Developing Content-Intensive Applications with XML Documents, Document Transformations and Software Components

José Luis Sierra, Alfredo Fernández-Valmayor, Baltasar Fernández-Manjón, Antonio Navarro
Dpto. Sistemas Informáticos y Programación
Fac. Informática. Universidad Complutense de Madrid.
28040,Madrid (Spain)
{jlsierra,alfredo,balta,anavarro}@sip.ucm.es

Abstract

*This paper describes DTC (*Documents, Transformations and Components*), our approach to the XML-based development of content-intensive applications. According to this approach, the contents of an application and other customizable features (e.g. the properties of its user interface) are represented in terms of XML documents. In DTC, the software of the application is organized in terms of reusable components capable of processing specific markup languages. In addition, we use document transformations to fit components and documents together, because they can be reused from pre-existing repositories. In this paper, we describe the DTC approach, illustrating its application in a case study. Because DTC encourages the explicit separation between the description of the application's variability (contents and other customizable features) and the application's operational support, the approach improves maintainability and reuse at both the information and software levels.*

Keywords: *Development Approach, Content-Intensive Applications, Descriptive Markup Languages, Software Components, XML*

1. Introduction

There are applications (e.g. e-learning systems) that integrate collections of highly structured documents regarding a given domain. Usually these documents are 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 kind of application *content – intensive*. The development of these applications can be facilitated with mechanisms capable of making the structure of the documents with contents explicit for people and machines. Generalized markup languages, such as SGML (*Standard Generalized Markup Language*) [9] or XML (*eXtensible Markup Language*) [23] provide these mechanisms. We have successfully used these technologies in the document-oriented development of content-intensive applications in several domains [5][13][14][16]. In all these domains we have described the contents and other customizable features of the applications by means of documents, we have marked up these documents with application-dependent, SGML or XML-based markup languages, and we have produced the applications by processing these documents.

Generalized markup languages allow the definition of a markup vocabulary and a set of grammatical rules to properly combine such vocabulary. Because it is possible to select the most suitable document grammar and vocabulary for each domain, the use of generalized markup languages avoids the rigidity that a single data model or encoding formalism imposes on the domain modeller. However, the markup languages defined only make it possible to describe how the information is structured. The use of a marked document for performing a particular task requires, in the end, the existence of an external program giving operational meaning to the markup language used to structure it. Building such a program can be a complex software development activity. Although the use of general purpose software and APIs, such as markup parsers and editors, can be helpful at lowering overall development complexity, it does not solve the most critical part of the problem: the construction of domain dependent semantics.

We think that an intelligent use of componentware technology can help to fill the existing gap between the syntax specification of a structured document and the desired operational semantics for the intended use of this document in the development of content-intensive

applications. We consider that the key idea is to devise components specialized in the interpretation of particular markup languages. In addition, such languages would not be committed to any specific problem domain. This practice would make components supporting such languages reusable for different purposes. In this way, we need mechanisms to bring different reusable software components together. In addition, if we reuse existing documents, we also need to integrate them in the overall application. Document transformations provide us with these mechanisms. We have put together these key ideas in our approach for developing XML-based content-intensive applications. We have called this approach DTC, from *Documents*, *Transformations* and *Components*. In this paper we describe our experiences with DTC: Section 2 outlines the DTC approach itself. Section 3 describes how DTC is applied in a case study. Section 4 describes some related work. Finally, section 5 outlines some conclusions and future work.

2. The DTC approach

Building an XML-based content-intensive application requires the provision of software for processing one or several classes of XML documents. Fig. 1a outlines a typical *full custom* structure for such applications [12][19]. The applications rely on the use of general-purpose parsing/generation frameworks [2] connected with application specific software. The main drawback of this organization is the coupling between this application-specific software and document structure, which hardly enables this software to be reused in different applications. In addition, the costs associated with its development from scratch are another important factor.

A different approach, which works for specialized uses such as XML-based web publication, enables pre-existing software to be reused, such as web servers and web browsers. The publication task is subsequently intended as a *transformation step* from the source XML documentation to a presentation format (in the case of web publication, HTML *–HyperText Markup Language*- [23]) for which processing software is already available (Fig. 1b). Consequently, development costs dramatically decrease. Unfortunately the feasibility of this approach strongly depends on being able to reformulate the task at hand as a publishing one. Of course it will be argued that the target presentation environment can always be extended with the loose processing capabilities (for instance, by means of scripting). But this again leads to the need to provide an important part of the domain-specific software from scratch.

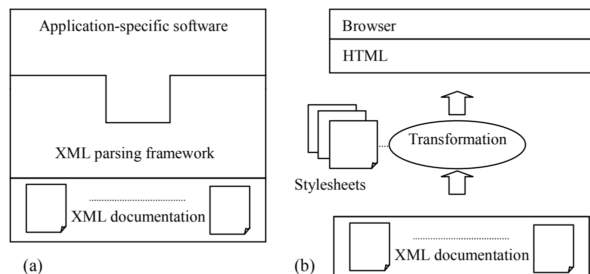


Fig. 1. (*a*) Typical full-custom structure for a XML-based content-intensive application. (*b*) Typical strategy followed in XML-based web publication.

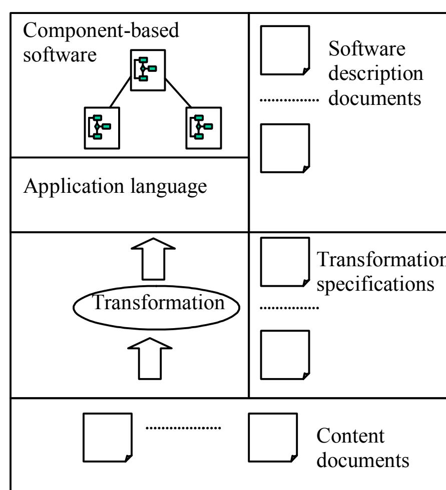


Fig. 2. Structure of a DTC application.

The DTC approach proposes an intermediate solution for developing XML-based content-intensive applications. Instead of recurrently building software from scratch, or, on the other side, trying to look for the universal language and browser, DTC suggests the use of software components specialized in processing specific classes of XML documents. In this way, each component is tightly associated with a markup language and can be understood as giving operational support for such a language. From this viewpoint, building a DTC application requires, on one hand, providing the content to be managed by the application in XML terms, and, on the other hand, properly combining a set of components for giving computational support to the final application. Such a process of combination leads to a component-based computational artifact able to process documents conforming to the languages associated with some of their components. In this way, having the application ready to be executed could also imply a

Draft version: See http://www.e-ucm.es/publications/articles.html for updated citation information

transformational step from the original XML documentation to the documents required by the components. Such transformations can, in turn, be described using suitable transformation specification documents (for instance, in terms of XSLT, *eXtensible Stylesheets Language Transformations* [23]). Finally, combination itself can make use of higher-level software components oriented to mastering other components. Such *combinator components* can also be parameterized in terms of suitable markup descriptions. Fig. 2 summarizes the general structure of a DTC application. The following subsections detail the most relevant features of the approach.

2.1. Content documents

In a DTC application the contents are structured in terms of marked documents that are jointly named *content documents*. Attending to their purpose, it is possible to distinguish between two different kinds of content documents. *Domain content documents* contain domain-specific information that can be reused across different applications (for instance, a dictionary, a botanical glossary, etc). On the other hand, there are *application-dependent content documents* with a clear meaning only inside a specific application (for instance, a document enclosing metrical information of a roadmap to be used in a graphical presentation). Application-dependent content documents are mainly oriented to *completing* the information provided by reusable domain content documents.

2.2. Application software, the application language and software description documents

The applications built according to DTC are component-based: they are built by means of the selection/construction, configuration and assembling of software components [20]. There are three main kinds of components in a DTC application: *markup interpreters*, *primitive facilities* and *combinators*.

Markup interpreters have a great relevance inside the DTC approach. They are oriented to processing a specific markup language. A good example of these interpreters is a component for processing weighted directed graphs described in terms of the XML DTD (*Document Type Definition*) of Fig. 3. This component can give several uses to the graphs represented in terms of the language defined by that DTD (e.g. searching the minimun-cost path, computing the minimun spanning tree, etc). Content processing mainly relies on the set of markup interpreters included in a DTC application. In this way, the markup languages associated with these components jointly define the *application language*: the language in which all the content processed by the application must be finally translated.

```
<!ELEMENT Graph (Arc|Node)*>
<!ELEMENT Arc EMPTY>
<!ATTLIST Arc
          Origin
           IDREF #REQUIRED
          Destination
           IDREF #REQUIRED
          Weight CDATA "1.0">
<!ELEMENT Node EMPTY>
<!ATTLIST Node
          id ID #REQUIRED>
```

Fig. 3. A markup language for representing weighted directed graphs. This DTD can be associated with a software component for processing DTD-conforming documents.

Primitive facilities are components that carry on basic functionality in the final application. For instance, basic GUI components (buttons, labels, menus, etc.) fit inside this category. Another class of primitive facilities is given by general XML processing components such as query or transformation engines that can help to assemble together different independently deployed, reusable components when they interchange information in XML terms. Finally notice that primitive facilities can have an associated markup language for enabling their configuration. Documentation required by such components has nothing to do with content processing. These documents, together with the other documents used for describing the structure or behaviour of the application software, are named *software description documents*.

Combinators make it possible to set up the way in which other components behave and interact. So, DTC does not commit itself to a pre-established combination strategy. Specific strategies are explicitly introduced by appropriate combinators. Some examples of combinators are typical *GUI containers*, such as those included in the AWT or Swing Java APIs. These combinators can come up with a markup language for configuring things such as look and feel, layout politics, etc. Another kind of combinators will be devoted to controlling the behaviour of simpler components. Good examples of these *controllers* are components supporting tailored scripting languages, control formalisms such as state machines or Petri nets, or event-driven component interconnection languages. Like primitive facilities, many of these components come with their associated configuration markup languages. In this way, their use in an application

Draft version: See http://www.e-ucm.es/publications/articles.html for updated citation information

requires the provision of the appropriate software description documents conforming these languages.

2.3. Putting it all together

Document transformations are specifications for deriving *result documents* from *source documents* (or, more precisely, from parse trees of source documents to parse trees of result documents [11]). Tree filter programming languages, such as XSLT or, in the SGML world, DSSSL (*Document Style Semantics and Specification Language*) [8] are normally used to specify such transformations. Because information produced and consumed by DTC software components is XML-structured, transformations enable the adaptation of these information flows between reusable components. In addition, transformations are used in DTC for mapping content documents into the application language.

When the software of a DTC application is built as described in the previous subsection, an application language is automatically induced. Such a language can be understood as the composition of all the languages for markup interpreters included in the application. Of course it could be possible to directly provide the application with documents written in this language. But doing so has several disadvantages: (i) it prevents reusing pre-existing domain documentation; (ii) application languages derived from those supported by reusable components could be difficult to understand for domain experts providing the contents; (iii) because each component can require different views of the same information, direct provision of contents in application language terms can lead to providing similar information multiple times; and (iv), application languages usually are *task oriented*; that is, information provided in terms of these languages will hardly be usable for any other purpose (it is the same reason why information is encouraged to be represented in tailored XML languages instead of being directly encoded in a task–oriented one, such as HTML). For all the reasons above, DTC encourages the separation of content and application languages, and to use document transformations for mapping the content of the application into the application language. Therefore, transformations can be thought of as *translations* from contents to application languages, such as the term is intended in the classical literature on language processors [1].

3. A case study: the subway application

In this section we present a case study for applying DTC in the development of a non–trivial application. The application provides an interactive graphic interface to find the best route between any two given stations in a subway network. We have instantiated the application in the subway network of Madrid (Spain).

```
<!ELEMENT SubwayNetwork
   (Stations,Corridors?,Lines)>
<!ELEMENT Lines (Line)+>
<!ELEMENT Line (Schedulers,Links)+>
<!ATTLIST Line id ID #REQUIRED>
<!ELEMENT Schedulers (Scheduler)+>
<!ELEMENT Scheduler EMPTY>
<!ATTLIST Scheduler
     StartTime CDATA #REQUIRED
     EndTime   CDATA #REQUIRED
     Frequency CDATA #REQUIRED>
<!ELEMENT Links (Link)+>
<!ELEMENT Link EMPTY>
<!ATTLIST Link
     OriginStation IDREF #REQUIRED
     DestinationStation IDREF #REQUIRED
     Distance CDATA #REQUIRED
     Speed CDATA #REQUIRED>
<!ELEMENT Corridors (Corridor)+>
<!ELEMENT Corridor EMPTY>
<!ATTLIST Corridor
          id ID #REQUIRED
          OriginStation  IDREF #REQUIRED
          DestinationStation
                         IDREF #REQUIRED
          TraversingTime CDATA #REQUIRED>
<!ELEMENT Stations (Station)+>
<!ELEMENT Station (Accesses,Tracks,Times) >
<!ATTLIST Station id ID #REQUIRED>
<!ELEMENT Accesses (Access)+ >
<!ELEMENT Access (#PCDATA) >
<!ATTLIST Access id ID #REQUIRED>
<!ELEMENT Tracks (Track)+ >
<!ELEMENT Track EMPTY >
<!ATTLIST Track
     id ID #REQUIRED
     Line IDREF #REQUIRED
     Direction IDREF #REQUIRED  >
<!ELEMENT Times (AscentTime |
                 DescentTime|
                 TransferTime)+ >
<!ELEMENT AscentTime EMPTY>
<!ATTLIST AscentTime
     Track  IDREF  #REQUIRED
     Access IDREF  #REQUIRED
     Time CDATA #REQUIRED>

<!ELEMENT DescentTime EMPTY>
<!ATTLIST DescentTime
       Access IDREF  #REQUIRED
       Track  IDREF  #REQUIRED
       Time   CDATA #REQUIRED>
<!ELEMENT TransferTime EMPTY>
<!ATTLIST TransferTime
       OriginTrack IDREF  #REQUIRED
       DestinationTrack  IDREF  #REQUIRED
       Time CDATA #REQUIRED>
```

Fig. 4. DTD for representing information about a subway network.

3.1. The application contents

The subway application includes: (i) a domain content document, marked according to the DTD of Fig. 4, with information about the subway structure (stations, corridors, accesses, subway lines, etc.) and timing (schedulers, trajectory times between different points of a station, average speed of each line, etc.), and (ii) an application- dependent content document with geometrical information for rendering the subway map. Notice that the direct provision of all this information can be a tedious work. This work can be avoided by building and using special–purpose editing tools for the required information. Section 4 will suggest how DTC can be extended for coping with these authoring activities.

3.2. Application software, the application language and software description documents

Application software is built using components for each one of the three categories introduced in subsection 2.2. In the first category, two markup interpreters are used. Firstly *Diagram*, giving support for a simple language that enables the description of 2D diagrams made of circles, straight line connections and text labels. Secondly *Graph*, giving operational support for a weighted directed graph description language, similar to that of Fig. 3. In the second category, primitive facilities, we use basic GUI facilities such as buttons and labels. In addition, we use a simple *Map* transformation engine for translating list of nodes (given in the language of the Graph component) into lists of stations. For managing these lists (in order to visualize them in the diagram representation of the subway map) we use a generic *XML processor* component, allowing the manipulation of documents in terms of their DOM (Document Object Model) trees [2]. Finally for the third category, combinators, we have used typical GUI containers, and for describing control, an *Automata* controller that gives support for a state–transition oriented formalism.
Notice that our application language is mainly given by the languages associated with the two markup interpreters. They support two different views of the subway network: a view as a graph, and a view as a diagram. Because a relation between these two views is required (for visualizing routes expressed as lists of nodes, in terms of the graph language), the map used for the *Map* transformation engine is also included in this language.

Finally, the software description documentation for the other components must be provided. To improve maintainability we group all this information in a single document (Fig. 5). From this document, individual software descriptions for each component are derived using simple query and transformation steps.

```
<subwayApplication>
  <mainWindow scalable="no"
              title=
             "subway route finder"/>
  <mainPanel background="pink">
    <row>
     <component>map</component>
     <component>lateralPanel
     </component>
    </row>
    <row>
     <component>controlLabel
     </component>
    </row>
  </mainPanel>
  ...
  <automata>
   <init state="init"/>
   <state id="init">
      <action>
        mainWindow>visualize();
       originLabel>changeText
                         (text = "");
       destinationLabel>changeText
                         (text = "");
       controlLabel>changeText
         (text =
           "Select origin station");
      </action>
    <transition state="selectingOrigen"/>
  </state>
  …
 </automata>
</subwayApplication>
```

Fig. 5. Part of the software description document for the subway application.

3.3. Putting it all together

Having the content documents and the computational support for the application, it only remains to put it all together. Thus we need to give the transformations from the content languages to the application one. A transformation enables the diagram view of the subway network to be generated. Such a transformation takes both the domain and the presentational documents as sources. A second transformation is used for generating the graph view. Fig. 6 shows a fragment of an XSLT filter for this transformation. Finally, another transformation is used to generate the document that encloses the relation between the two views.

Draft version: See http://www.e-ucm.es/publications/articles.html for updated citation information

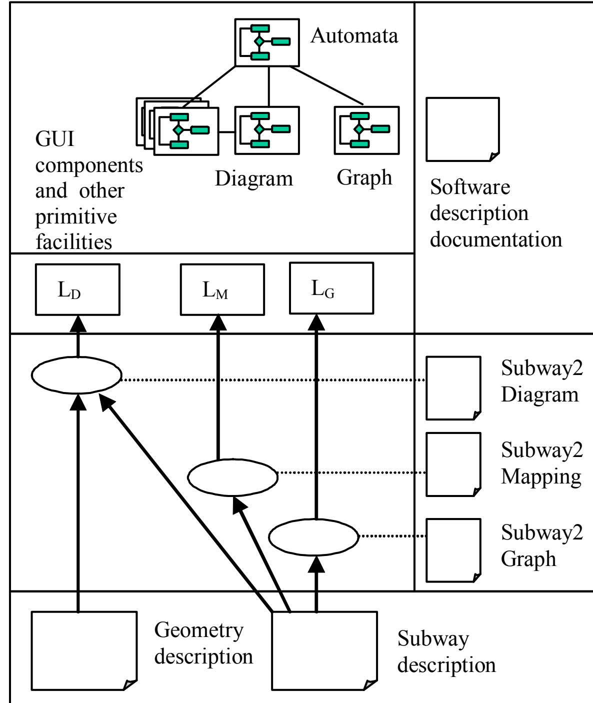
```
<xsl:template
    match="SubwayNetwork">
  <graph>
    <xsl:apply-templates/>
  </graph>
</xsl:template>
<xsl:template match=
          "Station | Track">
  <node id="{@id[1]}"/>
     <xsl:apply-templates/>
</xsl:template>
<xsl:template match="Access">
  <node id="{@id[1]}"/>
  <arc origin="{@id[1]}"
       destination="{../../@id[1]}"
       cost="0"/>
  <arc origin="{../../@id[1]}"
       destination="{@id[1]}"
       cost="0"/>
</xsl:template>
<xsl:template match="AscentTime">
  <arc origin="{@Track[1]}"
       destination="{@Access[1]}"
       cost="{@Time[1]}" />
</xsl:template>
<xsl:template match="DescentTime">
  <arc origin="{@Access[1]}"
       destination="{@Track[1]}"
       cost="{@Time[1]}" />
</xsl:template>
...
```

Fig. 6 Part of the XSLT specification document for transforming the subway description in a weighted graph

Fig. 7 sketches the structure of the final application. Fig. 8 shows a figure of the application itself.

4. Related work

DTC shares some features with the seminal work of Knuth on *literate programming* [10]. Literate programming enhances the comprehensibility of programs by identifying them with their documentation. In literate programming, a hypertext 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*). The ideas described in this paper differ from those of literate programming because in our approach only the high level aspects of the applications' variability, but not the code of the programs implementing these applications, are documented and marked up. Because of this, in our work, suitable markup languages are provided *for each* family of applications instead of using a fixed markup language, as in literate programming.



$L_D \equiv$ Diagram Language

$L_M \equiv$ Mapping Language

$L_G \equiv$ Graph Language

Fig. 7. DTC structure of the subway application.

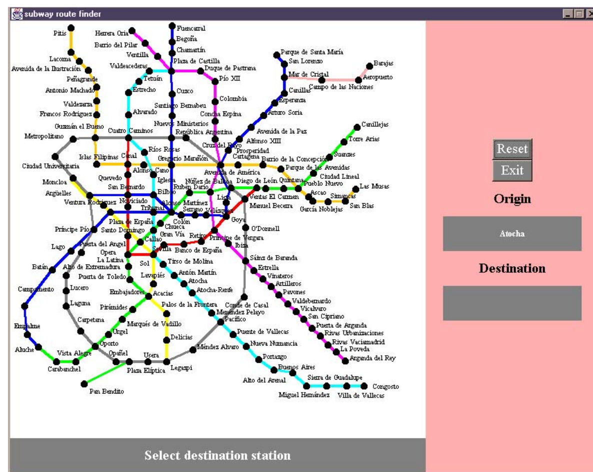


Fig. 8. Screenshot of the subway's route finder application built using the DTC approach.

HyTime [7], an SGML extension for the description of hypermedia applications, demonstrated that in some

domains, descriptive markup languages could be used for describing applications in terms of documents, and 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 of information interchange between applications and for many other uses. Nevertheless, the new application domains do not change the initial linguistic conception of generalized markup languages: in order to develop an SGML/XML application, the focus must be put on devising a markup language for describing the informational structure of the application domain. Once this language is available, one (or several) processor(s) must be provided, depending on the task to be solved using the marked up documents. This leads to the document-oriented spirit promoted by DTC.

Our work also shares many features with the approach to software development based on *Domain-Specific Languages* (DSLs [21]). Indeed, languages supported by the software components in a DTC application jointly define a DSL that can be further specialized in different types of contents using document transformations. A pioneering work in the application of SGML/XML for the definition of DSLs is [6]. In [22] relationships between markup languages and the DSL approach are highlighted. Although these works recognize the potential of markup metalanguages as a vehicle for defining DSLs, the stress is put on their use in formalizing abstract syntax, instead of their use as descriptive markup (meta) languages.

The document-oriented development of content-intensive applications promoted in this paper was formerly suggested in [3][4] as a vehicle to improve the production and maintenance of educational applications, and was consolidated after several experiences in the development of applications in different domains. The work in [5] provides information about the use of generalized markup technologies in the development of educational applications for the comprehension of texts written in a foreign language similar to that of the student's parent tongue. In [13][14] these ideas are used in the broader field of hypermedia domains. In [16] they are proposed for the development of knowledge-based systems. DTC was formerly proposed in [15]. In [17] a markup driven strategy to automatically assembly the components in the application software is described. In [18] a systematic approach to the formulation and the operationalization of the application languages is detailed. The main contributions of the present paper with respect to the previous works is to provide with a more complete characterization of the structure of the applications and a better taxonomy of the components used in the assembly of the application software. In addition a new and more precise characterization of the scope and the limits of the approach is given: the production and maintenance of content-intensive applications.

5. Conclusions and future work

The DTC approach improves the maintainability of content-intensive applications because of the explicit separation between content and computational machinery and because of the representation of information as human-readable and editable documents. This claim is based in our previous experiences in the document-oriented development of educational and hypermedia applications, as well as of knowledge-based systems. Indeed, many of the changes and updates in the application are at the document level with no programming effort. In addition, the DTC approach also takes advantage of component–based software construction modularity for easing update and maintenance. The DTC approach also encourages reusability at different levels. Domain content documents and DTDs can be reused for multiple purposes. Software components can also be reused in the construction of different applications. Finally, application software can be reused for building new applications in similar domains. Document transformations are used as the basic glue for enabling both reusable documentation and reusable software components to work together.

The most relevant shortcomings of the DTC approach, in its current state, are the complexity of efficiently managing the different sorts of information (domain, application and transformation specification documents, application software description, etc.) and the authoring of the application content documents. The complexity of the DTC process can be lowered with a suitable supporting tool. Currently we have developed a batch environment for doing all this work, but we plan to develop a graphic tool for supporting the DTC process. In order to improve DTC with authoring facilities, the same component-oriented and information and software separation ideas underlying the approach could be applied. Currently we are working on an extension of DTC oriented to the generation of domain-dependent document editors. The idea is to derive specialized editors from reusable DTC components (extended to support editing capabilities). Because such components must generate structured documents according to their supported languages, *inverse transformations* are needed for generating domain content documents from documents in the

application language. We refer to this approach as *inverse* DTC.

Acknowledgements

The Spanish Committee of Science and Technology (TIC2001-1462, TIN2004-08367-C02-02 and TIC2002-04067-C03-02) has supported this work.

References

[1] A. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley. 1986.

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

[3] B. Fernández-Manjón, A. Fernández-Valmayor, and A. Navarro. “Extending Web Educational Applications via SGML Structuring and Content-based Capabilities”. *IFIP Int. Conf. the Virtual Campus: Trends for Higher Education and Training*, Madrid, Spain, November 27-29, 1997.

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

[5] A. Fernández-Valmayor, C. López Alonso, A. Sèrè, and B. Fernández-Manjón. “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.

[6] M. Fuchs. “Domain Specific Languages for *ad hoc* Distributed Applications”. *First USENIX Conf. on Domain Specific Languages*, Sta. Barbara, CA, October 15, 1997.

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

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

[9] International Standards Organization. *Standard Generalized Markup Language (SGML)*. ISO/IEC IS 8879. 1986.

[10] D.E. Knuth. “Literate Programming”. *The Computer Journal* 27(2), 1984, pp. 97-111.

[11] E. Kuikka and M. Pentonnen. “Transformation of Structured Documents”. *Tech. Report* CS-95-46, University of Waterloo,1995.

[12] H. Maruyama, K. Tamura, and N. Uramoto. *XML and Java - Developing Web Applications*. Addison Wesley. 1999.

[13] A. Navarro, B. Fernández-Manjón, A. Fernández-Valmayor, and J. L. Sierra. “The PlumbingXJ Approach for Fast Prototyping of Web Applications”. *Journal of Digital Information (JoDI)* 5 (2), 2004.

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

[15] J. L. Sierra, B. Fernández-Manjón, A. Fernández-Valmayor, and A. Navarro. “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 Computer Congress*, Beijing – China, August 21-25, 2000.

[16] J. L. Sierra, B. Fernández-Manjón, A. Fernández-Valmayor, and A. Navarro. “A Document-Oriented Approach to the Development of Knowledge-Based Systems”. *Current Topics in Artificial Intelligence*, LNAI 2040, Springer-Verlag, 2004.

[17] J. L. Sierra, A. Fernández-Valmayor, B. Fernández-Manjón, and A Navarro. “Operationalizing Application Descriptions with DTC: Building Applications with Generalized Markup Technologies”. *SEKE'01*, Buenos Aires, Argentina, June 13-15, 2001.

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

[19] S. St.Laurent and E. Cerami. *Building XML Applications*. Osborne Mc Graw-Hill. 1999.

[20] C. Szyperski. *Component Software - Beyond Object-Oriented Programming.* Adisson Wesley. 1998.

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

[22] P. Wadler. “The next 700 markup languages”. *2º USENIX Conf. on Domain Specific Languages* (Invited Talk), Austin Texas, October 3-5, 1999.

[23] www.w3.org/ TR