Integration of Markup Languages, Document Transformations and Software Components in the Development of Applications: the DTC Approach

Jose-Luis Sierra, Baltasar Fernandez-Manjon, Alfredo Fernandez-Valmayor, Antonio Navarro
Dpto. Sistemas Informaticos y Programacion. Universidad Complutense de Madrid (UCM)
Avd. Complutense S/N 28040 Madrid-Spain
Tel: +34-91-3944654 FAX: +34-91-3944602
{jlsierra,balta,alfredo,anavarro}@sip.ucm.es

Abstract

This paper describes the DTC approach to the development of applications based on markup languages. DTC consistently combines *componentware* and markup technologies in a unified solution. Building an application according DTC supposes, on one hand, the provision of the set of documents describing the application at a purely declarative level (i.e. content, processes and interaction) and, on the other hand, the derivation of the application's computational machinery by assembling (reusable) software components. Each software component considered in DTC gives operational support and operational meaning to a type of documents. If necessary, documents describing the application can be integrated into the projected application's component structure using transformations. Transformations make possible to uncouple specific domain documents from reusable components. This explicit separation between the marked information, that describes the domain's application, and its computational support improves the maintainability of the applications promoting reuse at both information and software levels (documents and components).

Keywords

Generalized Markup Languages, Document Transformations, Reusable Software Components, XML, XSLT

1. Introduction

Generalized markup makes possible to syntactically define the structure of a document type. Because efforts are put on document structure instead of on a specific document processing, generalized markup allows reusing marked documents in multiple *markup applications* (programs doing real work with structured documents). This approach has been widely used and tested in the publishing domain [2][6][9] and we have extended this idea to the construction of more general software applications, especially hypermedia applications in the educational domain [4][5]. This work describes the DTC approach to the development of executable applications based on markup languages. DTC initials refer to *structured Documents*, *document Transformations* and *software Components*, the main technologies integrated in this approach. The idea underlying DTC can be outlined as follows. For building an application according to DTC, first of all, the different kinds of information (e.g. the domain information and the interaction with this information) to be used in the application must be provided as a collection of marked documents. In parallel the computational support for the application is built using software components. Each component is able to process a kind of documents giving them an operational meaning. The last step is to integrate the domain documents into the derived component structure. In a usual situation, components and application documents could not match perfectly (i.e. both could be potentially reused from pre-existing repositories). In this case DTC involves some adaptations for obtaining conformant documents. These adaptations are used to cope with the different structural and conceptual disagreements that can arise and those are achieved using document transformations. In this way, our approach promotes reuse at different levels (reuse of documents and documents schemas, reuse of the software components used to built applications, and even reuse of complete fragments of applications).

The structure of this paper is as follows. Section 2 briefly describes the different technologies involved in the DTC approach. Section 3 details the DTC approach itself. Section 4 analyses a non-trivial case study of a DTC application that provides an interactive graphical interface for information about the subway network of Madrid (Spain). Section 5 discusses the main advantages and shortcomings of the approach. Finally, section 6 gives some conclusions and outlines the future work.

2. Technologies integrated in the DTC approach.

This section introduces the different technologies integrated in the DTC approach: *generalized markup languages*, *software components*, and *document transformations*.

2.1. Generalized markup languages

Generalized markup languages [2] are devoted to describe the logical structure of documents. Usually this structure is understood as a hierarchical arrangement of *elements* with optional *attribute-value* pairs attached to them. This structure is formalized in terms of a document grammar or DTD (*document type definition*). The document structure is stated using a set of *tags* that, according to the grammatical rules of the DTD, draw the boundaries of the *elements* that are the actual content of the document. All this meta-information that represents the structure of a document is jointly named as *markup*. Two of the most popular generalized markup languages are SGML (*Standard Generalized Markup Language*) [6] [9] and XML (*Extensible Markup Language*) [19]. Now we use XML, but this approach could be easily extended to SGML.

Because generalized markup languages can be tailored for each domain of application by defining the appropriated DTD, they enable *denotability*. By *denotability* we understand the possibility to attach with the markup a consistent interpretation in the domain at hand. It supposes to be able of establishing a one to one correspondence between markup and information types in the domain of interest. Denotability is not feasible with a fixed markup repertory, because the potential existence of different domain referents that demand the same structure. The drawback for denotability is the need to explicitly define how to use the markup for each fixed markup repertory and for each application. Next subsections describe ways to cope with this problem.

2.2. Software components

In this work we propose the use of DTDs describing abstract markup languages attached with interpreters that give an operational meaning to documents conforming these DTDs. These interpreters support abstract uses of marked information. Because a non-trivial application can use several information sources, it would be possible to structure the computational parts of the application by means of an appropriated composition of several interpreters. This makes *componentware* technology [17] a suitable choice to give support for such interpreters. Each interpreter is a *software component* able to process information conforming a given DTD. These software components can execute a *set of actions*, are able to notify a *set of events* and, eventually, they can raise some *exceptional conditions*. In addition they support *integration* of documents conforming their DTDs, can act as *containers* of other components and they can offer a *visual interface*.

Components are completely documented. The

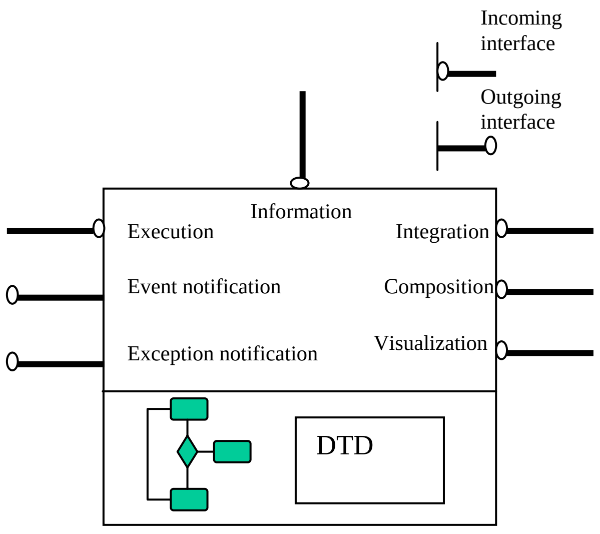

Figure 1. DTC component model that includes the DTD of the type of processed documents. The description of the services offered by the component is also done in XML.

description of the services offered by each component is done using XML (it can be accessed using the *information* interface). The DTD of the kind of documents processed by each component is also included. This description simplify the maintenance and reuse of components. Figure 1 outlines this component model (actual implementation of the model can be carried out by means of any well–known componentware technology or directly from scratch in any modern programming environment supporting dynamic code loading). Figure 2 shows the DTD included in a component for managing directed graphs.

The difference between the type of DTC components and generic XML or SGML processing tools relies mainly in semantics. Generic tools, such as markup parsers or markup editors are mainly focused on syntactical manipulation of marked documents, while components described here are focused on giving an operational meaning to a particular markup language. To enable component reusability, such languages and meanings must be necessarily abstract (for instance, directed graphs for the language given in Figure 2), but the use of document transformations (introduced in section 2.3) enables its specialization over particular domains.

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

Figure 2. A markup language for representing directed graphs. This DTD can be associated with a software component for processing DTD-conforming documents.

Components as those introduced here are used to build the component-based executable computational artifact for applications. Once this executable component structure is assembled, the application itself would be obtained processing the documents required by each component if available, or by directly deriving the documents required from each abstract DTD. However, as discussed in the next section, this approach has several shortcomings derived from possible disagreements between available documents and the specific documents required by components. Introduction of document transformations attempts to solve these shortcomings.

2.3. Document transformations

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 [14]). Result documents can vary both in structure and content from the original sources. Transformations are usually specified at the DTD level in order to allow their application to any document conforming that DTD.

A simple form of document transformation can be achieved using *enabling document architectures* (EDAs) [10][12]. An EDA is a set of *patterns* or *rules* that can be followed for writing markup applications. These patterns could be used, for instance, for representing lists of items, hyperlinks between chunks of information, etc, and their syntax is formally described in terms of DTDs (called *meta DTDs*). To apply these patterns when devising a DTD an architectural mapping from markup in the DTD to *markup forms* in the meta-DTD must be specified. These mappings basically rename elements into element forms and attributes into corresponding attribute forms. Following these mapping specifications, documents conforming the derived DTDs can be transformed into *architectural documents* conforming the appropriated meta DTD. The transformation process is carried out by means of *EDA processing*.

EDA's idea is useful for writing markup applications, because semantics can be associated with architectural markup and reused for each derived language. But the kind of transformations based on renaming introduced by EDAs could be insufficient for simultaneously supporting both documentation and software reuse. The reason is that each EDA imposes its own structural rules on the information and it would be very uncommon (and perhaps not even desirable) for pre-existing domain documentation to be structured in the terms required by the EDAs. Therefore more complex transformation processes are needed to cope with these potential disagreements.

The three main types of disagreements found when integrating pre-existing documents in applications made of reusable components are: a) *structural* (e.g. disagreements in the order or in the precedence of elements); b) *conceptual* (e.g. when the source document contains an `speed` and a `time` attribute and the result element requires a `space` attribute); and c) *incompleteness* (e.g. when you need additional presentational information for representing a graph). As discussed in [14], some kinds of structural disagreements can be solved applying simple syntax directed translators, while for conceptual disagreements could be better to use more powerful formal techniques (such as transducers based on attribute grammars) [1][13]. When differences between source and result documents are larger, or different documents need to be involved (as with incompleteness disagreements) manual methods based on tree filter programming languages would be the most pragmatic approach. In the SGML/XML world two of the well-known tree filter programming languages are the transformation language of DSSSL (*Document Style Semantics and Specification Language* for SGML) [11] and XSLT (*Extensible Stylesheet Language Transformations* for XML) [20]. In our experiments we have used XSLT for specifying tree filters for XML documents. Thus, the transformations are also specified by documents.

3. The DTC approach

Previous sections present the basic ingredients for formulating the DTC approach. Figure 3 outlines this approach. For building a generalized markup application according DTC the following activities must be carried out:

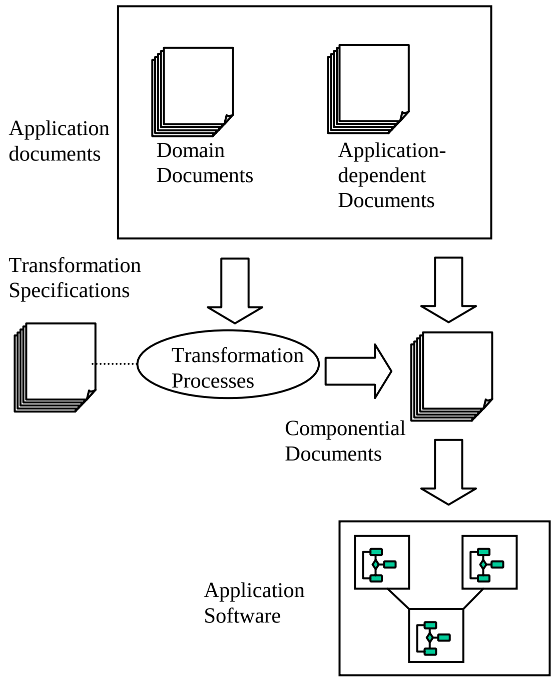

Figure 3. Schema of the DTC approach.

- Preparation of the set of basic documents that describe an application (the content, the processing and the interaction with the content information).
- Derivation of the component-based computational artifact (*application software*) for processing this information.
- Generation of the documents required by the software using document transformations as needed.

The next subsections detail each activity. Section 4 shows an example of how the DTC approach has been applied for building a non-trivial application.

3.1 Providing the application documents

The information describing a DTC application is structured in terms of marked documents. These documents are jointly named as *application documents*. Application documents can be roughly classified in the following categories:

- *Domain documents*. These documents contain domain specific information that could be reused across different applications (for instance, a dictionary, a botanical glossary, etc). Most of the reusable pre-existing documentation would lie in this category.
- *Application dependent documents*. These are documents with a low-level degree of reuse. Most of the documents in this category have a meaning only inside a specific application. Examples of this kind of documents are presentational information of a diagram, layout description of a GUI, etc. Here it is possible to distinguish two main types of application dependent documentation. One type is that of the documents oriented to solve incompleteness disagreements between domain documents and DTDs associated with the components used to build the application software. A second type is formed by those documents directly derived from the DTDs of some components.

3.2. Devising the application software

The application software is built by means of the configuration and assembling of software components following the component model described in section 2.2. These components can be selected from a library of reusable components or built from scratch. Here will be some components suitable for being reused in other developments and will be others specially designed to cover some specific application-dependent functionality. According to the introduced component model it is possible to classify components in several categories along several classification axes. A first classification allows us to distinguish between *primitive components* (the basic building blocks for the application construction) and *containers* (that allow the aggregation of a conglomerate of sub-components by means of the composition interface). Primitive components are subdivided, in their turn, into *markup interpreters* (devoted to give operational support for abstract DTDs) *primitive facilities* (components that carries out some basic functionality in the end application) and *mediators* (components devised to give support in the adaptation of information flows between components). Containers can be divided into *GUI containers* (oriented to display the visual representations of their sub-components) and *controllers* (oriented to describe the behavior of their set of sub-components).

3.3.Putting all together

Application software components demand documents in their supported languages. These documents are named as *componential documents*. Some componential documents can be derived from application documents using transformations. Transformations demand of suitable specifications that, in their turn, are enclosed in the corresponding specification documents (for instance, XSLT documents). Other application documents (those directly derived from DTDs attached with components) can be directly processed for the application software (so application and componential document sets can overlap).

In this way, the development of an application according DTC approach can be represented as a set of application documents linked to the application software by means of transformations. To obtain the executable application the componential documents must be generated by transformations, or taken directly from the application document set. Then the application can be executed by invoking the right action (or set of actions) over the components integrated in the application.

4. An example

In this section we present a case study of applying DTC in the development of a non–trivial application. The application provides an interactive graphical interface to find the minimum-cost path between any two given stations in a subway network. We have instantiated the application in the subway network of Madrid (Spain). We used Java as implementation technology, together with the Oracle Java Libraries for XML parsing and XSLT support [7] and the XAF engine (*XML Architectural Forms*) [8] both for developing our DTC prototype and for the DTC components involved in the case study.

4.1. The domain document

For building this application, a single domain document is considered. This document contains information about the structure of the subway network, schedulers, trajectory times between different points of a station, average speed in the different lines, etc. Figure 4 details the DTD used for representing this information. The actual document with this information for Madrid subway network fills around ten thousand lines of XML marked information.

```
<!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>
```

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

4.2. The application-dependent documents.

We need the following application dependent information:

- *Presentational information regarding the subway map.* Required information involves geometrical coordinates for each station, location of the names of such stations and a color for each line. Figure 5 shows the DTD used for structuring the document. Building such a document can be a tedious work. There are two ways of avoiding this work: renounce to a metrical presentation of the map (yet it is possible to generate a simpler graphical representation for each subway line) or build and use an special–purpose edition tool for generating the required information. Section 5 will suggest how DTC can be extended for coping with the second one.

```
<!ELEMENT presentationalInfo
           (station*,name*,line*)>
<!ELEMENT station EMPTY>
<!ATTLIST station
          id ID #REQUIRED
          x CDATA #REQUIRED
          y CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ATTLIST name
           x CDATA #REQUIRED
           y CDATA #REQUIRED>
<!ELEMENT line EMPTY>
<!ATTLIST line
           id ID #REQUIRED
           colour CDATA #REQUIRED>
```

Figure 5. DTD for documents containing presentational information required for the layout of the subway map.

- *Application control and interaction information*. This is the information needed for basic facilities, containers and control components. There should be a componential document associated with each occurrence of such components. To improve maintainability this information is put together in an interaction and control document. The needed componential documents can be generated from this document using EDA processing.

4.3. The software components and the application software

We have used eight prototypes of DTC components for devising the application software:

- Markup interpreters: *Diagram*, for supporting a simple language that enables the description of 2D diagrams made of circles, straight line connections and text labels, and *Graph*, that supports a language for describing weighted directed graphs, similar to that of Figure 2.
- Primitive facilities: *ButtonArragement* for setting collections of buttons, and *Label*, that makes possible to include static and dynamic text fragments in a GUI interface.
- A generic mediator: *XML manager*, for manipulating documents in terms of their DOM (*Document Object Model*) trees [18].
- GUI Containers: *Panel*, supporting a Java AWT bag layout–like layout mechanism, and *Window*, for working with windows with an AWT's card layout-like policy.
- A controller: *Automata*, allowing the description of application interaction and behaviour in terms of a state–transition oriented formalism.

Using these components, the application software for the subway application relies on the identification of the *component occurrences* to be used and to properly assemble such occurrences in a compositional structure (Figure 6).

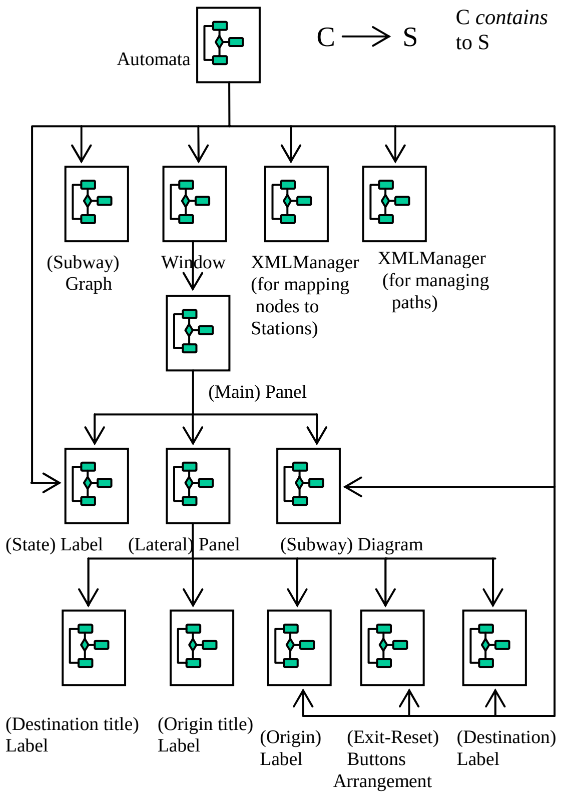

Figure 6. Compositional structure for the software of the subway route-finder application.

4. 4. Putting all together

Once available all the documents and the computational support for the application only rests to put all together. For doing it we need to perform the following transformations:

- A transformation for generating a description of the subway map in the language of the *Diagram* component. Such a transformation takes both the domain and the presentational documents as sources.

- A transformation for generating a graph representation of the subway network in the

```
<xsl:template match="SubwayNetwork">
  <graph>
    <xsl:apply-templates/>
  </graph>
</xsl:template>

<xsl:template match="Station">
  <node id="{@id[1]}"/>
     <xsl:apply-templates/>
</xsl:template>

<xsl:template match="Track">
  <node id="{@id[1]}"/>
</xsl:template>

<xsl:template match="Access">
  <node id="{@id[1]}"/>
</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>
...
```

Figure 7. Part of the XSLT specification document for transforming the subway description in a weighted graph.

language of the *Graph* component (Figure 7 shows a fragment of an XSLT filter for this transformation).

- A transformation for generating a document relating nodes in the graph representation with stations in the domain document. This document will serve as input for an occurrence of the *XML Manager* mediator that enables to relate nodes in the paths found by the *Graph* component with circles in the language of the *Diagram* component.
- A transformation (performed by EDA processing) for generating application control and interaction information from the interaction and control document.

Figure 8 outlines how all this information (application documents, application software description, transformations and architectural mappings) can be used to generate the final application (Figure 9).

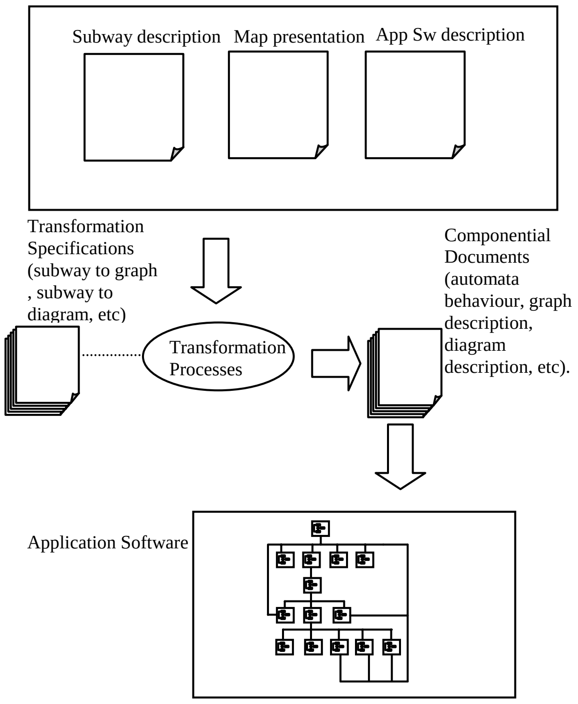
Subway description
Map presentation
App Sw description
Transformation Specifications (subway to graph , subway to diagram, etc)
Transformation Processes
Componential Documents (automata behaviour, graph description, diagram description, etc).
Application Software

Figure 8. DTC process for building the subway route finder application.

5. Discussion.

Previous section shows the viability of building applications according to the DTC approach. Assuming an appropriate support environment, the approach will improve the maintainability of applications, because the explicit separation between content and computational machinery and the representation of information in the form of human readable and editable documents [4][5].

Many of the changes and actualizations in the application will be at the document level with no programming effort. In our case study, changes in the aspect of the GUI is straightforward without coding (you only need to edit and modify the structured presentational information). In addition you can easily make more sophisticated changes. For instance, the introduction of a new subway line can be performed in a straightforward manner, changing the domain document, with zero programming effort. Or you could change the route search criteria with the appropriated reformulation of the transformation that produces the information attached with the *Graph* occurrence, and not more changes would be needed in the application.

DTC approach also take advantage of the component–based software construction modularity for easing update and maintenance. For instance, it would be easy to introduce different *Graph* occurrences for different time intervals, and, with the help of a *Clock* component, to generate a time-adaptable application: you only need to re-structure the components arrangement, to change the *Automata* description, and to write a transformation for generating the information required for each *Graph* occurrence.

As we have discussed, document transformations have a prominent role in reusability, because it is the *glue* that allows reusable domain documents and reusable software to fit together. Transformations as basic vehicles for enabling reuse are well-known both in software reusability [3] and in the construction of knowledge-based systems by means of reusable components [15] (there the role of transformations is played by *ontology translations* and *ontology mappings* [16]).

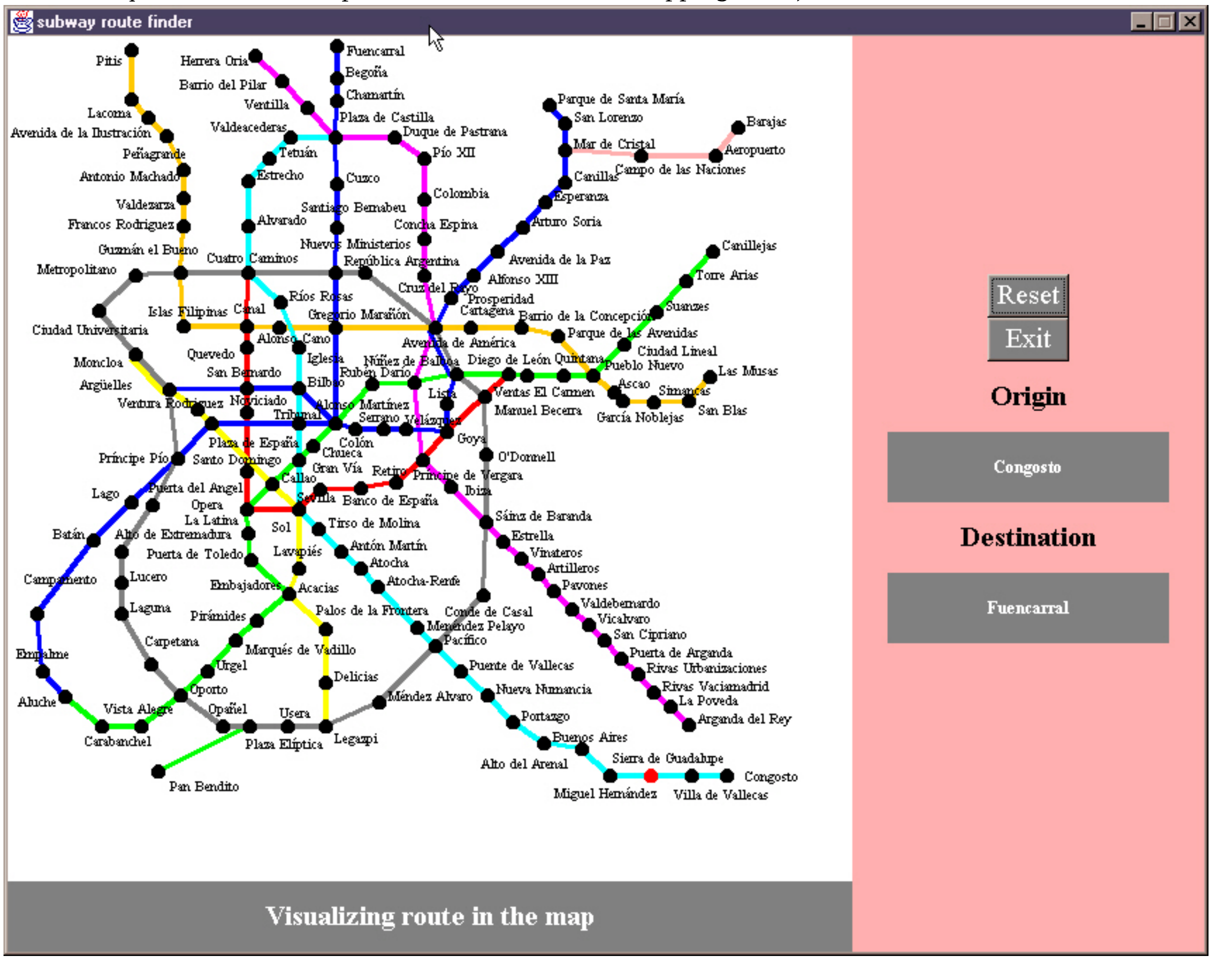

Figure 9. Screenshot of the subway's route finder application built using the DTC approach.

Moreover DTC approach encourages reusability at different levels. Domain 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 applications in similar domains (for instance, you could be thinking about reusing the subway application software in building an application for finding the best route in the roadmap of a country).

We also identify some shortcomings of the DTC approach, such as the complexity of managing efficiently the different sorts of information (domain, application and transformation specification documents, application software description, etc.), the static nature of the content documents, and the rather strong assumptions made respect to domain and application dependent information.
The complexity of the DTC process can be lowered with a suitable automation. Currently we have

developed a batch environment for doing all this work[1], but we plan to develop a graphical tool for supporting the DTC process.

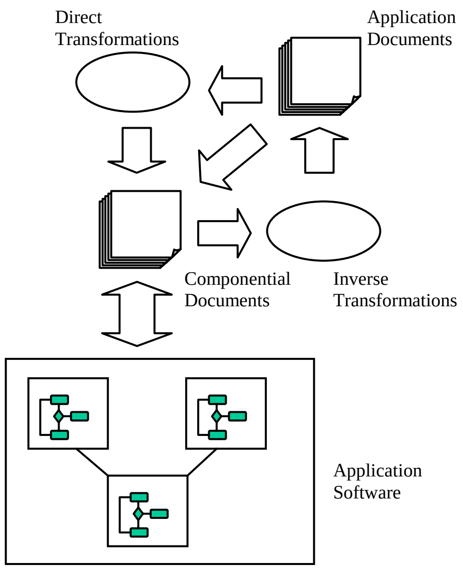

Figure 10. Schema of the extended (direct and inverse) DTC approach

Regarding the content documents, DTC assumes that these documents are given *prior to* the application execution. It is a serious constraint when you need to change this information as a consequence of the application execution. Fortunately this drawback could be easily solved either by generalizing the DTC component model adding a *reintegration* interface or by including special reintegration actions (in fact our *XML manager* mediator provides with one of such actions).

The shortcoming of domain and application information is more difficult to solve. Indeed, we think that many times (and it will be almost true in the beginning of the implantation of the DTC approach in an organization) the availability of suitable domain documents is rather unrealistic. Moreover, as we have illustrated in subsection 4.2, many times we will need application dependent documents which are complex and difficult to obtain. Depending of the application domain, use of standard structured document editing facilities can not be sufficient (as in the case of documents marked according to DTD of Figure 5). Because these considerations we think that for DTC being useful in the day-to-day software development practice it must be improved with authoring facilities. Fortunately we think that it can be easily achieved following the same component-oriented and information and software separation ideas underlying the approach described here. 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 their supported languages, *inverse transformations* are needed for generating domain documents from componential ones. We refer to this approach as the *inverse* DTC. The resulting extended (direct and inverse) DTC approach is outlined in Figure 10. Using the extended DTC approach you could devise an specialized editor for subway networks allowing to collect the information required by the DTDs of Figures 4 and 5, and then to specify an inverse transformation for generating the subway description document. Once made it you could apply the direct approach for mapping this information to the *Graph* component language or to reuse the domain document elsewhere.

6. Conclusions and future work.

We have presented the DTC approach for building software applications. DTC enables application maintainability and different levels of reuse. These goals are achieved integrating *componentware* and markup technologies in a unified framework. Reusable software components enable the derivation of specialized application software and the use of transformations facilitates the integration of domain documents within this software. Furthermore DTC can be easily extended for deriving specialized markup editors in order to lower the complexity of the document authoring.

The next steps in the project are to obtain a better characterization of the application domains where DTC is specially suited, and the development of a user-friendly DTC environment. As future work we are planning to perform a better classification of transformations inside the DTC framework, to make our component model more flexible and to refine our idea of *inverse* DTC.

Acknowledgements

The EU project Galatea (TM-LD-1995-1-FR89) and the Spanish Committee of Science and Technology (TIC97 2009-CE and TIC98-0733) have partially supported this work.

[1] In our prototype the DTC process is described in terms of a markup language that is processed by a DTC engine. Such an engine loads the needed components, makes the needed instantiations, transformations, document integrations and compositions and invokes the required actions.

References

1. Aho,V.A; Ullman, J.D.: “The Theory of Parsing, Compiling and Translation. Vols. I & II”. Prentice-Hall. 1972.
2. 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.
3. Feather,M.S.: “Reuse in the Context of a Transformation-Based Methodology”. In Biggerstaff, T.J.; Perlis,A.J.: “Software Reusability. Volume I: Concepts and Models”. ACM Press, 1989.
4. Fernandez-Manjon, B.; Navarro, A.; Cigarran, J.; Fernandez-Valmayor, A.: “Using Standard Mark-up in the Desing and Development of Web Educational Software”. In “Proceedings of Teleteaching 98”. Book Series of the Austrian Computer Society. 5th IFIP World Computer Congress. 1998.
5. Fernandez-Manjon, B.; Fernandez-Valmayor, A.: "Improving World Wide Web Educational Uses Promoting Hypertext and Standard General Markup Language Content-based Features". Education and Information Technologies, vol 2, no 3. 1997.
6. Goldfard, C.F.: “The SGML Handbook”. Oxford University Press 1990
7. http://technet.oracle.com/
8. http://www.megginson.com/XAF/
9. International Standards Organization: “Standard Generalized Markup Language (SGML)”. *ISO/IEC IS 8879*, 1986.
10. International Standards Organization: “Architectural Form Definition Requirements (AFDR)”. In “Hypermedia/Time-based Structuring Language (HyTime) – 2d Edition”. ISO/IEC 10744 . 1997.
11. International Standards Organization: “Document Style Semantics and Specification Language (DSSSL)”. ISO/IEC 10179. 1996.
12. Kimber, W. E.: "A Tutorial Introduction to SGML Architectures". ISOGEN International Corporation workpaper. 1997.
13. Knuth,D.: “Semantics of Contex-Free Languages”. Mathematical System Theory 2:2. 1971.
14. Kuikka, E.; Pentonnen, M.: “Transformation of Structured Documents”. Tech. Report CS-95-46. University of Waterloo. 1995.
15. Motta, E.: “Reusable Components for Knowledge Modelling”. IOS Press. 1999.
16. Park J.Y.; Gennari J.H.; Musen M.A.: “Mappings for Reuse in Knowledge-based Systems”. 11th Workshop on Knowledge Acquisition, Modelling and Management KAW 98. Banff, Canada, 1998.
17. Szyperski, C.: “Component Software: beyond Object-Oriented Programming”. Adisson Wesley. 1998.
18. W3C Candidate Recommendation.: “Document Object Model (DOM) Level 2 Specification Version 1.0”. 1999. http://www.w3.org/DOM
19. W3C Recommendation.: “Extensible Markup Language (XML) 1.0”. 1998. http://www.w3.org/XML/
20. W3C Recommendation.: “XSL Transformations (XSLT) Version 1.0”. 1999. http://www.w3.org/TR/xslt

Biography

- Mr. Jose-Luis Sierra is a Computer Science assistant professor at UCM. He is preparing a Ph.D. dissertation on the development of applications based on structured documents, document transformations and software components.
- Dr. Baltasar Fernandez-Manjon is a Computer Science professor at UCM. His research interests lie in the educational uses of computers, markup languages and user modelling.
- Dr. Alfredo Fernandez-Valmayor is a Computer Science professor at UCM and his research interest includes markup languages and its application to hypermedia and educational systems construction.
- Mr. Antonio Navarro is a Computer Science assistant professor at UCM. He is working in a Ph.D. dissertation centred on markup languages and hypermedia systems.