Delft University of Technology
Software Engineering Research Group
Technical Report Series

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

Danny M. Groenewegen, Eelco Visser

Report TUD-SERG-2010-033

TUD-SERG-2010-033

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

This paper is a pre-print of:

Danny M. Groenewegen, Eelco Visser. Integration of Data Validation and User Interface Concerns in a DSL for Web Applications. In Geert-Jan Houben, Nora Koch, Gustavo Rossi, and Antonio Vallecillo, editors, *Journal on Software and Systems Modeling - Theme Issue on Model-Driven Web Engineering*, Springer, 2010.

```
@inproceedings{GV10,
  title = {Integration of Data Validation and User Interface Concerns
           in a DSL for Web Applications},
  author = {Danny M. Groenewegen and Eelco Visser},
  journal = {Journal on Software and Systems Modeling - Theme Issue on
             Model-Driven Web Engineering},
  editor = {Geert-Jan Houben and Nora Koch and Gustavo Rossi
            and Antonio Vallecillo},
  year = {2010},
  publisher = {Springer},
}
```

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

Danny M. Groenewegen, 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 make all types of data validation explicit in their models, 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 [31] 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 [31], access control [13], and workflow [15], the results of which have been implemented as sub-languages of the WebDSL language [32]. In this paper, we address the domain of data validation and its interaction with the user interface. This paper is an extension of the identically titled short paper presented at the Second International Conference on Software Language Engineering (SLE 2009) [14].

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. Four essential kinds of data validation in web applications are: *value well-formedness rules*, *data invariants*, *input assertions*, and *action assertions*. *Value well-formedness rules* define the syntax of values of particular entity properties. Value properties are typically stored as strings, but may have to satisfy additional well-formedness constraints. For example, email addresses and zip codes cannot be arbitrary strings, but must conform to a particular syntax. *Data invariants* are functional constraints of single properties that cannot be expressed as syntactic constraints of single values, or coordination constraints between properties. For example, uniqueness of a primary key property, or a size limit on a collection property encoded by an integer property. *Input assertions* are checks based on input data that is not connected (directly) to persistent data, and hence cannot be specified as constraints on the data model. For example, a double password entry field requires the user to enter the same password twice to prevent typos. Only one of the password fields is actually stored in the data model. Thus, the equality constraint cannot be expressed as a data model invariant. Finally, *action assertions* are checks on the execution of operations. For example, the conclusion of an operation might notify the user by email. An assertion may require the notification to succeed.

Data validation rules are commonly used in combination with data models, e.g. OCL [1] constraints that specify class member invariants in UML class models. For other technical domains such as user interfaces and actions, where ordering and control flow is involved, data validation rules are usually implicitly defined in low-level code. Moreover, integration with these domains often results in separate mechanisms for data validation, having an adverse effect on application consistency. In addition to declarative and consistent notation, another issue is consistent error reporting. Validation feedback should be handled transparently, but also materialize in a way that fits naturally into the user interface.

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 [28], WebML [8], UWE [19], OOWS [25], and Hera [30] do not make all types of data validation explicit in their models. When generating code from models, as demonstrated for UWE [20], WebML [5], and Hera [11], validating data requires an escape from model to code, hampering full code generation and model expressivity. Web application frameworks provide high-level support for a subset of data validation tasks. For instance, Ruby on Rails [26] provides support for data invariants, whereas Java Server Faces (JSF) [7] supports input assertions, validating forms directly without taking the relation to the data model into consideration. In such approaches, the expression of other types of validation requires a fall back to low-level coding to implement the checking

of rules as well as the handling of error messages. ASP.NET [22] supports both input assertions and data invariants, but with an inconsistent interface.

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 [31], 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, (3) an example of the integration of models for multiple technical domains, and (4) an application of compilation by normalization for achieving a flexible and extensible implementation.

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 the inclusion of data validation in the WebDSL request lifecycle. Section 5 describes the implementation of data validation which is based on compilation by normalization. Section 6 evaluates the approach. Section 7 discusses related and future work, and Section 8 concludes.

2 WebDSL

WebDSL [31] is a domain-specific language for the development of web applications that integrates data models, user interface models, user interface actions, styling, access control [13], and workflow [15]. 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 [18, 22, 26] is static verification and abstraction from accidental complexity (boilerplate code). Compared to web modeling tools [29, 20, 24, 5], WebDSL combines high expressivity with good coverage (customization options). The WebDSL compiler generates a complete implementation in Java.

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` and `UserGroup` (Figure 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.

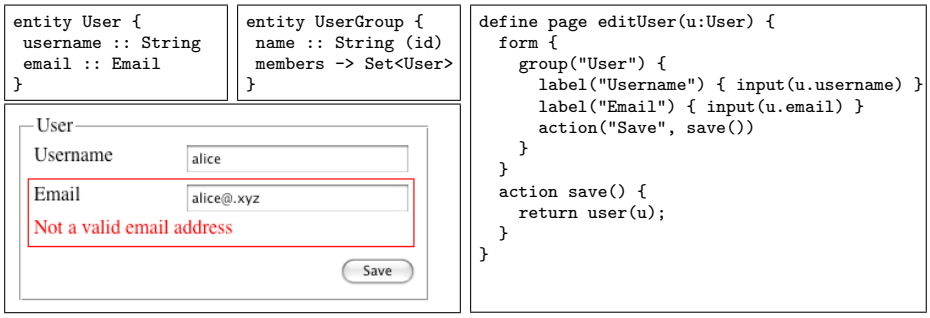


Fig. 1. Value well-formedness for `Email` type.

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 Figure 1 creates a page for editing the properties of `User 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` object are saved. Changes to existing entities are automatically stored, new entities need to be saved explicitly using the built-in `save()` entity method. 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.

In previous work we introduced the core components of WebDSL — pages, entities and actions — and described their implementation with transformations to the Seam Java framework [31]. Data validation is identified as part of the web application domain, but it is not addressed in the design. In our work on access control [13] we analyze various access control policies and show how to implement them in the WebDSL access control sub-language. Access control definitions are implemented by transformation to the existing core language. We have also examined workflow abstractions [15], which are built on top of the core and access control languages. Since we encountered problems in the 'translate to existing framework' approach [12], we changed to a custom framework which matches WebDSL better. While data validation can be encoded using checks in WebDSL actions, the core language provides no mechanism for placing error messages at the related components in pages. This requires a change in the request processing lifecycle of the core language. Furthermore, many validation constraints are expressible *declaratively*, associated to the data model.

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.

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. Note that validation rules are only used for input checks that require notification to the user. Checks and filtering to prevent post-data tampering and javascript injection are taken care of by the input and output components of WebDSL, such filtering should not have to be expressed in an application's validation rules. For example, in the case of `WikiText`, there is only a validation for the (very large) max length allowed, however, `output(WikiText)` filters HTML elements based on a restrictive whitelist after processing Markdown. Furthermore, checks and messages for built-in type validation can be customized in an application. For example, Figure 2 shows how the `Int` type format check and message can be controlled in an application.

```
type Int {
  checkFormat = /-?\d+/.match(this)
  messageFormat = this + " is not a valid number"
}
```

Fig. 2. Built-in validation customization.

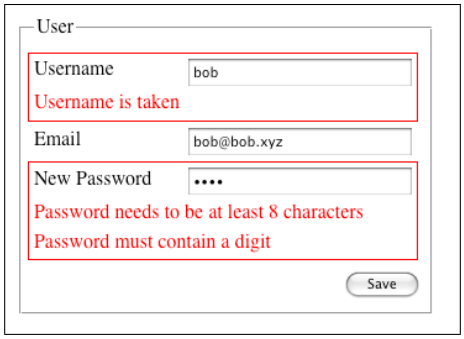
The `editUser` page in Figure 1 consists of a form with labeled inputs for the `User` properties. The `save` action persists the changes to the database, provided that all validation checks succeed. Since well-formedness validation checks are automatically applied to properties, the `email` property 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 [26] data invariants can be defined in a 'validate' method of the active record class, which

```
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")
}
```

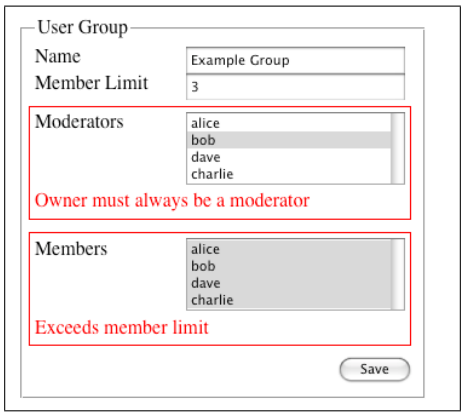



```
define page editUser(u:User) {
  form {
    group("User") {
      label("Username"){ input(u.username) }
      label("Email"){ input(u.email) }
      label("New Password") {
        input(u.password)
      }
      action("Save", save())
    }
  }
  action save() {
    return user(u);
  }
}
```

Fig. 3. Data invariants for `User` entity validation.

```
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")
  members(validate(membersWithinLimit(),"Exceeds member limit"))
  predicate membersWithinLimit() { members.length <= memberLimit }
}
```




```
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. 4. Data invariants for `UserGroup` entity validation.

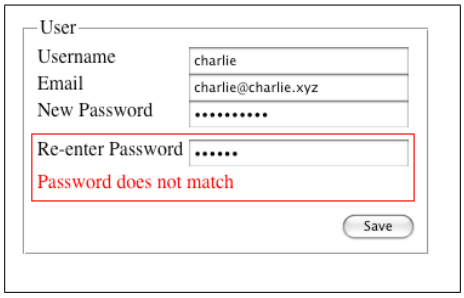
then gets called by the framework when validation is required. Multiple checks in a validation method tangle validation for different properties. The Seam [18] framework supports the specification of data invariants declaratively through annotations. However, these annotations consist of a limited number 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. In the examples in this paper, error messages are placed inline for conciseness. In general, error messages can also be placed inside a function or even be stored in the database (as entity property), depending on the requirements for configuration and internationalization of the application.

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.

Figure 3 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.

Figure 4 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. This is illustrated by the member limit check, which is added to the `members` properties. Note that although the check is only




```
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. 5. Form validation with input assertions.

attached to the `members` property, a form and action that changes only `memberLimit` would still check invariants for the whole entity before committing changes. Duplication in checks can be avoided by putting checks in predicates or functions. A predicate in WebDSL is a function that returns a `Bool`, its body a single expression that produces the result value.

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 [4] 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 text, 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. Since storing inputs happens before these validation rules are checked (see Section 4), 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 Figure 5 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.

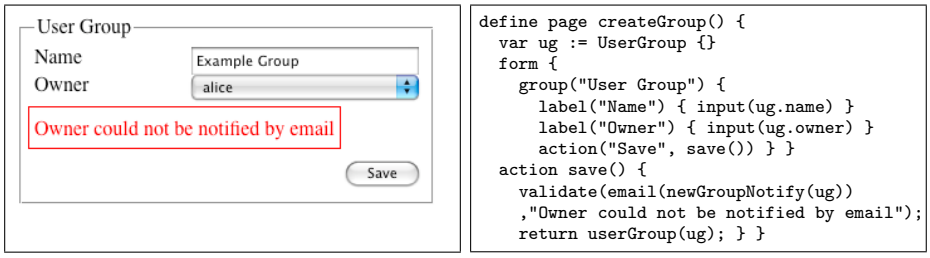


Fig. 6. Action assertion for `UserGroup` creation.

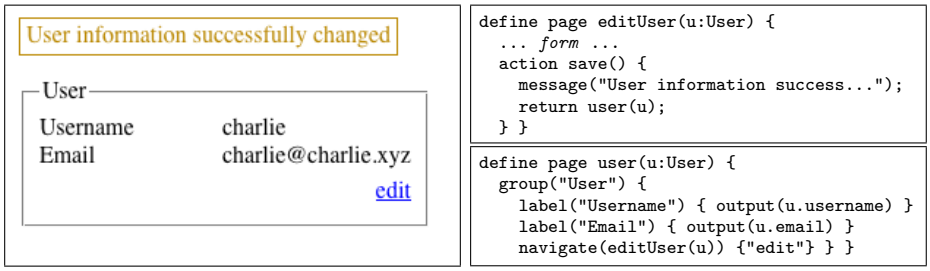


Fig. 7. Success message.

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. If such an assertion fails, the action processing needs to be 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 [26] 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 `validate` syntax. 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.

Figure 6 provides an example of an 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 Figure 7. 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.

4 Validation Mechanics

We have found that data validation concerns are often delegated to a web application framework in web engineering research, e.g. Struts for WebML [8] and Ruby on Rails for HyperDe [24], assuming that the framework will provide sufficient support. Examining such web frameworks (in our case Seam [18], ASP.NET [22], and Ruby on Rails [26]) reveals limitations in the data validation solutions. For example, validation is sometimes only supported in the view layer, making it hard to access the database and the data that is already loaded. Another issue can be that validation hooks are only provided for single input components, ignoring validation between multiple inputs. Value well-formedness checks might have to be repeated ad nauseam. The next problem is getting the messages in the right place. Usually, this is organized well for the directly supported validation features. However, for more advanced cases like multi-input and database access it becomes cumbersome, e.g. create an entire class for a single validation check, and error-prone, e.g. hook into the object-relational mapping (ORM) session. In the worst case, messages will need to be passed explicitly and message components need to be included in page definitions that can retrieve and display those passed messages. In the data validation solution we propose, these problems are addressed. Validation checks have access to all inputs and the ORM context. This section describes the integration of data validation into the request processing lifecycle, which is needed for such flexibility. A request is processed in five phases: convert request parameters, update model values, validate forms, handle actions, and render page or redirect. A database transaction is started for each request, this transaction is rolled back when there are validation errors, otherwise it is committed. The phases are illustrated in Figure 8 and will be covered separately.

4.1 Convert Request Parameters

Users interact with web applications through the browser. This process consists of request and response strings being exchanged between the web server and the browser. A form, such as displayed in the example in Figure 1, is defined by a response string,

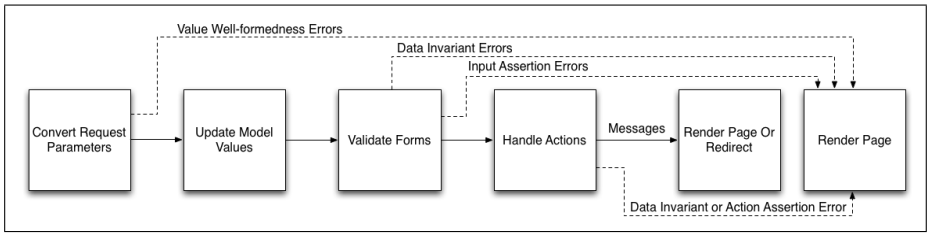


Fig. 8. WebDSL request processing lifecycle.

which is interpreted by the browser to produce components that allow user interaction. A user can fill in data in a text field, and press the submit button. The browser first collects the data from the form input fields, and constructs a request string to send to the web server, which receives the request string and parses it. Values from input fields can be accessed separately but are represented as strings. A web application bears the responsibility of converting these strings to actual types to be used in further processing of the request. Since such conversions are common in web applications, they are typically directly supported in frameworks. The supported types are the native types available in the language used to build the framework. WebDSL extends the usual set of primitive types with domain-specific types such as `Email` and `Secret`. Conversions from and to strings for these types are supported in the language itself.

Request parameter conversion is not possible if the incoming value is not well-formed. For example, a value of `"3f"` cannot be converted to an integer. Since a failed conversion invalidates any input it is not necessary to update the model before re-rendering the page with error messages. The `Value Well-formedness Errors` arrow indicates this situation. In a page render resulting from validation errors, input components that were submitted restore the submitted value instead of the original value. This allows a user to fix the entered data.

Each input in a page definition includes a template that renders the corresponding conversion error. This template wraps around inputs and labeled inputs which allows explicit indication of the erroneous input element. Validation message templates can be overridden in the application model to support flexibility in layout and style. A template definition in WebDSL can contain the same elements as a page definition. An error rendering template takes as argument the list of messages (a `List<String>`). The messages can be shown in several ways, e.g. as items in a bullet list, or on new lines below the input (as illustrated in Figure 1). Error templates and customization is discussed in more detail in Section 5.

4.2 Update Model Values

In the first phase, parameters are decoded from strings. In the 'Update Model Values' phase, these parameters are automatically inserted in data model entities. WebDSL supports such *data binding* through `input` elements. For example, the `input(u.email)` element declares that an input field should be displayed with the current contents of the email property of variable `u` of type `User`. Furthermore, when a user submits the containing form with a new value in the email field, the new value will be assigned to

`u.email`. An action finalizing this operation just needs to save the variable `u` in order to persist the new email address.

4.3 Validate Forms

The changes made through data binding have to be validated, this is performed after data binding for the whole form is completed. When an entity property is being validated, each validation rule defined on that property is checked, possibly producing multiple error messages. This can be observed in the password property example in Figure 3. Besides entity validations, there can also be validation rules in pages which need to be enforced; e.g., the repeat password check in Figure 5. The 'Validate Forms' phase traverses the form that is submitted and checks any validation it encounters. When at least one validation fails during this phase, further processing is disabled and errors are displayed, indicated by the `Data Invariant Errors` and `Input Assertion Errors` arrows.

Messages originating from entity validation are rendered in the same way as the conversion error messages. The same template is used, but now the argument contains validation error messages originating from the data model. The screenshot in Figure 3 shows this type of message. The validation messages for page validations are displayed at the location of the check in the form. If the validation is expressed in the context of an input element, then the input will display the error as if originating from a data model invariant (see Figure 5). If the validation check is not in the context of an input, it is rendered in-place.

4.4 Handle Actions

When all validation checks in previous phases have succeeded, the selected action is executed. During the execution of an action there can be action assertions that validate the data in the current execution state of the action. Moreover, data invariants are still checked during this phase and can produce validation errors as well. If any validation check fails, the entire action is cancelled. This means all the changes to the data model are reverted and rendering is initiated (`Data Invariant or Action Assertion Error` arrow). Only one error can be produced at a time since action processing will not continue when a validation fails.

Error messages produced during the 'Handle Actions' phase are placed at the executed action button (Figure 6). In this case, the error template wraps around the action invoking button instead of an input or label.

4.5 Render Page or Redirect

Validation messages produced in the previous phases result in a re-render of the same page with error messages inserted. If all validations succeed, the action results in a redirect to the same or a different page, possibly sending messages along which describe successful execution of the action (`Messages` arrow).

Success messages to other pages are handled by adding an implicit argument to each page containing the list of messages. When an action is finished executing and initiates

a redirect or re-render, the messages are passed along. A different template is used for success messages, since these are likely to use different colors than the templates for errors. By default this template is added at the top of each page, but the position can be customized as well by adding an explicit `messages` component in a page.

4.6 Ajax

WebDSL also supports asynchronous page updates. Instead of a full page refresh as response to an action, an action may selectively replace elements of the page with new elements. For example, a content-heavy page can have a small sidebar containing a placeholder with a login button. The action connected to this button can replace the placeholder with a login form. When a validation rule fails in such a form, the entire placeholder is replaced again, it will display the login form with validation errors just as with the validation in earlier examples in this paper. Automatically deriving more fine-grained validation checks while entering data in the form is not addressed in this paper.

5 Compilation by Normalization

The implementation of data validation is based on compilation by normalization [17]. Data validation language elements are transformed to more generic lower-level constructs in the WebDSL language. These lower-level constructs are reusable for other language features. Moreover, they can easily be tested in isolation, resulting in a robust implementation. Since these lower-level constructs are still at a higher abstraction level than the code that is ultimately generated, they greatly simplify the translation for the data validation abstraction. This section describes the implementation of validation checks (Section 5.1), the visualization of generated error messages (Section 5.2), and the mechanism for generic messages (Section 5.3).

5.1 Validation Checks

This section provides a more detailed explanation of the implementation of validation checks. The type of checks introduced in Section 3 are discussed separately.

Value Well-Formedness To implement or customize value well-formedness checks, a hook is needed to add validation to value types. The WebDSL definition `type x { typeelem* }` allows adding validation to (built-in) value types. For instance, the definition in Figure 9 checks that a request parameter of type `Int` can be successfully created from the incoming `String` (`input` is a special variable in this context that refers to the incoming request parameter `String`).

Figure 10 describes the semantics of these type validations using a transformation. The WebDSL code that does the parameter conversion is shown on the left, converting `input` (type `String` is added for clarity, it can be omitted because WebDSL supports local type inference in variable initializations) to `inputX`, where X is a value type. A

```
type Int {
  validate(input.parseInt() != null, "Not a valid Int")
}
```

Fig. 9. Value well-formedness definition.

value well-formedness definition on the type (similar to what is shown in Figure 9) adds validation to type X. This results in a transformation of the conversion code to include the validation checks, before performing data binding (`ref := inputX;`). Validations on built-in value types are part of the standard library, and custom value types can also take advantage of automatic well-formedness checks for inputs using the `type` definition.

```
var input : String :=
            requestParams().get(elemId());
if(input != null) {
  var inputX := input.parseX();
  if(inputX != null) {
    ... ref := inputX; ...
  }
}
```

```
type X { ... validate(check,message) ... }
```

⇒

```
var input : String :=
            requestParams().get(elemId());
if(input != null){
  validate(check,message);
  var inputX := input.parseX();
  if(inputX != null) {
    ... ref := inputX; ...
  }
}
```

Fig. 10. Well-formedness checks transformation.

The `validate` function that is called is shown in Figure 11. The function checks the passed condition and throws an exception with the message, contained in a `ValidationException` entity. Where this exception is caught and how errors are displayed is described later in this section. The `cancel()` function is a built-in function for cancelling changes made during request handling. More specifically, after finishing the current phase, any changes due to data binding or actions are reverted, and any phase between the current and render phase is skipped.

```
entity ValidationException { message :: String }
```

```
function validate(check : Bool, message : String) {
  if(!check) {
    cancel();
    throw ValidationException{ message := message };
  }
}
```

Fig. 11. Validate utility definitions.

Data Invariants Figure 12 illustrates the implementation of data invariant checks using transformations. These validations on entity properties are checked at two places in the lifecycle. Firstly, they are checked in the 'Validate Forms' phase. The transformation on `input(e.x)` adds the validation to each input of entity property `x`. Secondly, they are checked for every entity instance that is new or has changes at the end of an action. This is handled by adding the validation checks to an entity function hook, `beforeCommit()`, created specifically for this purpose. The Object-Relational Map-

ping (ORM) library is instructed to call this function for each 'dirty' object, before committing the changes.

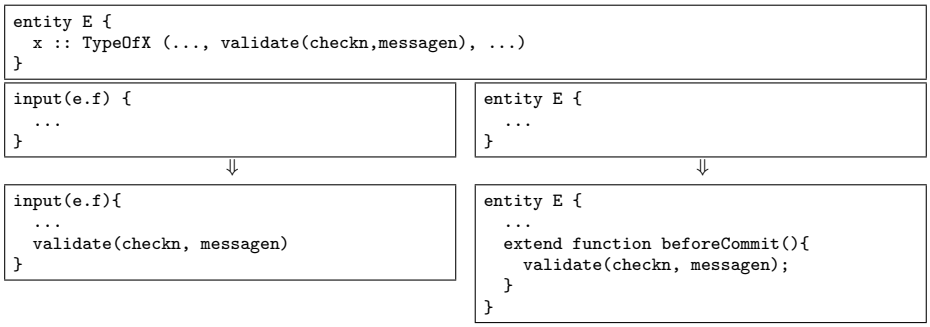


Fig. 12. Property validation transformation.

When multiple failed validation rules have to produce messages, e.g. an input with multiple nested validations, they are combined into a single call. This is illustrated in Figure 13. The call to `validateMultiple` will create multiple messages (`[elem1, ..., elemn]` is a list construction expression in WebDSL), the utility function and entities used for checking multiple validation rules at once are shown in Figure 14. The `Validation` entity stores a validation result and its message. `ValidationException-Multiple` can be used to throw an exception with multiple messages. In the `validate-Multiple` function all checks are first performed, then if any check failed, the messages of all failed checks are put into the thrown exception.

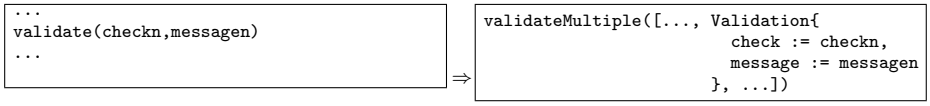


Fig. 13. Combine validations transformation.

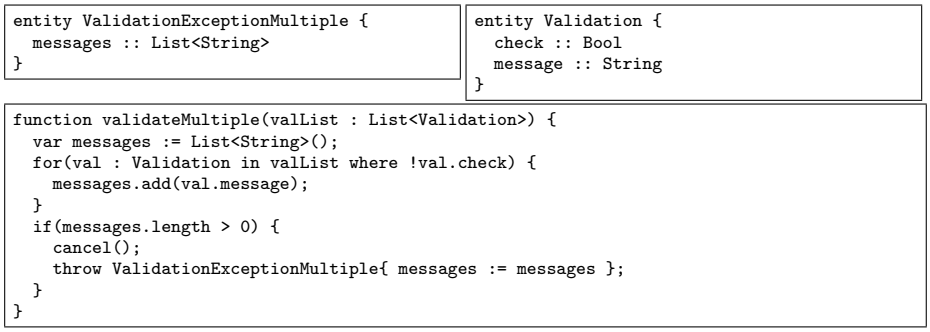


Fig. 14. Multiple validate utility definitions.

Input Assertions Input assertions (validation checks in a form) call the utility template definition in Figure 15. The `validate` template consists of an `executeValidate` block, which contains action code to be executed during the 'Validate Forms' phase of request processing (see Section 4). In this block the `validate` function is called, which was shown in Figure 11. A similar template is defined for `validateMultiple`.

```
define validate(check:Bool, message:String) {
  executeValidate { validate(check, message); }
}
define validateMultiple(list : List<Validation>) {
  executeValidate { validateMultiple(list); }
}
```

Fig. 15. Page validation utility template.

Action Assertions Action checks call the utility `validate` function (Figure 11) directly, the cancel function and exception mechanics are supported in this context as well. By reusing the `validate` function for each type of data validation the consistency of handling validation checks is enforced.

5.2 Reporting Validation Errors

Validation checks can produce error messages which have to be presented to the user. Applications can have various requirements for displaying such error messages, e.g. a specific style and the location of the error. The validation abstraction needs to cope with these requirements and should be flexible enough to easily create customizations.

In order to visualize validation errors, inputs, labels, and action buttons/links are wrapped in a `validationContext`. This wrapping is illustrated in Figure 16. The `input( x )` is wrapped in a `validationContext` element, which includes a reference to the error template that needs to be used for displaying the error, in this case `errorTemplate` (shown in Figure 17). This is further desugared to a template call with two explicit template arguments, the error template and the contents. This wrapping is done automatically, but a `validationContext` can also be explicitly added in an application in order to customize the error template that is used in that specific context.

```
input(x)
```
⇒
```
validationContext errorTemplate {
  input(x)
}
```
⇒
```
validationContext with {
  error(m : List<String>) {
    errorTemplate(m)
  }
  content() {
    input(x)
  }
}
```

Fig. 16. Validation feedback transformation.

The `validationContext` template is shown in Figure 18. This template takes care of catching the thrown exceptions and calling the error template. `try-catch-finally`

```
define allow-override errorTemplate(messages : List<String>) {
  block[class: error] {
    for(m: String in messages) {
      output(m)
    }
    elements
  }
}
```

Fig. 17. Error template.

in the context of a template has the following semantics: the `try` and `finally` body are handled in the pre-render phases, if no exception is caught they are handled in the render phase as well. If an exception is thrown in one of the earlier phases, the entire construct will be ignored from that point and in subsequent pre-render phases, and the exception is reproduced at the same point during the render phase. In the render phase, the matching `catch` and `finally` blocks are executed. In the `catch` block the error template is called with the wrapped content. The template call `elements` produces the elements that are children elements in the template call (in this case `content`). The default `errorTemplate` template can be redefined in an application, this behavior is enabled by the `allow-override` modifier. Note that the error template can also be customized for a specific case, using `validationContext` directly, or in a whole template call context, shown later in this section. The request-scoped variable `validationContextCount` keeps track of nesting. Request-scoped variable variables are created at the beginning of the request and cleaned up at the end of the request, and they are accessible from anywhere in the application. The `executeRender` block contains action code that is executed during the render phase of the request processing lifecycle (Section 4). `validationContextCount` counts the number of `validationContext` templates that have been entered recursively. The validation message will be shown at the outer validation context, when `validationContextCount == 1`. This will make sure that inputs with labels take the label as part of the erroneous content, and that an explicit `validationContext` template call takes precedence over an automatically generated one. It also becomes possible to place the error higher in the page element tree by adding an explicit `validationContext` element, e.g. to show errors above or below the form that encloses the input.

```
request var validationContextCount := 0

define validationContext() requires error(m: List<String>), content() {
  executeRender { validationContextCount++; }
  try { content }
  catch(ve : ValidationException) {
    if(validationContextCount == 1 ) { error([ve.message]) { content } }
    else { throw ve; }
  }
  catch(vem : ValidationExceptionMultiple) {
    if(validationContextCount == 1 ) { error(vem.messages) { content } }
    else { throw vem; }
  }
  finally { executeRender { validationContextCount--; } }
}
```

Fig. 18. Validation context template.

Customization of the error template can be done in an entire template context to avoid repetitively calling the custom template. Figure 19 shows a custom error template, `errorTemplateCustom`, and a call to `useValidationError`, which declares a new default error template for the content. The implementation of this context is done by generating a new uniquely named template and reusing the dynamically scoped local template redefinitions of WebDSL to set `errorTemplate` for the content. Each call to `errorTemplate` in the context of `CustomValidationContext1` calls the redefined version of `errorTemplate`, which uses the custom error template to show messages (`errorTemplateCustom`). The call to `useValidationError` is transformed to a call to the generated template (`customValidationContext` in the example). These error contexts can be used to allow a library with layout to enforce a corresponding error layout, without requiring the user of the library to set the error template.

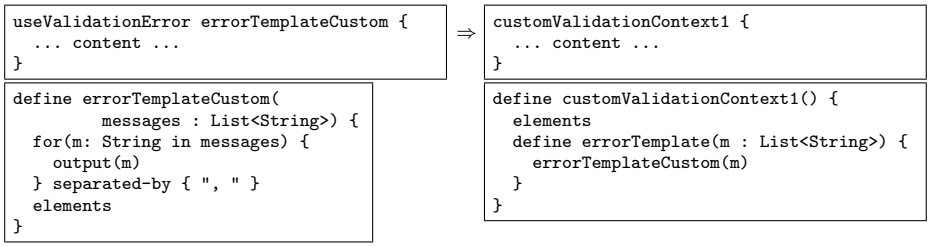


Fig. 19. Custom error context transformation.

5.3 Messages

Besides error messages we have also described generic messages. These messages are stored in a session entity (Figure 20). The `messageStorage` session entity has one property, a list of String messages. Session entities in WebDSL behave like singletons (one per browser session) that are automatically stored in the server session, and retrieved upon the next request. Since an action always results in a redirect (to avoid accidental repeated 'post' requests), the messages can be immediately retrieved and removed from the session when rendering the next page. This approach avoids polluting the URL with messages when redirecting. The `message` function is added for conveniently adding messages to the session entity.

The display of these messages is illustrated in Figure 20. Since messages are stored in the session entity `messageStorage`, they need to be retrieved and removed so they will not be shown again. They are taken from the session and put into the request-scoped variable `incomingMessages`. A call to the `messages` template is transformed to a call to the default `messageTemplate` which retrieves the messages through the function `getMessages()`. This function marks the messages as being shown, so they will not show up at the top of the page, and returns the list of messages. Since `getMessages()` can also be called directly the template used for showing messages can be customized. Furthermore, an application can override `messageTemplate`.

```
session messageStorage {
  messages :: List<String>
}
```

```
function message(m : String) {
  messageStorage.messages.add(m);
}
```

```
extend session messageStorage {
  function retrieve() : List<String> {
    var temp := List<String>();
    temp.addAll(messages);
    messages.clear();
    return temp;
  }
}
request var incomingMessages :=
                    messageStorage.retrieve()
function getMessages() : List<String> {
  messagesShown := true;
  return incomingMessages;
}
```

```
request var messagesShown := false
function getMessages() : List<String> {
  messagesShown := true;
  return incomingMessages;
}
define allow-override
 messageTemplate(messages : List<String>) {
  block[class := success] {
    for(m: String in messages) {
      output(m)
    }
    elements
  }
}
```

```
messages
```

⇒

```
messageTemplate(getMessages())
```

Fig. 20. Storing and receiving messages.

6 Evaluation

In order to assess the data validation abstraction presented in this paper, we have performed two case studies in which we analyzed the application of data validation in existing applications written in WebDSL.

Goal and Research Questions Our goal in these case studies is to analyze the coverage of the data validation abstraction, to assess the conciseness of data validation definitions, and to review the flexibility in presenting messages to the user. Our research questions are the following:

RQ1 What is the coverage of the validation abstraction with respect to the data validation requirements of the application?

RQ2 How concise are the data validation definitions?

RQ3 Is the default way of presenting errors acceptable, and are the customization options sufficient?

6.1 Case Study 1: Webdslorg

The Webdslorg application is used for the homepage of WebDSL[1]. It is a wiki application that allows developers and invited users to create a manual online, and to show news related to WebDSL. The application contains wiki page versioning, page composition, user management, and access control management. This application consists of around 1800 lines of WebDSL code. These lines constitute 10 data model entities, 27 pages, 70 templates, and 30 actions. Webdslorg is open source[2].

[1] http://webdsl.org

[2] https://svn.strategoxt.org/repos/WebDSL/webdslorg/trunk/

The main data validation requirements in this application occur in the create and edit user forms, the login form, and the create and edit page forms. To address RQ1 we first look at the usage of data validation in the application. The results are shown in Figure 21. There are 30 input components in the application, each input validates well-formedness implicitly. 8 data invariants are used, checking e.g. uniqueness and existence of properties ('not null' checks). The other types of data validation do not occur frequently, there are 3 input assertions and 2 action assertions.

Implicit Checks	Explicit Checks		
Value Well-Formedness	Data Invariants	Input Assertions	Action Assertions
30	8	3	2

Fig. 21. webdsl.org data validation

In the wikipage editing template there is a check to verify that the version being edited is still the latest version of that wikipage, and if not, the user is warned and the newer template is shown before finalizing the edit. This kind of warning behavior is not in the data validation abstraction and a workaround is created here using generic messages and action assertions. For the other data validation requirements the data validation abstraction is sufficient.

For RQ2 we look at the expressions used in data validation checks. In 9 out of 13 cases the check is a simple comparison, in the other 4 cases a function is called which decides the validation outcome. These functions are e.g. querying the database for uniqueness. Also the warning behavior mentioned requires a function to encode a workaround.

For RQ3 we determine how much error message customization is applied in Webdslorg. In this application there is a global override of the error and success messages. Since all the forms have a consistent layout, the error messages can also be displayed in one consistent way.

6.2 Case Study 2: Researchr

Researchr[3] is a tool for indexing, managing, and sharing bibliographic information of scientific publications. An important feature of Researchr is the identification of authors, editors, and advisors. Other features include:

- Author profiles with publications, affiliations, reviews
- Theses with advisors
- Indexing of proceedings and journals

Basic bibliography information can be further enriched by:

- Tagging: categorize publications with keywords
- Reviewing: private or public reviews of publications
- Bibliographies: collect publications on a subject

[3] http://researchr.org

– Usergroups: share bibliographies

Researchr consists of around 15000 lines of WebDSL code, which constitute 50 data model entities, 100 pages, 600 templates, and 190 actions.

To address RQ1 we look at the application of data validation in Researchr. The results are shown in Figure 22. Researchr contains 160 input fields which all get implicit value well-formedness checks. There are 20 data invariants, which enforce uniqueness of identifying properties and some multi-property constraints. 5 input assertions cover cases similar to the 'repeat password' check in Figure 5. Finally, there are 14 action assertions. Several complex operations in Researchr are related to conversion of imported data from DBLP[4] and from bibtex entries, these benefit from having action assertions to notify the user of problems occuring in the import and to prevent importing incorrect data.

Implicit Checks	Explicit Checks		
Value Well-Formedness	Data Invariants	Input Assertions	Action Assertions
160	20	5	14

Fig. 22. researchr.org data validation

Similar to what we found in case study 1 for RQ2, the checks in Researchr are mostly simple comparisons and a few more complex checks encapsulated in function definitions.

Researchr has several layouts for forms, some are placed in the main part of the page, others are small and located in a sidebar. This requires customization of error messages specific to these contexts (RQ3). The error messages for normal forms are customized by redefining the global template for error messages. The sidebar is nested in a `useValidationError` call (see Figure 19), in order to specify a different error template in that entire context.

7 Discussion

This section contains a discussion of related work, covering web modeling tools (Section 7.1), web application frameworks (Section 7.2), and form replacements (Section 7.3). This is followed by a discussion of future work (Section 7.4).

7.1 Web Modeling Tools

Several model-driven methodologies for creating web applications have been proposed in recent years, including OOHDM [28], SHDM [21], WebML [8], UWE [19], OOWS [25], and Hera [30]. 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.

[4] http://dblp.uni-trier.de

The Hera Presentation Generator [11] allows modeling forms to support editing data in the session. The persisted domain data of the application cannot be changed. Hera-S [29] 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. [20] 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 [24] 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 submitted data) and does not indicate how validation of that input can be performed. WebRatio [5] 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 [6] 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.

Book et al. [3] describe a formal model for user input evaluation and interface responses. Rules define validity, visibility, and availability of user interface widgets. Technical validation rules in their model correspond to value well-formedness rules, automatic checks based on the type of the input. Data model validation is similar to input assertions in this paper, validations related to specific inputs in forms. Only variables of primitive types are included in the model, so there is no clear connection to validation on data model entities. Their model does include intermediate validation of data while entering the form, e.g. validating when an input loses focus. Validation rules are constructed in a visual expression editor and execution is interpreted at run-time. The implementation of this model is part of the Cepheus framework.

From our study of the web modeling literature we conclude that data validation is not a large research theme in this area. The languages and tools typically only address simple form validation, without accessing the persisted data model.

7.2 Web Application Frameworks

JavaServer Faces (JSF) [7] provides abstractions for creating user interfaces in Java web applications. The core of the request processing lifecycle in JSF (without events and short-circuiting) is illustrated in Figure 23. The lifecycle is critically different at one point compared to the WebDSL lifecycle shown in Section 4, the 'Process Validations' phase occurs before updating of model values. Since the connection to the model has not been made at the point of validation, these validation rules only support checks based on the input value. Expressing validation rules covering multiple input fields is cumbersome as it requires that each relevant input has been processed (this validation needs to be expressed below the inputs in the page component structure) and requires accessing the components with identifiers.

The Seam framework [18] combines JSF with the Java Persistence API (JPA) [10]/Hibernate [2] for data model persistence. Because the data model is not updated before

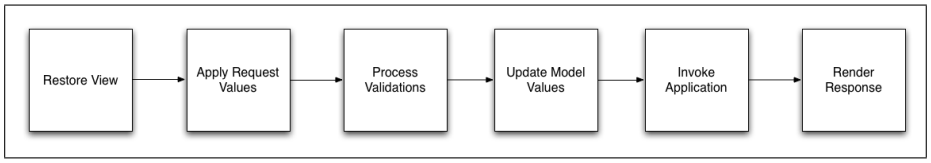


Fig. 23. JSF request processing lifecycle.

processing validations, validation constraints cannot be expressed on the data model (e.g. the member limit validation in Figure 4). It is necessary to encode these kinds of constraints as part of the 'Invoke Application' phase (Figure 23), in which the 'business logic' of the application is executed. Unfortunately, in the Seam framework the 'Update Model Values' phase also results in database updates. The database transaction has to be explicitly aborted in case the validation check fails at the 'Invoke Application' phase, and the old values have to be restored before rendering the response. Furthermore, placing error messages at this point in the lifecycle requires referring to explicitly named JSF page components. Keeping such names in sync is hard to maintain.

ASP.NET [22] provides form input validation controls which are executed at the client and the server. There are several built-in validation controls, such as required field and regular expression. Moreover, developers can create custom validation controls to provide code for validating the input (client-side is optional). These form controls are similar to the validation support in JSF, also not taking data models into account. Visual Studio provides a visual designer to create data models and generate classes with database mappings to be used with the Language-Integrated Query (LINQ) [23] feature in .NET. These generated classes allow validation to be expressed on the data model through predefined method names, such as `OnFieldChanging` and `OnValidate`, which can be implemented in partial classes. An issue is that validation is not defined in the same place as the entity, and has to be synchronized with the generated classes. The implemented function has to throw a specific exception, which needs to be caught in the page code and requires custom code to display the error message. Consequently, form validation and data model validation are supported by inconsistent mechanisms in this framework. Both mechanisms have limited access to the rest of the application.

In Ruby on Rails [26] validation can be specified in Active Record objects, which constitute data model entities. The implementation consists of defining functions with specific names, such as `validate` and `validate_on_create`, which are the hooks for validation. These functions get called when an object is saved. Defining such functions is more verbose than the validation rules in WebDSL and there is no static verification of the validation check. Built-in validation checks can be declared more concisely, leveraging the fertile ground of Ruby for embedded DSLs [9]. Nevertheless, this introduces two notations for the same concept. Value well-formedness checks, such as checking whether an input is a number, are typically built-in checks. These are not added automatically. Displaying errors for data model validation checks also has to be done manually by adding error message components to pages, which refer to an entity property or the entity as a whole. Form validation related to values not in the data model can be validated in the action handling code, and the message can be added by explicitly

placing it in the page template and connecting it by name. Generic errors and messages are handled similarly.

What can be observed in the various existing web framework solutions for validation is that the different types of validation are addressed with separate mechanisms. Furthermore, there is no automatic way of placing messages in pages for each of the validation types, often one type is preferred over the others.

7.3 Form Replacements

The XForms [4] standard is a successor to HTML forms. XForms separates the classical form into model, instance data, and user interface to allow better reuse. It provides a rich standard for interactive forms, which improves device-independence and reduces the need for scripting. Values are strongly typed, which allows automatic well-formedness checks. Furthermore, XForms supports validation rules to constrain the value space of data values collected by the model. While the standardization of an improved form component is a positive development, the features proposed are already available as Javascript libraries. It does not solve the integration problems with user interface and data validation, since only part of the validation concerns can be addressed in the environment of the form.

7.4 Future Work

In this paper we have not considered intermediate validation checks. Validation could be requested from the server for each input component separately, while the user is entering data into a form. In the examples shown, validation is performed when the save button is pressed. Providing feedback as early as possible can improve the user experience.

Checks are currently always performed on the server. Implementing checks client-side could be considered an optimization and does not obviate the need for server-side checks, since submitted data can be tampered with. Moreover, checks that require access to the database can only be performed at the server.

Besides strict validation errors which deny the operation, there are also softer constraints involved in developing a web application. The validation mechanism could be extended to include warnings and confirmations, e.g. require the user to click again to finalize an action. Furthermore, information messages could be added to assist the user in repairing typical mistakes, e.g. when the user repeats an error a few times, show extra information to help the user fill in the form.

Some data invariants can be translated to database schema constraints. Adding these to the underlying database schema will improve robustness of the application. For example, it will protect the programmer from certain errors when migrating old data to a new version of the application.

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 [16] and *repair* data so that it complies with the constraints or to suggest such repairs to the user.

In this paper we focus on how and where to express data validation. The checks consist of arbitrary expressions such as simple comparisons, collection membership tests, or function calls. Since most inputs in web application forms are strings, expressivity could be increased by incorporating a domain-specific language for string constraints. Scaffidi et al. [27] demonstrate that parsing technology can provide rich string input validation and feedback.

8 Conclusion

The domain-specific language engineering challenge for the web application domain [31] 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 truly 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. Object Constraint Language, OMG Available Specification, Version 2.0, 2006.
2. C. Bauer and G. King, editors. *Java Persistence with Hibernate*. Manning Publ. Co., 2006.
3. M. Book, T. Brückmann, V. Gruhn, and M. Hülder. Specification and control of interface responses to user input in rich internet applications. In *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 321–331, Washington, DC, USA, 2009. IEEE Computer Society.
4. J. M. Boyer, editor. *XForms 1.0 (Third Edition)*. W3C Recommendation, 2007.
5. M. Brambilla, S. Comai, P. Fraternali, and M. Matera. Designing web applications with WebML and WebRatio. *Web Engineering: Modelling and Implementing Web Applications*, pages 221–260, 2007.
6. D. Brown, C. Davis, and S. Stanlick, editors. *Struts 2 in Action*. Manning Publ. Co., 2008.
7. E. Burns and R. Kitain, editors. *JavaServer Faces Specification. Version 1.2*. Sun, 2006.
8. S. Ceri, P. Fraternali, and A. Bongio. Web Modeling Language (WebML): a modeling language for designing Web sites. *Computer Networks*, 33(1-6):137–157, 2000.
9. J. Cuadrado and J. Molina. Building Domain-Specific Languages for Model-Driven Development. *IEEE Software*, pages 48–55, 2007.
10. L. DeMichiel and M. Keith, editors. *JSR 220: Enterprise JavaBeans, Version 3.0. Java Persistence API*. Sun Microsystems, 2006.
11. F. Frasincar, G. Houben, and P. Barna. HPG: the Hera Presentation Generator. *Journal of Web Engineering*, 5(2):175, 2006.

12. D. M. Groenewegen, Z. Hemel, L. C. L. Kats, and E. Visser. When frameworks let you down. platform-imposed constraints on the design and evolution of domain-specific languages. In J. Gray et al., editors, *Domain Specific Modelling (DSM'08)*, pages 64–66, October 2008.
13. 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, *International Conference on Web Engineering (ICWE'08)*, pages 175–188, July 2008.
14. D. M. Groenewegen and E. Visser. Integration of Data Validation and User Interface Concerns in a DSL for Web Applications. In M. van den Brand and J. Gray, editors, *Software Language Engineering, Second International Conference, SLE 2009, Denver, USA, October, 2009. Revised Selected Short Papers*, Lecture Notes in Computer Science. Springer, 2009.
15. Z. Hemel, R. Verhaaf, and E. Visser. WebWorkFlow: An object-oriented workflow modeling language for web applications. In K. Czarnecki et al., editors, *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MODELS 2008)*, volume 5301 of *LNCS*, pages 113–127. Springer, September 2008.
16. J. Järvi, M. Marcus, S. Parent, J. Freeman, and J. N. Smith. Property models: from incidental algorithms to reusable components. In *GPCE*, pages 89–98, 2008.
17. L. C. L. Kats, M. Bravenboer, and E. Visser. Mixing source and bytecode. A case for compilation by normalization. In G. Kiczales, editor, *Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2008)*, pages 91–108. ACM, October 2008.
18. S. Kittoli, editor. *Seam - Contextual Components. A Framework for Enterprise Java*. Red Hat Middleware, LLC, 2008.
19. N. Koch, A. Kraus, and R. Hennicker. The authoring process of the UML-based web engineering approach. In *Web-Oriented Software Technology*, 2001.
20. A. Kraus, A. Knapp, and N. Koch. Model-driven generation of web applications in UWE. *Model-Driven Web Engineering (MDWE 2007), Como, Italy (July 2007)*.
21. F. Lima and D. Schwabe. Application modeling for the semantic web. In *Latin American Web Congress (LA-WEB'03)*, page 93, Washington, DC, USA, 2003. IEEE Computer Society.
22. M. MacDonald and M. Szpuszta. *Pro ASP. NET 3.5 in C# 2008*. Apress, 2007.
23. E. Meijer, B. Beckman, and G. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *Management of Data*, pages 706–706, 2006.
24. D. Nunes and D. Schwabe. Rapid prototyping of web applications combining domain specific languages and model driven design. In *International Conference on Web Engineering (ICWE'06)*, pages 153–160, 2006.
25. O. Pastor, J. Fons, and V. Pelechano. OOWS: A method to develop web applications from web-oriented conceptual models. In *Web Oriented Software Technology (IWWOST'03)*, pages 65–70, 2003.
26. S. Ruby, D. Thomas, and D. Heinemeier Hansson. *Agile Web Development with Rails, Third Edition*. Pragmatic Programmers, 2009.
27. C. Scaffidi, B. A. Myers, and M. Shaw. Topes: reusable abstractions for validating data. In *ICSE'08*, pages 1–10, 2008.
28. D. Schwabe, G. Rossi, and S. Barbosa. Systematic hypermedia application design with OOHDM. In *Proceedings of the the seventh ACM conference on Hypertext*, pages 116–128. ACM New York, NY, USA, 1996.
29. K. van der Sluijs, G. Houben, J. Broekstra, and S. Casteleyn. Hera-S: web design using sesame. In *International Conference on Web Engineering (ICWE'06)*, pages 337–344, 2006.
30. R. Vdovjak, F. Frasincar, G. Houben, and P. Barna. Engineering semantic web information systems in hera. *Journal of Web Engineering*, 2:3–26, 2003.
31. E. Visser. WebDSL: A case study in domain-specific language engineering. In R. Lämmel, J. Visser, and J. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE'07)*, volume 5235 of *LNCS*, pages 291–373. Springer, October 2008.
32. E. Visser et al. WebDSL. http://webdsl.org, 2007–2009.

TUD-SERG-2010-033
ISSN 1872-5392