Using Actions Charts for Reactive Web Application Modelling

Nina Geiger, Tobias George, Marcel Hahn, Ruben Jubeh, Albert Zündorf

University of Kassel, Software Engineering ,
Department of Computer Science and Electrical Engineering,
Wilhelmshöher Allee 73,
34121 Kassel, Germany
`[nina | tge | hahn | ruben | zuendorf]@cs.uni-kassel.de`
`http://seblog.cs.uni-kassel.de/`

Abstract. Building a rich internet application (RIA) requires the programming of various callbacks and listeners. AJAX like server requests require callback handler objects that react to the asynchronous server response. Active GUI elements like buttons or menu entries require action handlers. Using a timer queue requires appropriate event handlers, too. Programming all these handlers is tedious and error prone. Subsequent steps of e.g. initialization or of a protocol of server requests are scattered over multiple separated blocks of code. The control flow between these blocks is hard to retrieve. Some common variables have to be introduced in order to pass the application state between the different handler blocks. To overcome these problems, we propose to use an extension of UML statecharts, called action charts, dedicated to the modeling of callbacks and listeners. All kinds of handlers are modeled in a common uniform statechart notation. States become actions or handlers. Transitions represent the flow of execution. Variables are shared between actions providing a simple mechanism for passing the application state from one handler to the next. From such *action charts* we generate sourcecode that is compliant with the Google Web Toolkit (GWT). The generated code is pure Java code that facilitates validation and debugging of the modeled behavior. It can be translated to JavaScript and run inside the web browser using the GWT crosscomplier. The action charts and code generation are implemented as part of the open source CASE tool Fujaba.

1 Introduction

Modern web applications shift more and more application logic and even the data model to the client. This allows more interactive web applications (following the AJAX pattern). Furthermore, the traditional page-by-page application flow dissolves and is replaced by fully interactive widget based user interfaces with windows, buttons, lists, trees etc. This shift is possible due to several reasons: the runtime environment, the browser, has a powerful, fully programmable

JavaScript engine, and technologies like DOM and CSS allow the visible content to be manipulated in a very flexible manner during runtime.

Furthermore, development tools for web applications improve accordingly. Our work is based on the Google Web Toolkit [1] (GWT), which allows the application developer to implement the whole application, that is client and server code, in Java. Browser-understandable artifacts like HTML, CSS and ECMA-/JavaScript are generated on application deployment, the developer is faced with just a simple implementation language. The drawback of shifting the application more to the client is that we end up with a distributed application consisting of client and server components: our components have to communicate over an unreliable network connection with arbitrary latency. Dropping the server components completely is not possible, as the browser doesn't provide reliable persistent storage or even should not access or store data. If you take for example web shops, the provider wants to store order details and user information on server side for being able to finish the ordering process. Another case could be online homework systems, that can be used by schools and universities to administrate lectures. Here we have many privacy issues to solve. Only those parts of the application data model the client is allowed to change can be shifted to the client, restricting the access to the other parts. A student, for example, is only allowed to view his own grades, but is not allowed to view the ones of other students attending the same course. The access restrictions also have to ensure, that the student isn't able to change his grades.

The programming environments for web applications usually support client-server-communication following an asynchronous communication pattern, like GWT-RPC services. Because the underlying HTTP connection might fail and the server response time for a request is unknown, the client continues to work after issuing a server request, and will be informed by its local JavaScript Engine when the server response was received. That is called the asynchronous callback pattern. Technically, the client side of web applications is single-threaded, so this approach is necessary to have an always responding interactive application. But the asynchronous callback pattern is cumbersome for the application developer. When issuing a server request one has to pass a callback object and on server response the control flow continues in the corresponding callback object. Furthermore, when executing a sequence of server requests, the callback of the first request has to issue second request passing a callback that has to issue the third request and so on. Thus the code for this logically related sequence of steps is scattered over separated code blocks, in the case of Java even over separated classes. The control flow between these code blocks / classes is hidden in some callback parameters. In addition, these separated code blocks / classes need to provide extra means to enable the passing of a common application state between them.

A similar problem arises when developing the user interface: User input events are usually delivered through the observer pattern. So again, there is no sequential control flow which can be observed by looking at the code, but the event

[1] http://code.google.com/webtoolkit/

handling and application logic is scattered over or at least invoked by listener objects.

The approach presented in this paper shows Model Driven Engineering of Rich Internet Applications (RIAs) integrated into the CASE-Tool Fujaba (see: section 2). Using this Tool-Suite, which is build upon the Eclipse Platform, one is able to model the structure and behavior of an application via class diagrams and story diagrams. These diagrams are translated to Java code, afterwards. To enable the modeling of web applications, we adapted the standard Fujaba Tool and its diagrams. We are using statecharts with a special transition semantic to transparently map different kinds of events occurring in web applications. Because the execution semantics differs from statecharts, we call them *Action Charts*. This paper also includes an early proposal to bind the application's data model to user interface components (following the MVC pattern) with the same approach: data model changes are replicated to the client side and are able to update the corresponding UI component. This change mechanism is modeled with Action Charts based on adapted statecharts, too. Using adapted code generation mechanisms, we are able to generate completely functional web application code out of the Fujaba Tool. Formally, our Action Charts rely on the usual metamodel of uml statecharts. We use a stereotype ≪`ActionContainer`≫ to flag the different semantics and to trigger the special codegeneration.

1.1 Running Example

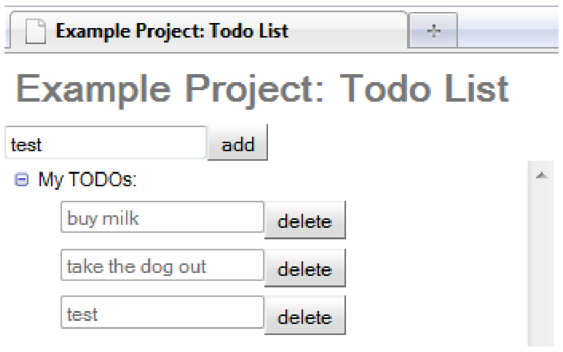

Fig. 1. Todo List User Interface

We have modeled a small application covering all the handlers and action chart events, as well as the persistent data storage and multi-user capabilities, called the *TodoListApplication*. Figure 1 shows the user interface of the application. Multiple users can share a TodoList, add new items, prioritize them and mark them as done. The GUI consists of several widgets: A tree with an input field and a Button per item. The Button is associated with a `ClickHandler`, the label is updated via a change event on the underlying `Entry` object residing in the shared data model. The application state is stored persistently on server side and replicated to all connected clients. This way, a user can close the application without loosing data. One example action chart implementing the application behavior can be seen in Figure 3, the corresponding application model consisting of just one class can be seen in Figure 2.

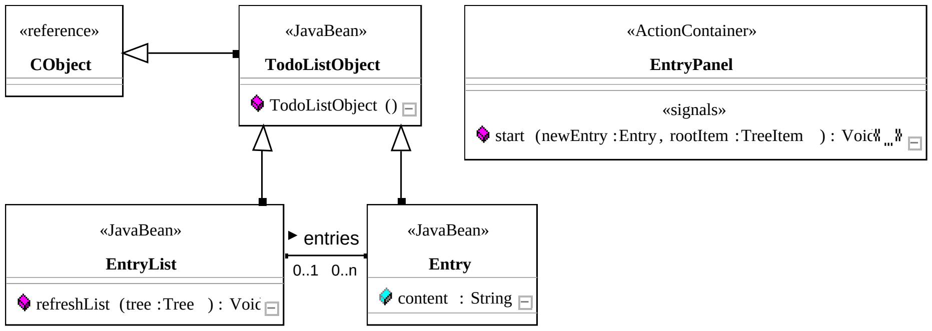

Fig. 2. Example Application: TodoList Class Diagram excerpt

This paper is structured as follows: First, in section 2 we will introduce the Fujaba Tool-Suite on which our approach is built. We will show the modeling and code generation capabilities of this tool and introduce our transformation chain. In section 3 we introduce our modeling approach using a running example. Section 4 describes in more detail the technical issues we have implemented in our action charts. Section 5 compares our approach to work that has already been carried out in the area of Web Engineering for Rich Internet Applications. Section 6 finally concludes and focuses on work we have planned for the future.

2 Modeling with Fujaba

The approach presented in this paper is implemented to extend and modify the existing statechart modeling capabilities of the Fujaba Tool Suite[2]. Fujaba is an open source CASE Tool, which offers software engineers to develop their applications completely model driven. It enables the design of an application using UML class diagrams, but also provides methods to create the application logic on model level, using story diagrams as visual programming language [4].

After modeling the application using the diagrams introduced above, the CodeGen2 Fujaba plugin [5] generates Java source code out of it. The code generation mechanism used in Fujaba is template based, which makes it easily

[2] http://www.fujaba.de

extendable, this in done for our approach of web development, for example. Another part of the Fujaba Tool which is extended for the use in web applications is the persistency and versioning library CoObRA [13]. This library works directly on the runtime object graph and saves it persistently as a stream of change events. We adapt this mechanism to provide persistency for our web clients, see below.

2.1 Story Driven Modeling

The most important part of the Fujaba notation are graph rewrite rules that allow to query and modify the application's object graph on a high level of abstraction. Figure 3 shows a cutout of an action chart used to initialize the applications user interface on the client side.

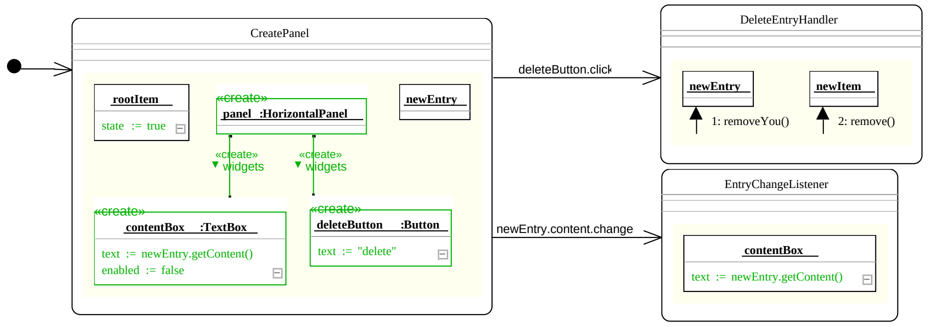

Fig. 3. Example Action Chart Diagram

Fujaba allows to embed graph rewrite rules into statechart states resp. actions of an action chart. These graph rewrite rules carry out it's operations. In Fujaba, a graph rewrite rule consists of an object graph that is used as a query pattern to be matched in the current runtime object graph. For example, the action `CreatePanel` of Figure 3 has a query pattern of three nodes `panel`, `contentBox`, and `deleteButton`. Our graph patterns distinguish bound objects and unbound objects. Bound objects are already matched to runtime objects and need not to be searched any more. In our notation, bound objects show only their name and omit their type. At execution time, unbound objects are matched onto runtime objects such that the overall match conforms to the search pattern graph. In addition to querying the runtime graph with a pattern graph, a graph rewrite rule also allows to model modifications of the matched elements. In our example, the `text` attribute of the `countLabel` object is changed to a new value by an assign expression. Graph elements may also be created or deleted using ≪`create`≫ and ≪`destroy`≫ stereotypes. Finally, a graph rewrite rule may contain collaboration messages allowing to do computations and to call methods on the matched objects.

Graph rewrite rules are an excellent means to model an applications behavior in terms of operations on its underlying object graph. The graph patterns allow to express complex graph queries that are (with some exercise) easy to read and to understand. Thereby, the programs are easier to extend and to maintain.

2.2 Model Transformation Chain and Application Architecture

As mentioned in the introduction, we try to shift as much as possible components of our application to the client side. Ideally, just the persistency layer remains on the server side. GWT supports that approach very well, because all code capable of running on the client also runs on the server, just the client-server communication (GWT RPC services) require a certain implementation convention. Because GWT already creates as client artifacts (HTML, CSS, JavaScript) out of Java source code, we just had to adapt our code generator to generate GWT compliant code and invoke GWT's Java-to-JavaScript compiler.

At runtime, the model is instantiated on both sides of the application and is synchronized, even between multiple clients and the server, using the WebCoObRA features introduced in [1]. This holds for the application data model which is generated out of the class diagram and its associated story diagrams. All the parts that are tightly related to the Graphical User Interface (GUI) part of the application are only run on the client side. This is the case especially for the action charts introduced in this paper, as these refer and instantiate GUI widgets, which are only available on the client side.

3 Modeling Approach and Application Architecture

As discussed, our typical web applications deploy model objects at runtime in the web client. GUI events are handled by listeners that query and modify the runtime model within the client. In addition, server requests may be issued. We use a MVC pattern to update the GUI in case of model changes. For persistency and multi user support, the runtime model is replicated on a server. These features can all be modeled using our action charts. Each action of such an action chart may contain a graph rewrite rule, which can call another action chart via collaboration statements. Furthermore, objects created or assigned in an action are action-chart-global, that means they can be referred or accessed in any successor action. Because each action is capable of receiving events at any time, our proposal doesn't follow a strict sequential execution. Reactive event-triggered actions resp. states can be seen as a short-hand notation for a parallel and-state with a waiting state preceding the actual reaction state. An action chart that has multiple reactive states (and thus many hidden wait states), actually treats all these event handlers as additional and-substates. However, all these parallel substates listen to disjoint events: each parallel substate listens to its own GUI or change or timer or RPC event. Because the client side JavaScript execution model is single-threaded, only one event is handled at a time and thus, although we deploy so many parallel substates, we don't observe any concurrency problems between these parallel substates. The following four sections discuss the distinct reactive features of our modeling approach:

3.1 RPC Services

Remote Procedure Calls (RPCs) are used in GWT applications to call methods on the server side of the application. Because of the non-blocking behavior of AJAX like applications, a callback object has to be passed when making the call. This callback object is used to inform the client side about the result of the call. The corresponding Interface of the GWT RPC framework distinguishes between a successful call and a call failure. RPC calls can be modeled in Fujaba graph rewrite patterns as collaboration statements.

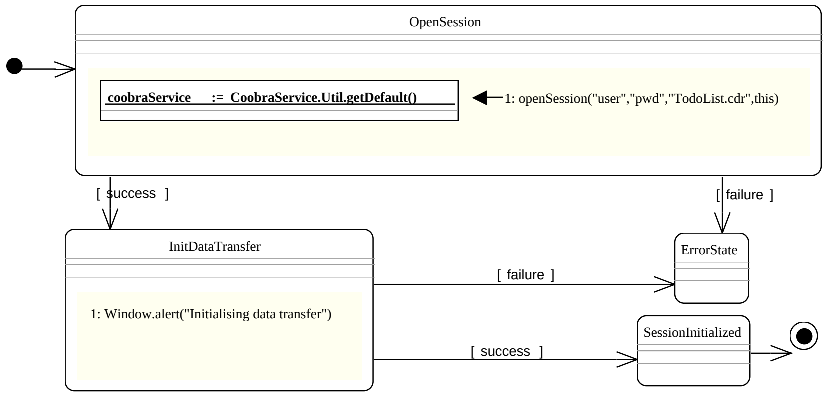
OpenSession
coobraService := CoobraService.Util.getDefault()
1: openSession("user","pwd","TodoList.cdr",this)
[success]
[failure]
InitDataTransfer
1: Window.alert("Initialising data transfer")
[failure]
ErrorState
[success]
SessionInitialized

Fig. 4. RPC Calls with action charts (excerpt)

The collaboration statement number 1 in action `OpenSession` shown in figure 4, calls the `openSession` method of the `CoobraService`. The last parameter of this method call indicates the callback object. As discussed in the realization section 4, at runtime each action of our action chart is turned into an inner class and a specific `FAction` object (cf. section 4). Thus, the `this` variable in the RPC call in state `OpenSession` refers to a runtime action object representing the `OpenSession` state. This makes the calling `OpenSession` action a callback. The action has two outgoing transitions, marked with success and failure. A successful RPC call will trigger the success transition to the next action, the failure transition could be used to display an error message, cf. section 4.

3.2 User Interface Events

During the execution of the initial state(s) of the application, one might initialize the user interface by instantiating GUI widget components and by building up the DOM tree. This can be done by adding widgets to the `RootPanel` consecutively. Figure 3 shows the initialization of some GUI widgets in the `CreatePanel` action. The definition of a `ClickHandler` for the instantiated button is shown in action `DeleteEntryHandler`, cf. Figure 3.

Click handlers are created by adding transitions labeled with `<objectname>-.click` to the action where the object is instantiated. The target action of this transition now works as Click handler and the designated behavior can be implemented by adding graph rewrite rules to this action. This graph rewrite rules may also use collaboration messages to invoke other methods that may implement more complex behavior. Such methods may be modeled using the usual Fujaba story diagram language, cf. [4]. Selection Events and Handlers for UI Components can be modeled the same way. The label of the outgoing transition has to conform `<objectname>.select` and the target action will be transformed to a Selection Handler, then.

3.3 Model changes via PropertyChange Events

Fujaba's code generation provides JavaBeans conforming setters and getters that also fire property changes. This enables our approach to react to model changes with PropertyChange events and the according handlers. In our action charts, we model the reaction to property changes as shown in action `EntryChangeListener` in Figure 3.

Action `OpenSession` of Figure 3 refers the `newEntry` object. This object has a property called `content:String`. To react to changes occurring to this property, we define a transition similar to the ones shown in section 3.2. The label of the transition now has to conform to `<propertyname>.change`. The target action of the transition now can be used to model the handling of the change event. Again, graph rewrite rules and method invocations can be used, here. The example shows how to set the text of the textbox to the value stored in the `content` property.

3.4 Timer Events

Timers are provided by the runtime environment and are currently the only possibility to achieve quasi-parallel execution of application logic. The application on the client side is limited by it's environment to utilize only a single thread. Using a timer, actions can be deferred or executed periodically, so this is an acceptable workaround and our proposal directly supports time-triggered actions. To schedule a timer callback object, we use `after <integer expression>` transitions. When the source action is reached, a GWT Timer is scheduled with the target action as handler object. The target action may again contain a graph rewrite rule and / or method invocations to model the desired handler operation.

4 Realization

To realize the modeling of web applications with our action chart approach it was necessary to completely change Fujaba's code generation concepts for statecharts.

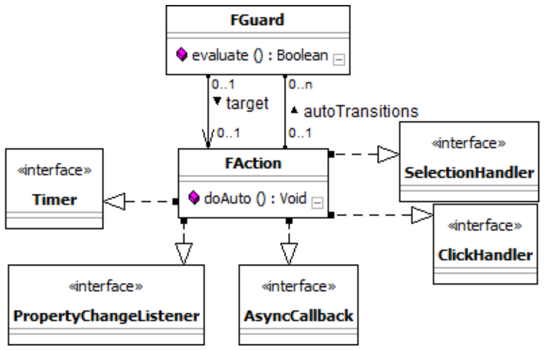

Fig. 5. Classdiagram showing parts of the runtime library.

Our new code generation concept turns each action into an inner class of the class owning the action chart. These inner classes have a `doAction` method that implements the corresponding graph rewrite rules. In addition, these inner classes inherit from our runtime library class `FAction` that in turn implements the interfaces of various GUI listeners, property change listeners, RPC callback classes, and GWT timer classes. Cf. Figure 5, which also shows an excerpt of the classes of the runtime library. Class `FAction` implements the different listener methods such that the action specific `doAction` is called.

In addition, we still generate an init method that creates at runtime an object structure that reflects the action chart structure. We also generate a `start` method that lazily calls the init method if the action chart object structure is not yet in place and that invokes the `doAction` of the initial action thus starting the action chart execution. At the end of each `doAction` a `doAutoTransitions` operation is invoked. This operation is generated within each `FAction` subclass. The `doAutoTransitions` operation evaluates the auto transitions leaving the corresponding action, evaluates the guard conditions and if true, the `doAutoTransitions` operation calls the `doAction` of the following action object. This realizes the synchronous execution of auto transitions in our approach.

If a listener transition leaves an action, we generate appropriate `addToListeners` method calls within the `doAction` of that action, which adds the neccessary listeners to the modeled objects at runtime. If e.g. a GUI event occurs, the corresponding listener object is informed via the usual listener mechanism and this eventually invokes the `doAction` of that listener action object.

As discussed, when calling a server operation via a RPC, we pass the current action object as the callback object parameter. Thus, on server response, the RPC runtime mechanism calls either method `onSuccess` or `onFailure` on the current action object. In class `FAction` we implement method `onSuccess` to lookup `success` links leaving the current action / callback object and to invoke the `doAction` of the reached action object. The `onFailure` method follows `failure` links, accordingly. In addition, the `resultValue` of the reached action object is set appropriately.

In our existing code generation concept for statecharts, each runtime statechart interpreter was running in its own thread. In a web browser, the whole Javascript application of a single web page runs in one single thread. The Javascript engine of this single thread handles all GUI events, timer events and RPC callback events. Thus, all these events are sequentialized by the Javascript engine and only one event is handled at a given time. This relieves us from many concurrency issues.

5 Related Work

The model driven development of web applications is analyzed by many research groups over the world. From our perspective there have been four that seem to be the most relevant ones.

First of all, there is the Object-Oriented Hypermedia Design Method (OOHDM) [14]. This method comprises of four steps: conceptual modeling, navigational design, abstract interface design and implementation. A second method, is UML-based Web Engineering (UWE) [10]. UWE is consists of five different phases and uses UML notations to model the web application. Another modeling approach mostly used for the development of e-commerce applications and hypermedia information systems is given with the Object-Oriented Web-Solutions (OOWS) [11]. This approach is supported by a commercial tool called OlivaNova. Lastly the Web Modeling Language (WebML) [2] introduces a notation to specify at a conceptual level complex websites. This notation is supported by a development environment called WebRatio in which web applications can be modeled and automatically generated.

Nevertheless, the four approaches mentioned before were intended to develop traditional page based web application, while the approach presented in this paper clearly focuses on Rich Internet /Ajax Applications. Model driven development of these kind of applications has become a major research activity in the area of model driven web engineering. There already exist a few approaches in the literature that deal with the special issues raised by Rich Internet Applications. For example there exists an extension for the UWE method intended for the development of RIAs [8]. Also the OOHDM modeling approach has been extended for the new needs of Rich Internet Applications, as can be seen in [9].

One main difference between these existing approaches and the modeling approach introduced in this paper is our ability to model application logic. This can be done even inside the action charts which are used to create event listeners, asynchronous remote procedure calls and other client side logic. The whole application can be modeled, no switch to code is needed.

[9] and [12] introduce modeling approaches which seem very close to the one introduced here, first. But in [9] and [12] it is necessary to have different models for navigation, application logic and the orchestration of user interface components. The orchestration model uses statecharts as well, but is different in the handling of these. Using story diagrams inside our states we are able to clearly express inside the statechart which actions to the application model and the user interface have to invoked when a state is reached. Additionally, applications created using our approach only have one entry point. This way, we completely avoid page changes inside the application. [7] are able to create events and timers using statecharts, as well. Again we see two benefits using the Fujaba approach. First we again have graph rewrite rules, which enable us to model complex object graph queries and modifications. Secondly, using our code generation, we provide the complete MVC pattern in our web applications. This means we are able to react to property change events occurring on client side. Such property changes can be modeled using statecharts, too.

6 Summary and Future Work

We have presented a new modeling approach for web applications. This approach allows to model RPC callbacks, GUI listeners, property change listeners, and timers in uniform way within action charts. The actual model query and modification operations are modeled using graph rewrite rules embedded in the actions. These rules share their object variables. Thus, it is easy to pass common model objects from one (construction) action to a subsequent (callback) action or to listener actions. Due to our experiences this is a tremendous improvement of the old situation, where each callback and each listener had to be modeled in its own class and passing model elements required explicit attributes in those classes. Challenging issues for the future of client-side web applications arise from the more and more growing capabilities of the browser runtime environment. We are exploring client-side persistency mechanisms to finally build a distributed persistence layer covering many client nodes and the server, which should introduce redundancy only when necessary or beneficial. With our new action chart approach, we are now able to model all parts of a web application, its GUI construction, the GUI handlers, RPC callbacks, timers, property change listeners, persistency and replication mechanisms, and client side model data in a concise and uniform notation. This paper presents a very simple web application. In the context of our FAST project [3] we have used this approach to build a relatively complex data transformation tool and a data type definition tool. Thus, we are confident that the approach scales to real world applications. Full support for all user input events (mouse actions, CSS events) is work in progress while our examples evolve. However there is still more future work to be done: Defining the user interface using story diagrams is somehow more intuitive than textually coding it, but is still cumbersome. With graphically modeled GUI widgets, one can see more clearly the relations, but the graph approach doesn't satisfy the hierarchical property of GUI components (there's always a root component with child components). A lot of components have to be created and placed in order to form a graphical user interface that is capable of doing its designated tasks. To ease this step of the design phase for web applications, we currently develop a What-You-See-Is-What-You-Get Editor for GWT and SmartGWT widgets. You can review the whole designer idea in [6].

References

1. N. Aschenbrenner, J. Dreyer, M. Hahn, R. Jubeh, C. Schneider, and A. Zündorf. Building Distributed Web Applications based on Model Versioning with CoObRA: an Experience Report. In *Proc. 2009 Intl. Workshop on Comparison and Versioning of Software Models*, pages 19–24. ACM, May 2009.
2. S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, and M. Matera. *Designing Data-Intensive Web Applications*. Elsevier, Amsterdam, Netherlands, December 2002.
3. The FAST Project. http://fast-fp7project.morfeo-project.org/, 2010.
4. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In *Proc. of the* 6^{th} *International Workshop on Theory and Application of Graph Transformation*. Paderborn, Germany, 1998.
5. L. Geiger, C. Schneider, and C. Reckord. Template- and modelbased code generation for MDA-Tools. In *3rd International Fujaba Days*, Paderborn, Germany, September 2005.
6. N. Geiger, C. Eickhoff, M. Hahn, I. Witzky, and A. Zündorf. Future web application development with fujaba. In P. V. Gorp, editor, *Proceedings of the 7th International Fujaba Days*, pages 51–55, November 2009.
7. N. Koch, M. Pigerl, G. Zhang, and T. Morozova. Patterns for the model-based development of rias. In M. Gaedke, M. Grossniklaus, and O. Díaz, editors, *ICWE*, volume 5648 of *Lecture Notes in Computer Science*, pages 283–291. Springer, 2009.
8. L. Machado, O. Filho, and a. Ribeiro, Jo˙Uwe-r: an extension to a web engineering methodology for rich internet applications. *WSEAS Trans. Info. Sci. and App.*, 6(4):601–610, 2009.
9. S. Meliá, J. Gómez, S. Pérez, and O. Díaz. A model-driven development for gwt-based rich internet applications with ooh4ria. In D. Schwabe, F. Curbera, and P. Dantzig, editors, *ICWE*, pages 13–23. IEEE, 2008.
10. A. K. Nora Koch. The expressive power of uml-based web engineering. pages 105–119, 2002.
11. J. F. Oscar Pastor, Silvia Abrahao. An object-oriented approach to automate web applications development. In *Electronic Commerce and Web Technologies*, pages 16–28, 2001.
12. S. Pérez, O. Díaz, S. Meliá, and J. Gómez. Facing interaction-rich rias: The orchestration model. In D. Schwabe, F. Curbera, and P. Dantzig, editors, *ICWE*, pages 24–37. IEEE, 2008.
13. C. Schneider. *CoObRA: Eine Plattform zur Verteilung und Replikation komplexer Objektstrukturen mit optimistischen Sperrkonzepten.* PhD thesis, 2007.
14. D. Schwabe and G. Rossi. The object-oriented hypermedia design model. *Commun. ACM*, 38(8):45–46, 1995.