Proceedings of the IASTED International Conference on Software Engineering (SE2006), Pages 13-18, Innsbruck, Austria, 2006

WEB APPLICATION DEVELOPMENT EMPLOYING DOMAIN-SPECIFIC LANGUAGES

Martin Nussbaumer, Patrick Freudenstein, Martin Gaedke

University of Karlsruhe, Institute of Telematics,
IT-Management and Web Engineering Research Group
Engesser Str. 4, D-76128 Karlsruhe, Germany
{nussbaumer, freudenstein, gaedke}@tm.uni-karlsruhe.de

ABSTRACT
In Web application development projects, the specification of the envisioned solution is a time-consuming task suffering from communication problems between the developers and the business. Based on our experiences gained in several real-world projects, we propose an approach combining Domain-Specific Languages and a supporting technical platform. Web application development can thus be performed by composing building blocks and configuring them with DSL programs.

KEY WORDS
Web Engineering, Domain-Specific Languages, Conceptual Modeling, Reuse, XML and Web Services

1. Introduction

Assuring efficient and clear communication between the various actors in software development projects is a key factor. Based on the evaluation of meanwhile over 50,000 IT projects within the last ten years, the Standish Group's annual CHAOS Reports list aspects like strong user involvement and clear business objectives among the top five success factors for IT projects [1]. Particularly in the fields of requirements specification and conceptual design establishing a common understanding and avoiding misunderstandings between the developers and the business becomes decisive.

In the context of Web Engineering, a multitude of approaches allowing for an extensive, systematic and formal specification of aspects of a distributed Web-based solution have emerged, e.g. [2, 3]. They usually provide very expressive and powerful concepts and notations which makes them a good means of communication within the developer team. However, learning these languages is usually a rather time-consuming task. For stakeholders involved in Web application development projects, who are mostly non-programmers with diverse academic and non-academic backgrounds, this effort is often unreasonably high.

In contrast to these "heavy" languages, Domain-Specific Languages (DSLs), also known as "Little Languages" [4], have recently gained increasing attention. In [5] they are defined as "programming languages or executable specification languages that offer, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain". Being highly focused on a small, specific aspect of the solution and providing dedicated concepts and (graphical) notations from the problem domain, DSLs are easy to learn, understand and use, both for the developers *and* the business. The usability of a DSL can be further enhanced by supplying dedicated graphical representations and accompanying editors for particular stakeholder groups. Similar to programs written in a general purpose language, e.g. Java or C#, a DSL program can be transformed into executable code by a dedicated DSL compiler.

Our vision of DSL-based Web Engineering is to enable stakeholders and domain experts themselves to understand, validate, modify, and even develop aspects of a distributed Web-based solution on the basis of DSLs. Thus, we aim at smoothing communication problems and misunderstandings out and thus achieving more efficient and more successful Web application development projects.

The remainder of this paper is organized as follows. In section 2, we motivate our idea of DSL-based Web Engineering based on our lessons learned in a large-scale university-wide Enterprise Application Integration (EAI) project. Following, in section 3, we present our approach towards DSL-based Web Engineering, consisting of an evolutionary DSL framework and an underlying technical platform. In section 4, we portray an extract of our DSL catalogue and describe one DSL dealing with form-based user interaction in more detail. Afterwards, in section 5, we perform the evaluation of our approach based on a real-world scenario. Finally, we draw the conclusions and present future work.

2. Lessons Learned

In the following, we report from our experiences gained in several real-world Web application development projects and in particular in the large-scale university-wide Enterprise Application Integration (EAI) project "Karlsruhe's Integrated Information Management (KIM)" [6]. In the following, we portray the communication problems between the various project participants and

stakeholders arising in the course of requirements specification and conceptual design.

Especially in projects dealing with the development of distributed, large-scale Web-based solutions, communication between the business and the developers is being aggravated by the great diversity of involved stakeholders. This is due to the fact that, in such distributed scenarios, the stakeholders usually have completely different professional and educational backgrounds. Thus, it is comprehensible that each group uses its own "language" when talking about aspects of the solution. Regarding the specification of business processes in the KIM project, for example, the variety of languages ranged from written natural language over Petri nets [7] to the Business Process Modeling Notation (BPMN). In addition, we found stakeholders to be predominantly non-programmers and rarely available for interview sessions. Consequently, in order to assure an efficient and preferably far-reaching collaboration between stakeholders and developers, languages for the design of facets of a distributed Web-based solution should be easy to learn, understand and use.

Over the last years, a lot of languages trying to cover their problem domain as exhaustive as possible by including a broad range of concepts and notations have been elaborated. In our experiences, these kinds of languages proved to be a good means of conceptual and logical design within the developer team. Surprisingly, in the most cases, our attempts of employing them for the collaboration with stakeholders failed due to their complexity. The time and effort necessary for learning a common set of such languages turned out to be not feasible.

Hence, we recognized simplicity as a key factor to usability and effectiveness. Based on our experiences, using small and simple languages and thus empowering all stakeholders to understand and use them turned out to be a more successful option. In the majority of cases, a language covering about 80% of a problem domain was completely sufficient and yielded in turn to a much better usability and efficiency. Further improvements could be achieved by including high-level abstractions and concepts of the problem domain and by providing graphical notations tailored to the characteristics and needs of particular stakeholder groups.

3. A DSL-based Web Engineering Approach

With regard to our experiences as described in the previous section, we discovered Domain-Specific Languages (DSLs) to be an ideal alternative to the existing "heavy-weight" conceptual design languages. DSLs can be characterized as small, simple and highly focused languages for specifying clearly identifiable aspects of a solution. Moreover, they employ idioms, concepts and (graphical) notations of the associated problem domain. Thus, they are easy to learn, understand and use, especially for domain experts without software development skills.

Figure 1 shows the elements of our approach towards DSL-based Web Engineering which is based on the principles of evolution and reuse. We differentiate between two phases in the course of a continuous evolution: Development for Reuse comprises the design and development of a DSL and Development with Reuse covers the usage of a DSL for the specification and development of a part of a distributed Web-based solution.

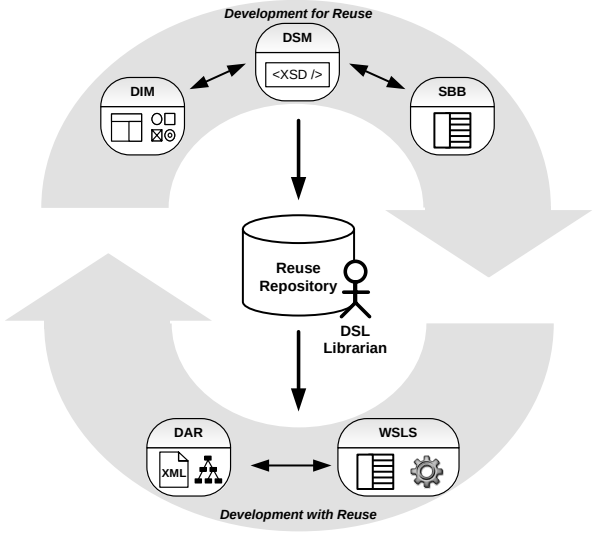

Figure 1: Overview of our evolutionary and reuse-oriented approach towards DSL-based Web Engineering

In our approach, a DSL consists of three components. The *Domain-Specific Model (DSM)*, usually an XML Schema Document, represents the formal schema for all solutions that can be described with the DSL. Thus, the DSM has to be designed in accordance with the problem domain the DSL is intended for. Based on the DSM, a *Domain Interaction Model (DIM)* comprises a dedicated (graphical) notation being as intuitive as possible for a particular stakeholder group. The DIM is tightly coupled to the DSM; however, it needs not to cover all of its aspects. By using a DIM, stakeholders can employ the DSL, i.e. understand, validate and even create DSL programs, without being confronted with complicated source code. Instead, the DIM should provide concepts and notations derived from the problem domain and thereby should be easy to understand and use. In order to meet different requirements and characteristics of various stakeholder groups, a dedicated DIM for each group could be included in a DSL. A further enhancement to the usability and effectiveness of a DSL can be achieved by accompanying editing tools based on the notations specified in the DIMs.

Besides the design of a DSM and one or more DIMs, the development of a *Solution Building Block (SBB)*, being capable of executing the DSL programs, completes the "Development for Reuse" phase. An SBB can be seen as a software component whose behavior can be configured through a DSL program, usually in terms of an XML document. We use the WebComposition Service Linking

System (WSLS) [8] as technical framework for the SBBs. WSLS is based on the core ideas of the WebComposition approach [9] and aims at facilitating the systematic evolution of Web applications by reusing software artifacts and emphasizing the "configuration instead of programming" paradigm. In our approach, the WSLS framework allows for the systematic composition and configuration of SBBs.

During the "Development with Reuse" phase, the *Domain Abstract Representation (DAR)* is developed. The DAR represents the specification of a concrete solution within the DSL's problem domain. In other words, the DAR is a DSL program. Consequently, it is based on the DSM and modified by using one or more DIMs. As the DSM is usually specified as an XML Schema, the DAR is serialized and stored in an XML document based on the DSM. However, in contrast to today's integrated development environments (IDE), the editing process using DIM notations is not performed on this serialized form. Modifications are rather carried out directly on the abstract model itself. Thus, DSL programs can be edited in a more powerful way than it would be possible if interacting with the DAR's serialized form. After having developed a DAR, its XML representation is passed to the DSL's associated SBB, i.e. a component of the WSLS framework. The SBB in turn adapts its behavior according to the DAR and thereby executes it. Web application development can thus be performed in an evolutionary manner by composing SBBs and configuring them with DARs.

Both the emerging variety of DSLs for the diverse aspects of distributed Web-based solutions and each DSL itself underlie a continuous evolution. Based on experiences gained from employing DSLs in collaboration with various stakeholders, existing DSLs are improved, new ones are developed and some could even be removed. To support a systematic and efficient usage and management of DSLs in such an evolutionary environment, we propose the installation of a central *Reuse Repository* [10] and the incorporation of a *DSL Librarian* team role.

The Reuse Repository acts as the central storage for DSLs and associated metadata and provides means for their efficient organization, management and retrieval. During the "Development for Reuse" phase, new or modified DSLs are classified and stored in the repository. Therefore, versioning features and a sophisticated classification schema are essential. Later on, in the "Development with Reuse" phase, existing DSLs are searched in the repository depending on factors like the problem domain, the application type or the kind of stakeholders. This again requires an advanced classification schema and appropriate search functionalities.

The DSL Librarian is responsible for the efficient management and usage of DSLs. She accompanies the project team throughout the development process and puts emphasis on efficient reuse when developing and consuming DSLs. Furthermore, she assists in finding and applying DSLs and is in charge of the administration of the Reuse Repository and its contents as well as the conservation of gained experiences and knowledge.

4. DSL Catalogue

In this section, we describe three DSL's out of our Reuse Repository catalogue targeting important aspects of Web applications: Navigation, Web-based process guidance, and form-based user dialogs. The presentation of each DSL is divided up into a statement about the problem domain, the description of the Domain-Specific Model and the introduction of a Domain Interaction Model. While the first two DSLs are depicted focusing only on their most important characteristics, the form-based user interaction is described in more detail.

4.1 Link List

Problem Domain: Building linking structures, e.g. menus or index, by interconnecting application domains and their related chunks of information is a central task in the development of Web applications.

Domain Specific Model: The Link List DSM is based on XLink and provides concepts for structuring links and specifying the traversal behavior from the source to the target (embed or replace).

Domain Interaction Model: So far, we have developed a DIM which defines symbols for a "Level", an "Internal Link", an "External Link" and a "Connector". A "Level" groups a set of links, the link types specify the link target and the traversal behavior, and the "Connector" realizes the aggregation of links and nesting of levels.

4.2 Web-Based Process Guidance

Problem Domain: In Web applications, user guidance processes, i.e. the traversal of a set of application domains according to events triggered by the user, play an important role.

Domain Specific Model: The concepts known from finite State Machines (FSM) represent the foundation of our DSM. The realization of our DSM is based on XLink and includes elements for specifying states (i.e. application domains), transitions and events.

Domain Interaction Model: Due to the characteristics and requirements found in the KIM project, we employed Petri nets as a DIM, just focusing on very simple and intuitive constructs.

4.3 PetriX Dialog Modeling

Problem Domain: Terry R. Schussler stated at the "Mac World San Francisco '98" conference: "Interaction is not animation. It's not audio. It's not video. It's user control and dynamic experience." Especially in the graphical environment of the Web, interface design has to deal with constructing meaningful behavior while placing interaction gadgets. Thus, in order to foster comprehensive dialog modeling towards dynamic user experiences, a dedicated DSL is needed.

Domain Specific Model: We decided to use XForms [11] for our DSM as it is a recommendation of the W3C. Although most user agents still don't support native XForms, most of the big vendors promised support, like e.g. the Mozilla browser [12] or a plug-in for the Internet Explorer. While traditional form mechanisms introduced in HTML 2.0 are fundamental for HTML, XForms will become an integral part of the XHTML 2.0 specification. Furthermore, parts of XForms can be reused in other markup languages like SMIL, SVG or WML.

XForms distinguishes three parts to separate presentation from data and behavior: the *XForms model*, *instance data* and the *user interface*. The *XForms model* solely describes the logical elements of a form. Therefore, it consists of an *XML instance* that collects the values entered by a user while editing the form. Beyond that, the model provides submission information about the server's location and the encoding style. In contrast, the *user interface* describes layout and appearance of embedded form controls that are bound to the model's data instance. Data binding in XForms is established by using XPath expressions.

XForms allows for declaring XML event handlers that are capable of capturing high-level semantics like e.g. displaying a message box. Thus, an XForms-based application enhances the accessibility while not exclusively relying on scripting technologies as it is common today. Hence, each form control possesses the ability to define events and event handlers according to XML Eventing.

Figure 2 gives an overview of the controls defined in XForms and the additional structuring elements "group", "repeat" and "switch". The "Interaction Carrier" is responsible for linking the controls with the model's data instance by referencing them via XPath expressions or XForms-specific bindings. Furthermore, an interaction carrier provides means for defining action statements to specify dedicated behavior.

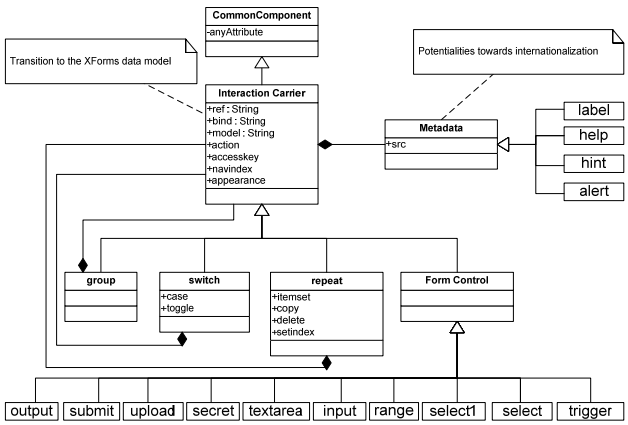

Figure 2: The XForms user interface concepts as a high-level UML class diagram.

Domain Interaction Model: As a means for supporting dialog engineers and stakeholders in creating and modifying user interface descriptions, we customized Microsoft Visio with dedicated support for PetriX diagrams. The *PetriX* approach combines the power of Petri nets [7] for describing workflows and behavior with the abstraction and extensibility of the XForms controls to create user interfaces. Consequently, a form modeled with PetriX is divided in two logical units: partitions (places) and interaction structures (transitions that connect places).

A partition place encapsulates a set of interaction carriers which group and enrich them with additional layout information. The layout of a partition can be specified by applying Cascading Style Sheet (CSS) attributes and providing an appearance hint on how the partition should be rendered on a client. Partitions can be nested and thus can contain further partitions.

An interaction structure defines the behavior between the connected partition places. Hence, an interaction structure can dynamically control which parts of a form are visible and thus can be filled out by a user. Especially if parts of the form are dependent from data values or user actions, an interaction structure reduces the form's complexity by hiding unnecessary parts. Figure 3 shows two typical interaction structures that are common in form design: *previous-next* and *choice*. Their behavior is expressed with a Petri net transition that can be triggered by dedicated form controls in a partition.

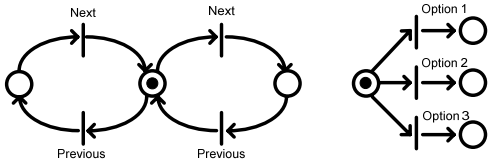

Figure 3: The Interaction Structures "Previous-Next" (left hand) and "Choice" (right hand) in their Petri net representation.

After the dialog structure has been modeled with Visio, the Petri net-like models are mapped to XForms code using XML transformations. Therefore, we developed an XSLT-based transformation engine which takes the XML export from Visio as input. The resulting Domain Abstract Representation is then passed to a corresponding Solution Building Block being able to render the XForms code using a mixture of HTML and JavaScript.

5. PetriX applied – an example

In the following, we outline how the PetriX DSL can be used in practice based on an example from our university. Within the scope of the development of an employee self-service portal for our university's department, we reused the experiences gained from the KIM project. As an initial step, we built the portal structure and components on top of our technical platform WSLS by composing Solution Building Blocks configured with appropriate DSLs.

Figure 4 depicts the Domain Interaction Model for PetriX which was introduced in section 4.3. We customized

Microsoft Visio to support dialog engineers in creating user interfaces by using the drag & drop facility of the graphical editor. Thus, it allows the placement of the pre-defined model symbols and their annotation with additional properties from the fields of behavior, attributes and usage. In Visio, a so-called "stencil" facilitates the definition of graphical notation sets as depicted on the left hand side of Figure 4 (1a, 1b).

We defined the two stencils: *Petri Net* to model behavior and *XForms* to support the basic control module of XForms. The drawing canvas contains the form partitions which are connected by transitions to shape the dedicated behavior imposed by the interaction structures (Figure 4 - area 2). The depicted diagram displays a *choice* allowing for the activation of one of the connected partitions depending on the selected value of the `xforms:select1`. Visio offers means for the annotation of the pre-defined graphical notations. The modular characteristics of XForms for combining attribute groups facilitate managing and applying annotation sets (3a). The selected `xforms:input` control comprises a dedicated set of annotation attributes like data referencing or CSS support (3b).

In the following we describe how a dialog modeled with PetriX can be mapped to a corresponding XForms code fragment. Therefore, we regard the scenario depicted in Figure 4for travel expenses, where a user first selects the type of transportation, i.e. one of the values "car", "train" or "plane". Depending on that selection, the relevant part of the form is rendered dynamically to query only relevant values.

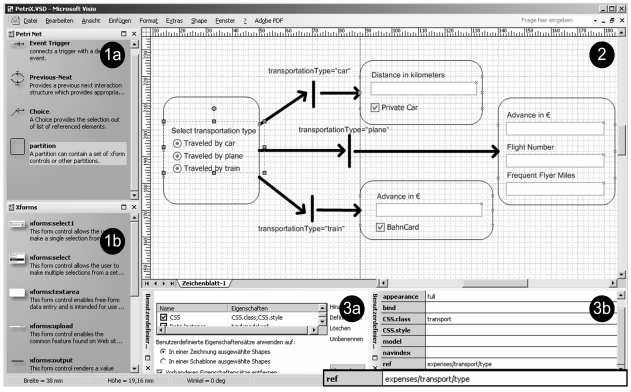

Figure 4: The PetriX DIM realized with Microsoft Visio

In the case of the `xforms:select1` control, the selected value is bound via an XPath expression (expenses/transport/type) to a data instance which is part of the XForms model (1).

```
<xforms:model>                                          (1)
 <xforms:instance>
  <expenses xmlns="urn:forms:expenses">
   ...
   <transportation>
     <type />
     <distance />
     <privateCar />
     <flightnumber />
     <frequentFlyerMiles/>
     <bahncard />
   </transportation>
   ...
  </expensesForm>
 </xforms:instance>
</xforms:model>
```

After the PetriX diagram is completed, it is exported from Visio as an XML document. We used an XSL(T)-based transformation tool which provides dedicated support for processing Visio Xml documents. The transformation of the PetriX model is realized according to *term rewriting*. In the depicted example, the transitions from the selection partition to the partitions *car*, *plane* and *train* can be expressed by the XForms module *switch* as follows: The `xforms:select1` control is bound to the corresponding data instance and writes the current selected value, e.g. car. The corresponding XForms code fragment can be constructed by term rewriting according to the annotated values like e.g. the referenced XPath expression (2).

```
<select1 ref="expenses/transport/type">      (2)
 <label>Select transportation type</label>
  <item>
    <label>Traveled by car</label>
    <value>car</value>
  </item>
  <item>
    <label>Traveled by train</label>
    <value>train</value>
  </item>
  <item>
    <label>Traveled by plane</label>
    <value>plane</value>
  </item>
</select1>
```

The *choice* interaction structure from our example is expressed by using the XForms *switch* module (3). For each connected transition an XForms *case* is rendered which can display the relevant part of the form at runtime. This relevance is expressed by the current selected value of the `xforms:select1` control.

```
<switch ref="expenses/transport/type">       (3)
 <case id="car">
  term rewriting of connected partition "car"
 </case>
 <case id="plane">
  term rewriting of connected partition "plane"
 </case>
 <case id="train">
  term rewriting of connected partition "train"
 </case>
</switch>
```

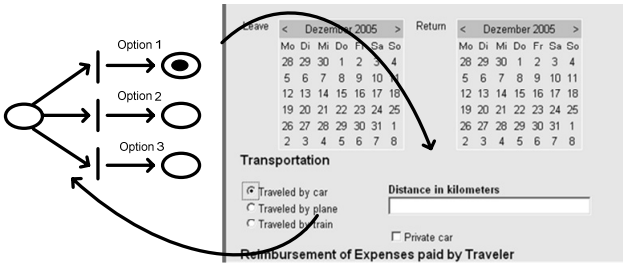

Figure 5: The interaction structure "Choice" and the corresponding rendered XForms dialog.

Figure 5 depicts the relevant part of the rendered form, which corresponds to the given example "Select Transportation". The interaction structure is in the state after a user has selected the option "car". Consequently, the form only renders the controls associated with the partition "car".

6. Summary & Future Work

In this paper, we presented our approach towards the development of distributed Web-based solutions employing Domain-Specific Languages. We reported from our experiences in the field of communication between developers and stakeholders gained in several real-world projects and pointed out the major difficulties leading to misunderstandings and insufficiently or not fulfilled requirements. Existing conceptual modeling languages offer complex concepts and notations and attempt to cover their problem domain as exhaustive as possible. However, regarding the collaboration with stakeholders who are usually non-programmers and have no experiences in the field of software development, they require too much time and effort for learning, understanding and using them.

Our approach's underlying vision is the direct involvement of stakeholders in the development effort by enabling them to autonomously understand, validate and specify parts of a distributed Web-based solution. Thus, simplicity is a key requirement. We identified Domain-specific Languages (DSLs) to be an ideal building block for our approach. DSLs can be characterized as simple, highly-focused languages for solving clearly identifiable aspects of a distributed Web-based solution and incorporate concepts, idioms and notations from the problem domain. As they are subject to continuous evolution, we introduced an evolutionary framework for their systematic development, management and usage. Our technical platform for the execution of DSL programs consists of Solution Building Blocks (SBBs) who can be systematically composed and configured with DSL programs on the basis of the WebComposition Service Linking System (WSLS).

Following, we presented an extract from our DSL catalogue and described a DSL for the problem domain of form-based user interaction called "PetriX". The DSL's formal model employs concepts from XForms and Petri nets and includes an easy to use graphical notation which was supplemented by a dedicated tool support based on Microsoft Visio. In the last section, we exemplarily demonstrated the usage of PetriX in a real-world scenario dealing with the development of a travel expenses report feature within an employee self-service portal.

Future work deals with the continuous evolution of our DSL catalogue based on the experiences gained in current projects. We aim at improving existing DSLs and developing new ones in order to cover more and more aspects of distributed Web-based solutions. With respect to the PetriX DSL, we are analyzing requirements concerning behavioral aspects of forms and user interaction in general and derive further interaction structures to be included in the DSL. Furthermore, advanced presentation aspects should be considered. Beyond that, we strive for enhancements of our approach's technical infrastructure like a semantic classification schema for the repository allowing for context-based searches and an integrated environment for DSL-based Web application development.

7. References

[1] The Standish Group International, *CHAOS Research* - Research Reports, 1994-2005: http://www.standishgroup.com.

[2] Ceri, S., Fraternali, P., and Bongio, A., Web Modeling Language (WebML): A Modeling Language for Designing Web Sites, *Proc. 9th International World Wide Web Conference (WWW)*, Amsterdam, Nethderlands, 2000, 137-157.

[3] Schwabe, D., Rossi, G., and Barbosa, S., Systematic Hypermedia Design with OOHDM, *Proc. ACM International Conference on Hypertext' 96*, Washington, USA, 1996.

[4] Bentley, J.L., Programming pearls: Little languages, *Communications of the ACM*, *29* (8), 1986, 711-721.

[5] Deursen, A.V., Klint, P., and Visser, J., Domain-Specific Languages: An Annotated Bibliography, *ACM SIGPLAN Notices*, *35* (6), 2000, 26-36.

[6] *KIM Project Homepage* - 2005, University of Karlsruhe: http://www.kim.uni-karlsruhe.de/ (24.04.2005).

[7] Petri, C.A., *Kommunikation mit Automaten*, Dissertation, Technische Universität Darmstadt, Darmstadt, 1962.

[8] Gaedke, M., Nussbaumer, M., and Meinecke, J., WSLS: An Agile System Facilitating the Production of Service-Oriented Web Applications, in *Engineering Advanced Web Applications*, S.C. M. Maristella, Editor. 2005, Rinton Press. p. 26-37.

[9] Gaedke, M. and Turowski, K., Integrating Web-based E-Commerce Applications with Business Application Systems, *Netnomics Journal, Baltzer Science Publishers*, *2* (2000), 2000, 117-138.

[10] Gaedke, M., Rehse, J., and Graef, G., A Repository to facilitate Reuse in Component-Based Web Engineering, *Proc. International Workshop on Web Engineering at the 8th International World-Wide Web Conference (WWW8)*, Toronto, Ontario, Canada, 1999.

[11] Dubinko, M., et al., *XForms 1.0* - W3C Recommendation, 2003, W3C: http://www.w3.org/TR/2003/REC-xforms-20031014/.

[12] Beaufour, A., et al., *Mozilla XForms Project* - 2005: http://www.mozilla.org/projects/xforms/.