Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 1385–1390

ISBN 978-83-60810-51-4

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 describing an implemented simple document authoring system.**

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

I. Introduction

COLLABORATIVE applications often allow its users to specify their own work patterns and authoring processes. The applications need to have an ability to integrate customised workflow process models. The way the workflow process models are designed, integrated and used depends on the structure of the particular collaborative application. Often in web applications, the Model-View-Controller (MVC) software design pattern is incorporated and a three-layer architecture, in which the presentation, the functional processing and the data management are separated concepts, is used. We assume that using the workflow process models in the functional processing layer can enrich its functionality. The workflow process models 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, with a Domain-Specific Language (DSL) as its key enabling technology, is a way to abstract away from implementation-specific details. DSLs specify what should be executed rather than how and 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 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. Metamodelling is the way to design and integrate semantically rich languages in a unified way [1]. 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 implementing a collaborative document authoring system and designing the DSL by applying the metamodelling approach. It 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 is conducted in Section 4. The domain-specific features are visualised and expressed on a metamodel in Section 5. The architecture of collaborative applications is described in Section 6. The implementation details of a simple document authoring system are outlined in Section 7. Section 8 highlights our future work.

II. Background and Related Work

A. Document Management

Focus on collaborative activities by using the model-driven approach has been applied in [2]. The work presents generic modelling approach only for collaborative ubiquitous software architectures. Document composition and life-cycle have been addressed in [3] where a theoretical framework and practical guidelines for modelling composite document behaviour are proposed. The model-driven approach to define document management applications by using Eclipse Modelling Framework is described in [4], however the work focuses only on document management as a top layer on the repository and does not consider other collaborative functionalities.

A framework for collaborative document procedures has been proposed in [5]. The work describes a run-time document workflow engine based on XML technologies. Although we have used similar constructs and XML technologies in our

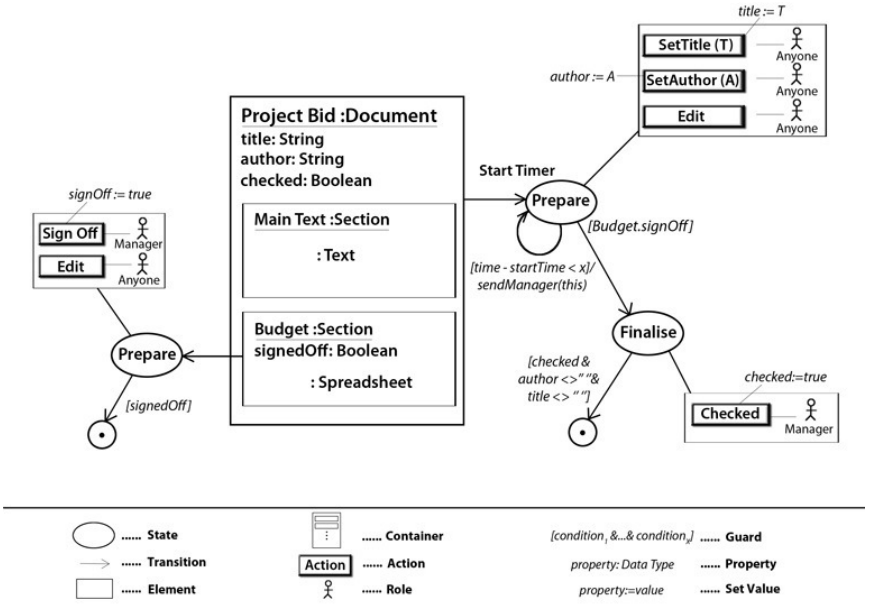

Fig. 1. Model for collaborative document authoring

prototype implementation, our work offers a domain-specific tool to build solutions at a higher level of abstraction.

Domain-specific modelling is applied to support document engineering in [6]. Although the work describes a workflow control language, it focuses more on formal specification and document rendering in directory publishing rather than on behaviour of multi-structured documents.

B. Domain Specific Languages

The more software products become robust and complex, the more sophisticated tools are required in software development. DSLs are tools to cope with the increasing user demands and the gap between IT and business concepts [7]. The need for new languages for various growing domains is strongly increasing [8]. The benefits of using DSLs to application development have been highlighted in [9].

Convergence of the model-driven approach and DSL by using a metamodel is described by [10], [11] where a 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. Using metamodels is required particularly in cases where a language needs to be defined precisely. For example, the UML specification contains one large metamodel [1].

III. Case Study

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

[1]UML 2.3, http://www.omg.org/spec/UML/2.4.1/

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 their own multi-structured document authoring processes and specify actions for particular roles.

IV. Domain Model

In this section, a collaborative authoring model for the 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 a test case. The structure of a research proposal funding bid and the bidding process are illustrated in Fig 1. This model has been simplified for the purpose of the paper.

The *Project Bid* as a main document contains descriptions in the 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 their 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 users with the role of *Manager* or *Anyone*. The role *Anyone* represents any user from the work team. While a user with the role of *Manager* can *Sign off* the budget, other collaborators are allowed to 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 *Manager* signs off the *Budget*, the state of the *Project Bid* is changed from *Prepare* to *Finalise*.

B. Domain Features

The graphical representation of the authoring process illustrated in Fig 1 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.

Element Containership: Document structure is always a key to its management. Structure encompasses well-defined manageable sub elements 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.

Properties: To ensure that elements are accessible, identifiable and discoverable, properties that provide information about the elements need to be defined. Properties can be associated with the *Container* or *Element* widgets. The control properties are used to guide transitions between two states.

Workflow: Workflow defines a series of connected steps that must be accomplished to produce a required output. 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.

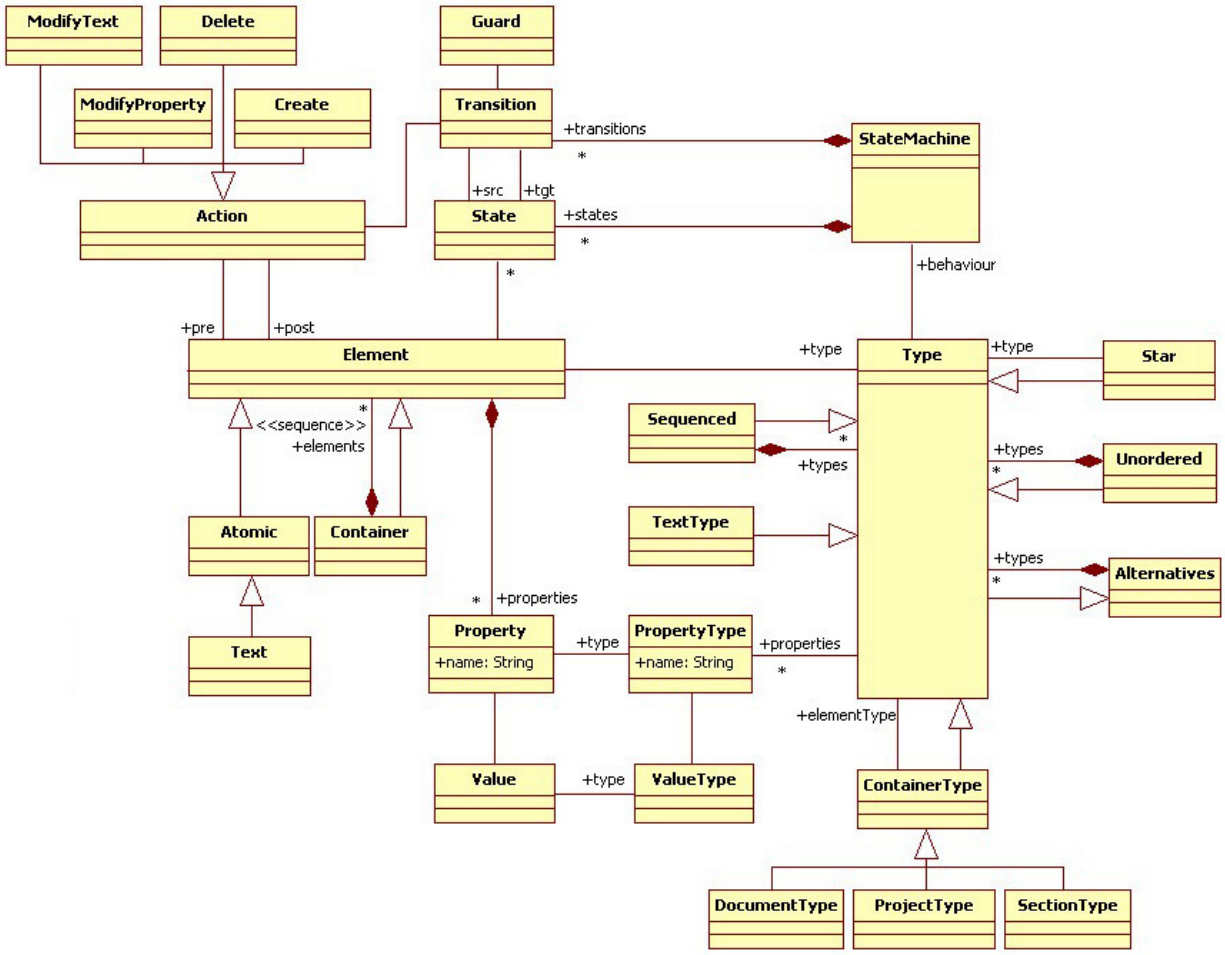

Fig. 2. Metamodel

Document authoring: A number of people can contribute and share documents. Therefore, a set of roles needs to be defined. A role is associated with a set of permissions and obligations. Roles are occupied by actors.

V. DSL Features

In this section, the domain concepts and their relationships are progressively defined. The information is visualised on a metamodel (Fig. 2).

A. Document Structure

A system manages a collection of documents. A document is a sequence of sections and sub-sections. 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 bid 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*. Each section may have some additional 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 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 instance 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 lifecycle 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. A 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 its own type. For instance, a document would be an element of DocumentType. The 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, specifies the possible multiplicity of a particular type within a container. Elements have properties and element types have property types. This hierarchy of types at the metamodel level enables the modelling of reusable document templates.

F. Element Lifecycle

Each element goes through a plan that is described in terms of a collection of (partially ordered) *tasks* that must be performed by actors in designated roles (Fig 3). Each *task* corresponds to an *action*. Element states are associated with actions blocks. An *ActionBlock* is a container of actions. Each action can be performed by a number of roles. 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. Explaining the conditions for each action in detail is out of scope of this paper.

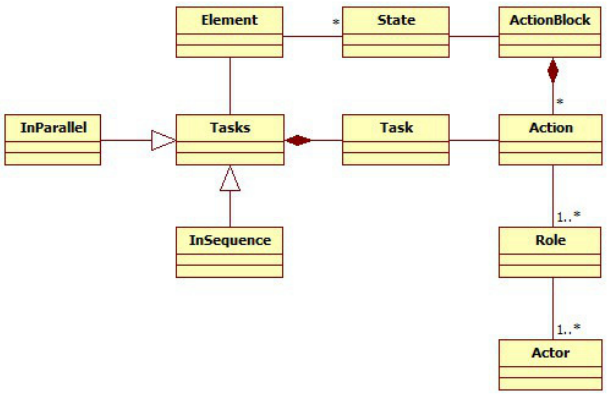

Fig. 3. Element Lifecycle

VI. Architecture of Collaborative Applications

The language proposed in this paper is a tool that can support the design of collaborative processes. The aim of this section is to describe an overall architecture of collaborative applications and the applicability of the DSL. The system architecture comprises three layers: content management system (CMS), DSL specific engine and user interface (UI).

The UI layer should provide a means of input and output, allowing users to interact with the system. Roles and collaborative groups should be managed in this layer. The DSL-specific engine is a layer placed on top of a content management system and should drive behaviour of collaborative applications dynamically at runtime. This engine, based on the MVC design pattern, should process concrete DSL models, supports document structuring, workflow and authoring. The content management layer has the data and content management functionalities and supports version controlling.

A. Overview

Events are actions created in the UI layer and originate from user actions. The events form inputs for the system. Created events are first 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 the element structure, its workflow state and a set of actions performable in the next step.

```
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);
}
```

First, 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` represents the containership of elements to which the current element belongs. 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 a user logs 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 an element or its property in a content management system.

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 includes information about the corresponding updated element structure and 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 the UI. In the UI, a button is created for each action. However the details of how the UI is created are outside the scope of the paper.

Events: An event occurs within a particular actions block, e.g., when a user presses a button. An event handler is responsible for handling the event.

VII. Implementation Details

A prototype of a collaborative document authoring system has been implemented. This section describes the implementation details of the system. 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, 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.

A. Document Management

The DSL is defined for multi-structured documents. The content model used in the data management layer should support the document structure. The content model of Rudiment has been designed as an extension of the content model of Alfresco. By modelling content types, content aspects and content metadata (properties), we gained the ability to control the lifecycle and manage element types as needed. In Alfresco, if the content model is customised, actions must be additionally specified. 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 has been implemented as an intermediary between Alfresco and Drupal for the bidding process model. The engine is created by using Alfresco's REST-based web script framework. A set of web scripts have been defined. Each web script has been associated with a basic action such as *Create Document*, *Add Element*, *Add Metadata*, *Update 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. System Execution Example

An illustrative example of the execution process, when the *Create Bid* button is clicked, is shown in Fig 4. A user with a role of manager decides to create a bid from the template. The *Create Bid* button is clicked in Drupal and it invokes a web script associated with the action. The bid is created and persisted in Alfresco. It is in the *Prepare* state. The populated XML template is returned to the UI layer. It contains information about the bid structure and a list of all available

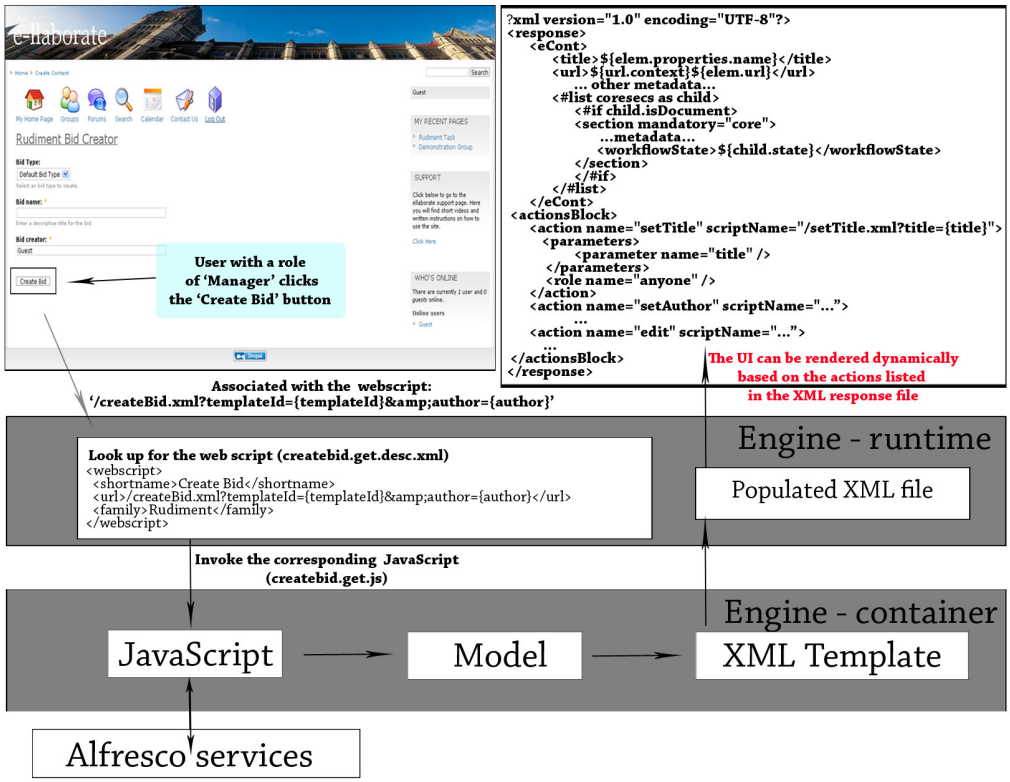

Fig. 4. System Overview

actions associated with the *Prepare* state in the model: *Edit*, *Set Author* and *Set Title*. In the following step, the XML document is used to render the UI. The following buttons are created in the UI: *Edit*, *Set Author* and *Set Title*.

VIII. Conclusion and Future Work

In this paper, we presented the DSL features for the collaborative document authoring domain. 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. The pivot for the work has been a research bid but the tool can be extended to serve any collaborative task. The future work will include the implementation of a more generic DSL-engine.

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.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] V. Djukic, I. Lukovic, and A. Popovic, "Domain-specific modeling in document engineering," in *Computer Science and Information Systems (FedCSIS), 2011 Federated Conference on*, pp. 817–824, 2011.

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

[8] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Vlkel, "Design guidelines for domain specific languages," in *The 9th OOPSLA workshop on domain-specific modeling*, 2009.

[9] 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*, IEEE, 2003.

[10] 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.

[11] 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.