Integration of Data Validation and User Interface Concerns in a DSL for Web Applications

Danny M. Groenewegen and Eelco Visser

Software Engineering Research Group, Delft University of Technology, The Netherlands
d.m.groenewegen@tudelft.nl, visser@acm.org

Abstract. Data validation rules constitute the constraints that data input and processing must adhere to in addition to the structural constraints imposed by a data model. Web modeling tools do not address data validation concerns explicitly, hampering full code generation and model expressivity. Web application frameworks do not offer a consistent interface for data validation. In this paper, we present a solution for the integration of declarative data validation rules with user interface models in the domain of web applications, unifying syntax, mechanisms for error handling, and semantics of validation checks, and covering value well-formedness, data invariants, input assertions, and action assertions. We have implemented the approach in WebDSL, a domain-specific language for the definition of web applications.

1 Introduction

The engineering of web applications requires catering for a number of different concerns including data models, user interfaces, actions, data validation, and access control. In the mainstream technology for web application development these concerns are supported by loosely coupled languages that require abundant boilerplate code and lack static verification. The domain-specific language engineering challenge for the web application domain [21] is to realize a concise, high-level, declarative language for the definition of web applications in which the various concerns are supported by specialized sub-languages, yet linguistically integrated, and from which implementations can be derived automatically. This requires investigation and understanding of, and the design of appropriate domain-specific languages for each of the sub-domains of the web application domain. Moreover, it requires the seamless linguistic integration of these separate languages that ensures the consistency of models in the different domains and that leverages their combination. This research program is relevant for the discovery of good abstractions for the web engineering domain. It is also relevant as a case study in the systematic development of families of domain-specific languages.

In previous work we have studied the domains of data models and user interface definitions [21], access control [6], and workflow [7], the results of which have been implemented as sub-languages of the WebDSL language [22]. In this paper, we address the domain of data validation and its interaction with the user interface.

The core of a data-intensive web application is its data model. The web application must be organized to preserve the consistency of data with respect to the data model during updates, deletes, and insertions. The core consistency properties of a data model are formed by structural constraints, that is, the data members of and relations between entities. Some consistency properties cannot be expressed as structural constraints. Furthermore, some data integrity constraints do not pertain directly to persistent data.

Data validation rules constitute the constraints that data input and processing must adhere to in addition to the structural constraints imposed by the data model.

A high-level web engineering solution should provide a uniform and declarative validation model that integrates with the other relevant technical models. In addition to ensuring data consistency by enforcing a validation model, the integration of data validation in a web application requires a mechanism for reporting constraint violations to the user, indicating the origin of the violation in the user interface with a sensible error message and consistent styling. Model-driven methodologies such as OOHDM [18], WebML [4], UWE [10], OOWS [15], and Hera [20] do not make data validation concerns explicit in their models. When generating code from models, as demonstrated for UWE [11], WebML [2], and Hera [5], validating data requires an escape from model to code, hampering full code generation and model expressivity.

In this paper, we present a language design that integrates declarative data validation rules with user interface models in the domain of web applications, unifying syntax, mechanisms for error handling, and semantics of validation checks, and that covers value well-formedness, data invariants, input assertions, and action assertions. We have implemented the approach in WebDSL [21], a domain-specific language for the definition of web applications. The main contributions of this paper are (1) the design of abstractions for data validation in web applications for concise and uniform specification of value well-formedness, data invariants, input assertions, and action assertions, (2) the seamless integration of data validation rules and user interface definitions, and (3) an example of the integration of models for multiple technical domains.

In the next section we give a brief introduction to WebDSL and the running example used in the rest of the paper. Section 3 discusses validation features necessary for web applications, namely value well-formedness, data invariants, input assertions, and action assertions. Section 4 discusses related and future work, and Section 5 concludes.

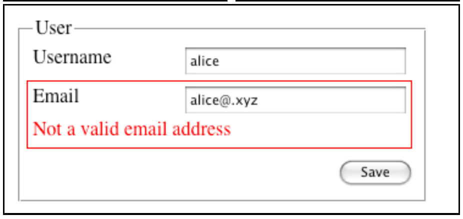
2 WebDSL

WebDSL [21] is a domain-specific language for the development of web applications that integrates data models, user interface models, user interface actions, styling, access control [6], and workflow [7]. While these different concerns are supported by separate domain-specific sub-languages, the static semantics of the language enforces the integrity of the different concerns of an application model. What distinguishes WebDSL from web application frameworks in general purpose languages [9,13,16] is static verification and abstraction from accidental complexity (boilerplate code). Compared to web modeling tools [19,11,14,2], WebDSL combines high expressivity with good coverage (customization options). The WebDSL compiler generates a complete implementation in Java or Python.

In this section we give an overview of the features of WebDSL needed in this paper and introduce the running example used to discuss data validation in this paper. We illustrate the various categories of data validation with a small user management application. The example application consists of two data model *entities*, namely `User`

```
entity User {
  username :: String
  email :: Email
}
```

```
entity UserGroup {
  name :: String (id)
  members -> Set<User>
}
```



Not a valid email address

```
define page editUser(u:User) {
  form {
    group("User") {
      label("Username") { input(u.username) }
      label("Email") { input(u.email) }
      action("Save", save())
    }
  }
  action save() {
    return user(u);
  }
}
```

Fig. 1. Value well-formedness for `Email` type

and `UserGroup` (Fig. 1). Data model definitions describe the persistent data model in a WebDSL application. Data model entities consist of *properties* with a name and a type. Types of properties are either value types (indicated by `::`) or associations to other entities defined in the data model. Value types are basic data types such as `String` and `Int`, but also domain-specific types such as `Email` that carry additional functionality. Associations are *composite* (the referer owns the object, indicated by `<>`) or *referential* (the object may be shared, indicated by `->`). Associations can be to *collections* such as `Set` or `List`, demonstrated by the `members` property of the `UserGroup` entity.

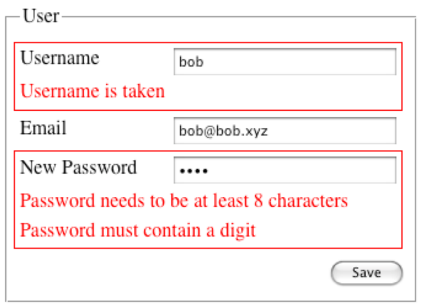
Page definitions in WebDSL describe the web pages that allow users to view and modify data model entities. Page definitions consist of the name of the page, the names and types of the objects passed as parameters, and a presentation of the data contained in the parameter objects. For example, the `editUser(u:User)` definition in Fig. 1 creates a page for editing the properties of `User` entity `u`. WebDSL provides basic markup operators such as `group` and `label` for defining the structure of a page. Navigation is realized using the `navigate` element, which takes a link text and a page with parameters as arguments. Furthermore, page definitions can be reused by declaring them as template. Templates can be included in page definitions by supplying the associated parameters. In addition to *presenting* data objects, pages can also *modify* objects. For example, the content of a `User` entity can be modified with the `editUser` page. The page element `input(u.username)` declares an appropriate form input element based on the type of its argument; in this case a text field. A data modification is finalized by means of an *action*, which can apply further modifications to the objects involved. For example, in the `save` action the changes to the `User` are saved. The `return` statement of an action is used to realize *page flow* by specifying the page and its arguments where the browser should be directed after finishing the action.

3 Validation Abstractions

Data validation is required in multiple contexts in web applications. In this section we distinguish four variants, show how these are expressed in WebDSL using declarative data validation rules, and how error messages are integrated in the user interface.

```
entity User {   username :: String (id)   password :: Secret   email :: Email   }
```

```
extend entity User {
  username(validate(isUnique(),"Username is taken"))
  validate(password.length >= 8, "Password needs to be at least 8 characters")
  validate(/[a-z]/.find(password), "Password must contain a lower-case character")
  validate(/[A-Z]/.find(password), "Password must contain an upper-case character")
  validate(/[0-9]/.find(password), "Password must contain a digit")
}
```



User

Username	bob
Username is taken	
Email	bob@bob.xyz
New Password	••••
Password needs to be at least 8 characters	
Password must contain a digit	

```
define page editUser(u:User) {
  form {
    group("User") {
      label("Username"){ input(u.username) }
      label("Email"){ input(u.email) }
      label("New Password") {
        input(u.password)
      }
      action("Save", save())
    }
  }
  action save() {
    return user(u);
  }
}
```

Fig. 2. Data invariants for `User` entity validation

3.1 Value Well-Formedness

Value well-formedness checks verify that a provided input value conforms to the value type. In other words, the conversion of the input value from request parameter to an instance of the actual type must succeed. This type of validation is usually provided by libraries or frameworks. However, it has to be declared explicitly, and possibly at each input of a value of the type. In WebDSL, value well-formedness rules are checked automatically. WebDSL supports types specific for the web domain, including `Email`, `URL`, `WikiText`, and `Image`. Automatic value well-formedness constraints for all value types provides decent input validation by default. Moreover, these built-in type validation checks and messages can be customized in an application.

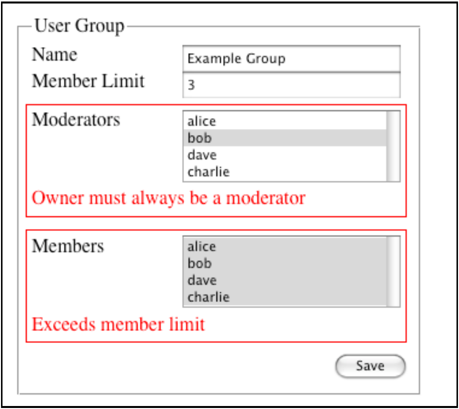
The `editUser` page in Fig. 1 consists of a form with labeled inputs for the `User` entity properties. The `save` action persists the changes to the database, provided that all validation checks succeed. (Changes to existing entities are automatically stored in WebDSL, new entities need to be saved explicitly using the `save()` method.) Since well-formedness validation checks are automatically applied to properties, the `email` property is validated against its well-formedness criteria. The result of entering an invalid email address is shown in the screenshot: a message is presented to the user and the action is not executed.

3.2 Data Invariants

Data invariants are constraints on the data model, i.e. restrictions on the properties of data model entities. These validation rules can check any type of property, such as a reference, a collection, or a value type. By declaring validation in the data model, the validation is reused for any input or operation on that data. In Ruby on Rails [16] data invariants can be defined in a 'validate' method of the active record class, which

```
entity UserGroup {   name :: String (id)        owner -> User          memberLimit :: Int
                     moderators -> Set<User>    members -> Set<User>   }
```

```
extend entity UserGroup {
  validate(owner in moderators, "Owner must always be a moderator")
  validate(owner in members, "Owner must always be a member")
  validate(members.length <= memberLimit, "Exceeds member limit")
}
```


User Group

Name	Example Group
Member Limit	3
Moderators	alice, bob, dave, charlie

Owner must always be a moderator

Members	alice, bob, dave, charlie

Exceeds member limit

```
define page editUserGroup(ug:UserGroup) {
  form {
    group("User Group") {
      label("Name") { input(ug.name) }
      label("Member Limit") {
        input(ug.memberLimit)
      }
      label("Moderators") {
        input(ug.moderators)
      }
      label("Members") { input(ug.members) }
      action("Save", save())
    }
  }
  action save() {
    return userGroup(ug);
  }
}
```

Fig. 3. Data invariants for UserGroup entity validation

then gets called by the framework when validation is required. Multiple checks in a validation method tangle validation for different properties. The Seam [9] framework supports the specification of data invariants declaratively through annotations. However, these annotations consist of a limited number of built-in checks and an escape to specify a custom class that handles validation for a property. In the worst case each validation rule needs a separate class, incurring the syntactic overhead of Java class declarations several times.

Validation rules in WebDSL are of the form `validate(e,s)` and consist of a Boolean expression `e` to be validated, and a String expression `s` to be displayed as error message. Any globally visible functions or data can be accessed as well as any of the properties and functions in scope of the validation rule context.

Validation checks on the data model are performed when a property on which data validation is specified is changed and when the entity is saved or updated. Validation is connected to properties either by adding the validation in the property annotation or by referring to a property in the validation check. More specific validation checks are supported which are only checked when the entity is in a certain state, these are `validatesave`, which is checked when an entity is saved for the first time, `validateupdate`, checked on any update, and `validatedelete`, checked before deleting the entity. The validation mechanism takes care of correctly presenting validation errors originating from the data model. For form inputs causing data invariant violations the message is placed at the input being processed. When data model validation fails during the execution of an action, the error is shown at the corresponding button.

Fig. 2 presents an extended `User` entity with several invariants and a `password` property. The `username` property has the `id` annotation, which indicates the property is unique and can be used to identify this entity type. The isUnique member function (a generated function that takes into account the existence of an 'id' property) is called to verify this constraint. The password property is annotated with validation rules that express requirements for a stronger password. By declaring validation rules in the entity, explicit checks in the user interface can be avoided. Both the WebDSL page definition and the resulting web application page are shown below the entity definition.

Fig. 3 shows more advanced validation rules, which express dependencies between the properties of an entity. The UserGroup entity is extended with an owner reference, a moderators set, and a memberLimit value. The editUserGroup page allows the owner to edit some of the UserGroup properties. The validation rule on the moderators set expresses that the owner should always be in this set of moderators (similarly, the owner should always be a member). The member set is constrained in size based on the memberLimit value. Validation rules that cover multiple properties, such as the 'owner in moderators' check, are performed for all input components of properties the validation is specified on. However, the checks can be added to a single property as well, in order to specialize the error message.

3.3 Input Assertions

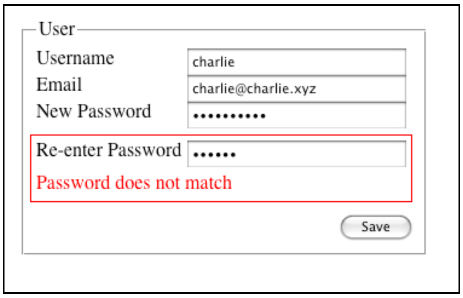
Input assertions are necessary when the validation rule targets an input that is not directly connected to the persisted data model. These types of constraints are easy to address in the form environment itself. For example, a validation check in XForms [1] verifies properties of the entered form data. The model in XForms, on which validation is specified, is a model of the input data produced by the form. Unfortunately, such form validation solutions are not integrated with validation on the application data model. For example, an input for an entity produces the identifier as form data, in the XForms model it is just a String, but in the application data model it is an entity reference.

Validation checks in WebDSL pages have access to all variables in scope, including page variables and page arguments. The placement and order of validation rules does not influence the results of the checks. Visualization of errors resulting from validation in forms are placed at the location of the validation declaration. Usually such a validation rule is connected to an input, which can be expressed by placing the validation rule as a child element of input.

The example in Fig. 4 demonstrates the final addition to the user edit form, an extra password input field in which the user must repeat the entered password. This validation cannot be expressed as a data invariant, since the extra password field is not part of the User entity. Therefore, the rule is expressed in the form directly, where it has access to the page variable p. This variable contains the repeated password whereas the first password entry is saved in the password field of User entity u. When entering a different value in the second field the validation error is presented, as can be seen in the screenshot.

3.4 Action Assertions

Action assertions are predicate checks at any point in the execution of actions and functions for verification during the processing of inputs. The action processing needs to be



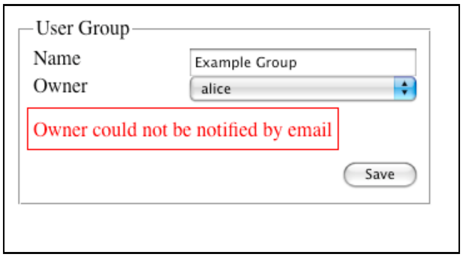
User

Username	charlie
Email	charlie@charlie.xyz
New Password	••••••••••
Re-enter Password	••••••

Password does not match

```
define page editUser(u:User) {
  var p: Secret;
  form {
    group("User") {
      label("Username") { input(u.username) }
      label("Email") { input(u.email) }
      label("New Password") {
        input(u.password) }
      label("Re-enter Password") { input(p) {
        validate(u.password == p,
        "Password does not match") }
      }
      action("Save", action{ } ) } } }
```

Fig. 4. Form validation with input assertions



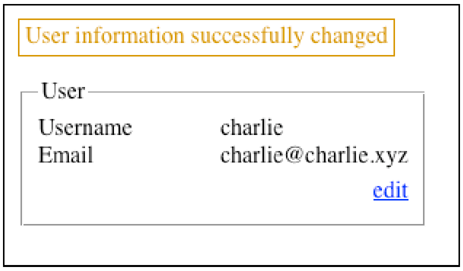
User Group

Name	Example Group
Owner	alice

Owner could not be notified by email

```
define page createGroup() {
  var ug := UserGroup {}
  form {
    group("User Group") {
      label("Name") { input(ug.name) }
      label("Owner") { input(ug.owner) }
      action("Save", save()) } }
  action save() {
    validate(email(newGroupNotify(ug))
    ,"Owner could not be notified by email");
    return userGroup(ug); } }
```

Fig. 5. Action assertions for `UserGroup` creation



User information successfully changed

User

Username	charlie
Email	charlie@charlie.xyz

```
define page editUser(u:User) {
  ... form ...
  action save() {
    message("User information success...");
    return user(u);
  } }
```

```
define page user(u:User) {
  group("User") {
    label("Username") { output(u.username) }
    label("Email") { output(u.email) }
    navigate(editUser(u)) {"edit"} } }
```

Fig. 6. Success message

aborted, reverting any changes made, and the validation message has to be presented in the user interface. This type of validation is not directly supported in existing solutions, requiring an investment in finding appropriate hooks in the implementation. For example, Ruby on Rails [16] assumes validation is specified in data model classes, errors are passed through those model classes and the form mechanism is built around that. There is no mechanism for a validation check as part of a controller action, this requires a low-level encoding that passes the check result and error message, or wrapping validation in a data model class.

WebDSL supports this type of validation transparently using the same validation rules. The errors resulting from action assertion failures are displayed at the place the execution originated, e.g. above the submit button which triggered the erroneous action.

Fig. 5 provides an example of action assertion. On the right is a page definition for a `createGroup` page which allows creating new `UserGroup` entities. The constraint expressed in the `save` action is that creating a new group requires email notification to the specified owner (which might not be the user executing this operation). The `newGroupNotify` email definition retrieves an email address from its `UserGroup` argument (through `ug.owner.email`) and tries to send a notification email to the owner of the new group. When this fails, for instance because there is no mail server responding to the email address, the call returns false and the validation check produces the error. This result is shown on the left in the screenshot.

Generic error handling, such as problems with a database commit, can also be expressed using action assertions. The web application can then display an error message in the form instead of redirecting to a default error page.

3.5 Messages

This section has described assertions that report erroneous behavior in actions. Related to such action assertions, is a generic messaging mechanism for giving feedback about the correct execution of an action. This requires a place to show messages, for instance by adding a default message template at the top of each page. Furthermore, the message should be declared in the action code. An example of such messaging is shown in Fig. 6. The `save` action of the `editUser` page gives a message to the page redirected to, namely `user`. The result of the executed action is shown on the left.

3.6 Validation Mechanics

A page request in WebDSL is processed in the following five phases: *Convert request parameters*: check value well-formedness validation rules for page arguments and input parameters, then convert these to the correct types. *Update model values*: check data invariants for input data, and then insert in data model entities. *Validate forms*: check input assertions in page definitions. *Handle actions*: perform action, abort if an action assertion fails (in that case no changes are made to the data model). *Render or redirect*: show page, including produced validation errors. Redirect if an action executed successfully.

4 Discussion

Web Modeling Tools Several model-driven methodologies for creating web applications have been proposed in recent years, including OOHDM [18], SHDM [12], WebML [4], UWE [10], OOWS [15], and Hera [20]. WebDSL goes beyond being a methodology for designing web applications and providing a path to actual implementation by leveraging full code generation. The transformation from problem space to solution space is completely automated. In this paragraph we discuss how these methodologies and their tools relate to WebDSL in general, and data validation integration in particular.

The Hera Presentation Generator [5] allows modeling forms to support editing data in the session. The persisted domain data of the application cannot be changed. Hera-S [19] also incorporates persisting form input data through update queries. The only example in the paper of such an update shows incrementing a view counter, a simple operation that does not process form input data. Kraus et al. [11] present the generation of partial web applications from UWE models. An application skeleton is generated including JSP pages and navigation between them. Forms and input data are not discussed, which probably means it is part of the custom code. HyperDe [14] is a tool that allows online creation of web applications designed with the SHDM method. The paper shows an example of an input field for a person's email address. This involves manual construction of data binding (showing the email and reading it from the submit data) and does not indicate how validation of that input can be performed. WebRatio [2] is a tool for generating web applications based on the WebML method. The conceptual WebML models do not model data validation concerns, while WebRatio does have form validation features. These can be directly mapped to validation features in the underlying Struts [3] framework. Validation which goes beyond the form, such as querying the database, has to be implemented in a Struts validator class. This implementation requires intricate knowledge of the translation process and implementation platform. From our study of the literature we conclude that declarative modeling of data validation is ignored in model driven web engineering. As a result, validation concerns require an escape from model to code, hampering full code generation and model expressivity.

Future Work. The current validation model focuses on verifying that the data satisfies a set of constraints. Actions that break these constraints are forbidden and result in an error message. An alternative approach would be to solve constraints automatically [8] and *repair* data so that it complies with the constraints or to suggest such repairs to the user. Since most inputs in web application forms are strings, expressivity of validation rules could be increased by incorporating a domain-specific language for string constraints. Scaffidi et al. [17] demonstrate that parsing technology can provide rich string input validation and feedback.

5 Conclusion

The domain-specific language engineering challenge for the web application domain [21] is to realize a concise, high-level, declarative language for the definition of web applications in which the various concerns are supported by specialized sub-languages, yet linguistically integrated, and from which implementations can be derived automatically. This paper presents a solution for the integration of data validation, a vital component of web applications, into a web application DSL that includes data models, user interfaces, and actions. This solution unifies syntax, mechanisms for error handling, and semantics for data validation checks covering value well-formedness, data invariants, input assertions, and action assertions. Our approach improves over current web modeling tools by providing declarative data validation rules from which a complete implementation is generated. Unlike web application frameworks, our solution supports different kinds of data validation uniformly. The integration of data validation rules into WebDSL, a web application DSL that supports data models, user interfaces, and actions, allows web application developers to take a truely model-driven approach to the design of web applications, concentrating on the logical design of an application rather than the accidental complexity of low-level implementation techniques.

References

1. Boyer, J.M. (ed.): XForms 1.0, 3rd edn. W3C Recommendation (2007)
2. Brambilla, M., Comai, S., Fraternali, P., Matera, M.: Designing web applications with WebML and WebRatio. In: Web Engineering: Modelling and Implementing Web Applications, pp. 221–260 (2007)
3. Brown, D., Davis, C., Stanlick, S. (eds.): Struts 2 in Action. Manning Publ. Co. (2008)
4. Ceri, S., Fraternali, P., Bongio, A.: Web Modeling Language (WebML): a modeling language for designing Web sites. Computer Networks 33(1-6), 137–157 (2000)
5. Frasincar, F., Houben, G., Barna, P.: HPG: the Hera Presentation Generator. Journal of Web Engineering 5(2), 175 (2006)
6. Groenewegen, D.M., Visser, E.: Declarative access control for WebDSL: Combining language integration and separation of concerns. In: Schwabe, D., Curbera, F. (eds.) International Conference on Web Engineering (ICWE 2008), July 2008, pp. 175–188 (2008)
7. Hemel, Z., Verhaaf, R., Visser, E.: WebWorkFlow: An object-oriented workflow modeling language for web applications. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 113–127. Springer, Heidelberg (2008)
8. Järvi, J., Marcus, M., Parent, S., Freeman, J., Smith, J.N.: Property models: from incidental algorithms to reusable components. In: GPCE, pp. 89–98 (2008)
9. Kittoli, S. (ed.): Seam - Contextual Components. A Framework for Enterprise Java. Red Hat Middleware, LLC (2008)
10. Koch, N., Kraus, A., Hennicker, R.: The authoring process of the UML-based web engineering approach. In: Web-Oriented Software Technology (2001)
11. Kraus, A., Knapp, A., Koch, N.: Model-driven generation of web applications in UWE. In: Model-Driven Web Engineering (MDWE 2007), Como, Italy (July 2007)
12. Lima, F., Schwabe, D.: Application modeling for the semantic web. In: Latin American-Web Congress (LA-WEB 2003), Washington, DC, USA, p. 93. IEEE Computer Society, Los Alamitos (2003)
13. MacDonald, M., Szpuszta, M.: Pro ASP. NET 3.5 in C# 2008. Apress (2007)
14. Nunes, D., Schwabe, D.: Rapid prototyping of web applications combining domain specific languages and model driven design. In: International Conference on Web Engineering (ICWE 2006), pp. 153–160 (2006)
15. Pastor, O., Fons, J., Pelechano, V.: OOWS: A method to develop web applications from web-oriented conceptual models. In: Web Oriented Software Technology (IWWOST 2003), pp. 65–70 (2003)
16. Ruby, S., Thomas, D., Heinemeier Hansson, D.: Agile Web Development with Rails, 3rd edn. Pragmatic Programmers (2009)
17. Scaffidi, C., Myers, B.A., Shaw, M.: Topes: reusable abstractions for validating data. In: ICSE 2008, pp. 1–10 (2008)
18. Schwabe, D., Rossi, G., Barbosa, S.: Systematic hypermedia application design with OOHDM. In: Proceedings of the the seventh ACM conference on Hypertext, pp. 116–128. ACM, New York (1996)
19. van der Sluijs, K., Houben, G., Broekstra, J., Casteleyn, S.: Hera-S: web design using sesame. In: International Conference on Web Engineering (ICWE 2006), pp. 337–344 (2006)
20. Vdovjak, R., Frasincar, F., Houben, G., Barna, P.: Engineering semantic web information systems in Hera. Journal of Web Engineering 2, 3–26 (2003)
21. Visser, E.: WebDSL: A case study in domain-specific language engineering. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) Generative and Transformational Techniques in Software Engineering II. LNCS, vol. 5235, pp. 291–373. Springer, Heidelberg (2008)
22. Visser, E., et al.: WebDSL, 2007–2009, `http://webdsl.org`