A Document-Oriented Paradigm for the Construction of Content-Intensive Applications

JOSÉ LUIS SIERRA, ALFREDO FERNÁNDEZ-VALMAYOR, AND BALTASAR FERNÁNDEZ-MANJÓN

Dpto. Sistemas Informáticos y Programación. Fac. Informática. Universidad Complutense de Madrid
C/ Profesor José García Santesmases s/n. 28040 Madrid, Spain
Email: jlsierra@sip.ucm.es

In this paper we describe our work on the formulation of a *document-oriented paradigm* for improving the construction and maintenance of *content-intensive applications* (i.e. applications that make intensive use of the information provided by the experts in a given domain: the *contents*). According to this paradigm, the development of a content-intensive application must be the result of close collaboration between two kinds of actors: *domain experts* and *developers*. The goal of this collaboration is the authoring of (i) a set of documents describing the most relevant aspects of the application (i.e. the contents and other relevant customizable features); (ii) a grammar describing a domain – specific markup language that will be used to make the structure and the data in these documents explicit; and (iii) a suitable processor for this language. The final running application will be automatically produced by processing the marked documents with this processor. The use of this paradigm in the development of content-intensive applications can increase the initial cost of application production, but in the long run it can substantially improve maintenance and portability, and promote information and software reuse as well. We have successfully applied this paradigm to the development of educational and hypermedia applications, and knowledge-based systems. From these experiences, we have found that the feasibility of the paradigm depends to a great extent on having mechanisms that enable the incremental definition of the markup languages and the incremental construction of their processors. This has led us to the formulation of a document-oriented approach for the development of content-intensive applications tightly coupled with these principles of incremental formulation and operationalization of domain-specific markup languages.

1. INTRODUCTION

There are applications (e.g. educational applications) that integrate collections of highly structured documents regarding a given domain. These documents are usually authored by experts in this domain, and for an application of this kind, the processes involved in updating, maintaining and fine-tuning can be more costly and critical than those involved in its initial development. We shall call this type of applications *content-intensive*. For the last ten years our research efforts have been oriented on a search for mechanisms to enhance the production and maintenance of this kind of applications. The entire life cycle of content-intensive applications greatly depends on the existence of appropriate mechanisms to organize efficient communication between the different actors involved in their development: *domain experts*, who are the owners and/or the authors of the contents, and *software developers*, who are the experts in computer science responsible for building, maintaining and updating the final application. We have found that descriptive markup technologies are the adequate support technology for the development of this kind of applications, because they empower the representation of the contents as human-readable documents, understandable by domain experts and able to be processed by developers [1]. In this way, domain experts provide developers with the contents' documents, (documents with all the information the application deals with), while the developers document other *operational* aspects of the application not derivable from the contents (e.g. the relevant features of the application's user interface). The resulting documents describe the operational aspects of the application and can be subsequently understood and modified by experts so as to notify developers of possible changes or additions to these operational aspects. All these documents are marked up using an appropriate, easy-to-use, *domain-specific descriptive markup language* (DSML). This makes the production and maintenance of a running application possible by automatically processing those documents with a suitable processor for this language.

We have successfully applied this document-oriented paradigm to the development of several educational and hypermedia applications as well as to knowledge-based systems. These experiences have revealed that although the use of this paradigm can increase the initial production costs of an application, in the long run it can substantially facilitate its maintenance and

portability. These experiences have also revealed that for the feasibility of the paradigm it is of critical importance to adopt an incremental strategy for the formulation of the defined languages mirroring the changing markup needs of domain experts and developers that arise during the creation and the entire life cycle of the application. Thus, the documental description of an application and the use of formal languages for describing the structure of the application's documentation -instead of for programming the application itself, tightly coupled with techniques for the incremental definition of these languages and the modular construction of their processors, constitute the main contribution of the work we present here. Our experiences in applying the paradigm to different kinds of projects have led us to systematize all this work with the formulation of the ADDS approach, the PADDS technique and the OADDS operationalization model. ADDS (*Approach to Document-based Development of Software*) is a generic approach to regulating document-oriented development. Therefore this approach strongly relies on the selection, adaptation and/or definition of markup languages for each application domain, as well as on the construction of their processors. PADDS (*DSML Provision in ADDS*) is a technique for the modular definition and adaptation of such languages. In turn OADDS (*Operationalization in ADDS*) is a model that regulates the modular construction of their processors.

In this paper we present the mentioned systematization and the other different aspects involved in the formulation of our document-oriented paradigm. The paper is organized as follows: Section 2 motivates the document-oriented paradigm. Section 3 summarizes our main experiences with the paradigm in different domains. Section 4 describes the systematization of the paradigm that has arisen from these experiences. Section 5 outlines a qualitative evaluation of our proposal based on our experimental work. Section 6 presents some related work. Finally section 7 provides some conclusions and lines of future work. The development of applications for travel recommendation in subway networks will be used as a case study throughout the paper to illustrate the different aspects discussed.

2. MOTIVATING THE DOCUMENT-ORIENTED PARADIGM

In this section we motivate the document-oriented paradigm. We start by describing the communication problems that arise between domain experts and developers during the production and maintenance of content-intensive applications (subsection 2.1). Then we propose the aforementioned paradigm as a solution that combines the use of descriptive markup technologies in the publishing domain with the approach to software development based on domain-specific languages (subsection 2.2).

2.1. The Production and Maintenance of Content Intensive Applications

Typical scenarios for full-custom development of content-intensive applications demand a strong interaction between experts and developers (Figure 1). In these scenarios, domain experts provide developers with the application contents. Then the developers analyze these contents and build an initial application, which is evaluated by the domain experts. As a result of this evaluation, the domain experts can propose modifications and/or improvements in the application to the developers, who produce an enhanced application, starting a new evaluation. This iterative behavior, characterized by the production / modification of applications, will be called the *production loop*. This loop eventually finishes when a satisfactory application has been obtained but it will need to be started again when the application needs to be updated. This scenario is exemplified in the subway case study. Here the *network organizers* (domain experts) can provide the developers with the documents on the organization of the network (structure, timing information, schedules, etc.), who produce an application for travel recommendations in this network. Possible modifications can affect the encoding of the network's information, as well as operational aspects (e.g. information messages in the user interface).

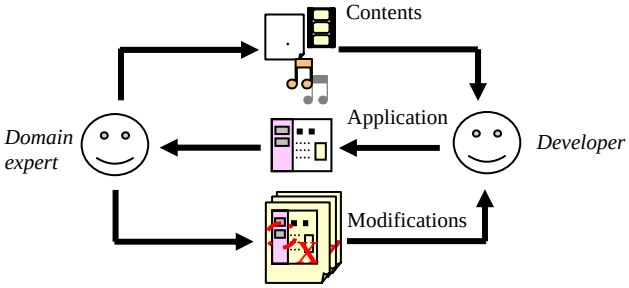

FIGURE 1. A typical scenario for full-custom development of content-intensive applications

The main advantage of the full-custom approach is its flexibility, because it avoids the expressive limitations imposed by particular authoring tools. Our experience indicates that this flexibility becomes critical for the development of complex applications, such as those described in [2]. However, this approach exhibits several difficulties that might decrease productivity:

- *Low efficiency of the production loop*. For instance, in applications that manage high volumes of contents and exhibit complex interactions with the user, when domain experts want to make a change in the contents they must usually annotate their modifications directly onto snapshots of the applications' screens [2]. These annotations are manually analyzed by the developers to produce the proper application.

- *Poor reusability of the contents*. Application contents can be represented in heterogeneous and proprietary formats (they could even be printed material). In addition, contents may need to be encoded again in new application-dependent formats by the developers. This makes it difficult for contents to be reused in the development of new applications and even in future evolutions of the same application, thus leading to one of the biggest problems of content-intensive applications: the brevity of their life cycle. A typical example can be seen in the development of educational software, where it is common to see how excellent material, obtained at an enormous cost employing multidisciplinary groups of experts, quickly becomes obsolete upon the appearance of new technology on the market (e.g. the transition from videodisc or CD-ROM to a web-based environment) [3][4].
- *Low software reusability*. This is because the application's programs are usually very dependent on the particular application at hand, making it difficult, if not impossible, to reuse them in the development of other applications.

These shortcomings are difficult to avoid with conventional, developer-oriented, software technologies. For example, object-oriented frameworks [5], widely used to improve software reusability, are not specially suited for improving communication between experts and developers, and modeling languages like Unified Modeling Language, UML [6], which provide very valuable artifacts for improving communication between software developers are poorly understood by experts in other knowledge areas. While these technologies continue to be valuable for developers, they must be complemented with more human-centered mechanisms if we want to fully include domain experts in the production loop.

2.2. The document-oriented paradigm

Our solution to the production and maintenance of content-intensive applications has been inspired by modern electronic publishing techniques as well as by the approach to software development based on domain-specific languages:

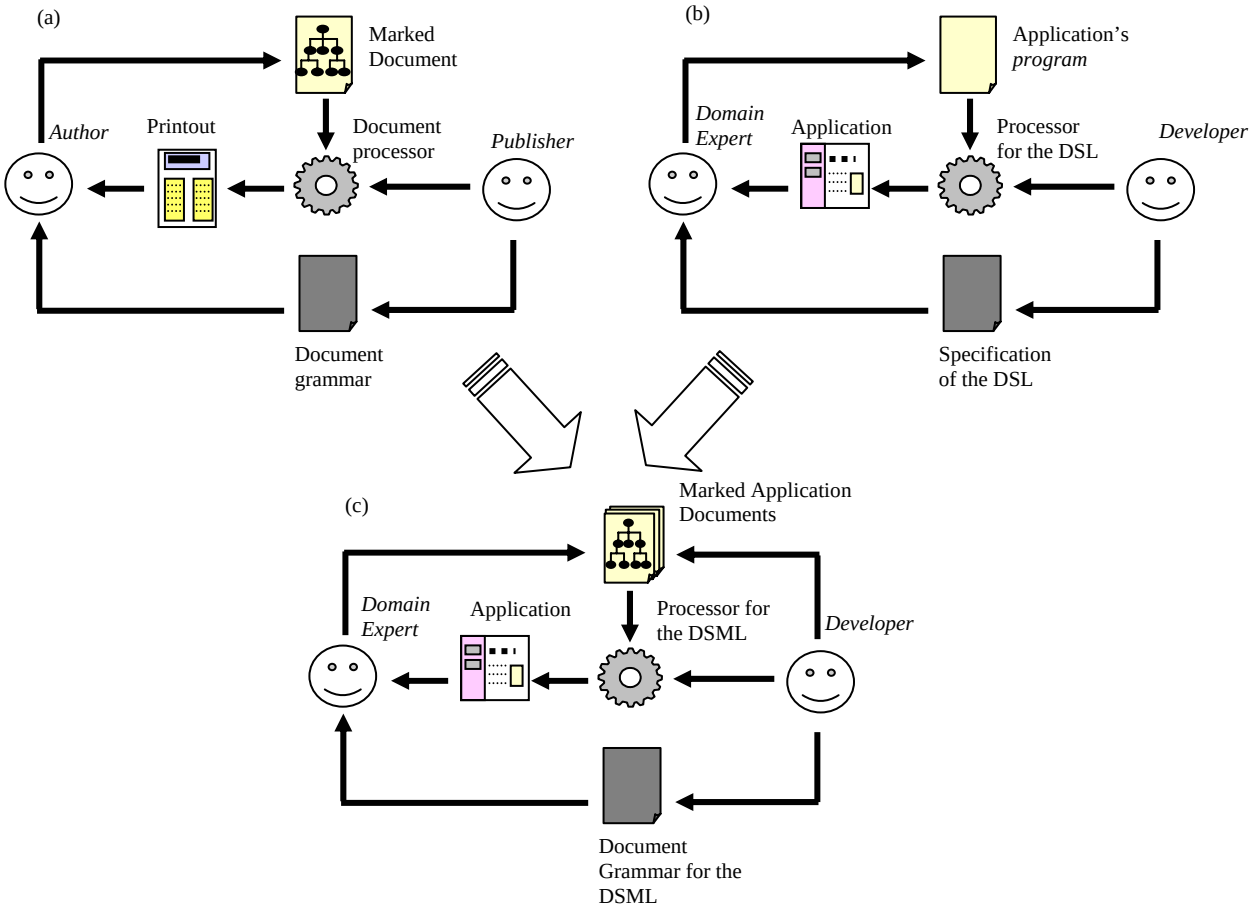

FIGURE 2. (a) Use of descriptive markup languages in the electronic publishing domain; (b) Use of domain-specific languages in software development ; (c) The document-oriented construction of content-intensive applications blends the two approaches.

- The publishing industry has faced similar problems regarding the full-custom development of content-intensive applications for centuries. The advent of information and communication technologies and electronic documents brings to publishing new opportunities for improving the production loop. For these, the use of descriptive markup languages is especially relevant [7][8]. Descriptive markup languages are focused on the description of the logical structure of the documents instead of on how these documents must be finally processed. This structure is generally understood as a hierarchical arrangement of *elements* with optional *attribute-value* pairs attached to them. These elements and their

associated attributes are *marked up* on the document using *tags*, which are combined according to the grammatical rules of the language. As these languages are involved with structure instead of with processing, they are easy to understand and easy for authors to use. Figure 2a outlines how descriptive markup languages can be used to involve authors in the publishing process. Here the publisher provides a descriptive markup language characterized as a *document grammar*. This grammar describes the structures allowed for this type of documents and can be defined by means of a DTD, *Document Type Definition* [8][9], or any other suitable grammatical formalism [10]. The publisher also provides a *document processor* that can be applied to the marked documents in order to produce printouts automatically. Thus, the author can markup its original, the resulting marked document can be validated against the document grammar, and the printout can be automatically produced using the processor. Because the document conforms the markup language, the defects remaining in the printouts are due to misunderstandings either in the contents or in the intentional (although grammatically correct) use of the markup. However, as the markup is descriptive, both types of defects can be solved by the author on the marked document (either directly or by using a specialized editor, like [11]), letting the publisher produce new printouts at no cost.

- *Domain-specific languages* (DSLs) have been recognized as very valuable mechanisms for involving domain-experts in the development process [12][13][14][15]. According to [15], a DSL is a *programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.* As suggested in Figure 2b, application development according to the DSL approach closely resembles publishing based on descriptive markup technologies. The focus on the domain lets domain-experts use DSLs to *program* applications in these languages. In turn, the roles of developers are to design suitable DSLs for the application domains at hand, as well as to provide suitable processors for these DSLs.

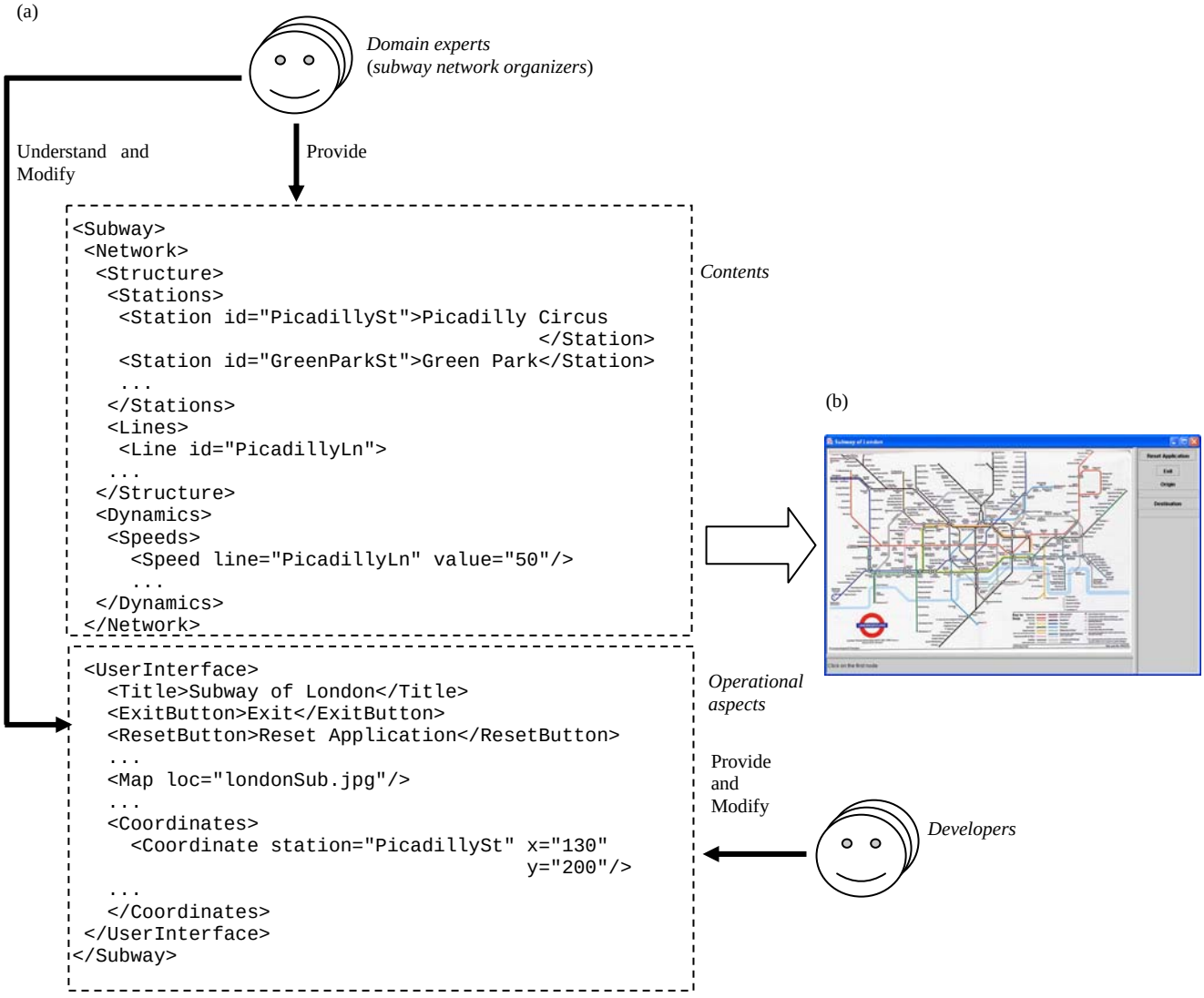

FIGURE 3. (a) *Documentation* of the traveling recommendation application of the London subway network; (b) Application documented in (a).

The document-oriented paradigm blends the DSL approach with the use of descriptive markup languages in the electronic publishing domain as suggested in Figure 2c, which depicts a first approach to this paradigm. Here the documents are not

restricted to textual prose, but enclose application contents and other operational aspects of the application. These *application documents* are marked up with an *application* DSML that is specific for (and varies with) each type or family of applications. Final running applications are then automatically produced by processing these marked documents in the same way that printouts are produced from the marked originals in Figure 2a, or that applications are generated from programs in the DSLs in Figure 2b. Notice that developers also play an active role in the authoring of the application documents since they provide and maintain the documentation of the operational aspects. As pointed out in the introduction, domain experts are able to understand and maintain part of this operational documentation. Note that the key point here is the readability of descriptive markup, which constitutes the basis of the privileged communication channel that lets domain experts communicate modifications and collaborate with developers in tuning up most application details.

This collaborative *documentation* process is schematized in Figure 3a using the subway case study for the subway network of London. The subway network organizers document the relevant aspects of the network (e.g. structure and dynamics) and mark these up with the DSML for this application domain. In turn, developers initially document and mark up the relevant operational aspects of the application (e.g. properties of the user interfaces such as names of buttons and labels, informative messages, a pointer to the image with the map and the coordinates of the stations in this image[1]). The use of the appropriate descriptive markup makes it possible for all these aspects to be subsequently understood by network organizers, who, together with the developers, can modify these to fine-tune the application's appearance. This leads to the authoring of documents like the one partially shown in Figure 3a. These documents are automatically processed to generate traveling recommendation applications like the one depicted in Figure 3b.

The document-oriented paradigm palliates some of the shortcomings of the full-custom approach described above:

- Firstly, the efficiency of the production loop is increased because modifications in the applications can be directly annotated on their documents, and these are immediately reflected in the applications produced with no effort. In this way production and maintenance loops become the same and can be extended at almost no cost for the entire life cycle of the application.
- Secondly, well-established markup standards (SGML [8] and XML [9]) can be used to improve the reusability of the contents.

(a) (b)

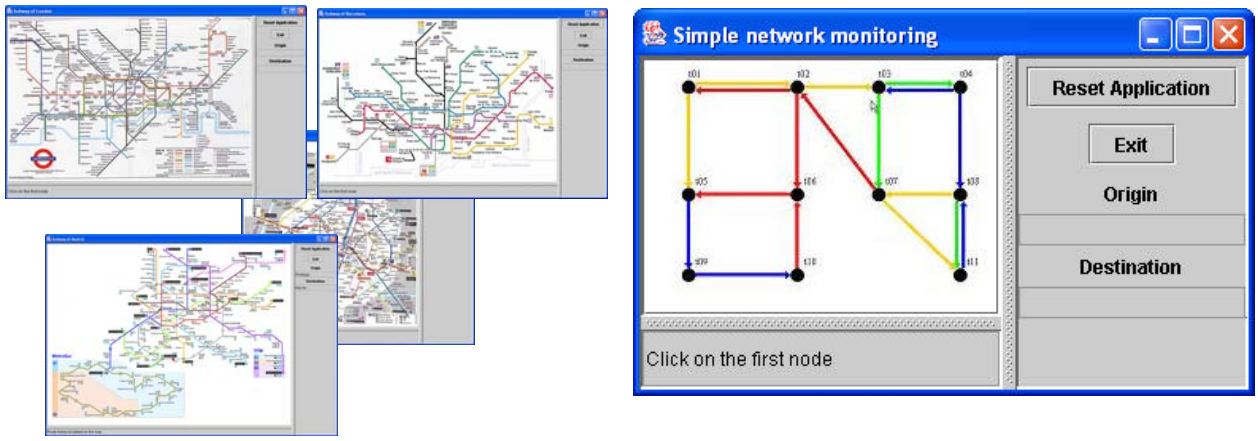

FIGURE 4. (a) The DSML for a given domain can be used to produce multiple applications in this domain; (b) A DSML can integrate sublanguages that can be reused in the definition of new languages for other application domains.

- Lastly, a DSML can be thought of as an explicit characterization of a family of content-intensive applications in a given domain, and each application can be executed from its documentation using the same processor for that DSML, thus sharing the benefits of the generative approaches to software reuse [12][16]. In effect, the DSML used in the subway case study will allow the production of multiple applications in this domain, one for each subway network (Figure 4a). Furthermore, a DSML can integrate different sublanguages that can be reused together with their associated processors in the definition of other languages. A typical scenario arises when the sublanguage for marking up the application's contents is reused for the definition of other DSMLs. For example, the sublanguage for marking up subway networks can be reused in a DSML for a geographical information system. Another one arises when the sublanguage for marking up the operational aspects is combined with a sublanguage for marking up a different kind of contents. For example, the sublanguage for marking up the operational aspects in the subway example can be reused in the formulation of a markup language for a simple computer network monitoring application (Figure 4b). Nevertheless, it is important to note that these reuse-oriented uses of the document-oriented paradigm are a collateral consequence of its main aim: to manage the production and maintenance of the successive releases of a single content-intensive application.

[1] Since we are not referring to geographical coordinates but to presentational ones, they are better documented in the GUI instead of in the subway network.

Despite all these advantages, if the DSML is initially pre-established and can not be extended, the paradigm is equivalent to using a particular authoring tool whose inner representation/interchange format is established by the language. Thus, the preservation of the full-custom approach's flexibility requires a DSML *evolution* to accommodate the changing expressive needs of domain experts and developers. Instead of being conceived of as an immovable entity, the markup language for an application domain must be considered as a dynamic object that evolves during the entire life cycle of the applications in this domain, adapting the markup needs as they are discovered. Not only does this evolution affect the descriptive nature of the language, but its operational aspects (i.e. its processor) as well. Therefore, the document-oriented paradigm is exposed to the same shortcomings as those in the DSL approach: the high initial costs associated with the explicit formulation, operationalization and maintenance of the language [15]. Its success will greatly depend on the appropriate management of the incremental provision of DSMLs and on the incremental development of their processors. We have realized all these considerations during our experiences with the paradigm described in the next section.

3. EXPERIENCING THE DOCUMENT-ORIENTED PARADIGM

In this section we briefly outline some representative examples of how our research group at the Complutense University of Madrid (Spain) has applied the document-oriented paradigm for the last ten years. These experiences include: production, maintenance and portability of content-rich educational hypermedia applications (subsection 3.1), prototyping of hypermedia applications (subsection 3.2), maintenance of knowledge models during the development of knowledge-based systems (subsection 3.3), construction of repositories of learning objects in specialized domains (subsection 3.4), definition of domain-specific learning profiles in learning management systems (subsection 3.5), and production and maintenance of adventure videogames with an educational use (subsection 3.6).

3.1. Production, maintenance and portability of educational hypermedia

The document-oriented paradigm as described in this paper was formerly used to improve the maintenance of *Lire en Français* [2][17][18], an educational hypermedia developed in the context of the EU Socrates-Lingua Project *Galatea* (*Apprentisage de la compréhension en langues voisines – Learning to comprehend familiar tongues*). *Lire en Français* was oriented to allow learners whose native tongue was a Romance language to understand texts written in French. The development of this system involved two communities with very different expertise: *linguists* and *developers*. The linguists were responsible for providing the texts together with their associated structures (e.g. contextual dictionaries), as well as the pedagogical strategy followed in the system. The developers were responsible for implementing the actual system.

To improve the communication loop between linguists and developers suitable types for electronic documents were identified, both for the texts and the contextual dictionaries. In addition, the structure of these documents was formalized using SGML DTDs, which were used for the linguists to mark up their documents. In order to automatically incorporate the modifications in the documents inside the production system, developers built suitable processors for the SGML-based markup languages used. This drastically decreased the maintenance effort. Indeed, while with the traditional process an average-complexity change in the contents and/or in the interface could take about a week due to communication overloads, with the document-oriented approach it could be carried out in a few hours. Later in the project we realized that not only could the contents be documented, but the high-level features of the user interface as well. Therefore we evolved our initial DSML, as well as our processing software, to tackle the markup of these additional aspects.

The document-oriented development of the CD release of *Lire en Français* was very valuable in deploying the application in a web environment. This experience was carried out in the context of the project SIMBA (*Sistema de Información Basado en Lenguajes de Marcado y Agentes – Information System based on Markup Languages and Agents*), a research project sponsored by the Spanish Committee of Science and Technology. The new web release of the system was rebuilt from the documents produced for the original CD-based system. With this we realized the benefits of the document-oriented approach in reusing contents and other features, when porting a content-intensive application to other environments. In this new scenario, the DSML and its processor also experienced further evolutions.

3.2. Model-driven prototyping of content-intensive hypermedia applications

During our experiences in the educational domain we also realized the importance of modeling the interaction with contents in an implementation-independent and highly conceptual way. This led us to look for ways of modeling the separation of the main concerns (i.e. contents and navigation) in a content-intensive hypermedia application. As a result of this effort we formulated *Pipe*, a conceptually simple and usable hypermedia modeling framework promoting this separation [19][20][21]. Pipe's models separate contents, navigational schemas, and the mapping functions relating them. Such models are the basis for a model-driven development process of hypermedia applications. The most characteristic aspect of this development process, which is called *Plumbing*, is to promote the iterative model-driven formulation of a prototype of the final application, which sets the requirements for the routine development of the final production system.

Prototyping in *Plumbing* is the result of a collaboration process between *hypermedia designers* and *developers*. To enable this collaboration, suitable ways of describing the different components of the Pipe models must be established. In the *Plumbing XJ* incarnation of the abstract *Plumbing* process model we have chosen a naïve version of the document-oriented paradigm [21]. For doing so, we have devised an XML-based *navigation* DSML for structuring documents with the

navigational schema. In addition, we promote the formulation of application-specific XML markup languages for structuring the contents of each particular application. Prototypes are then automatically generated from documents with the contents and documents with the navigation schemas by using a Java-written processor called AGP (*Automatic Generator of Prototypes*). At a first stage, only static aspects of prototyped applications were considered. Later we evolved the navigation DSML as well as the AGP processor to cope with dynamically generated contents.

3.3. Document-oriented maintenance of knowledge models

Knowledge-based systems constitute a typical example of an application domain where the interaction between domain-experts and developers is especially critical [22]. During the nineties several model-driven approaches were proposed, which conceive the development of a knowledge-based system as the explicit formulation of a *knowledge mode* able to tackle and structure the different sorts of knowledge comprised by such a system [23]. This model must be subsequently translated into a suitable implementation-oriented representation. Therefore the participants in the development process (*knowledge engineers* and *system developers*) must face the maintenance problems derived from the translation of the changes in the model into the underlying implementation. As reported in [24], in the document-oriented paradigm we have found a pragmatic solution for carrying out this task. According to this solution, suitable document types for embodying each type of knowledge dictated by the model are identified and suitable DSMLs for structuring these documents are designed. In addition, we structure the system's inference engine as a processor for the markup languages used.

We have conducted several experiments to test the document-oriented maintenance of knowledge models during the development of knowledge-based systems, centered on the route recommendation application described in the present work. As indicated in [25], during these experiments the markup language devised as well as the associated processor went through several evolutions, both in the markup of the subway network and in the markup of the main features of the user interface.

3.4. Construction of repositories of learning objects in specialized domains

Chasqui[2] [26][27][28] is a family of systems for the construction of domain-specific repositories of learning objects [29]. The members of this family have been used to build repositories associated with several academic museums in the Complutense University[3]. This family of applications has been developed in the context of two projects: REI – MLH (*Recursos Educativos e Informativos basados en componentes distribuidos: Metodologías, Lenguajes y Herramientas – Educational and Informational Resources based on distributed components: Methodologies, Languages and Tools*), funded by the Spanish Committee of Science and Technology, and *Chasqui 2*, sponsored by the Committee of Industry and Tourism and carried out in collaboration with Telefónica I+D (the research and development branch of the first Spanish telecom company).

The construction of a repository according to the Chasqui systems starts with the formulation of a conceptual representation-independent model of a learning object. This model is then mapped into a particular implementation. In accordance with the document-oriented paradigm, this implementation supposes the formulation of DSMLs for structuring the educational resources, as well as the reuse of already available e-learning standards (e.g. LOM, *Learning Object Metadata* [30], for representing the metadata associated with the objects, or IMS CP, *Content Packaging* [31], for packaging them).

In the first Chasqui systems learning objects were directly mapped onto database representations. While the systems let users author learning objects using domain-specific authoring tools, we found serious difficulties regarding portability and interoperability with other systems and authoring tools. This led us to strongly adopt the document-oriented paradigm in the new releases of the systems. These new releases supported exportation and importation of learning objects using web services [32] that are implemented as processors for the markup languages structuring such objects. Currently we are also considering the documentation of other features of the application, which would allow us to describe whole Chasqui systems as marked documents as well as to generate them with an evolved processor.

3.5. Formulation of domain-specific application profiles in an IMS-based experimental learning management system

<e-Aula> [33][34] is an experimental learning management system implementing several IMS e-learning specifications. This system was initially developed in the context of the project <e-Aula> (*Entorno de aula virtual basado en lenguajes de marcado –XML– y estándares educativos – Virtual Classroom Environment based on markup languages – XML- and learning standards*), funded by the Spanish Committee of Science and Technology. Ongoing improvements are being carried out in the context of the project Metalearn *(Metodologías, Arquitecturas y Lenguajes para la creación de Servicios Adaptativos para e-learning – Methodologies, Architectures and Languages for the development of Adaptative Services in e-learning*), sponsored by the Committee of Education and Science. <e-Aula> is also equipped with an authoring system, which offers basic functionalities to edit, play, export and import IMS content. This system is constructed according to the document-oriented paradigm to yield a highly modular and extensible environment capable of being specialized in many

[2] *Chasqui* means *messenger* in *Quechua*, the language spoken in the *Inca Empire*.

[3] Two are currently available to the public: museum of Archeology *Antonio Ballesteros* (http://macgalatea.sip.ucm.es/chasqui), and museum of history of Computer Science *José García Santesmases* (http://www.fdi.ucm.es/migs/).

different domain-specific learning scenarios.

The <e-Aula> authoring system is organized following a so-called *manifest driven approach* [35]. In this architecture the different functionalities of the system are driven by the IMS *manifests* attached to the IMS contents. An IMS manifest is an XML document structuring the educational contents inside an IMS content package [31]. Thus the core of the system is made up of four processors for this language, one processor per functionality. In addition, this approach promotes the specialization of the IMS contents in different domain-specific learning scenarios by defining specialized content types, together with the corresponding DSMLs used to structure them. This specialization is mirrored in the evolution of the aforementioned processors for enabling the edition, playing, importation and exportation of the new learning material. We have tested this mechanism in supporting static HTML content, domain-specific static XML based-content (e.g. XML documents containing frequent-asked questions), moderately interactive content as assessments expressed in IMS QTI, *Question Test Interoperability* [36], or highly interactive content such as the educational adventure videogames described in the next subsection.

3.6. Document-oriented development of adventure videogames

<e-Game> [37][38] is a system for the rapid development of educational-oriented adventure videogames, which have been formulated in the context of the <e-Aula> platform. Videogames of this genre integrate a large number of contents, which are authored by game designers without strong programming skills. Therefore, designers must collaborate with developers in order to produce the final videogames. <e-Game> takes full advantage of the document-oriented paradigm to control this collaboration.

The development of a game with <e-Game> is driven by the game's storyboard, which is authored by the game designers. This storyboard is structured in terms of a DSML, the <e-Game> language, which includes domain concepts like *scenarios* and *cutscenes*, *items* and *characters*, and *conversations*. The language also allows designers to manage a very high-level view of the game state in terms of *flags* and *conditions* formulated on flags[4]. Videogames are actually produced from <e-Game> documents by using an <e-Game> engine, a highly extensible and configurable processor which can be tailored by the developers for each particular videogame or for each specific evolution of the <e-Game> language.

4. SYSTEMATIZING THE DOCUMENT ORIENTED PARADIGM

In this section we describe our systematization of the document-oriented paradigm. This systematization is the result of the experiences described in the previous section. As aforementioned, the emphasis of this systematization is put on managing the evolutionary nature of the markup languages and their associated processors, as well as the incremental development of particular applications. In subsection 4.1 we describe ADDS, our approach to the document-oriented paradigm. The incremental formulation of DSMLs is tackled in subsection 4.2, while the incremental construction of their processors is described in subsection 4.3.

4.1. The ADDS approach

ADDS is an approach that systematizes application construction according to the document-oriented paradigm. ADDS has been formulated and refined for several years [25][39][40][41][42][43]. In [39][40] the approach was called DTC: *structured Documents, document Transformations and software Components* (see also [44] for an updated revision of DTC). It is important to notice that ADDS is not a development environment, but it is better thought of as a *process model* in the sense used in Software Engineering [45] (see [1] for the role of ADDS as a process model in the context of software comprehension). While in implementing ADDS we can take advantage of modern content management infrastructure (e.g. content management systems [46], XML-native databases [47], etc), the approach does not compromise itself with any of these solutions. Since it is a systematization of a document-oriented paradigm, the nature of ADDS is eminently metalinguistic. ADDS encourages the explicit formulation and operationalization of DSMLs and also recognizes the iterative nature of the life cycle of content-intensive applications (development, maintenance and evolution). These iterations are usually characterized by the creation and/or modification of marked documents, and by the automatic production of running applications from these documents during the production loop. Occasionally some markup needs not addressed by the current DSML may be discovered, leading to an evolution of this language. Thus, an *evolution* loop is introduced in the development of applications for governing these evolutions. The main aspects of ADDS are summarized in Figure 5.

The products and activities in ADDS are depicted in Figure 5a. According to the document-oriented paradigm, four main types of products are contemplated:

- The *application documents* describing the different application aspects.
- The *application DSML* used to mark up these documents.
- The *processor* used to produce running applications from the application documents.
- The *applications* themselves.

ADDS incorporates activities to produce each one of these products (Figure 5a):

[4] This conception of state is also very valuable in order to asses the progress of the players (learners) towards the pedagogical goals imposed by the games.

- The *DSML Provision* activity, to provide the application DSML, which will usually be described using a suitable declarative grammar-based formalism.
- The *Operationalization* activity, whose goal is the construction of the processor for the markup language.
- The *Application Documentation* activity that yields the application documents.
- The *Application Production* activity, where the running application is produced.

In addition, notice that a *repository of DSMLs* is also considered as input in the *DSML Provision* activity. This repository will store previously defined languages and will be used to facilitate this activity. Sometimes the language required will already be available in the repository and the provision activity will be reduced to a suitable search in such a repository. In other cases the language will be obtained by performing some adaptations on an existing one. Even for those cases in which a new language must be defined, this definition can be carried out by combining simpler ones, thus decreasing the cost of this activity in the mid-term. This combination will be facilitated by the use of declarative grammatical formalisms to describe the markup languages. Figure 5b exemplifies all these aspects with the subway case study.

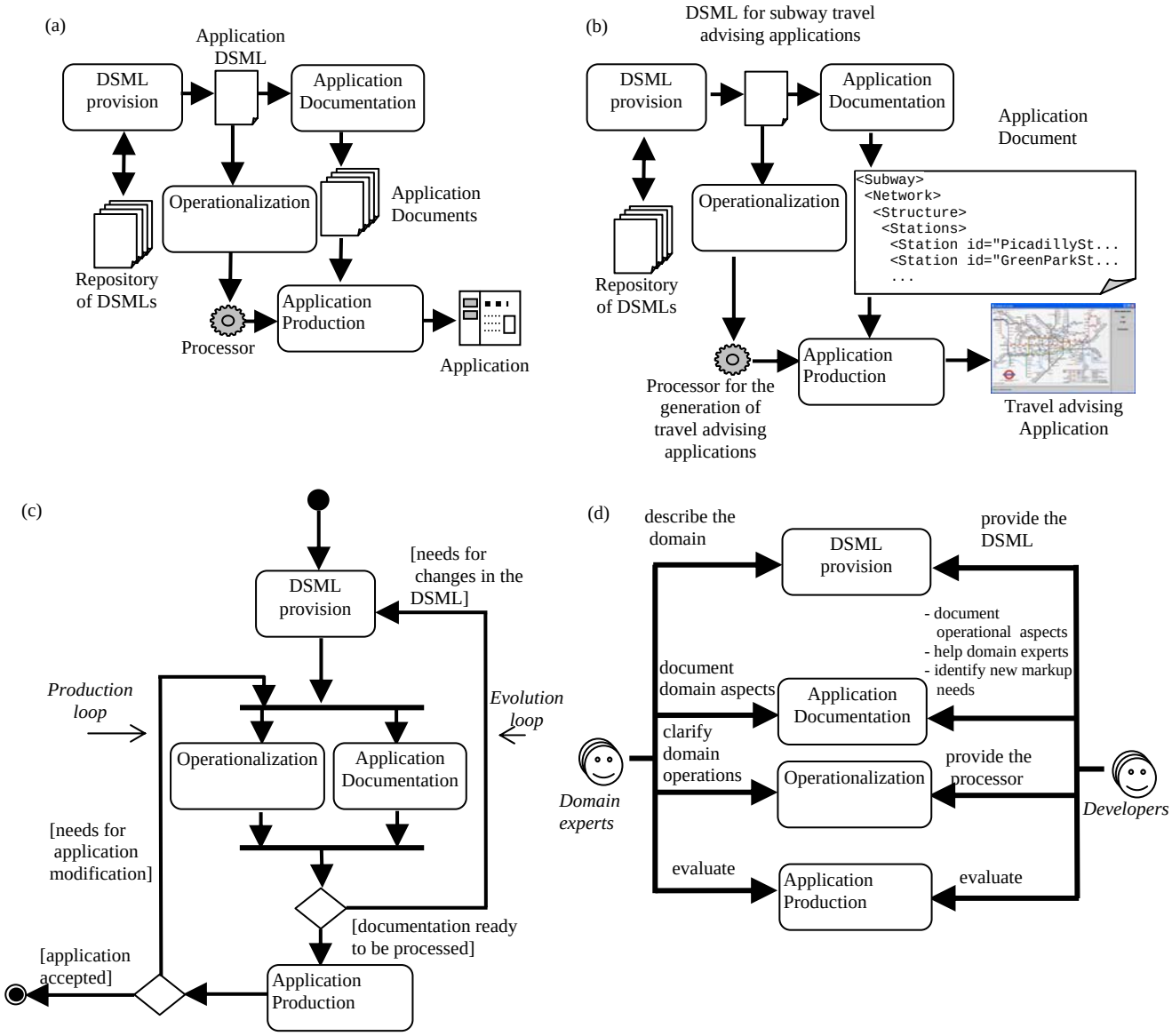

FIGURE 5. (a) Products and activities in ADDS; (b) Exemplification of (a) with the subway case study; (c) Sequencing of the activities; (d) Responsibilities of domain experts and developers.

Notice that activities in ADDS differ to some extent from the activities usually identified in the development of software based on domain-specific languages: *language design* and *language implementation* [15]. Indeed, language design is partially carried out during the *DSML Provision* activity, where the descriptive aspects of the markup language, but not the operational ones, are tackled. This independence between structural characterization (i.e. syntax and contextual/type constraints) and operational meaning is in the primary spirit of descriptive markup [7][8]. Operational aspects are subsequently addressed during the *Operationalization* activity, together with implementation issues. Indeed, the same DSML could have multiple processors associated because the same DSML can be reused in different domains with different operational meanings. This

is consistent with [48] where having a single declarative meaning and a (usually open) multiplicity of operational meanings is identified as one of the main features of descriptive markup languages. Besides, as mentioned above, the *DSML Provision* activity does not necessarily imply the definition of a new language, since a suitable language may already be available, or it can be readily provided by performing simple adaptations on an existing one. Nevertheless, the language selected and/or adapted can subsequently take on new unanticipated operational meanings during operationalization in the context of the new application domains addressed.

The sequencing of the ADDS activities is represented in Figure 5c, where the *production* and the *evolution* loops are also highlighted:

- The production loop usually involves the *Documentation* activity, followed by *Application Production*. This leads to an iterative document-oriented development of the application. For instance, in the subway example, a preliminary subset of the subway network can be initially documented in order to provide a first working prototype of the application. Next, this documentation can be completed to deal with the overall network, and, then, in a third iteration the user interface details can be fine-tuned. New maintenance iterations may arise during application exploitation when the network changes (for instance, due to the addition of a new station or a new line). Nevertheless, sometimes the *Operationalization* activity can also be performed during production to correct any bugs in the processor, although these cases will typically be less frequent than changes in the documents.
- On the other hand, evolution loops arise when new markup needs are discovered during the *Documentation* activity. Such needs may be due to a refinement in the structure of an application document, or, at later stages of the application's life cycle, to the incorporation of new aspects into these documents to address new requirements. Consequently, the usual production loop is abandoned, and the *DSML provision* activity is performed again. Therefore, the DSML is extended in order to contemplate the new markup needs (i.e. the DSML *evolves*). In the subway example, the DSML can evolve to include new structural elements in the networks (e.g. corridors) together with their associated dynamics. Another example of evolution is the inclusion of different user interface styles (e.g. evolution from a simple console-based user interface to a graphic one). The evolution of the DSMLs must be mirrored at the operational level by the evolution of the corresponding processor. Thus, the *Operationalization* activity must incorporate mechanisms to manage this evolution.

Finally, the responsibilities of domain experts and developers in the different ADDS activities are outlined in Figure 5d. Notice that both kinds of actors participate actively not only in *Documentation*, where they carry out a collaborative authoring of the application documents, but also in the other three activities. The participation of domain experts in *DSML Provision* is indeed crucial, because they can advise developers about the right markup vocabularies and structures to use in the domain, letting developers formalize the DSML using a suitable grammatical formalism. Their participation in *Operationalization* is equally critical, because they can clarify the domain aspects of the operations that will finally be included in the processors.

The effective use of ADDS supposes the definition of the different activities and products in terms of specific protocols, procedures and technologies. We have experimented with concrete implementations of the *DSML Provision* and *Operationalization* activities, which are described in the following subsections.

4.2. Incremental Provision of DSMLs in ADDS

The feasibility of the document-oriented paradigm depends to a great extent on appropriate mechanisms for the incremental provision of markup languages, as argued in the previous sections. Although the incremental provision of the languages could be addressed with standard techniques for the definition of their syntaxes and of their contextual constraints [49], this activity can be greatly facilitated by the adoption of suitable modularization constructs. For this purpose, we have developed PADDS (*DSML Provision in ADDS*), a technique for enabling the incremental definition and adaptation of DSMLs in ADDS in a modular way. This subsection surveys this solution, which was initially proposed in [25][43] and which is detailed in [50].

The main construct in PADDS is the *linguistic module*. Intuitively, a linguistic module gives a grammatical characterization of a part of the final DSML. Hence, markup languages in ADDS can be defined by formulating more complex linguistic modules from the combination and extension of simpler ones. A linguistic module can define a set of element types together with their associated attribute lists. In addition, the technique introduces three mechanisms that facilitate the combination of these modules:

- On one hand, the content models in a module can refer to definitions in other modules. In purely grammatical terms, this is equivalent to the combination of pre-existing grammars by adding new productions.
- On the other hand, a module can include a set of *parameters*. These parameters play a role similar to the parameter entities in the SGML or XML DTDs [8][9] and they can be specialized with element types in the owner module or in other modules. Hence they can be grammatically interpreted as *undefined* non-terminals, leading to grammars with *holes* that can subsequently be filled.
- Finally, a module can add new attribute lists to element types defined in other modules.

Once the linguistic modules for a DSML are available, the language itself must be described with a *document grammar*. This grammar is obtained by following a *grammar production specification*. The aim of this specification is to resolve conflicts between linguistic modules (e.g. name conflicts) and to adapt the concrete markup vocabulary to different contexts (e.g. different languages). Therefore, by providing alternative production specifications, it is possible to obtain different profiles of the same DSML. As we have realized during our experiences, the adaptation of the markup vocabulary to a particular community of experts greatly contributes to increasing the usability and acceptance of the markup language in such

a community, since this adaptation helps to capture the particular socio-cultural circumstances of the experts. It should be remembered that one of the primary aims of descriptive markup is to be readable by both people and machines. Indeed, the importance of tag names in a descriptive markup language to minimize the cognitive demands during the markup process has been largely recognized in the community of descriptive markup [7]. For instance, an English expert will feel more comfortable using a language with tags in English rather than one with tags in Chinese, regardless of the fact of that she is dealing with isomorphic markup languages. This is especially true when minimization conventions are chosen for tag names (e.g. an English speaker expert could find natural the abbreviation `sn` for *subway network*, while a Spanish speaker could find `rm` more natural, since *subway network* is *red de metro* in Spanish).

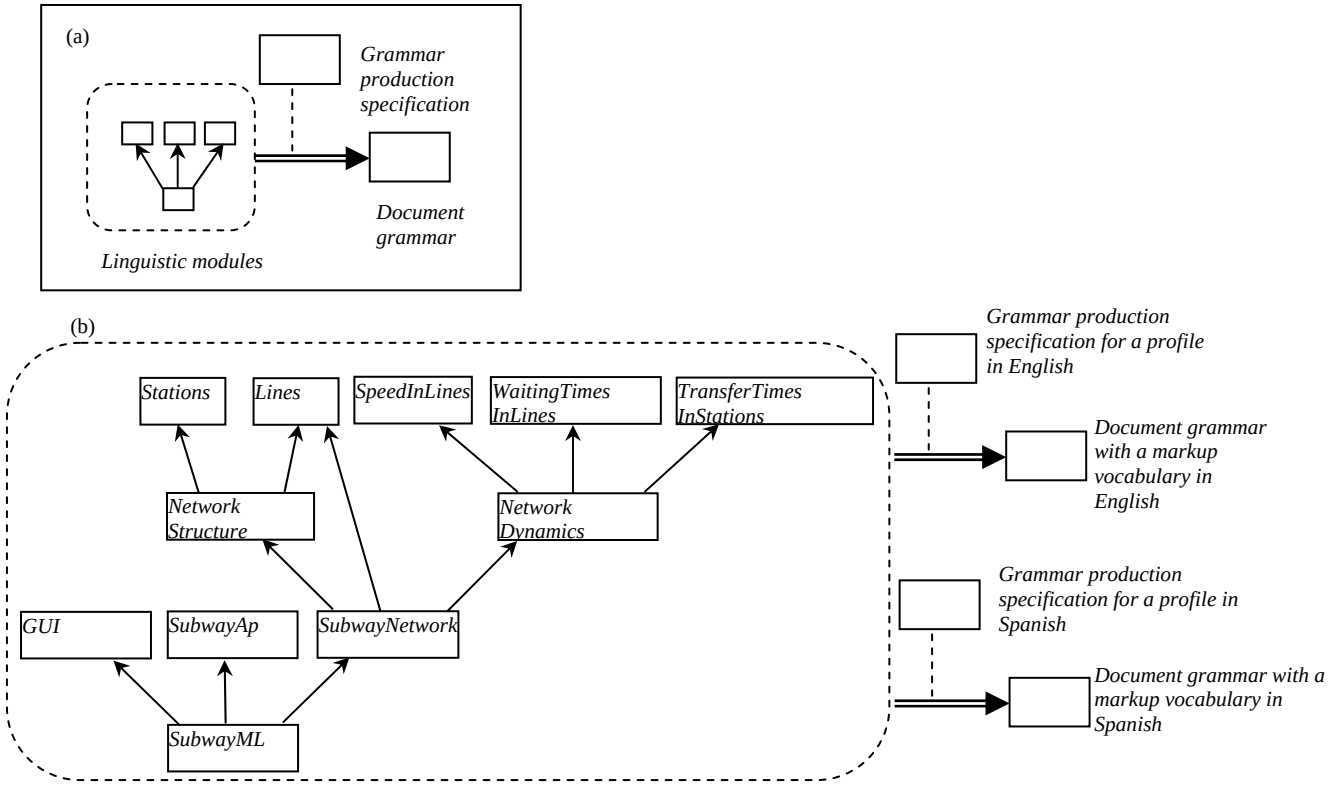

FIGURE 6. (a) Structure of a DSML description according to PADDS; (b) Structure of the description of the DSML for the subway case study. When a module (e.g. *SubwayNetwork*) refers to and/or extends definitions in another one (e.g. *NetworkStructure* or *Lines*), an arrow starting in the referent and ending in the referred is used.

(a)

```
Module: Lines
Grammar:
<!ELEMENT Lines (Header?,LinesTitle?,
                 LinksTitle?,(Line)+)>
<!ELEMENT Header (#PCDATA)>
<!ELEMENT LinesTitle (#PCDATA)>
<!ELEMENT LinksTitle (#PCDATA)>
<!ELEMENT Line (Name,Link+)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Link EMPTY>
<!ATTLIST Link origin IDREF  #REQUIRED
               destination IDREF #REQUIRED
               length NMTOKEN #REQUIRED>
```

(b)

```
Module: SubwayNetwork
Grammar:
<!ELEMENT Network (Header?,
                   NetworkStructure.Structure,
                   NetworkDynamics.Dynamics)>
<!ELEMENT Header(#PCDATA)>
<!ATTLIST Lines.Line id ID #REQUIRED>
```

(c)

```
...
Module: Lines
  Map: Lines  → Líneas
       Header → Encabezado
       LinesTitle → TítuloLíneas
       LinksTitle → TítuloTramos
...
```

FIGURE 7. (a) Definition of the *Lines* linguistic module; (b) Definition of the *SubwayNetwork* linguistic module. In order to refer to definitions in other modules, a *dot* notation is used, like in `NetworkStructure.Structure`; (c) Part of a grammar production specification.

DSML provision in terms of PADDS is depicted in Figure 6. Figure 6a depicts the structure of a DSML description according to PADDS. Figure 6b exemplifies this in the subway case study. There the description includes linguistic modules

for marking up the subway networks, as well as modules for marking up the relevant aspects of the user interface. All these modules are combined to produce a suitable grammar. This combination is carried out following an appropriate production specification. It is actually possible to give several of these specifications to get alternative grammars. As a simple but useful example, Figure 6b suggests a production specification that sets up the names of the tags in English, and another one to do the same in Spanish.

PADDS is a conceptualization and is independent of any particular description formalisms. Therefore, it could be implemented using any suitable grammatical formalism [10], although our experiences with the technique have been completely based on the XML markup metalanguage[5] [9] and we have used XML DTDs for describing both the grammatical aspects of the linguistic modules and the final document grammars. Production specifications are in turn conceived as sets of renaming rules for the markup vocabularies of the linguistic modules, thus allowing for the resolution of the different name conflicts between the modules' DTDs. Because name conflicts are solved at the grammatical level, the use of namespaces [51] is not necessary in this implementation. In our opinion, this facilitates the *Documentation* activity for domain experts because it is done in simple terms, which is one of the main objectives of ADDS. Figure 7a shows the definition of the *Lines* linguistic module that governs the markup of the lines of a subway network. Figure 7b shows in turn the definition of the *SubwayNetwork* linguistic module. Notice that this module refers to the element type `Structure` (defined in the *NetworkStructure* module) and `Dynamics` (defined in the *NetworkDynamics* module). More important, notice that the module extends the definition of the element type `Line` (defined in the *Lines* module) with a new `id` attribute. This is the reason for the arrow from *SubwayNetwork* to *Lines* in Figure 6, which at a first glance may seem puzzling. The extension is performed to glue together the structural and the dynamical aspects of the subway network, since markup of the dynamic aspects must refer to the lines documented in the network structure. Finally, Figure 7c depicts part of a grammar production specification for the DSML in the subway example. According to this specification, the markup vocabulary of the final grammar will be in Spanish.

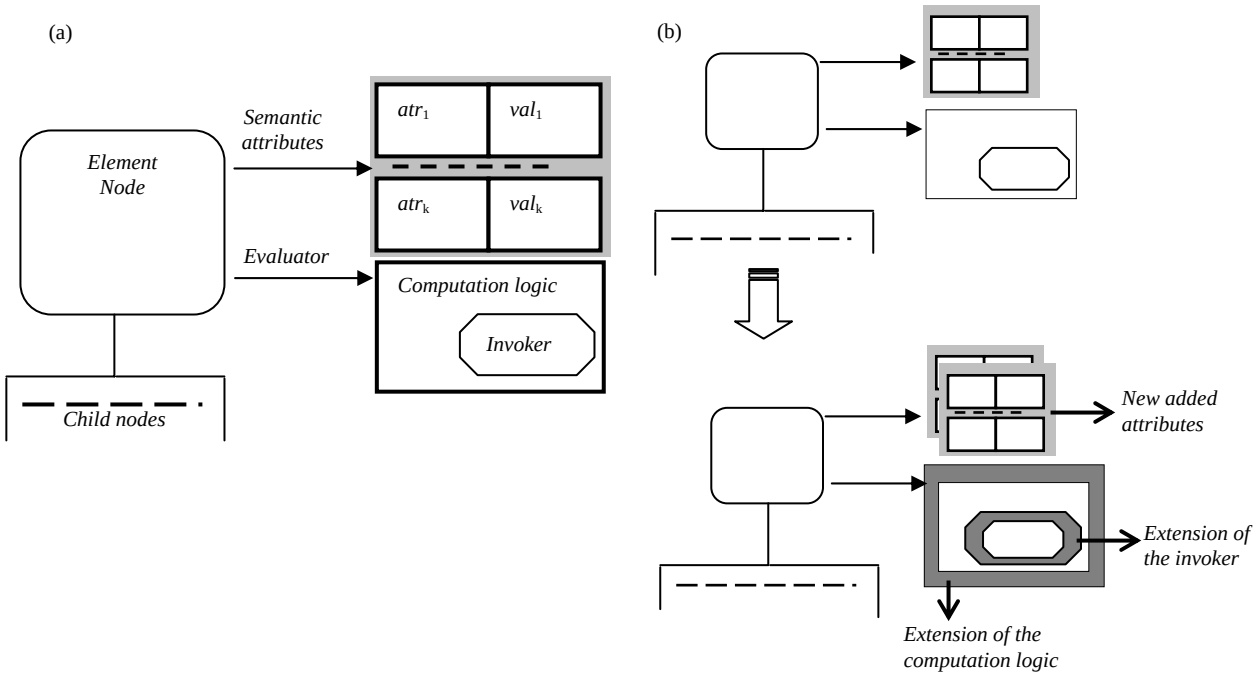

FIGURE 8. (a) Element nodes are operationalized with *semantics attributes* and with an *evaluator*; (b) the attributes to be added to an element node can be incrementally determined and the associated evaluator and its insider invoker can be incrementally extended as well.

4.3. Incremental Operationalization of DSMLs in ADDS

The incremental provision of the DSMLs must be accompanied by their incremental operationalization (i.e. the application of incremental techniques to the construction of their processors). As with the incremental provision of the language's syntax and contextual constraints, the adoption of suitable modularization techniques can facilitate incremental operationalization with respect to the use of more conventional language processor development techniques. Nevertheless, while the *DSML provision* mainly involves syntax, and thus its modular implementation is facilitated by the adoption of markup standards, the incremental operationalization deals with semantics, which is substantially more complex to modularize. This is usually identified as the *semantics modularity problem* in the literature, and has to do with the construction of language processors from reusable components in such a way as to facilitate the evolution of these processors when their associated languages

[5] It is important to point out here that XML *is* not used to describe the application, but it will be used to structure the documents describing it.

change [52][53][54]. For this purpose, we have formulated OADDS (*Operationalization in ADDS*) as a conceptual model for the incremental construction of processors in ADDS that fits in with the peculiarities of the PADDS technique. The origin of OADDS is in [39][40]. In [55] a first attempt to introduce semantic modularity mechanisms in the model is given. In [25][41][42][43] earlier formulations of OADDS are presented. The model in its current form is detailed in [50]. In this subsection we overview the model. For the sake of simplicity the more difficult technical details are omitted but can be found in [50].

OADDS is based on the well-known techniques of syntax-directed translation [49], widely used in the compiler construction domain, although the model also takes advantage of the descriptive nature of markup languages to promote their incremental operationalization. The model regulates the incremental development of processors that are used to generate applications from their describing documents. The generation process is usually performed by:

- Parsing the documents into their corresponding document trees.
- Operationalizing the element nodes of the resulting trees with a set of *semantic attributes* and an *evaluator* (Figure 8a). The semantics attributes are used to contain the different sub-products involved in the generation process, while the evaluator is used to compute the values of such attributes.
- Setting the values for the semantics attributes associated with the roots of the documents trees and activating the evaluators assigned to those roots.

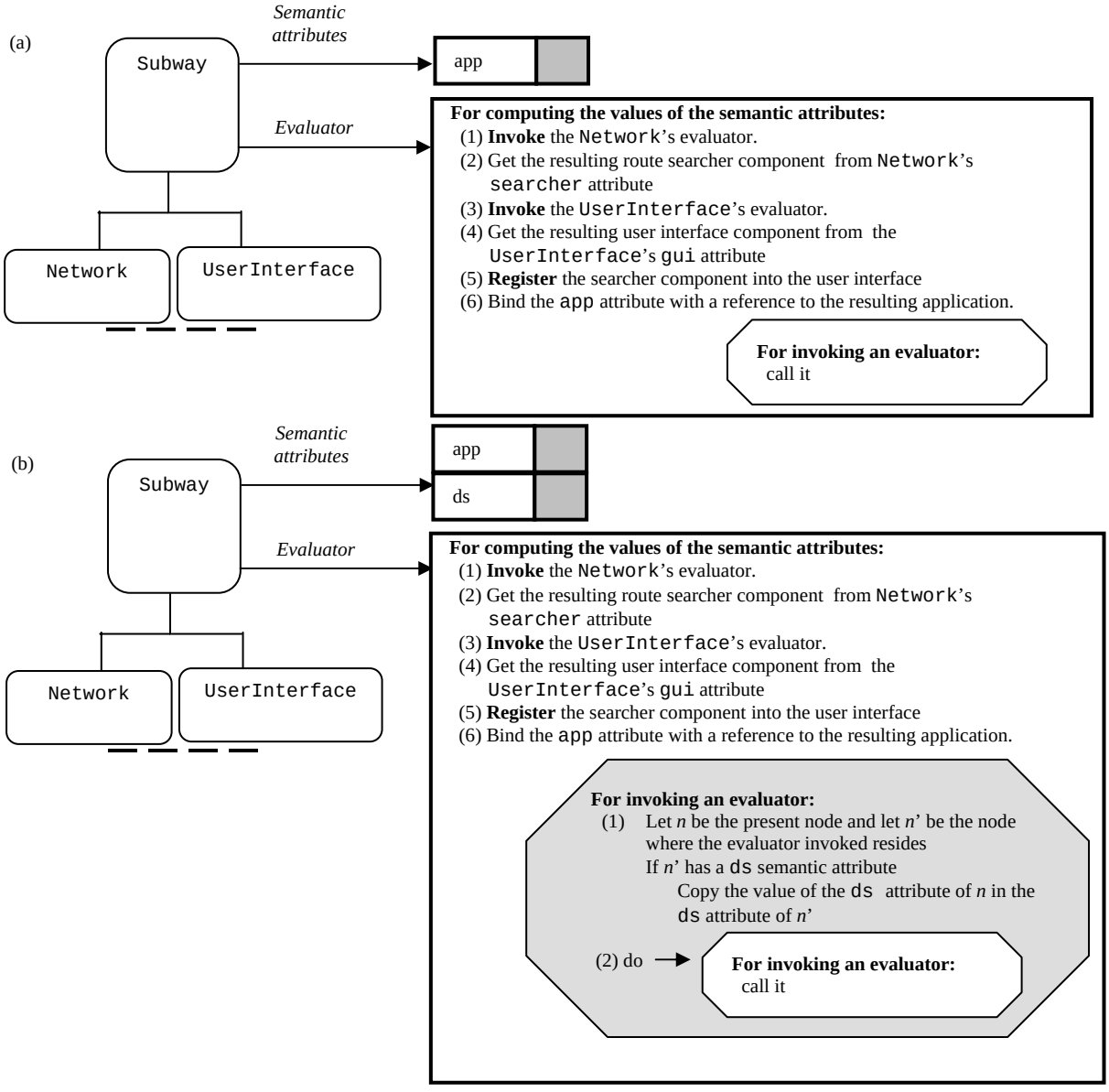

FIGURE 9. (a) Initial operationalization for element nodes of `Subway` type; (b) Extension of the operationalization in (a) for dealing with lists of disabled stations. The logic added to the evaluator is shaded.

Computations of the attribute values are entirely governed by evaluators, which encapsulate how to calculate those values in an application-specific way. In performing such computations, an evaluator belonging to a node can invoke other evaluators in the neighborhood. For doing so, the evaluator includes a pluggable *invoker* component, which encapsulates how

the invocation must actually be carried out. Therefore, evaluators never call other evaluators directly, but they do use their associated invokers. This organization promotes the incremental construction of the processors, since it eases the addition, valuation and propagation of new semantics attributes in the document trees while preserving the already available evaluation machinery (Figure 8b). Indeed, with each new processor's increment it is possible to specify:

- New semantic attributes to be added to the element nodes.
- How to wrap the associated evaluators to deal with the computation of the new added attributes.
- How to wrap the associated invokers to deal with the propagation of these new attributes among the nodes in the document tree. As we have realized in our experimental developments, in many relevant cases an invoker can be extended and changed without affecting the rest of the evaluator.

To illustrate this mechanism let us consider the operationalization of the elements of `Subway` type, which is used for marking up the documentation of the subway applications in our case study (Figure 9a):

- The `app` semantic attribute is used to refer the generated application.
- The evaluator encodes the top-level strategy to generate the application: getting a *route searcher* component from the network's description, a *user interface* component from the corresponding description, and the application itself by combining the two. In getting the two subcomponents the evaluator invokes those associated with the `Network` and the `UserInterface` children. This invocation is done using an invoker that simply calls the evaluators without performing any additional work.

Let us now consider how this operationalization can be extended to allow the disabling of stations in the network (e.g. to contemplate *out of service* stations). For this purpose, a list of disabled stations must be propagated to the tree associated with the description of the network, which will affect the generation of the route searcher and the user interface where these stations will be marked as *out of service*. The resulting operationalization is depicted in Figure 9b. According to this, to propagate this list across the `Subway` element node:

- A new semantic attribute, `ds`, is added to make reference to the aforementioned list.
- The evaluator's invoker is extended to manage the propagation**.** So when the `Subway`'s evaluator invokes, for instance, the `Network`'s evaluator, before executing the older invoker the added extension copies the value of `ds` in the cited child node, as also happens when the `UserInterface`'s evaluator is invoked.

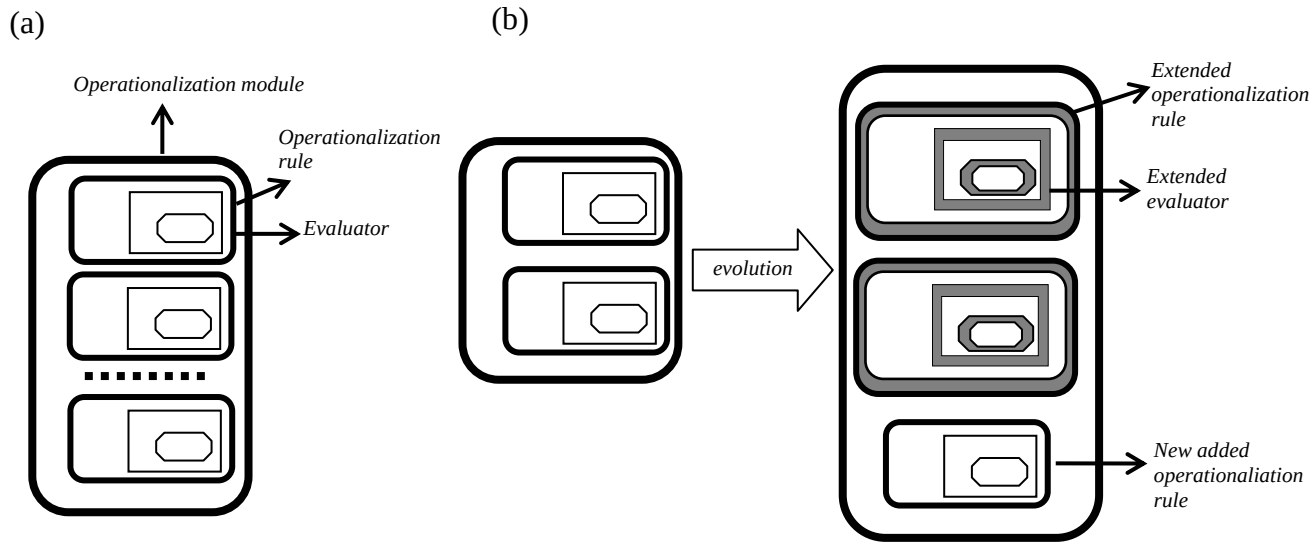

FIGURE 10. (a) Structure of an operationalization module; (b) the evolution of the operationalization modules are managed by extending the operationalization rules and the insider evaluators.

These simple techniques help to manage the evolution of processors when the supporting DSMLs evolve. With each linguistic module a set of *operationalization rules* is provided to decorate element nodes with semantic attributes and evaluators. In OADDS such a set of rules constitutes a so-called *operationalization module* (Figure 10a). When the linguistic module is extended, new rules dealing with the new types of elements are added to the operationalization module. Since the extension can affect the existing operationalization machinery (e.g. due to the introduction of new attributes that must be propagated across the document trees), pre-existing sets of rules might be also extended. This extension usually supposes planning for the introduction of new semantic attributes and extending the evaluator used. As previously mentioned, this last extension is performed either by wrapping the old evaluator or by extending its invoker (Figure 10b). Also notice that the extension mechanisms in OADDS resemble the conventional mechanisms of class-based inheritance in object-oriented systems [56]. Indeed, the extension of a module with new operationalization rules is analogous to subclassing an existing class by adding new methods. On the other hand, the extension of an existing evaluator (by wrapping it or by extending its invoker) is similar to overriding a method in a superclass. Nevertheless, while conventional class-based object-oriented

inheritance is a static relationship (i.e. the base classes for subclasses form part of the definition of such subclasses), in OADDS it is not necessarily true. In OADDS it is possible to devise extensions for evaluators and invokers that are kept independent of the components extended. For instance, the invoker extension applied in Figure 9b could also be applied whenever the list of disabled stations must be propagated from a parent node to its children. This situation is analogous to getting classes that are parametric in their superclasses. In the object-oriented domain, this kind of construct is usually called *mixins*. Mixins have been recognized as valuable mechanisms for generalizing conventional class-based inheritance, as well as other types, [57] and they have also been used to promote modularity in the design of language processors [53].

As a conceptual model, OADDS is independent of any specific formal semantic framework. Although we have not made any special attempt to formalize the model, in our opinion, works like [58] on modular structural operational semantics and the classical work of Knuth on attribute grammars [59] (as well as the proposals derived from this [60]) could be good starting points in this direction. This formalization could be useful in exposing further features of the operationalization process and in designing domain-specific notations for describing OADDS processors. Also notice that the combination of such a formalization of OADDS and an appropriate formalization of PADDS could yield a systematic method for the modular design of the static and operational aspects of DSMLs. Furthermore, the separation of PADDS and OADDS preserves the separation between language formulation and operationalization, as promoted by descriptive markup. Likewise, the conceptual nature of the model does not fix any particular implementation technology. Nevertheless, the model can be easily implemented as an object-oriented framework. The details of this OADDS implementation can be found in [25].

5. EVALUATING THE DOCUMENT-ORIENTED PARADIGM

In this section we outline a first evaluation of the current state of the document-oriented paradigm and of its systematization. This evaluation, although qualitative, has been contrasted with the experiences described in section 3. The main strengths of the approach are summarized in subsection 5.1. On the other hand, the main weaknesses detected are presented in subsection 5.2.

5.1. Strengths

During our experiences with the document-oriented paradigm we have realized how it facilitates the evolution, maintenance and tuning up of content-intensive applications, because they are described in the form of readable and editable documents, understandable by both domain experts and developers. The approach also improves information reuse because well-known markup standards (SGML/XML) can be used in the production of the documents. This improvement is especially relevant for content-intensive applications (e.g. e-learning applications), where the representation of the contents must be portable and where the use of standard formats is critical in order to extend their life cycles. Lastly, the paradigm also improves software reusability, because once a suitable processor of a DSML for an application domain is available, this can be used on the documents of all the applications in this domain.

As indicated in section 3, we have realized that the need to manage the evolution of the applications implies the need for managing the evolution of the markup languages and of their associated processors. The main contribution of ADDS with respect to other content and language driven development approaches is that in ADDS, languages are incrementally defined during the production and maintenance processes, which makes the inherent complexity of the approach more affordable. In ADDS it is possible to start the development of an application with a small prototype based on a very simple markup language that only reveals the structure and the data needed for this first prototype. In successive iterations the language can be extended to cover all the requirements of the application. This evolutionary nature of the DSMLs and their processors provides the flexibility required by the development, maintenance and evolution of complex applications, like those described in section 3. The language is extended when new markup needs are discovered. This also facilitates the use of DSMLs, because it avoids the inclusion of very general or sophisticated descriptive artifacts.

Another important feature of ADDS is the clear separation between the descriptive and the operational concerns. This separation eases a rational distribution of roles between the different stakeholders during application maintenance. Domain experts can concentrate their efforts on maintaining the documentation (including the operational aspects), while developers can focus their work on maintaining and improving the processor. We have observed these benefits even without the consecution of evolutionary iterations.

Although simple, PADDS, the provision technique for languages, has shown to be very useful during the incremental formulation of markup languages. The concept of *linguistic module* has proved to be quite valuable for managing this incremental formulation. Also, it is simple enough to be easily assimilated by the average developer and understood by domain experts. In addition, the use of DTDs as descriptive formalism has proved very valuable, despite its limited expressive power [61]. While XML DTDs are simpler than other schema languages [10], we have found several advantages in their use. On one hand, they are an integral part of the XML standard, and on the other hand, and more importantly, according to our experience working with domain experts, DTDs are simple-to-use and more understandable mechanisms for these experts.

Although a bit more complex than PADDS, OADDS, the model for the incremental construction of processors, has also been very useful in dealing with the evolution of processors for markup languages. The concept of *evaluator* has been well accepted among developers because it resembles the concept of *template rule* used in transformation languages like XSLT

[62]. In addition, the ease in giving object-oriented implementations provided by OADDS promotes its integration with widely used object-oriented frameworks for document processing [63].

5.2. Drawbacks

The main drawback in our proposal, which is actually inherited from the DSL approach [15], is in the cost and complexity of setting up the entire production environment for each application domain. Applying ADDS supposes formulating a DSML and building its processor. While these tasks can be easy for simple resource-like configuration languages, for more complex scenarios it could demand heavy domain-analysis and domain-design / implementation activities [12]. This is why in ADDS we have put our emphasis on the incremental definition and operationalization of DSMLs.

Another important drawback of our approach is the lack of clear guidelines in order to identify when an application domain is suitable or unsuitable to be faced with a document-oriented paradigm. In our opinion, these guidelines must arise on the basis of experience. We have always considered it necessary to discover types of content-intensive applications that will pay for the effort, i.e., where formulating a specific markup language and building its processor is better than relying on a conventional configuration schema (e.g. a resource file or some tables in a relational database). This led us to the efforts described in section 3, where interaction between domain experts and developers was especially critical for the final success (including the whole life cycle) of the application.

While we have also identified the authoring problem as a key point to the successful application of the document-oriented paradigm, this problem has not yet been totally addressed by our approach. Indeed, while our efforts have been focused on producing applications from the documents describing them, the approach barely contemplates how these documents must be produced by domain-experts. In its current state experts need some working knowledge of XML technologies (e.g. editing XML documents) to be productive. A solution to this drawback could imply the incorporation of ways to produce not only processors, but domain-specific editors for applications documents as well.

PADDS is one of the possible techniques for the incremental provision of DSMLs. This technique can be actually recognized as a pragmatic systematization of the *architectural forms* proposals in the descriptive markup scenarios [64]. While the technique simplifies some of the obscure mechanisms for encoding the architectural mappings in those techniques, it also leaves out the possibility of an arbitrary number of generalization / specialization levels in the markup vocabulary. Our technique only allows for two such levels: the linguistic module level, and the level of the produced grammars. In addition, although the smart use of parameters can enable the refinement of the element's content models, the technique also misses empowering the architectural mechanisms for performing arbitrary refinements of such structures (i.e. to interleave additional markup structures in the existing ones when required).

OADDS is also only one of the possible models for carrying out incremental construction of processors in ADDS. While this model has been valuable for managing incremental operationalization, in our experiences with the model we have also identified several shortcomings. On one hand, the model does not contemplate any particular mechanisms to carry out the static analysis of the built processors (e.g. static typing of the built processors, circularity checking of attribute dependencies, etc). Sometimes this can make the development of OADDS processors difficult, because some tricky defects are only discovered during application production time. Also OADDS lacks a clear model for the applications built. In this way, OADDS is comparable to a transformation language like XSLT, which specifies the transformation mechanism, but does not specify the nature of the target document types**.** In our opinion this is not a disadvantage but a feature, because it allows many application models to be accommodated although we have not explicitly defined any of these models, outside of our earlier works on component-oriented DTC applications [44]. Therefore, in our experiments we have adopted *ad hoc* mechanisms to manage the evolution of the generated applications when the processor generating them evolves. Finally, OADDS does not consider any specific mechanisms for facilitating the operationalization of document trees when they are modified during run time. These mechanisms will be mandatory when facing the construction of domain-specific editors with our approach.

6. RELATED WORK

HyTime [65], a SGML-based language for the description of hypermedia applications, demonstrated that in some domains descriptive markup languages can be used for describing applications in terms of documents and that the applications described could be generated by processing these documents. XML and its related technologies have generalized the use of descriptive markup languages as a standard way for information interchange between applications and for many other uses. While in almost all these approaches markup languages are pre-established, we have been compelled to adopt a more dynamic approach, where markup languages evolve according to the markup needs of domain experts and developers.

The seminal work of Knuth on *literate programming* [66] (see also [67]) realizes the benefits of identifying the programs and their documentation. In literate programming, a hypertextual representation of the program code is promoted, which is interleaved with its documentation. The result (a *web*) is a narration of the program, in the same way that the program would be presented in a programming textbook. These documents are marked up for enabling both the assembling of working programs (*tangling*) and the production of documentation printouts (*webbing*). In our approach we also use documents to describe applications. Nevertheless, we only document and markup the high level aspects of the main features of the application, but not the code of the programs implementing these applications. Because of this, in our work, suitable markup

languages are provided *for each* application domain instead of using a fixed one like in literate programming. Nevertheless, notice that literate programming could be used to document the processors themselves in order to enhance their production and maintenance by developers. In [25][42] we propose an initiative in this direction, where processors are documented following an *elucidative* style [68].

The document-oriented paradigm resembles the work in content-management systems, either in broad [46] or in specialized domains (e.g. learning content management systems [69]). In these systems contents are usually structured in terms of descriptive markup languages to enable their processing. Nevertheless, content-management systems are usually built with a specific use in mind (usually the publication of contents in different media). The work in the present paper is not necessarily constrained to content publishing, but also takes in many other application domains. Nevertheless, as previously discussed, it is possible to take advantage of the infrastructure provided by an existing content-management system to leverage the implementation of the document-oriented approach.

The document-oriented paradigm also resembles some tendencies in software reuse. As mentioned in section 2, work in application families [70] is similar to the document-oriented paradigm in the sense that a single DSML can be understood as an explicit characterization of a family of related content-intensive applications. Nevertheless, in our opinion this similarity is collateral. The distinctive features of a content-intensive application let us see its successive releases as an application family, but the stress of the document-oriented paradigm is on producing and maintaining a single content-intensive application, instead of a whole family of related applications. This stress is also mirrored in our incremental, just-in-time approach for the formulation and operationalization of markup languages. As mentioned in section 2, the artifacts built in producing and maintaining an application can also be used in the development of other similar applications, but this family-oriented use of the paradigm is also collateral. In the same way, it is possible to see similarities between our approach and work in software product lines [71]. Indeed, as a collateral consequence of the modular nature of markup languages and processors in our approach, the different markup languages and the processors for each member of a line of content-intensive applications could be supported by a common repository of linguistic and operationalization modules.

Our work shares many features with the language-driven approaches to software development [72][73], which systematize the use of domain-specific languages in the development of software. A typical language-driven approach starts formulating a domain model, which supports the DSL. The DSL itself is firstly characterized in terms of its abstract syntax, which serves as the basis for its operational semantics, as well as for different alternative concrete syntaxes (e.g. visual and textual ones). Our proposal can be thought of as a particular case of a language driven approach with a narrower purpose: to identify suitable document types for describing applications and to devise appropriate descriptive markup languages for describing the structure of these documents.

There are several initiatives in the application of descriptive markup technologies for supporting domain-specific languages. A pioneering work in using SGML/XML for the definition of DSLs is [48]. During [74] the relationships between markup languages and the DSL approach were highlighted. In [75] a comprehensive work about using XML technologies to implement the DSL approach is detailed. Although these works recognize the potential of markup metalanguages as a vehicle for defining DSLs, the stress is put on their use to characterize the formal syntax of a language instead of on their use to make the structure (elements and their lexical attributes) of different types of documents explicit. In our opinion descriptive markup languages pay off with respect to more conventional computer (programming) languages when they are designed with the second use in mind. According to this, applications are described in documents using narrations in natural language and other expressive media (e.g. images, audio, video, etc.). Suitable descriptive markup languages are then used to describe the structure of these documents which is relevant for generating the application (but not for describing the application itself, as with the first use). With this we get a midway solution between the idyllic one of automatically generating the application from its raw documentation in natural language and the more conventional one of producing it from its description in a (maybe domain-specific) programming language. By separating markup and application descriptions, documents describing applications can exist without the markup, documents of the same type can be marked up with different DSMLs, and thereby exposing a different meaningful structure [76]. The same markup language can also be used on different types of documents if they have a common abstract structure [64]. In our opinion this flexibility of the descriptive markup language concept provides the expressive freedom required by domain-experts and developers during the production and maintenance of content-intensive applications.

The Jargons approach [77][78] is also similar to our work. In Jargons, DSLs are directly formulated, and even operationalized (using a scripting language) by domain experts. While the conception of this author-driven design of DSLs is consistent with our work, we consider it unrealistic to assign language provision and operationalization responsibilities to domain experts. Instead, for this purpose a community of developers is involved. Moreover, Jargons does not contemplate the semantic modularity problem in the operationalization of DSLs. This problem is critical when these languages evolve.

Modern schema languages [10] provided modularization facilities for the incremental definition of XML-based descriptive markup languages. With PADDS we don't attempt to provide an alternative to these proposals, but we abstract some useful patterns for the incremental definition of these languages. In addition, these patterns can be subsequently implemented by using a specific grammatical formalism. We have chosen a DTD-based implementation because DTDs are easy-to-use by developers and easy-to-understand by domain-experts. This choice does not rule out alternative implementations that use more sophisticated schema languages. In fact, as mentioned in section 5, the model is based on the

mechanisms behind *architectural forms* in HyTime [64][65]. In addition, proposals for modularization of DTDs like that used in the modularization of XHTML [79] have also inspired our technologies.

OADDS resembles the ideas behind attribute grammars [59][60]. In attribute grammars semantic attributes are added to symbols in a context-free grammar. The computations of the values for these attributes are specified with semantic equations attached to the grammar's productions and are expressed in terms of semantic functions. Therefore, an operationalization module in OADDS resembles an attribute grammar, and an operationalization rule resembles the semantics attached to a production where all their equations have been packaged together. Nevertheless, while in attribute grammars there is a strong coupling between syntax (the production rules) and semantics (the equations) this is not necessarily true in OADDS, where this coupling is naturally loose. This decoupling eases the evolution of the processors because the markup structures can be refined without breaking pre-existing operationalization modules. On the other hand, one of the main advantages of attribute grammars with respect to OADDS is that, while in attribute grammars the order in which attributes are computed can be derived from the dependencies given by the semantic equations, in OADDS evaluation control must be made explicit by evaluators. While we are aware of the benefits of this feature of attribute grammars with respect to OADDS (e.g. amenability for static analysis and for the automatic generation of processors from high level specifications), we have realized that evaluators, being more coarsely grained than semantic equations, are less knowledge-demanding for the average developer.

OADDS is inspired by the techniques for the construction of modular language processors. These techniques have been popularized inside the functional programming community, where the main approach is based on *monads* and *monads transformers* [52], although it is also possible to find proposals in the object-oriented paradigm (based on the use of the aforementioned *mixins* [53]), and also in the attribute grammar approach, where modularity is usually achieved by means of some sort of attribution patterns [54]. As pointed out in section 5, one of the main pragmatic limitations of OADDS is to obviate the static typing of the OADDS processors. While static typing of modular processors remains an open problem, approaches to static typing followed in [52][53] could be adapted to the OADDS context. In addition, other types of static checking that are not currently contemplated in OADDS could be incorporated into the model to facilitate the construction of processors (e.g. circularity tests in attribute grammars [60]).

7. CONCLUSIONS AND FUTURE WORK

This paper describes our work on the formulation of a document-oriented paradigm for the development of content-intensive applications. According to this paradigm, these applications can be automatically obtained by processing marked documents describing their main features: their informational contents and operational aspects. We have successfully applied different versions of this paradigm to the development of educational and hypermedia applications, and also to the development of knowledge based systems. As a result we have systematized it, thereby formulating the ADDS approach, the PADDS technique and the OADDS model.

We point out that the basic idea in the document-oriented approach is to use documents to ease communication between domain-experts and developers, and to use markup languages to let developers automatically produce the applications from their describing documents. Therefore, the document-oriented paradigm eases the collaboration between domain-experts and developers during the whole life-cycle of a content-intensive application. In addition, the use of descriptive markup also improves the reuse of the information integrated in the applications both in their future releases and in the development of other related applications. This feature is mandatory in order to avoid the obsolescence of proprietary formats in the production and maintenance of content-intensive applications. Finally, the approach also promotes software reusability, because the same DSML and the same processor can be used to develop many different applications in the domain. With this we are not claiming that the document-oriented approach is the only way to produce and maintain content-intensive applications, but we have accomplished this is an adequate way in domains where contents can be meaningfully arranged as *documents*. For example, while the document-oriented approach may not be specially well-suited for developing an arcade videogame, it could be suitable for specific genres of videogames (e.g. educational adventure videogames, where the development can be driven by a document with the game's storyboard). On the other hand, while the approach is not a substitute for a database-centric architecture in a typical e-commerce application, it is specially suited for developing an educational web site for a virtual museum.

The key point for the practical use of the document-oriented paradigm is to manage the evolution of the markup languages and of their associated processors. Without this evolution, languages become obsolete, because they do not fulfill the new expressive needs of the different participants during the development process. In addition, the evolution contributes to more usable languages because they include exactly the required expressive artifacts (no more, no less). Finally, this evolutionary nature helps to palliate the high costs associated with exhaustive domain-analysis and domain-engineering activities in other approaches to software development based on domain-specific languages. Indeed, with an evolutionary approach, these costs can be amortized throughout all the life-cycles of the applications in the domain. PADDS is the technique we have developed and integrated in ADDS to implement the incremental formulation of domain specific markup languages. The evolution of these languages is mirrored in the OADDS model for the incremental construction of modular processors.

As stated in section 3, currently we have centered our attention on the field of distributed e-learning systems [27][28][35]. Therefore, our current work is oriented towards continuing with the improvement of ADDS' pragmatic applicability by testing it on several additional projects in the domain of distributed e-learning applications. With this work we hope to achieve further refinements and improvements in our formulation of the paradigm. In addition, we are interested in a better

characterization of the authoring problems in ADDS, not only in the *Documentation* activity, but in the others as well. Finally, as future work, we are considering the experimentation with alternatives to PADDS and OADDS based on object-oriented attributed grammars [60].

ACKNOWLEDGEMENTS

The Spanish Committee of Science and Technology (TIN2004-08367-C02-02, TIC2002-04067-C03-02 and TIN2005-08788-C04-01) has supported this work. We also would like to thank Antonio Navarro for his collaboration on earlier stages of this work.

REFERENCES

[1] Sierra, J. L. Fernández-Manjón, B. Fernández-Valmayor, A. and Navarro, A. (2005) Document-Oriented Construction of Content-Intensive Applications. *International Journal of Software Engineering and Knowledge Engineering –Special Issue on Maturing the Practice of Software Artefact Comprehension-*, 15(6), 975-993.

[2] Fernández-Valmayor, A. López Alonso, C. Sèrè, A. and Fernández-Manjón, B. (1999) The Design of a Flexible Hypermedia System. *IFIP WG3.2-WG3.6 Conference*: *Building University Electronic Educational Environments*, University of California Irvine, California, USA, 4-6 August, pp. 51-66. Kluwer, Dordrecht, The Netherlands.

[3] Collis, B. and Strijker, A. (2004) Technology and Human Issues in Reusing Learning Objects. *Journal of Interactive Media in Education*, 4, http://www-jime.open.ac.uk/2004/4/.

[4] Schlusmans, K. Koper, R. and Giesbertz, W. (2003) Work Processes for the Development of Integrated e-learning Courses. In Schlusmans, K. Koper, R. and Giesbertz, W (eds), *Integrated eLearning*. RoutledgeFalmer, London, UK.

[5] Fayad, M.E. and Schmidt, D.C. (1997) Object-Oriented Application Frameworks. *Communications of the ACM*, 40(10), 32-38.

[6] Booch, G. Rumbaugh, J. and Jacobson, I. (1998) *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts,USA.

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

[8] Goldfarb, C. F. (1990) *The SGML Handbook*. Oxford University Press, Oxford, UK.

[9] Bray. T. Paoli, J. Sperberg-McQueen, C.M. and E. Maler (eds). (2000) Extensible Markup Language (XML) 1.0 (Second Edition). *W3C Recommendation*, www.w3.org.

[10] Lee, D. and Chu, W.W. (2000) Comparative Analysis of Six XML Schema Languages. *ACM SIGMOD Record*, 29(3), 76-87.

[11] Kim, L. (2003) *The Official XMLSPY Handbook*. Wiley Publishing, Indianapolis, USA.

[12] Czarnecki, K. and Eisenecker, U. (2000) *Generative Programming: Methods, Techniques and Applications*. Addison-Wesley, New York, USA.

[13] Hudak, P. (1998) Domain-specific languages. In Salus, P. H. (ed), *Handbook of Programming Languages Vol.* II: *Little Languages and Tools*. McMillan Technical Publishing, Indianapolis, USA.

[14] Thibault, S.A. Marlet, R. and Consel, C. (1999) Domain-Specific Languages: From Design to Implementation: Application to Video Device Drivers Generation. *IEEE Transactions on Software Engineering*, 25(3), 363-377.

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

[16] Krueger, C.W. (1992) Software Reuse. *ACM Computing Surveys*, 24(2), 131-183.

[17] Fernández-Manjón, B. and Fernández-Valmayor, A. (1997) Improving World Wide Web Educational Uses Promoting Hypertext and Standard General Markup Languages. *Education and Information Technologies*, 2(3), 193-206.

[18] Fernández-Manjón, B. Fernández-Valmayor, A. and Navarro, A. (1997) Extending Web Educational Applications via SGML Structuring and Content-based Capabilities. *IFIP International Conference the Virtual Campus: Trends for Higher Education and Training*, Madrid, Spain, 27-29 November, pp. 244-259. Chapman-Hall, London, UK.

[19] Navarro, A. Fernández-Manjón, B. Fernández-Valmayor, A. and Sierra, J.L. (2002) Formal-Driven Conceptualization and Prototyping of Hypermedia Applications. *5th International Conference on Fundamental Approaches to Software Engineering FASE 2002*, Grenoble, France, 8-12 April, pp. 308-322. Springer-Verlag (Lecture Notes in Computer Science 2306), Berlin, Germany.

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

[21] Navarro, A. Fernández, B. Fernández-Valmayor, A. and Sierra, J.L. (2004) The PlumbingXJ Approach for Fast Prototyping of Web Applications. *Journal of Digital Information*, 5 (2), http://jodi.tamu.edu/Articles/v05/i02/Navarro/.

[22] Juristo, N. and Pazos, J. (1993) Towards a joint life cycle for software and knowledge engineering, *IFIP Transactions* A-27, 119–138.

[23] Studer, R. Fensel, D. Decker, S. and Benjamins, V. R. (1999) Knowledge Engineering: Survey and Future Directions. In Puppe, F (ed), *Knowledge-based Systems: Survey and Future Directions*. Springer-Verlag (Lecture Notes in Artificial Intelligence 1570), Berlin, Germany.

[24] Sierra, J. L. Fernández-Manjón, B. Fernández-Valmayor, A. and Navarro, A. (2004) A Document-Oriented Approach to the Development of Knowledge-Based Systems. In Conejo, R. Urretavizcaya, M. and Pérez-de-la-Cruz, J.L (eds), *Current Topics in Artificial Intelligence*. Springer-Verlag (Lectures Notes in Artificial Intelligence 3040), Berlin, Germany.

[25] Sierra, J.L. (2004) *Hacia un Paradigma Documental de Desarrollo de Aplicaciones – Towards a Document-Oriented Paradigm to Application Development*. Ph.D. Thesis (in Spanish), Universidad Complutense, Madrid, Spain.

[26] Fernández-Valmayor, A. Guinea, M. Jiménez, M. Navarro, A. and Sarasa, A. (2003) Virtual Objects: An Approach to Building

Learning Objects in Archeology. In Llamas-Nistal. Fernández-Iglesias, M. J. and Anido-Rifon, L. E (eds). *Computers and Education: Towards a Lifelong Society*. Kluwer Academic Publishers, Dordrecht, The Netherlands.

[27] Navarro, A. Sierra, J. L. Fernández-Valmayor, A. and Hernanz, H. (2005) From Chasqui to Chasqui 2: An Evolution in the Conceptualization of Virtual Objects. *Journal of Universal Computer Science*, 11(9), 1518-1529.

[28] Sierra, J. L. Fernández-Valmayor, A. Guinea, M. Hernanz, H. and Navarro, A. (2005) Building Repositories of Learning Objects in Specialized Domains: The Chasqui Approach. *5th IEEE International Conference on Advanced Learning Technologies ICALT'05*. Kaohsiung, Taiwan, 5-8 July, pp. 225-229. IEEE Society Press, Los Alamitos, California, USA.

[29] Polsani, P. R. (2003) Use and Abuse of Reusable Learning Objects. *Journal of Digital Information*, 3(4), http://jodi.tamu.edu/Articles/v03/i04/Polsani/.

[30] IEEE. (2002) *IEEE Standard for Learning Object Metadata*. IEEE Standard 1484.12.1-2002, http://standards.ieee.org/.

[31] IMS. (2004) *IMS Content Packaging Information Model Version 1.1.2 Final Specification*. IMS Specification, http://www.imsglobal.org/.

[32] Cerami, E. (2002) *Web Services Essentials*. O'Reilly, Sebastopol, California, USA.

[33] Fernández-Manjón, B. and Sancho, P. (2002) Creating cost-effective adaptive educational hypermedia based on markup technologies and e-learning standards. *Interactive Educational Multimedia*, 4, 1-11.

[34] Sancho, P. Manero, B. and Fernández-Manjón, B. (2004) Learning Objects Definition and Use in <e-aula>: Towards a Personalized Learning Experience. *TC10 / WG10.5 EduTech Workshop – 18th IFIP World Computer Congress*, Tolouse, France, 22-27 August, pp. 177-186. Kluwer, Dordrecht, The Netherlands.

[35] Sierra, J. L. Moreno-Ger, P. Martínez-Ortiz, I. López-Moratalla, J. and Fernández-Manjón, B. (2005) Building Learning Management Systems Using IMS Standards: Architecture of a Manifest Driven Approach. *4th International Conference on Web Based Learning – ICWL 2005*, Hong-Kong, China, 31 July-3 August, pp. 144-156. Springer-Verlag, Lecture Notes in Computer Science 3583, Berlin, Germany.

[36] IMS. (2005) *IMS Question & Test Interoperability v 2.0*. IMS Specification, www.imsglobal.org.

[37] Moreno-Ger, P. Martínez-Ortiz, I. and Fernández-Manjón, B. (2005) The <e-Game> Project: Facilitating the Development of Educational Adventure Games. *Cognition and Exploratory Learning in Digital Age CELDA'05*, Porto, Portugal, 14-16 December, pp. 353-358, 2005, IADIS, http://www.iadis.org/.

[38] Martínez-Ortíz, I. Moreno-Ger, P. Sierra, J. L. and Fernández-Manjón, B. (2006) Production and Maintenance of Content-Intensive Videogames: A Document-Oriented Approach. *3th International Conference on Information Technology: New Generations ITNG'06*, Las Vegas, USA, 10-12 April, *in press*. IEEE Society Press, Los Alamitos, California, USA.

[39] Sierra, J. L. Fernández-Manjón, B. Fernández-Valmayor, A. and Navarro, A. (2000) Integration of Markup Languages, Document Transformations and Software Components in the Development of Applications: the DTC Approach. *International Conference on Software ICS 2000 – 16th IFIP World Computer Congress*, Beijing – China, 21-25 August, pp. 191-200. Publishing House of Electronic Industry, Beijing, China.

[40] Sierra, J. L. Fernández-Valmayor, A. Fernández-Manjón, B. and Navarro, A. (2001) Operationalizing Application Descriptions with DTC: Building Applications with Generalized Markup Technologies. 13th *Conference on Software Engineering and Knowledge Engineering SEKE'01*, Buenos Aires, Argentina, 13-15 June, pp. 379-386. Knowledge Systems Institute, Skokie, Illinois, USA.

[41] Sierra, J. L. Fernández-Valmayor, A. Fernández-Manjón, B. Navarro, A. (2003) Building Applications with Domain-Specific Markup Languages: A Systematic Approach to the Development of XML-based Software. 3th *International Conference of Web Engineering ICWE 2003*, Oviedo, Spain, 14-18 July, pp. 230-240. Springer-Verlag (Lecture Notes in Computer Science 2722), Berlin, Germany.

[42] Sierra, J.L. Fernández-Valmayor, A. Fernández-Manjón, B. and Navarro, A. (2004) ADDS: A Document-Oriented Approach for Application Development. *Journal of Universal Computer Science*, 10(9), 1302-1324.

[43] Sierra, J. L. Fernández-Manjón, B. Fernández-Valmayor, A. and Navarro, A. (2005) Document-oriented Software Construction based on Domain-Specific Markup Languages. *International Conference on Information Technology: Computing and Coding* ITCC'05, Las Vegas, USA, 4-6 April, pp. 392-397 Vol. II. IEEE Society Press, Los Alamitos, California, USA.

[44] Sierra, J. L. Fernández-Valmayor, A. Fernández-Manjón, B. and Navarro, A. (2005) Developing Content-Intensive Applications with XML Documents, Document Transformations and Software Components. *31th Euromicro Conference on Software Engineering and Advanced Applications*, Porto, Portugal, 31 August-2 September, pp. 4-11, IEEE Society Press, Los Alamitos, California, USA.

[45] Pressman, R.S. (2004) *Software Engineering*: *a Practitioner's Approach 6th Edition*. McGraw-Hill, Columbus, Ohio, USA.

[46] Boiko, B. (2005) *Content-management bible, 2nd edition*. Wiley Publishing, Indianapolis. USA.

[47] Graves, M. (2001) *Designing XML Databases*. Prentice-Hall, Englewood Cliffs, New Jersey, USA.

[48] Fuchs, M. (1997) Domain Specific Languages for *ad hoc* Distributed Applications. *First Conference on Domain Specific Languages*, Santa Barbara, California, USA, 15 October. USENIX.
Available at http://www.usenix.org/publications/library/proceedings/dsl97/fuchs.html.

[49] Aho, A. Sethi, R. and Ullman, J. D. (1986) *Compilers: Principles, Techniques and Tools*. Adisson-Wesley, Reading, Massachusetts,USA.

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

[51] Bray, T. Hollander, D. and Layman, A (eds). (1999) Namespaces in XML, *W3C Recommendation*, www.w3.org.

[52] Liang, S. Hudak, P. and Jones, M.P. (1995) Monad Transformers and Modular Interpreters. *22nd ACM Symposium on Principles of Programming Languages*, San Francisco, California, USA, pp. 333-343, 24-25 January. ACM Press, New York, USA.

[53] Duggan, D. A. (2000) Mixin-Based Semantic-Based Approach to Reusing Domain-Specific Programming Languages. 14th *European Conference on Object-Oriented Programming ECOOP'2000*, Cannes, France, 12-16 June, pp. 179-200, Springer-Verlag (Lecture Notes in Computer Science 1850), Berlin, Germany.

[54] Kastens, U. and Waite, W.M. (1994) Modularity and Reusability in Attribute Grammars. *Acta Informatica*, 31(7), 601-627.

[55] Sierra, J. L. Fernández-Manjón, B. Fernández-Valmayor, A. and Navarro, A. (2002) An Extensible and Modular Processing Model for Document Trees. *Extreme Markup Languages 2002*, Montreal, Canada, 6-9 August. IdeAlliance, www.idealliance.org. Available at http://www.mulberrytech.com/Extreme/Proceedings/html/2002/Sierra01/EML2002Sierra01.html
[56] Taivalsaari, A. (1996) On the Notion of Inheritance. ACM Computing Surveys 28(3), 438-479
[57] Bracha, G. and Cook, W. (1990) Mixin-based inheritance. *ACM SIGPLAN Notices*, 25(10), 303-311
[58] Mosses, P. D. (2004) Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61, 195-228.
[59] Knuth, D.E. (1968) Semantics of Context-free Languages. *Mathematical Systems Theory*, 2(2), 127-145.
[60] Paakki, J. (1995) Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation. *ACM Computing Surveys*, 27(2), 196-255.
[61] Makoto, M. Lee, D. and Mani, M. (2001) Taxonomy of XML Schema Languages Using Formal Language Theory. *Extreme Markup Languages 2001*, Montreal, Canada, 12-17 August, IdeAlliance, www.idealliance.org.
Available at http://www.mulberrytech.com/Extreme/Proceedings/html/2001/Murata01/EML2001Murata01.html
[62] Clark, J (ed). (1999) XSL Transformations (XSLT) Version 1.0. *W3C Recommendation*, www.w3.org.
[63] Birbeck, M et al. (2001) *Professional XML 2nd Edition.* WROX Press. Birminghan,UK.
[64] Kimber, W. E. (1998) A Tutorial Introduction to SGML Architectures. *ISOGEN Whitepaper*, ISOGEN International, www.isogen.com.
Available at http://xml.coverpages.org/kimberArchIntro980113.html
[65] ISO. (1997) *Hypermedia/Time-based Structuring Language (HyTime) – 2nd Edition*. ISO/IEC Standard 10744, http://www.iso.org/.
[66] Knuth, D.E. (1984) Literate Programming. *The Computer Journal*, 27(2), 97-111.
[67] Cordes, D. and Brown, M. (1991) The Literate Programming Paradigm. *IEEE Computer* 26(6), 52-61.
[68] Nørmark, K (2000) Requirements for an elucidative programming environment. *8th International IEEE Workshop on Program Comprehension* IWPC'00, pp. 119-128, Limerick, Ireland, 10-11 June, IEEE Society Press, Los Alamitos, California, USA.
[69] Collier, G. (2002) e-Learning Application Infrastructure. *SUN White Paper*, Sun Microsystems, www.sun.com.
Available at http://www.sun.com/products-n-solutions/edu/.
[70] Coplien, D. Hoffman, D. Weiss, D. (1998) Commonality and Variability in Software Engineering. *IEEE Software* 15(6), 37-45.
[71] Macala, R. R. Stuckey, L. D. and Gross, D. C (1996). Managing domain-specific product-line development. *IEEE Software*, 13(3), 57-67.
[72] Clark, T. Evans, A. Sammut, P. and Willans, J. (2004) An eXecutable Metamodelling Facility for Domain Specific Language Design. *The 4th OOPSLA Workshop on Domain-Specific Modeling*, Vancouver, Canada, 24 October, Technical Report TR-33, University of Jyväskylä, Finland.
Available at http://www.dsmforum.org/events/DSM04/papers.html.
[73] Mauw, S. Wiersma, W. T. and Willemse, T. A. C. (2004) Language-driven System Design. *International Journal of Software Engineering and Knowledge Engineering*, 14(6), 625-664.
[74] Wadler, P. (1999) The next 700 markup languages. *Second USENIX Conference on Domain Specific Languages* (Invited Talk: Slides available at http://homepages.inf.ed.ac.uk/wadler), Austin, Texas, USA, 3-5 October. USENIX, www.usenix.org.
[75] Cleaveland, J. C. (2001) *Program Generators with XML and Java*. Prentice-Hall, Englewood Cliffs, New Jersey, USA.
[76] Renear, A. Mylonas, E. and Durand, D. (1996) Refining our Notion of What Text Really Is: The Problem of Overlapping Hierarchies. In Hocky, S. and Ide, N. (eds), *Research in Humanities Computing*. Oxford University Press, Oxford, UK.
[77] Nakatani, L.H. Ardis, M.A. Olsen, R.G. and Pontrelli, P.M. (2000) Jargons for Domain Engineering. *ACM SIGPLAN Notices*, 35(1), 15-24.
[78] Nakatani, L.H. Jones, M. (1997) Jargons and Infocentrism. *First ACM SIGPLAN Workshop on Domain-Specific Languages DSL'97*, Paris, France, 18 January, pp. 59-74. ACM Press, New York, USA.
[79] Altheim, M. Boumphrey, F. Dooley, S. McCarron, S. Schnitzenbaumer, S. and Wugofski T. (eds). (2001) Modularization of XHTML, *W3C Recommendation*, www.w3.org.