Supporting Entailment Constraints in the Context of Collaborative Web Applications

Patrick Gaubatz and Uwe Zdun
Faculty of Computer Science
University of Vienna, Vienna, Austria
firstname.lastname@univie.ac.at

ABSTRACT

Collaborative Web applications allow several users to collaboratively work on the same artifact. In addition to popular use cases, such as collaborative text editing, they can also be used for form-based business applications that often require forms to be filled out by different stakeholders or stakeholder roles. In this context, the different stakeholders often need to fill in different parts of the forms. For example, in an e-health application a nurse might fill in the details and a doctor needs to sign them. Role-based access control and entailment constraints provide means for defining such restrictions. So far entailment constraint have mainly been studied in the context of workflow-based architectures, but not for collaborative Web applications. We present a generic approach for the specification and enforcement of entailment constraints in collaborative Web applications that supports their real-time nature and the non-prescriptive order in which tasks can be performed. Further, we discuss a model-driven implementation approach of our concepts and lessons learned and limitations.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Web Applications; D.2.2 [**Software Engineering**]: Design Tools and Techniques

Keywords

Collaborative Web Application, Entailment Constraint, RBAC

1. INTRODUCTION

Collaborative Web applications such as Google Docs[1], Etherpad[2], or Creately[3] aim to efficiently support the joint work of different teams members, allowing them to collaboratively work on the same artifact at the same or a different time. As such collaborative Web applications are getting more and more popular, it is interesting to study their use for typical business applications that often require multiple forms to be filled out by different stakeholders or stakeholder roles. The basic approach required for such form-based collaborative applications has for instance been studied in MobWrite [8], which enables users to collaboratively fill out HTML forms. However, the – in this context crucial – aspect of access control has mainly been studied on a per-document level so far.

In form-based applications often situations occur in which different stakeholders or stakeholder roles need to fill in different parts of the forms. If restrictions on who is allowed to fill in which parts at which time exist, then proper access control must be ensured. For example, in an e-health application a nurse might fill in the details and a doctor needs to sign the document. Or a doctor files a report and a second doctor needs to check and sign the document (to realize the so-called four eyes principle). Access control in such situations has been studied in the context of role-based access control (RBAC) [9]. In RBAC, roles are used to model different job positions and scopes of duty within an information system. These roles are equipped with the permissions to perform tasks. Human users (subjects) are assigned to roles according to their work profile [11]. The examples given above have been generalized under the term entailment constraints [12], i.e. constraints that place some restriction on the subjects who can perform a task x given that a certain subject has performed another task y.

So far entailment constraints have mainly been studied in the context of workflow-based architectures (see for instance [2, 12, 14]), but not for collaborative Web applications. In this paper, we will introduce a novel approach for specifying and automatically enforcing entailment constraints in collaborative Web applications. Our approach aims to support their real-time nature and the non-prescriptive order in which tasks can be performed in collaborative Web applications through a server-side constraint checking service. On client side we propose non-intrusive modifications that integrate this constraint checking service into the collaborative Web application. A constraint model is used to integrate the client and server side in a way that requires the Web developer only to make minor modifications, whereas all security-related aspects are specified by a security expert. That is, our approach enables the separation of concerns between these two roles. In our prototype implementation we use a model-driven approach to automatically generate all required artifacts from the constraint model. That is, only the specification of the constraint model is necessary to augment a collaborative Web application with entailment constraint enforcement. Using the model-driven implementation is only an option in our approach: It is equally possible to manually hook our Web services with a few extra steps into existing collaborative Web applications.

This paper is structured as follows: In Section 2 we introduce entailment constraints. We motivate our work in Section 3. Our approach is described in Section 4. In Section 5 we discuss a pro-

[1]`https://docs.google.com`
[2]`http://etherpad.org`
[3]`http://creately.com`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAC'13 March 18-22, 2013, Coimbra, Portugal.
Copyright 2013 ACM 978-1-4503-1656-9/13/03 ...$15.00.

totypical implementation, and Section 6 explains how it can be used to resolve the motivating example. After discussing related works in Section 7, and the lessons learned and limitations of our approach in Section 8, we conclude in Section 9.

2. ENTAILMENT CONSTRAINTS

As explained before, entailment constraints are a concept in the RBAC domain, defined as follows [12]: A *task-based entailment constraint* places some restriction on the subjects who can perform a task x given that a certain subject has performed another task y.

Different kinds of entailment constraints can be distinguished [12]: Mutual exclusion and binding constraints are typical examples of entailment constraints. They can be subdivided into *static mutual exclusion* (SME) and *dynamic mutual exclusion* (DME) constraints. An SME constraint defines that two tasks must never be assigned to the same role and must never be performed by the same subject (i.e. to prevent fraud and abuse). This constraint is global with respect to *all instances* in an information system. In contrast, DME refers to individual instances and can be enforced by defining that two tasks must never be performed by the same subject in the *same instance*. In contrast to mutual exclusion constraints, binding constraints define that two bound tasks must be performed by the *same* entity. In particular, a *subject-binding* constraint defines that the same individual who performed the first task must also perform the bound task(s). Similarly, a *role-binding* constraint defines that bound tasks must be performed by members of the same role but not necessarily by the same individual.

3. MOTIVATING EXAMPLE

In the context of health care, a medical record is used to document a patient's medical history and care. It is maintained by health care professionals (e.g. doctors) and includes information such as therapy plans, various results, and reports. There is an ongoing trend towards electronic medical records. For many of these records, a number of different health care professionals (e.g. doctors, nurses, administrative persons) have to complete a form.

Today, often the health care professionals are confronted with strict, standardized forms with precisely specified form fields, which are "hard-coded" into a custom-made (legacy) application. Adding new fields or changing existing ones is usually an error-prone and cumbersome task. An alternative solution would be workflow-based (or pageflow) applications. That is, the application consists of workflow tasks executed in a prescribed order. Each subset of the form fields, to be filled out by a specific person or user role, would be realized using a single workflow task. Executing the workflow will eventually lead to the completion of the form. The workflow-based solution has the advantage over the hard-coded solution that modifications of the control flow are possible without touching the source code. However, both solutions have a major disadvantage: Their control flows are statically prescribed at design time. It is not possible to leave this "prescribed path". This is a problem because the whole process might easily get stuck. For example, a missing signature from a doctor, who is currently off-duty, might prevent other health care professionals to proceed to the next group of form fields.

These problems led us to study in how far medical records can be created by letting users complete forms using a collaborative Web application in which ordinary HTML forms are used. In contrast to the previously described solutions a collaborative Web application would not exhibit the problems of a "prescribed path". Instead, it allows form fields to be filled out concurrently by various users at the same time. Thus, it can easily accommodate "unforeseen"

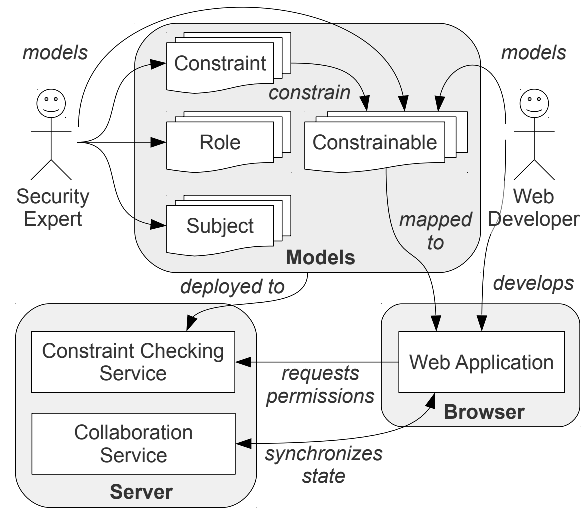

Figure 1: Architectural Overview of the Approach

deviations from the originally intended workflow.

This leads to the requirement to enforce *entailment constraints* in such collaborative Web applications: For some form fields it is required to precisely specify who is allowed to enter which information when. For example, a medical record form might contain an input field where the doctor in charge has to enter a documentation about the prescribed therapy. Before the final discharge of the patient, we might want the same doctor who prescribed the therapy to sign the complete medical record again. Thus, we are effectively constraining these two form fields using a *subject-binding* constraint. Another example is that for quality assurance reasons (i.e. realizing the four-eye principle) another doctor has to sign the whole record. Here, a *dynamic mutual exclusion* constraint between the two signature fields would provide means to prevent the same person to sign both fields. In this paper, we describe – to the best of our knowledge – the first approach to specify and enforce such entailment constraints in collaborative Web applications.

4. APPROACH

4.1 Approach Overview

Figure 1 gives an architectural overview of our approach. The figure illustrates two distinct stakeholder roles, Web developers and security experts, whose tasks can be clearly separated in our approach. At design time the security expert first models the roles and subjects required for the Web application (see Section 4.2). In our approach, a Web application consists of constrainable elements (e.g. a button, a JavaScript method, or even a remote service invocation). Both the Web developer and the security expert model these constrainable elements. The latter may then use constraint models to make the constrainable elements subject to various entailment constraints. The Web developer role realizes the collaborative Web application. The collaborative aspect is typically realized using a Publish-Subscriber architecture. That is, every state change of the Web application is distributed to other session participants using a collaboration service (i.e. the message broker).

From the Web developer's point of view, the development process, described so far, does not differ to the "standard approach" to developing collaborative Web applications. Only the following additional steps have to be carried out by the Web developer to guarantee compliance to the entailment constraints (see also Sec-

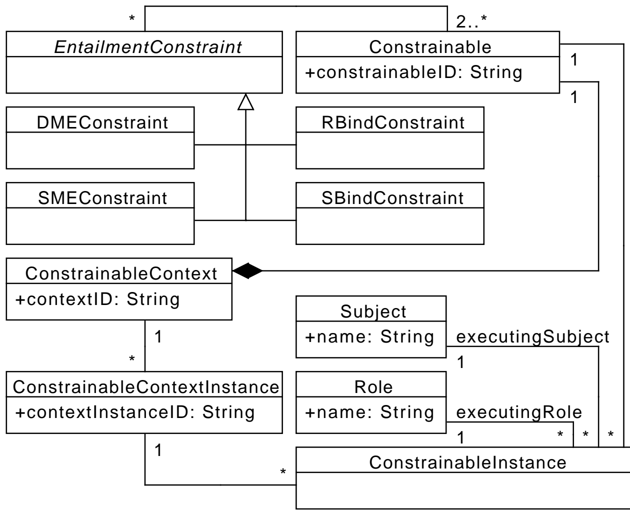

Figure 2: The Constraint Model

tion 4.3). Firstly, he or she has to map the (abstract) constrainable elements to concrete implementation-level artifacts (e.g. HTML elements) of the Web application. Whenever one of these constrained elements is invoked (e.g. a button is clicked), the Web application must request the permission to do so from a constraint checking service. This service uses a model-based constraint checking engine which either allows or denies the invocation. As the engine's decision is based on the defined subjects, roles, constraints and constrainables models, these modeling artifacts have to be deployed to the engine before. Only if the constraint service has permitted the invocation, the Web application should actually change its internal state (e.g. by calling the button's `onclick` handler) and eventually notify the session participants via the collaboration service of this particular state change. Conversely, if the invocation has been denied, the Web application must prevent the state change.

4.2 Constraint Model

The core element of our approach is a generic constraint model. Figure 2 shows an UML2 class diagram of this model. The model specifies that the *Constrainables* (mentioned before) can be constrained by an arbitrary number of *EntailmentConstraints*. Every *Constrainable* has a unique *constrainableID* and belongs to a *ConstrainableContext*. A *ConstrainableContext* has a unique *contextID*, aggregates *Constrainables*, and constitutes a self-contained execution domain or environment. More precisely, it denotes a distinct "collaboration canvas" of a Web application. For example, in a collaborative text editor there could be a *ConstrainableContext* with the ID "text document". Depending on the type of collaborative Web application, the application as a whole or just a distinct part of it constitutes a *ConstrainableContext*. We use the classes *Subject* and *Role* to model the available subjects and roles.

These model elements are usually defined at design time and instantiated at runtime using the two classes *ConstrainableContextInstances* and *ConstrainableInstance*. A *ConstrainableContext* can have an arbitrary number of *ConstrainableContextInstances*. For example, considering the previously mentioned "text document" example, a group of users may collaboratively work on a single *ConstrainableContextInstance* with a *contextInstanceID* "text document instance xyz". Whenever a *ConstrainableContextInstance* is created, a *ConstrainableInstance* has to be created for every aggregated *Constrainable* of the respective *ConstrainableContext*. That is, a *ConstrainableInstance* is an instance of exactly one *Constrainable* and is part of exactly one *ConstrainableContextInstance*.

When a *ConstrainableInstance* is invoked, the *executingRole* and *executingSubject* associations are used to establish relations to the respective *Role* and *Subject* objects actually invoking the *ConstrainableInstance*.

Method	Parameters	Return
contextInstance	contextID, contextInstanceID	–
invoke	contextID, contextInstanceID, constrainableID, subject, role	Boolean

Table 1: Interface Excerpt of the Constraint Checking Service

4.3 Runtime Enforcement and Architecture

In the previous section we have presented a generic model for defining abstract constrainable elements and making them subject to different types of entailment constraints. This section discusses the runtime architecture needed to actually enforce the compliance of the Web application with regard to the defined constraints. In essence, our approach requires both, a dedicated server-side component and modifications to the client-side application logic.

4.3.1 Server-side Constraint Checking Service

On server side, a generic and self-contained service realizes the Policy Decision Point (PDP) [7], the entity that actually decides if an invocation is to be allowed or not according to defined entailment constraints. Generally, a PDP needs to know about existing subjects, roles, and policies to be able to actually make decisions. Hence, the service needs to have access to the full set of modeling artifacts created by the security expert at design time.

Table 1 lists two essential methods that the constraint checking service has to provide to the Web application. Firstly, the Web application must call the `contextInstance` method, before any constraint checking can be done at all. It also has to provide both, a `contextID` (e.g. "text document") of an existing *ConstrainableContext* and a unique `contextInstanceID` (e.g. "text document xyz"). The service is then able to instantiate and initialize the classes *ConstrainableContextInstance* and *ConstrainableInstance*.

After `contextInstance` has been called, the service is initialized. The second mandatory method, `invoke`, must be called (by the Web application) before a constrainable element is invoked. Note that this method requires numerous parameters to be supplied: The service needs to know which subject (`subject`), using which role (`role`) is going to invoke a specific constrainable element (`constrainableID`). Furthermore, a context instance (`contextInstanceID`) and a context (`contextID`) need to be specified. If the invocation is allowed (i.e. no entailment constraints are violated), the service will then respond with the Boolean value `true`, and the Web application may finally perform the actual invocation of the constrainable element. Otherwise `false` is returned, and the invocation must be prevented. Whenever an invocation is allowed, the service will assign the *executingSubject* and *executingRole* relations of the corresponding *ConstrainableInstance* object, according to the provided `subject` and `role` parameters which is required by the underlying constraint checking algorithms (see [12] for details on the algorithms).

4.3.2 Client-side Modifications

In addition to the server-side constraint checking service, our approach also requires making modifications of the client-side application logic. These modifications would typically be performed

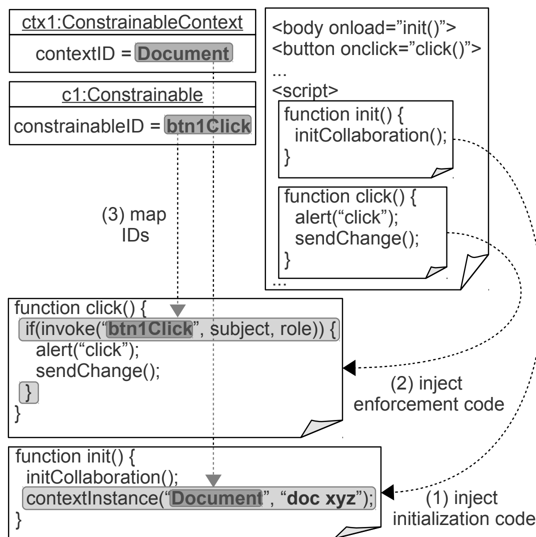

Figure 3: Required Mappings and Modifications

automatically using a model-driven code generator (see Section 5 for a discussion of our prototype). However, it is also possible to perform the modifications manually. This way it is possible to hook our Web services into existing collaborative Web applications using the few additional steps described in this section.

Figure 3 illustrates the required modifications. In general, the Web developer has to (1) call the `contextInstance` method and (2) use the `invoke` method of the constraint checking service and enforce its decision whenever a constrainable element is invoked.

Figure 3 illustrates a collaborative document editor application, in which our constraint model contains a *ConstrainableContext* with a *contextID* "Document" and an (examplary) constrainable element with a *constrainableID* "btn1Click". The figure shows an excerpt of the application's main HTML file. We can see that the browser will execute the `init()` method as soon as the `<body>` element has been parsed. The embodied `initCollaboration()` method initializes the Web application's collaboration functionality. Furthermore, there is a `<button>`. When it is clicked, an alert box is shown and this state change is propagated to other session participants using `sendChange()`.

The first required modification is the injection of the initialization code in which we have to call the `contextInstance` method of the constraint checking service. In the example, we specify that the constaint service should create an instance of the "Document" *ConstrainableContext* from our defined model and give the instance an ID of "doc xyz". Next, we have to inject the actual enforcement code, which is done by inserting a call to the constraint service's `invoke` method into the `click()` method. The result of this modification is that the original method body will only be executed, if the constraint service allows it.

5. THE COCOFORM IMPLEMENTATION

This section discusses a concrete implementation of the previously described approach. The developed prototype is called Constrainable Collaborative Forms (CoCoForm)[4] and can be used to realize the e-health record case from Section 3. The basic idea is that an ordinary HTML form (e.g. an electronic health record form) can be filled out collaboratively by different users at the same time. Parts of this form are subject to entailment constraints.

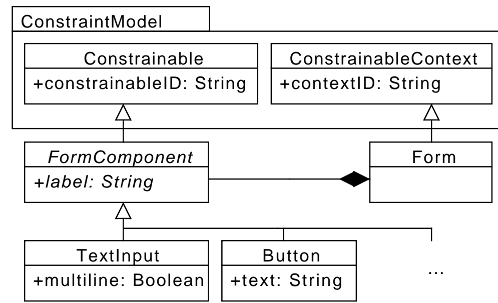

Figure 4: The WebForm Model

We decided to use a model-driven development approach for the implementation. Hence, we extended the Constraint model (see Figure 2), as can be seen in Figure 4. Every instance of *Form* constitutes a self-contained *ConstrainableContext*. A *Form* aggregates *FormComponents*. More precisely, a *FormComponent* can be a *Button*, a *TextInput*, and so on. As these components are subtypes of *Constrainable*, they can be constrained by entailment constraints.

Both models, the Constraint and the WebForm model, have been implemented in Frag [15], a Java-based, interpreted, tailorable language, specifically designed for the task of model-driven development. Frag supports both model-driven generation and interpretation of models at runtime. Hence, we have also implemented the model-based, runtime constraint checking engine in Frag. The next step was the development of the constraint checking service. We chose a RESTful service interface design, implemented in Java, using the JAX-RS API[5] and Jetty[6] as our servlet container. The service returns JSON data and is merely a HTTP-based connector between the Web application and the constraint checking engine.

The actual Web application consists of a single HTML5 document and a generic JavaScript library. We use the Open Cooperative Web Framework [13] for all collaborative aspects of our Web application. It consists of a JavaScript library, a Java servlet (i.e. the *Collaboration Service* component depicted in Figure 1), and realizes a Publish-Subscriber architecture.

For the mapping of the constrainable elements to concrete implementation-level artifact (see Section 4.3.2), the CoCoForm implementation leverages a model-driven code generator. It is used to automatically generate an instantly deployable HTML5 skeleton document from a WebForm model instance. The actual mapping information of each *FormComponent* (i.e. the constrainable element) and each *Form* (i.e. the constrainable context) is attached to the corresponding HTML5 tags. More specifically, we annotate the tags using custom (HTML5) `data-*` attributes. For example, an instance of *Form* with a *contextID* "f1" will result in a `<form data-context-id="f1">` tag. Analogously, an instance of *Button* with a *constrainableID* "b1" will be transformed to `<button data-constrainable-id="b1">`.

At runtime, the generic JavaScript library then uses these attributes to automatically register `onclick` (for buttons) and `onchange` (for text input fields) handlers for the corresponding elements. Whenever these callback functions are executed (e.g. a button has been clicked), the application calls the constraint

[4]A (proof-of-concept) CoCoForm demo application is available at `http://demo.swa.univie.ac.at/cocoform`

[5]JAX-RS, `http://jax-rs-spec.java.net`

[6]Jetty, `http://eclipse.org/jetty`

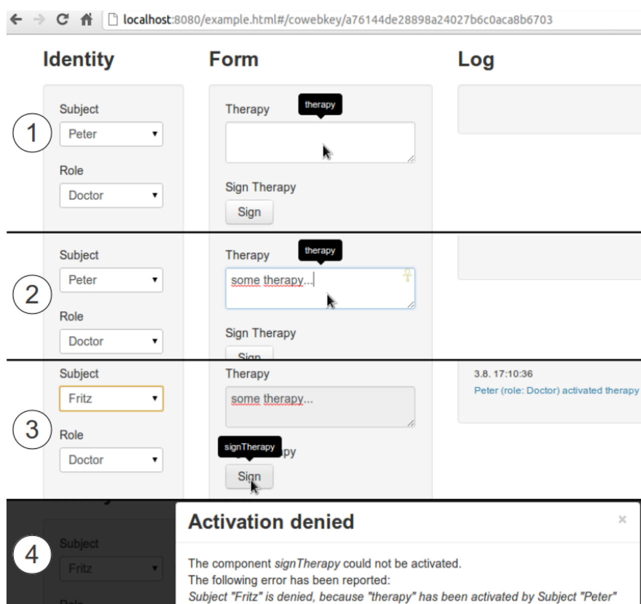

Figure 5: Subject-Binding with CoCoForm

service's `invoke` method and enforces the returned decision. If the invocation is allowed, the application disables the corresponding *FormComponent* to prevent further editing (i.e. components can only be invoked once). Secondly, the state of the component (e.g. the actual `value` of the input field, as well as the `disabled` flag) is distributed to the other session participants.

6. MOTIVATING EXAMPLE RESOLVED

Let us revisit the motivating example from Section 3. We will discuss the CoCoForm implementation of the subject-binding example from Section 3. Figure 5 shows a few screenshot excerpts of an example form. There are two form components: a text input field, which is used to document a patient's therapy, and a button, which is used to sign the documentation. For these two components a subject-binding constraint has been defined. In the first screenshot we can see the empty form. Next, *Peter*, a *Doctor*, fills out the therapy text input field. At the same time, *Fritz* joins the session and sees that the input field has already been filled out by *Peter*. Additionally, he tries to sign this form. However, he receives an error message, saying that he is not allowed to sign. This is due to the subject-binding constraint, which eventually requires *Peter* to sign. In a similar way, CoCoForm supports the definition of all entailment constraints required in the e-health case.

7. RELATED WORK

There are already some frameworks and libraries that facilitate the development of collaborative Web applications. For instance, the Open Cooperative Web Framework [13] consists of a set of JavaScript libraries and a generic Java servlet. The beWeeVee SDK [3] is a .NET-based framework and requires the Microsoft Silverlight browser plugin to be installed. MobWrite [8] is another approach for enabling real-time collaboration. However, it is restricted to synchronizing HTML forms, and the reusability and applicability is thus somewhat limited. Heinricht et al. [4] present a generic collaboration infrastructure aimed at transforming existing single-user Web applications into collaborative multi-user Web applications. In principle, our approach embraces the usage of already existing libraries and approaches. In fact, we used the Open Cooperative Web Framework to implement the collaboration aspects of CoCoForm (see Section 5). However, there is one requirement: The synchronization process of the library/framework must be interceptable. More precisely, we must prevent state changes, which have not been permitted by the constraint checking service, to be synchronized. Thus, these service invocations have to be conducted before any synchronization takes place.

The concept of task-based entailment constraints originally originates the domain of business processes and workflows. Bertino et al. [2] introduce the notion of assigning roles or subjects to tasks in a workflow and making them subject to *separation of duty* constraints. Wainer et al. [14] propose a system architecture that clearly separates the permission service from the workflow engine. Furthermore, they also present a modeling solution for specifying *binding of duty* constraints. Strembeck et al. [12] present a set of generic algorithms that ensure the consistency of entailment constraints. We used these algorithms to implement our constraint checking engine (see Section 5). In general, the existing literature – almost exclusively – examines entailment constraints in a workflow and business process context. As a result, the presented solutions are aligned with concepts that are specific for these contexts. However, in the context of collaborative Web applications, we can not resort to concepts like *task*, *process instance*, and so on. Hence, we generalized the already existing works and proposed a generic model (see Section 4.2). Instead of constraining tasks in a process, in our approach we are constraining abstract constrainable elements, which have to be mapped to concrete implementation-level artifacts.

A lot of work has been conducted in the area of RBAC in the context of Web applications and services. Ahn et al. [1] present an approach for injecting RBAC into an already existing Web-based workflow system. They propose a special reverse proxy that is able to enforce RBAC rules transparently to the actual Web application behind. Sohr et al. [10] and Hummer et al. [5] propose a similar approach in the context of Web Services. More precisely, they present generic interceptors that can be plugged into (Java-based) Web service stacks. These interceptors intercept service invocations and are then able to prevent the actual invocation in case of a policy violation. Again, this happens transparently to the underlying service implementation. In contrast, our approach requires modifications of the original client-side application logic to be made. This drawback is due to the way modern HTML5/JavaScript-based Web applications work. However, the usage of Aspect-oriented programming could help mitigating this problem (see Section 8).

8. LESSONS LEARNED

Using the CoCoForm prototype (see Section 5) we have been able to demonstrate the feasibility of our approach (see Section 1). More precisely, we showed that the concept of task-based entailment constraints can be adapted to fit into the context of collaborative Web applications. In the following paragraphs we want to discuss our lessons learned and the limitations of our approach.

The genericity of the proposed constraint model (see Section 4.2) and the constraint checking service (see Section 4.3) allow our approach to be applied to many different types of collaborative Web applications. Moreover, a single instance of the constraint checking service can potentially handle an arbitrary number of Web application instances. Thus, it is sufficient for an organization to maintain one instance of the service (maybe in replicated form).

Another positive aspect of our approach is that it follows the principle of separation of concerns. That is, the definition of roles, subjects, and constraints is completely decoupled from the actual

application. Thus, a security expert does not need to care about any implementation-level artifacts at all, whereas the Web developer does not need to care about anything related to RBAC.

We have implemented our approach with model-driven techniques to automate the generation of all additional constraint checking code. It is also possible to use the approach using manual modifications of the Web application. That is, existing code can also be instrumented this way and used with our approach by manually following the steps illustrated in Figure 3.

There are also some limitations. Firstly, our approach induces a slight performance penalty due to the required extra call of the constraint checking service. However, in the context of collaborative Web applications, which are innately prone to requiring lots of service calls, the effect of this extra call is more or less negligible.

Another limitation are the needed adaptations of the client-side application (see Section 4.3.2). The code, needed to enforce the constraint checking service's decision, has to be embedded directly into the application logic. This results in scattered and tangled code which is hard to maintain. As we have already pointed out, a model-driven code generator can avoid this issue. Besides that, aspect-oriented programming (see, e.g. [6]) can be used to decouple the enforcement-related code from the actual application code.

In our approach, the client application handles enforcement. From a security perspective, however, we often cannot trust code that is executed on the client (i.e. in a Web browser). The reason is that we cannot prevent a potential attacker from modifying the code to be executed. For instance, let us assume that there is a JavaScript method that calls the constraint checking service. An attacker might effectively undermine the enforcement just by overwriting this method and preventing the service from getting called. The effects of such client-side code injections can be contained by preventing any unauthorized state change from being distributed to other session participants. This can be achieved by using public-key cryptography. That is, the constraint checking service returns digitally signed permission documents in which the signature covers both, the actual decision and all parameters. Whenever the Web application wants to send a synchronization event to the collaboration service, it has to attach the signed permission to the request. The latter is then routed through an enforcement proxy. This proxy will only forward the request to the service, if the signature is valid (i.e. the permission document has not been tampered with) and the invocation has been permitted by the constraint checking service. In general, all (server-side) services belonging to the Web application must be tunnelled through the enforcement proxy. With these modifications we can guarantee that client-side code injections do not lead to a server-side state change or an impact on session participants.

9. CONCLUSIONS

Our approach demonstrates that the concept of task-based entailment constraints can be adapted to fit into the context of collaborative Web applications. That is, we can support collaborative editing of form-based applications with no prescribed order, and precisely specify constraints on who can perform which tasks when. We presented a generic approach that can be applied to many different collaborative Web applications. It requires some modifications to existing Web applications, but these – as well as the generation of all other required artifacts – can optionally be automated with model-driven development techniques. Our approach introduces some security concerns in untrusted environments, but these can be mitigated using public-key cryptography (see Section 8).

As future work, we will address these limitations and try to explore and establish the concept of RBAC in the context of collaborative Web application further. Furthermore, we will apply our approach to other types of collaborative processes. In particular with regard to dynamic processes (e.g. text editing or modeling) we will have to deal with completely dynamic document and constraint models (i.e. models that change at runtime).

10. REFERENCES

[1] G.-J. Ahn, R. Sandhu, M. Kang, and J. Park. Injecting rbac to secure a web-based workflow system. In *Proceedings of the fifth ACM workshop on Role-based access control*, RBAC '00, pages 1–10, New York, NY, USA, 2000. ACM.

[2] E. Bertino, E. Ferraria, and V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information and System Security*, 2(1):65–104, 1999.

[3] BeWeeVee. BeWeeVee – Life collaboration framework. `http://www.beweevee.com/`.

[4] M. Heinrich, F. Lehmann, T. Springer, and M. Gaedke. Exploiting single-user web applications for shared editing: a generic transformation approach. In *Proceedings of the 21st international conference on World Wide Web*, WWW '12, pages 1057–1066, New York, NY, USA, 2012. ACM.

[5] W. Hummer, P. Gaubatz, M. Strembeck, U. Zdun, and S. Dustdar. An integrated approach for identity and access management in a soa context. In *16th ACM Symposium on Access Control Models and Technologies*, 2011.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP'97 Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0053381.

[7] S. Kunz, S. Evdokimov, B. Fabian, B. Stieger, and M. Strembeck. Role-based access control for information federations in the industrial service sector. In *ECIS*, 2010.

[8] MobWrite. MobWrite - Real-time Synchronization and Collaboration Service. `http://code.google.com/p/google-mobwrite/`.

[9] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *Computer*, 29(2):38 –47, 1996.

[10] K. Sohr, T. Mustafa, X. Bao, and G.-J. Ahn. Enforcing role-based access control policies in web services with uml and ocl. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, pages 257–266, Washington, DC, 2008. IEEE Computer Society.

[11] M. Strembeck. Scenario-driven Role Engineering. *IEEE Security & Privacy*, 8(1), January/February 2010.

[12] M. Strembeck and J. Mendling. Generic algorithms for consistency checking of mutual-exclusion and binding constraints in a business process context. In *Proceedings of the 2010 international conference on On the move to meaningful internet systems - Volume Part I*, OTM'10, pages 204–221, Berlin, Heidelberg, 2010. Springer-Verlag.

[13] The Dojo Foundation. Open Cooperative Web Framework. `http://opencoweb.org/`.

[14] J. Wainer, P. Barthelmes, and A. Kumar. W-RBAC - A Workflow Security Model Incorporating Controlled Overriding of Constraints. *International Journal of Cooperative Information Systems (IJCIS)*, 12(4), Dec 2003.

[15] U. Zdun. Frag. `http://frag.sf.net/`.