Specifying and Enforcing Workflows in Ruby on Rails

BY

Daniele Rossetti
Laurea, Politecnico di Milano, Milan, Italy, 2011

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2013

Chicago, Illinois

To my family.

To my friends.

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor Lenore Zuck for her steady and precious support. She gave me substantial liberties, but was always available for a lot of good advices and formative chit-chats when I needed them. I also thank Tim Hinrichs and V.N. Venkatakrishnan, who have been helping me since the very beginning of my work and greatly contributed to the development of this thesis. Finally, I want to say to everyone, who directly or indirectly helped me and supported me during the time I wrote this thesis: Thank you!

DR

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Nowadays, Web applications are afflicted by numerous vulnerabilities and there exist many attacks that exploit them to execute malicious tasks. In this thesis we focus on vulnerabilities related to workflows, which are sequences of steps that the user must perform in order to complete some transaction. When the Web application fails to correctly enforce the workflows, undesired violations may be allowed. Currently, there is no systematic methodology for enforcing workflows and the implementation is left to the developer, which may result in a weak application, vulnerable to attacks. In order to address this issue, we present the framework *Workflower*, which allows the developer to easily specify workflows and automatically enforce them. The framework allows the specification to be declarative and separated from the application logic, so that it is easier to understand and maintain. The specification is securely and automatically enforced in the application, so that any violation is prevented without requiring the developer to manually implement any defense. Additionally, it supports several features such as concurrent workflows, multiple instances workflows, automatic redirection and request resuming.

CHAPTER 1

INTRODUCTION

Web applications are quickly becoming the most common way to access services offered by different environments and they are being used to provide functionalities that range from simple tasks such as writing an email to more delicate operations such as payments for online shopping or banking operations. The more common web applications are, the more they have become an attractive target for attacks. Numerous vulnerabilities afflict implementation of web applications and, consequently, many attacks target those vulnerabilities to execute malicious and unauthorized operations[1].

Here we focus on a particular type of vulnerabilities of web applications – those whose origin is the *workflow* of the applications. Roughly speaking, each application can be associated with a workflow logic that captures the "correct" order of executions of steps that the user needs to perform in order to complete some transaction, e.g., pay before checkout. *Workflow attacks* exploit ill management of the workflow logic (and e.g., may allow to checkout before paying). Ideally, one would think of a *workflow logic* as the high-level description of the intended workflow, and the *actual workflow* as the workflows practically allowed by the application. Of course, one would wish for them to be the same, however, in practice, they could differ, which renders them vulnerable to security attacks based on this incompatibility.

Workflow attacks are enabled when the design of the application does not strictly enforce the intended workflow logic and permits violation of the behavior assumed by the developer[2]. For example, an attacker may be able to skip a required step and avoid providing with payment information on a checkout process, and directly purchase an item without ever paying for it. This could happen when the application does not not enforce proper restriction policies on the navigation, thus allowing an attacker to violate the intended workflow.

Typically, to defend against workflow attacks, a web application is written so that every possible entry point to the application enforces the developer's intended workflow logic. Each time the application receives an HTTP request, it checks if the request indeed follows the intended workflow. Usually, a module stores the current navigation state, e.g., whether or not the current user has logged in or has already visited a certain page, and then other modules use this information to deny or authorize access to other parts of the application. In addition, the application needs to check the correctness of the data carried on by the HTTP request, so that the it can be properly processed and the navigation can move forward.

Unfortunately, to date, there are no algorithmic solutions to enforce worflows logic in web applications[4]. The implementation is arbitrary left to the developer, who needs to take care of the vast number of scenarios and variables that may render a manual implementation cumbersome and error-prone. We identified the following issues relating to workflows enforcements:

1. Having no methodical ways for specifying or enforcing workflows logic entails their enforcements in an ad-hoc manner and having it tightly intertwined with the application logic in the implementation.
2. Having both application and workflow logics intertwined in the implementation renders the maintenance of the code less manageable.
3. This intertwining may obfuscate redundancy in implementation of workflow logic, causing lack of consistency and potential vulnerabilities when similar components are not identified as such, and modifications of them are not consistently applied.

To address these issues, we present *Workflower* — a framework that allows the developer to easily specify and automatically enforce the intended workflow logic. *Workflower* enables to separate the specification of the workflow logic from the application logic. *Workflower*, allowing for a declarative specification of a workflows logic, makes it easy to understand and maintain. The goals behind the design of *Workflower* are:

1. Providing the developer a clear and declarative way to specify the workflow logic and its intended interaction with the application logics.
2. Starting from the specification of the workflow logic, *Workflower* automatically synthesizes its implementation and enforcements in the web application.
3. *Workflower* allows for application logic to be modified while enforcing the same workflow logic, as well as for workflow logic to be modified without altering the application logic.

Thus, our approach consists of the developer writing the specifications to define the workflow logic, by specifying sequences of steps that need be performed to complete a transaction. *Workflower* automatically enforces those requirements by guaranteeing, with each HTTP request, that all preconditions are met and no violations occur.

We conclude the introduction with a simple example of the use of *Workflower*.

Checkout

Consider the checkout transaction workflow logic depicted in the solid lines of Figure 1. The dashed line in the figure describes a workflow that should be prevented. An example

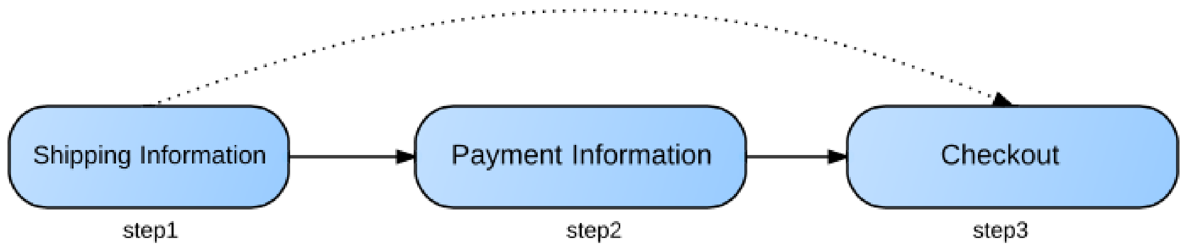

Figure 1: Checkout process example.

of typical implementation is described in Listing 1.1, where each step checks whether the right sequence of actions occurs before it performs validates the input and performs its operation.

Listing 1.1: Typical approach

```
#step 1
if user_loggedin
        set_visited(step1)
        display shipping_form
else
        redirect_to login_page
end
#step2
if is_visited(step1) and valid_shipping_data
        set_visited(step2)
        display payment_form
else
        redirect_to step1
end
#step3
if is_visited(step2) and valid_payment_data
        set_visited(step2)
        checkout_procedure
else
        redirect_to step2
end
```

Note that the code of Listing 1.1 is redundant – each step has the flavour ”if previous step was taken and the input is valid then goto the next step else redirect somewhere else” – the application logic (preparing pages for display) is not clearly separated from the workflow logic.

Listing 1.2 shows how an equivalent code is described with *Workflower*. Note that the code provides for better readability and maintainability. The declarative specifications allow to easily define an ordered sequence of steps, enriched by proper requirements, which is then automatically enforced in the application.

The rest of the dissertation is organized as follows: In Chapter 2 we describe of current methodologies for enforcement of workflow logics in HTTP, as well as provide with a short introduction to Ruby-on-Rails (RoR) on top of which *Workflower* is built. Chapter 3 de-

Listing 1.2: *Workflower* example

```
#workflows specifications
checkout_process:
        step1 : requires (user_logged_in), redirect_to login_page
        step2 : requires(valid_shipping_data)
        step3 : requires(valid_payment_data)

#step 1
        display shipping_form
#step2
        display payment_form
#step3
        checkout_procedure
end
```

scribes in detail our approach. Chapter 4 describes the implementation of *Workflower* and discusses some of its intricacies. Chapter 5 presents an evaluation of *Workflower*. Chapter 6 summarize the related literature, and we conclude in Chapter 7.

CHAPTER 2

BACKGROUND

2.1 Enforcing workflows through HTTP: Where we are

A workflow defines a set of sequences of steps that the application expects the user to perform in order to complete some transaction. The sequences describe a dependence between the next step on the previous ones. The *stateless*[6] nature of the HTTP protocol that treats each request as independent of prior ones makes the implementation of workflow logic in HTTP non trivial. To address the lack of a state and the independence between HTTP transactions, two main expedients can be used, namely *Session variables* and *HTTP Parameters*.

Session variables are used to overcome the stateless behavior of the HTTP protocol by storing information associated to the user current session and allows the server to follow necessary details about the transaction. Session variables are only alive as long as the session exists and they are invalidated as soon as the user leaves the application, by closing the browser, for example.

HTTP Parameters[7] are data used to provide additional information to an HTTP request as to enable an exchange of informations between multiple HTTP requests. They exist as either GET parameters that are appended to the URL, (e.g., `www.myhost.com/path/to/page?parameter=value`) or as POST parameters, that are enclosed in the body of the

HTTP request. For example of a POST request with parameters see Listing 2.1. The most common use is gathering data from the user and exchanging it between pages.

Listing 2.1: POST parameters example

```
POST /somepage.php HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 27

param1=value1&param2=value2
```

The common approach used to enforce intended workflow logic is to combine the use of session variables and HTTP parameters with the proper validations[3]. One of the typical techniques for doing so is to store ad-hoc values in the session variables so that the application can tell whether the user visited a certain page.

Listing 2.2 shows a toy example of this approach, where the workflow consists of a simple sequence of two pages, *page_one* and *page_two*, that must be visited in order.

Listing 2.2: Typical workflow example #1

```
# request page_one
session['page_one_visited'] = TRUE
#request page_two
if session['page_one_visited'] == FALSE
        redirect_to page_one
else
        display page_two
end
```

A more realistic is displayed in Listing 2.3. It shows a fragment of an application that enforces that the user has logged-in in order to access certain pages. When this condition is not met, the user is directed to the login page, from which the user will be returned to

Listing 2.3: Login example

```
# login page
login procedure
if session['pending_request'] is_set
        redirect_to session['pending_request']
# page_one
if user_not_logged_in
        redirect_to login_page
        session['pending_request'] = page_one
else
        display page_one ...
end
#request page_two
if user_not_logged_in
        redirect_to login_page
        session['pending_request'] = page_two
else
        display page_two ...
end
```

the originally requested page. This example can also be used to demonstrate the amount of redundant code and checks needed to enforce proper workflow logic even for a small fragment of an application. It is to be anticipated that this overhead is significant for "real-life" applications such as often encountered in e-commerce.

2.2 Ruby on Rails

While the concepts and architecture for our approach can be applied to a variety of Web development frameworks, *Workflower* is built on top of Ruby on Rails and leverages some of its features. Ruby on Rails (RoR)[8] is a web application development framework written in the Ruby programming language. The goal of the framework is to make web programming easier, faster, and more productive. It comes standard with various sane defaults, assumes various conventions that users must also follow, embraces the RESTful approach[15], highly encourages DRY or "Don't Repeat Yourself", and it is fundamentally based on the Model

Listing 2.4: Controller example

```
class LogicController < ApplicationController
        def index
                #do stuff e.g. fetch the database, process the parameters and so on
                render "index" #render the index view template
        end
        def contacts
                #do stuff
                render "contacts" #render the contacts view template
        end
end
```

View Controller (MVC)[16] architecture, which is a software architecture model that separates the concerns between the representation of information (the Model), the interaction with that information (the Controller), and its presentation (the View).

The basic idea of the MVC pattern adopted by RoR is that each page's request has to be served by a specific controller. More specifically, each HTTP request is handled by a function defined within a controller, called an *action*. Each action takes an HTTP request as input and returns a web page. An example of a simple controller is shown in Listing 2.4. Each action of a controller is in charge of executing the logic of the associated application, possibly by fetching the database or processing the request parameters, and dynamically rendering the template of the page. From hereon, we will refer to an action a on controller c as $c\#a$. For example, we refer to the action *index* of the controller *LogicController* in Listing 2.4 as *logic_controller#index*. RoR relies on the *router* to map, consistent with the developer's specifications, each URL request to the proper *controller#action*. An example of some router entries are shown in Listing 2.5

Listing 2.5: Router example

```
    match '/' => 'logic_controller#index'
    match '/contacts' => 'logic_controller#contacts'
```

These entries specify that an incoming request to the home page is served by the action *index* of the controller *LogicController*, while a request to the contacts page is served by the action *contacts* of the controller *LogicController*. Figure 2 presents an overview of how RoR handles requests.

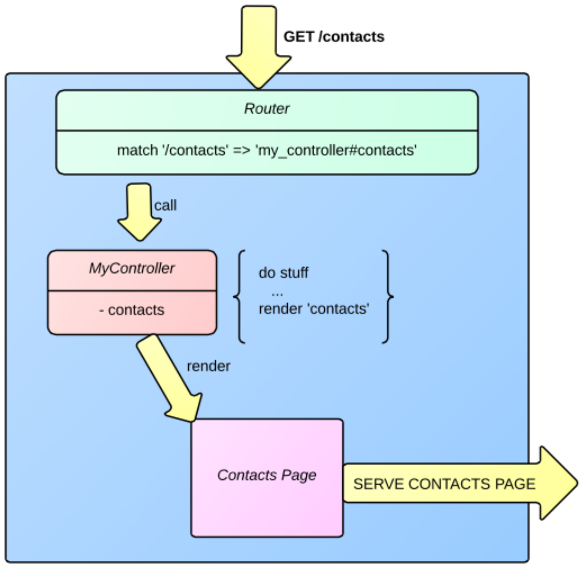

Figure 2: Ruby on Rails request handler overview.

CHAPTER 3

APPROACH

This Chapter describes the notion of workflow logic, how it can be described independently of the application logic and how it provides sufficient expressiveness so to represent a large number of use-cases. We further describe how the workflow logic is enforced by *Workflower* in the application, with all its challenges and solutions.

3.1 Workflow Specification

A Web application can potentially allow the user to navigate through any existing Web page in an unconstrained order. We define *navigation path* as an unconstrained sequence of pages that the Web application allows to visit, where each page represents a *step* of the *navigation path*. A *workflow* is a special category of navigation paths and describes a constrained sequence of steps that the developer wants the user to perform in order to complete some transaction. The constraints may enforce a particular order on the sequence of steps accessible by the user and require conditions about requests parameters or the session. A *workflow logic* is the set of all the workflows associated with a web application, and it is clearly a subset of the set of navigation paths.

Workflower allows to define workflows logic through the *workflows specification*, which permits to list all the workflows allowed by the developer. Each workflow can be described by enumerating all the steps that it consists of, as follows:

$$workflow_x = (step_0, step_1, step_2, ..., step_{n-1}, step_n)$$

where each step is mapped to a web page

$$step_i = page_k | page_k \in \textit{\{set of existing web pages\}}$$

and the i-th step of a workflow is the step that occupies the i-th position in the sequence

$$workflow_x[i] = step_i$$

This specification demands the navigation to follows the order such that:

$$\forall i, 0 \leq i \leq n, access(workflow_x[i]) > access(workflow_x[i-1])$$

where the predicate *access($step_i$)* describes the action of visiting the page mapped to $step_i$ and the relation $>$ act as *"immediately after"*. Thus, the semantic of the specification translates in

"access $step_1$ then $step_2$ ... then $step_n$"

Moreover, if we define the following predicates:

$ACCESSED(step_x)$ = *TRUE* ***iff*** *the $step_x$ has been accessed*

$ACCESSIBLE(step_x)$ = *TRUE* ***iff*** *the $step_x$ is allowed to be accessed*

$$ACCESSED(step_x) \implies ACCESSIBLE(step_x)$$

we can claim, from the description above, the following property:

$$\forall i, 0 < i \leq n, ACCESSIBLE(workflow_x[0]) \wedge (ACCESSIBLE(workflow_x[i]) \implies ACCESSED(workflow_x[i-1]))$$

This property guarantees that the user will reach a certain step only after having gone through all the preceding steps, thus respecting the defined order.

Furthermore, workflows may not only requires the user to navigate through the web pages in a particular order, but to satisfies some conditions related to the requests parameters or the session. Thus, we enrich the definition of step with a set of conditions that needs to be verified before accessing the step:

$$workflow_x[i] = step_i = (page_k, conditions_i)$$

where $conditions_i$ is a boolean expression containing conditions on requests parameters or session variables. This description states that the user needs to fulfill the requirements described by $conditions_i$ when trying to access $page_k$, mapped to $step_i$ of $workflow_x$. From this, we adapt the property above such that:

$$\forall i, 0 < i \leq n, ACCESSIBLE(workflow_x[i]) \implies$$
$$(conditions_i \land ACCESSED(workflow_x[i-1])) \land ACCESSIBLE(workflow_x[0])$$

Moreover, the specification allows to define an action to be triggered in case the conditions are not satisfied. As a matter of facts, when the user lacks of some requirements, a common solution in practice is to redirect the user to a more sensible page, which is in charge of fulfilling those requirements. Thus, each step allows the developer to dictate what happens when the user violates the requirements. Currently we support redirection to another workflow as action to be triggered in case of a requirements violation and it can be defined as:

$$workflow_x[i] = step_i = (page_k, conditions_i, redirect\_to(workflow_y))$$

which states that if the user tries to access $page_k$, mapped to $step_i$ of $workflow_x$, and fails in fulfilling the requirements described by $conditions_i$, then it will be redirected to the firs step of $workflow_y$. Lastly, the specification also allows to create *multiple instance workflows*. With this term, we refer to workflows that can be entered, or *instantiated*, multiple times and coexist during the same session. The multiplicity of the instances is due to the fact that the user is allowed to access a workflow many times in different tabs or windows of a browser, inside the same session, thus creating more coexisting instances of the same workflow. Furthermore, one of the things we have discovered is that different behaviors may be needed in this situation, i.e. some workflows are qualitatively different than others in terms of the number of instances of the workflow any given user is supposed to be involved in. For example, typically a login workflow is one where only a single instance is permitted: there can not exist coexisting multiple instances of a login process since the state of the user, which the workflow will eventually modify, is unique. In contrast, assume an application that allows the user to upload pictures through a workflow-based process. The user may want the chance of opening multiple tabs on the browser at the same time and initiate the submission process on each of them to upload different pictures at once. Consequently, the specification allows a developer to dictate whether a given workflow can be instantiated multiple times or just once using the keyword *multiple*.

To sum things up, the workflow specification allows to:

- Define the workflow logic, i.e. list all the accepted workflows.

- Define a workflow as a sequence of steps that the navigation must follows.
- Define a step as a Web page, with conditions that needs to be verified before accessing it, and with an action to be triggered when the conditions are not met.
- Define Multiple Instance Workflow

3.2 Workflow Enforcements

Workflower must ensures that the workflow logic, described in the specification, is correctly enforced in the application and that the navigation proceeds accordingly. Thus, to guarantee that the navigation is consistent with the specification and no violation occur, we make sure that the user can visit the requested page only if the step mapped to the page is accessible. Accordingly to what we defined in Section 3.1, the accessibility of a step is defined by the property:

$$\forall i, 0 < i \leq n, ACCESSIBLE(workflow_x[i]) \implies$$
$$(conditions_i \wedge ACCESSED(workflow_x[i-1])) \wedge ACCESSIBLE(workflow_x[0])$$

Consequently the enforcement is accomplished by ensuring that this property holds at every request, with the following algorithm. We keep track of the last step accessed by the user, as:

$$accessed\_step = (workflow_x[i])$$

Furthermore, each time a request for $page_z$ arrives:

1. Fetches the step $step_k$ of some workflow $workflow_y$ associated with the requested page such that

$$step_k = (page_z, conditions_k, redirect\_to(workflow_j))$$

2. Checks whether the step $step_k$ if accessible

   - Check the conditions $conditions_k$
   - If $k > 0$ check whether the previous step has been accessed

$$k = i + 1 \land x = y | accessed\_step = (workflow_x[i])$$

3. if accessible, update the accessed step and let the user visit the page

$$accessed\_step = (workflow_y[k])$$

4. if not accessible, redirect to the page mapped to the first step of the specified workflow $workflow_j$

By applying this algorithm to each incoming request, we ensure that the navigation is consistent with the workflow logic.

3.3 Workflow Enforcement - Challenges

We now describe the main challenges related to the workflow enforcement.

3.3.1 Accessed Step

In the enforcement approach described above, we are assuming that once a step is accessible, the corresponding page can be visited and the step can be considered as successfully accessed(phase 3 of the enforcement algorithm). Nevertheless, we need to clarify that in practice, once the user request a page, some process takes care of generating it and subsequently returning it to the user to be displayed. Thus, assuming that an accessible step can be immediately considered as accessed, would mean considering that the process that generates the page always succeeds in its task. On the contrary, some event, that would make the process deviates from its expected path, could still occur in this stage, namely some internal error, e.g. SQL exceptions, or logical error. In this case, even if the execution of the process terminates, the step should not be considered as successfully accessed because of the errors that occurred. Assume, for instance, a sign-in process, whose last step requires the user to submit a username which is going to be stored in the database if it has not been taken already. From the point of view of *Workflower*, once the requirements are fulfilled, e.g. the username has been chosen and submitted, and the action gets executed, it would be impossible to distinguish a successful registration, which ends with an update of the database, from an unsuccessful one, where the database can not be update, because, for instance, the username has already been chosen. That is because the action will terminate in both case, but the two outcomes will be very different. Under those circumstances, the main issues is how to distinguish a correct execution from an invalid one, since in presence of an incorrect execution the step can not be considered as accessed.

3.3.2 Concurrent workflows

One of the challenges related to the design of *Workflower* is about how to allow the user to be concurrently involved in several workflows during the same session. Despite the fact that the session is unique, a user may be enrolled in multiple worklows at the same time. As a simple real use case, assume the user is submitting a workflow-based survey on a tab of the browser, and at the same time he is involved, on another tab, in a checkout process in the same application. It could be the case that the user may want to moves forward on a workflow and then switches to the other one, without completing the first, and keep moving back and forth as he prefers. Without a proper approach, when the user switches from a workflow to another, the state of the left workflow will be lost and he would need to start over in case he tries to resume it.

3.3.3 Ambiguity

While parsing the workflows specification, ambiguities may arise when the same page appears on different workflow, as shown in Figure 3 (since our work is based on the RoR environment, each page is represented by a *controller#action*). The given specification

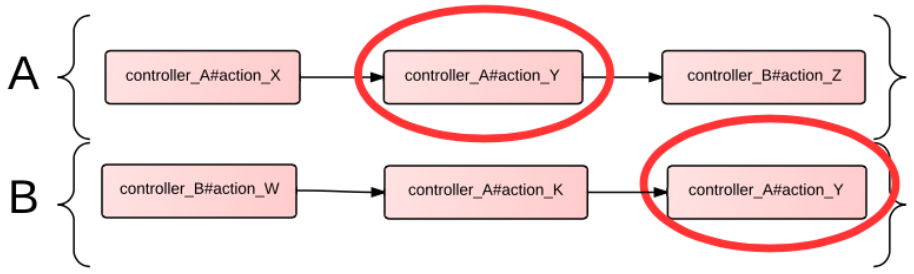

Figure 3: Ambiguos workflow.

describes two workflows which share the same web page for different steps. This would be the case where a page could perform multiple actions, depending on the request parameter for instance. Notice that nothing is specified about the requirements of the steps, meaning that the two ambiguous steps may have different requirements, even if they share the same page. To better understand where the ambiguity hides, assume a request for a page mapped to *controller_A#action_Y* occurs and the specification is the same as shown on Figure 3. *Workflower* need to verify the proper conditions and behave accordingly, but without further expedients, we could not tell if the request is addressed to *workflow A* rather than to *workflow B*, meaning that we could not decide whether to check the requirements for $step_2$ of $Workflow_A$ or for $step_3$ of $Workflow_B$.

More generally, there exists two steps $step_i$ and $step_j$ of two different workflows, $workflow_x$ and $workflow_y$, with different conditions, $conditions_i$ and $conditions_j$, mapped to the same page $page_k$.

$$workflow_x[i] = step_i = (page_k, conditions_i, ..)$$

$$workflow_y[j] = step_j = (page_k, conditions_j, ..)$$

With this specification we do not know which requirements needs to be checked when a request for $page_k$ arrives, which means that we can not guarantee that the navigation will be consistent with the workflow logic.

3.3.4 Multiple Instance Workflows

As described in Section 3.1, we want to allow the developer to create *multiple instance* workflows, namely workflows that can be instantiated multiple times and can coexists during the same session. The problem here is how to make each instance independent of each other, since we do not want the state of one instance to be influenced by the state of a different instance. In other words, a request for a certain page, mapped to a certain step of a multiple instance workflow, must be treated differently whether it belongs to one instance or another. Clearly, without further modification, this would be unfeasible, since an HTTP request to a certain page would be indistinguishable to another request to the same page.

3.3.5 Redirection and Resume

The challenge about the redirection behavior we discussed in Section 3.3.4 comes with the addition that we want to resume, at a proper time, the original request of the user, meaning that once he completes the workflow he has been redirected to, he should be brought back to the page he requested at first. In order to accomplish this, it will be needed a smart and aware navigation tracking system, able to detect whether it is the right time to resume the user original request.

3.4 Workflow Enforcement - Solution to Challenges

Here we describe the solutions needed to solve the challenges we defined on Section 3.3.

3.4.1 Accessed State

As described in Section 3.3.1, once the requirements for a step are met and the process that takes care of generating the web page gets executed, we need a way to recognize whether the execution ends in a successful way or not. Since it would be impossible to *Workflower* to analyze all the possible paths that the code could follow and eventually realize whether the execution ends with a positive outcome or not, we introduced a way that allows the developer to exchange information with *Workflower* and suggest to it whether to consider the action as unsuccessfully execute, namely some internal or logic error occurred. Considering that, once the constraints are satisfied and no information from the developer suggests differently, we assume the step as effectively executed.

$$ACCESSED(step_i) \implies ACCESSIBLE(step_i) \wedge \neg(INTERNAL\_ERRORS_i)$$

3.4.2 Cuncurrent Workflows

To let multiple workflows be active at the same time, it was necessary to extend the bookkeeping component to trace the user navigation to a more advanced level. As a matter of facts, without a proper tracking, when the user switches from a workflow to another, the state of the left workflow will be overwritten by the new one. For this reason, we defined, for each active session, a set :

$$active\_workflows = \{\}$$

that we use to keep track of the active workflows, i.e. all the workflows where the user is currently enrolled. More specifically, as soon as a request for a page occurs *Workflower* parses the workflows specification as follows:

1. Fetches the step $step_i$ and $workflow_x$ associated with the requested page
2. Checks whether the step if accessible, i.e. if its constraints are satisfied and whether active_workflows contains $workflow_x[i-1]$, in case $i > 0$.
   (a) If it is accessible, allows the process execution to begin
      i. If it is accessed successfully, namely no internal errors occurred, updates the set of active workflow

$$active\_workflows = active\_workflows \cup \{(workflow_x[i])\}$$

If there exists an entry $workflow_x[i-1]$, removes it, so that only the last step of a workflow is tracked.

ii. If some internal errors occur, does not update the set of active workflows.

(b) If it is not accessible, denies the request, i.e. does not let the process executes and does not update the set of active workflows.

By the end of this process, at every request, the navigation tracer will be aware of any possible steps of any possible workflow that the user may be active in.

3.4.3 Ambiguity

In Section 3.3.3 we broadly described how ambiguities may arise from the workflows specification. More specifically, we identified three base types of ambiguities that could happen:

1. *Ambiguity between initial steps.* Two workflows share the same initial step, as show in Figure 4

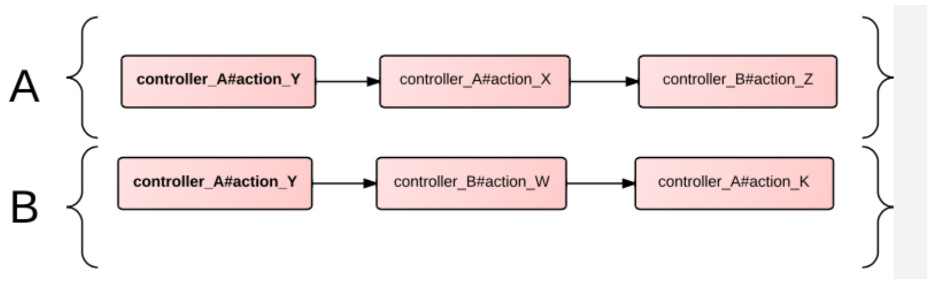

Figure 4: Ambiguity between initial steps.

2. *Ambiguity between non-initial steps.* Two workflows share one step, which is not initial, as show in Figure 5

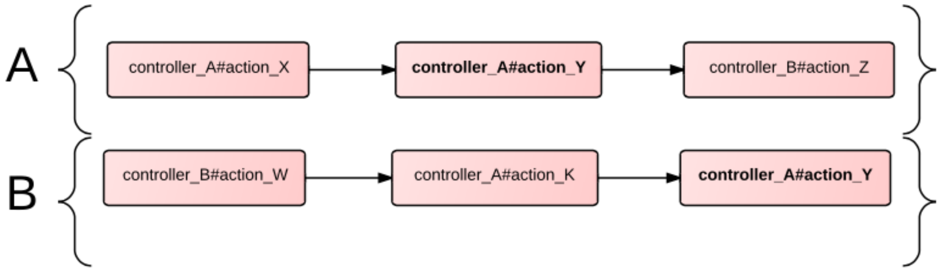

Figure 5: Ambiguity between non-initial steps.

3. *Ambiguity between an initial step and a non-initial step.* Two workflows share one step, which is initial in one workflow and non-initial in the other, as show in Figure 6

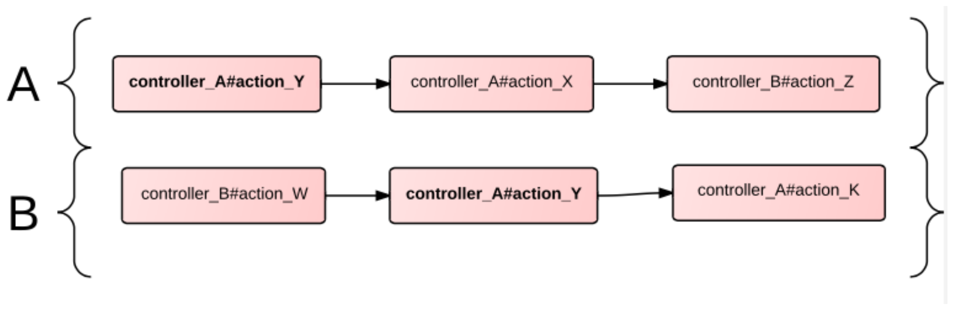

Figure 6: Ambiguity between an initial step and a non-initial step.

These three base kinds of ambiguities can be combined and extended to more than two workflows or more than a shared step to create any kind of complex ambiguous workflow. Solving these ambiguities by precisely identifying to which workflow the request is addressed is not feasible, since the request does not contain information that may suggest which workflow it is referring to and, furthermore, we would not be able to tell which step the user will move to after the current request. However, we aim to prove that this is not an issue and that the navigation constraints, described by the workflow logic, are preserved even in the presence of ambiguities. As a matter of facts, the workflow logic is enforced in such a way that the user can visit the requested page only if the step mapped to the page is accessible. Consequently, in case of ambiguities, the navigation can be consistent with the workflow logic only if there exists at least one step whose condition are verified. To make this principle holds, we had to extend the fetching phase (phase 1 at page 22) of the multiple workflow process showed in Section 3.4.2, so that every step mapped to the page is retrieved. Consequently, if at least one step is accessible, the user is allowed to visit the page. Furthermore, if more than one step is successfully accessed, we consider each of them as active. In other words, in case the accessed step is shared among several workflows, we consider all of them as active. This assumption is needed to preserve the navigation consistency with the workflow logic.

Thus, the ambiguities are solved according to the following algorithm:

1. In presence of *ambiguity between initial steps*, when a request for the ambiguous page occurs, we fetch all the initial steps mapped to the requested page, check the requirements for each of them and update the active workflows accordingly.

   Assume the specification is the same as the one shown in Figure 4 and a request for *controller_A#action_Y* occurs. Therefore, if the requirements are fulfilled for both the initial steps of $workflow_A$ and $workflow_B$ and the step is successfully accessed, namely no internal errors occur, the tracker will update the set of active workflows as:

   $$active\_workflows = \{workflow_A[0], workflow_B[0]]\}$$

   and the execution of the action will be started. If for instance only the requirements of $step_0$ of $workflow_A$ are fulfilled, the active worflows would be:

   $$active\_workflows = \{workflow_A[0]\}$$

   If none of the initial steps has its requirements satisfied, then none of the step is accessed, the request is denied and the execution of the action will not be started.

2. In presence of *ambiguity between non-initial steps*, when a request for the ambiguous page occurs, we fetch only those steps, mapped to the requested page, belonging to active workflows, check the requirements for each of them and update the active workflows accordingly.

   Assume the specification is the same as the one shown in Figure 5 and a request for *controller_A#action_Y* occurs and the active workflows are as:

$$active\_workflows = \{(workflow_A[0])\}$$

Therefore, we fetch only $step_1$ of $workflow_A$, since $workflow_B$ is not active at the moment, and we process the step with its requirements as before, meaning that if at least one step has its requirements satisfied then the step is accessed and the action is allowed to execute. On the other hand, if the active workflows were as:

$$active\_workflows = \{(workflow_A[0]), (workflow_B[0])\}$$

we would have fetched both the steps belonging to $workflow_A$ and $workflow_B$ and check both the requirements. Notice, in this case, that while $step_1$ of $workflow_A$ has a chance to be accessed, $step_2$ of $workflow_B$ will not be accessed since $step_2$ of $workflow_B$ has not been visited.

3. In presence of *ambiguity between an initial step and a non-initial step*, when a request for the ambiguous page occurs, we fetch only those steps, mapped to the requested page, that are <u>initial</u> steps for <u>any</u> workflow and that are <u>non-initial</u> steps only for <u>active</u> workflows, then check the requirements for each of them and update the active workflows accordingly.

   Assume the specification is the same as the one shown in Figure 6 and a request for *controller_A#action_Y* occurs and the active workflows are as:

$$active\_workflows = \{workflow_B[0]\}$$

Therefore, we need to fetch both $step_0$ of $workflow_A$, since it is initial, and $step_1$ of $workflow_B$, since it belongs to an active workflow. Afterwards, we process the steps as we described in the previous cases.

To sum things up, we handle the ambiguity issue, not by trying to identifying one single step of a single workflow, but instead by checking whether at least one step is accessible. Moreover we recognize as active all the shared steps that are successfully accessed. Doing so, we are assuring that the navigation is consistent with the workflow logic, since if at least one step can be accessed then the request would be accepted, while it would be denied otherwise.

Beside the ambiguities discussed so far, there exists another kind of ambiguity that we did not addressed yet. It might be the case where a specification like the one shown in Figure 7 could be needed, where multiple steps belonging the same workflow share the same page. An example of real application could be a questionnaire application that

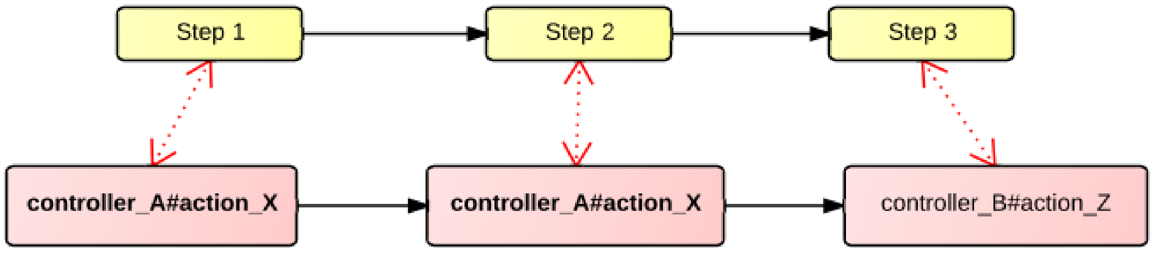

Figure 7: Example of ambiguous workflow with repeated pages.

uses the same page to dynamically display the questions, based on the request parameters which specifies the range of questions to be displayed, as shown in Figure 8. With the

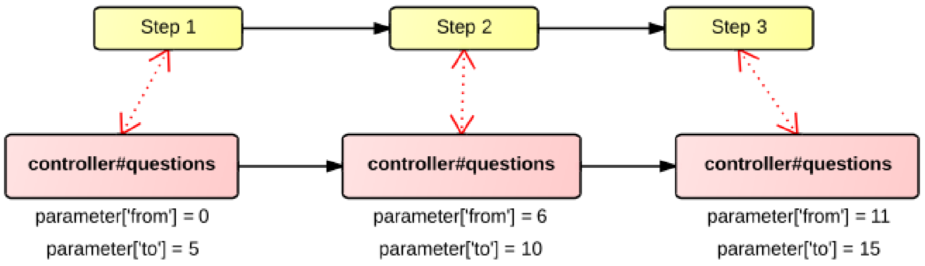

Figure 8: Example of ambiguos workflow for a questionnaire application.

approach presented so far, it would be unfeasible to handle properly each request since identifying a step simply according to the mapped page would lead to an uncontrolled behavior. While we did not implemented a solution for this issue, which we consider an isolated case not much relevant to our study, we believe that it could be easily managed by extending the proposed approach so that each step would be identified not only by the page but also by the request parameters it expects. Doing so, from the previous example, we could identify, for instance, the first step as

$$step_1 = (\textsf{controller\#questions}, \{param_{from} = 0, param_{to} = 5\})$$

which would be different from the second step, being

$$step_2 = (\textsf{controller\#questions}, \{param_{from} = 6, param_{to} = 10\})$$

so that any ambiguity would be solved.

3.4.4 Multiple Instance Workflows

To address the issues related to *Multiple Instance Workflows* described in Section 3.3.4, we provided a way to distinguish different instance of the same workflow, based on the use of *tokens*. The high level approach is to associate each instance of a workflow with a token and embed that token into every request so that we could identify each time the proper instance. Specifically, each time an initial step of a multiple instance workflow is accessed, we generate a token, unique in the context of that workflow, and we inject it into the output page (details about the implementation will follow on Chapter 4) so that it will be carried on every subsequent request coming from that page. Therefore, we extend algorithm as follows:

1. Each time an initial state $step_0$ is accessed and a token t_x is generated, update the active workflows set as follows:

$$active\_workflows = active\_workflows \cup \{(workflow_x[0], t_x)\}$$

2. Each time a non initial state $step_i | i > 0$ is accessed and the request comes with the token t_z, update the active workflows set, as follows:

$$active\_workflows = active\_workflows \cup \{(workflow_x[i], t_z)\}$$

   removing the entry $(workflow_x[i-1], t_z)$

Moreover, we need to extend the conditions of the steps to require that the token received with a request is valid and associated with an existing workflow instance :

$$ACCESSIBLE(step_i, t_x) \implies ACCESSIBLE(step_i) \wedge VALID(step_i, t_x)$$

where

$$VALID(step_i, t_x) \iff \exists w | (w[i-1], t_x) \in active\_workflows$$

As a last expedient, we need to handle tokens in case of ambiguities. As a matter of facts, suppose there exists multiple instance ambiguous workflows and a request for a shared page occurs. Assume for simplicity we are dealing with the first type of ambiguity among initial steps (see page 23). In this scenario we would have the same page with different generated tokens and only one token to render as output. In order to remove this kind of ambiguities, we modify our approach by making sure that each page is associated with exactly one token, even if it is shared by different workflows. This means that every time we need to generate a token for a page, we check whether there exists any token already associated with that page and we assign that same token in case it does. For instance, assume the specification given in Figure 9 and the currently active workflows as:

$$active\_workflows = \{(workflow_B[0], t_x)\}$$

Now assume that a request for *controller_A#action_Y* occurs with token t_x and that the requirements are satisfied for all $step_0$ of $workflow_A$, $step_1$ of $workflow_B$ and $step_0$ of $workflow_C$. The active workflows will be then updated as described before, leading to:

$$active\_workflows = \{(workflow_B[1], t_x), (workflow_A[0], t_x), (workflow_C[0], t_x)\}$$

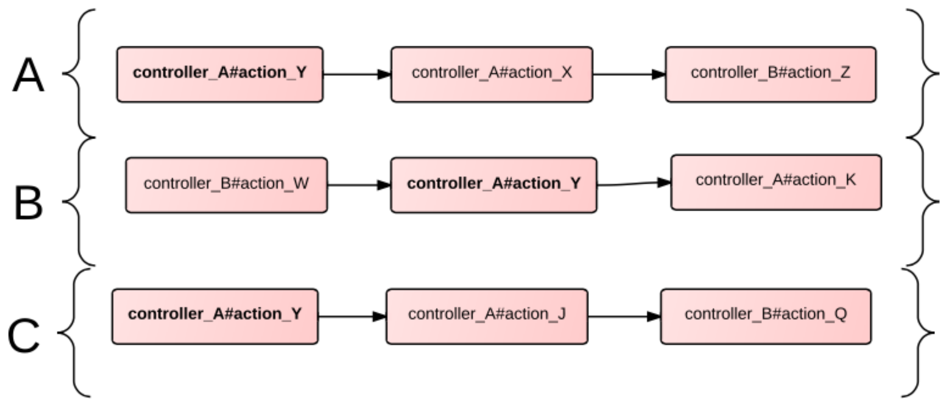

Figure 9: Example of ambiguos multiple instance workflow

3.4.5 Redirection and Resume

In order to support redirection and request resume, we extend our approach so that every time a step is not accessible, the user gets redirected to the workflow described in the specification. We refer to this workflow as *supporting workflow*. Moreover, a *pending request* gets allocated so that, once the *supporting workflow* is entirely executed, the original request will be resumed.

Specifically, every time a redirection occurs, a *pending request* is created as follows:

$$pending\_requests = pending\_requests \cup (resumable\_workflow, supporting\_workflow)$$

Furthermore, every time a final step of a workflow is successfully accessed, *Workflower* checks whether there are any pending request associated with that workflow, so that the user gets redirected back to the proper *resumable_workflow*.

Lastly, in case of ambiguities, which occur whether a *controller#action* is shared by different steps and there exist <u>more than one</u> step with unfulfilled requirements, we could end up with multiple supporting workflows where the user could be redirected to. To solve this issue, we allow the developer to specify a *default action*, namely a *default support workflow*, that will be triggered whether the described ambiguity occurs.

CHAPTER 4

IMPLEMENTATION

In this section we are going to present the most interesting details about the implementation of *Workflower*, how it is integrated in the RoR environment and the API we make available to the developer in order to write workflows specifications.

4.1 Extending Ruby on Rails

Workflower is implemented in Ruby and consists of an extension of Ruby on Rails. We chose it because we believe that RoR is a really complete mature framework, widely spread and used by major brands[1] (e.g. *Twitter, Github, Groupon, Yellow Pages, Hulu, Shopify*) and rich of features that we leveraged and that allowed us to easily implement all the functionalities that *Workflower* provides. Ruby on Rails is by design a deeply customizable framework, born under the DIY or "Do It Yourself" philosophy, that easily allows third developers to add new features and integrate them as plugins, or *gems*[9] in Ruby terminology, that are automatically loaded with the RoR infrastructure and that can fully exploits all of its functionalities. Thus we build *Workflower* as a Ruby gem, so that it can be easily deployed and integrated in any RoR application.

[1] `http://rubyonrails.org/applications`

4.2 Workflows Specifications with *Workflower*

Workflower allows the developer to enforce workflows with a very little effort. The workflows specifications must be written in a dedicated file *'workflows.rb'* automatically created into the RoR application's configuration directory that it is loaded once the application is launched. The syntax required by *Workflower* to describe a specification is shown in Listing 4.1

Listing 4.1: Workflows specification syntax

```
1   workflow "workflow_name", multiple_instance:(true || false) do
2           step conditions: {"boolean_expression"} , page: "controller#action",
                redirect_to_wf:'some_workflow'
3           step conditions: {"boolean_expression"} , page: "controller#action"
4           ...
5   end
```

With the code shown in Line 1 a workflow named *"workflow_name"* is created by the keyword *"workflow"* with the optional parameter *"multiple_instance"* that allows the developer to specify whether the workflow can be instantiable multiple times. Lines 2 and 3 shows the syntax needed to describe a step: the keyword *step* creates a new step for the current workflows, fetching the requirements needed to access the step from the *"conditions"* parameter, where the developer can specify any kind of boolean expression exploiting any valid RoR API to access session variables (e.g. *session['some_variable']*) or request parameters (e.g. *params['some_parameter']*). Besides, the step description must contain the page mapped to that step, by providing the associated *controller#action*, through the *"page"* parameter. Optionally he could also specify whether the user should be redirect to some other workflows in case the *conditions* are not met, by using the *"redirect_to"* parameter. Furthermore, the sequence of steps is built according to the order in which the steps are specified.

4.3 Exploiting Ruby on Rails features

Instantiation

The first stage of a Ruby on Rails application lifecycle consists of the instantiation of the *ApplicationController*, that keeps running as long as the application lives and it is responsible to handle each incoming request by invoking the proper controller associated with the requested page. In order to have our workflows management system be included into the application so that it could fully interact with all the RoR environment, we exploited the marvelous metaprogramming nature of Ruby to extend the *ApplicationController* at runtime so that it would load the workflows specifications as soon as the application gets initiated and launched. The loading of the specifications consists of the execution of the configuration file, *'workflows.rb'*, described in Section 4.2, that will allocate and create the proper data structure that will be needed to correctly process the requests.

Data Structure

Once the specifications are parsed and loaded, data structures are allocated to represent the workflows component. All the existing workflows are stored in an array where each workflow is represent as a class, shown in Listing 4.2, containing an array of steps and attributes that characterize it. The *steps* arrays contains the steps that the workflow consist of and they are represented as the class shown in Listing 4.3.

Listing 4.2: Workflow class

```
@workflows = [:workflow_x,:workflow_y..]

Class Workflow
        @identifier = :workflow_x
        @steps = [step1,step2,step3 ..]
        @multiple_instance = false
```

Listing 4.3: Step class

```
Class Step
        @page = "controller#action"
        @conditions = "boolean␣condition"
        @redirect = "some_wf"
```

In order to efficiently interact with these data, we also provide some *index* structures that allows us to quickly identify and access the proper data. Namely, we define an *hashmap*, whose *key* is the page identifier, i.e. *controller#action*, while the *value* is a set containing the steps mapped to that page. Consequently, each time a request for some page occurs, we can immediately obtain the proper steps.

Listing 4.4: Indexes

```
@index["controller1#action2"] = {step_x,step_y}
@index["controller2#action3"] = {step_z,step_y}
```

All the data structures provided, namely the *index* and the *workflows*, are made available as *global variables* and they can only be read and not modified once they have been instantiated. Doing so, we make sure that if the application lives in a multithreaded environment, no race conditions or consistency issues arise.

Navigation State

In order to perform the proper requirements validations we need to track the navigation per user, meaning that we need to keep trace of the accessed steps for each user currently involved in using the application, by associating a set of *active workflows*, as we showed in Section 3.4.2. So as to accomplish that, we exploit session variables and we forge one that acts as the *active workflows* set. Thus, by accessing *SESSION[active_workflows]*, we can obtain the navigation tracker associated with the active user and succeed in following the user navigation correctly.

Listing 4.5: Navigation state

```

    session['active_workflows'] = {(:wf_x,:step_1),(:wf_2,:step_3)}
```

Runtime

Once *Workflower* is loaded into the system, it needs to interact with the requests handling system so that it can check if a page is allowed to be accessed depending on its requirements and behave properly according to the approaches described in Section 3.4. More specifically it needs to intercept a request as soon as it occurs, verify whether the requirements are met before allowing the execution of the *controller#action* and consequently keep track of the navigation. This behavior can be accomplished thanks to the many hooks[10], that RoR provides, around the request handling: more specifically we are interested in the hooks before the execution of the *controller#action*, i.e. *before hook* and after the execution, i.e. *after hook*. Thus, we exploit those hooks to implement our solutions:

- *Before Hook:* This hook gets called just before invoking the controller to start its execution. Thus we inject here the code that check whether the requirements are met and let the execution begins. Otherwise, we deny the controller to start its execution and we redirect the user to the proper workflow, in case it is needed. We show in Listing 4.6 a simplified version of the *before_hook*

  Listing 4.6: Before hook

  ```
  def before_hook(requested_page,token)
          mapped_steps = get_possibile_steps(requested_page)
          mapped_steps.each do |step|
                  if check_token(step,token) && check_conditions(step)
                          allowed_steps << step
                  else
                          denied_steps << step
          end
          if allowed_steps.empty?
                  if denied_steps.count > 1
                          AMBIGUOS DESIGN
                  else
                          redirect_to denied_steps.first.redirect
                  end
          else
                  mark allowed_steps as about to be accessed
          end
  end
  ```

  Furthermore, the function *get_possible_states* fetches the steps mapped to the requested page, according to the approaches described in Section 3.3.3 to handle ambiguities, by combining the use of the *index* and the *active workflows* structures.

- *After Hook:* This hook gets called as soon as the controller terminates its processing phase. As we described in Section 3.4.1, we need to know whether some internal errors occurred during the controller execution. In order to accomplish that, we make a global variable *WORKFLOWS_ERRORS* accessible to the developer so that it can provide us the needed information from inside the controller. Thus, in presence of no errors, we update the state of the user navigation to reflect the current step access and proceed with tokens management if necessary. Otherwise, we leave the state as it is and communicate the error to the user. We show in Listing 4.7 a simplified version of the *after_hook*

Listing 4.7: After hook

```
def after_hook(WORKFLOWS_ERRORS)
        if !WORKFLOWS_ERRORS
                mark allowed_steps as successfully accessed
                generate/update token if necessary
        else
                notify the user about the error
        end
end
```

Redirection

Furthermore, we want to briefly describe how the redirection to another workflows is accomplished, in case the conditions to access a step are not met. RoR provides a useful API, called *redirect_to*, which inject a *3xx Status Code*, which stands for redirection according to the HTTP standard, into the HTTP response header, so that the browser will proceed by requesting the proper page, which is specified by fetching the first step of the workflow where the user should be redirected to.

Multiple Instance Workflows

Lastly, we describe how we implement the solution for the multiple instance workflow shown in Section 3.4.4. The high level approach requires to identify each request with a token associate to a certain workflow, in order to distinguish among requests addressed to different instances of the same workflow. Firstly, we assume that a request is triggered by the user visiting a page through:

1. *Links:* the user requests the next page by clicking on a link contained in the currently visited page.
2. *Forms:* the user requests the next page by submitting a form contained in the currently visited page.

From this assumption, we make sure to embed the proper token, associate with the workflow whom the page belongs to, to each request caused by any of the two methods above. Thus, we embed the token with two strategies, simarly to the method described by Jovanovic et al.[11] to prevent CSRF attacks:

1. We append the token to each link's destination as a GET parameter, so that a link

   ```
                  <a href="next_page">
   ```

   becomes

   ```
                  <a href="next_page?wf_token=some_token">
   ```

2. We embed an hidden input to each form, as follows:

   ```
                  <input type="hidden" name="wf_token" value="some_token">
   ```

More specifically, we provide two options to accomplish this result:

1. We inject a javascript script into the final rendered page, just before the HTTP response is sent back to the user, by exploiting another RoR useful hook, *before_render*.

Thus, the script will run on the client and inject the token into every link and form that the parser detects.

2. We override two functions provided by RoR that helps the developer to quickly build links and forms, namely *link_to* and *form_tag*, so that the result includes the token.

Thanks to this approach we are sure that almost every request coming from a page will carry on the token needed to identify the correct instance of the workflow. Furthermore, the implementation could be extended to handle also situation where links and forms are dynamically generated by Javascript scripts, which are not yet supported by the current implementation. Beside that, this approach comes with a drawback, namely an overwhelming number of tokens are embedded in the page, even into links or forms that are not associated with a multiple instance workflow. To guarantee that this issue would not cause misbehaviors we provide a tokens tracking system that assure that the tokens are validated exclusively when the request is addressed to a page belonging to a multiple instance workflow and discarded otherwise.

Complexity

Lastly, we discuss the complexity of the algorithms we implemented. We built *Workflower* keeping in mind that the code gets executed each time an HTTP request occurs. Thus, we implemented it trying to affect the application performances as little as possible. Firstly, the specification gets loaded and the data structures gets allocated just once, as the application gets launched. Thus, it does not influence on the processing of a request. The *before_hook* gets executed at every request and it consists of:

- *Retrieving the steps mapped to the requested page:* $O(1)$, thanks to the *index* and *active workflows* structures that allow direct access.
- *Checking the requirements for each retrieved steps:* $O(n)$, where *n* is the number of retrieved steps, i.e. ambiguous steps. The requirements and token validation can be accomplished with $O(1)$ complexity, by using the proper expedients, e.g. *hashmaps*.

The *after_hook* gets executed after the controller execution and it consists of:

- *Updating the state for each retrieved steps:*$O(n)$, where *n* is the number of retrieved steps, i.e. ambiguous steps.
- *Token Management:* generating or updating tokens requires $O(1)$, since it is done by exploiting hashmaps as well.

Consequently, the overall complexity is $O(n)$, where *n* is the number of steps associated with the requested page. This means that ambiguity of the design will penalize the processing of a request.

CHAPTER 5

EVALUATION

To evaluate *Workflower*, we built a skeleton e-commerce application in Ruby on Rails, specified the intended workflows, observed the behavior of the application with and without the workflows being enforced, and measured the performance overheads of the workflow enforcement implementation.

5.1 E-commerce application and Workflow Specification

The application features some basic functionalities of an e-commerce platform. Namely, it consists of a simple catalog of products that the user can add to its own cart and eventually purchase through a checkout process. The checkout consists of a multiple step sequence, where the user is asked to provide shipment informations and payment details and eventually review and submit the order. The application also presents an administrative multiple step process that allows products to be added to the catalog, by providing the product details and a picture. Ultimately, login processes are provided to distinguish between regular users and administrator users. Namely, the checkout process can be accessed only by authenticated users, while the administrative process is accessible only to administrators. In addition, the login processes consists of multiple step, where the user is required to provide its credential and answer a security question.

As far as the RoR implementation concerns, we are going to show the most interesting parts of the application. Notice that the code shown as been simplified to allow an easier reading. Listing 5.1, 5.2 and 5.3 show respectively the skeleton of the *Checkout Controller*, responsible for the checkout process, the *Catalog Controller*, responsible for the catalog management, and the *Login Controller*, responsible for the login process.

Listing 5.1: Checkout Controller

```
    class CheckoutController < ApplicationController
     def valid_payment(payment)
      #check the validity of payment information
end
     def valid_shipment(ship)
      #check the validity of shipping information
 end
     def payment
      #ask for payment information
 end
     def shipment
      #process payment information
      #ask for shipment information
     end
     def submit
      #retrive payment information
      #submit the order
     end
    end
```

These controllers only takes care of the functional parts of the processes they are in charge of. Namely they provide methods to validate the user input, to store it and process the request.

Listing 5.2: Catalog Controller

```
   class CatalogController < ApplicationController
    def valid_details(details)
     #check the validity of the details
end
    def valid_picture(picture)
     #check the validity of the picture
end
    def details
     #ask for product details
end
    def picture
     #process product details
     #ask for product picture
    end
    def add
     #add the product
    end
   end
```

Listing 5.3: Login Controller

```
   class LoginController < ApplicationController
    def is_logged
     #check if the user is logged in
    end
    def is_admin
     #check if the user is logged as admin
    end
    def valid_login(credentials)
     #check the validity of the credentials
end
    def valid_answer(answer)
     #check the validity of the security answer
end
    def credentials
     #ask for the credentials
end
    def security_question
     #process credentials
     #ask for security question
    end
    def submit
     #log the user in
    end
   end
```

On the other hand, the workflows constraints are specified and enforced thanks to *Workflower*. Namely we want to enforce workflows as such:

- *Login Workflow* – the first step gathers the login credentials, the second step ask for a security question, while the third step logs the user in.
- *Checkout Workflow* – the first step gathers the payment information, the second step gathers the shipping information, while the third step review and submit the order. Furthermore, it requires the user to be logged in, otherwise it should redirect the user to the login process and afterwards resume the checkout.
- *Catalog Management Workflow* – the first step gathers the product details, the second step let the administrator submit a picture, while the third step add the product to the catalog. Furthermore, it requires the administrator to be logged in , otherwise it should redirect him to the login process and afterwards resume the process. In addition, this workflow needs to be multiply instantiable, since we want to allow to the user to add multiple products at once, because, for instance, pictures uploading may require some time.

The workflows are graphically shown in Figure 10.

From the overview of the workflows we want to enforce, we can easily write the specification as shown in Listing 5.4. One notice how the workflows specifications and the application logic are neatly separated, allowing a clear understanding of what the application does and how the user interaction is expected to be. Moreover, once the workflows

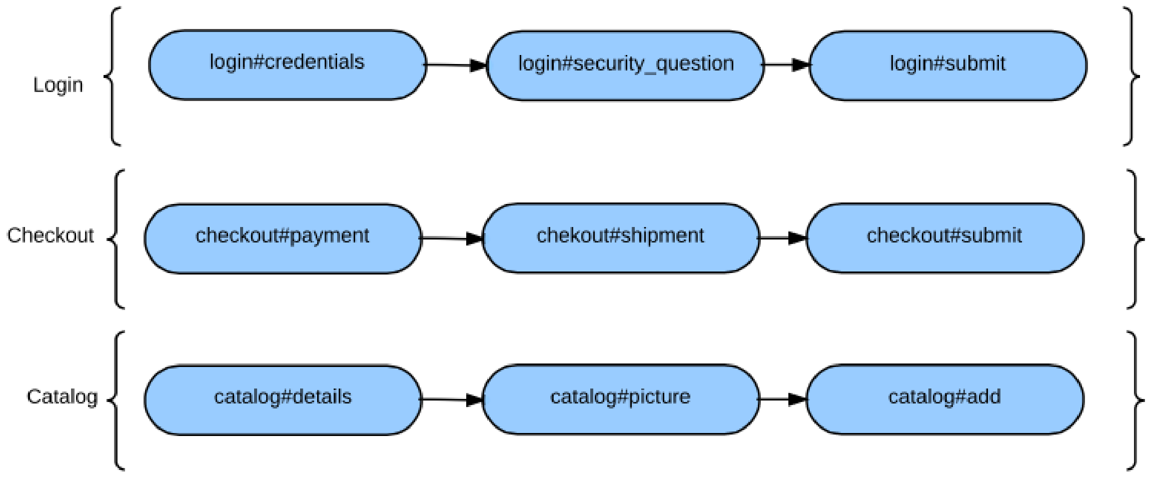

Figure 10: Workflows for the sample application.

Listing 5.4: Workflows Specification

```
workflow :login do
 step page:'login#credentials'
 step conditions: {valid_login(params[:login])}, page:'login#security_question'
 step conditions: {valid_answer(params[:question])}, page:'login#submit'
end

workflow :checkout do
 step conditions: {is_logged}, page:'checkout#payment', redirect_to_wf: 'login'
 step conditions: {valid_payment(params[:card_number])}, page:'checkout#shipment'
 step conditions: {valid_ship(params[:shipping_info])}, page:'checkout#submit'
end

workflow :catalog, multiple_instance:true do
 step conditions: {is_admin}, page:'catalog#details', redirect_to_wf: 'login'
 step conditions: {valid_details(params[:details])}, page:'catalog#picture'
 step conditions: {valid_ship(params[:picture])}, page:'catalog#add'
end
```

overview has been defined, the complete specification follows naturally from it. The building of the application took barely one hour, and all of the time has been spent on the actual logic of the application, without taking care of the workflows requirements. These are indeed automatically enforced from the specification, which has been written in few minutes, so that the correct behavior is provided without having the developer spending time on implementing defense from workflows attacks.

5.2 Enforcements

To ensure that our implementation actually enforced the intended behavior, we tested that:

1. *the navigation followed the described ordered*. Namely, we wanted to test that one step was accessible only if the preceding step had been already visited. Thus, we tried to access the second step of the checkout process and provide shipment informations without having already submitted the payment informations (first step of the checkout process). The framework correctly identified the violation and redirected us to the correct step, i.e. the first one.

2. *the inputs were properly validated.* Namely, we wanted to test that each step validated the request parameter so that they could have been correctly processed in the succeeding step. Thus, we tried to access the shipment step providing invalid payment informations, e.g. invalid credit card number. The framework successfully detected the incorrectness of the inputs and asked us to submit again correct information.

3. *redirection were supported.* Namely, we wanted to test that, if the user tried to initiate a workflow without the proper requirements, the framework would have redirected the user to the workflow in charge of fulfilling those requirements. Thus, we tried to access the checkout workflow without being authenticated. The framework detected the violation and automatically redirected us to the login workflow. Furthermore, once we completed the logic process, we got taken back to the checkout workflow.

4. *Concurrent Workflows were supported.* We wanted to test whether the user was allowed to be concurrently enrolled in several workflows during the same session. Thus, we initiated the checkout workflow and the catalog management workflow in two different browser tabs and tried to alternate between them. The framework showed the expected behavior, allowing to independently proceed in the two different workflows and correctly validating the requirements for each of them.

5. *Multiple Instance were supported.* We wanted to test whether the user was allowed to instantiate multiple instances of the same workflow, when the specification allowed it. Thus, we initiated both the checkout workflow and the catalog management workflow twice, in different tabs, and observed the behavior. The instances of the catalog management workflows were correctly treated as independent of each other, as expected from the specification. Namely, proceeding in one instance did not reflected on the other instance of the same workflow. On the contrary, the two instances of the

checkout workflow were treated as they were the same, namely a progress in one instance was shown also in the other.

5.3 Performance

In order to measure the actual impact on the application performances, due to the use of *Workflower*, we ran some tests. More specifically, we measured the time needed for the request to be processed in different cases, with and without *Workflower*. The test were performed using the testing suite included in Ruby on Rails, which we used to measure the time needed to generate the response once the request occurs. Furthermore, we built a fictitious application, consisting just of random pages, and measured the performances with different types of workflows specifications. Thus, we performed a base test case (Test A) and modified it to show how the performances would have been affected by the number of workflow (Test B), the number of steps (Test C), number of active instances of the same workflow(Test D-E) and the number of concurrent workflows (Test F-G). Afterwards, we focused on the impact due to ambiguities and built Test H-L to identify the overhead due to the number of steps shared between different workflows.

- *Test A:* Specification with three 4-steps workflows, no ambiguities, no multiple instances and no concurrent workflows. We requested iteratively any page of each workflow and compute the average time to build the response.
- *Test B:* We modified test A to include twenty workflows.
- *Test C:* We modified test A to include 10-steps workflows.

- *Test D:* We modified test A to include 2 active instances of the same workflow.
- *Test E:* We modified test A to include 10 active instances of the same workflow.
- *Test F:* We modified test A to include 2 concurrent active workflows.
- *Test G:* We modified test A to include 10 concurrent active workflows.
- *Test H:* Specification with five workflows sharing the same page. We requested the shared page.
- *Test I:* Specification with ten workflows sharing the same page. We requested the shared page.
- *Test L:* Specification with twenty workflows sharing the same page. We requested the shared page.

The results of the tests are shown in Table I. Tests A-G showed that the use of the framework slightly impacts the application performances in case of no ambiguities and, furthermore, that the overhead is independent of factors such as the number of steps, the number of workflows, the number of active instances and the number of concurrent workflows. On the other hand, Tests H-L showed that the overhead increases when the specification includes ambiguities, e.g. shared states. Moreover, as we expected from the complexity discussion we presented in Section 4.3, the processing time is proportional to the number of shared steps. Nevertheless, the performances overhead brought by ambiguities is still acceptable for a real-use small or medium application. As a matter of facts, we assume that Tests I-L are cases that can rarely occur in real-life applications.

TABLE I: PERFORMANCES

Test Case	Processing Time with Framework	Processing Time without Framework
A	7ms	6ms
B	7ms	6ms
C	7ms	6ms
D	7ms	6ms
E	7ms	6ms
F	7ms	6ms
G	7ms	6ms
H	8ms	6ms
I	9ms	6ms
L	11ms	6ms

CHAPTER 6

RELATED WORKS

In this section we compare our work to closely related existing approaches to workflow specification and enforcement. TableII summarizes how the related work compares.

TABLE II: RELATED WORKS

Work	Approach	Specification	Enforcements
Halle et al.	FSA	Manual	Automatic
Memento	FSA	Learning	Automatic
Swaddler	Anomaly Detection	Learning	Automatic
Book et al.	Formal Input Validation	Manual	Automatic
Jayaraman et al.	Design Methodology	Manual	Manual
Workflower	**Framework**	**Manual**	**Automatic**

The work of Halle et al.[13] is the most closely related to ours and they base their approach on modeling web applications as state machines, which has been previously suggested in the work of Yuen et al.[12]. More specifically, they present a runtime enforcement mechanism that associate the workflow of a web application to a state machine model specified by the developer through an XML schema. Thus they restrict the user navigation through conditions validation and navigation tracking, based on session variables and request parameters. Therefore, similarly to us, they provide automatic workflows enforcement by building a model from the specification manually given by the developer. In contrast, they focus on basic instances of workflows, i.e. they do not discuss and handle cases of concurrent workflows or multiple instance workflows, which, on the other hand, our work takes into account.

Another approach based on state machines is Memento[5], which models a web application's behavior using a deterministic finite automata (DFA), directly derived from the application. They use DFA to defend against cross-site request forgery (CSRF), cross- site-scripting (XSS) attacks and workflows attacks. In contrast with our work, they automatically extract the DFA by analyzing the source code of the application, and this inherently fails in capturing workflows, intended by the developer, that the code does not clearly express.

Swaddler[2] is another example of approach based on the extraction of the workflows by learning from code. Swaddler is a server-side method that uses a anomaly detection approach and probabilistic models for characterizing the attributes of internal session variables and for associating invariants with blocks of code for automatic detection of workflow violations. The detection effectiveness however is dependent on the accuracy of learning the invariants associated with blocks of code and does not prove to be always effective.

Differently, Book et al.[14] present a formal model for specifying input validation rules for web applications and present a framework where an implementation can be generated from the formal specification. While their approach is based on manual specification and automatic enforcements as ours, they just focus on validating input provided by the user, which can lead to an incomplete solution for workflows attacks. In contrast, our approach combines constraints on sequences of user actions with constraints on input data. Moreover, our approach can be integrated in existing applications and does not require users to adopt a new framework for developing web applications.

Lastly, Jayaraman et al.[4] present a systematic methodology for constructing web applications to avoid the attacks in the first place. The proposed methodology leads to applications that are secure from several forms of forgeries by design. Their work addresses the lack of a systematic approach and our work embeds most of principles they describes, e.g. session and parameters validation. Nevertheless, they did not provide an automatic technique to implement the methodology they describe.

CHAPTER 7

CONCLUSIONS

In this thesis we described the concept of workflow logic, how it is needed to capture the expected user interaction with the application, and how the lack of systematic methodologies for implementing workflow logics may lead to a weak application, vulnerable to attacks. We presented the approach we followed to build the *Workflower* framework, which allows the developer to specify and automatically enforce the workflow logic. Namely, *Workflower* enables to specify workflow logic separately from the application logic, in a declarative manner, so that the specification clearly describes the expected behavior and the maintenance of the code is proved to be easier. From the specification, the workflow logic is automatically enforced in the application, so that the navigation fulfills all the requirements and no violations occur. Furthermore, *Workflower* supports concurrent workflows, multiple instances workflows, automatic redirection and request resuming.

While the described approach can be applied to different Web development framework, we currently built *Workflower* on top of Ruby on Rails, which made the development of several features easier. We evaluated the usability and effectiveness of *Workflower* by building a simple e-commerce application and tested the performances of the implementation. We showed how the current implementation does not significantly affect the requests processing time in most of the cases, while it may bring penalties in case of workflows with enforcement ambiguity.

Future Works

Lastly, we aim to propose possible enhancements that may extend our work in order to provide additional features.

Firstly, the described approach does not handle ambiguities where the same page appears more than once in the same workflow. Despite that, our method is extensible enough to easily allow the modification we already discussed at the end of Section 3.4.3, i.e. identify each step not only by the page, but also by the request parameters it expects.

In addition, *Workflower* could reach more expressive power if it would give the possibility of associating the proper redirect action for each violated step requirement, instead of having one general redirect for any violation. Assume, for instance, that a workflows requires the user to be logged in and to have previously read and agreed the application terms and conditions. Since the conditions are independent of each other, they would need different redirection in case either of them is violated. One possible solution would be modifying the specification and substitute the set of requirements with a set of couples *(requirement, action)*, so that each condition has the proper redirect action.

Lastly, a more substantial modification would be adapting the presented approach so that it would accepts specifications more complex than simple sequences of steps, e.g. with possible branches. Despite we believe that extending our approach, to handle the described modification, is feasible, it would require a deeper study focused on the issues that may arise from having workflows consisting of complex paths.

CITED LITERATURE

1. Christey, S., and Martin,R. A.: Vulnerability type distributions in CVE. http://cwe.mitre.org/documents/vuln-trends/index.html.
2. Cova, M., Balzarotti, D., Felmetsger, V., and Vigna, G.: Swaddler, An Approach for the Anomaly-based Detection of State Violations in Web Applications, 2007.
3. Cova, M., Balzarotti, D., Felmetsger, V., and Vigna, G.: Multi-Module Vulnerability Analysis of Web-based Applications, 2007.
4. Jayaraman, K., Lewandowski, G., Talaga, P. G., Chapin, S. J., and Hafiz, M.: Modeling User Interactions for (Fun and) Profit: Preventing Workflow-based Attacks in Web Applications, 2010.
5. Jayaraman, K., Lewandowski, G., Talaga, P. G., and Chapin, S. J.: Memento: A Framework for Hardening Web Applications, 2009
6. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T.: Hypertext Transfer Protocol, HTTP/1.1. RFC 2616 ,1999. http://www.w3.org/Protocols/rfc2616/rfc2616.html.
7. Hypertext Transfer Protocol, Protocol Parameters. http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html.
8. Ruby on Rails. http://rubyonrails.org.
9. RubyGems. http://docs.rubygems.org/read/chapter/20.
10. Ruby on Rails, Callbacks. http://api.rubyonrails.org.
11. Jovanovic, N., Kirda, E., and Kruegel, C.: Preventing cross site request forgery attacks, 2006.
12. Yuen S., Kato K., Kato D., and Agusa K.: Web automata: A behavioral model of web applications based on the mvc model, 2006.
13. Halle, S., Ettema, T., Bunch, C. and Bultan, T.: Eliminating Navigation Errors in Web Applications via Model Checking and Runtime Enforcement of Navigation State Machines, 2010.
14. Book, M., Bruckmann, T., Gruhn, V., and Hulder, M.: Specification and control of interface responses to user input in rich internet applications.
15. Fielding, R.T., Taylor, R.N.: Architectural styles and the design of network-based software architectures, 2000.
16. Reenskaug, T.: THING-MODEL-VIEW-EDITOR - an Example from a planningsystem, Xerox PARC, 1979.

VITA

Daniele Rossetti

Education	**B.S., Engineering of Computing Systems** Politecnico di Milano, Milano, Italy 2011
	M.S., Computer Science (*current*) University of Illinois at Chicago, Chicago, IL 2013
	M.S., Engineering of Computing Systems (*current*) Politecnico di Milano, Milano, Italy 2013
Working experience	**Research Assistant at University of Illinois at Chicago** Fall 2012 to Spring 2013