ADDS: A Document-Oriented Approach for Application Development

José Luis Sierra
(Dpto.Sistemas Informáticos y Programación, Fac. Informática, Universidad Complutense
Madrid, Spain
jlsierra@sip.ucm.es)

Alfredo Fernández-Valmayor
(Dpto.Sistemas Informáticos y Programación, Fac. Informática, Universidad Complutense
Madrid, Spain
alfredo@sip.ucm.es)

Baltasar Fernández-Manjón
(Dpto.Sistemas Informáticos y Programación, Fac. Informática, Universidad Complutense
Madrid, Spain
balta@sip.ucm.es)

Antonio Navarro
(Dpto.Sistemas Informáticos y Programación, Fac. Informática, Universidad Complutense
Madrid, Spain
anavarro@sip.ucm.es)

Abstract: This paper proposes a *document oriented paradigm* to the development of content-intensive, document-based applications (e.g. educational and hypermedia applications, and knowledge based systems). According to this paradigm, the main aspects of this kind of applications can be described by means of *documents*. Afterwards, these documents are marked up using descriptive domain-specific markup languages and applications are produced by the automatic processing of these marked documents. We have used this paradigm to improve the maintenance and portability of content-intensive educational and hypermedia applications. ADDS (*Approach to Document-based Development of Software*) is an approach to software development based on the document oriented paradigm. A key feature of ADDS is that formulation of domain-specific markup languages is a dynamic and eminently pragmatic activity, and markup languages evolve according to the authoring needs of the different participants in the development process (*domain experts* and *developers*). The evolutionary nature of markup languages in ADDS leads to OADDS (*Operationalization in ADDS*), the proposed operationalization model for the incremental development of *modular* markup language processors. Finally, the document-oriented paradigm can also be applied in the construction of OADDS processors that are also described using marked documents. This paper presents our ADDS approach, including the operationalization model and its implementation as an object-oriented framework. The application of our document-oriented paradigm to the construction of OADDS processors is also presented.

Keywords: Software Development Approach, Domain-Specific Markup Languages, Software Maintenance, Software Evolution, Modular Language Processors, XML
Categories: D.3.2, D.3.4, D.2.13, D.1.0, D.3.3, D.2.3, D.2.7, I.7.2

1 Introduction

Documents are the basic tool for regulating and structuring communication inside a large number of organizations and are the basis for a broad class of software applications (e.g. educational and hypermedia applications, and knowledge based systems). The adoption of what we call a *document-oriented paradigm* for application development takes advantage of this fact. According to this paradigm, the main aspects of these applications (e.g. their data and the relevant parts of their behaviors) can be described using documents. Therefore, the applications themselves can be built by the automatic processing of these documents. The feasibility of this paradigm depends on the existence of mechanisms capable of making the structure of the documents describing the applications explicit for people and machines. Descriptive domain-specific markup languages [Coombs,87] provide these mechanisms.

The approach described in this paper was formerly suggested in [Fernández-Manjón,97a][Fernández-Manjón,97b] as a vehicle to improve the development and the maintenance of educational applications. The work in [Fernández-Valmayor,99] reports on the application of these ideas in the context of the EU LINGUA project *Galatea*. The main methodological goal of *Galatea* was the provision of a set of guidelines governing the development of educational applications. The instructional goal was to obtain comprehension of texts written in a foreign language close to the mother tongue of the student. In *Galatea* the communication between the two main actors in the development process, *linguists* and *software developers*, was articulated via marked documents. Indeed, after evaluating the application, the linguists could include their modifications in the documents, and mark up these documents with easy-to-use descriptive markup languages specific to this domain. Then the marked documents were automatically processed by the developers to incorporate the modifications in the final application.

The experience in *Galatea* led us to generalize the approach in the broader field of hypermedia domain. The result was the *PlumbingXJ* approach for the fast prototyping of hypermedia applications [Navarro,02][Navarro,03][Navarro, 04a][Navarro, 04b]. In this approach, prototypes for complex hypermedia applications can be easily characterized by marked documents and they can be automatically produced from these documents using a prototype generator. This facilitates the interaction between *domain experts* and *developers* during the prototyping stage in the development of complex content-intensive hypermedia.

This paper presents a step by step overview of ADDS (*Approach to Document-based Development of Software*), an implementation of the document-oriented paradigm that underlies the works in *Galatea* and *PlumbingXJ*. Previous work in ADDS, including several experimental developments applied to different domains, can be found in [Navarro,00][Sierra,00][Sierra,01][Sierra,02][Sierra,03][Sierra,04]. According to ADDS, the development of an application starts with the provision of a *domain-specific descriptive markup language* (DSML) that will be used to make the semantic structure and the relevant data of the document describing the application explicit. Next, this document is written and marked up using the DSML provided. The application itself is obtained by the automatic processing of this marked document with a suitable processor for the DSML defined. Then application development evolves iteratively while the document is modified and refined. This iterative process

can lead to the modification and/or extension of the DSML. In this way, the use of markup languages in ADDS is dynamic and pragmatic: DSMLs are not conceived as static entities, but evolve along with the expressive needs of designers and developers.

DSML evolution implies, in its turn, the evolution of its associated processors. Consequently, to facilitate this evolution, a suitable mechanism for the incremental development of these processors must be provided. OADDS (*Operationalization in ADDS*) is the ADDS model for the incremental construction of processors. OADDS encourages the construction of *modular* processors from components that can be combined and extended as the associated DSMLs evolve.

OADDS is independent of any specific implementation technology. Nevertheless, OADDS can easily be implemented as an object-oriented framework. The main advantage of this implementation is that it promotes its integration with widely used object-oriented frameworks for document processing [Birbeck,01]. Finally, OADDS processors can themselves be described by documents. Therefore, the document-oriented paradigm can be applied in the construction of these processors.

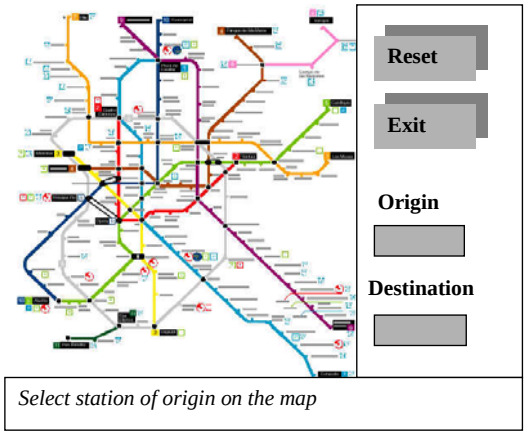


Figure 1: Sketch of a user interface for a route-searching application in subway networks. Users select the stations of origin and destination on the map, and the application computes and visualizes a route between the selected stations

The structure of the paper is as follows: Section 2 describes the ADDS approach. Section 3 describes the OADDS operationalization model. Section 4 presents an object-oriented framework implementing OADDS. Section 5 outlines OADDSML (*OADDS Markup Language*), a (meta) DSML for marking up OADDS processor documents. Section 6 describes some related work. Finally, section 7 gives some conclusions and future work. Throughout the paper, we will use the domain of the applications for searching for routes in subway networks as a case study to exemplify the different aspects described in our approach. Figure 1 depicts a proposed user interface for this type of applications.

2 The ADDS Approach

The ADDS approach introduces a set of guidelines for the analysis, construction and maintenance of document-based applications. Figure 2a shows the activities considered in ADDS and the sequencing of these activities. Figure 2b illustrates the products produced and consumed by these activities. Figure 2c shows the participants in these activities, together with the roles they play. The following subsections describe several of these aspects in detail.

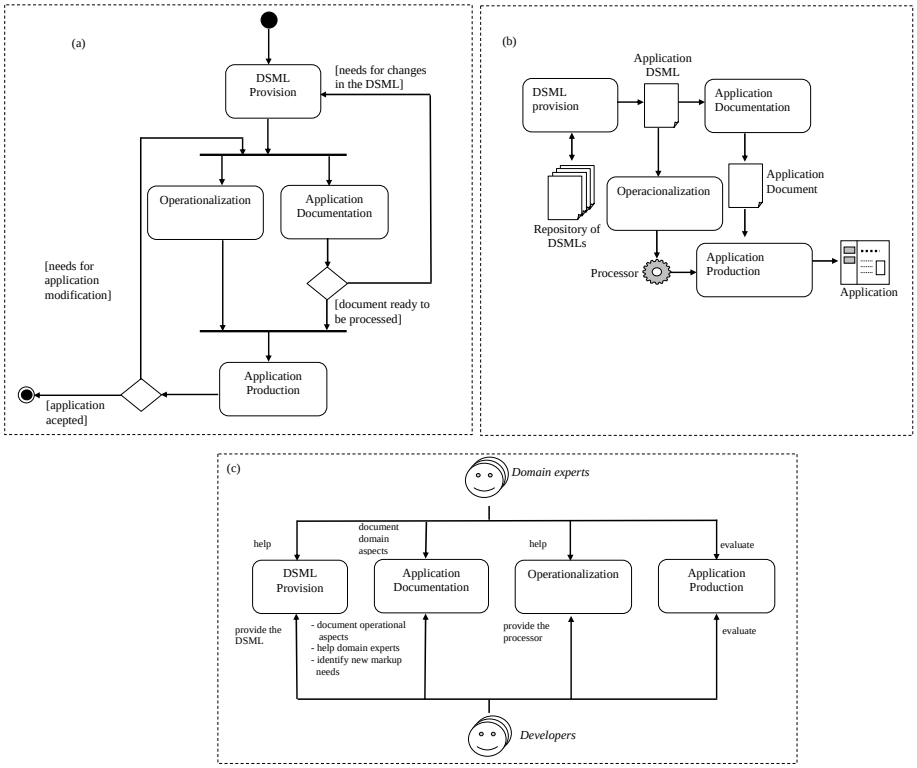
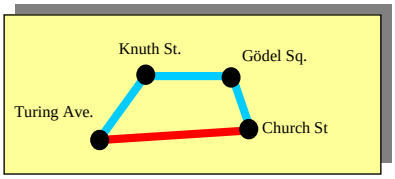


Figure 2: (a) Activities in ADDS and their sequencing, (b) products in ADDS and their production/consumption relationships with the activities, (c) participants in ADDS and their roles in the activities

(a)



(b)

```
<Subway>
 <Network>
  <Structure>
   <Stations>
    <Station id="TURINGAVE">Turing Ave.</Station>
    <Station id="KNUTHST">Knuth St.</Station>
    <Station id="GÖDELSQ">Gödel Sq.</Station>
    <Station id="CHURCHST">Church St.</Station>
   </Stations>
   <Lines>
    <Line id="BLUE">
     <Name>Blue Line</Name>
     <Link origin="TURINGAVE"
           destination="KNUTHST"
           length="5"/>
     <Link origin="KNUTHST"
           destination="GÖDELSQ"
           length="10"/>
     <Link origin="GÖDELSQ"
           destination="CHURCHST"
           length="500"/>
    </Line>
    <Line id="RED">
     <Name>Red Line</Name>
     <Link origin="TURINGAVE"
           destination="CHURCHST"
           length="10"/>
    </Line>
   </Lines>
  </Structure>
  <Dynamics>
   <Speeds>
     <Speed line="BLUE" value="50"/>
     <Speed line="RED" value="80"/>
   </Speeds>
   <WaitingTimes>
     <WaitingTime line="BLUE" value="10"/>
     <WaitingTime line="RED" value="5"/>
   </WaitingTimes>
   <TransferTimes>
    <TransferTimesInStation station="TURINGAVE">
      <Transfer origin="BLUE" destination="RED"
                time="4"/>
    </TransferTimesInStation>
    <TransferTimesInStation station="CHURCHST">
      <Transfer origin="BLUE" destination="RED"
                time="5"/>
    </TransferTimesInStation>
   </TransferTimes>
  </Dynamics>
 </Network>
 <UserInterface>
   <Title>Route searching in subway networks</Title>
   <ExitButton>Exit</ExitButton>
   <ResetButton>Reset Application</ResetButton>
   <OriginLabel>Origin</OriginLabel>
   <DestinationLabel>Destination</DestinationLabel>
   <Map loc="toysubwayEN.jpg"/>
   <InformativeMessages>
     <FirstStationSelection>Click on the first
      station</FirstStationSelection>
     <SecondStationSelection>Click on the second
      station</SecondStationSelection>
     <RouteVisualization>Route being visualized
      on the map</RouteVisualization>
   </InformativeMessages>
   <Coordinates>
     <Coordinate station="TURINGAVE" x="65" y="157"/>
     <Coordinate station="KNUTHST" x="157" y="29"/>
     <Coordinate station="GÖDELSQ" x="328" y="33"/>
     <Coordinate station="CHURCHST" x="365" y="135"/>
   </Coordinates>
 </UserInterface>
</Subway>
```

Figure 3: (a) A fictitious little subway network, (b) marked document describing the route-searching application for the network in (a)

2.1 Products

The construction of an application according to ADDS comprises the following kinds of products:

- The *application document* is a document describing the main application aspects. Such aspects include the *contents* (*data*) used by the application and other relevant aspects of the application (e.g. the structure of the GUI, and even some aspects regarding application behavior). This document is marked up and evolves throughout the development process. Usually, this document can include two different types of aspects: (i) *domain aspects*, information related to the domain of the problem solved by the application, and (ii)

operational aspects, information not directly related to the problem domain, but required to produce the final application. In our case study, this document will include domain aspects concerning the network structure (e.g. list of stations and links in each line) and dynamics (e.g. average speeds and waiting times in each line, and transfer times for each station).This document will also include operational aspects concerning high-level variability of the user interface (e.g. button and label names, informative messages and station coordinates on the subway map). Figure 3a drafts a fictitious little subway network and Figure 3b shows the document of the route-searching application for this.

- The *application* DSML. Description, using a *document grammar*[1], of the domain-specific descriptive markup language used to make the structure and the data of the application document explicit. This language could evolve during the development of the application to accommodate the evolution of its requirements. In our example application domain, the application DSML will enable the kind of markup outlined in Figure 3b.

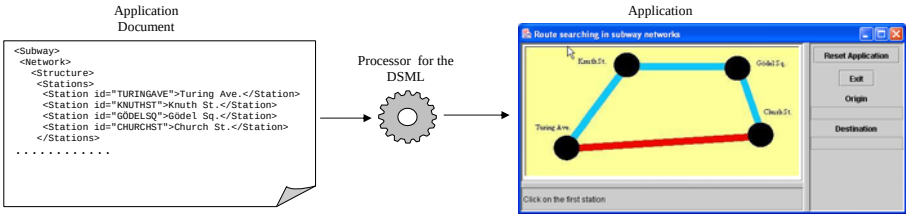


Figure 4: Applications are produced by the automatic processing of the documents describing them with suitable processors

- The *repository of* DSMLs. Descriptions of descriptive markup languages already available that can be combined and/or extended to achieve new DSMLs. The grammar-based declarative description of DSMLs in ADDS eases the combination and extension of simpler languages to yield more complex ones. Therefore, this repository helps to decrease the cost associated with the formulation of new languages. In our application domain, this repository will include the different sublanguages that constitute the application DSML (e.g. a sublanguage for marking up tables documenting subway network structures, another for subway network dynamics, a third one for the variability of the user interface, etc).
- The *processor*. Processor for the application DSML. This processor enables the production of the application from its document.
- The *application*. Application produced by processing the application document with the processor for the DSML used to mark it up (Figure 4).

[1] ADDS uses a grammatical formalism based on XML DTDs [W3C], although the approach can easily be adapted to any other document grammar formalism (e.g. [Lee,00]).

2.2 Participants

ADDS differentiates between two kinds of participants in the construction of an application:

- *Domain experts*. They are experts on different aspects of the application's problem domain. Hence, they are responsible for documenting and maintaining the aspects of the application related to their domain of expertise, although they could also understand and modify some of the operational aspects. The kind of experience expected from such experts strongly depends on the application domain. Because the same application can integrate aspects from different knowledge areas, this community of experts is interdisciplinary in nature. In our application domain, there are experts in network organization, able to document net structure and dynamics, but also there are experts able to deal with operational aspects, such as experts in graphic design, who may provide the subway maps, and experts in application customization, who can adapt the application to different use scenarios by modifying the documentation of the operational aspects.
- *Developers*. They are experts in computer science. Their main responsibilities are the formal definition of the application DSML, using an appropriate grammar formalism, and the construction of the processor for this DSML. They are also responsible for documenting the operational aspects in the application document. Like the domain experts, it is possible to distinguish different kinds of developers. *Software experts* are experts in the development of the software infrastructure for the operations in the application domain (e.g. graph-searching algorithms in the subway domain). *DSML experts* are experts in the formulation and maintenance of the DSMLs. *Experts in language processors* are experts in the provision and maintenance of the processors for the DSMLs. The basic semantic actions of these processors will use the software provided by the software experts. Finally, *experts in document management* are responsible for managing the organization of the application document. Note that an expert could participate in different categories (e.g. an expert in language processors could also play the role of a software expert).

2.3 Activities

ADDS introduces the following activities:

- *DSML provision*. This activity is the most characteristic and critical in the document-oriented paradigm, and the most knowledge-demanding task. The goal of this activity is to produce an appropriate application DSML. Such a DSML can be formulated from scratch, or it can integrate other languages already available in the repository of DSMLs. This integration is facilitated by the modular nature of the grammar-oriented declarative description of languages. The formulation of this DSML will usually be based on documentation and documentation styles used in the application domain, and will take advantage of the experience in the development of similar applications (represented by the DSMLs previously used in the domain).

- *Application documentation*. The goal of this activity is the authorship and markup of the application document describing the aspects required to produce the application.
- *Operationalization*. This activity yields an appropriate processor for the application DSML. The provision of such a processor can be incrementally carried out, and follows the OADDS operationalization model described in the next section.
- *Application Production*. In this activity, the application is produced by applying the DSML processor to the application document.

ADDS follows an incremental development strategy. The sequencing of these activities introduces the following loops in the production of an application (Figure 5):

- *Maintenance loop*. In this loop the application document is processed to produce the application. The evaluation of the application could lead to the modification of the application document in order to achieve a better result. For instance, in our example application domain, we can document an initial subset of the subway network, in order to provide a first working prototype of the final application. Next, we can complete this documentation to tackle the overall network, and, then, in a third iteration we can tune the operational aspects. New maintenance iterations can arise during application exploitation when the network changes (for instance, due to the addition of a new station or a new line). Note that, from a software development point of view, the cost associated with each iteration is usually small (ideally zero), because the production of the application is reduced to applying the DSML processor to the application document. But sometimes this processor might need to be adapted to correct some errors and/or misunderstandings of the operational meaning required for the DSML. Usually, these cases will be less frequent than changes in the document.

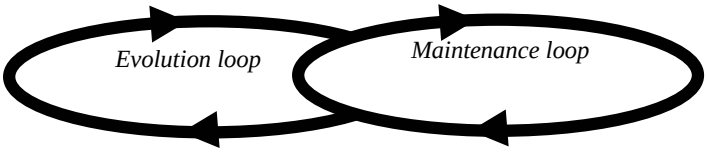


Figure 5: ADDS application development loops

- *Evolution loop*. This loop, less frequent, arises during the documentation activity, when new markup needs are identified. Such needs can arise due to a refinement of the document structure, or the incorporation of new aspects to take into account new requirements. Thus, the usual maintenance cycle is left, and the DSML provision activity is performed again (i.e. the DSML evolves). The consequences of this evolution are the addition of new structures to the DSML and the corresponding extension of the DSML's processor. Finally, the usual maintenance loop is entered again. In our subway example, the DSML could 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 interfaces styles (e.g. evolutions from a simple console-based user interface to a graphical user interface).

Note that DSMLs in ADDS evolve incrementally avoiding the high costs associated with exhaustive domain and program family analysis used in other approaches to software development based on domain-specific languages [Van Deursen,00].

3 OADDS: Operationalization in ADDS

OADDS defines the operationalization model that yields an appropriate processor for the application DSML. OADDS combines three types of processes (Figure 6): (i) *document tree construction* for the application documents, (ii) *document tree operationalization* by assigning different *operational components* to the element nodes in the trees, and (iii) *evaluation* of the operationalized document trees. These processes can be compared with those arising in classical syntax-directed construction of language processors [Aho,86]. Indeed, the construction of document trees is analogous to the *syntax analysis* stage, the tree operationalization generalizes the *syntax tree decoration* stage, and the tree evaluation is analogous to the corresponding *evaluation* stage of the decorated syntax tree. Anyway, given that descriptive markup makes the tree structures on the documents explicit, it is feasible to decouple these three processes, setting up the basis for a better modularity of the processors. In addition, while tree construction and tree evaluation processes are invariant for every processor, tree operationalization is specific for each language. OADDS regulates the tree operationalization process, allowing for an incremental formulation of the processor.

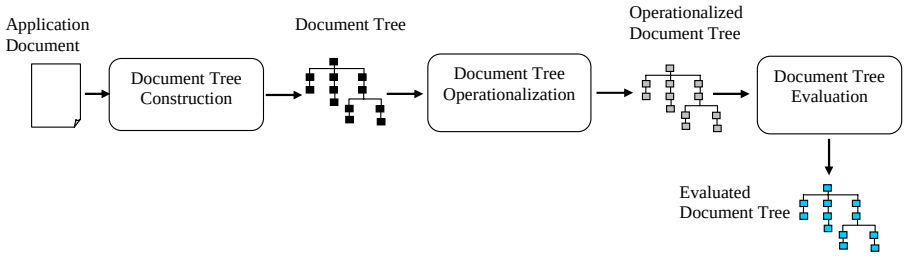


Figure 6: The three types of basic processes carried out by the OADDS processors

The following subsections detail the different aspects regarding OADDS processor construction.

3.1 Document Tree Construction

Document tree construction is carried out using a component, called the *tree builder*, which allows the construction of suitable representations of the document trees associated with the application documents. Previously to the tree construction, this

component performs the analysis of the document and validates it using the document grammar of the DSML. The tree builder can use a standard framework for the parsing of structured documents [Birbeck,01]. This same component can be reused in every processor, which contributes to substantially decreasing the overall cost in the construction of processors. Indeed, syntactic analysis aspects are an important part of the development, as illustrated in the standard literature about language processor construction [Aho,86].

3.2 Document Tree Operationalization

Figure 7 sketches the different types of components used in the formulation of OADDS operationalization processes, together with their main relationships.

The operationalization of document trees is performed by assigning a set of suitable *basic operational components* to each element node. These components establish how their associated nodes are to be evaluated. More precisely, with each element node in the document tree it is possible to assign (Figure 8a):

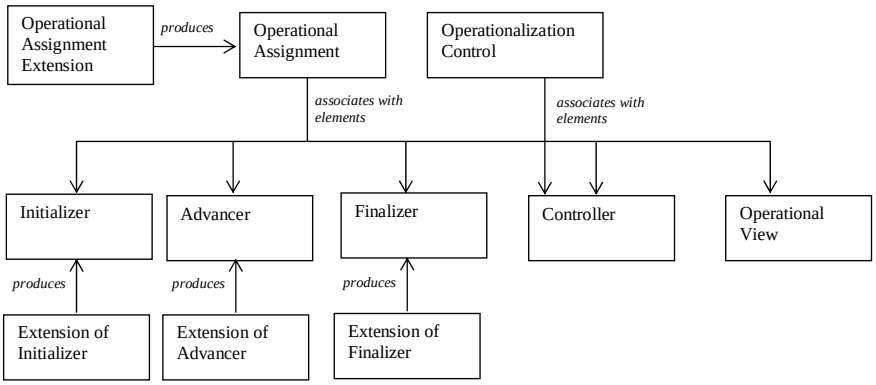


Figure 7: Operational components involved on the formulation of operationalization processes for document trees in ADDS

- A set of *operational views*, which encapsulates the results of the evaluation of this node.
- A *controller*, which is used to establish an evaluation order on the neighbors of the node. This is a procedure that, taking an element as input, returns the list of nodes that will be used to continue the evaluation of the tree.
- An *initializer*, which is used to initiate the evaluation of the node. This is a procedure taking the element to be initiated as input.
- An *advancer*, which allows the continuation of the node evaluation after the evaluation of each neighbor. This component also allows the interruption of the evaluation process. This is a procedure that takes the element and one of its neighbors as input, carries out the continuation of the evaluation, and returns *false* if this evaluation has to be interrupted, or *true* in other cases.
- A *finalizer*, used to finish the evaluation of the node. This is a procedure taking the element to be finished as input.

These types of components can be related to classic concepts of syntax directed translation. Indeed, operational views correspond to *attribute tables* associated with the different nodes in the syntax tree. Initializers, advancers and finalizers encapsulate the semantic actions performed on these nodes during the translation process. Finally, controllers decide the traversal of the tree during such a process. The reason to break the classical process of tree evaluation into these components is to facilitate the modular construction of processors and their evolution according to changes in the DSML. Indeed, the incremental formulation of the operationalization processes is based on the incremental formulation of the basic operational components assigned to the element nodes. In order to do so, it is possible to use *extensions of initializers* (*of advancers*, *of finalizers*). These components make the way to add new functionalities to one or more initializers (advancers, finalizers) explicit.

Figure 8b illustrates the operationalization of the elements of type `Network`, used to mark up the documentation of the subway networks in our example application domain. These elements have an operational view associated, called *Digraph view*, which will contain a reference to an interpretation of the subway network as a weighted directed graph. Figure 8c, using pseudo-code, describes the controller, the initializer and the advancer for this type of nodes.

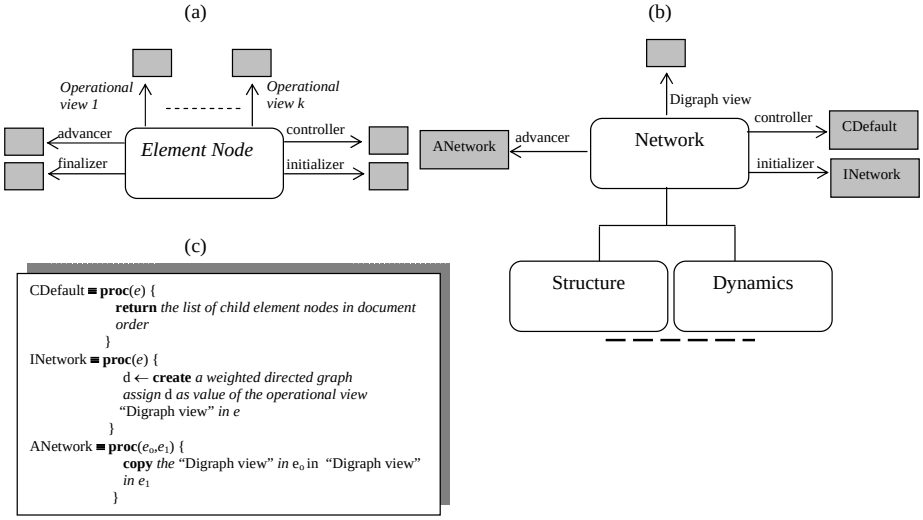


Figure 8: (a) Operationalization of element nodes, (b) Operationalization for elements of type Network, *(b) pseudo-code for the components named in (b)*

The operationalization process is governed by:

- An *operational assignment*, which is used to associate basic components with element nodes. Operational assignments can be incrementally formulated by using *operational assignment extensions*.
- An *operationalization control*, which decides the traversal of the document tree during such an operationalization.

As with the basic operational components, it is also possible to relate these two components with classic scenarios of syntax-directed translation. Therefore, operational assignments are analogous to *translation schemas*. In this way, operationalization controls can be applied to control the application of the assignments. This can be useful, for instance, for the incremental operationalization of trees, where trees are subjected to dynamic modifications and are incrementally operationalized after each modification.

Finally, operationalization itself is carried out using a component called *operationalization driver*. This component is configured by means of an operational assignment and an operationalization control. This can be applied to an element node for operationalizing the tree where the node is located. This application supposes:

1. The application of the operational assignment to such a node for yielding the associated basic operational components.
2. The assignment of these components with the node.
3. The application of the operationalization control to this node to obtain a controller.
4. The operationalization, in order, of the nodes provided by the controller.

3.3 Evaluation of Document Trees

The evaluation of a document tree uses the controllers, initializers, advancers and finalizers associated with element nodes to update the values of the operational views in these nodes. The applications described by the documents can usually be obtained from the operational views in the roots of their trees after tree evaluation. For instance, in our example application domain, the evaluation of the sub-tree for the documentation of the network yields a weighted directed graph, while the evaluation of the sub-tree corresponding with the user interface variability produces a description of such variability. These results are integrated to produce the final application.

The evaluation process is carried out using a component called *evaluation driver*. This component starts the evaluation by *visiting* an element node of the tree. Each time that an element node *e*, which has an associated controller *c*, an initalizer *i*, an advancer *a* and a finalizer *f*, is visited:

1. *i* is invoked on *e*.
2. *c* is used to iterate on the neighbors of *e*.
3. Each time the evaluation of a neighbor *n* of *e* finishes, *a* is invoked on *e* and *n*. If the result returned by *a* is *false*, the evaluation of *e* is interrupted.
4. The driver invokes *f* on *e* when all the neighbors have been visited.

3.4 Incremental Development of Processors

The OADDS model encourages *semantic modularity* [Hudak,98]. Semantic modularity in OADDS is reflected in the nature of the operationalization processes. Indeed, in this model it is possible to incrementally add new features of local evaluation to the element nodes, by extending their initializers, advancers and finalizers (Figure 9). The newly added features facilitate the achievement of the following extensions:

- *Propagation of new values* from an element node to their neighbors, from the neighbors to the element node, and from neighbors to neighbors.

- *Interruption of the evaluation*. This will usually arise as a response to an error.

Note that these are the typical extensions achieved in the different approaches to the modular development of language processors (e.g. [Kastens, 92][Hudak,98][Duggan,00]).

The realization of this type of extensions in OADDS is straightforward. Therefore, each new feature added is performed by means of (i) a set of extensions of initializers (advancers, finalizers) that act on the extended initializers, advancers and finalizers respectively, and (ii) an operational assignment extension, which acts on the extended assignments, and applies extensions of the appropriate type to the components produced by these extended assignments. For instance, in our example application domain, the inclusion of more advanced interaction capabilities in the user interface will need the list of stations in the network in addition to its interpretation as a graph. The propagation of this list can be incrementally added to the evaluation process by considering appropriate extensions for the available basic components and the operational assignments. On the other hand, the set of operational views provided are obtained from those provided by the extended assignments, together with the operational views introduced by the extension.

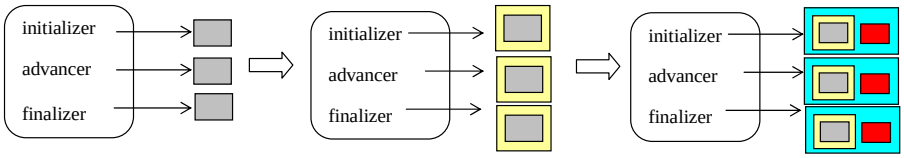


Figure 9: Incremental development of OADDS processors depends on the incremental extension of the components characterizing the local evaluation of the element nodes in the document tree

This schema suggests a systematic way for the incremental development of processors, centered on the incremental formulation of their operationalization processes. This incremental formulation supposes that:

1. It starts with the *default operational assignment*, which applies *default* initializers, advancers and finalizers (which do not perform any action) to each element node.
2. The new initializers, advancers and finalizers are formulated as extensions of the pre-existing ones.
3. The applications of these extensions are performed by means of appropriate operational assignment extensions.
4. The final operational assignment is obtained by composing the assignment extensions in an appropriate order. The result is composed of an assignment extension which establishes the evaluation control regime by assigning appropriate controllers to each element node.

The approach is similar to that described in [Hudak,98], based on the composition of *monad transformers* and the application of the resulting transformer to the *unit monad*.

4 An Object-Oriented Framework for OADDS Support

OADDS has been implemented as an object-oriented framework. Figure 10 shows the organization of such a framework in layers. The framework distinguishes:

- A *basic layer* including classes representing the basic concepts of the operationalization in OADDS.
- A *processing layer* defining the elements required to provide the basic processes for the processors. This layer also characterizes the interface followed by such processors.
- A *component layer* containing the concrete operational components used in the operationalization of DSMLs.

In this way, the basic and processing layers introduce the foundation classes and interfaces used for building OADDS processors. In addition, the component layer supposes the implementation of these processors as an appropriate extension and specialization of such foundations classes and interfaces. The following subsections describe these layers.

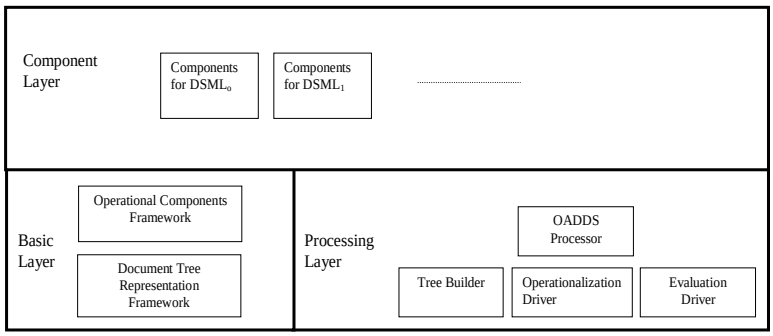


Figure 10: Layered Organization of the framework for OADDS support

4.1 Basic Layer

This layer introduces the concepts for representing document trees, as a specialization of DOM [W3C], and those for the OADDS operational components (*controller*, *initializer*, *finalizer*, *operational assignment*, etc.). This layer includes:

- A *document tree representation framework*. This is a specialization of DOM allowing for the representation of document trees in a way that is appropriate for performing the operationalization and evaluation of trees in OADDS (Figure 11a). Consequently, it introduces the implementation `OADDSElement` of the DOM `Element` interface (the interface for the element nodes in the document trees). This implementation wraps both the operationalization of the element node and the structural representation of the node (in terms of a standard implementation of `Element`). Operationalization itself is represented using objects of type `Operationalization`. Moreover, the framework includes an

implementation `OADDSDocument` of the DOM `Document` interface (the interface characterizing the DOM document trees and containing factory methods for each type of node in such trees). Thus, the tree builder configures the parsing framework with this implementation, allowing the production of document trees appropriate for the OADDS processing.

- An *operational components framework*. Figure 11b characterizes the interfaces for the different operational components used in the formulation of operationalization processes. The framework also introduces several default components. Note that the object-oriented paradigm makes the explicit provision of extensions for the basic components unnecessary, because such extensions can be characterized as implementations of the `Initializer`, `Advancer` and `Finalizer` interfaces that take the extended components as parameters in their constructors.

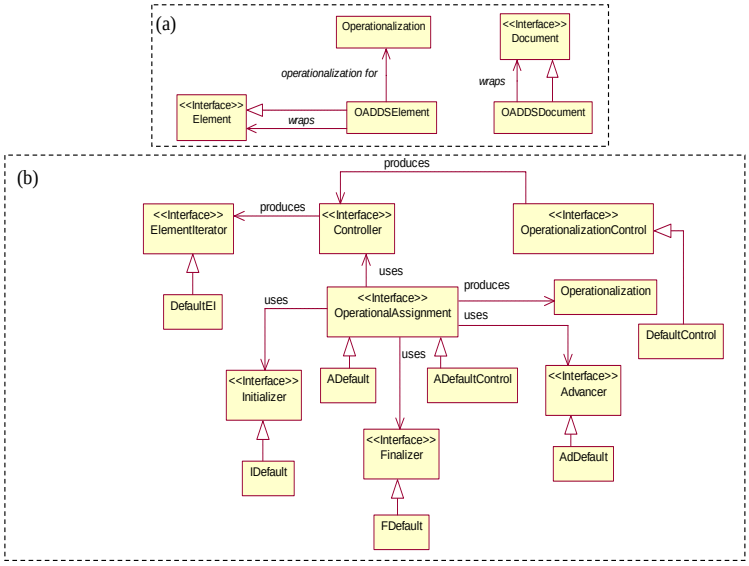


Figure 11: (a) Document tree representation framework, (b) Operational components framework

4.2 Processing Layer

The processing layer (Figure 12) characterizes the components performing the basic processes in OADDS (*tree builder*, *operationalization* and *evaluation* drivers), and also the common interface implemented by the processors.

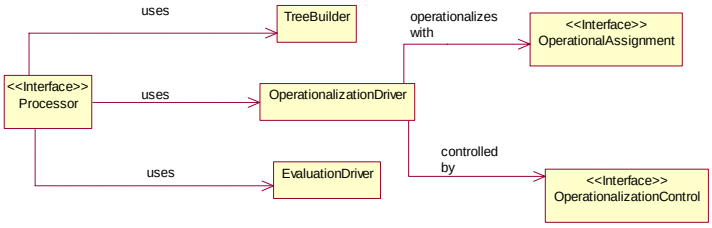


Figure 12: Class diagram for the processing layer

4.3 Components Layer

The component layer contains the components used in the operationalization of the different DSMLs. Almost all the extensions of the operationalization framework are carried out in this layer.

5 Documentation of OADDS Processors

The OADDS processors themselves can be viewed as particular cases of applications that can be built following the document oriented paradigm. In this way, it is possible to document this kind of processors, to mark up such documents with an appropriate DSML, and to process these documents in order to produce the documented processors. The DSMLs used to mark up the documentation of processors are called *meta* DSMLs.

OADDSML (*OADDS Markup Language*) is a meta DSML allowing the markup of the documentation of the different operational components in OADDS. Therefore, the OADDSML documents describe collections of OADDS components, including the OADDS processors. Such documents can be processed with an OADDSML processor to execute the documented OADDS processors. The OADDSML processor can be located at the same level as the classical generators of language processors (e.g. YACC [Johnson 75]).

OADDSML introduces the following documentation styles for the operational components:

- Documentation of controllers is given in terms of *vicinities* of the associated element nodes. The OADDSML markup allows the formalization of these descriptions in terms of regular path expressions [Abiteboul,00], using XPath [W3C].
- Initializers, advancers and finalizers are documented by enumerating the literal information from the document (*parameters*) and the operational views required by the evaluations carried out by these components, and by referring to the *semantics actions* performing such evaluations. The OADDSML markup of this documentation formalizes the enumeration of the parameters and the views (using, again, XPath), and marks up the given semantic actions.

- Operationalization controls are documented by *control rules*. Each rule describes the association of controllers with element types. The OADDSML markup formalizes the conditions of these rules (using XPath), and marks up the controllers specified by them.
- Operational assignments are documented by means of *operationalization actions*. Each action indicates how to operationalize the different types of element nodes. OADDSML enables the formalization (using XPath) of the applicability conditions for the actions, and the markup of each operationalization aspect carried out by these actions.
- Processors are documented by defining the different operationalization processes to be performed on the document trees, and by referring to the domain-dependent logic of the processor using such processes. OADDSML allows the markup of each one of these aspects.

OADDSML follows the modular nature of OADDS. Consequently, from the OADDSML documents it is possible to refer to components documented in other documents and even to components provided directly in terms of the operationalization framework.

OADDSML hides the complexities of the direct use of the operationalization framework from the developers. Indeed, with OADDSML the development of processors is restricted to:

- Documenting and marking up the different operational components. Such documentation and markups exhibit a higher abstraction level than the direct programming of the components in terms of the framework.
- Providing modules with *procedures* (classes with static methods) containing the basic semantic actions, together with the domain-dependent logic for the processors. These collections of procedures can be provided in simpler terms than those given by the operationalization framework, and they can serve as a link with the basic software provided by the ADDS experts in software.

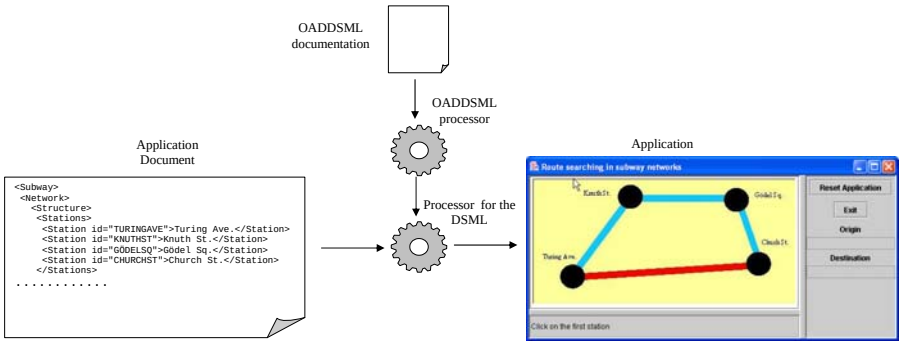


Figure 13: The OADDSML documentation can be processed with an OADDSML processor for yielding the documented OADDS processor

The provision of an OADDS processor for OADDSML allows for the production of OADDS processors from their documentation. In their turn, the processors

produced can be applied to the corresponding application documents for yielding the documented applications. This behaviour is illustrated in Figure 13.

6 Related Work

The original use of descriptive markup languages was for the processing of electronic documents [Goldfarb,90]. HyTime [HyTime,97], an SGML [Goldfarb,90] extension for the description of hypermedia applications, demonstrated that in some domains, these kinds of languages could be used for enabling the full description of applications in terms of documents that, in turn, can be processed for building the final application. Moreover, proposals like DSSSL [DSSSL,96] proved that this document-oriented paradigm could be used not only for the applications, but also for describing the processors used to produce the applications from their documentation. XML [W3C] 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. Note that most of these approaches conceive of markup languages as *static* entities. On the contrary, ADDS takes a more pragmatic position, where markup languages are considered *dynamic* objects that evolve when the contents, or the markup needs of these contents, change. OADDS gives an operational solution to this dynamic nature of the languages, encouraging the construction of modular processors from components that can be extended and adapted according to language evolution.

The document oriented paradigm has the same aim as other classical approaches to derive programs from structured documents and diagrams [Warnier,81][Orr, 81][Jackson,75], and the more recent efforts to derive programs from models [Frankel, 2003]. Nevertheless, our approach has a more limited scope and methodology: it is oriented to building content-intensive and document-based applications (e.g. educational and hypermedia applications and knowledge-based systems) and it takes advantage of a linguistic and meta-linguistic orientation. The document-oriented paradigm and ADDS also share some features with the seminal work of Knuth on *literate programming* [Knuth, 84]. This work makes the benefits of identifying the programs and their documentation explicit. In literate programming, the program's code is interleaved with its documentation, 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 and the production of documentation printouts. The ideas described in this paper differ from literate programming, because only the high level aspects of the applications, but not the code of the programs implementing them, are documented and marked up. The code itself is implicitly contained in the processor for the DSML used in the markup, in the same way that the assembler code for the programs in a high-level programming language is contained in the compiler for this language. Because of this, in our work, suitable DSMLs are provided *for each* application domain instead of using a fixed markup language, as in literate programming. In this way, literate programming could be seen as a particular application of our document-oriented paradigm. Indeed, OADDSML documents are similar to documents in the literate programming paradigm, because they are documenting processors for other DSMLs. This reveals the metalinguistic nature of ADDS.

ADDS also shares many features with the approach to software development based on *Domain-Specific Languages* (DSLs [Van Deursen,00]). [Fuchs,97] is a pioneering work in the application of SGML/XML for the definition of DSLs. In [Wadler,99] the relationships between markup languages and the DSL approach is highlighted. Although these works recognize the potentiality of markup metalanguages as a vehicle for defining DSLs, the stress is put on their use to formalize abstract syntax, instead of their use as descriptive markup (meta) languages.

Jargons foundations [Nakatani,97][Nakatani,99] are similar to that promoted by ADDS. 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 ADDS, ADDS considers it unrealistic to assign language design and operationalization responsibilities to domain experts. Instead, ADDS involves a community of developers for this purpose. Moreover, Jargons does not contemplate the semantic modularity problem in the operationalization of DSLs. This problem is critical when these languages evolve.

Modular language processor construction has been popularized inside the functional programming community, where the main approach is based on *monads* and *monads transformers* [Hudak,98], although it is also possible to find proposals in the object-oriented paradigm (based on the use of *mixins* [Duggan,00]), and in the attribute grammar approach to the construction of language processors [Knuth, 68][Kastens, 92]. OADDS semantic modularity mechanisms are inspired by these proposals, and also resemble the extension mechanisms of methods in CLOS [Steele,90]. Indeed, the extensions of initializers, advancers and finalizers are similar to the definition of *before*, *around* and *after* methods in CLOS. In this sense controllers are analogous to primary methods.

ADDS generalizes the methods for the construction of educational applications for foreign language text comprehension presented in [Fernández-Valmayor,99]. ADDS also generalizes the methods for the generation of hypermedia prototypes from XML documents describing the hypermedia contents and navigation presented in [Navarro,02][Navarro,03][Navarro, 04a][Navarro, 04b]. Works in [Sierra,00][Sierra,01][Sierra,03][Sierra,04] show the evolution of ADDS. In [Sierra,00][Sierra,01] the approach was called DTC (*structured Documents, document Transformations and software Components*). The use of this approach for the construction of applications in the transport networks domain (more precisely, subway networks) is described in [Sierra,00]. Work in [Navarro,00] explores its use in the educational hypermedia domain. Work in [Sierra,04] outlines the use of ADDS in the development of knowledge-based systems.

The origin of OADDS is in [Sierra,01]. In [Sierra,02] a first attempt to introduce semantic modularity mechanisms in the model is given. In [Sierra,03] a more specific OADDS formulation based on attribute grammars is presented.

7 Conclusions and Future Work

This paper introduces our document-oriented paradigm for the development of content-intensive, document-based applications. According to this paradigm, these applications are obtained by the automatic processing of the marked documents describing their main aspects. ADDS is an implementation of the paradigm based on

the authorship needs of the different participants. In ADDS, the DSMLs used to mark up the documents evolve according to such needs. This evolution is reflected, at the operational level, in the OADDS model for the incremental construction of modular processors. OADDS has been implemented as an object-oriented framework. In addition, the document-oriented paradigm itself can be applied in the development of OADDS processors. OADDSML is a markup language that allows the markup of documentation for the different operational components used in such development. Documented processors can be produced from the OADDSML documents.

The ADDS approach eases the development and maintenance of applications, because these can be described in the form of human readable and editable documents, understandable both by domain experts and by developers. Then, final running applications can be obtained by processing these documents with a suitable processor on the basis of their markup. Therefore, a DSML is an explicit characterization of a family of applications in a given domain, and each application can be executed from its documentation using the same processor for that DSML.

The approach also improves application portability, because well-known markup standards (e.g. XML [W3C]) can be used in the production of the documents. This improvement is especially critical for content-intensive applications, such as those arising in the educational domain, where the representation of the contents in portable and standard formats is critical in order to enlarge the applications‘ life cycles [Fernández-Manjón,97a][Fernández-Manjón,97b].

The evolutionary nature of the DSMLs provides the flexibility required by the development of complex applications. The DSML can be indeed extended when new markup needs are discovered. This also eases the use of DSMLs, because it avoids the inclusion of very general or sophisticated descriptive artifacts. Semantics modularity in OADDS helps to manage the operational impact produced by the evolution of these languages, allowing the reuse of processors when the processed languages are extended and/or reused to yield more complex DSMLs.

Finally, the recursive application of ADDS to its operationalization activity leads to the formulation of meta-DSMLs, like OADDSML, that simplify the development of OADDS processors.

Current work is oriented to the formulation of ways of devising DSMLs based on the documentation of the application domains, and in the markup of this documentation, using suitable schema languages. We are also working on the reformulation of different document processing models (tree oriented, event oriented, processing models of CSS and XSLT [W3C], attribute propagation models, etc.) in terms of OADDS. The next step includes the formulation of a type system for OADDSML, and its implementation using reflection mechanisms in the object-oriented implementation language for the operationalization framework. Finally, we are considering as future work the systematic design of different experiments for testing the feasibility of the document-oriented paradigm in different application domains and for comparing ADDS with other possible implementations of this paradigm.

Acknowledgements

The Spanish Committee of Science and Technology (TIC2001-1462 and TIC2002-04067-C03-02) has supported this work. We also thank the five anonymous referees for their useful comments to the previous versions of this paper.

References

[Abiteboul,00] Abiteboul,S.; Buneman,P.; Suciu,D. Data on the Web: From Relations to Semistructured Data and XML. Morgan Kaufmann. 2000

[Aho,86] Aho, A.; Sethi, R.; Ullman, J. D. Compilers: Principles, Techniques and Tools. Adisson-Wesley. 1986

[Birbeck,01] Birbeck,M et al. Professional XML 2nd Edition. WROX Press. 2001.

[Coombs,87] Coombs, J. H.; Renear, A. H.; DeRose, S. J. Markup Systems and the Future of Scholarly Text Processing. Communications of the ACM 30/11 (1987) 933-947.

[DSSSL,96] International Standards Organization. Document Style Semantics and Specification Language (DSSSL). ISO/IEC 10179. 1996.

[Duggan,00] Duggan, D. A Mixin-Based Semantic-Based Approach to Reusing Domain-Specific Programming Languages. ECOOP'2000. Cannes. France. June 12-16 2000

[Fernández-Manjón,97a] Fernández-Manjón,B.; Fernández-Valmayor,A. Improving World Wide Web Educational Uses Promoting Hypertext and Standard General Markup Languages. Education and Infformation Technologies 2(3). 193-206. 1997

[Fernández-Manjón,97b] Fernández-Manjón,B.; Fernández-Valmayor,A.; Navarro,A. Extending Web Educational Applications via SGML Structuring and Content-based Capabilities. In Wibe,J.; Verdejo,F (Eds). The Virtual Campus. Trends for Higher Education and Training. 244-259. Chapman-Hall. 1997

[Fernández-Valmayor,99] Fernández-Valmayor, A.; López Alonso, C. Sèrè A. Fernández-Manjón,B. Integrating an Interactive Learning Paradigm for Foreign Language Text Comprehension into a Flexible Hypermedia system. *IFIP WG3.2-WG3.6* Conf. Building University Electronic Educational Environments. Univ. California Irvine, California, USA August. 4-6 1999

[Frankel, 2003] Frankel, D.S. Model Driven Architecture: Applying MDA to Enterprise Computing. John Wiley & Sons. 2003.

[Fuchs,97] Fuchs, M. Domain Specific Languages for *ad hoc* Distributed Applications. First Conf. on Domain Specific Languages. USENIX. Sta. Barbara. CA. October 17-17. 1997

[Goldfarb,90] Goldfarb, C. F. The SGML Handbook. Oxford University Press. 1990

[Hudak,98] Hudak,P. Domain-Specific Languages. Handbook of Programming Languages V. III: Little Languages. And Tools. Macmillan Tech. Publishing. 1998

[HyTime,97] International Standards Organization. Hypermedia/Time-based Structuring Language (HyTime) – 2d Edition. ISO/IEC 10744 . 1997.

[Jackson,75] Jackson.M. Principles of Program Design. Prentice-Hall. 1975

[Johnson 75] Johnson,S.C. YACC-yet Another Compiler-Compiler. Computing Science Technical Report 32. AT&T Bell Laboratories. 1975

[Kastens, 92] Kastens,U.; Waite,W.M. Modularity and Reusability in Attribute Grammars. Tech. Report CS-613-92. University of Colorado. 1992

[Knuth, 68] Knuth,D.E. Semantics of Context-free Languages. Math. Systems Theory 2. 1968

[Knuth, 84] Knuth,D.E. Literate Programming. The Computer Journal 27(1). 97-111. 1984

[Lee,00] Lee,D.; Chu,W.W. Comparative Analysis of Six XML Schema Languages. ACM SIGMOD Record. 29(3). 2000

[Nakatani,97] Nakatani,L.H.; Jones,M. Jargons and Infocentrism. First ACM SIGPLAN Workshop on Domain-Specific Languages DSL'97. Paris, France. January 1997

[Nakatani,99] Nakatani,L.H.; Ardis,M.A.; Olsen,R.G.; Pontrelli,P.M. Jargons for Domain Engineering. 2º Conf. for Domain Specific Languages. USENIX. Austin. Texas. October 3-6. 1999

[Navarro,00] Navarro,A.; Sierra, JL.; Fernández-Manjón, B.; Fernández-Valmayor, A. XML-based Integration of Hypermedia Design and Component-Based Techniques in the Production of Educational Applications. In M. Ortega and J. Bravo (Eds). Computers and Education in the 21st Century. Kluwer Publisher. 2000

[Navarro,02] Navarro, A.; Fernández-Manjón, B.; Fernández-Valmayor, A.; Sierra, J.L.Formal-Driven Conceptualization and Prototyping of Hypermedia Applications. FASE 2002. Grenoble. France. April 8-12. 2002

[Navarro,03] Navarro, A.; Fernández-Manjón, B.; Fernández-Valmayor, A.; Sierra, J.L. An XML-based Approach to Fast Prototyping of Web Applications. Third Int. Conf. on Web Engineering ICWE 2003. Oviedo. July 14-18. 2003

[Navarro, 04a] Navarro, A.; Fernández-Valmayor, A.; Fernández-Manjón, B.; Sierra, J.L. Conceptualization prototyping and process of hypermedia applications. International Journal of Software Engineering and Knowledge Engineering. Special issue on Modeling and Development of Multimedia Systems In press

[Navarro, 04b] Navarro A.; Fernández B.; Fernández-Valmayor A.; Sierra J.L. The PlumbingXJ Approach for Fast Prototyping of Web Applications. Journal of Digital Information (JoDI). Special issue on Information Design Models and Processes 5 (2), http://jodi.ecs.soton.ac.uk/Articles/v05/i02/Navarro/

[Orr, 81] Orr,K. Structured Requirements Definitions. Ken Orr & Associates Inc. 1981

[Sierra,00] Sierra, J. L.; Fernández-Manjón, B.; Fernández-Valmayor, A.; Navarro, A. Integration of Markup Languages, Document Transformations and Software Components in the Development of Applications: the DTC Approach. Int. Conf. on Software ICS 2000. 16th IFIP World Comp. Congress. Beijing - China. August 21-25. 2000

[Sierra,01] Sierra, J. L.; Fernández-Valmayor, A.; Fernández-Manjón, B.; Navarro, A. Operationalizing Application Descriptions with DTC: Building Applications with Generalized Markup Technologies. 13th Int. Conf. on Software Engineering & Knowledge Engineering SEKE'01. Buenos Aires. Argentina. June 13-15. 2001

[Sierra,02] Sierra, J. L.; Fernández-Manjón, B.; Fernández-Valmayor, A.; Navarro, A. An Extensible and Modular Processing Model for Document Trees. Extreme Markup Languages 2002. Montreal. Canada. August 4-8. 2002

[Sierra,03] Sierra, J. L.; Fernández-Valmayor, A.; Fernández-Manjón, B.; Navarro, A. Building Applications with Domain-Specific Markup Languages: A Systematic Approach to the

Development of XML-based Software. Third Int. Conf. on Web Engineering ICWE 2003. Oviedo. July 14-18. 2003

[Sierra,04] Sierra, J. L.; Fernández-Manjón, B.; Fernández-Valmayor, A.; Navarro, A. A Document-Oriented Approach to the Development of Knowledge-Based Systems. In Conejo,R.; Urretavizcaya,M.; Pérez-de-la-Cruz,J.L.Current Topics in Artificial Intelligence.LNAI 2040. Springer-Verlag. 2004

[Steele,90] Steele JR, G.L. Common LISP: The Language (Second Edition). Digital Press. 1990

[Van Deursen,00] Van Deursen, A. Klint, P.Visser, J. Domain-Specific Languages: An Annotated Bibliography. ACM SIGPLAN Notices. 35(6). 2000.

[W3C] www.w3.org/TR

[Wadler,99] Wadler,P. The next 700 markup languages. Invited Talk of the 2º USENIX Conf. on Domain Specific Languages. USENIX. Austin. Texas. 1999

[Warnier,81] Warnier,J.D. Logical Construction of Systems. Van Nostrand Reinhold. 1981