Operationalizing Application Descriptions in DTC: Building Applications with Generalized Markup Technologies [1]

J.L. Sierra, A. Fernández-Valmayor, B. Fernández-Manjón, A. Navarro
Dpto. Sistemas Informáticos y Programación, Facultad de Informática,
Universidad Complutense de Madrid, Avda. Complutense S/N, 28040 Madrid, Spain
{jlsierra, alfredo, balta, anavarro}@sip.ucm.es

Abstract: This paper describes the *operationalization process* (i.e. the step from application descriptions to executable applications) followed in DTC (structured Documents, document Transformations and software Components), an approach to develop applications using generalized markup technologies. DTC encourages the definition of XML-based *domain-specific languages* (DSLs) for describing each relevant aspect of the application. These DSLs are composed to obtain a single application DSL. Structured documents describing the application are the input for an operationalization process that yields a component-based artifact implementing the application. Operationalization process is performed in terms of a flexible architecture, where software components interact for assembling the application software in a collaborative, domain-dependent, way. Main benefits of our approach are software reuse and maintenance. These benefits are obtained through: a) the separation between high-level application description and application implementation and b) the provision of a flexible architecture, technologically neutral, enabling multiple implementation strategies.

Keywords: Content-based Applications, Application Development, Domain Specific Languages, Markup Technologies, Software Components, XML.

1 Introduction

There are application domains where the provision of the contents to be processed by the application is a critical part of the development process. We call this kind of applications *content-based applications*, because they process highly structured contents provided by domain experts, whose prior knowledge in software development is not guaranteed. This situation has been largely described in domains such as knowledge based systems [16], or, in our case, in the development of educational applications [2][3][9]. We consider that *domain-specific languages* (DSLs) [17] are crucial for the successful development of these content-based applications. A good definition for a DSL is that given in [1]: *a domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.* For content description, we are also interested in DSLs with more descriptive domain oriented meanings [4].

DTC (structured Documents, document Transformations and software Components) is our approach for building content-based applications using generalized markup technologies. Building an application according to DTC starts with the description of its most relevant aspects (e.g. contents, presentation, interaction) in terms of a collection of structured documents. Then, these documents are processed to obtain the executable application. The description of a DTC application is achieved by firstly selecting or devising suitable DSLs for each of these different application aspects. All these languages must be described with the same meta-language, so the same repertory of technologies can be used for their integration in a single description framework. DTC uses XML (eXtensible Markup Language) [19] as the common syntactic framework.

DTC encourages the explicit separation between content documents and documents involved with the description of the application's behavior. So a first distinction between *content* and *application* DSLs arises. These languages are intended for different users: content DSLs are used by domain experts, while application DSLs are used by software developers. The integration of both kinds of DSLs will be done by means of document transformations written by developers.

Application DSLs are composed for obtaining a single *application description* DSL. This language guides an *operationalization process*, so descriptions conforming it can be turned onto executable applications.

This paper describes the DTC approach focusing on the operationalization process. The structure is as follows. Section 2 outlines DTC. Section 3 describes the operationalization framework used for turning DTC descriptions onto executable applications made of discrete

[1] The EU project Galatea (TM-LD-1995-1-FR89) and the Spanish Committee of Science and Technology (TIC97 2009-CE,TIC98-0733 and TIC2000-0737-C03-01) have partially supported this work.

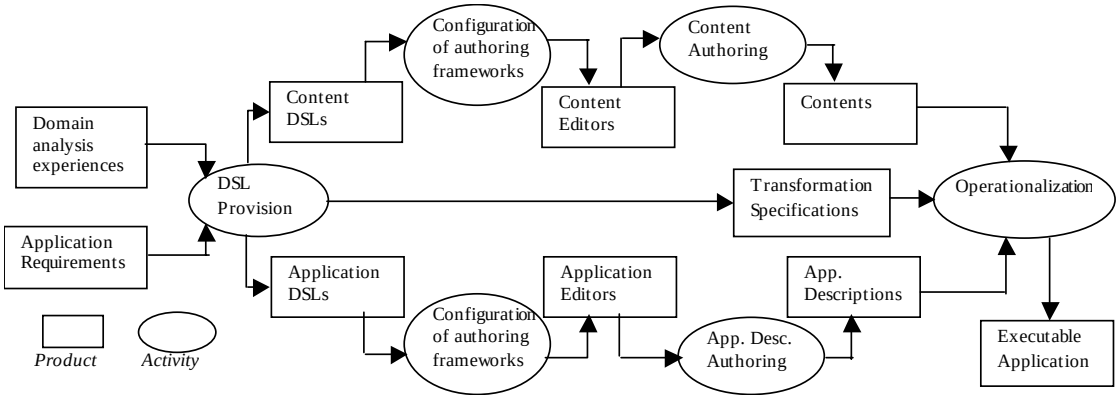

Fig. 1. Main products and activities involved on application development according DTC. From application requirements and domain analysis experiences, suitable DSLs are provided. These DSLs are devoted both for content description and for describing another application aspects. Transformations are also provided for mapping contents onto interpretation DSLs. Definition of suitable authoring syntaxes enables authoring frameworks to be configured. The results are editors tailored both for content and application description authoring. Content descriptions, transformation specifications and application descriptions are the inputs to an operational process producing the desired executable application.

software components instances. Section 4 gives an example. Section 5 discusses some related work. Finally, section 6 gives some conclusions and outlines future work.

2 The DTC approach

We have defined our approach DTC (structured Documents, document Transformations and software Components) [10][13][14] for building content-based applications. DTC makes an extensive use of the DSL principle. Building an application, according to DTC, can be largely viewed as the process of formulation, transformation, composition and operationalization of DSLs. At a very high level, DTC splits application development into two broad activities: one giving an explicit description of the application in a suitable DSL and a second, providing operational support for the DSL used to describe the application. In addition, when describing the application, DTC encourages an explicit separation between the description of contents and the description of the other application aspects.

Contents are described in terms of one or several DSLs, called *content DSLs*. Ideally, these DSLs are *use-neutral*, in the sense that they are involved with the descriptive and structural aspects of the information in the application domain, instead of dealing with how this information is going to be *used* or *processed* by the application. So, content DSLs can be formulated using a vocabulary close to the domain experts providing the contents. Therefore, descriptions conforming these content DSLs can be reused in multiple ways, either inside the same application or between different applications in the same domain area. However, for enabling particular uses of a given content, additional contextual or use dependent information will be required. For example, in a transport network application, we can have a DSL for describing the relevant data of the transport network domain (e.g. structure and timing). But, at this level it is not important the inclusion of additional data for describing presentational information (e.g. metrical coordinates of the connected places, fonts and colors) that will be necessary for visualizing the transport network. DTC contemplates the description of this kind of use enabling information, providing additional content DSLs, called *use dependent content DSLs* (this information could be provided by a different expert).

Once content description has been decided, the uses of this information in the application must be stated. This is achieved again by selecting DSLs for describing how contents must be interpreted inside the application. Because this fact, we call these DSLs *interpretation DSLs*. The link between content and interpretation DSLs is actually described with *transformations* written by developers. For instance, the search of routes in a transport network can be easily reformulated as a route search problem in a weighted directed graph. In this way, the DSL describing transport networks can be interpreted in terms of a language for describing weighted directed graphs. The actual interpretation is stated by writing a transformation from the transport network DSL onto the graph description DSL.

Other application aspects, not directly related with content interpretation, must be described in terms of additional *application DSLs*. These DSLs allow the description of other aspects, such as GUI structure and layout, user interaction, and control processing. Interpretation DSLs are considered just as a particular kind of application DSLs. Finally, an appropriate composition of the application DSLs results in a single *application description* DSL. Application itself is described in terms of this final composite DSL.

For defining DSLs in DTC we are currently using XML. In this way, each DSL is conceived as a XML based markup language. XML provides a common syntactic framework suitable for representing, in a structured way, all the information relevant for application's deployment. In addition, XML is accompanied with a set of complementary technologies that make easier a DSL-based application development with independence of particular,

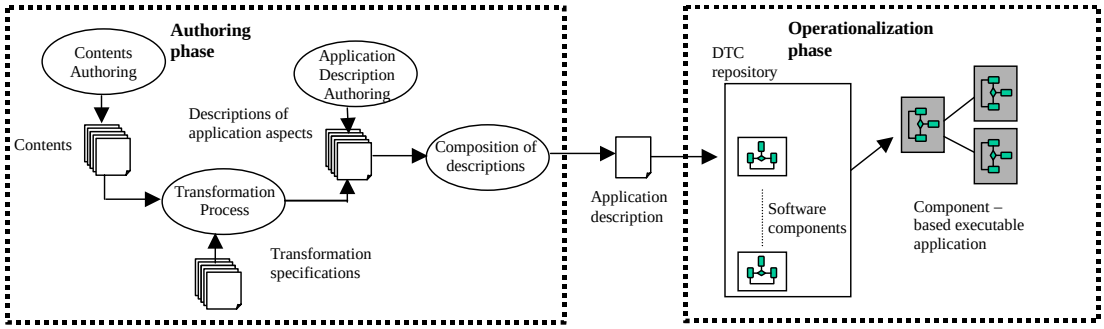

Fig.2. Schema of the building process for a particular DTC application instance.

vendor-oriented, implementation platforms. However, for some application domains, direct provision of XML structured documents could be inconvenient (e.g. providing all the description of a huge transport network directly in terms of raw XML). Thus DTC also introduces the concept of *document authoring*. This way, generic *authoring frameworks* (e.g. visual editors) can be configured by defining *authoring syntaxes* together with translation relationships between these syntaxes and the specific XML-based markup languages.

Finally, it is the *semantic* associated to interpretation and application DSLs. DTC is mainly concerned with *operational* semantics. In this way, each application DSL must be accompanied by a computational artifact (i.e. a component) giving one (or a set) of operational(s) meaning(s) to the information conforming this language. Regarding content DSLs, DTC does not prescribe a particular way for describing the associated semantics (i.e. it could be either given in an informal document or established using some suitable formal technique). Indeed, when transformations between content DSLs into interpretation ones are defined, operational semantics of the interpretation languages could be automatically attributed to the content ones.

Fig 1. outlines the main activities and products involved in the development of an application according DTC. In this work we are mainly interested on the process that makes DTC application descriptions operational. Next section describes our solution.

3 Construction of applications using DTC descriptions

Fig.2 sketches how a particular DTC application instance is built. In this process we identified two main phases:

- The *authoring* phase, where application descriptions are provided. Contents are authored and, then, they are transformed onto interpretations. Description of other application aspects relies on additional authoring. Finally, authored application descriptions and derived interpretations are composed in an overall application description.
- The *operationalization* phase, where the actual executable program is obtained from documents with the application description and a set of appropriate components. The ingredients in this phase are: a) the application description produced in the authoring phase, b) suitable *software components* giving operational support to the DSLs, and c) a *DTC repository* grouping all the components.

Next subsections go inside the details of the operationalization process.

3.1 The operationalization ingredients

The operational meaning of a DTC application is obtained by assigning computational artifacts to descriptions. Because DTC does not compromise itself with any particular DSL, this operationalization process must be necessarily open. In this way, the existence of a universal engine that, given any DTC application description, yields the associated artifact is clearly meaningless. On the other hand, DTC encourages the formulation of description languages by an appropriate combination of simpler languages, each one for the description of a particular application aspect. Thus, applications can be implemented using the components attached with those simpler languages. Hence, in DTC, language composition has an operational counterpart in composition of software components.

A more in-depth view of the language composition approach followed in DTC reveals the convenience of maintaining the semantics of some languages *parametric* in some of their aspects. For instance, a language for describing control and interaction with a stated-based formalism can be parametric in the repertory of control actions. Those aspects can, in their turn, be described by other DSLs. Actually, the ability for finding and using suitable description languages able to be parameterized is a key aspect for enabling the compositional approach encouraged by DTC. On the implementation side, the direct consequence of this approach is the distinction between two different categories of software components:

- *Atomic components.* Components that perform entirely their functionalities without the need of delegating further tasks in other artifacts. These components give support for non-parametric application DSLs. An example of this sort of components, in the application

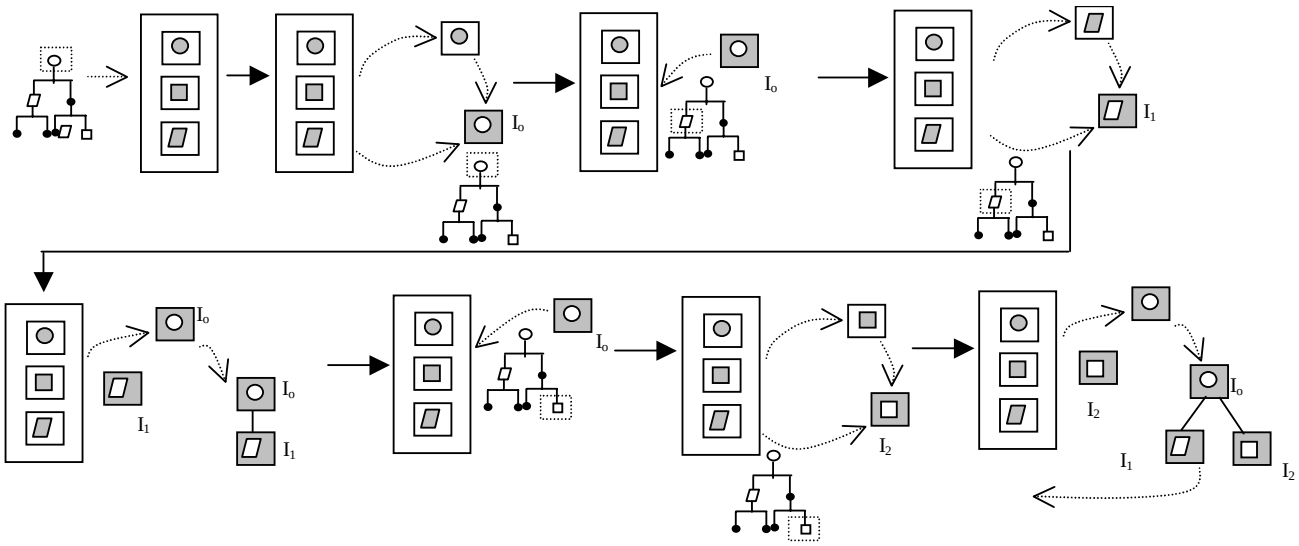

Fig. 3. Example of the DTC operationalization process behavior. Process starts with the DTC repository being queried with a context for the root of the description tree. Then the repository selects the right component for processing this context (that labeled by ○). The next step for the repository is to instantiate the component (let be it called I_o) and to delegate it the process of the context. After a bit of processing, I_o decides to query the repository for the processing of the context in ▱. The repository behaves as described above, selecting and instantiating the corresponding component (let be the new instance called I_1), and delegating the context processing in the resulting instance. I_1 eventually finishes the processing without further interaction, so control is returned to the repository that, in its turn, returns it to I_o (together I_1). I_o establishes some sort of relationship with I_1 and continues processing its contexts until deciding to query again the repository (this time with the context for □). This leads to the creation of a new instance (I_2) which is engaged on the processing of the context. I_2 carries out the task without further interaction, so the control is returned to the repository, and, from the repository to I_o that, after establishing the corresponding relationship with I_2, finishes the processing. The control is returned to the repository that returns I_o (and, consequently, the assembled artifact) as result of the operationalization process.

domain of finding routes in transport networks, is the component for finding paths in a weighted graph.

– *Combinators.* Components that require the help of simpler components for performing their tasks. These combinators are associated with parametric DSLs. For instance, a component associated with the state-based interaction language, or any of the *containers* found in GUI building frameworks are good examples for combinators.

Components work on the *abstract syntax* [15] of the associated DSLs. Because we are using XML as a common meta-language, we can get a common abstract syntax represented by the ordered attributed trees associated with XML documents. For managing these trees, a DOM (Document Object Model) [18] compliant interface can be used.

Once a suitable component repertory and a particular application description is available, components must be properly instantiated and their instances assembled to obtain the executable application. Again the consideration of an universal assembler is out of discussion, because assembling strongly depends on the particular semantics of each DSL. And the semantics of a DSL is confined to its associated component. In this way, the key idea is *to allow the components themselves to control the operationalization process*. So components must be equipped, on one hand, with the ability to process descriptions conforming their associated languages. On the other hand, they must be able to delegate the processing in other components when they reach description parts outside their scope. For doing so, they can use the DTC *repository*. Next subsection gives the details.

3.2 The operationalization process

Operationalization in DTC emerges as a collaborative process between the components associated with the DSLs. In this way, components work on the document tree associated with the single application description. For enabling this collaboration they use a common *DTC repository*.

The operationalization process starts by querying the DTC repository with an *initial context*. For a *context* we mean a reference to a node (the *context node*) in the tree instance of the single application DSL. The initial context node usually (but not necessarily) is the document element node.

Behavior of the DTC repository is very simple: when it receives a context, it firstly checks if a component was previously instantiated as the result of processing this context. If it is the case, the repository returns the resulting instance. If not, it queries the available components until it finds one *able* to process the context. To implement this behavior, components are equipped with an interface for deciding in which context they can be applied to. Concrete implementation of this interface is not relevant for the process, but it could be as simple as checking the tag and/or the *namespace* [20] associated with the context node, or as complex as involving a complete validation of the document structure rooted in this node. When the repository finds a component applicable in the context, it instantiates this component, records the instance as engaged

```
<!ENTITY % condition "(and| or | not | fired
                       | vareq | eventeq)">
<!ENTITY % action "(seq | if | do |
                    set | lit)">

<!ELEMENT automaton   (init,state*)>
<!ATTLIST automaton id  IDREF #REQUIRED>
<!ELEMENT init    EMPTY>
<!ATTLIST init state CDATA #REQUIRED>
<!ELEMENT state
   (%condition;?,%action;,transition?)+>
<!ATTLIST state id ID #REQUIRED>
<!ELEMENT transition EMPTY>
<!ATTLIST transition state IDREF #REQUIRED>
<!ELEMENT and (%condition;,%condition;)>
<!ELEMENT or (%condition;,%condition;)>
<!ELEMENT not (%condition;)>
<!ELEMENT fired  EMPTY>
<!ATTLIST fired event CDATA #REQUIRED
                   of CDATA #REQUIRED>
<!ELEMENT vareq EMPTY>
<!ATTLIST vareq var CDATA #REQUIRED
                 to CDATA #REQUIRED>
<!ELEMENT eventeq EMPTY>
<!ATTLIST eventeq  event CDATA #REQUIRED
                      of CDATA #REQUIRED
                      to CDATA #REQUIRED>
<!ELEMENT seq (%action;)*>
<!ELEMENT if
     (%condition;,%action;,%action;?)*>
<!ELEMENT do (with?,param*,result*)>
<!ATTLIST do action CDATA #REQUIRED
         in IDREF #IMPLIED>
<!ELEMENT with EMPTY>
<!ATTLIST with tag CDATA #REQUIRED>
<!ELEMENT param (%action;)>
<!ATTLIST param pname CDATA #REQUIRED>
<!ELEMENT result  EMPTY>
<!ATTLIST result pname CDATA #REQUIRED
               storeIn CDATA #REQUIRED>
<!ELEMENT set (%action;)>
<!ATTLIST set var CDATA #REQUIRED>
<!ELEMENT lit (#PCDATA)>
```

Fig. 4.Document grammar for `automaton`. Automata are described in terms of their states, their actions and their transitions. Each state have associated a set of (possible guarded) actions and transitions. In addition automata are allowed to use a finite number of variables for storing intermediate values. When the automaton enters a given state, it waits for a condition to be true. When it occurs, it executes the associated action and it either goes to the state given by the associated transition, if present, or, otherwise, stays in the current state. Conditions are described in terms of fired events, and variable and event values, and they can be composed using boolean operators. Basic actions are to set variables, to perform external actions or to exhibit literal values. They are composed in sequence and by using alternatives.

on the context processing, and delegates the processing to this instance[2].

When a component instance *A* receives a context it processes the context in a component dependent way. Normally this processing supposes for *A* to configure itself with the information in the subtree rooted on the context node. During tree traversal, *A* can decide to delegate the processing of another context. Indeed, this will be true for *combinators*, when descriptions associated with DSL parameterizations will be reached. Delegation supposes to query the repository with the new context in order to obtain an appropriate component instance, let it be called *B*. Once *B* is available, *A* can use this instance for their own purposes (e.g. for delegating in it some tasks when the application will be activated; in general we speak about a *relationship* being established between the two instances) and it can continue processing the original context if required.

In a normal situation, when the repository is queried with a context for an external entity, it returns as a result a component instance. In its turn, this instance can have established relationships with another instances, and so on, producing a component-based implementation of the application described in the context. For executing the application, the appropriate methods of the instances in this application can be invoked. Fig.3 graphically exemplifies the operational process.

The operational architecture enables a great variety of behaviors, apart from that described above. There, application description was completely processed for yielding a component aggregation implementing the description. We will call it as a *normal* behavior. Anyway, component instances are free to delay the process until needed. This leads to a kind of *lazy* operationalization process, where instances remember their associated application contexts and they interact with the repository during application execution. Another sort of more complex behaviors could also arise when new descriptions are generated *on the fly*, during application execution, and subsequently operationalized. Such dynamical generation could be due to some user interaction or to the inclusion of other dynamic information sources in the application implementation. Of course that all these kind of behaviors can be combined in a single application component repertory as needed.

4 An example

In this section we describe how a real DTC application has been produced using the processes described in the previous sections. The application provides an interactive graphical interface to find the best route between any two given stations in a subway network. This application was previously developed assembling software components manually, as reported in [14] (see sections 5 and 6 for differences between our current approach and previous work in DTC). Porting the application to the operationalization framework described in this work has been quite straightforward. We have made use of the namespace mechanism for avoiding name collisions

[2] Sort of things such as coping with the possibility of having multiple components applicable to the same context (for instance, in order to use this architecture for automatically indexing a huge repository of components) is out of scope in our approach. It is the responsibility of the application developer to ensure these ambiguities do not arise by a proper setting of the operationalization framework for a given application or application domain. In case of ambiguity, the first choice is taken. When the search for a component fails, the operationalization fails.

between different application languages when composing them into a single description[3] and we have adapted the components in the old implementation to the new operationalization schema, by equipping them with a selection function matching the document element and namespace for the supported languages. In the next subsections we give more details about this example of DTC application.

4.1 Content DSLs

We use a content DSL (`subway`) for describing information about the subway network. Such information involves structural aspects of the network, as well as timing information (schedules, access and transfer times in each station, average speed in each line, etc). Because we want to provide a graphical interface to the network, similar to the actual map facilitated by the subway company, we have a use-dependent content DSL (`subwayStyle`). This DSL enables us to code this additional information. The syntax of these DSLs is given in [14].

4.2 Interpretation DSLs and Transformations

We have used the following interpretation DSLs (some of them are described in [14]): (i) a language (`graph`) for describing directed weighted graphs, (ii) a language (`diagram`) for describing diagrams made of circles, labels and straight-line connections, (3) a language (`mapping`) for relating terminologies in the graph and the diagram languages.

The `mapping` language relates visual representations of stations with interpretations of such stations as nodes in the graph. Indeed, each station has associated a single circle in terms of the `diagram` language, but multiple nodes in terms of `graph`[4]. So, visualizing sequences of nodes in the visual representation requires knowledge about how nodes are associated with stations. Generally speaking, we think that this kind of *mapping* information is needed when independent aspects must be subsequently composed in a single application description.

In order to interpret contents in terms of these languages three transformation specifications are used: (i) a transformation for deriving `diagram` descriptions from descriptions in `subway` and `subwayStyle`, (ii) a transformation for deriving `graph` descriptions for `subway` ones, (iii) a transformation for deriving `mapping` descriptions for `subway` ones. All these transformations are specified using XSLT (eXtensible Stylesheet Language Transformations) [23].

4.2 Other application DSLs.

The other application aspects have been described using the DSLs for describing GUI elements (`window`, `panel`, `label` and `buttomArragement`) and a state transition based DSL (`automaton`) for describing interaction and control. These languages and their associated components have been slightly modified from a previous version to obtain a better conformance with the language composition-based approach described here. Fig. 4. shows the document grammar for the markup language associated with `automaton`[5].

```
<!ELEMENT subwayApp
  (behaviour,mainWindow,mainPane,
   lateralPane,subwayDiagram,controlLabel,
   controlButtoms,originTitleLabel,
   originLabel,destinationTitleLabel,
   destinationLabel,subwayGraph,
   graph2diagram)>
<!ELEMENT behaviour      (automaton)>
<!ELEMENT mainWindow     (window)>
<!ELEMENT mainPane       (panel)>
<!ELEMENT lateralPane    (panel)>
<!ELEMENT subwayDiagram  (diagram)>
<!ELEMENT controlLabel   (label)>
<!ELEMENT controlButtoms
            (buttomsArrangement)>
<!ELEMENT originTitleLabel (label)>
<!ELEMENT originLabel (label)>
<!ELEMENT destinationTitleLabel (label)>
<!ELEMENT destinationLabel (label)>
<!ELEMENT subwayGraph      (graph)>
<!ELEMENT graph2Diagram    (mapping)>
...
```

Fig.5. Top-level structures for the application description language. The omitted definitions directly correspond with the application DSLs described in subsections 4.2 and 4.3.

4.3 The application language.

The application language is obtained as a direct composition of the different application DSLs. Fig. 5 outlines the top-level structure of the resulting application language. Required relationships between the different languages are described by means of attributes. For instance, automata descriptions make references to external actions performed on other descriptions. These actions are described using two attributes in the `do` elements: `action` (for naming the action) and `in` (for referring the part of the description being the target of the action). For simplicity, the ID – IDREF XML basic linking mechanism is used, but this constrain could be easily removed by using a more complex query support (for instance, XPATH expressions [21]).

[3] For simplicity, we omit the details of namespace use in the subsequent descriptions. Indeed, we directly outline document grammars in terms of the simpler DTD formalism instead of using the more powerful, but more verbose, XML Schema [22].

[4] A station is structured, in its turn, in accesses, tracks, corridors, etc. These elements are not visually represented.

[5] Because the possibility for appropriate authoring syntaxes, we can exhibit some degree of syntactic verbosity in the description. We are currently working on a general approach for authoring this sort of descriptions by using visual grammars [5] for defining concrete syntaxes and by providing translators from editions conforming these grammars to XML documents.

Fig.6. Screenshot for the subway application instantiated in the subway of Madrid (Spain).

4.4 Operationalization

The operationalization process relies on components supporting each application DSL presented above. Because composition introduces additional vocabulary, it could seem that some sort of adaptation is required to cope with it[6]. Nevertheless, in this language all the new vocabulary is merely structural, and it can be skipped. It is done by querying the DTC repository with a context pointing the `automaton` description, instead one for the document element. Fig. 6. shows an snapshot of the resulting application for a `subway` description corresponding with the subway of Madrid (Spain).

5 Related work

The operational process described here substantially differs from previous work reported on DTC [10][13][14]. In those works operationalization was conceived as the manual provision and composition of the component structure for the application software. Therefore, descriptions were a consequence of operationalization. A static component-based structure were manually assembled for each application. Currently our approach is more description-language centered, being software structure a consequence from the composition of languages. The main advantages of the new way of operationalizing DTC descriptions are a clear distinction between the description and the implementation levels and a greater flexibility in the operationalization schemas.

DSLX [7], a framework for the operationalization of XML-based DSLs, and Jargons [8], an approach for defining DSLs by composing simpler ones, have strongly influenced our current work. The main difference between our operationalization schema and the DSLX architecture relies on composition. While in DSLX a *document processor* (an interpreter) would be provided for each DSL, our *interpreters* are automatically derived by assembling simpler artifacts: reusable software components. On the other hand, Jargons were mainly involved with composition criteria. So, operationalization in Jargons was conceived as the association of a chunk of code, written in an scripting language called Fit, with each *element type* involved in the DSL[7]. Our operational schema is more flexible, because *semantic* association is able to consider more context apart from the *tag* of a given element node, and because our collaborative schema enables a greater repertory of behaviors, either in operationalization and in execution stages.

XML *data binding* proposals [12] have also several common points with our operational approach. The key idea under data binding is to specify a mapping between a schema language and an object-oriented model. Once this mapping is available, each document can be *compiled* into language-specific object oriented representations. The main advantage is to enable the manipulation of documents in language-specific terms, instead of using general-purpose frameworks, such as DOM. Once these representations are available, application logic is provided to work on them. The main difference with our approach is that these mappings are associated with the schema language instead with particular document grammars.

Finally, several proposals have been made for processing XML documents using software component technologies [6][11]. The main difference with DTC is that DTC effort is put in describing applications to a higher level of abstraction, instead on giving alternatives to existing implementation technologies. Because the explicit distinction between description and implementation, concrete implementation could be included in DTC as required with little effort.

6 Conclusions and future work

This paper describes a flexible operationalization process in the development of content-based applications. The main benefit of this process is to provide a clear separation between application description and implementation technologies. This aspect is specially crucial in the development of content-based applications, where maintainability and portability are key factors for the final success of the project [9]. In previous works, application description were directly achieved by providing and assembling software components, each one able to process a type of *componential documents* conforming their supported markup languages. In this work application description is understood at a *linguistic level*, as the outcome of the composition of different application DSLs.

[6] We are currently working on a more in-depth identification of the kind of adaptation stuff required when composing DSLs for obtaining application description languages.

[7] Jargons actually does not make use of markup languages for defining DSL syntax, but they use a common syntax based on ground terms that can be directly translated into XML. Jargons associates an action which each functor type.

Once the description is available, operationalization becomes as a different, independent stage of the application development. This improves maintainability, because application maintenance is no longer involved with any sort of software arrangement, being instead performed at the higher abstraction level of the application description DSL. This also improves application portability, because descriptions are more independent from particular implementation technologies.

The operationalization process described here also enhances DTC operationalization flexibility. The implementation is driven by the application descriptions, instead of being manually programmed.

Currently we are working on a better characterization of how application DSLs can be composed into a single application description DSL, together with a method to parallel composition mechanisms at the implementation level. In the future we plan to apply DTC for developing web-based educational applications.

7 References

[1] Deusen,A.V.; Klint, P.; Visser,J. Domain-Specific Languages: An Annotated Bibliography. ACM SIGPLAN Notices 35(6). 2000.

[2] Fernández Manjón, B.; Fernández-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, pp. 193-206. 1997.

[3] 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* Conference Building University Electronic Educational Environments. University of California Irvine, California, USA August. 4-6 1999

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

[5] Marriott,K.; Meyer,B.; Wittenburg,K.B.A. Survey of Visual Language Specification and Recognition. Marriot,K.Meyer,B (eds). Visual Language Theory. Springer-Verlag. 1999.

[6] Martin,B. Creating Distributed Applications Using Xbeans. http://www.xbeans.org/. 2000

[7] Morrow,P.; Alexander,M. Domain Specific Languages – Tools for Better Programming. PCAI Magazine. Vol 13. Issue 1. Jan/Feb 1999.

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

[9] Navarro,A.; Fernández-Valmayor,A.; Fernández-Manjón,B.; Sierra,J.L. A Practical Methodology for the Development of Educational Hypermedias. ICEUT 2000 - 16th IFIP. World Computer Congress 2000. Beijing-China. August 21-25, 2000.

[10] 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. M. Ortega and J. Bravo (Eds) Computers and Education in the 21st Century: Invited papers from the Spanish Congress on Computers in Education (CONIED'99). Kluwer Publisher. 2000

[11] Pfeiffer, R.; Clack, A. XML Productivity Kit for Java - Programming Guide. IBM Development Center. 1999.

[12] Reinhold,M. A XML Data-binding Facility for the Java™ Plataform. Sun Microsystems. 1999

[13] Sierra, JL.; Fernández-Manjón, B.; Fernández-Valmayor,A.; Navarro, A. Developing Applications with XML Documents, Documents Transformations and Software Components. 10th International Conference on Computing and Information. ICCI-2000. Kuwait. November 18-21. 2000

[14] Sierra,JL.; 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. ICS 2000 - 16th IFIP World Computer Congress 2000. Beijing-China. August 21-25. 2000.

[15] Stoy,J.E. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press. 1977.

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

[17] Thibault,S. Domain-Specific Languages: Conception, Implementation and Application. Ph.D. Dissertation. Université de Rennes. 1998.

[18] W3C Proposed Recommendation. Document Object Model (DOM) Level 2 Specification Version 1.0. http://www.w3.org/DOM. 2000.

[19] W3C Recommendation. Extensible Markup Language (XML) 1.0. http://www.w3.org/ XML. 1998

[20] W3C Recommendation. Namespaces in XML. http://www.w3.org/ TR/REC-xml-names. 1999.

[21] W3C Recommendation. XML Path Language (XPath 1.0) Version 1.0. (Second edition) http://www.w3.org/ TR/xpath. 2000.

[22] W3C Proposed Recommendation. XML Schema (Parts 0,1,2) http://www.w3.org/XML/Schema.html. 2001.

[23] W3C Recommendation. XSL Transformations (XSLT) Version.1.0.http://www.w3.org/TR/xslt. 1999.