SAFE Extensibility for Data-Driven Web Applications

Raphael M. Reischuk*
Saarland University
Germany

Michael Backes
MPI-SWS, Saarland University
Germany

Johannes Gehrke
Cornell University
NY, USA

ABSTRACT

This paper presents a novel method for enabling fast development and easy customization of interactive data-intensive web applications. Our approach is based on a high-level hierarchical programming model that results in both a very clean semantics of the application while at the same time creating well-defined interfaces for customization of application components. A prototypical implementation of a conference management system shows the efficacy of our approach.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Extensibility*; D.2.7 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering (CASE).*

General Terms

Design, Languages, Security.

Keywords

Customization, Software-as-a-Service, Data Management.

1. INTRODUCTION

More and more software is delivered through the web, following today's cloud idea of delivering Software as a Service (SaaS). The code of such rich internet applications (RIAs) is split into client and server code, where the server code is run at the service provider and the client accesses the application through a web browser [6]. In data-driven web applications, the state of the application resides in a database system (or in a key-value store), and users interact with this persistent state through web clients. In this paper, we propose a framework for *personalizing* such data-driven web applications [9]. By personalization we mean that a user has the capability of customizing the functionality of an RIA to fit her unique application needs.

As a first example, consider Facebook user Mark, who no longer likes a single news feed for all of his contacts; Mark wants to split the news feed into two columns, one for his friends and one for his business contacts. Today, Mark would have to wait (and hope) for Facebook to create this functionality as part of an upgrade of its interface. We envision a world where Mark could take the initiative himself; he could directly "program" this extension and integrate it for himself into the running Facebook application. Mark could also provide this extension as an "App" to other users who desire the same functionality. Note that this is not a "Facebook Application" as enabled by the Facebook API, but it is a customization of the core user-facing Facebook functionality through a user-defined extension.

As a second example, consider a conference management system such as Microsoft's Conference Management Tool (CMT). From time to time, the team behind CMT introduces a new feature that has been long requested by the community (see, for example, the features currently marked "(new!)" on the CMT website [15]). None of these extensions is difficult to build, but today any changes are only within the realm of the CMT developers. In addition, due to limited resources, the team only incorporates extensions requested by the majority of users and thus forgoes the opportunity to serve the long tail. For example, consider Surajit who wants to run his conference with shepherding of borderline papers. Currently, Surajit has to wait and hope that the CMT team considers his functionality important enough to release it as part of its next upgrade. However, we believe that innovation and integration of such new functionality can be significantly increased if Surajit could directly take initiative, program the extension himself, and then share it with others in the research community who desire similar functionality. Thus we want custom extensions to be built by any member of the community instead of being left only to the CMT team.

In both of these examples, personalization of an existing data-driven web application by a third party who was not the developer of the original application is the key to success. Note that personalization not only benefits the user who programmed it; an extension could later on be shared with other users, making the application automatically an "extension app store" where users can (1) run the RIA directly as provided, (2) personalize it with any set of extensions developed and provided by the community, (3) personalize it themselves through easy and well-defined user interfaces, and then (4) share or sell their extensions to the community.

The tremendous benefits of personalization also come with huge challenges. First, the often organic growth of today's RIAs makes it hard to keep track of the diversity of locations

*Work done while the author was visiting Cornell University.

WWW 2012, April 16–20, 2012, Lyon, France.
ACM 978-1-4503-1229-5/12/04.

to which code has to be integrated, thereby obeying various security and safety constraints regarding, for instance, namespaces and assertions. This dispersion of code "all over the place," which is exacerbated by the integration of different programming models and languages for the client and server, makes it hard to bundle functionality for replacement through personalization. But since developers cannot anticipate all possible ways of extending an application, how do we design a web application such that future extensions are easy to integrate? Second, the code of the extensions will have to be activated, it may have to pass data back and forth with other application components, and it requires access to the state of the application in the database. How do we address the security concerns of integrating such untrusted code into a running web application?

In this paper, we propose SAFE, a framework for the design of data-driven web applications. Let us give a brief overview of SAFE and its features.

Design for Personalization

SAFE structures data-driven web applications into a *hierarchical programming model* inspired by Hilda [19, 18]. Functionality is clustered into so-called *f-units* that contain all the relevant code to implement a component of the application. The control flow of the application has a clean hierarchical semantics: An f-unit is activated by its parent f-unit and becomes its child resulting in a tree of activated f-units. This so-called *activation tree* naturally corresponds to the hierarchical DOM structure of an HTML page. There are two well-defined points of information flow for an f-unit: Its activation call, through which the f-unit was activated by its parent f-unit, and queries to the database where the state of the application is stored. Thus a user who would like to personalize an application simply has to replace an existing f-unit with a new f-unit of her choice or design. Such customizations are dynamic in that f-units are registered and activated without stopping the running system. These dynamic software updates (DSU) avoid costly unavailabilities of the running system [17, 8].

SAFE has a security model that is tailored towards the integration of untrusted code by splitting the code of an f-unit automatically between client and server. Database queries specified by a programmer will never appear in the client code, sanitization of query values to prevent SQL injection attacks automatically occurs on the server, and event handlers for asynchronous update request end up in the client. SAFE also contains a reference monitor which takes care of all low-level details such as secure registration of f-units, access control, and verification of user actions and requests received from the client.

Note that even only achieving modularity when designing data-driven web applications is nearly impossible. The f-units in SAFE can be thought of as classes in object-oriented programming. For web applications, however, there are several different languages (for example, HTML, PHP, Java, JavaScript, SQL, CSS) providing different data models for the different application layers (e.g., the relational model for databases, Java objects for the application logic, hyperlinks for website structure, and form variables for web pages). This variety makes it hard to achieve modularity since fragments of different languages are in different parts of the source tree. Usually, a single JavaScript command like `include('moduleA')` is not sufficient. Assume, as an example, `moduleA` is responsible for displaying some `<div>` elements which are supposed to appear only two seconds *after* the main HTML page has been loaded. In this case, an event handler for `onload` events of the document has to be modified accordingly. Typically, such an event handler is a named JavaScript function, referenced in the `<body>` tag of the main HTML page: `<body onload='pageLoaded()'>`. The JavaScript function `pageLoaded()` is uniquely declared at some other location, most likely in the `<head>` area of the HTML page. This declaration has to be updated if `moduleA` needs some actions to be performed when the page has been loaded; some lines of JavaScript code have to be added to the body of the function. For a different language, for example for PHP, the integration of new functionality again is different. Another difficulty in the integration of new functionality is to ensure that namespaces of different pieces of code do not interfere. Assume that we have two code fragments `A` and `B` which each have an HTML element with id `studentList` and corresponding CSS specifications. A namespace concept would separate the CSS for `A` from the CSS of `B`, and we have to add this manually in order to resolve this conflict. As part of its hierarchical programming model, SAFE provides solutions to address all these problems.

Client-Server Consistency

Modern interactive web applications give the user a feeling of locally executing a fully-fledged software binary by communicating with the server asynchronously. The typical way of implementing this is through client-side event-driven programming. One challenge when writing this client-side code is that the state of the application at the client can be different from the state at the server, since other clients simultaneously connect to the same application and may modify the state of the system at the server, for example when one user updates a data item that another user is currently displaying. To avoid such inconsistent updates, the programmer would have to manually include all kinds of consistency checks, which is error-prone and cumbersome. SAFE alleviates the developer from this burden by making consistency checks a first-class citizen in the model, providing an easy to use *SQL-based declarative state monitoring interface* that automatically derives the necessary checks. SAFE automatically compiles the developer code to safe state transitions which cleanly abstracts out concurrent updates into standard serialization semantics known from interacting with a database.

Ease of Development

SAFE also includes many different mechanisms to minimizing the amount of low-level code that a developer has to write. (1) Programs in SAFE are written in SFW, a high-level programming language that abstracts away many low-level code fragments through appropriate high-level statements. For example, it is often cumbersome to specify explicit loops and to iterate over the objects of a particular data structure thereby struggling with implementation details like counters, pointers or break conditions of that par-

ticular loop. SFW contains high-level constructs for many of these commonly re-occurring patterns. (2) One of the design principles of SFW is that we did not invent a new language, but rather create a *framework that encompasses existing languages.* Our framework provides the full expressiveness of languages like HTML, PHP, SQL, and Javascript, but allows for shorter, yet semantically precise *shortcuts that significantly reduce the amount of code* a developer has to write. (3) Note that application developers may have to know about other elements in the DOM tree in order to ensure that all elements have pairwise unique IDs, and other elements are correctly addressed, e.g., whether an element has the `innerHTML` property or the `value` property. SAFE's modularization fosters *local understanding* because it automatically ensures that IDs are unique and that the developer only needs to locally care about the elements of the corresponding f-unit. (4) Today a lot of similar event-driven code for asynchronous server requests has to be written. However, the code for the update of an exam grade in a course management system is not much different from the code of updating matriculation number of a student. In the spirit of DRY (Don't Repeat Yourself), as in *Ruby on Rails* [10], SAFE requires the developer to specify information and code at most once. For example, the code for the initial rendering of an f-unit is also used later to provide partial updates of modified data. No complicated event handlers have to be specified to rebuild certain elements in the browser's DOM tree. Another feature to reduce the amount of hand-written code is the paradigm of *convention over configuration*: SAFE decreases the number of decisions a developer has to make by establishing useful conventions on parameters and names of variables.

Structure of the Paper. The remainder of the paper is structured as follows. Section 2 describes our novel hierarchical programming model and shows how it achieves client-server consistency, personalization, and security. In Section 3, we describe some of the interesting aspects of the implementation of SAFE. We outline our initial experiences with SAFE and discuss future work in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2. SAFE

This section introduces the *Safe Activation Framework for Extensibility* (SAFE). SAFE provides automatic application state consistency and safe extensibility. We first introduce our application model (Section 2.1), then we show how to handle updates to the application state (Section 2.2), how to model extensibility (Section 2.3), and how to achieve security (Section 2.4). Each of these sections concludes with a small example showing how the described functionality is specified using SAFE. Additional information is available online at `http://www.safe-activation.org`.

2.1 Application Model

SAFE provides a hierarchical programming model which naturally builds upon the hierarchical DOM structure of web pages. The most constitutive components in SAFE are its so called *f-units*, see Figure 1 for an illustration. An f-unit clusters all code fragments for a specific functionality within a web page, including the business logic, the visual appearance, and the interaction with users or other f-units. This clustering provides a clear level of abstraction through well-defined *interfaces* for each f-unit. The modularity of an f-unit relieves the programmer from struggling with variable scopes and their interference.

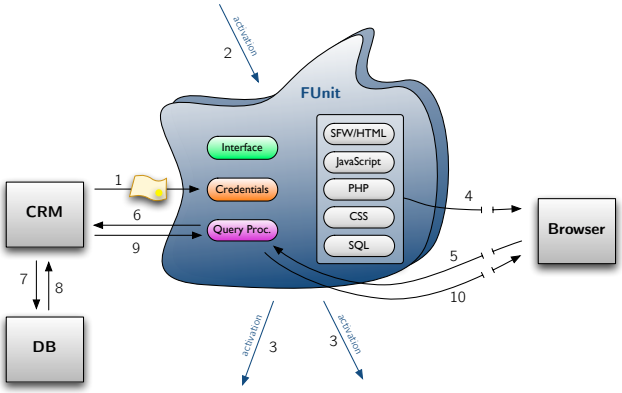

Figure 1: Integration of an f-unit.

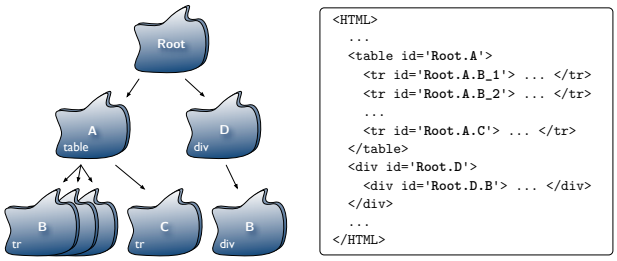

Figure 2: Activation tree and its corresponding web page.

As a result, this abstraction provides an elegant way of composing web pages out of several different f-units. A web page is modeled as a so called *activation tree* (inspired by Hilda [19, 18]) in which f-units are organized hierarchically. Figure 2 shows an example of an activation tree with its corresponding HTML code. A node in the activation tree corresponds to one or more nodes in the HTML DOM tree.

The integration of an f-unit `F` in the activation tree is referred to as *activation* of `F` (Fig. 1, step 2). More precisely, an f-unit is activated by its parent f-unit and thereby receives activation data through well-defined interfaces. The f-unit can use the activation data or data obtained directly from the database through queries to display parts of the web page. An f-unit can also activate other f-units, its child f-units (Fig. 1, step 3).

Activation comes in two kinds: (1) In the example in Figure 2, the root f-unit performs *static activations* of the f-units `A` and `D`. These activations are independent from the data the root f-unit has. Assume, for instance, that f-unit `A` represents a table. The developer might wish to display the headline of the table independently from whether there are entries in the table or not. In contrast, (2) f-unit `A` performs *dynamic activations* of f-unit `B`. The dynamic activation of child f-units is data-driven in that (a) the number of activated instances corresponds to the number of items from the activation source, e.g., from a database query, and (b) the parameters passed to the i-th activated f-unit contain ex-

actly the i-th data item from the activation source. For instance, if the result of a database activation query consists of n rows, then n instances of `B` will be activated, one for each row r_i. Child f-unit i obtains row r_i as activation parameters. The activation of `C` in f-unit `A` is again static. This f-unit could, for instance, display a row summarizing properties of the rows above.

Whenever an instance of an f-unit is activated, the corresponding compiled HTML/JS/CSS code is made available in the activation tree. Eventually, the activation tree is linearized to a single HTML document by transforming subtrees to nested HTML elements (Figure 2). After the activation tree has been constructed, the corresponding code for HTML/JS/CSS is sent to the client (Fig. 1, step 4).

Activation is expressed through *activation calls* in our high-level *modeling language* SFW, which is a straightforward extension of HTML: all HTML elements and also PHP and JavaScript can be used as in traditional web application implementations. Activation calls are at the core of SFW and can as such be used in any HTML context.

Example 1. The static activation of f-unit `A` in f-unit `Root` as shown in Figure 2 is expressed by the activation call

```
<activate:A(initParam1,initParam2,...)/>
```

where $initParam_i$ are activation parameters, i.e., values to flow from `Root` to `A`. For this static call, one instance of `A` is activated independently from the content of the activation parameters. The dynamic call for f-unit `B`, however, results in activated f-units only if the result of executing the specified *activation query* is not empty:

```
<activate:B query='SELECT ...'/>
```

More precisely, for each returned tuple $(v_1, v_2, \ldots, v_k)$, one instance of f-unit `B` is activated with the activation parameters $(v_1, v_2, \ldots, v_k)$. Instead of specifying a database query, it is also possible to provide an array of key/value pairs. Each such pair results in one activation with the particular values. All code for the preparation of an activation, e.g., setting up an activation array, is enclosed in the activation tag: `<activate:C array=$tmp> ... array_push($tmp,...); ... </activate>`. *

2.2 Updates

Recall that web pages nowadays are not static pages: they contain a lot of reactive code for event-driven modifications of the overall application state. SAFE's methodology to automatically handle such updates and to maintain state consistency, also for concurrent updates, is explained in the following.

Assume the client's browser interacts with the delivered HTML page and eventually sends some update request back to the web server (Fig. 1, step 5). The corresponding f-unit in the activation tree processes this request and generates a database query q, for which SAFE automatically verifies various safety and security properties. These include checks for state consistency, access control, and prevention of code injection. After the query q has been executed, SAFE automatically triggers all f-units in the activation tree which have an out-dated state. These f-units are rebuilt and sent to the client.

SAFE alleviates the developer of an f-unit `F` from caring about the freshness of its state while `F` is updating the application state. Moreover, the developer does not have to provide code for partial updates of any f-units in the tree. The developer only specifies the *update query* q that is supposed to be executed for some event attached to an element in `F`. Let us explain how this works through the following example.

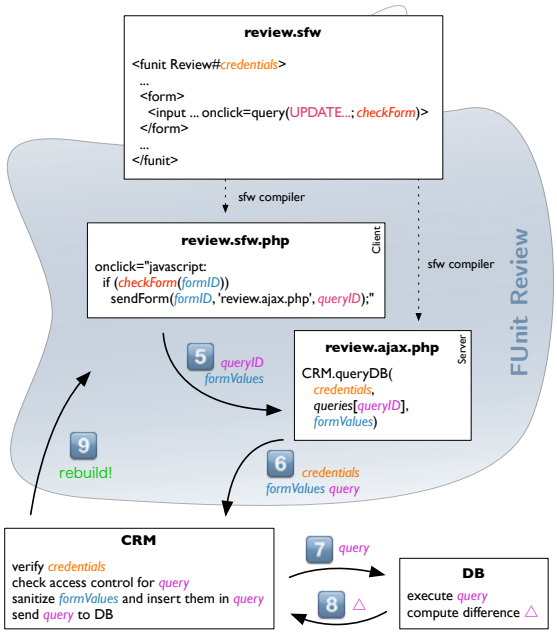

Figure 3: Update of the application state.

Example 2. Figure 3 shows parts of a conference management tool. The uppermost code box shows a code fragment of the specification of the f-unit `Review` in our modeling language SFW. The f-unit contains – among other elements – a form and an input element with an `onclick` event. This event is fully specified by a database query and a Boolean check function `checkForm`. Loosely speaking, upon a click, SAFE executes the specified query against the current database state assuming that (1) the execution of `checkForm(formID)` evaluates to true, and (2) SAFE has verified that the query is *safe*, i.e., the query is not based on an out-dated state. The `formID` is an automatically derived identifier for this form, which is unique in the activation tree and hence also in the HTML DOM tree. The developer can specify the check function arbitrarily, or just omit it and solely specify the query. The technical parts in the transition of steps 5 to 8 are not relevant for the semantical model, and hence explained in Section 3. *

SAFE executes the query (step 5 to step 8) and computes a difference Δ to the previous database state. Based on this difference, SAFE automatically triggers the corresponding f-units in the current activation tree and tells them to update their state if necessary (step 9). To this end, f-units can *subscribe* to database differences: f-unit `F` can specify a so-called *subscription function* $sub_F(\Delta)$ in order to receive a notification whenever sub_F returns non-empty results for Δ.

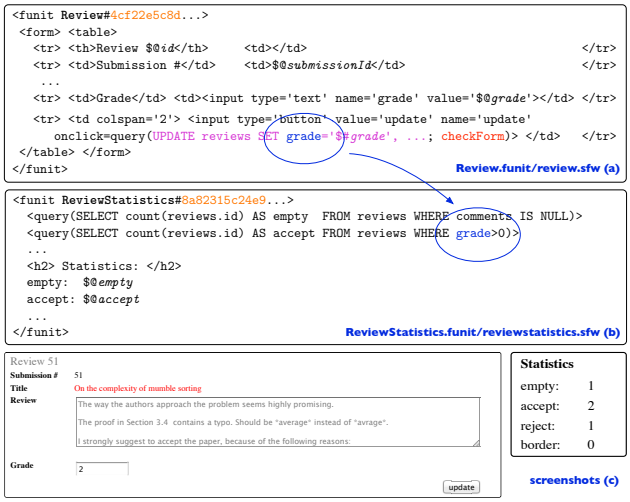

Figure 4: (a,b) SFW code for two f-units, (c) screenshot of the representation in a web browser.

We refer to Section 3.2 for more the details on the difference, in particular for details on concurrent updates.

Example 2 (continued). Assume an f-unit dynamically activates a list of reviews, each specified as f-unit `Review` (Figure 4a). Furthermore, let there be a static activation of one instance of f-unit `ReviewStatistics` (Figure 4b). A screenshot of the representation of one such review in a browser is shown below the SFW code (Figure 4c). Most notably, this example shows the concise and elegant way of a specification of web application code in the modeling language SFW: Values obtained from activation calls `<activate...>` or from simple queries `<query...>` are accessible with the prefix `$@`, e.g., `$@id`, `$@submissionId`, form values are accessible via the prefix `$#`, e.g., `$#grade`.

The blue arrow highlights the data dependency between the update query in `review.sfw` and the select query in `reviewstatistics.sfw`. According to the specified subscription functions of `ReviewStatistics`, SAFE automatically triggers the f-unit to refresh whenever the subscription functions return a non-empty result. In this case, an update of a review issued by `Review` would cause `ReviewStatistics` to be refreshed. (Even if `ReviewStatistics` has not specified any subscription function, SAFE automatically triggers the f-unit based on the database columns that have been read upon `ReviewStatistics`'s most recent activation.) The simple subscription functions `com` and `acc` for `Review` as shown in the last section of the interface in Figure 6 achieve more fine-grained control: The result of `com` contains any update that affects the comments (`SELECT comments`) of the reviews (`FROM reviews`). The result of `acc` contains any review information (`SELECT *`) for updated reviews (`FROM reviews`) with a negative grade prior to an update (`BEFORE grade<0`) and a positive grade afterwards (`AFTER grade>0`).

We stress that the example shows simplifying syntactic sugar for SQL queries that are typically more complicated. SAFE translates the extended SQL syntax to standard SQL code and creates the corresponding triggers. ∗

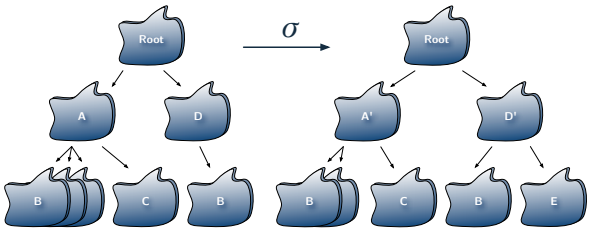

Figure 5: Customization σ.

2.3 Extensibility

Recall that *customization* refers to the action of modifying an existing web application (into previously not considered directions). The extension of the running system might be provided by an untrusted third party. SAFE implements customization by activating an f-unit `G` instead of an initially specified f-unit `F`. In other words, in the activation tree, the node initially representing `F` is replaced by a different node for `G`. More formally, customization is a substitution $\sigma : \mathcal{U}_{|\mathcal{T}} \to \mathcal{U}$ mapping f-units $\mathcal{U}_{|\mathcal{T}}$ in the activation tree $\mathcal{T}$ to other f-units $\mathcal{U}$.

Example 3. Consider the customization $\sigma = [\texttt{A} \mapsto \texttt{A'}, \texttt{D} \mapsto \texttt{D'}, u \mapsto u]$ as shown in Figure 5. The initial activation tree (left) is transformed to the customized activation tree (right): F-unit `A` is replaced by f-unit `A'` which activates only *two* instances of `B` (e.g., because `A'` uses a different activation query for `B`). Furthermore, f-unit `D` is replaced by `D'` which additionally activates a single instance of f-unit `E`. ∗

This example demonstrates how new functionality is integrated in a web application. Moreover, it shows that a customization does not only affect a single f-unit, but instead can affect entire subtrees. As an f-unit consists of code for both business logic and visual appearance, customization is not restricted to changes in the visualization, e.g., different background colors, font-sizes. In an implementation in practice, users can individually specify customizations and provide them to other users within a network. The system provider may approve every such customization once.

The domain of a customization mapping must address only the intended f-units in the activation tree: A *general* customization σ_1(`B`) = `B'` would affect four f-units in the left activation tree in Figure 5. A more *specific* customization σ_2(`Root.D.B`) = `B'` would only affect one f-unit. The domain of customization mappings for a specific f-unit $u \in \mathcal{U}_{|\mathcal{T}}$ must take into account a (partial) path from the root f-unit down to u. The reference to u can hence be seen as an *address pattern* that has to match a path in $\mathcal{T}$. The leaf of the path `Root.D.B` is matched by the address patterns `B`, `D.B`, and `Root.D.B`.

All customized f-units u_i' in a customization $\sigma_3(u_i) = u_i'$ are called with the same activation parameters as the initial u_i, which is necessary for (1) modularity: the activating f-unit v_i shall not have to know whether u_i is activated or u_i'. SAFE ensures that the intended f-units are activated. In other words, v_i shall not be affected by a customization of any of its child f-units. (2) Information flow: it is more accurate to reason about information flow if information is propagated only in top-down direction in the activation tree.

```
1  INPUT:
2    init = ($!reviewerId)
3    activate = {$@id, $@title, $@comments, $@grade}
4
5  ACTIVATION:
6    FooBar.init = ($!reviewerId)
7    FooBar.activate = {$@id}
8
9  DATABASE:
10   read = {reviews.id}
11   write = {reviews.comments, reviews.grade}
12
13 SUBSCRIPTION:
14   com: SELECT comments FROM reviews
15   acc: SELECT * FROM reviews WHERE
16          BEFORE grade<0 AND
17          AFTER  grade>0
```

Figure 6: Sample f-unit interface. The sections `INPUT`, `ACTIVATION`, `DATABASE`, and `SUBSCRIPTION` correspond to the arrows 2, 3, 6, and 9 in the Figures 1 and 3.

2.4 Security

There are three ways for an f-unit to send and receive data which is displayed or used to activate other f-units: (1) An f-unit can receive data from its parent f-unit upon activation through the *activation parameters*. (2) An f-unit can have direct access to the database for both reading and modifying data. (3) An f-unit can activate other f-units and thereby send information to them.

Information Flow

In order to reason about information flow in a web application, each f-unit `F` needs to declare an *interface* int_F. A sample interface for an f-unit to display a single paper submission in a conference management system is shown in Figure 6: Upon activation, the f-unit expects a reviewer ID and some activation parameters, e.g., the review ID, the submission title, etc. The f-unit activates one child f-unit, `FooBar`. Finally, the f-unit reads the column `id` of the database table `reviews` and writes updates to the columns `review` and `grade`. The submission title is not updated. The subscription functions as introduced in Section 2.3 are also specified through the interface. SAFE verifies that the queries occurring in subscription functions also obey the access control constraints, as specified in the DATABASE section of the interface.

Access Control

SAFE inspects the interface int_F at the initial registration of f-unit `F` in the web application. The service provider, who is running the service, has to decide whether the specified interface is appropriate. If so, cryptographic credentials are hand out to `F` (Fig. 1, step 1) and int_F is translated to corresponding access control constraints. From this point on, `F` is allowed to read and write the respective database columns after authenticating using the provided credentials. The access to any other column is not permitted.

We stress that any user data (e.g., login credentials, names, and related access control policies) is dynamic information of a web application. In contrast to f-units, these first-class citizens are part of the content of the web application and therefore stored in the database. SAFE thus far only maintains access control on f-units; access control on user access must be modeled by the developer, as ever before.

Access Control for Customized F-units

In order to ensure access control for customized f-units, we define interface int_F to be *at least as restrictive* as interface int_G for f-units `F` and `G`, denoted by $int_F \preceq int_G$, if the following two conditions hold.

$$(\mathtt{INP}_F \cup \mathtt{DBR}_F) \subseteq (\mathtt{INP}_G \cup \mathtt{DBR}_G)$$
$$(\mathtt{ACT}_F \cup \mathtt{DBW}_F) \subseteq (\mathtt{ACT}_G \cup \mathtt{DBW}_G)$$

The sets $\mathtt{INP}_F$, $\mathtt{DBR}_F$, $\mathtt{DBW}_F$, and $\mathtt{ACT}_F$ represent the corresponding fields in the interface for f-unit `F`. A customization σ is called *safe* if $\forall u \in \mathcal{U} : int_{\sigma(u)} \preceq int_u$. SAFE generally verifies whether customization is safe.

However, in order to add new functionality to a web application through customization, $\sigma(\mathtt{F})$ might require more access than `F` does, hence $int_{\sigma(F)} \not\preceq int_F$. Such a special case is called *declassified customization*. A declassified customization requires special approval by the system provider.

3. IMPLEMENTATION

This section details particular insights about the implementation. These details are hidden from the developer.

A substantial component in the implementation of SAFE is its *CRM*, the *Centralized Reference Monitor*. The *CRM* controls the interaction of f-units with the database and achieves consistency for concurrent data updates. Moreover, the *CRM* maintains the *registration* and the *activation* of f-units. Upon *registration* of an f-unit `F`, the *CRM* hands out cryptographic credentials (Fig. 1, step 1), which `F` will use for authentication at the *CRM* later. Furthermore, the *CRM* derives access control constraints for `F` (also explained later). These constraints are maintained by the *CRM* and considered dynamically whenever a database query is received from `F`. From this point on, `F` is registered in the web application: its functionality can be integrated to a web page via an activation of `F`.

In SAFE, all communication between client and server is based on asynchronous message transfer. It is no longer appropriate to reload the entire web page in case of a (possibly small) update from the client: The browser would blank out while waiting for a new page to be computed and delivered by the server. Any browser state not been sent to the server, e.g., non-submitted forms, cursor positions, scroll-bar positions, would be lost. We therefore rely on partial updates. The entire page is reloaded only if essentially all elements of a page need to be updated.

SAFE redeems the developer from caring about such partial updates and from implementing the corresponding JavaScript event-handlers. The developer simply specifies the query that is supposed to be executed for a certain event, e.g., for a click or a keystroke. More precisely, the specification file `review.sfw` in Figure 3 is compiled into the files `review.sfw.php` and `review.ajax.php`. The first file represents client code, while the second file resides on the server.

3.1 Updates from the Client

This section shows the implementation details of client updates based on the example shown in Figure 3. To this end, let $\mathcal{U}$ be the set of available f-units, let $\mathcal{U}_{|\mathcal{T}} \subseteq \mathcal{U}$ be the set of f-units in the current tree $\mathcal{T}$, let $\mathcal{DBS}$ be the set of database states, let $\mathcal{DBC}ols$ be the set of columns in the database, for instance, $\mathcal{DBC}ols$ = { `students.sid`, `students.name`, ..., `courses.cid`, ...}, let $\mathcal{Q}$ be the set of database queries. Let $affCols : \mathcal{DBS} \times \mathcal{Q} \to \wp(\mathcal{DBC}ols)$ be the set of database columns affected by a query for a given database state. Let $readCols : \mathcal{U} \to \wp(\mathcal{DBC}ols)$ be the set of database columns a given f-unit reads upon activation.

Now assume an event at the client triggers a specified update query $q \in \mathcal{Q}$ for f-unit `Review` for execution against the current state of the database. The subsequent steps are explained in more detail in the following.

5. The client code calls the specified check function `checkForm`. Upon success, the client calls SAFE's function `sendForm` which expects the corresponding `formID`, a URL where to send the query to, and a unique identifier `queryID` for the query. Note that the query never appears in the client code, but instead an unforgeable unique cryptographic identifier is inserted automatically. The query itself occurs only in the server code, in this case in the file `review.ajax.php`. The same holds true for the credentials.

6. The f-unit sends its `credentials`, the form values, and the actual query to the *CRM*. If the *CRM* is currently not executing any other request, i.e., the *CRM*'s state is *idle*, the *CRM* sets its state to *busy*. Otherwise the request is refused. Next, the *CRM* verifies the authenticity of the f-unit, verifies the access control constraints for the specified query according to the constraints derived at registration time. Finally, the *CRM* checks whether the query originates from a sufficiently up-to-date client (details are explained in Section 3.2). Furthermore, the *CRM* sanitizes all form values, i.e., special characters such as quotes and semicolons are escaped. Through instantiation of the sanitized values, the *instantiated query* $\tilde{q}$ is obtained and ready to be executed.

7. The *CRM* sends the query $\tilde{q}$ to the database where $\tilde{q}$ is executed at the current state $s \in \mathcal{DBS}$. The execution of $\tilde{q}$ yields a difference $\Delta(s, \tilde{q}, s')$ from the database, which transitions to a new state $s' \in \mathcal{DBS}$: $s \xrightarrow{\tilde{q}} s'$.

8. The *CRM* obtains the difference Δ from the database.

9. All f-units $f_i \in \mathcal{U}_{|\mathcal{T}}$ that have a subscription to the difference Δ are notified by the *CRM*. More precisely, f-unit f_i is notified when $sub_{f_i}(\Delta) \neq \varnothing$. The notification message consists of the evaluated function $sub_{f_i}(\Delta)$. If no explicit subscription function is declared for an f-unit f_i, the function $sub_{f_i}(\Delta) := affCols(s, \tilde{q}) \cap readCols(f_i)$ is used as a default. Finally, the *CRM*'s state is set to *idle*.

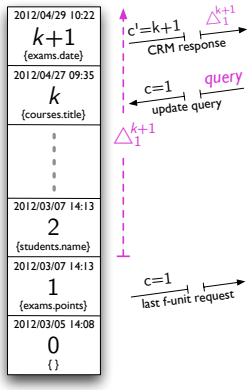

Figure 7: Concurrency clock.

3.2 Concurrent Updates

To overcome the synchronization problem when the *CRM* is interacting with several clients, each of which having possibly out-dated views of the application state, the *CRM* is extended by a logical clock to track causality. This clock is updated whenever an f-unit issues a query to update the database.

Formally, a *clock entry* is a tuple $\langle c, t, \Delta, u \rangle$, where $c \in \mathcal{C}$ is a strictly monotonically increasing integer representing the *clock's value*, and $t \in \mathcal{TS}$ is a *timestamp* capturing the time of update. Additionally, information about the database difference Δ is stored, together with the f-unit $u \in \mathcal{U}$ that has issued the update query.[1]

Assume f-unit `F` has received its last information at clock value $c = 1$ (cf. Figure 7). Due to changes by other users, the current clock value has increased to $k > c$. Now, assume `F` issues a query q_c^k, i.e., a query based on local clock value c, but at current *CRM* clock value k. The *CRM* checks whether q_c^k is *clock-safe*, which intuitively means that the query is only built on values that have not been altered since the creation of the query. More formally, let $clockCols(i)$ be the set of affected columns by a database update at clock value i. Then, $cols(\Delta_c^{c'}) = \bigcup_{c<i\leq c'} clockCols(i)$ is the set of modified database columns for the clock interval $(c, c']$. Let the domain of *readCols* be lifted to sets of f-units in the straight-forward way. Let the set $\mathtt{F}^*_{\mathcal{T}}$ contain all f-units in the activation tree $\mathcal{T}$ that are on the path from the root node to `F`, i.e., $\mathtt{F}^*_{\mathcal{T}}$ is the smallest set satisfying $\{\mathtt{F}\} \cup \{pred_{\mathcal{T}}(u) \mid u \in \mathtt{F}^*_{\mathcal{T}}\} \subseteq \mathtt{F}^*_{\mathcal{T}}$, where $pred_{\mathcal{T}}(u)$ is the direct predecessor of the node corresponding to f-unit u in $\mathcal{T}$. We say a query q_c^k is *clock-safe* if $cols(\Delta_c^k) \cap readCols(\mathtt{F}^*_{\mathcal{T}}) = \varnothing$.

If the query is checked to be clock-safe, the *CRM* creates a new clock entry with value $k+1$ containing the difference Δ received from the database, i.e., $clockCols(k+1) = \Delta$. The *CRM* returns to `F` not only the result of the query, but also the new clock value $c' = k+1$. If the query is not clock-safe, the query cannot be executed. The calling f-unit `F` is asked to retry with a refreshed local state.

[1]The details on the difference Δ are strongly implementation-specific and not relevant here. It is only necessary that it be feasible to extract from Δ any information about the updated database tuples.

3.3 Customization

We assume a hierarchy of principals as depicted below. A single *service provider* (e.g., the IT service of a university) offers a global software service (e.g., a university-wide course management system) to be used by a certain set of people (e.g., the members of the university's faculties). The service is complete in that all basic functionality is deployed (e.g., adding students, assigning students to courses, updating grades, etc.). Moreover, we assume that security policies are implemented correctly (e.g., only eligible staff is allowed to access grades, students cannot learn other students' grades, etc.).

Service Provider
Customizers
Clients
Users

Whenever the service shall be tailored to individual requirements, customized functionality is implemented by the *customizers* (e.g., the department of philosophy, or the law school). These customizers can either add new functionality to the system or personalize existing functionality. We assume that such modifications are not in the interest of all users of a system, but for a subset of them. The sports department might integrate an e-commerce shop for their students to purchase corporate sports clothing, which the law school might not need. But instead, the law school is running a small library that shall be integrated in the system.

The *clients* are devices (e.g., desktop computers, tablet PCs, smartphones, etc.) that can interact with the original service or with customizations thereof. Customizations can be seen as "apps" that can be installed *for* a client. Following the "software-as-a-service" paradigm, these apps are not installed *on* the clients, but are integrated to the main service on the servers of the service provider.

Finally, *users* are authenticated clients. For instance, a student might use a desktop computer of the law school to check his class schedule. He authenticates with his student ID, and sees the customized user interface the law school provides. Customization is hence related to clients rather than to users. This, however, is not a restriction as customization can also be applied to users so that users can choose customizations themselves.

Our implementing of customization mappings uses cookies that are stored in the client browser. The cookies do not store the mappings itself but a reference to the mapping. The mappings are stored on the application servers. This provides a simple method to share customizations between different clients and users.

3.4 Authenticity

In order to ensure that all constraints on the access of database fields are met, we require that an f-unit `F` authenticate before communicating to the database. The authenticity of f-unit `F` is established via the credentials $cred_{\texttt{F}}$ (Fig. 1, step 1). The credentials depend on some password p of the *CRM* and on the name of the corresponding f-unit:

$$cred_{\texttt{F}} = \mathsf{hash}(p \,\|\, \texttt{F})$$

where $\mathsf{hash}(\cdot)$ is an unkeyed hash function such as `sha256` and $\|$ is string concatenation. For state-of-the-art hash functions, it is believed to be computationally infeasible to find collisions or a pre-image x given $\mathsf{hash}(x)$.

4. DISCUSSION

We first discuss our prototypical implementation, and then consider future work.

4.1 Implementation

We have implemented a prototype of SAFE in the languages Perl and PHP. Our current system consists of the SFW compiler and the security reference monitor; we are using an Apache HTTP server, a standard application server, as middle tier. Our current system consists of roughly 3,000 lines of code.

SAFE currently works as follows. Our SFW compiler compiles each f-unit independently from other f-units and independently from the main web application because we want to give developers the ability to update a web application incrementally. Since our goal is to enable personalization, the code of the main application may in general not be available to a customizer, i.e., the compiler must be able to translate f-units separately from other f-units. The compilation of an f-unit creates a directory which is added to the working directory of the web server hosting the web application. The web server must be running in a PHP environment in order to handle customization dynamically.

The demonstrate the efficacy of our approach, we implemented a simple conference management system. Our system implements the full workflow of a basic conference with paper submission, the review cycle, and notification. However, the compiled code has over 3.6 times the size of the code specified by a developer using our framework. This shows the huge amount of code that developers usually specify although it is redundant or could be derived automatically.

We defined some customizations which users in the conference management system can choose from. A reference to the selected customization is stored as a cookie on the client. It was interesting to see how different browsers on one computer obtain different pages from the server, although both browsers had the same user logged in.

4.2 Future Work

While we clearly only have a proof of concept so far, we are excited about the many interesting open directions that our approach has created. We list a few in the next paragraphs.

Efficiency through Dynamic Code Partitioning. Due to different operation purposes, the code of an RIA's business logic should be split into client and server code in a *dynamic* manner [18, 5]. If, for instance, the client is a light-weight smartphone, then hard computation tasks shall better be performed on a powerful web server. However, sorting a small table shall better be performed locally in order to avoid communication overhead. These optimizations provide more autonomous and reactive user interfaces, but on the other hand, they may increase the need for more careful inspection of information flow and data privacy. So far, SAFE splits code automatically only with respect to security aspects. **Efficiency and Security through Dynamic Data Storage.** As done for automated code partitioning

in SAFE and in other approaches [18, 5], also *data* can be stored both at the client side and at the server side. A dynamic selection of the storage location can speed up the performance of RIAs due to reduced traffic between client and server.

Access Control. The current implementation of access control based on database columns per f-unit could become more fine-grained in that columns can be conditioned on certain values. **Declassified Customization.** In cases for which customization is not safe, the system provider has to manually approve the customization mapping. There is a need for an automated methodology to approve such declassified customizations.

Combining Extensions. A user may want to incorporate more than one extension from the "extension app store." What does it mean for two extensions to be "compatible," and how can we automatically check for compatibility between extensions?

Scalability of Concurrent Updates. It might be worth investigating how the CRM can operate in a distributed fashion serving thousands of clients at the same time.

Automatic Offline Mode. Even in the absence of the web server, the client part of an RIA shall be working up to a certain extend without crashes or major inconveniences.

5. RELATED WORK

Model-Driven Engineering (MDE)

The model-driven engineering approach structures the specification of an application by the abstract specification of individual domain models [16]. These formal models are well-suited for the design of distributed web applications: as in SAFE, both modularity and the compatibility of models with each other achieve reusability of code.

The MDE approach separates business logic (using platform independent models, PIM) from the technical aspects (using platform specific models, PSM). Executable code is derived from a (sub)set of these specified models. The compatibility of different such models allows for an efficient adaptation to different environments: A platform independent model for displaying sorted interactive student lists can be compiled to implementations for powerful server farms, but also for lightweight smartphones. However, our notion of customization is not covered by MDE since several models in MDE would have to change in order to customize a single module.

On the one hand, it is important to find the right level of abstraction, which is a key feature of the specified models. On the other hand, all models need to be formal enough. For instance, a specification in the unified modeling language (UML) is generally not fine-grained enough to automatically generate executable code. SAFE relies on domain-specific languages (DSL) for the implementation of individual f-units, but still provides abstraction in terms of f-unit interfaces.

Unified SQL-Based Approaches

In order to analyze information flow in web applications in a precise manner, application code has to be clustered to a certain number of units, each with a well-defined interface. The *activation framework* of Hilda [19] uses a unified SQL-like language to describe so-called application units. A tree of activated application units allows for realtime inspection of information flow and conflict detection. Upon activation of a child unit C by its parent unit P, information is propagated from P to C, and also from a database D. Upon some user action in C, the data associated with the action is returned to P, which processes this data and passes it to its parent unit, and so on. The root unit uniformly maintains the overall application state and updates the database.

While Hilda has several attractive features, it has several shortcomings. First, while its unifying language is good for developing semi-static applications, today's RIAs with JavaScript for interactivity are outside the scope of Hilda. Second, while the recursive activation along the tree is good to understand information flow, it requires that data for an application unit at the leaf of the activation tree be passed through all of its ancestors. Thus a simple modification of a leaf unit that for example displays an additional field from the database requires modification of all of its parent units in order to modify the information that is passed along. The activation framework of SAFE is inspired by Hilda, but it addresses Hilda's shortcomings in that SAFE supports traditional web development languages (and syntactically useful simplifications thereof), and SAFE simplifies information flow in the activation tree since f-units can directly access the database and SAFE never propagates data from a child f-unit to its parent f-unit.

FORWARD [7] is another web application framework, which – similar to Hilda – removes Java and JavaScript code fragments and replaces them with an SQL-like language. Although powerful Turing-complete languages accomplish often simple tasks that also SQL-like languages could accomplish, programmers are used to develop web applications using a certain set of languages different from pure SQL. Recent work [13] has shown that web developers actually prefer traditional imperative (scripting) languages such as PHP, JavaScript, and Java to model web applications as compared to an all-declarative approach as in FORWARD.

Specification Languages

Programming languages for web application development are often too low-level, and thus programmers spend too much effort on unimportant implementation details. A recent study [7] has shown that for one line of SQL code, there is a modest of 1.5 lines of Java code for business logic and 61 lines of Java code for binding SQL to Java and JavaScript. Most of this code can clearly be generated automatically. In addition, having to manually write low-level code introduces more bugs and security vulnerabilities. SAFE's high-level specification language SFW abstracts away most implementation details. Such a declarative language lets the application developer focus on *what* functionality shall be achieved, rather than *how* to achieve it. A carefully designed SFW compiler takes care of implementation details and provides much better code in terms of performance and security. The SFW language is oriented on what programmers have been using for decades. Yet another programming language would not be accepted by most programmers.

Other Related Work

There is a big number of frameworks addressing some of the problems mentioned in this paper. However, none of them covers the topic of customization. *Jaxer* [2] and *Phobos* [3] are web development frameworks that use the same set of Java/JavaScript-based languages for both browser and server thereby addressing the language heterogeneity problem. *Greasemonkey* [12] and *NoScript* [14] are extensions that allow the Firefox browser to locally customize the way a web page is rendered. These browser extensions are restricted to (1) the Firefox browser, to (2) JavaScript operations and (3) to web pages that are already delivered to the client. Customizations can only be shared via an external third channel, not via the web application itself.

In contrast, server-side application development is achieved by App2You [1], a graphical framework that allows users to create form-oriented web applications by outlining the pages of the application. The framework does not require programming experience or knowledge of web technologies. Our notion of customization is different from App2You's view of creating customized web applications: applications are derived from templates (App2You) instead of customized after deployment (SAFE). SproutCore [4] is a framework for web applications having the business logic on the client side. SproutCore aims at availability and efficiency of client code, in particular for mobile devices that are not connected to an application server. As in SAFE, updates to HTML and CSS code are performed automatically. Hanus and Koschnicke have recently presented a framework [11] to support the development of web applications based on an entity-relationship model. As for SAFE, this approach ensures application state consistency. Applications are specified in the declarative modeling language Curry which provides a strong typing machinery. However, many programmers consider functional languages such as Curry cumbersome to use for web application.

6. CONCLUSIONS

We have presented SAFE, a new activation-based CASE framework for the development of web applications with support for safe extensibility and concurrency. SAFE not only eases the development of web applications tremendously, but also ensures certain security properties by design. We have implemented a prototypical compiler for SAFE and have modeled a course management system in SAFE. Our framework shows the efficacy of the first steps into a novel interesting direction.

This material is based upon work supported by the National Science Foundation under Grant IIS-1012593, the iAd Project funded by the Research Council of Norway, and a Google Faculty Award. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

7. REFERENCES

[1] App2you, `http://app2you.com/site`, 2012.
[2] Jaxer, `http://www.jaxer.org`, 2012.
[3] Phobos, `http://phobos.java.net`, 2012.
[4] SproutCore, `http://blog.sproutcore.com`, 2012.
[5] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Building secure web applications with automatic partitioning. *Comm. ACM*, 52(2):79–87, 2009.
[6] P. Fraternali, S. Comai, A. Bozzon, and G. T. Carughi. Engineering rich internet applications with a model-driven approach. *ACM Trans. on the Web*, 4(2), 2010.
[7] Y. Fu, K. W. Ong, Y. Papakonstantinou, and M. Petropoulos. The SQL-based all-declarative FORWARD web application development framework. In *CIDR '11*, 2011.
[8] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE on Trans. Soft. Eng.*, 22(2):120–131, 1996.
[9] N. Gupta, F. Yang, A. J. Demers, J. Gehrke, and J. Shanmugasundaram. User-centric personalized extensibility for data-driven web applications (demonstration paper). In *SIGMOD '07*, 2007.
[10] D. H. Hansson. Ruby on Rails, `http://rubyonrails.org`, 2012.
[11] M. Hanus and S. Koschnicke. An ER-based framework for declarative web programming. In *PADL '10: 12th Int. Symp. on Practical Aspects of Decl. Lang.*, 2010.
[12] A. Lieuallen, A. Boodman, and J. Sundström. Greasemonkey, `https://addons.mozilla.org/en-US/firefox/addon/greasemonkey`, 2012.
[13] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: A platform for secure distributed computation and storage. In *SOSP '09: 22nd ACM Symposium on Operating Systems Principles*, 2009.
[14] G. Maone. Noscript, `https://addons.mozilla.org/en-US/firefox/addon/noscript`, 2012.
[15] Microsoft Research. Microsoft's academic conference management service (CMT), `http://cmt.research.microsoft.com/cmt`, 2012.
[16] Object Management Group Inc. Model driven architecture (MDA), document ormsc/2001-07-01. http://omg.org/cgi-bin/doc?ormsc/2001-07-01, 2012.
[17] S. Subramanian, M. W. Hicks, and K. S. McKinley. Dynamic software updates: a vm-centric approach. In *PLDI '09: ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2009.
[18] F. Yang, N. Gupta, N. Gerner, X. Qi, A. J. Demers, J. Gehrke, and J. Shanmugasundaram. A unified platform for data driven web applications with automatic client-server partitioning. In *WWW '07: 16th Int. Conf. on World Wide Web*, 2007.
[19] F. Yang, J. Shanmugasundaram, M. Riedewald, J. Gehrke, and A. Demers. Hilda: A high-level language for data-driven web applications. In *ICDE '06*, 2006.