Realizing Changes by Aspects at the Design Level

Valentino Vranić and Branislav Kuliha
Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
Ilkovičova 2, Bratislava, Slovakia
vranic@stuba.sk, kobliha@centrum.sk

***Abstract*—The cost of a change is high, but changes are an inevitable part of software development lifecycle, which comes to be recognized under a more general term: software evolution. To mitigate this problem, an approach to aspect-oriented change realization has been proposed earlier based on the idea of representing change by aspect. In many cases, software development relies on graphical modeling, mainly UML, and thus a legitimate question is how aspect-oriented change realization could be supported at the modeling level. This paper proposes an approach to achieve this based on Theme, a comprehensive approach to aspect-oriented analysis and design. One of the results of the work reported here is a catalog of change type models for the domain of web applications comprising the models of eleven specification change types and seven implementation change types. Apart from the examples presented in the paper, the approach was successfully applied to a real web mail system. As no dedicated Theme modeling tool is available, a UML profile for both analytical (Theme/Doc) and design part (Theme/UML) of the Theme approach has been designed and implemented in IBM Rational Software Architect.**

***Index Terms*—software evolution; change realization; aspect-oriented programming; software modeling; Theme; UML; parameterized types**

I. Introduction

The cost of a change is high, but changes are an inevitable part of software development lifecycle, which comes to be recognized under a more general term: software evolution. To mitigate this problem, an approach to aspect-oriented change realization has been proposed earlier [1]–[4] based on the idea of representing change by aspect [5]. The idea also appeared elsewhere independently [6], [7].

Aspect-oriented programming appears here as a logical choice due to its capability to affect code in a declarative manner. Plugging in and unplugging changes, or even their transfer to other programs, suddenly becomes possible. In many cases, software development relies on graphical modeling, mainly UML, and thus a legitimate question is how aspect-oriented change realization could be supported at the modeling level. This paper proposes an approach to achieve this based on Theme, a comprehensive approach to aspect-oriented analysis and design.

The rest of the paper is organized as follows. Section II presents the background of the aspect-oriented change realization approach. Section III introduces the position of the Theme approach to aspect-oriented analysis and design within the work presented here. Section IV explains the approach to modeling specification change types using Theme/Doc. Section V explains the approach to modeling implementation change types using Theme/UML. Section VI presents the evaluation results. Section VII discusses related work. Section VIII concludes the paper.

II. Aspect-Oriented Change Realization

Aspect-oriented change realization [1]–[4] is an approach to treating changes as separate concerns. Rooted in the early idea of representing changes as aspects relying on aspects' intrinsic capability of affecting other code without having to change this code as such [5], the approach has developed to incorporate reasoning about changes at two levels: specification and implementation (called domain specific and generally applicable in the prior work).

Although the idea is applicable in other contexts, probably the best motivating example is so-called software customization. In customization, a general application is being adapted to the client's needs by a series of changes [3]. With each new version of the base application, all the changes have to be reapplied to it. If it would be possible to maintain changes as separate modules, they could be either directly applied to the new main version or this could be done with some adaptation (see Figure 1).

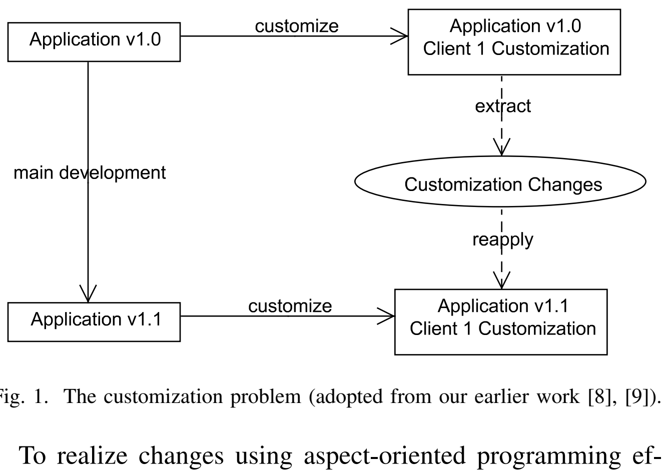


Fig. 1. The customization problem (adopted from our earlier work [8], [9]).

To realize changes using aspect-oriented programming effectively, a two-level aspect-oriented change realization model has been proposed [1], [3]. Consider a change request containing the requirement to add a backup SMTP server to ensure delivery of the notifications to users as a part of the customization of the affiliate marketing software [3], [8]. Technically speaking, each time the affiliate marketing software needs to send a notification, it creates an instance of the corresponding class which handles the connection to the SMTP server. In more general terms, an SMTP server is a kind of a resource that needs to be backed up. Abstracting from the context (SMTP server), this may be identified as a kind of the Introducing Resource Backup specification change type. There is an effective way of implementing this type of change expressed as an implementation change type called Class Exchange. Each implementation type is accompanied with a code scheme that needs just to be adapted to the context. A catalog of change types that have been identified in the web application domain including the correspondence between them is available [3].

III. Employing the Theme Approach

Many software development projects employ graphical software modeling, mainly UML. With respect to this, there is a need to enable aspect-oriented change realization at the modeling level. The Theme approach [10] is an aspect-oriented modeling approach based around the notion of *theme* that represents a general perception of a concern. The Theme approach works with themes from the very start by employing a sort of their automatic identification in requirements and expressing them in a form of a network of relationships, ending with the UML-based design of individual themes with a clear transition to implementation. Here, we are interested in applying the two notations the Theme approach brings—Theme/Doc and Theme/UML—whose relevant details will be explained as needed in the following sections.

The general idea of the approach proposed here is to use Theme/Doc to express specification change types on one side, and to use Theme/UML to express implementation change types on the other side. These will form a catalog of changes in much the same way as with the original, code based aspect-oriented change realization.

Applying the catalog assumes comparing the application model to the cataloged specification change types. Consequently, it appears that the model to be compared has to be a Theme/Doc model. However, this is somewhat relaxed since a Theme/Doc model can be transformed into the graphical part of a use case model [11], and use cases are widely used in modern software development. In fact, as is going to be explained later, we employed a UML CASE tool and the Theme/Doc themes were actually represented as stereotyped use cases. For the purposes of this paper, we will stick to the assumption of the pre-existence of the Theme/Doc model of the target application.

The realization of a change assumes applying the implementation change type that came out of the Theme/Doc based change analysis, which is analogical to the code based aspect-oriented change realization example of which was given in the previous section. Logically, the best way to go is to have the target application designed in Theme/UML. As in reality this is not going to be very probable, the fastest way to obtain a Theme/UML model from a common UML model is to consider the whole target application UML model as one theme. Everything else would require some form of refactoring to partition the model into themes, and this is beyond the scope of this paper.

IV. Modeling Specification Change Types

Given a change request, it is necessary to decompose it into one or more actual changes. Suppose we have a simple web mail system embracing user registration, sending and receiving e-mail, and necessary administration functionality. Consider a change request that states that the administrator should be able to block and unblock account from the accounts view (CHR03). This change request actually consists of two changes. One change follows directly from the change request: a new column has to be added to the list of accounts. This column should contain a control element to call appropriate service to block or unblock selected user's account. Another purpose of this column is to show the user's account status. The other change is implicit: check whether the user is blocked during the logging in since blocked users mustn't be allowed to log in.

We represent these two changes formally as two requirements, as can be seen in Figures 2 and 3. As no dedicated Theme modeling tool is available, a UML profile for both analytical (Theme/Doc) and design part (Theme/UML) of the Theme approach has been designed and implemented in IBM Rational Software Architect (IBM RSA). A requirement is there represented as a stereotyped UML artifact.

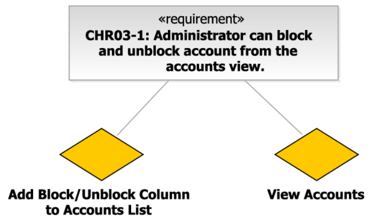

«requirement»
CHR03-1: Administrator can block and unblock account from the accounts view.
Add Block/Unblock Column to Accounts List
View Accounts

Fig. 2. Add Block/Unblock Column to Accounts List: the theme–relationship view.

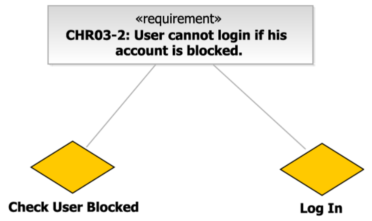


Fig. 3. Check User Blocked: the theme–relationship view.

Next, we identify concerns, i.e., themes, implied by each requirement. Themes, displayed as rhomboids, are in our IBM RSA models represented as stereotyped use cases (as has been already mentioned in the previous section). They are connected to requirements by associations. How to identify relevant themes from requirements goes beyond this paper, but a simplified approach is to extract all concerns that may stand on their own from the requirement. In this, it is a good idea to go beyond the requirement's description towards clarification in direct contact with those who posed it.

Speaking of changes and themes that can be expected there, there are themes of two kinds: some represent the target of a change or changes and some represent the changes themselves. As we are working with focused changes gained by a decomposition of a change request, the expected case is to have one theme of each kind per change.

In CHR03-1 (Figure 2), we have identified two themes: *Add Block/Unblock Column to Accounts List* and *View Accounts*. The same is done with the second change, CHR03-2 (Figure 3). In this case, the model denotes the requirement to check if the user is blocked triggered by the behavior responsible for logging the user in.

In the Theme vocabulary, both CHR03-1 CHR03-2 represent shared requirements. Such requirements have to be resolved preferably—and in our approach necessarily—by declaring one of the themes as crosscutting and making it responsible for introducing the requirement into other themes. In our approach, the crosscutting theme is always the one that represents a change as such. You can see both our changes in a so-called crosscutting–relationship view in Figures 4 and 5. In terms of our IBM RSA UML profile, the crosscutting edges are actually stereotyped dependencies.

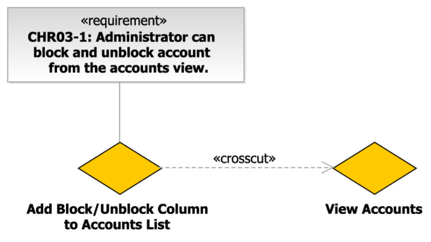

«requirement»
CHR03-1: Administrator can block and unblock account from the accounts view.
«crosscut»
Add Block/Unblock Column to Accounts List
View Accounts

Fig. 4. Add Block/Unblock Column to Accounts List: the crosscutting view.

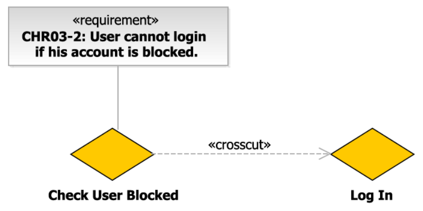


Fig. 5. Check User Blocked: the crosscutting view.

Now that we have identified and analyzed the specification changes that the change request contained, we need to figure out how to design them. To do this, we first need to determine the types of the changes. This is the moment when we can take the advantage and make use of the catalog of change types, and apply the two-level change model. When browsing through the catalog, we look for similarities between our specification changes and the specification change types within the catalog. We rely on knowing the actual implementation or design model. For example, we know that our account list is implemented as a grid. With respect to this, we determine that *Add Block/Unblock Column to Accounts List* is of the *Add Column to Grid* type. Figure 6 compares this change to our change type. The *Show Grid* theme in the specification change type recalls our *View Accounts* theme, but this is nothing but a lucky coincidence and in other cases the names of the themes don't have to be related. Nevertheless, the thing that must match is the behavior that these two themes describe, which is the behavior of showing the grid that is to be crosscut.

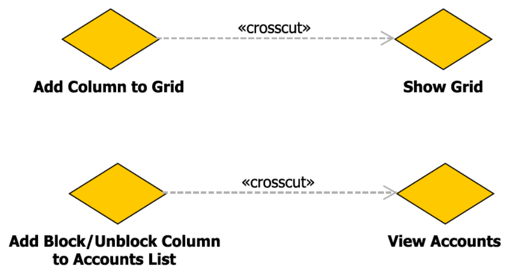


Fig. 6. Comparing *Add Block/Unblock Column to Accounts List* and the *Add Column to Grid* change type: specification change type identification.

We identify a specification change type for the second change in a similar manner. This is not straightforward as the previous case. We can look at blocking user's logging in two ways: as the *Introduce User Rights Management* or as *Introduce Additional Constraint on Fields* change type. The operation of logging in consists of the form validation and submission. If the user name and password are correct, the form validation is correct. The triggering behavior can be the validation, as well as the submission itself. The approach leads towards viable solutions and it is up to the developer to decide what is more suitable and more simple to design and implement. The choice also depends on the application environment and the technologies used. In our example, we decided for *Introduce Additional Constraint on Fields*. Figure 7 shows the identification of a change with this specification change type.

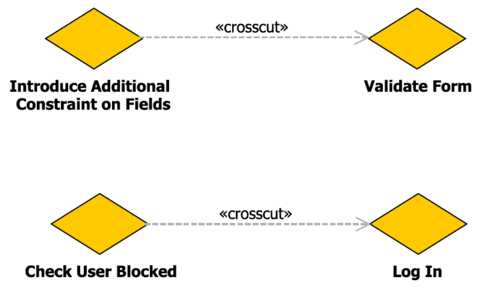


Fig. 7. Comparing *Add Block/Unblock Column to Accounts List* and the *Check User Blocked* change type: specification change type identification.

V. Modeling Implementation Change Types

As with the code based aspect-oriented change realization, here we also maintain a catalog, albeit in a graphical form. Each specification change type is connected to the implementation change type by a trace dependency. Sometimes there are multiple possibilities to choose from for the design. The choice should not be difficult to make because the design is closer to the implementation environment and we should now know whether the design is feasible.

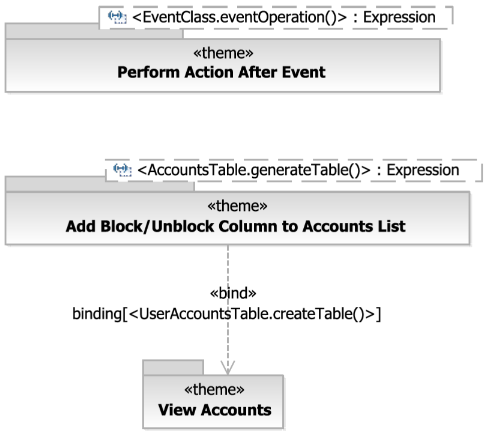
When designing the changes, we model the triggered behavior while using the scheme of the identified implementation change type. This doesn't mean that we copy-paste the sequence diagram and rename some elements. The scheme is only to provide directions on the structure and behavior similarly as design patterns do.

The *Add Block/Unblock Column to Accounts List* theme design is shown in Figure 8. In terms of Theme/UML, this is an aspect theme and all changes in our approach are supposed to eventually end up as aspect themes. This is in accordance with what is popularly known as aspect-oriented programming, where aspects affect the base code in some manner. In the aspect-oriented software development research this is actually only a part of a more general view of aspect-orientation that comprises also approaches to modularizing concerns (including the crosscutting ones) into a set of elements that stand on equal basis. This is not just an academic endeavor as properties of this so-called symmetric aspect-oriented programming are reflected in contemporary programming languages [12]; however, this is out of the scope of this paper.

In Theme/UML, aspect themes are represented as parameterized packages comprising structural (class diagram) and behavioral models (typically sequence or activity diagrams). The first parameter is by convention the one that triggers the behavior. In case of our *Add Block/Unblock Column to Accounts List* theme, the trigger is the AcountsTable.generateTable() operation. Parameters do not represent the actual application elements: they just stand for these elements to be supplied by the bind relationship.

In order for anything to happen, the sequence diagram has to match the behavior sequence in the target (base) theme. The point is in capturing certain behavior, which is practically an operation call, and performing some other behavior before, after, or instead of it, including repeatedly or conditionally executing the captured behavior. To distinguish the capturing of this behavior from the behavior itself, the _do_ prefix is used. Thus, a call to an operation with the _do_ prefix means the actual call ending in executing the corresponding behavior. Here, it may be observed that sequence diagrams in Theme/UML combine both pointcuts and advice, as these elements are known in the prevailing AspectJ style of aspect-oriented programming.

Getting back to our example, what happens is that after the table has been generated, an additional column for blocking users is created. Generally speaking, we performed an action after event. This corresponds to the implementation change type that realizes the *Introduce Additional Constraint on Fields* specification change type: *Perform Action After Event*. Note that we can add some other actions after the table generation and it will still be of the same type. Figure 9 shows binding the change with the *View Accounts* theme. Here we specify the real replacement for the template parameters.



<EventClass.eventOperation()> : Expression
«theme»
Perform Action After Event
<AccountsTable.generateTable()> : Expression
«theme»
Add Block/Unblock Column to Accounts List
«bind»
binding[<UserAccountsTable.createTable()>]
«theme»
View Accounts

Fig. 9. Add Block/Unblock Column to Accounts List: the binding.

The processing of the second change is similar. We design the change with the help of *Additional Parameter Check*, which is the implementation change type tracing the *Introduce Additional Constraint on Fields* specification change type. Just like with the first change, we produce a sequence diagram of the triggered behavior (Figure 10). Here, the trigger is the UserLoginForm.validateForm() template parameter operation. Prior to the form validation, the checkUserBlocked() operation is called. In this operation, the user's status is determined with respect to blocking and the form's validation status is set accordingly.

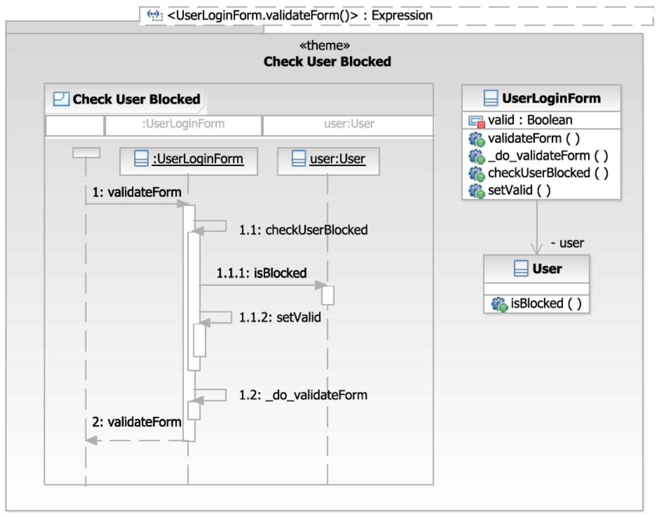


Fig. 10. Check User Blocked: the theme design.

After this, the original validation process can be performed. As an alternative to setting the form's valid attribute false, a validation exception can be drawn to prevent the form from being submitted. The binding of the corresponding change theme is depicted in Figure 11. Note that the actual parameter bears the same name as the formal parameter. Theme/UML allows for this, though it may be confusing.

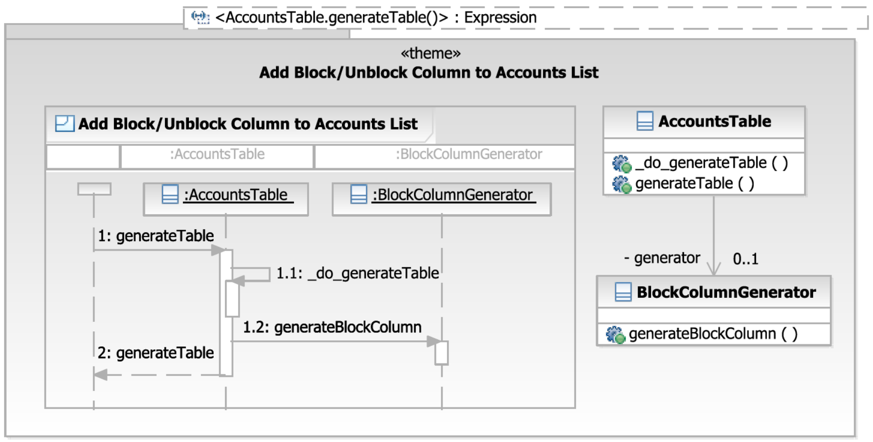
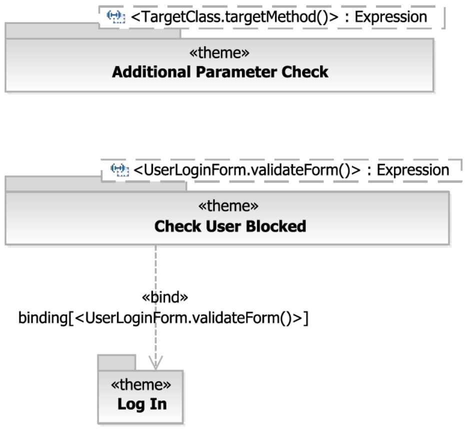


Fig. 8. Add Block/Unblock Column to Accounts List: the theme design.



<TargetClass.targetMethod()> : Expression
«theme»
Additional Parameter Check
<UserLoginForm.validateForm()> : Expression
«theme»
Check User Blocked
«bind»
binding[<UserLoginForm.validateForm()>]
«theme»
Log In

Fig. 11. Check User Blocked: the binding.

VI. Evaluation

Apart from the preliminary examples of the proposed approach to modeling aspect-oriented change realizations some of which are introduced in this paper, the evaluation was conducted on a real web mail system. We focused on the capability and usability of the modeled change types the catalog we developed comprising all the specification and implementation change types identified in the earlier work on code based aspect-oriented change realization (the domain specific change type is introduced first) [3]:

- One Way Integration: Performing Action After Event
- Two Way Integration: Performing Action After Event
- Adding Column to Grid: Performing Action After Event
- Removing Column from Grid: Method Substitution
- Altering Column Presentation in Grid: Method Substitution
- Adding Fields to Form: Enumeration Modification with Additional Return Value Checking/Modification
- Removing Fields from Form: Additional Return Value Checking/Modification
- Introducing Additional Constraint on Fields: Additional Parameter Checking or Performing Action After Event
- Introducing User Rights Management: Border Control with Method Substitution
- User Interface Restriction: Additional Return Value Checking/Modifications
- Introducing Resource Backup: Class Exchange

Two change requests for the web mail system were studied, analyzed, designed, and implemented. The catalog of change type models showed to be useful and sufficiently correct. Some problems appeared mainly in the process of implementation caused by the actual frameworks and technologies used. Therefore, the catalog of change type models should be somehow aware also of the variability in implementation technologies. The implementation of the changes confirmed the correctness of the design to implementation transformation process guidelines of the Theme approach [10].

While evaluating the catalog of change type models, we also found that there should be some feedback from the developers using the catalog to help its improvement and extension. Software developers should track deviations from the cataloged change type models. This is valuable for future development in the same domain. For example, if developers find a new way to design a specification change type or identify a new implementation change type, they should analyze it and propose it for incorporating into the corresponding catalog. The specification and implementation change types in the catalog are more or less dependent on the actual application design and on characteristics of the application domain.

We managed to almost fully rely on the catalog. However, the catalog is there not to give directives, but to provide recommendations.

It is difficult to compare the effectiveness of this approach or to quantify the effort of the change realization. However, within the common, non aspect-oriented modeling, we can observe that the larger the change is, more diagrams we have to edit. In our approach, the change is focused in one diagram and the original design is not being affected at all.

VII. Related Work

Join point designation diagrams [13], [14] are another aspect-oriented modeling technique aiming specifically at expressing pointcuts in a graphical way. Preliminary findings [15] indicate they are complimentary to Theme/UML improving its ability to specify crosscutting and thus could be of much help in improving the approach proposed here.

The Distributed Systems Group at the University of Dublin created a tool for composition with transformation using OpenArchitectureWare implemented as an Eclipse plugin [16]. Their tool supports Theme/UML composition based on the XMI format, which can be applied to weave the aspect changes into the target model to obtain it as a whole, common UML model.

Object-oriented role analysis and modeling (OOram) [17], [18], whose ideas have been revived in the DCI (Data, Context and Interaction) approach [19], initially announced as DCA (Data–Context–Algorithm) [18], enables a general separation of concerns. Each concern is modeled within its own model and these models are composed to achieve a complete behavior. This recalls themes and their composition making the OOram notation a potential alternative to the Theme/UML notation we used. Composite structure modeling in UML, which employs the concept of role, too, is worth exploring for its potential to mimic OOram's notation within UML.

VIII. Conclusions and Further Work

In this paper, an approach to modeling aspect-oriented change realizations is proposed. The approach relies on our earlier work on that resulted in a two-level aspect-oriented change realization model [1], [3]. Regarding the modeling notation, the approach is based on Theme [10], a comprehensive approach to general aspect-oriented modeling.

One of the results of the work reported here is a catalog of change type models for the domain of web applications comprising the models of eleven specification change types and seven implementation change types. Apart from the examples presented in the paper, the approach was successfully applied to a real web mail system. As no dedicated Theme modeling tool is available, a UML profile for both analytical (Theme/Doc) and design part (Theme/UML) of the Theme approach has been designed and implemented in IBM Rational Software Architect.

Further work should aim at incorporating regular developers' contributions to the catalog and change type diversity management. Also, it is important to explore the possibilities of adjusting the modeling notation to be as close as possible to what standard UML modeling tools support. With respect to this, switching from the Theme/Doc notation to use case diagrams appears quite appealing [11].

Acknowledgments

The work reported here was supported by the Scientific Grant Agency of Slovak Republic (VEGA) under the grant No. VG 1/1221/12. This contribution/publication is also a partial result of the Research & Development Operational Programme for the project Research of Methods for Acquisition, Analysis and Personalized Conveying of Information and Knowledge, ITMS 26240220039, co-funded by the ERDF.

References

[1] M. Bebjak, V. Vranić, and P. Dolog, "Evolution of web applications with aspect-oriented design patterns," in *Proceedings of ICWE 2007 Workshops, 2nd International Workshop on Adaptation and Evolution in Web Systems Engineering, AEWSE 2007, in conjunction with ICWE 2007*, Como, Italy, Jul. 2007, pp. 80–86.

[2] V. Vranić, R. Menkyna, M. Bebjak, and P. Dolog, "Aspect-oriented change realizations and their interaction," *e-Informatica Software Engineering Journal*, vol. 1, no. 3, pp. 43–58, 2009.

[3] V. Vranić, M. Bebjak, R. Menkyna, and P. Dolog, "Developing applications with aspect-oriented change realization," in *Proceedings of 3rd IFIP TC2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2008, Revised Selected Papers*, ser. LNCS 4980. Brno, Czech Republic: Springer, 2011.

[4] R. Menkyna and V. Vranić, "Aspect-oriented change realization based on multi-paradigm design with feature modeling," in *Proceedings of 4th IFIP TC2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2009, Revised Selected Papers*, ser. LNCS 7054. Krakow, Poland: Springer, 2012.

[5] P. Dolog, V. Vranić, and M. Bieliková, "Representing change by aspect," *ACM SIGPLAN Notices*, vol. 36, no. 12, pp. 77–83, Dec. 2001.

[6] I. Bluemke and K. Billewicz, "Aspect modification of an EAR application," in *Proceedings of International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering, CIS^{2}E 08*. Krakow, Poland: Springer, Dec. 2008, to appear.

[7] ——, "Aspects in the maintenance of complied program," in *Proceedings of 5th International Conference on Dependability of Computer Systems, DepCoS-RELCOMEX 2008*. Szklarska Poręba, Poland: IEEE, Jun. 2008, pp. 253–260.

[8] V. Vranić, "Aspect-oriented change realization: Approach, design patterns, and beyond," 2008, series of lectures at Lancaster University, UK. http://fiit.stuba.sk/~vranic/pub/lu/abstract.html.

[9] ——, "Aspect-oriented change realization," Habilitation thesis (submitted in fulfillment of the requirements for the Associate Professor degree), Slovak University of Technology in Bratislava, Slovakia, 2011.

[10] S. Clarke and E. Baniassad, *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley, 2005.

[11] V. Vranić and P. Michalco, "Are themes and use cases the same?" *Information Sciences and Technologies, Bulletin of the ACM Slovakia*, vol. 2, no. 1, pp. 66–71, 2010, special Section on Early Aspects at AOSD 2010.

[12] J. Bálik and V. Vranić, "Symmetric aspect-orientation: Some practical consequences," in *Proceedings of NEMARA 2012: International Workshop on Next Generation Modularity Approaches for Requirements and Architecture, at AOSD 2012*. Potsdam, Germany: ACM, Mar. 2012.

[13] D. Stein, S. Hanenberg, and R. Unland, "Query models," in *Proceedings of 7th International Conference on the Unified Modeling Language, UML 2004*, ser. LNCS 3273. Lisbon, Portugal: Springer, Oct. 2004.

[14] D. Stein, P. Sánchez, S. Hanenberg, L. Fuentes, and R. Unland, "Facilitating the exploration of join point selection," in *Aspect-Oriented Models Models in Software Engineering, Workshops and Symposia at MoDELS 2010*, ser. LNCS 6627. Olso, Norway: Springer, 2011.

[15] A. Jackson and S. Clarke, "Towards the integration of Theme/UML and JPDDs," in *8th International Workshop on Aspect-Oriented Modeling, held in conjunction with 5th International Conference on Aspect-Oriented Software Development, AOSD'06*, Bonn, Germany, Mar. 2006.

[16] Distributed Systems Group, Trinity College Dublin, University of Dublin, "Theme/uml," http://www.dsg.cs.tcd.ie/aspects/themeUML.

[17] T. Reenskaug, P. Wold, and O. A. Lehne, *Working With Objects: The OOram Software Engineering Method*. Prentice Hall, 1996.

[18] T. Reenskaug, "Programming with roles and classes: The BabyUML approach," in *Computer Software Engineering Research*. Nova Publishers, 2007.

[19] J. Coplien and G. Bjørnvig, *Lean Architecture: for Agile Software Development*. Wiley, 2010.