of inference rules. In order to give a finite characterization, a finite number of *rule patterns* can be provided instead by using *syntactic variables*. We will use a *cursive* font to denote syntactic variables in our specification.

$$\frac{\Phi_o \,;;...;;\, \Phi_k}{\Psi_o \,;;..;;\, \Psi_n}$$

Fig. 6. Structure of the inference rules. Each Φ_i in the premise and Ψ_j in the conclusion are formal expressions.

5.1. Designing the Computation States

Computation states will closely mirror the components identified in the reference architecture. Therefore, typically such states will include:

$$\frac{\vdash \langle a,n\rangle \in \rho}{\vdash \rho_a = n}$$

$$\frac{\vdash \langle a,n\rangle \notin \rho}{\vdash \rho_a = 1}$$

$$\vdash \left(\rho_a := n\right) = \left\{\langle a',n'\rangle \mid \langle a',n'\rangle \in \rho \wedge a' \neq a\right\} \cup \left\{\langle a,n\rangle\right\}$$

Fig. 7. Consulting and setting the counters associated with the answers.

Term	Intended meaning
run-ordered	The learner wants to start the tutorial.
answered (*a*)	The learner has given the answer *a* to the last question.

Fig. 8. Terms in the *view*2*controller* stream.

Term	Intended meaning
do-start	The system execution is starting.
do-speech(*s*)	The system is proffering the speech *s*
do-end	The system execution is ending.

Fig. 9. Terms in the *controller*2*view* stream.

- A representation of the model, in terms of the language for structuring the contents provided. As stated in the previous section, typically this representation will be a set of ground terms. Therefore, we can use typical first-order logic and set theoretical constructs and operations in querying such representations. This lets us reuse an appropriate axiomatization for such basic constructs –see, for instance the work of Lawrence and Grabczewski (1996). In this way, we only define the set-related domain-specific notations specifically introduced for supporting particular specifications. In the subsequent examples, we will use $\vdash \Phi$ for denoting a set-theoretical formula Φ that must hold.
- A *control term*, which is the term of the model which is currently under consideration. This term will be used to decide how the execution is to proceed.
- A *context*, which contains additional information used to perform the transitions and expresses the dependence of the actions on previous responses.
- The streams identified in the reference architecture. For those applications where the model is not updated, but only queried, the *controller*2*model* stream can be omitted. In order to represent streams we suggest using a tuple-oriented notation. Empty streams will be noted by empty tuples. If *s* is an input stream and *t* is a term, with <*t*,*s*> we will represent the input stream whose first element is *t* and whose rest is *s*. Analogously, if *o* is an output stream and *t* is a term, by <*o*,*t*> we will represent the result of writing *t* in *o*. Also, in facing this aspect of the design of the computation state, it is very relevant to characterize the kind of terms that can appear in each one of the streams.

In our example, we will represent computation states as 5-tuples of the form <θ,*T*,ρ,*in*,*out*>, where:

- θ is the control term and *T* is the structure of the tutorial represented as a set of terms, as described in the previous section.
- ρ is the context, which in this example will be constituted by a set of counters associated with the answers. Formally, the elements of this set will be pairs of the form <*a*,*n*>, with *a* an answer identifier, and *n* the associated counter. The notation introduced in Fig. 7 is a convenient way of consulting and setting these counters. Notice that, if the counter is not set, its value is considered 1, since it will be the first time that the learner proffers the associated answer.
- *in* and *out* are the *view*2*controller* and the *controller*2*view* streams, respectively. Since in this example the model is not updated, the *controller*2*model* stream can be dropped from the computation state. In Fig. 8 we characterize the possible terms in the *view*2*controller* stream. In Fig. 9 we show the possible terms in the *controller*2*view* one. In this case, the structure of these streams is very simple: *view*2*controller* can contain the order for running the system, as well as the learner's responses, while *controller*2*view* will only contain the commands for presenting the system's speeches, as well as for announcing the init and the end of the system's execution.

In addition to this kind of states, we will also introduce a special format for initial and final states:

- Initial states will be represented as <*T*,*in*,*out*>, with *T* a tutorial and *in* and *out* the *view*2*controller* and the *controller*2*view* streams.

- Final states will be represented as <*out*>, with *out* the resulting *controller*2*view* stream.

5.2. Establishing the Semantic Rules

For the specification of the semantic rules we will use a notation of the form $\Psi \rightarrow \Psi'$ to denote the transition from the state Ψ to the state Ψ'. Furthermore, at the top of the rules we will put the applicability conditions, while at the bottom we describe the resulting transitions. Therefore, we will adhere to a small-step style (Mosses, 2006), which also eases the transition to a subsequent implementation. If needed, big-step style semantics can be subsequently obtained as usual, by taking the transitive closure of the involved transition relations (Fig. 10).

$$\frac{\Phi \rightarrow \Psi}{\Phi \rightarrow^{+} \Psi}$$

$$\frac{\Phi \rightarrow^{+} \Psi \;;;\; \Psi \rightarrow^{+} \Sigma}{\Phi \rightarrow^{+} \Sigma}$$

Fig. 10. Axiomatizing a big-step transition relation $\rightarrow^{+}$ in terms of a small-step one $\rightarrow$.

In Fig. 11 we show the semantics rules for our example. These rules formally state the informal behavior outlined in section 2.1:

$$\frac{\vdash \text{start}(s) \in T}{\langle T, \langle \text{run-ordered}, In\rangle, Out\rangle \rightarrow \langle \text{start}(s), T, \varnothing, In, \langle Out, \text{do-start}\rangle\rangle} \textbf{ starting}$$

$$\frac{\vdash \text{speech}(s, ns) \in T}{\langle \text{start}(s), T, \rho, In, Out\rangle \rightarrow \langle \text{speech}(s, ns), T, \rho, In, Out\rangle} \textbf{ starting-speech}$$

$$\frac{\vdash \text{speech}(ns, nns) \in T}{\langle \text{speech}(s, ns), T, \rho, In, Out\rangle \rightarrow \langle \text{speech}(ns, nns), T, \rho, In, \langle Out, \text{do-speech}(s)\rangle\rangle} \textbf{ speaking1}$$

$$\frac{\vdash \text{question}(q) \in T}{\langle \text{speech}(s, q), T, \rho, In, Out\rangle \rightarrow \langle \text{question}(q), T, \rho, In, \langle Out, \text{do-speech}(s)\rangle\rangle} \textbf{ speaking2}$$

$$\frac{\vdash \text{end}(e) \in T}{\langle \text{speech}(s, e), T, \rho, In, Out\rangle \rightarrow \langle \text{end}(e), T, \rho, In, \langle Out, \text{do-speech}(s)\rangle\rangle} \textbf{ speaking3}$$

$$\frac{\vdash \text{feedback}(q, a, \rho_a, s) \in T}{\langle \text{question}(q), T, \rho, \langle \text{answered}(a), In\rangle, Out\rangle \rightarrow \langle \text{feedback}(q, a, \rho_a, s), T, \rho_a := \rho_a + 1, In, Out\rangle} \textbf{ evaluating}$$

$$\frac{\vdash \text{speech}(s, n) \in T}{\langle \text{feedback}(\_, \_, \_, s), T, \rho, In, Out\rangle \rightarrow \langle \text{speech}(s, n), T, \rho, In, Out\rangle} \textbf{ starting-feedback}$$

$$\langle \text{end}(e), T, \rho, \langle\rangle, Out\rangle \rightarrow \langle\langle Out, \text{do-end}\rangle\rangle \textbf{ ending}$$

Fig. 11. Semantic rules for the Socratic tutoring system.

- The *starting* rule models the execution's init. For this purpose, the starting speech is queried in the tutorial. Also notice that the context with the answers' counters is set to the empty set (i.e. in virtue of Fig. 7, the counter for every answer will be 1). Finally, a suitable command announcing the beginning of the execution is written in the *controller*2*view* stream.
- The *starting-speech* rule models the beginning of a speech. The first speech is picked from the tutorial, and it is set as the control term. Therefore, the system is prepared to proffer such a speech.
- The *speaking1* rule models how the system proffers a speech which is followed by another speech: a command to proffer the speech is written in the *controller*2*view* stream, and the next speech is set as the control term. The proffering of a speech when it is followed by a question point (*speaking2* rule) or by an end point (*speaking3* rule) is modeled in an identical way.

- The *evaluating* rule models what happens in a question point. The answer of the learner is read from the *view*2*controller* stream, a suitable feedback is picked from the tutorial, which is set as the control term, and the counter associated with the answer is incremented.
- The *starting-feedback* rule models the offering of feedback by the system. Basically, it is reduced to starting a new speech (notice that with each _ we denote an anonymous different syntactic variable).
- Finally, the *ending* rule models how the system finishes the execution. For this purpose, no more user interactions are allowed (i.e. the *view*2*controller* stream must be empty), and a command announcing the end is written in the resulting output stream.

$$\frac{\vdash \text{start}(s) \in T \;;;\; \vdash \text{speech}(s, ns) \in T}{\langle T, \langle \text{run-ordered}, In\rangle, Out\rangle \rightarrow \langle \text{speech}(s, ns), T, \varnothing, In, \langle Out, \text{do-start}\rangle\rangle} \textbf{ starting}$$

$$\frac{\vdash \text{speech}(ns, nns) \in T}{\langle \text{speech}(s, ns), T, \rho, In, Out\rangle \rightarrow \langle \text{speech}(ns, nns), T, \rho, In, \langle Out, \text{do-speech}(s)\rangle\rangle} \textbf{ speaking1}$$

$$\frac{\vdash \text{question}(q) \in T}{\langle \text{speech}(s, q), T, \rho, In, Out\rangle \rightarrow \langle \text{question}(q), T, \rho, In, \langle Out, \text{do-speech}(s)\rangle\rangle} \textbf{ speaking2}$$

$$\frac{\vdash \text{feedback}(q, a, \rho_a, s) \in T \;;;\; \vdash \text{speech}(s, n) \in T}{\langle \text{question}(q), T, \rho, \langle \text{answered}(a), In\rangle, Out\rangle \rightarrow \langle \text{speech}(s, n), T, \rho_a := \rho_a + 1, In, Out\rangle} \textbf{ evaluating}$$

$$\frac{\vdash \text{end}(e) \in T}{\langle \text{speech}(s, e), T, \rho, \langle\rangle, Out\rangle \rightarrow \langle\langle\langle Out, \text{do-speech}(s)\rangle, \text{do-end}\rangle\rangle} \textbf{ ending}$$

Fig. 12. Result of simplifying the semantics of Fig. 11.

During our experiences in the e-learning domain we have established how the effort employed in this kind of specifications pays out, since these specifications are very valuable in anticipating the more obscure aspects of the application's dynamic behavior without being obfuscated by technological and/or implementation details. As we said before, based on these specifications it is possible to perform rapid prototyping of the applications using, for instance, approaches similar to those described by Clément et al. (1986). Finally, the formal flavor of the specifications provides the opportunity to apply optimizations and refinements in very early stages of the development process. For instance, in the specification of Fig. 11 it is possible to meld the *starting* and *starting-speech* rules into a single one. The same is true for *speaking*3 and *ending*, and for *evaluating* and *starting-feedback*. The resulting specification, which is sketched in Fig. 12, is a more compact specification, and therefore it can lead to a smaller and simpler implementation.

6. Some Implementation Guidelines

In this section we examine how to organize the final applications from their design in the terms stated in this paper. While the particular implementation strategies depend on the particular technologies and platforms chosen, we still can abstract some general implementation guidelines that can be useful in most of the cases. In Fig. 13 we illustrate a general organization, which is a refinement of the reference architecture exposed in section 2. In this organization:

- The controller is architected with a set of *control rules*, which implements the semantics rules identified in the behavioral dimension. These components have access to a global state, which can refer to the structural representation of the model.
- Communication between the view and the controller is performed using call-back mechanisms. The controller registers itself as an observer in the view, and it is called as response to the user's events. Then, the control rules are activated, and in consequence, a *view updater* is invoked in the view. During this process, appropriate *user's actions* and *update commands* are transferred between the controller and the view. Therefore, this call-back mechanism is a straightforward simplification of the stream-based communication used during specification. Indeed, readings from the *view*2*controller* stream are converted into callbacks, while writing the *controller*2*view* is converted into calling the view updater.

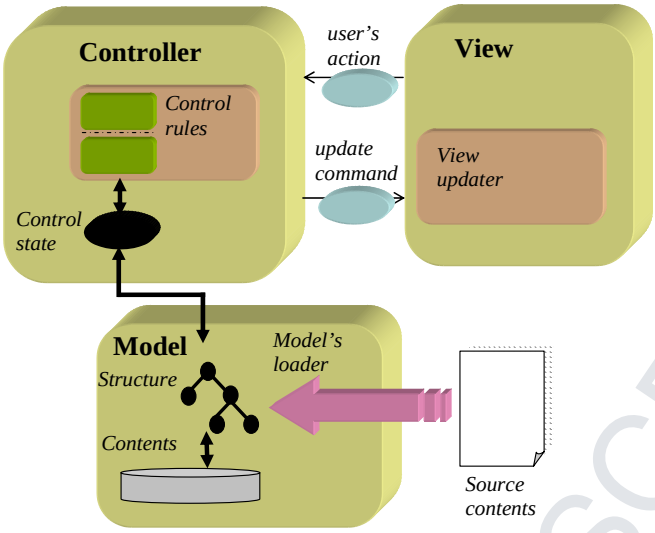

Fig. 13. A possible organization for the final implementations.

- Contents structures are encapsulated in the model. Updating these contents is performed by invoking the appropriate operations on the model structure, which in turn will act on the actual contents. Also notice that this structure can be referenced in the user's actions and in the update commands, therefore making the relevant parts of the model accessible for the view. Finally, notice that we explicitly identify a *model's loader*, which enables us to load bodies of author-oriented contents in the application's model. As indicated in section 3, in our experiences we have made extensive use of descriptive markup technologies to implement this stage. Indeed, we provide authors with appropriate domain-specific descriptive markup languages, and they provide the contents as documents structured according to such languages (Sierra et al, 2005a; Sierra et al, 2006b).

In Fig. 14 we exemplify the high-level structure of the Socratic tutoring system in the terms proposed in this section. This implementation has been deployed as a web-based application and is architected as follows:

- We architect the controller in terms of the simplified set of semantics rules shown in Fig. 12. Also, the control state is derived by dropping the streams from the operational semantics' computational state: it maintains a reference to the overall model's structure, another reference to the current control term, and a table with the answers' counters.
- The *view updater* is implemented using a Java applet. Indeed, by inspecting the operational semantics the proactive nature of the controller is revealed in a very early stage of the development process: once the controller has read a user's action, it can communicate to the view several update commands (i.e. the speeches proffered by the tutoring systems can proceed in several acts). This behavior is not directly supported in HTTP, and therefore we need an active component in the client side (in this case, the control applet) taking care of it. Indeed, this kind of behavior is very common in the e-learning domain, where similar solutions have been also adopted –e.g. the runtime in SCORM– and the rigorous design of the application can help to anticipate it.
- The model exhibits a tutorial structure that is a direct implementation of the structuring language's abstract syntax described above. The actual contents are organized as a set of basic multimedia assets (e.g. images, videos, sounds, etc.) and HTML pages. In order to produce all these components, we introduce a user-friendly XML-based markup language (Bray et al, 2000), which can be used by instructors to structure the tutorials. The resulting XML documents are then processed to produce the abstract representations of the structure of the tutorials. Instructors can prepare the actual contents directly as HTML pages, but they can also use other domain-specific XML-based markup languages. The resulting XML documents can be transformed into final HTML presentations using suitable XSLT transformations (Clark, 1999).

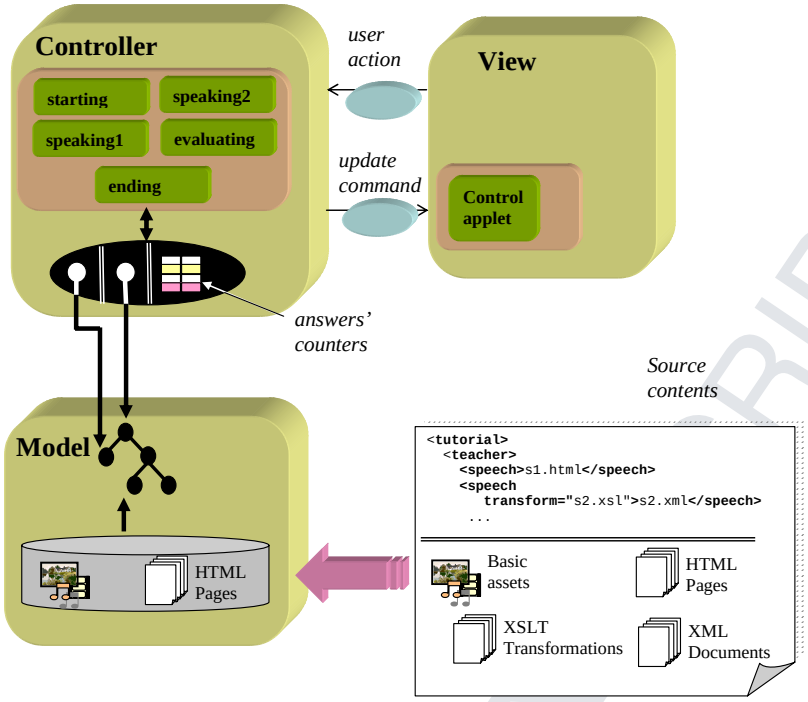

Fig. 14. High-level organization of the Socratic Tutoring System's implementation

Currently we are re-factoring this implementation in terms of AJAX technology (Paulson, 2005). Regardless of the migration of the controller aspect to the client side, the implementation is being substantially facilitated by the earlier effort made in the formal design, which has been shown to be fairly independent of the subsequent platform or technology.

7. Related Work

In this section we present some work related to our language-driven proposal and we discuss the similarities, differences, advantages, disadvantages and limitations of our approach with respect to these proposals. As an exhaustive analysis is not possible we have specifically focus on: a) approaches to the specification of interactive behavior based on state machines (section 7.1), b) approaches based on general-purpose formal specification languages (section 7.2), and c) approaches based on general-purpose software modeling languages (section 7.3).

7.1. Approaches based on state machines

System behavior in our language-driven approach is ultimately characterized in terms of computation states and transitions between those states. In many cases, state transition systems have been used for characterizing the interactive behavior of user interfaces (Parnas 1969; Jacob 1983; Wasserman 1985; Green 1987). In particular, several formal specification approaches based on finite state machines have been proposed in the web-based applications scenario, which is also the main focus of our approach. Draheim and Weber (2005) propose the use of *bipartite state machines*. These machines comprise states where the user is observing web pages, and states where he/she is acting on a web *form*. The second kind of state can further trigger a web action, which produces a new web page to be seen. Therefore, the resulting machine is bipartite. A similar approach is described by Doubrovski (2001). In Doubrovski's work the state machines used are called *interaction machines*, and transitions are fired by predicates in the application's context, which can include the result of the user's actions (e.g. input collected on a form). Actually, applications modeled using this approach are limited to the so-called *submit/response* interfaces, where all the screen or page updates must be triggered by distinguished user actions (e.g. clicking a link or pressing a submit button). While many interesting web-based applications can be grouped into this category, it leaves out applications where the view is automatically updated as a consequence of the proactive behavior of the server side (e.g. AJAX technologies in the web domain). As we have already discussed, applications beyond the *submit/response* style naturally arise in domains such as e-learning. For this purpose, more sophisticated approaches can be used. Comai and Fraternali (2001) propose Statecharts (Harel 1987; Horrocks 1999) as the semantic base for WebML (Ceri et al, 2000), a structurally-rich language for the design of web applications. While the resulting machines are more complex than the previously mentioned ones (e.g. they are equipped with concurrent states in order to allow the activation of

multiple units of content in the same page), they still stay in the finite state machine category. Moreover, the original WebML proposal is semantically neutral and also covers a wide variety of aspects that are beyond the scope of our proposal (e.g. presentation, personalization, etc.). Proposals such as Pipe (Navarro et al. 2004), a modeling approach for hypermedia and web-based applications, are similar in nature to WebML. Pipe models are equipped with default browsing semantics that is also finite state based.

According to the guidelines previously presented in this paper, each domain-specific language denotes a (usually, although not necessarily, finite) state-machine operating as a translator and whose input language are sequences of user's actions, and whose output(s) language(s) are sequences of presentation commands (and/or of model updating commands). In this way, when domain-experts impose structure on the contents of the application using a collaboratively defined domain-specific language, they are implicitly providing a state-machine based specification of the application's interactive behavior. While in the other approaches referenced in this section the focus is put on characterizing the interactive behavior of each particular application in terms of a state-machine using a predefined notation, in our language-driven approach developers and domain-experts collaborate to define a notation appropriate for an entire family of applications in a domain. Since in domains involving content-rich applications, such as e-learning, those who really know the peculiarities of each kind of applications are domain-experts themselves (e.g. those who really know how to build a Socratic tutor in Medicine are the medical instructors), by providing them with languages closer to the domain, the active participation of domain-experts in the whole life cycle of the applications will be facilitated and therefore the final quality of the *models* produced will be improved. Hence the main implications of using operational semantics as a previous and systematic step to state-machine based specification are as follows:

- There is not a predefined notation to describe finite-state machines (such as the Draheim and Weber', the Doubrovski's or the WebML ones). Instead, developers and domain-experts collaborate to produce notations specifically tailored for each intended type of application.
- Each particular description in one of these domain-specific languages denotes a (usually finite) state-machine, which is characterized by the language's operational semantics.
- These machines are implicitly specified and maintained by the domain-experts when they structure the contents using the domain-specific languages, but with the advantage that actually they are not aware of this fact! The machines themselves are the *semantics* of the structures imposed. Experts are only required to use a suitable concrete binding of a notation that they understand (e.g. encoded with a suitable descriptive markup language), since it is close to their domains of expertise, and since they have actively collaborated during their conception. Therefore, specifying the interactive behavior using these domain-specific notations is easier for domain-experts than doing it using a wider-domain or a general-purpose (domain-independent) notation.
- The main shortcoming of our approach with respect to those proposing predefined notations is the drawback common to any approach based on the design and implementation of domain-specific languages: the costs associated with the production and maintenance of specific languages for each application domain (van Deursen et al. 2000; Mernick et al. 2005). Nevertheless, as we have said before, for domains where close collaboration between domain-experts and developers is required, these costs will pay off during the subsequent development and maintenance stages, since applications will be largely maintained, and even produced, by domain-experts with little or no intervention of developers.

Notice also that, when defining the structure of each domain-specific notation, that is, the abstract syntax of each language, developers should include suitable expressive mechanisms to deal with the interaction context (e.g. dialogue history). Addressing the context however is not a distinctive feature of our approach, since it is an intrinsic feature of any reasonably complex interaction style, and thus it should be addressed by any method used for interactive systems description. For instance, Dranhein and Weber' *formcharts* (a particular type of bipartite state machines) constrain dialogue by using an extension of the OCL language (Clark and Warmer 2002). Doubrovski's interaction machines use predicates on the application's internal state in order to constrain transitions. WebML uses contextual links in its *navigation models* to carry some information (*context*) from source to destination units, which can be used to determine the actual content to be shown. Similar mechanisms are also used in our previous Pipe modeling approach for hypermedia and web-based applications. In the language-driven approach interaction context must be also addressed, but the way to do it will depend on (and will vary with) each specific application domain. When defining the structure of domain-specific languages, developers should include suitable expressive mechanisms to deal with the context, and these mechanisms will be particular to each application domain addressed. The use of these mechanisms by domain-experts will be easier than the use of general-purpose and universal ones. For instance, in our Socratic tutorial example counters play this role, although in more complex cases using more complex mechanisms could be needed to describe context (e.g. stacks or any other recursive feature). In addition, the operational semantics should clearly describe how to propagate and manipulate context during interaction (e.g. in our example, it is done by updating counters each time the learner gives a particular answer).

Finally, it could be questioned why structural operational semantics is chosen to characterize the state-machines associated with each *phrase* in the domain-specific notations instead of another more usual formalism in human-computer interaction. Regarding finite state-machines, it is done because pure finite automata are not enough, since, in the end, for

each domain-specific language we must characterize an *interpreter* for *all* the finite state machines that can be described using the language and therefore the number of states of this interpreter could be infinite. For instance, in the Socratic Tutoring System it would not be pedagogically sound to limit the number of answers anticipated by the instructor, the number of times that the learner can respond a particular answer, or the number of distinguished feedbacks for each response. Thus, while a particular Socratic tutor has a finite number of states, we can not anticipate the number of states that an arbitrary Socratic tutor can have. Therefore, we will be unable to model our resulting Socratic Tutoring System (i.e. the system able to run any Socratic tutorial) as a finite automaton. Of course the basic computational model of finite state machines can be extended. In recursive transition diagrams (Woods, 1970) transitions can refer to other diagrams, and the resulting formalism is equivalent in power to context-free grammars and pushdown automata (Woods, 1970). Augmented transition networks (ATNs) (Woods, 1970) add records and actions to compute the value of these records, and they are equivalent to Turing Machines and similar to translation schemas in the description of programming languages and their processors (Aho et al. 2007). Indeed, in *principle* anything modeled with structural operational semantics could be also modeled as an ATN. The reason to choose structural operational semantics instead of ATNs (or, equivalently, translation schemas for those developers preferring more *textual* and symbolic formalisms) is that this formalism is especially suited to *talk about* the dynamic meaning of a language, while ATN-like formalisms are more *translation-oriented*. Remark again than in the language-oriented approach we are dealing with a two-level linguistic situation: on one hand, the domain-specific language, on the other hand, the languages involved in the man-machine communication, as well as in the communication between application components. We may find it natural to think of an ATN for a particular Socratic tutorial, but, in order to deal with *all* the possible Socratic tutorials using the ATN formalism (or a computationally-complete equivalent one) we should devise some sort of *universal machine*. Indeed, this task can be easier tackled using operational semantics rules, where the transitions can be explicitly axiomatized as showed in the Socratic tutoring example.

7.2. Approaches based on general-purpose formal specification languages

There are also several initiatives focused on the use of more general-purpose formal modeling languages in the specification of web applications. Rezazadeh and Butler (2005) propose the B-method (Abrial 96) for this purpose, and they exemplify their approach with the modeling of a complex multi-tier web application. Syriani and Mansour (2003) propose SDL (Mitschele-Thiel 2001) as the basic modeling language. Deutsch et al (2006) use a relational approach in the specification of data-driven web applications, and they impose additional constraints on the specifications in order to make verifiability of the resulting applications decidable. A similar approach is adopted by Redouane (2004), with the use of a first-order definitional specification language, and by German (2002), who applies Hadez, a specification language similar to Z (Spivey 1992), in the modeling of a web museum. Regardless of being more extensive than our approach, which is mainly focused on high-level model structure and interaction, these approaches are usually *data-driven*. Ultimately, they conceive web applications as a suitable set of data types representing the contents, together with operations defined in these data types.

In our opinion comparing our approach with general-purpose formal modeling languages cited above such as SDL, Z or B is not easy mainly because the purpose is very different. On one hand, the aforementioned works mainly focus on the use by developers of the formal modeling languages for modeling *particular* web applications. This use is out of consideration of whether we need to make domain-experts responsible for describing relevant parts of the application. In effect, regardless of their expressivity, domain-experts without a prior background in Computer Science and/or formal methods could hardly use general-purpose formal modeling languages such as Z, SDL or B for making the structure of the application's contents explicit. Therefore, in order to align both approaches we should use those formal modeling languages with another objective: instead of modeling particular applications, we might model *application frameworks*, which subsequently could be extended and instantiated to yield particular applications. For instance, and thinking of our particular example of Socratic tutors, we could use a general-purpose language to give a set of modules with data types and operations representing the commonalities of such systems. In this respect, it is possible to find several formal specification initiatives in the field of interaction architectures and frameworks. Regarding the formal specification of MVC applications, Hussey and Carrington (1997) use Z to model this kind of architecture and to compare it with presentation-abstraction-control (Coutaz, 1987), another well-known architecture for interactive applications. Abstract Data Views (Cowan and Lucena, 1995) is another formal modeling technique of interactive applications that promotes the separation between the model and the user interface. The approach distinguishes between Abstract Data Objects (the application components) and Abstract Data Views (the interfaces to these components), and also regulates the connections between the two types of artifacts. Similar abstractions, this time based on the concept of *interactor*, are managed in the Abstraction-Display-Controller formal modeling approach (Markopoulos et al, 1997). In the Interactive Cooperative Objects, framework (Bastide et al, 2003) structural aspects are described using object-oriented concepts, while behavioral aspects are described using Petri nets. Still, all these approaches are largely data-driven, instead of language-driven. Therefore, the problem of *instantiating* these frameworks must be faced, and it should be lastly carried out at the level of the formal general-purpose modeling language used. As said before, it could hardly be performed by a domain-expert (e.g. an instructor who writes Socratic tutors) without the help of an external tool that hides the representational complexities of formal (e.g. algebraic or

logic-based) modeling languages. Hence additional authoring support for domain-experts should be provided, leading to a similar approach to that of using a suitable domain-specific language.

Thus, the main advantage of the language-driven approach with respect to the direct use of general-purpose formal modeling languages is to directly address the design of a language that can be used by domain-experts. As disadvantages we could cite the need to model some semantic constructs from scratch, which could be present in a general-purpose modeling language as primitive features, and whose modeling might not be so straightforward. As an example, we could think of the primitive support for concurrency in formalisms such as SDL, Petri's nets or Harel's statecharts mentioned in the previous discussion. A possible solution for this complexity could be the use of a suitable metalanguage for specifying structural operational semantics with built-in support for modularity to facilitate the reuse of pre-built semantic modules in the conception of new ones. For this purpose, proposals addressing the modular specification of the semantics of computer languages like the described by Mosses (2004) could be helpful.

7.3. Approaches based on general-purpose software modeling languages

General modeling languages, such as UML (Booch et al, 1998), as well as their specific web-oriented profiles (Conallen, 1999), are very valuable for expressing the main concepts of the application's design. While UML and the related technologies are very valuable artifacts for the description of the high-level aspect of the system (e.g. architectural design of the final implementation), in our opinion the description of the structure and the semantics of computer languages can take advantage of the more specific and rigorous approaches developed by the programming languages community. Indeed, language design and implementation is a very mature field, which has its own techniques regardless of the use of UML and UML-related technologies. For instance, instead of using UML to describe a context-free grammar, most developers would prefer a more standard notation, such as Backus-Naur Form (Naur 1960). The same is true of structural operational semantics versus UML if they must precisely describe the dynamic behavior of a computer language. While UML interaction diagrams can be very valuable for describing dynamics of pre-established groups of objects, it is not necessarily true when describing the evaluation of the abstract syntax trees/structures of a computer language, whose number of objects depend on each particular phrase among the infinite phrases of the language. Also, since some language processing techniques can rely on recursion, the modeling of recursive behaviors in UML must be also faced (Tenzer and Stevens, 2006).

However, our language-driven approach has more points in common with the recent model-driven engineering approaches to the use of software modeling languages (Beydeda et al. 2005). These approaches have also been applied to the field of interactive web applications (Koch and Kraus, 2003). Indeed, in model-driven engineering it is possible to distinguish a prior metamodeling activity, where, by using a *metamodeling* language, a suitable *modeling* language is provided, which can subsequently be used for modeling particular applications. Using a domain-specific modeling language makes modeling easier since it makes the translation of domain-concepts into the language easier. A similar consideration also holds for language-driven approaches, and, in particular, for the approach proposed in the present work. Perhaps the main difference lies in the operational flavor of language-driven approaches, which leads to the precise definition of the language's operational semantics in order to allow for the subsequent implementation of a suitable language processor.

8. Conclusions and Future Work

In this paper we have presented a language-driven approach to the high-level design of interactive content-intensive applications architected according to the MVC pattern. The approach is focused on characterizing the model's structure with a language specifically designed for the kind of applications at hand, and also on describing their interactive behavior with an appropriate operational semantics for this language. The approach has been successfully used in the e-learning domain, where applications are usually based on contents with sophisticated structures, which can be used to drive the interactive behavior of the applications.

One of the main features of the language-driven approach is promoting a rational collaboration between domain-experts and developers. Indeed, the role of the application *designer* is largely carried out by the domain-experts providing and structuring the contents and planning the interactions. For instance, a *designer* of a Socratic tutor for a subject in Biology is a biologist, not a computer scientist. In their turn, developers are *designers* of domain-specific languages. Regarding the cost for a developer to create a new language and integrate it into the proposed architecture, it largely depends on the nature of the language. Indeed, developers must devise, with the collaboration of domain experts, suitable abstract structures for the language, as well as suitable concrete bindings of the language (e.g. as an XML-based descriptive markup language). More important, they must specify its operational semantics, to refine it and to abstract suitable control rules to be integrated in the final controller. Here the most critical part is to find a suitable language able to address all the expressive needs required by domain-experts. Because once the language structure and semantics have been defined, the subsequent implementation can be carried out systematically by using, for instance, the guidelines proposed in section 6. Since finding the perfect language can imply a prohibitively high initial cost associated with an exhaustive domain-analysis, we promote an iterative and incremental strategy, as described in the section 3. This strategy can contribute to alleviating these initial

costs by amortizing them during application production and maintenance, and during the production and maintenance of new applications.

The adoption of the language-driven approach (i.e. the design of new languages) depends on several factors. The most relevant one, in our opinion, is the need to actively involve domain-experts in the production and, more important, in the maintenance of the application. It also depends on the type of domain expert and on his/her prior knowledge of pre-existing languages. For instance, in technical domains it could be possible to reuse pre-existing domain-specific languages, such as we report in some of our experiences with the use of DocBook (a descriptive markup language for the elaboration of technical manuals –Walsh 99) for the production and maintenance of learning contents in the context of the <e-Aula> platform mentioned in section 3 (Martínez-Ortiz et al. 2006). This is because experts in technical domains are able to understand some of the more advanced technical details behind wider-domain languages such as DocBook. Finally, it also depends on the availability of languages in the domain that, conveniently adapted, can fill the development needs. Indeed, another strategy to leverage the cost of formulating and maintaining new languages is to reuse part or the totality of languages already proposed in the domain. For instance, the IMS standardization efforts in e-learning are largely reduced to different domain-specific languages oriented to expressing different features of an e-learning system. While in our opinion these languages can sometimes be unnecessarily expressive for the application at hand, it is still possible to reuse some of their parts, or even to reuse some of their concepts without needing to *reinvent the wheel*. In this case the effort of educating domain-experts to assimilate the pre-existing languages with respect to the effort of developing a new language specially tailored to the experts' needs should be adequately pondered. Hence two different, although complementary possibilities arise: (i) a *bottom up* approach, where the languages are specifically provided for each development scenario, and many times these languages *emerge* from the practice of structuring the contents (e.g. by marking them up with descriptive markup), and (ii) a *top down* approach, where the structure of the contents is analyzed as a preliminary step, and then suitable languages are devised to describe such a structure.

Regarding the expressive limitations of the languages obtained, it is a consequence of the domain-specific nature of the approach. For instance, the language for the Socratic tutorials introduces a particular pedagogical strategy oriented to formative learning. If any other pedagogical strategy is required, the language can be extended in new language design iterations, or, depending on the pedagogical discrepancy, a new language specially tailored to the new requirements can be developed. In this sense the approach differs from the use of *universal* languages such as IMS Learning Design (Koper and Olivier 2004), which has pedagogical neutrality as one of its main design assumptions. Again a trade-off between domain adequacy and generality arises. In the IMS Leaning Design community a way of addressing it is by formulating *patterns* of learning designs, which subsequently can be used to produce concrete designs expressed in the language (McAndrew et al. 2006). In our opinion, it is equivalent in spirit to the application of a language-driven approach. The advantage of the language-driven approach with respect to patterns in wider-domain language as IMS Learning Design is to provide better linguistic support (in effect, in last terms patterns will be referred to general linguistic constructs, which hinders usability). The main disadvantage is again the effort of designing, producing and maintaining new domain-specific languages. Here the appropriate combination of a *bottom up* and a *top down* approach could help. Indeed, currently we are investigating how to apply the approach to systems driven by suitable adaptations and projections of educational modeling languages such as the cited IMS Learning Design and other similar ones (Koper and Tattersall, 2005). Ideally, it could be facilitated by the use of mechanisms for the incremental design of the languages, which also could contribute to making the evolution of the applications easier. For this purpose, we are considering the use of the aforementioned approaches to the modularization of operational semantics (Mosses, 2004) and ways to link these with our approach to the incremental construction of translators for descriptive markup language described in (Sierra et al, 2005b). Another line of work is to align some of the principles behind the approach with current tendencies in e-learning. As discussed in (Sierra et al. 2007a), abstract syntaxes for domain-specific languages could well be identified with the *information models* that are usually developed in relation with the different IMS e-learning specifications. Indeed, these specifications are firstly characterized as suitable data models, which are subsequently mapped onto different concrete formats (usually XML-based descriptive markup languages). Finally, we are also working on the development of a systematic way to encode prototypes for the application's controllers based on the operational semantics specifications. We follow patterns similar to those of the already mentioned works of Clément et al. (1986) on the prototyping of *natural semantics* of programming languages to carry out rapid prototyping using the Prolog language (Sterling and Shapiro 1994). We also use standard Prolog static metaprogramming facilities to define a concrete syntax embedded in Prolog that can be also used by domain-experts in order to involve them in the prototyping activity. Finally, we use Prolog's coroutining mechanisms to emulate the communication channels with the application's model and view. This approach is detailed in (Sierra et al. 2007b).

Acknowledgements

The Spanish Committee of Education and Science (Projects TIN2004-08367-C02-02 and TIN2005-08788-C04-01) and the Regional Government / Complutense University of Madrid (research group 910494) have partially supported this work.

References

Abelson, H., Sussman, G.J., 1996. Structure and Interpretation of Computer Programs, Second Edition. MIT Press.

Abrial, J.R., 1996. The B-Book: Assigning Programs to Meaning. Cambridge University Press

Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D., 2007. Compilers: principles, techniques and tools (second edition). Addison-Wesley.

ADL-SCORM, 2003. Advanced Distributed Learning - Shareable Content Object Reference Model. Faulkner Information Services.

Bastide, R., Navarre, D., Palanque, P., 2003. A Tool-Supported Design Framework for Safety Critical Interactive Systems. Interacting with Computers, 15, 309-328.

Beydeda, S., Book, M., Gruhn, V (eds)., 2005. Model-driven software development. Springer

Booch, G., Rumbaugh, J., Jacobson, I.,1998. The Unified Modeling Language User Guide. Addison-Wesley.

Bork, A.,1985. Personal Computers for Education. Harper & Rows.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E. (Eds), 2000. Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation.

Ceri, S., Fraternali, P., Bongio, A., 2000. Web Modeling Language (WebML): a Modeling Language for Designing Web Sites. Computer Networks 33, 137-157.

Clark, J. (Ed), 1999. XSL Transformations (XSLT) Version 1.0. W3C Recommendation.

Clark, T., Evans, A., Sammut, P., Willans, J.,2004. An eXecutable Metamodeling Facility for Domain Specific Language Design. The 4th OOPSLA Workshop on Domain-Specific Modeling. Technical Report TR-33, University of Jyväskylä, Finland.

Clark, T., Warmer. J (eds), 2002. Object Modeling with the OCL: The Rationale behind the Object Constraint Language. Lecture Notes in Computer Science 2263. Springer

Clément, D., Despeyroux, J., Despeyroux, T., Hascoet, L., Kahn, G, 1986. Natural Semantics on the Computer. Proceedings of the France-Japan AI and CS Symposium.

Collier, G.,2002. e-Learning Application Infrastructure. Sun Microsystems White Paper.

Comai, S., Fraternali, P.,2001. A Semantic Model for Specifying Data-Intensive Web Applications Using WebML. International Semantic Web Working Symposium.

Conallen, J.,1999. Modeling Web Application Architectures with UML. Communications of the ACM 42(10), 63-70.

Coombs, J. H., Renear, A. H., DeRose, S. J.,1987. Markup Systems and the Future of Scholarly Text Processing. Communications of the ACM, 30 (11), 933-947.

Coutaz, J.,1987. PAC, An Object Oriented Model for Dialog Design. Proceedings of INTERACT'87, 431-436.

Cowan, D. D., Lucena, C. P.,1995. Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse. IEEE Transactions on Software Engineering 21(3), 229-243.

Deutsch, A., Sui, L., Vianu, V., Zhou, D.,2006. A System for the Specification and Verification of Interactive, Data-Driven Web Applications. 2006 ACM SIGMOD International Conference on Management of Data, 772-774.

Doubrovski, V., 2001. Towards Formal Specification of Client-Server Interactions for a Wide Range of Internet Applications. Proceedings of the First Asia-Pacific Conference on Web Intelligence. Lecture Notes in Artificial Intelligence 2198, 153-162.

Draheim, D., Fehr, E., Weber, G.,2003. Improving the Web Presentation Layer Architecture. Proceedings of the 5th Asia-Pacific Web Conference. Lecture Notes in Computer Science 2642, 324-332.

Draheim, D., Weber, G., 2005. Modeling form-based interfaces with bipartite state machines. Interacting with Computers 17, 207-228

Fernández-Valmayor, A., López Alonso, C., Sèrè, A., Fernández-Manjón, B., 1999. The Design of a Flexible Hypermedia System. Proceeding of the IFIP WG3.2-WG3.6 Conference: Building University Electronic Educational Environments, 51-66.

Friesen, N., 2005. Interoperability and Learning Objects. An Overview of e-Learning Standardization. Interdisciplinary Journal of Knowledge and Learning Objects 1., 23-31.

German, D. M., 2002. Using Hadez for Formally Specify the Web Museum of the National Gallery Art. 2nd International Workshop on Web Oriented Software Technology.

Green, M., 1987. A survey of three dialogue models. ACM Transactions on Graphics 5(3), 244-275.

Harel, D., 1987. Statechars: A visual formalism for complex systems. Science of Computer Programming 8, 231-274.

Horrocks, I., 1999. Constructing the user interface with statecharts. Addison-Wesley.

Hussey, A. Carrington, D.,1997. Comparing the MVC and PAC Architectures: a Formal Perspective. IEE Proceedings – Software Engineering 144(4). 224-236

Ibrahim, B.,1989. Software Engineering Techniques for CAL. Education & Computers 5, 215-222.

Jacobs, R.J.K., 1983. Using formal specifications in the design of a human-computer interface. Communications of the ACM 26(4), 259-264.

Koch, N., Kraus, A.,2003. Towards a Common Metamodel for the Development of Web Applications. Proceedings of the 3rd Conference on Web-Engineering. Lecture Notes in Computer Science 2722, 497-506.

Koper, R., Olivier, B.,2004. Representing the learning design of units of learning. Educational technology & society 7(3), 97-111.

Koper, R., Tattersall, C. (Eds), 2005. Learning Design: A Handbook on Modeling and Delivering Networked Education and Training. Springer.

Krasner, G.E., Pope,T.S., 1988. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk 80 System. Journal of Object Oriented Programming 1(3), 26-49.

Lawrence, C.P., Grabczewski, K, 1996.. Mechanizing Set Theory. Journal of Automatic Reasoning. 17(3), 291-323.

Markopoulos, P., Rowson, J., Johnson, P., 1997. Composition and Synthesis with a Formal Interaction Model. Interacting with Computers, 9, 197-223.

Martínez-Ortíz, I., Moreno-Ger, P., Sierra, J. L., Fernández-Manjón, B., 2006. <e-QTI>: A Reusable Assessment Engine. Proceedings of the 5th International Conference on Web-based Learning. Lecture Notes in Computer Science 4181, 34-145.

Martínez-Ortiz, I., Moreno-Ger, P., Sierra, J.L., Fernández-Manjón, B., 2006. Using DocBook and XML Technologies to Create Adaptive Learning Content in Technical Domains. International Journal of Computer Science and Applications, 3(2), 91-108.

McAndrew, P., Goodyear, P., Dalziel, J., 2006. Patterns, Designs and Activities: Unifying Descrptions of Learning Structures. International Journal of Learning Technology 2(3), 216-242

Mauw, S., Wiersma, W. T., Willemse, T. A. C., 2004. Language-driven System Design. International Journal of Software Engineering and Knowledge Engineering, 14(6), 625-664.

Mernik, M., Heering, J., Sloane, A.M., 2005. When and how to Develop Domain-Specific Languages. ACM Computing Surveys, 37(4), 316-344.

Mitschele-Thiel, A., 2001. Systems Engineering with SDL, John Wisley & Sons

Moreno-Ger, P., Sierra, J. L., Martínez-Ortiz, I., Fernández-Manjón, B., 2007. A Documental Approach to Adventure Game Development. Science of Computer Programming 67(1), 3-31

Mosses, P. D., 2004. Modular structural operational semantics. Journal of Logic and Algebraic Programming, 60-61, 195-228

Mosses, P. D., 2006. Formal Semantics of Programming Languages: An Overview. Electronic Notes in Theoretical Computer Science 148(1), 41-73.

Naur, P (ed)., 1960. Report of the Algorithmic Language ALGOL-60. Acta Polythecnica Scandinavica. Applied Mathematics and Computing Machiney Series. N. 5.

Navarro, A. Fernández-Valmayor, A. Fernández-Manjón, B. Sierra, J. L., 2004. Conceptualization prototyping and process of hypermedia applications. International Journal of Software Engineering and Knowledge Engineering, 14(6), 565-602

Parnas, D., 1969. On the use of transition diagrams in the design of a user interface for an interactive computer system. Proceedings of the 1969 24th national conference. ACM press, 379-385

Paulson, L. D., 2005. Building Rich Web Applications with AJAX. IEEE Computer, 38(10), 14-17.

Plotkin, G.D., 1981. An Structural Approach to Operational Semantics. Technical Report DAIMI FN-19. Computer Science Dept. Aarhus University.

Redouane, A., 2004. Towards a New Method for the Development of Web-Based Applications. Third IEEE International Conference on Cognitive Informatics, 116-122.

Rezazadeh, A., Butler, M., 2005. Some Guidelines for Formal Development of Web-Based Applications in B-method. 4th International Conference of B and Z Users. Lecture Notes in Computer Science 3455, 472-492.

Sierra, J. L. Fernández-Valmayor, A., Guinea, M., Hernánz, H., 2006. From Research Resources to Virtual Objects: Process model and Virtualization Experiences. Journal of Educational Technology & Society 9(3), 56-68.

Sierra, J. L., Fernández-Manjón, B., Fernández-Valmayor, A., Navarro, A, 2005. Document-Oriented Construction of Content-Intensive Applications. Int. Journal of Software Engineering and Knowledge Engineering, 15(6), 975-993.

Sierra, J.L., Fernández-Manjón, B., Fernández-Valmayor, A., 2007. Language-driven development of web-based learning applications. Proceedings of the 6th International Conference on Web-Based Learning. University of Edinburgh, United Kingdom.

Sierra, J. L., Fernández-Valmayor, A., Fernández-Manjón, B., 2006. A Document-Oriented Paradigm for the Construction of Content-Intensive Applications. Computer Journal, 49(5), 562-584.

Sierra, J.L., Fernández-Valmayor, A., Fernández-Manjón, B., 2007. How to prototype an educational modeling language. Proceedings of the IX International Simposium on Computers in Education. Porto, Portugal

Sierra, J. L., Moreno-Ger, P., Martínez-Ortiz, I., Fernández-Manjón, B., 2007. A Highly Modular and Extensible Architecture for an Integrated IMS-based Authoring System: The <e-Aula> Experience. Software-Practice & Experience 37(4), 441-461.

Sierra, J. L., Navarro, A., Fernández-Manjón B., Fernández-Valmayor, A., 2005. Incremental Definition and Operationalization of Domain-Specific Markup Languages in ADDS. ACM SIGPLAN Notices, 40(12), 28-37.

Spivey, J.M., 1992. The Z Notation: A Reference Manual, 2nd Edition. Prentice-Hall

Sterling,L. Shapiro,E. The Art of Prolog. MIT Press. 1994

Syriani, J. A., Mansour, N., 2003. Modeling Web Systems Using SDL. Eighteenth International Symposium on Computer and Information Sciences. Lecture notes in Computer Science 2869, 1019-1026.

Tenzer, J., Stevens, P., 2006. On modeling recursive calls and callbacks with two variants of Unified Modeling Language state diagrams. Formal Aspects of Computing 18(4), 397-420.

Van Deursen, A., Klint, P., Visser, J., 2000. Domain-Specific Languages: An Annotated Bibliography. ACM SIGPLAN Notices, 35(6), 26-36.

Walsh, N., Muellner, L., 1999. DocBook, the definitie guide. O'Reilly.

Wasserman, A.I., 1985. Extending state transition diagrams for the specification of human-computer interaction. IEEE Transactions on Software Engeneering 11(8), 699-713.

Woods, W.A., 1970. Transition network grammars for natural language analysis. Communications of the ACM 13(10), 591-606.