Weaving Web Applications with WebDSL (Demonstration)

Danny M. Groenewegen
Software Engineering Research Group,
Delft University of Technology, The Netherlands
d.m.groenewegen@tudelft.nl

Eelco Visser
Software Engineering Research Group,
Delft University of Technology, The Netherlands
visser@acm.org

Abstract

WebDSL is a domain-specific language for the development of web applications that integrates data-models, user-interface models, actions, validation, access control, and workflow. The compiler verifies the consistency of applications and generates complete implementations in Java or Python. We illustrate the key concepts of the language with a small web application.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.3.4 [*Programming Languages*]: Processors

General Terms Languages

Keywords domain-specific languages, web application model, data model, data binding, access control

1. Motivation

The implementation of web applications comprises many technical concerns, including data representation, querying, and modification, user input, data validation, user interface design, and navigation. These concerns are often addressed by separate languages. For example, in (one configuration of) the Java web programming platform we find the Java general purpose programming language, the SQL query language (or some dialect such as HQL), the JavaServer Faces (JSF) presentation language with the EL expression language for accessing data, the CSS stylesheet language, and other XML schemas for configuration such as page flow declarations.

While separation of concerns and 'choosing the right language for the job' are conceptually appealing, the amalgam of languages used in a single web application project are typically poorly integrated, with an adverse effect on productivity and software quality caused by boilerplate code, loose coupling, and a lack of static verification.

2. WebDSL

WebDSL [3] is a domain-specific language for the development of web applications that integrates data models, user interface models, actions, styling, access control [1], data validation, and workflow [2]. While these different concerns are supported by separate domain-specific sub-languages, the static semantics of the language verifies the consistency of the different concerns of an application model. The WebDSL compiler generates a complete implementation in Java or Python without the need to write further code in these languages.

3. Example: WebTasks

We illustrate the features of WebDSL with a small web application for managing tasks (WebTasks). During the demonstration we will build this application from scratch, while introducing the concepts of the language. The next page illustrates the main concepts with a fragment of the WebTasks application. Figure 1 shows screenshots of three types of pages from the WebTasks application. Figures 2 to 7 show the WebDSL code for these pages.

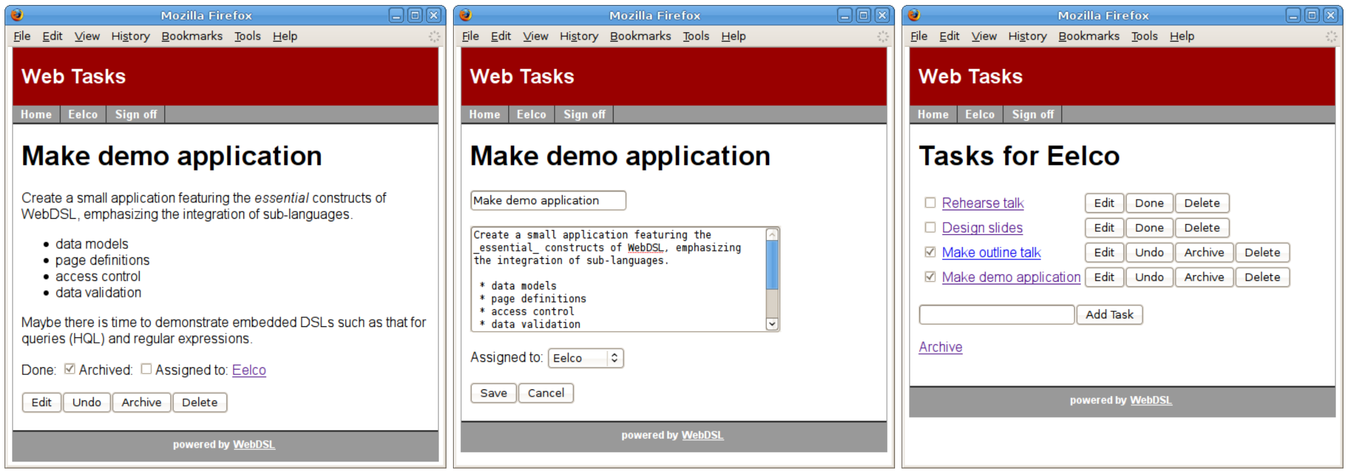
Acknowledgments This research was supported by NWO/JACQUARD project 638.001.610, *MoDSE: Model-Driven Software Evolution.*

References

[1] D. M. Groenewegen and E. Visser. Declarative access control for WebDSL: Combining language integration and separation of concerns. In D. Schwabe and F. Curbera, editors, *Eighth International Conference on Web Engineering (ICWE 2008)*, pages 175–188. IEEE CS Press, July 2008. best paper award.

[2] Z. Hemel, R. Verhaaf, and E. Visser. WebWorkFlow: An object-oriented workflow modeling language for web applications. In *Model Driven Engineering Languages and Systems (MODELS 2008)*, volume 5301 of *LNCS*, pages 113–127. Springer, 2008.

[3] E. Visser. WebDSL: A case study in domain-specific language engineering. In *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, volume 5235 of *LNCS*, pages 291–373, Heidelberg, October 2008. Springer.



Make demo application

Create a small application featuring the *essential* constructs of WebDSL, emphasizing the integration of sub-languages.

- data models
- page definitions
- access control
- data validation

Maybe there is time to demonstrate embedded DSLs such as that for queries (HQL) and regular expressions.

Done: ☑ Archived: ☐ Assigned to: Eelco

Make demo application

```
Create a small application featuring the
_essential_ constructs of WebDSL, emphasizing
the integration of sub-languages.

 * data models
 * page definitions
 * access control
 * data validation
```

Assigned to: Eelco

Tasks for Eelco

☐ Rehearse talk	Edit Done Delete
☐ Design slides	Edit Done Delete
☑ Make outline talk	Edit Undo Archive Delete
☑ Make demo application	Edit Undo Archive Delete

(a) task page (b) task edit page (c) task list

Figure 1. Screenshots of the WebTasks application.

```
entity User {
  username :: String (id, name,
              validate(isUniqueUser(this), "Username is taken"))
  password :: Secret
  tasks    -> List<Task>
  todo     -> List<Task>
           := [t | t : Task in this.tasks where !(t.archived)]
}
entity Task {
  name        :: String (name)
  description :: Text
  done        :: Bool
  archived    :: Bool
  user        -> User (inverse=User.tasks)
}
```

Figure 2. *Data model* defines entities with properties. Entity declarations are mapped to a database schema and objects are automatically persisted to the database. Validation constraints (`username`) pose extra requirements on entities. A derived property (`todo`) is a transient property whose value is computed from other properties.

```
define page task(task : Task) { main{
  section{ header{output(task.name)}
    output(task.description)
    par{ "Done: " output(task.done)
         "Archived: " output(task.archived)
         "Assigned to: " output(task.user) }
    manageTask(task) } } }
```

Figure 3. *Page definition* (Fig 1(a)) defines view of properties of the parameter objects of the page.

```
define page edittask(task : Task) { main{
  section{ header{output(task.name)}
    form{
      par{ input(task.name) }
      par{ input(task.description) }
      par{ "Assigned to: " input(task.user) }
      action("Save", save())
      navigatebutton(task(task), "Cancel")
      action save() { return task(task); } } } } }
```

Figure 4. Data input forms (Fig 1(b)) provide automatic *data binding* of form fields to entity properties.

```
define page tasks(user : User) { main{
  section{
    header{"Tasks for " output(user.username) }
    table{ taskList(user.todo) }
    par{ addTask(user) }
    par{ navigate(archive(user)){"Archive"} } } } }
```

Figure 5. Page definition for user task list (Fig 1(c)) with *navigation* to `archive` page.

```
define taskList(tasks : List<Task>) {
  for(task : Task in tasks) {
    row{ output(task.done) output(task) manageTask(task) } } }

define addTask(user : User) {
  var newTask : Task := Task{ done := false }
  action addtask() { user.tasks.add(newTask); newTask.save(); }
  form{ input(newTask.name) action("Add Task", addtask()) } }

define manageTask(task : Task) {
  action done() { task.done := true; }
  action undo() { ... }
  action delete() {
    var user := task.user; user.tasks.remove(task);
    task.delete(); return tasks(user); }
  form{
    navigatebutton(edittask(task), "Edit")
    if(!task.done) { action("Done", done()) }
    else { action("Undo", undo()) ... }
    action("Delete", delete())
  }
}
```

Figure 6. *Template definitions* define page fragments that can be reused in multiple page definitions.

```
access control rules

  principal is User with credentials username, password

  rule template manageTask(task : Task) {
    securityContext.loggedIn
    && task.user == securityContext.principal
  }
```

Figure 7. *Access control rules* restrict access to pages, templates, or actions using Boolean constraints on the data model.