Applied Metamodelling to Collaborative Document Authoring

Anna Kocurova and Samia Oussena
School of Computing and Technology
University of West London
London, UK
Email: Anna.Kocurova,Samia.Oussena@uwl.ac.uk

Tony Clark
School of Engineering and Information Sciences
Middlesex University
London, UK
Email: T.N.Clark@mdx.ac.uk

***Abstract*—This document describes a domain specific language tailored for collaborative document authoring processes. The language can support communication between content management systems and user interfaces in web collaborative applications. It allows dynamic rendering of user interfaces based on a collaboration model specified by end users. The construction of the language is supported by a metamodel. We demonstrate the use of the proposed language by implementation of a simple document authoring system.**

***Index Terms*—collaborative authoring; multi-structured document; metamodelling; domain specific language.**

I. Introduction

Internet capabilities have been evolving at an amazing rate and Web provides a truly collaborative environment. Geographically distributed people can collaborate and work together on authoring tasks. Their cooperative effort is established by using web collaborative applications which manage coordination, communication, file sharing and document authoring. With increased user requirements and expectations, the applications become more sophisticated, providing rich functionalities and more powerful user interactions. Hence the architecture of collaborative applications is more complex. Often in web applications, the Model-View-Controller (MVC) software design pattern is incorporated and a three-layer architecture is used. The presentation, the functional processing and the data management are separated concepts.

Typically, the functional processing layer drives communication between the presentation and data management tiers. The presentation tier, as the topmost level of collaborative applications, displays information to users. It communicates with the functional processing level in which the business logic and application's functionality is controlled. Data and content are stored and managed in the data management tier. By using the MVC, the model containing domain objects is used to interact with the data management layer. The logic of the collaborative applications is included in the controller. The controller handles requests made in the presentation layer. The view generates output for the presentation layer.

Collaborative applications often allow its users to specify own work patterns and authoring processes. The applications need to have an ability to integrate customised workflow process models. The way how the models are designed, integrated and used depends on structuring of the particular collaborative application. We assume that using models can merge more functions of the functional processing layer. If the workflow models are enriched, they can serve as the domain models in the MVC pattern and also support dynamic building of user interfaces for each collaborative case. Therefore, a platform-independent and user-friendly approach that can help to develop such domain models is needed.

The Model Driven Approach (MDA) is a way that is capable to abstract away from implementation specific details. A Domain Specific Language (DSL) is the key enabling technology for the MDA. DSLs specify what should be executed and not how. In addition, they are more customisable to the particular context and domain. DSLs are languages with usually intuitive syntax and constructs, allowing solutions to be expressed in the problem domain. Moreover, by the employment of the DSL, all abstract constructs that are resistant to change can be captured.

Our aim in this paper is to propose a DSL targeted to collaborative processes and address the complexity of the development of web collaborative applications. Our focus is put on multi-structured document authoring, management and workflow. A notation that facilitates the construction of models in the language is proposed. The notation includes a number of graphical icons which represent the domain concepts and relationships between them. A metamodelling approach is used to define all constructs, the relationships that exist between constructs and well-formed rules that state how the constructs can be combined to create models.

This paper reports on our experiences of designing the DSL by applying the metamodelling approach and contributes to the domain-specific functionality of collaborative applications. The paper is structured as follows. Section 2 discusses related works. In Section 3, we outline a case study in which a collaborative authoring process is described. Based on the case study, a model for the authoring process is proposed and domain analysis conducted in Section 4. The model is used to drive communication between the business logic layer and user interface. Domain analysis is used to define the common domain constructs. The domain-specific features are visualised and expressed on a metamodel in section 5. The metamodel defines the basic semantics of the DSL. DSL

engine is described in Section 6. Finally, implementation details of a simple document authoring system are outlined in Section 7. Final discussion of the approach is provided in Section 8. Section 9 highlights our future work.

II. Background and Related Work

A. Document Authoring

The collaborative document authoring domain has been studied around for many years. The studies have focused on particular problems; none of them has demonstrated the metamodelling approach for capturing the domain concepts. Focus on collaborative activities by using the MDA has been applied in the work of [1]. The work presents generic modelling approach only for collaborative ubiquitous software architectures. Document composition and life-cycle have been addressed in work of [2] where a theoretical framework and practical guidelines for modelling composite document behaviour are proposed. Although, in [3] the MDA to define document management applications by using Eclipse Modelling Framework is described, the work focuses only on document management as a top layer on repository and does not consider other collaborative functionalities.

A framework for collaborative document procedures has been proposed in [4]. The work describes a run-time document workflow engine based on XML technologies. Although we have used some similar constructs and XML technologies in our prototype implementation, our work offers a code generic tool to build solutions at a higher level of abstraction.

B. Domain Specific Languages

The more the software products become robust and complex, the more sophisticated tools are required in software development. Domain-specific modelling (DSM) is a software engineering methodology for designing and developing software systems. DSM raises the level of abstraction beyond programming and enables the solution to be specified in a language that directly uses concepts and rules from a specific problem domain [5]. DSLs are possible tools to cope with the increasing user demands and the gap between IT and business concepts [6]. Domain Specific Language (DSL) describes the various abstract facets of a system. The need for new languages for various growing domains is strongly increasing [7]. The benefits of using DSLs to application development have been highlighted in [8].

The MDA aims to generate a formal specification and executable code from models. The models define the static and dynamic components of the system. Convergence of the MDA and DSL by using metamodel is described by [9], [10] where metamodel is promoted as the abstract syntax for a DSL that may be used in various situations. Metamodels capture the essential features of the application domain and describe the valid models permitted in the domain. Metamodels have been used in a wide variety of application domains, although often named differently such as 'data model' or 'language schema'. Using metamodels is required particularly in cases where a language needs to be defined precisely. For example, the UML specification contains one of the largest metamodels [1]. Other examples of metamodels can be found in process modelling [2] or in data warehousing [3].

III. Case Study

The work described in this paper is derived from the Rudiment project [http://samsa.tvu.ac.uk/rudiment]. Rudiment has been developed as a single document authoring system based on the following case study.

Academic researchers want to have their own environment for collaborative work because they collaborate on a number of multi-structured documents such as research proposals or research papers. Typically, sections or subsections of the documents are independently edited by certain participants with specific roles. Collaborators need to regularly access, retrieve and edit the working sections of the documents using a prescribed set of rules. Their complex processes of document production and management need to be customised by their internal rules for collaboration. An agreed work pattern (workflow) can enhance productivity of a team. They expect to have a user-friendly tool that would enable them to design own multi-structured document authoring processes and specify actions for particular roles. Their authoring environment should provide an appropriate user interface for each process based on user role.

IV. Domain Model

In this section, a collaborative authoring model for case study is illustrated. Domain analysis is used to define all common domain-specific constructs.

A. Collaborative Authoring Model

A research proposal authoring process is used as test case. The process is expressed as a model. This model has been simplified for the purpose of the paper. The structure of a research proposal funding bid and a lifecycle of the bidding process are illustrated in Fig. 1.

The *Project Bid* as a main document contains data field descriptions in form of properties such as *title*, *author* and *checked*. Each property has assigned a property type. The document is multi-structured, thus composed from a number of sections. In this case, *MainText* and *Budget* are shown. The *Project Bid* document and the *Budget* section have their own lifecycles and conform to own work patterns of completion. The states of elements are represented by the oval graphical symbols and are associated with control boxes. A control box contains a set of possible run-time actions. The actions can be performed by specified roles. For example, if the *Budget* section is in the *Prepare* state, the section can be further processed only by people with the role of *Manager* or *Anyone*. The role *Anyone* in this case means that this action does not depend on a role and any user belonging to the team of collaborators can perform it. While a person with the role

[1]UML 2.3, http://www.omg.org/spec/UML/2.3/
[2]SPEM 2.0, http://www.omg.org/spec/SPEM/2.0/
[3]CWM, http://www.cwmforum.org/

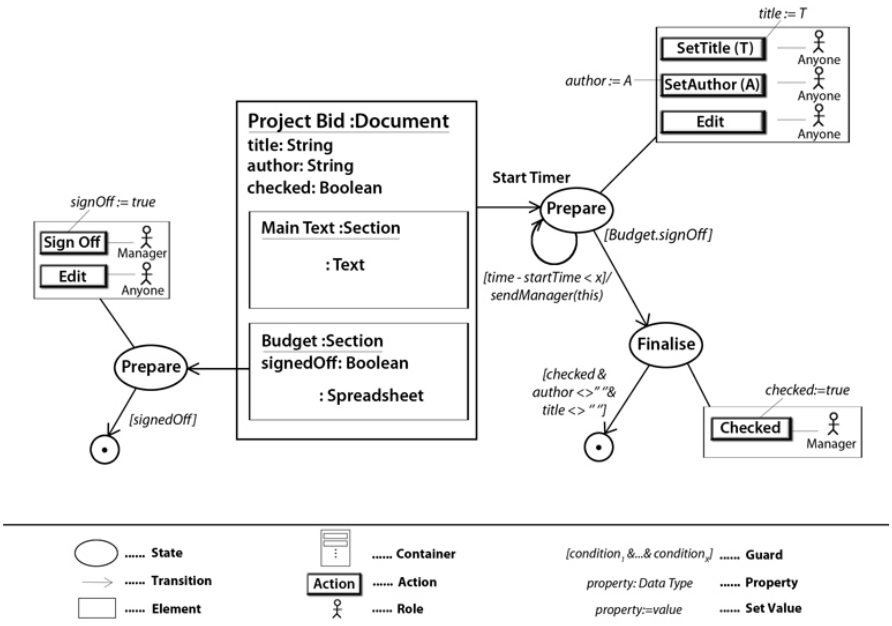

Fig. 1. Model for collaborative document authoring

of *Manager* can *Sign off* the budget, other collaborators are allowed only *Edit* it. The actions might change values of attributes of the corresponding element and trigger certain transitions between states either of the element itself or its parent element. For instance, when a user with a role of *Manager* signs off the *Budget*, the *Project Bid* state is changed from *Prepare* to *Finalise*.

B. Domain Features

The graphical representation of the authoring process illustrated in Fig. 1 expresses the authoring process in an understandable and intuitive form. It uses a mix of textual and graphical icons to visualise the document hierarchy and the collaborative workflow at run-time. The model can be supplied to the collaborative applications which will drive the collaboration at runtime according to this model. Based on the case, we have defined the following set of features that the DSL must support.

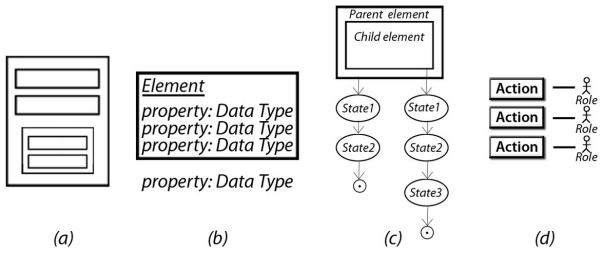

Fig. 2. Domain Features

Element Containership: One of the basic qualities of a document is its structure. Document structure is always a key to its management. Structure encompasses well-defined manageable subelements of a document and their relationships to each other. The *Container* and *Element* widgets in the model scheme can be used to build a multi-level hierarchy of a document. The *Container* widget can contain other containers or atomic elements (Fig. 2a).

Properties: To make sure that elements are accessible, identifiable and discoverable, elements need to be specified by data called metadata or properties. The structural and descriptive metadata describes properties relevant to an element and are associated with the corresponding *Container* or *Element* widget (Fig. 2b). The control metadata is used to guide transitions between two states.

Workflow: Workflow defines a series of connected steps that must be accomplished to produce a required output. Workflow management is an important part of content management. In order to put a document in the center of the workflow process, content-oriented workflow is used. Each workflow step alters the document or its element in a certain, predesigned way. Transitions are driven by guards. Workflow can be designed for any parent or child element as demonstrated in Fig. 2c. The flow of the document is controlled by the values of control metadata which values might be changed as results of certain actions.

Document authoring: Collaboration allows a large number of people to contribute and share documents. To facilitate a smooth document authoring process and controlled access to data, a set of roles needs to be defined. A role is associated with a set of permissions and obligations. Roles are occupied by actors. Each workflow state of a particular document element is accompanied by a set of actions performable by certain roles (Fig. 2d).

V. DSL Features

In this section, the domain concepts and their relationships are progressively defined. The information is visualised on a metamodel (Fig. 3). The metamodel is the central asset that defines semantics of the DSL.

A. Document Structure

A system manages a collection of documents. A document is a sequence of sections, sub-sections etc. The structure of elements is a tree where the leaves of the tree are blocks of text, diagrams, etc. Specific types of elements (e.g. projects, documents, sections, etc.) are specializations of Container. There may be many different types of atomic element (only Text is shown).

All elements must be specialized to fit individual collaborative document needs. For example, the structure of a research proposal may be different from the structure of a final project report. Specialization must support both, structure and behaviour. Variability of structure is achieved by allowing elements to be arbitrary sized trees marked up with any number of properties. A *property* is just a name-value association. In our example, the Project Bid is a document composed of two sections: Main Text and Budget. The Project Bid has various properties such as *title*, *author* and *checked* and each section may have some additional own properties such as the Budget section has assigned the *signedOff* property.

B. Roles and Actions

Interaction with all elements is via actions. An *action* can create or delete a project element. An action may modify an element by changing the text or modifying a property value. We have identified the following basic action types: create; delete; modify-text; modify-property. Actions are performed by actors, i.e. project members who have specific roles. For

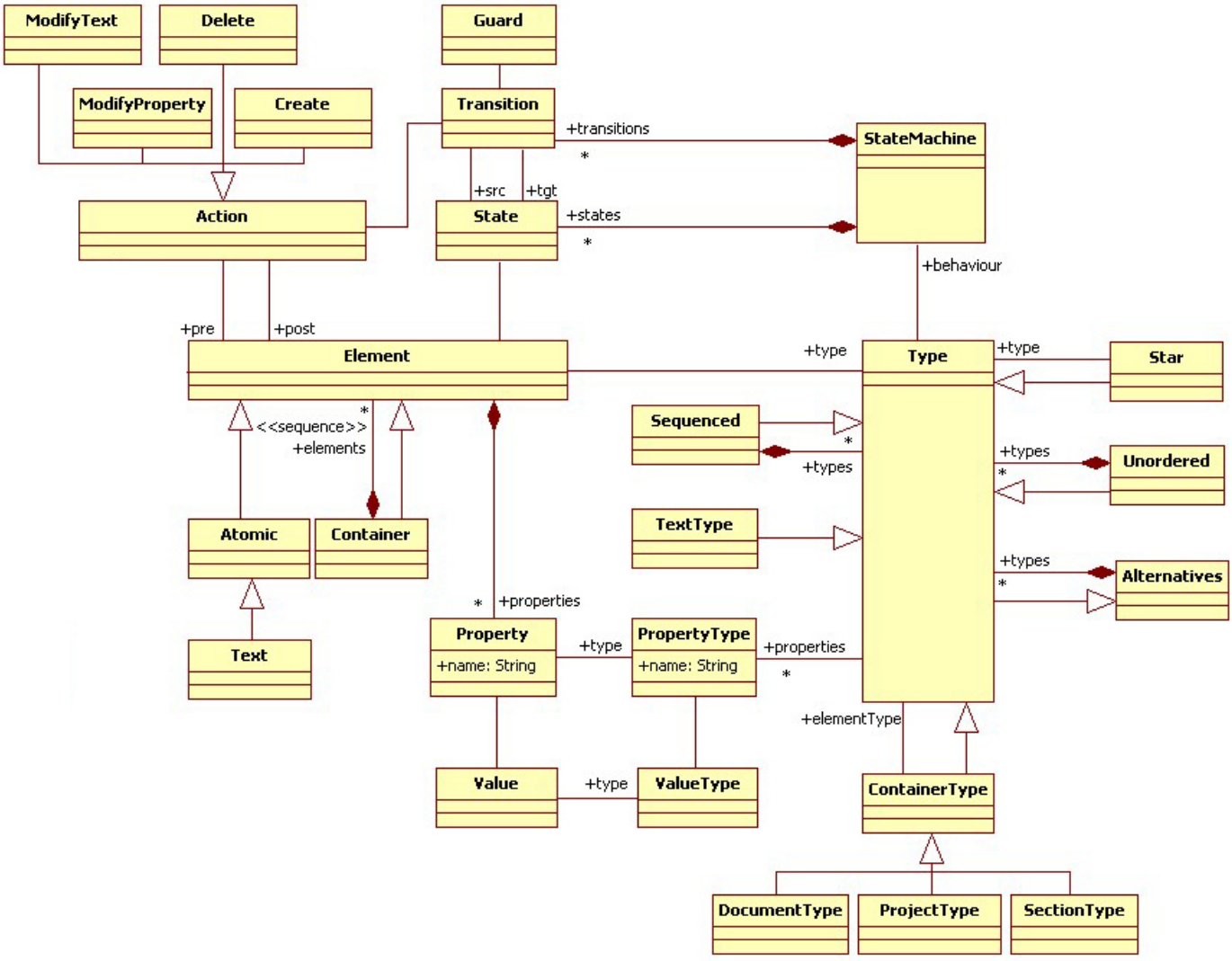

Fig. 3. Metamodel

example, a project member may create a new document or delete a section of an existing document. A project must define, a-priori, a collection of role types.

For instance, only *Manager* can *Sign off* the Budget section but *Anyone* is allowed to *Edit* it.

C. States

At any given instant of time, an *element* is in a specific *state* that is defined by its property values, the states of its component elements (if any) and the text it contains. The life-cycle of an element describes a sequence of states that it has occupied and the actions that have occurred to cause changes of the states. For example, the Budget section might be in its *Initial* state or the *Prepare* state.

D. Transitions

A *transition* between two states of the same element corresponds to an *action* that has occurred. The result is an important change in state. Each transition is associated with a source and target state.

The transitions and states are operated by a state machine. Guard for a transition can be indicated. It contains conditions that must be true for the transition to be triggered. In Rudiment, the Budget goes from the *Initial* state to the *Final* state through the *Prepare* state. The condition, [signedOff], is a guard for the latter transition.

E. Element Types

Each element in Rudiment has an own type. For instance, a document would be an element of DocumentType. A structure of elements is based on the structure of their types. A *type* is a specialization of *ContainerType*, therefore it may represent a collection of other types that can be sequenced or unordered. The Star class, an arbitrary sized sequence of elements, points out on the possible multiplicity of a particular type within a container. Elements have properties, element types have property types. This hierarchy of types at the metamodel level enables to model various templates.

F. Workflow

A project has a plan that is described in terms of a collection of (partially ordered) tasks that must be performed by actors in designated roles. Each task is a specification of an element state within the project and is therefore achieved by performing a (possibly composite) activity. A plan may be associated with each component element of a project or may just be associated with the top-level element (the project container). At any given time it is possible to check what the outstanding tasks of an element are. There will be conditions expressed in terms of the element properties and subcomponents that must be satisfied before the element may change from one state to another. Workflow in the collaborative bidding process is essential in order to improve efficiency and reliability of the process. The event is usually an action triggered by an execution condition, for example by clicking a button. The event is accompanied by pre- and post-conditions such as a new section can added to document only if the document has been created and allows further structuring. The workflow at the metamodel level is depicted in Fig. 4. The semantics are shown in Figure 1 where the consequence of states is modelled and each state is accompanied by a set of available actions. Explaining the conditions for each action in details is out of scope of this paper.

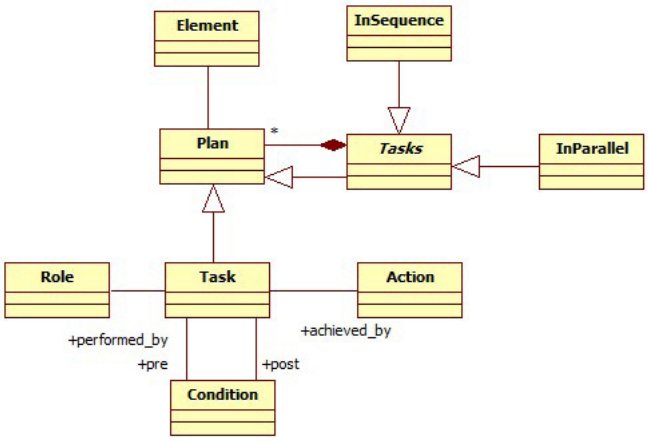

Fig. 4. Workflow

VI. DSL Engine

The language proposed in this paper is a tool that can support the design of collaborative processes and dynamic building of user interfaces. The aim of this section is to describe an overall architecture of a collaborative system and applicability of the DSL. As shown in Figure 5, the system is composed from three layers: content management system (CMS), DSL specific engine and user interface (UI).

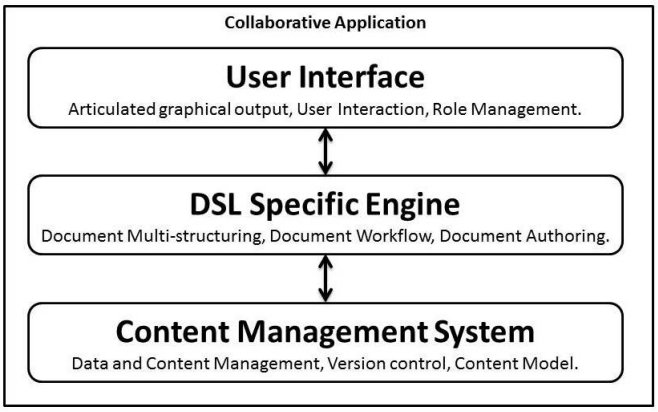

Fig. 5. Overall Architecture

The UI layer should be capable of displaying graphical output and support user interaction. Support of role definition and group management should be also handled in this layer. The DSL specific engine is a layer placed on top of a content management system and drives behaviour of collaborative applications dynamically at runtime. This engine, based on

the MVC design pattern, processes concrete DSL models, supports document structuring, workflow and authoring. The content management layer has data and content management functionalities and supports version controlling.

A. Overview

Our engine targets a layer between CMS and UI. Events are actions created in the UI layer and originate from user actions. The events form inputs for our system. Created events are firstly processed by an event handler. Internal processing of events that lead to outputs is handled by a controller as shown in Listing 1. The outputs are responses returned to the UI layer. The responses contain information about an element structure and element workflow state and a set of actions performable in next step.

```
Listing 1:
response:= initial actionsBlock;
while true {
  command (a,e,ws,r,p) := reduce(prog);
  (eCont, ws')
          := perform(command, state);
  (actionsBlock) := obtainActBlock(e,ws');
  response
          := populate(eCont, actionsBlock);
  event   := wait_for_event();
  prog    := handle(event);
}
```

Firstly, the program `prog` is evaluated and reduced to produce a command with respect to the invoked `action`, `a`, current `element`, `e`, its `workflow state`, `ws`, user `role`, `r` and other `parameters`, `p`. Performing the command with respect to a particular application `state` and current context results in an updated `workflow state`, `ws'`, and an updated `element containership`, `eCont`. The `eCont` represent the containership of elements to which the current element belongs to. Based on the element and its workflow state, a new `actionsBlock` is obtained. The `actionsBlock` represents a set of actions available in next step. The `actionsBlock` and `eCont` form a new `response`.

The first response is an initial `actionsBlock` which contains a set of actions. This is returned when user logins in the system. The application then is in an awaiting state for a next `event`. A handler for the event is defined in the `actionsBlock` which returns a new program according to the given element, its workflow state and role of user.

Based on the cycle, the additional features are defined:

Command: Commands deal with accessing and updating of element or its property in a content management system. Each collaborative process has a local state associated with an element that can be updated by performing commands. The extent of the local state is the state of the element containership to which the element belongs to and its workflow state.

Actions Block: An actions block consists of actions associated with a workflow state of a particular element.

Response: A response can be seen as an intermediary that communicates all updates back to the UI layer. The response is populated by the corresponding updated element containership and actions block which includes a list of all actions performable in the next step by the current user. Actions returned in an action block are used to dynamically render UI. In the UI, a button is created for each action. However the details of how the UI are created is outside the scope of the paper.

Events: An event occurs within a particular actions block when for example the user presses a button. An event handler is responsible for handling the event. The semantics of the language is partially defined in terms of handling the externally generated events.

Structural operational semantics for the engine has been defined by using λ-calculus which is the highly expressive, flexible and easily extended standard. However, due to its volume, the semantics is not included in the paper and will be described in our future work.

VII. Implementation Details

The proposed DSL is validated through the implementation of a prototype of a collaborative document authoring system. This section describes the implementation details of the system. The major focus has been put on the DSL engine so two existing technologies acting as data management layer and user interface layer have been used. There are a number of open source systems supporting document management with rich functionalities and different deliveries that can be customised, extended and integrated. In the Rudiment project, the engine has been implemented as an intermediary between Alfresco and Drupal. Alfresco has acted as a data management layer and Drupal has been used as a user interface layer. The technology mix represents a powerful tool to create the required environment. The engine can be populated by the concrete DSL models. We have used the bidding process model described in the case study as a test pivot.

A. Architecture

The DSL is defined for multi-structured documents. Thus data management layer is significant for the successful implementation of the engine. Data management is often driven by a content model. The content model needs to directly support the multi-level document structure. The content model of Rudiment has been designed as an extension of the content model of Alfresco. By modelling own content types, content aspects and content metadata (properties), we gained the ability to control lifecycle and manage element types as needed. In Alfresco, if the content model is customised, actions must be additionally specified too. A set of actions for each element type, including creating, editing, modifying or removing the element, has been predefined. In addition, actions to modify values of metadata were enabled.

The engine is created by using the Alfresco's REST-based web script framework. A set of web scripts have beed defined. Each webscript has been associated with a basic action such as *Create Document*, *Add Element*, *Add Metadata*, *Update*

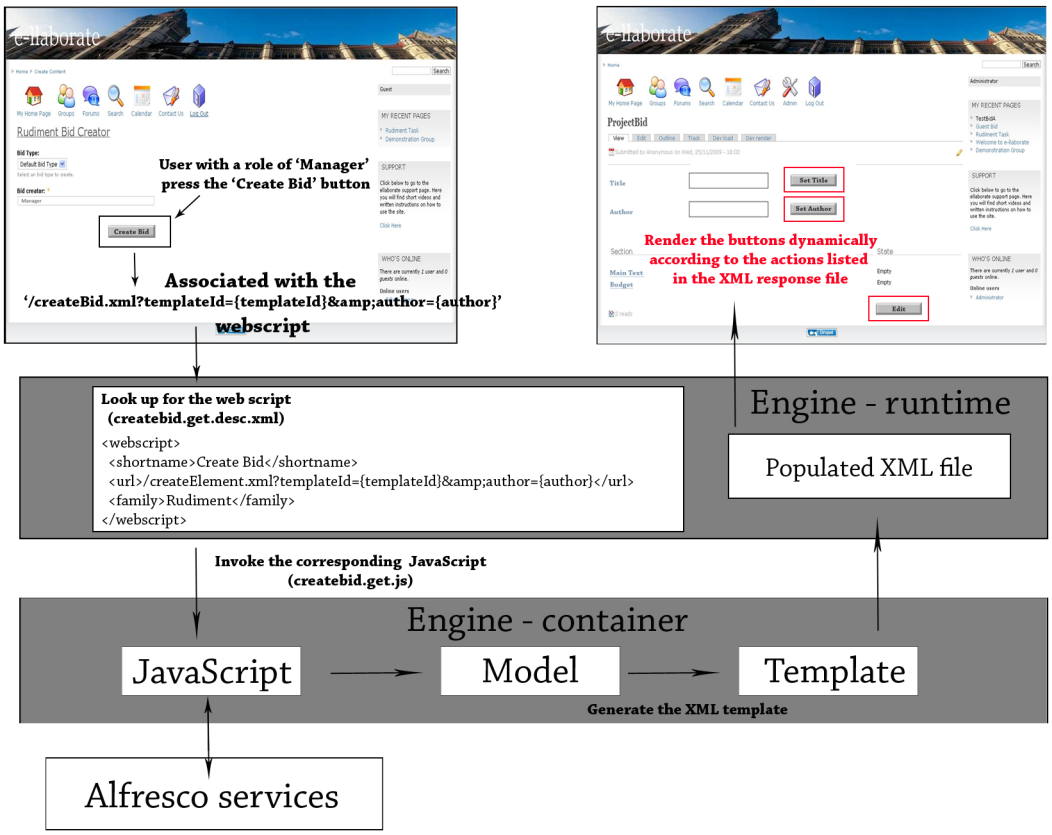

Fig. 6. System Overview

Metadata, etc. This flexible and lightweight framework uses URL addressability of objects in the Alfresco repository and the objects are accessed from a frontend application, in our case Drupal, by using XMLHttpRequests.

B. XML response

Using an XML format in the response template allows dynamic handling of interaction. The populated XML template returns information about the element, its structure, workflow states and all available actions performable in next step. An illustrative example of the execution when the *Create Bid* button is clicked is shown in Fig. 6. A user with a role of manager decides to create a bid from the template. The *Create Bid* button is clicked and it invokes a webscript associated with the action. The bid is created and persisted in Alfresco. It is in the *Prepare* state. Considering the role of user, the generated XML template is returned and contains information about the bid structure and a list of all available actions associated with the *Prepare* state in the model: *Edit*, *Set Author* and *Set Title*. In the following screen, the XML document is used to render UI. A corresponding button is created for each action included in the XML document: *Edit*, *Set Author* and *Set Title*. The populated XML template is shown in Listing 2.

VIII. Discussion

The proposed graphical DSL represents the model driven approach to the development of collaborative applications. Users of the DSL can model their own collaborative processes. The models can be used to dynamically build interfaces at runtime. By developing the abstract syntax of the DSL as a metamodel, a vocabulary of domain concepts has been represented. This vocabulary deals with form, structure and relationships that exist between the concepts. Metamodelling is the way how to design and integrate semantically rich languages in a unified way [11]. The metamodel proposed in this paper specifies multi-structured document structure, lifecycle and behaviour. It can ensure consistency across tools built from the models. The developed tools conform to policies and regulations stated for this domain.

Our approach to the development of the DSL supports its further evolution. The metamodel can be adapted, modified or extended. Constructing the language is an iterative process where the constructs can be continuously added if needed. The pivot for the DSL definition has been a research bid, however, the tool can be extended to serve any collaborative task. The concept of document management can be extended to content management, when the graphical icon for element containership might be used to represent content such as image, file or any other media. To specify a particular content

type, a number of additional properties can be associated with the particular element. Content structuring can be supported in the same way as in case of document structuring. Content-centric workflow can be modelled with the graphical icons for states, transitions and guards.

```
Listing 2:
?xml version="1.0" encoding="UTF-8"?>
<response>
 <eCont>
  <title>${elem.properties.name}</title>
  <url>${url.context}${elem.url}</url>
   ... other metadata...
  <#list coresecs as child>
     <#if child.isDocument>
      <section mandatory="core">
           ...metadata...
          <workflowState>${child.state}
          </workflowState>
         </section>
       </#if>
   </#list>
 </eCont>
 <actionsBlock>
   <action name="setTitle"
scriptName="/setTitle.xml?title={title}">
          <parameters>
             <parameter name="title" />
           </parameters>
           <role name="anyone" />
    </action>
    <action name="setAuthor"
              scriptName="...    >
             ...
    <action name="edit"
               scriptName="...    >
             ...
 </actionsBlock>
</response>
```

Although the proposed metamodel captures the most common domain constructs of the domain, it is still only a first step towards developing a more generic and automated approach. Moreover, this paper considers only a particular aspect of the communication between layers. Describing communication protocols between layers, content management system access or authentication are out of the scope of the paper.

IX. Conclusion and Future Work

In this paper, we have presented the DSL features for the collaborative document authoring domain and the way how user interfaces can be built dynamically. Precise DSL semantics will be described in our future work. Our further work will focus on the extension of the DSL to other areas of the collaborative applications domain.

References

[1] I. B. Rodriguez, G. Sancho, T. Villemur, S. Tazi, and K. Drira, "A model-driven adaptive approach for collaborative ubiquitous systems," in *Proceedings of the 3rd workshop on Agent-oriented software engineering challenges for ubiquitous and pervasive computing*, (London, United Kingdom), pp. 15–20, ACM, 2009.

[2] S. Battle and H. Balinsky, "Modelling composite document behaviour with concurrent hierarchical state machines," in *Proceedings of the 9th ACM symposium on Document engineering*, (Munich, Germany), pp. 25–28, ACM, 2009.

[3] N. Boyette, V. Krishna, and S. Srinivasan, "Eclipse modeling framework for document management," in *Proceedings of the 2005 ACM symposium on Document engineering*, (Bristol, United Kingdom), pp. 220–222, ACM, 2005.

[4] A. Marchetti, M. Tesconi, and S. Minutoli, "XFlow: an XML-Based Document-Centric workflow," in *Web Information Systems Engineering – WISE 2005*, pp. 290–303, 2005.

[5] S. Kelly and J. Tolvanen, *Domain-specific modeling*. Wiley-IEEE, 2008.

[6] A. Kleppe, *Software Language Engineering*. Addison-Wesley, Dec. 2008.

[7] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Volkel, "Design guidelines for domain specific languages." http://www.dsmforum.org/Events/DSM09/Papers/Karsai.pdf, 2009.

[8] J. Gray and G. Karsai, "An examination of dsls for concisely representing model traversals and transformations," in *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, pp. 10–pp, IEEE, 2003.

[9] T. Cleenewerck and I. Kurtev, "Separation of concerns in translational semantics for DSLs in model engineering," in *Proceedings of the 2007 ACM symposium on Applied computing*, (Seoul, Korea), pp. 985–992, ACM, 2007.

[10] I. Kurtev, J. Bézivin, F. Jouault, and P. Valduriez, "Model-based DSL frameworks," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, (Portland, Oregon, USA), pp. 602–616, ACM, 2006.

[11] T. Clark, P. Sammut, and J. Williams, "Applied metamodelling: A foundation for language driven development | lambda the ultimate." http://lambda-the-ultimate.org/node/2711, 2008.