Domain-specific Languages for Service-oriented Architectures: An Explorative Study

Ernst Oberortner, Uwe Zdun, Schahram Dustdar

Distributed Systems Group, Information Systems Institute,
Vienna University of Technology, Vienna, Austria
{e.oberortner|zdun|dustdar}@infosys.tuwien.ac.at

Abstract Domain-specific languages (DSLs) are an important software development approach for many service-oriented architectures (SOAs). They promise to model the various SOA concerns in a suitable way for the various technical and non-technical stakeholders of a SOA. However, so far the research on SOA DSLs concentrates on novel technical contributions, and not much evidence or counter-evidence for the claims associated to SOA DSLs has been provided. In this paper, we present a qualitative, explorative study that provides an initial analysis of a number of such claims through a series of three prototyping experiments in which each experiment has developed, analyzed, and compared a set of DSLs for process-driven SOAs. Our result is to provide initial evidence for a number of popular claims about SOA DSLs which follow the model-driven software development (MDSD) approach, as well as a list of design trade-offs to be considered in the design decisions that must be made when developing a SOA DSL.

1 Introduction

Service-oriented Architectures (SOA) use platform-independent interfaces, or services, for performing business processes [7]. A SOA, in which services realize individual process steps or tasks, is called a *process-driven SOA* [27]. Process-driven SOAs deal with multiple concerns, such as orchestration of business processes, information in processes, collaboration between processes and services, data, transactions, human-computer interaction, service deployment, and many more. Hence, many domains need to be considered. Furthermore, a SOA has different stakeholders, including various domain experts and technical experts [25].

Using Domain-Specific Languages (DSL) for SOAs, based on Model-driven Software Development (MDSD) [19,8], enables technical experts and domain experts to work at higher levels of abstraction compared to using technical interfaces, executable process models, or service interface descriptions, such as programming APIs, Business Process Execution Language (BPEL) code, or interfaces described in the Web Service Description Language (WSDL). Furthermore, MDSD can be used for separating concerns. Hence, the multiple concerns of process-driven SOAs can be modeled independently through MDSD.

This paper presents an explorative study in which we developed a number of MDSD-based DSL prototypes, as well as a model-driven infrastructure to generate a running process-driven SOA from the models expressed in the DSLs. We present three

experiments, in which we have focused on finding design decisions and/or trade-offs for developing model-driven DSLs. DSLs for domain experts (from now on called *high-level DSL*) and DSLs for technical experts (from now on called *low-level DSL*) were designed and developed. Two of our experiments deal with model-driven DSLs developed for process-driven SOAs, and the third one focuses on extending SOAs with Web user interfaces (UI), i.e., non-process-driven SOA concerns.

An in-depth study of specific claims about model-driven DSLs for SOAs was conducted. The claims target on (1) a systematic development approach for model-driven DSLs for SOAs, (2) the multiple concerns of SOAs, (3) the different levels of abstraction presented to the different stakeholders, and (4) on providing facilities for extensions. An analysis of the claims was made for each experiment in order to collect evidences and counter-evidences for claims about model-driven DSLs for SOAs. Hence, our results aim to help in making design decisions and considering the relevant design trade-offs, when engineering a model-driven DSL for SOAs.

This paper is organized as follows: Section 2 gives some background information on MDSD. Next, Section 3 describes our research method and discusses in detail the previously mentioned claims. Section 4 provides a description of our research experiments. The observations and results of the experiments are discussed in Section 5. Section 6 compares to related work. Finally, Section 7 concludes the paper.

2 Background: DSLs in Model-driven Software Development

In the initial phase of our study we decided to focus on the MDSD approach to develop DSLs for SOAs (details about the reasons can be found in Section 3). Before we go into details of the technical parts of our study, we want to briefly explain the background.

MDSD is based on the notion of DSLs or specification languages for modeling various types of models. DSLs are small languages that are tailored to be particularly expressive in a certain problem domain. The DSL describes knowledge via a graphical or textual syntax which is tied to domain-specific modeling elements through a precisely specified language model. That is, the DSL elements are defined in terms of the language model that can be instantiated in concrete application model instances. The application model instances are defined in the DSL's concrete syntax which represents the language model. The OMG's MDA proposal [17] is one specific MDSD approach that has some notable differences to our MDSD approach in general – especially in its sole focus on interoperability and platform independence.

An MDSD tool introduces some way to specify transformations. There are different kinds of transformations, such as model-to-model or model-to-code transformations. Also, different ways to specify transformations, such as transformation rules, imperative transformations, or template-based transformations, exist. In any case, the ultimate goal of all transformations in MDSD tools is to generate code in executable languages, such as programming languages or process execution languages. The MDSD tools are used to generate all those parts of the executable code which are schematic and recurring, and hence can be automated.

3 Research Method and Approach Overview

Many DSLs for specific aspects of SOAs have been designed (see for instance [16,12]), but to the best of our knowledge, no study provides evidence for specific aspects and claims associated to SOA DSLs. Hence, this research field is clearly of explorative nature. For this reason, we have decided to use an explorative, qualitative research method to get insights and evidences in this first study, following a similar approach to constructing a grounded theory [10,21]. Our plan is to use the results of this study in our future research to conduct more detailed qualitative and quantitative studies about specific aspects of our results.

In our case, the initial analysis has been performed by developing a number of DSLs in various projects (among others we considered those reported in [26,11]), as well as a thorough literature review and discussions with both experienced and new DSL developers. There are many ways to implement a DSL, such as using MDSD or extending a dynamic language (see [9] for details). Following our first results, we decided to investigate further on one specific kind of DSL development style: DSLs developed using MDSD (as proposed in [19,12]). We have decided for this style because, in our experience, the explicit support for language models is useful for representing the various concerns and stakeholders of a process-driven SOA. However, this decision limits the generalizability of the results of our study: not necessarily the results are applicable without modification for other styles of developing SOA DSLs.

After the initial investigation phase, we decided to conduct an in-depth study of specific claims associated to MDSD-based SOA DSLs using a controlled series of three prototyping experiments. In each experiment, we have developed a number of MDSD-based DSL prototypes, as well as a model-driven infrastructure to generate a running process-driven SOA from the models expressed in the DSLs. All three prototyping experiments have been conducted in a project that has run for 12 month and included 4 developers. Two developers worked with approximately 50% of their time for the full project duration, one contributed 20% of his time for the full duration, and one contributed approximately 50% of his time for 5 month. The project did not only include DSL development, but also development of other artifacts, such as models and transformations, needed to obtain a running prototype solution.

In particular, we investigated the following claims in-depth:

- Developing model-driven DSLs follows a systematic development approach [19,12].
- A process-driven SOA encompasses multiple concerns, such as orchestration of business processes, information in processes, collaboration between processes and services, data, transactions, human-computer interaction, service deployment, and many more. To express these concerns, it is claimed that using DSLs and language models reduces the complexity of the overall system, compared to a system developed without DSL/MDSD support [4].
- Using DSLs and language models for expressing SOA concerns enables developers and other stakeholders to work at a higher level of abstraction compared to using technical interfaces, such as programming APIs, executable process models expressed in BPEL code, or service interface descriptions such as WSDL (see

[25]). Hence, DSLs can be tailored by providing constructs that are common to the domain the different stakeholders work in [5]. This enhances the readability and understandability of each DSL for the different stakeholders. But, the different levels of abstractions imply the definition of integration points or transformations between the constructs of the DSLs from the different layers [6].
- Due to the different levels of abstraction, it is claimed that language models should provide clear extension points for integrating new concerns [20].

In our study, we performed three controlled experiments, in which a number of DSL prototypes have been developed:

- Realization of a number of DSLs for process-driven SOA basic concerns (basic concerns)
- Extension with additional DSLs for supporting long-running transactions and human participation (extensional concerns)
- Realization of DSLs for non-process-driven SOA concerns: extensions of process-driven SOAs with Web applications, especially Web UIs (external concerns)

Step-by-step we analyzed the various claims by reviewing and analyzing the design decisions made in our project. Within each experiment, we compared the different DSLs and their artifacts (such as DSL syntax, language models, transformations, and extension points) and used the results as input for our study. Also, the inputs led to refactoring of the DSLs in order to improve them. In addition, with each additional experiment stage, we compared the DSLs between the stages. That is, we followed a constant comparison method, as advocated by grounded theory approach [10,21], throughout our study. For comparison, we used different methods, such as expert reviews of our DSLs and models, student experiments with the models, and the application of the DSLs and models in industrial case studies.

4 Study Details

Figure 1 outlines a systematic development approach of an MDSD-based DSL architecture, as proposed in [15,19,20]. High-level and low-level DSLs represent appropriate language models. Language models can have multiple DSL syntaxes. Furthermore, language models can have multiple language model instances, which are defined using the DSL's syntax. High-level DSL syntaxes, language models, and model instances extend low-level DSL syntaxes, language models, and model instances respectively. Low-level DSLs provide constructs that are tailored for technical experts, whereas high-level DSLs are tailored for domain experts. A suitable separation of concerns can be established by splitting the language model into high- and low-level models, where the high-level model extends the low-level model. Hence, a separation of technical and domain concerns can be established to present only the appropriate concerns to each of the different groups of stakeholders.

In this approach, high-level concerns, relevant for non-technical stakeholders, are distinguished from low-level technical concerns to achieve better understandability for the different stakeholders. Due to the diverse backgrounds and knowledges of the different stakeholders, it makes sense to present to each group of stakeholders only the

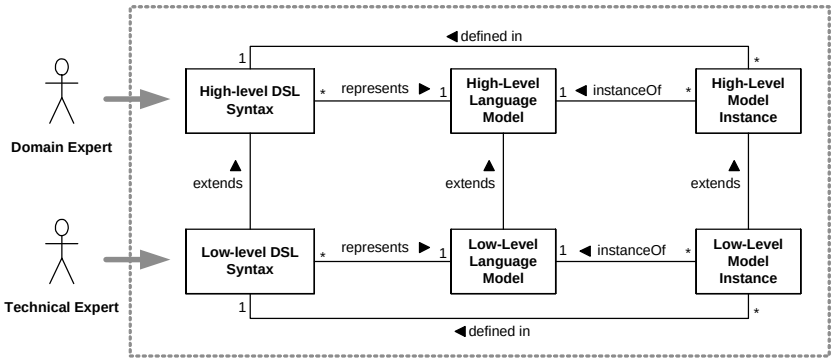

Figure 1. MDSD - DSLs

models they need for their work, and omit other details, as proposed in [25]. That is, high-level DSLs, designed with support for the domain experts, enable to work in a language in which domain concerns are depicted in or close to the domain's terminology. For instance, in the banking domain terms like account, bond, fund, or stock order are used in the high-level DSL. Low-level DSLs, in contrast, are utilized by technical experts to specify the technical details missing in the high-level DSLs. These details are needed by the model-driven code generator to turn the model instances, expressed in the DSLs, into a running system. For instance, in the process-driven SOA domain, relevant low-level concerns are service, service deployment, process variable, or database connection.

In the field of this study, we tried to provide evidence or counter-evidence for the claims summarized above. In particular, we evaluated the claims for three experiments. The experiments deal with process-driven SOAs, as well as an extension of process-driven SOAs with Web UIs. We focused in our experiments on the design decisions made and on the design trade-offs that have been considered. At first we will to describe the experiments in detail and after that in Section 5 our main results.

In the first experiment, the language models were designed together and at the same time. The extension points were specifically designed for integrating the language models. The organization of the language models is shown in Figure 2(1). A Core language model provides the extension points for modeling the basic concerns of process-driven SOAs, such as collaboration, controlflow, and information. During the second experiment, the extension points were used for introducing extensional concerns for which the extension points in the basic models have not originally been designed for. The language model structure of the second experiment is shown in Figure 2(2). In the third experiment, we investigated in how far external extensions, i.e., non-process-driven SOA concerns, can be integrated with the existing language models for process-driven SOAs. In particular, we integrated Web UIs with the process-driven SOA models. The organization of the Web UI's language models is depicted in Figure 2(3).

Process-driven SOA Basic Concern Language Models

The first experiment concentrates on basic concerns of process-driven SOAs, as well

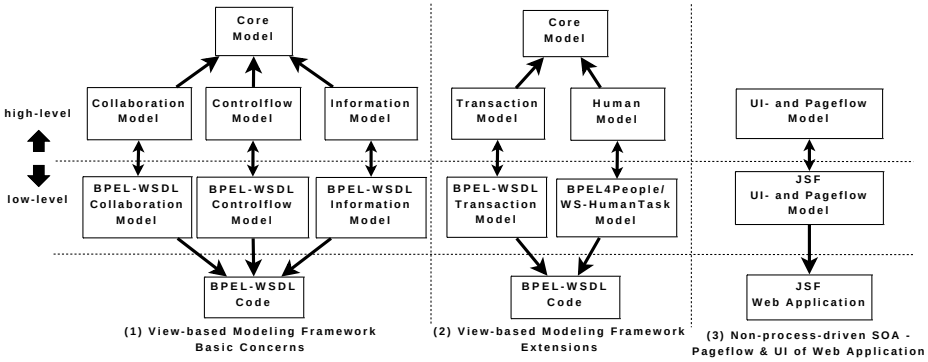

Figure 2. Experiments Overview

as providing high- and low-level DSLs for the different stakeholders [5]. The View-based Modeling Framework (VbMF) [23,24] is a model-driven framework for reducing the development complexity in process-driven SOAs, as well as to improve interoperability and reusability of models. It provides multiple language models, high-level and low-level, each responsible for a different concern of process-driven SOAs (i.e., controlflow, collaboration, and information). The structure of high-level and low-level language models is shown in Figure 2(1).

In this experiment, we used a systematic development approach as follows. First, high-level language models were designed. A central Core language model provides extension points for defining new language models for the appropriate concerns. Furthermore, it provides extension points for various language models for basic and extensional concerns. The following language models extend the Core language model for modeling basic concerns of process-driven SOAs:

- The **Controlflow** language model offers constructs for modeling controlflows of business processes, which consist of many activities and control structures. Activities are process tasks, e.g., service invocations or data handling. The execution order of activities is described through control structures, e.g., conditional switches.
- To compose the functionality provided by services or other processes, the **Collaboration** language model is used. This language model extends the Core language model to represent interactions between a business process and its partners.
- The **Information** language model represents the flow of data objects inside the business process. Furthermore, it provides a representation of message objects traveling back and forth between the process and the external world.

For each high-level language model, except the Core language model, the low-level language models were designed as an extension of the high-level language models. Both the high-level and low-level language models are close to the concepts of BPEL and WSDL. Finally, the DSL syntaxes were developed. The high-level DSL syntaxes are based on the constructs of the high-level language models, whereas low-level DSL syntaxes are based on the constructs of the low-level language models. Hence, domain

experts can – with the help of technical experts – use the high-level DSLs for modeling domain concerns, and technical experts can model technical concerns with the low-level DSLs.

Process-driven SOA Extensions of VbMF
In contrast to the first experiment, which analyzed the basic concerns of process-driven SOAs, this experiment uses the introduced extension points of the Core language model for integrating extensional concerns, such as transaction and human language models. The goal of this experiment is to figure out if the systematic development approach, used in the first experiment, can be applied for extensional concerns of process-driven SOAs. The structure of the high- and low-level language models for both experiments is shown in Figure 2(2).

To extend VbMF for long-running transactions, transactional concerns were integrated into the VbMF through a newly defined language model [23,24]. In the same way as the Controlflow, Collaboration, and Information language models were created, the first step was to design a high-level **Transaction** language model which extends the Core language model. Afterwards, a low-level Transaction language model was designed which extends the high-level Transaction language model. Like the low-level language models of the first experiment, the low-level Transaction language model is based on BPEL and WSDL concepts too. Finally, high-level and low-level DSLs were developed to support the modeling of transactions, based on the constructs of the appropriate language model.

A second extension of VbMF is the support of human participation in SOA-based business processes [22]. Again, a high-level **Human** language model was designed which extends the Core language model. Human aspects are assigned to processes and activities. A low-level Human language model extends the high-level Human language model, and it is based on concepts from BPEL4People [3] and WS-HumanTask [2]. Finally, high- and low-level DSLs were implemented, based on the appropriate language models, to support the modeling of human tasks for SOA-based business processes.

Non-Process-Driven SOA Extensions: Web User Interfaces
The third experiment followed again the systematic development approach, as adopted in the first two experiments. The language model hierarchy is depicted in Figure 2(3). The goal of this experiment is to figure out, if the systematic development approach can also be applied to extensions of process-driven SOAs with non-process-driven SOA concerns. The experiment deals with the modeling of Web UIs for Web pages, as well as process-oriented modeling of the pageflow through Java-like IF-ELSE statements. Web UIs contain the input and output components which are displayed to the user on Web pages. The pageflow provides the basis for selecting the subsequent Web page that should be displayed to the user, dependent on the current page and user interactions, e.g., which link or button is pressed by the user.

First, the high-level language model is introduced for modeling the pageflow and the UIs of the Web pages. A low-level language model for modeling the pageflow is introduced which is based on the pageflow definition of JavaServer Faces (JSF) [1] Web applications. The DSLs were implemented to provide suitable modeling of the pageflow and the UIs. The developed DSLs provide constructs that are very similar to

the language model. In this experiment, there is no need for a mapping between the constructs of the DSL and the constructs of the language model.

5 Study Results

In this section, we summarize the evidences and counter-evidences we found in our explorative experiments. First, the experiments provided some useful insights into design decisions required during the design of model-driven DSLs for process-driven SOAs:

- A design decision for the relation between DSL syntax and language model constructs must be made. We observed that in all three experiments the relationship between the names used in the DSL syntaxes and the names of the constructs defined in the language models was a concern. In all three experiments, we decided that the DSL syntaxes provide constructs that are named equivalently to the constructs in the language model. If the DSL syntax constructs are not named equivalently to the language model constructs, a more complex mapping between DSL and language model constructs is required, which means that extra efforts are required to develop this mapping. The mapping might also make the relationships between syntax constructs and models harder to understand. However, with a different naming in models and syntaxes, the syntax and modeling elements can be tailored more easily.
- In all three experiments, the low-level language models are extensions of the high-level language models. Hence, a relationship exists between them. A design decision must be made, in which order and dependency the high-level and low-level models are designed. The high-level language models can be designed first, and afterwards the low-level language models. Hence, domain concerns can be expressed close to their domain notions, such as compliance concerns in business processes. Another possible design approach is to derive the high-level language models from the low-level language models, which are based on technical concerns, e.g., constructs similar to BPEL (as done in our basic models). In this case, emphasis must be put on the high-level design of technical concerns, in order to make them understandable to domain experts, too. This is often not easy. Yet another approach is to design high- and low-level language models and DSLs in parallel. The main problem lies in the huge differences of the offered constructs between the languages. An example are languages like the Business Process Modeling Notation (BPMN) and BPEL. This approach requires a mapping between the often incompatible high-level and low-level language models, with possible inconsistencies. A part of this design decision is the development order of the high- and low-level language models and DSL syntaxes. If possible, the design of the high-level DSL syntax and language models should be performed together with the domain experts.
- In the first two experiments, which deal with basic and extensional concerns of process-driven SOAs, multiple language models where used. Multiple language models reduce the complexity by separation of concerns. This leads to providing tailored views for the different stakeholders. The main challenge of splitting lies in finding appropriate extension points for merging models. Poor extension points can lead to inconsistencies between the models. In addition, merging through extension

points is more complex than using modeling abstractions, such as associations. In the third experiment, one language model is used for modeling the pageflow and UIs of Web applications. Having only one language model does not provide a good separation of concerns for the development team and other stakeholders, but, on the other hand, there is no need for providing suitably designed extension and integration points, as well as possibly complex merging algorithms for the integration of multiple models. The design decision is whether it makes sense to split one language model into multiple models or not, and if splitting is chosen, where to split. Trade-offs for this design decision concern the number of concerns, development teams, and stakeholders.

Second, we found the following evidences for model-driven DSLs for claims associated to Section 3:

- It is possible to follow a systematic development approach, such as the one described in our three experiments in Section 4. In our case, this is not only valid for process-driven SOAs but also for non-process-driven SOA concerns, such as in our case Web applications.
- The systematic development approach used for the basic concerns of process-driven SOAs, such as controlflow, collaboration, or information of process-driven SOAs, can be followed for modeling extensional concerns, such as the transactional or human concerns in our experiments.
- Through a separation in high- and low-level DSLs, it is possible to support different stakeholders with different background and knowledge, i.e., domain experts and technical experts.
- Model-driven DSLs can enhance the understandability and readability for the individual stakeholders of a process-driven SOA. Furthermore, MDSD-based DSLs can reduce the complexity of process-driven SOAs.

Finally, the following counter-evidences, for the claims described in Section 3, should be considered as design trade-offs for the development of model-driven DSLs for SOAs:

- It is possible that the integration of high- and low-level concerns lead to DSL language design issues, such as redundancy in languages, inconsistencies, and which language should be chosen for overlapping concerns.
- Detailed separations of one language model into multiple ones can result in loose coupling of the different language models. Thus, the result is: the more detailed the separation, the more complex the model integration points for merging the different application models. Possible ways to achieve model integration are name-based matching, ontology-based matching, or inheritance. Hence, there is a trade-off between the complexity of the integration points and the degree of separation of concerns achieved in the language models.
- We observed another trade-off between model integration point design for the different stakeholders and the understandability, as well as the readability. The more complex the integration points are, the less understandable and readable the DSLs and/or their language models get in many cases. Hence, enhancing understandability and readability for one type of stakeholders increases the complexity of

integrating models for other stakeholders. That is, the complexity for stakeholders, who need to integrate and understand all models at once, can rise even though the complexity for individual stakeholders decreases.

6 Related Work

Pitkänen and Mikkonen [18] argue that well designed DSLs, modeling tools, and code generators increase the productivity. They concentrate on lightweight and modular DSLs instead of full-blown DSLs. Some situations of full-blown DSLs are described, e.g., several different implementation platforms. The lightweight approach can be an aid in defining the scope and concepts of DSLs before the implementation of a full-blown DSL starts. In comparison to our study, the systematic development approach can be applied to lightweight, as well as full-blown DSLs. The different design decisions and/or trade-offs, described in Section 5, are also valid for developing lightweight model-driven DSLs.

Bierhoff et al. [14] describe an incremental approach for developing DSLs. First, they choose an application and develop a DSL which is expressive enough to describe the application. Also, domain boundaries are defined. Then, the DSL grows until it is too expensive to extend it more. The approach is demonstrated on CRUD applications, i.e., create, retrieve, update, delete applications. The approach by Bierhoff et al. reflects the evolution of our three experiments described in Section 4. Also, we started by an initial experiment and extended it incrementally.

Maximilien et al. [16] developed a DSL for Web APIs and Services Mashups. A number of interesting design issues for DSLs are mentioned: (1) levels of abstraction, (2) terse code, (3) simple and natural syntax, and (4) code generation. These goals are very similar to our proposed claims. The developed DSL is used for SOAs, and the described approach and results are in line with our results.

Tolvanen [13] provides a guidance for defining and developing DSLs based on his long-year experiences in building DSLs. The development process is divided into four phases: (1) Identifying abstractions, (2) specifying the language models, (3) creating notations for the language based on the language models, and (4) defining model validators and code generators. The development phases are very similar to our observations. We started by defining abstractions of the domain, designed high- and low-level language models, developed a DSL with notations equivalent to the language models. Also, we provide model validators and code generators. The proposed approach by Tolvanen is similar to our systematic development approach for model-driven DSLs: (1) identifying the concepts of the domain and their relations, (2) designing the language models, (3) developing the DSLs based on the language models, and (4) generating code of valid domain models through a code generator.

7 Conclusion

The scope of this paper is to provide an initial study on a systematic development approach for SOA DSLs based on MDSD. It is likely, but not necessary, that many of our results are also valid for other implementation techniques for DSLs. We followed the

MDSD-based DSL approach quite closely. Hence, our results should be valid for a wide range of DSL tools.

We have addressed a broad range of process-driven SOA concerns (including basic, extensional, and external concerns). One result is that all of them can be expressed relatively easy using a distinct language model and integrated with existing language models using simple techniques such as inheritance or matching algorithms. However, it is possible that other process-driven SOA concerns exist, for which this is not easily possible.

Also, this paper discusses the design decisions and/or trade-offs we observed, as well as evidences and counter-evidences for the claims around model-driven DSLs for SOAs. Model-driven DSLs can enhance complexity, understandability, and readability for the individual stakeholders of SOAs. Therefore, DSLs can be tailored for domain experts and technical experts. But enhancing understandability and readability for domain experts, decreases understandability and readability for technical experts and vice versa.

During our study, we have used only a limited number of comparison and analysis techniques (such as code reviews, expert reviews, and student experiments). Other comparison or analysis methods might reveal properties that have not been revealed so far with our techniques used. Hence, we want to follow the systematic development approach in more studies and analyze the results.

Acknowledgement
This work was supported by the European Union FP7 project COMPAS, grant no. 215175.

References

1. JavaServer Faces Technology. http://java.sun.com/j2ee/javaserverfaces.
2. A. Agrawal, M. Amend, M. Das, C. Keller, M. Kloppmann, D. König, F. Leymann, R. Müller, G. Pfau, K. Ploesser, R. Rangaswamy, A. Rickayzen, M. Rowley, P. Schmidt, I. Trickovic, A. Yiu, and M. Zeller. Web Services Human Task (WS-HumanTask), version 1.0, 2007.
3. A. Agrawal, M. Amend, M. Das, C. Keller, M. Kloppmann, D. König, F. Leymann, R. Müller, G. Pfau, K. Ploesser, R. Rangaswamy, A. Rickayzen, M. Rowley, P. Schmidt, I. Trickovic, A. Yiu, and M. Zeller. WS-BPEL extension for people (BPEL4People), version 1.0, 2007.
4. Anton Jansen and Jan Bosch. Software Architecture as a Set of Architectural Design Decisions. In *WICSA '05: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, pages 109–120, Washington, DC, USA, 2005. IEEE Computer Society.
5. Arno Schmidmeier. Aspect oriented DSLs for business process implementation. In *DSAL '07: Proceedings of the 2nd workshop on Domain specific aspect languages*, page 5, New York, NY, USA, 2007. ACM.
6. Arturo J. Sánchez-Ruíz, Motoshi Saeki, Benoît Langlois, Roberto Paiano. Domain-Specific Software Development Terminology: Do We All Speak the Same Language? http://www.dsmforum.org/events/DSM07/papers/sanchez-ruiz.pdf.
7. D. K. Barry. *Web Services and Service-oriented Architectures*. Morgan Kaufmann Publishers, San Francisco, CA, 2003.

8. M. Fowler. Language workbenches and model driven architecture. http://www.martinfowler.com/articles/mdaLanguageWorkbench.html, June 2005.
9. M. Fowler. Language workbenches: The killer-app for domain specific languages? http://www.martinfowler.com/articles/languageWorkbench.html, June 2005.
10. B. Glaser and A. Strauss. *The discovery of grounded theory*. Aldin, New York, 1967.
11. M. Goedicke, K. Koellmann, and U. Zdun. Designing runtime variation points in product line architectures: three cases. *Science of Computer Programming*, 53(3):353–380, 2004.
12. J. Greenfield and K. Short. *Software Factories: Assembling Applications with Patterns, Frameworks, Models & Tools*. J. Wiley and Sons Ltd., 2004.
13. Juha-Pekka Tolvanen. Domain-Specific Modeling: How to Start Defining Your Own Language. http://www.devx.com/enterprise/Article/30550 (last accessed: July 2008).
14. Kevin Bierhoff and Edy S. Liongosari and Kishore S. Swaminathan. Incremental Development of a Domain-Specific Language That Supports Multiple Application Styles. In *OOPSLA 6th Workshop on Domain Specific Modeling*, pages 67–78, October 2006.
15. Luoma, J., Kelly, S., Tolvanen, J.-P. Defining Domain-Specific Modeling Languages: Collected Experiences. *Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling (DSM04), Vancouver, British Columbia, Canada*, 2004.
16. E. M. Maximilien, H. Wilkinson, N. Desai, , and S. Tai. A domain specific-language for web apis and services mashups. In *Proceedings of 5th International Conference on Service Oriented Computing (ICSOC), LNCS 4749, Springer-Verlag*, pages 13–26, Vienna, Austria, 2007.
17. OMG. MDA Guide Version 1.0.1. Technical report, Object Management Group, 2003.
18. Risto Pitkänen and Tommi Mikkonen. Lightweight Domain-Specific Modeling and Model-Driven Development. In *OOPSLA 6th Workshop on Domain Specific Modeling*, pages 159–168, October 2006.
19. T. Stahl and M. Voelter. *Model-Driven Software Development*. J. Wiley and Sons Ltd., 2006.
20. Steve Cook. Domain-Specific Modeling and Model Driven Architectures. http://www.bptrends.com, 2004.
21. A. Strauss and J. Corbin. *Grounded theory in practice*. Sage, London, 1997.
22. Ta'id Holmes and Huy Tran and Uwe Zdun and Schahram Dustdar. Modeling Human Aspects of Business Processes - A View-Based, Model-Driven Approach. In *ECMDA-FA*, pages 246–261, 2008.
23. H. Tran, U. Zdun, and S. Dustdar. View-based and model-driven approach for reducing the development complexity in process-driven SOA. In W. Abramowicz and L. A. Maciaszek, editors, *BPSC*, volume 116 of *LNI*, pages 105–124. GI, 2007.
24. H. Tran, U. Zdun, and S. Dustdar. View-based integration of process-driven soa models at various abstraction levels. In *R.-D. Kutsche and N. Milanovic, Editors, Proceedings of First International Workshop on Model-Based Software and Data Integration MBSDI 2008*, pages 55–66. Springer, April 2008.
25. Vito Perrone and Davide Bolchini and Paolo Paolini. A Stakeholders Centered Approach for Conceptual Modeling of Communication-Intensive Applications. In *SIGDOC '05: Proceedings of the 23rd annual international conference on Design of communication*, pages 25–33, New York, NY, USA, 2005. ACM.
26. U. Zdun. Tailorable language for behavioral composition and configuration of software components. *Computer Languages, Systems and Structures: An International Journal*, 32(1):56–82, 2006.
27. U. Zdun, C. Hentrich, and W. van der Aalst. A survey of patterns for service-oriented architectures. *International Journal of Internet Protocol Technology*, 1(3):132–143, 2006.