World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:2, No:3, 2008

Representing Shared Join Points with State Charts: A High Level Design Approach

Muhammad Naveed, Muhammad Khalid Abdullah, Khalid Rashid, and Hafiz Farooq Ahmad

***Abstract*—**Aspect Oriented Programming promises many advantages at programming level by incorporating the cross cutting concerns into separate units, called aspects. Join Points are distinguishing features of Aspect Oriented Programming as they define the points where core requirements and crosscutting concerns are (inter)connected. Currently, there is a problem of multiple aspects' composition at the same join point, which introduces the issues like ordering and controlling of these superimposed aspects. Dynamic strategies are required to handle these issues as early as possible. State chart is an effective modeling tool to capture dynamic behavior at high level design. This paper provides methodology to formulate the strategies for multiple aspect composition at high level, which helps to better implement these strategies at coding level. It also highlights the need of designing shared join point at high level, by providing the solutions of these issues using state chart diagrams in UML 2.0. High level design representation of shared join points also helps to implement the designed strategy in systematic way.

***Keywords*—**Aspect Oriented Software Development, Shared Join Points.

I. Introduction

ASPECT Oriented Programming [1] [2] is a new software development paradigm which enables to increase the comprehensibility, adaptability and reusability by modularizing the crosscutting concerns into the units called "aspects" [3] [4]. It provides solutions of many real time problems that neither the object oriented nor procedural languages can sufficiently handle [2] [5]. "Aspect" in AOP is like a class entity which mainly differs in instantiation and inheritance [3]. Other constructs of AOP are join points, pointcuts, advices and introductions [2] [5], among all, join point is more important. Join point is defined as a well defined execution point in a program [3].

Join points represent the key concept in Aspect-Oriented Software Development (AOSD). Join points define the places where two concerns i.e. core and aspectual, crosscut each other [2] [3] [5] [6] [7] [8] [9]. Main task of aspect-oriented introductions [2] [5], among all, join point is more important. Join point is defined as a well defined execution point in a program [3].

Join points define the places where two concerns i.e. core and aspectual, crosscut each other [2] [3] [5] [6] [7] [8] [9]. Main task of aspect-oriented designers is to identify set(s) of join points, where two concerns interconnect to each other, and provide suitable representation for join points [8] [9] [10].

In many cases, a join point is superimposed by multiple aspects at the same time, known as a shared join point [5] [8] [10]. There are many example scenarios (one discussed in section 4), where multiple aspects are being superimposed on the same join point [8] [10] [11]. Currently, there are problems with shared join points at implementation level due to uncertain execution behavior of superimposed aspects [8] [10] [11]. Since multiple aspects are being superimposed, it becomes difficult to judge what will be the exact execution order? If an aspect does not work; how to control the execution order of other aspects? There is not sufficient support available for these issues at implementation level, but there are some indirect support and recommendation details for AOP languages [5] [8] [11]. For example, AspectJ provides precedence construct for ordering and do not provide any direct support for controlling [5].

These issues are novel and being discussed at implementation level only [8] [10]. These issues require the strategies to order and control the superimposed aspects at run time. The strategy presented at implementation level requires to be modeled for shared join points at the early software development stage [12] [13]. Due to the significance of join points, particularly, the shared join points, the representation of issues regarding the shared join points in an aspect-oriented development environment is a major task for aspect-oriented designers at high level design. This high level representation can reduce the work cost by the formulation of early design decisions.

Software design is an important activity in the software development. It is like a blueprint of the software to be built [14].Recently, Aspect Oriented Software Development is making strong progress on the implementation level, but the extensive support at design level is still insufficient [13].

This work is a part of our research project for Department of Computer Science at International Islamic University, Islamabad, Pakistan.

Muhammad Naveed is a student of MS (Software Engineering) at International Islamic University Islamabad, Pakistan. Also working as a Software Engineer in ESOLPK, 12 SNC Center, Blue Area, Islamabad, Pakistan (e-mail: naveed_2334@yahoo.com).

Muhammad Khalid Abdullah is a student of MS (Software Engineering) at International Islamic University Islamabad, Pakistan (e-mail: am.khalid@yahoo.com).

Khalid Rashid is with Faculty of Applied Science, International Islamic University Islamabad, Pakistan (e-mail: drkhalid@yahoo.com).

Hafiz Farooq Ahmad is with Department of Computer Science, International Islamic University Islamabad, Pakistan (e-mail: farooq@comtec.co.jp).

Unified Modeling Language (UML) [7] of OMG group is one of the most popular modeling languages to design different artifacts of the software systems. UML [15] provides numerous diagrams to model properties of a software system [16]. It has become an industry standard now for a while. It provides a verity of diagrams that can be used to model software for aspect oriented development paradigm [17] [18]. Among these diagrams, state charts are very important to model the dynamic behavior of the system. When the decision on what action is to take in response to a given input, the state chart is an effective design tool [18]. This work explains how the shared join points can be modeled using state charts of UML 2.0 at high level design. It also proposes a methodology to formulate the strategies based on dynamic decisions at high level design and finally the formulated strategy is implemented in AspectJ, one of the most prominent Aspect Oriented Programming languages [5].

The rest of the paper is organized as follows; Section 2 provides literature review. Section 3 presents proposed methodology for modeling shared join points with state charts at high level design. Section 4 describes the application of the proposed methodology to a case study, and finally, Section 5 concludes the paper.

II. Literature Review

A detailed analysis of the problem aroused by shared join points is discussed by Nagy et al. in 2005. Multiple aspects' superimposition on the same join point affects the functionality of each other due to different execution orders among them. Software engineering perspective of Shared Join Point problems is also discussed. It is recommended that, to offer one solution which satisfies only a single case is not preferred. AOP languages should offer a rich set of language mechanism for composition specifications, so that, the developers may choose the right specification for their problem. It is very important to identify conflicts among aspects at shared Joint point for the safety and correctness. The already presented core model [10] is enhanced by adding more constraints and the composition rules for multiple constraints. The integration of the purposed model with AspectJ is also presented. This model can be used with AspectJ, if AspectJ support the named advices. Also the Join Point interface has to be extended for this purpose. For ordering, AspectJ uses declare precedence construct and for controlling, the construct presented in Core Model needs language support [8] [10].

Anis C. et al. presented an interaction model on the basis of Interaction Specification Language (ISL) for modularizing crosscutting concerns of component based applications. The main idea of interactions is to rewrite a method body using the reaction (advice). The interaction model is used to handle the issues arouse by the Shared Join Point in a way that, the composition mechanism generates an advice, which is the result of merging all advices at that join point. Whenever a shared join point is reached one single advice is executed, which is semantically equivalent to the composed advice. The merging mechanism is based on a finite set of merging rules. The software engineering properties such as analyzability and predictability can also be achieved by using this tool. Testing and verification becomes much simpler [11].

Mahoney et al. described the importance of extended Finite State Machines in order to capture the dynamic behavior of systems [18]. A state chart is connected to a class that specifies all behavioral aspects of the objects in that particular class. They also describe that Aspect Oriented Modeling can help in bridging the gab between software design and implementation through the use of advanced features of state charts. They have proposed a framework which helps in simplifying the design of core requirements and cross cutting concerns.

Mahoney elaborated the need of crosscutting concerns of reactive systems using state machines. State Charts are used to describe the dynamic behavior of separate concerns. The core and aspectual requirements are represented by state in different orthogonal regions. He addressed the communication mechanism in orthogonal regions through broadcast events. The broadcast events are used as a mechanism for implicit weaving of aspect and core model in state charts [19].

The programming constructs of AspectJ are introduced by Kiczales et al. The application of advices of two conceptually and semantically independent aspects at the same join point is addressed. It also described that the programmer does not need to control relative ordering of such advice [5].

Mohamed Mancona Kande et al. explained the basic concepts of AspectJ, a state of the art Aspect Oriented Programming Language. Standard UML is used for modeling these concepts and limitations of UML are highlighted. Some extensions to UML are proposed to overcome these limitations. A bottom up approach is followed for designing classes and aspects of Aspect Oriented Programming [16].

The concepts of Join Point as Static Join Point and Dynamic Join Point are addressed. UML association classes (along with their new features), ports and connectors are used among components for modeling [7].

Stein D. et al. presented an approach to model the join points with the help of Join Point Indication Diagram (JPID) and Join Point Designation Diagram (JPDD). JPID is presented for the indication of join points in core model while JPDD is presented for the indication of join points in aspects [18] [19] [20], but it does not address any solution for Shared Join Points.

There is massive work on modeling, modeling join points as well as on aspect oriented programming where as the work on shared point is only at implementation level. There is no solution presented by the researchers to represent particular issues of shared join point at high level in formulation of the suitable design strategies for shared join points.

III. Proposed Methodology

This section presents the proposed methodology based on state charts of UML 2.0. The section is divided into two sub-sections. First describes, why to use state charts for shared join point modeling and second subsection explains the proposed design methodology.

A. State Charts for Shared Join Point Modeling

The Unified Modeling Language (UML) has become industry standard for modeling general and as well as for specific purpose software artifacts. State charts in UML 2.0 [15] are very important means of modeling and capturing the dynamic behavior of objects. State chart related to a class, can specify all the behavioral aspect in that class [18]. A state chart diagram is represented through a state machine which models the individual behavior of the object. State machines throughout the UML versions remained almost same [20]. However, some new elements like entry and exit point are introduced in UML 2.0 [15]. Some of the elements of UML 2.0 like, composite state, choice pseudostate and terminate pseudostate are very important means to model the behavior.

B. Methodology

The proposed design methodology mainly uses composite state of UML for the multiple aspects' composition at the shared join point. The model consists of three main composite states. These composite states are composition of the aspectual and core requirements. Aspectual requirements are further subdivided into two composite states; *beforeCompositeState* and *afterCompositeState.* If multiple aspects are superimposed before the core requirement, then their ordering and controlling will be handled in *beforeCompositeState*. And if the multiple aspects are superimposed after the core requirement, then their ordering and controlling will be handled in *afterCompositeState*. This means that each composite state is responsible for handling issue related to superimposed aspects contained by that composite state. The core requirement is composed in the *coreCompositeState*. These composite states are named as just for understanding. So, at the abstract level, the aspectual and core requirement compositions are handled in composite states. The big picture of proposed methodology is shown in Fig. 1.

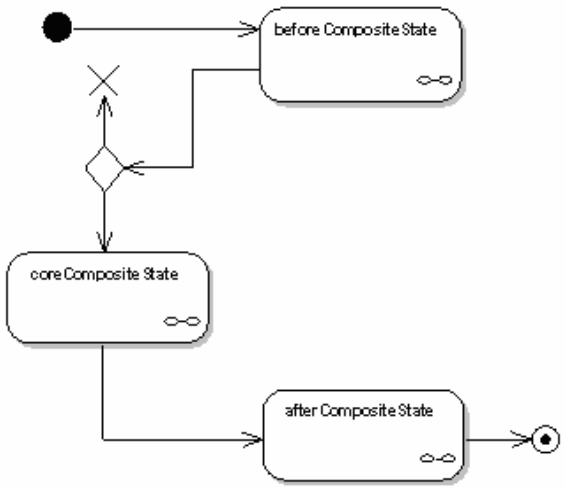

Fig. 1 Abstract view of proposed methodology

IV. Case Study

To understand the problems of shared join point, it becomes more convenient if we consider following case study. There are also some examples related to these issues, discussed in [8] [11]. The case study is about university course registration system. There are some requirements such as, no student will be allowed to register course without prior submission of fee, and also no course will be registered if its prerequisites are not passed by the student. The system should maintain log and database persistence. In this scenario of university course registration, *CourseRegistration* class fulfils the core requirement of the system. The requirement of course registration is fulfilled by *registerCourse()* method. The other requirements are implemented in different aspects.

Suppose that requirement of logging is implemented by aspect named *Logging*. The responsibility of *Logging* aspect is to maintain the log of every entrance to a method, so *Logging* aspect should run before the *registerCourse()* join point which is now well defined point in the program.

There are other requirements of student fee checking and course prerequisites checking for the course being registered by the student. These requirements are implemented by *CheckFee* and *CheckPreRequisite* aspects respectively. These aspects should also run before the *registerCourse()* join point. Now, all three aspects will be executed on the *registerCourse()* join point at the same time. In other words this join point is being shared by the multiple superimposed aspects.

The problem of ordering and controlling among these three aspects arises. If either of the aspects does not provide the desired results, the course should not be registered. The last requirement is to check the database persistence implemented by *DBPersistence* aspect, which ideally should run at the end when course has been registered.

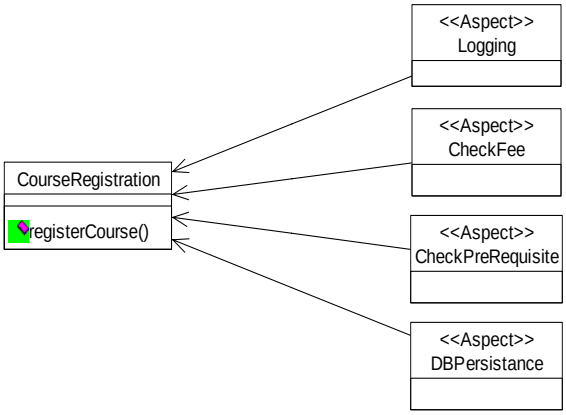

Fig. 2 Multiple aspects' composition

Suppose that database is up and *DBPersistence* aspect should work perfectly if the course has been registered. Now, the superimposition of three aspects requires great efforts to handle their ordering and controlling issues. The superimposition of the multiple aspects at the join point *registerCourse()* are shown in Fig. 2. All four aspects are superimposed with *registerCourse()* join point. The ordering and controlling issues are discussed at implementation level and some strategies and software engineering rules are also highlighted in [8] [11]. In order to define better ordering and controlling strategies, one needs to model shared join points at high level design, so that the ordering and controlling

strategies can be implemented perfectly at implementation level. This reduces the work cost by identifying suitable strategies at the early software development stage.

A. Application of the Proposed Methodology

The proposed methodology uses state charts, a design constructs of UML 2.0 [15] to represent shared join point for the case study discussed in university course registration scenario.

This model comprises of three main composite states; *beforeCompositeState* is to formulate the aspects that need to be run before the core functionality which will be composed in the second composite state *coreCompositeState*. Last composite state is *afterCompositeState* which should contain the aspect(s) that require(s) to be run after the core functionality as shown in Fig. 3. Purposed model can be customized. For example, there can be another composite state for those aspects which need to be executed before as well as after the core requirement i.e. around the core requirement.

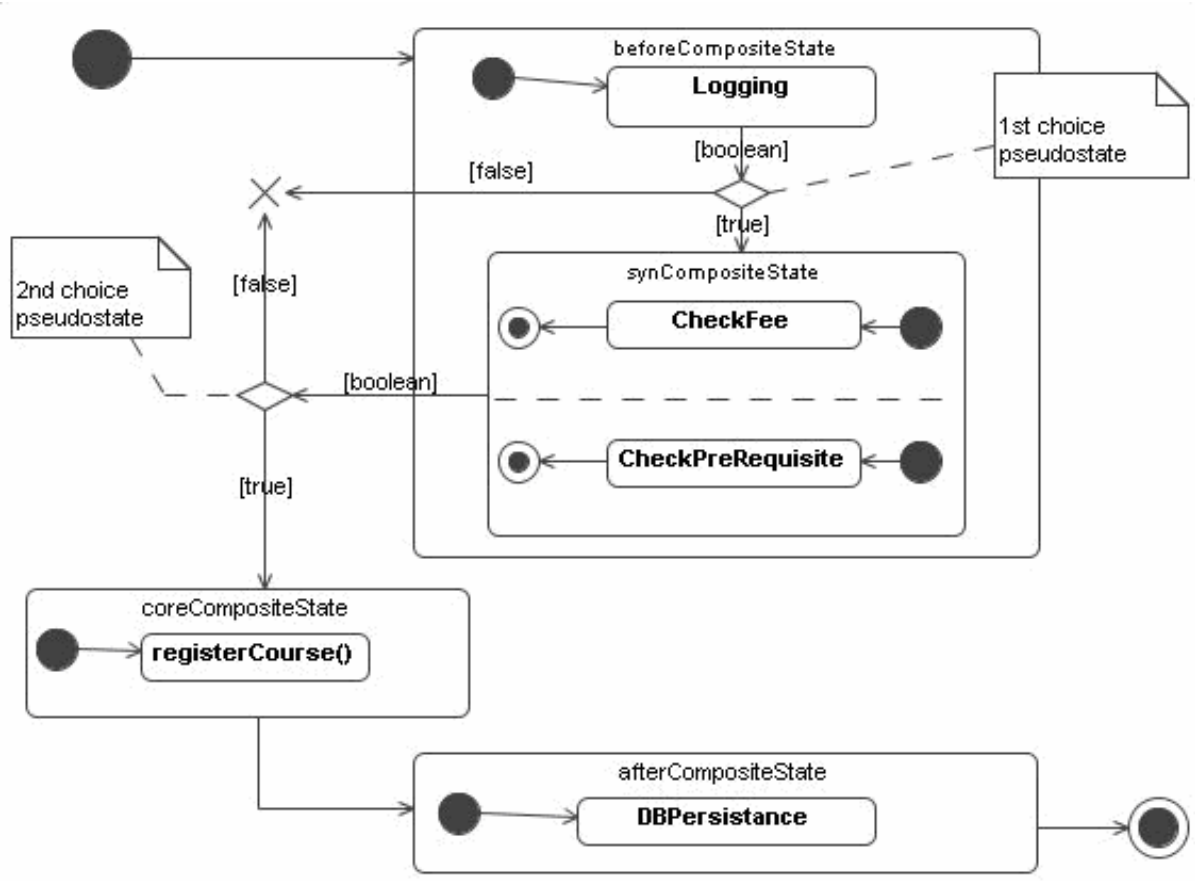

Fig. 3 Detail design of proposed methodology for case study

Two different categories of composite states are defined; one implementing the core requirement and other implementing the aspectual requirement. We further sub-categories the aspectual composite state into before and after composite states. By having three composite states at abstract level, we can easily model the core and aspectual requirements in a systematic way.

We are more concerned with the first composite state i.e. *beforeCompositeState* that contains the aspect need to be executed before the core requirement and on the bases of their results the core requirement will be implemented. This composite state contains an aspect *Logging* and a sub-composite state *syncCompositeState* which contains two aspects *CheckFee* and *CheckPreRequisite* in its orthogonal region for their concurrent execution. The aspects in *beforeCompositeState* are superimposed and required to be ordered and controlled. If we consider the above discussed scenario, the *Logging* aspect should always run first. This *Logging* aspect will transmit boolean guard value to first choice pseudostate as input, which will evaluate these boolean guard values and decides where to transmit the object. If the boolean guard value is evaluated as true, the object will be forwarded to *syncCompositeState* otherwise it will be sent to terminate pseudostate, which destroys the object action. If the object is in *syncCompositeState* and the boolean guard value has been referred to *CheckFee* and *CheckPreRequisite* aspects in the orthogonal regions, both will synchronously run and their results will be transmitted as boolean guard values to second Choice pseudostate that evaluates the boolean guard value. If the value is true, it implements the course registration core requirement otherwise object will be sent to terminate pseudostate. Synchronous handling of two aspects takes place

because they don't require any ordering constraint. Any one of the aspects in *syncCompositeState* could run after the other. Also the synchronous handling of these aspects in the orthogonal regions will be handled by default as run to-completion, in which each event is handled before the next occur [20].

The *DBPersistence* aspect is an aspect that should run after core requirement, so there is no need to order or control this aspect. This representation allows the designers to show a high level ordering and controlling mechanism for superimposed aspects. It also shows how to resolve ordering and controlling issues discussed. It provides flexibility to designers to apply any of the strategies given in [9] at high level design. By selecting a suitable strategy to resolve the issue at high level design will help the programmers to implement the strategy in an ideal way. It provides additional feature of handling the new aspectual requirement in systematic way. The presented design of shared join point allows the designer to represent shared join point independent of the implementation details at abstract level [21]. The design at such an abstract level can provide benefits like scalability, by representing new conflicting aspects in the model, reusability, reusing the design strategy for other similar shared join points, and maintainability, if any of the conflicting aspects to be removed or to be added by identifying its effect on the other aspects. Early representation of ordering and controlling issues reduces the work cost by identifying the complexity of issues and formulating suitable strategies to solve the issues accordingly. Early the problem is identified cheaper to solve.

V. Conclusion

Previously shared join points were discussed at implementation level and to the best of our knowledge there is no research done for this particular problem at design level. The paper describes the need of modeling shared join point at high level design. In this regard, state charts of UML 2.0 have been used for modeling at high level. The main elements of state charts are composite states which compose the superimposed aspects at shared join point. At abstract level, the composite states are used to categorize into core and aspectual requirements. The strategies for ordering and controlling are implemented through detailing the state charts using the choice pseudostate as dynamic selection element and guard values to evaluate the next transition. State charts represent a better way to handle shared joint point Issues at high level design. This allows the designers to design issues regarding the shared join points at early design stages, and programmers can implement these strategies accordingly. A case study is presented with proposed methodology, which shows in detail how issues(s) regarding shared join point(s) can be represented at high level design.

References

[1] Aspect-Oriented Software Development. http://www.aosd.net.

[2] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, Ch., Loingtier, J.and Irwin, J.: "*Aspect-Oriented Programming*". In Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97) (Yväskylä, Finland, June 9-13, 1997). Springer-Verlag, Berlin Heidelberg, 1997, LNCS 1241, Pages 220–242.

[3] Stein, D., Hanenberg, St., Unland, R..: "*An UML-based Aspect-Oriented Design Notation For AspectJ*". In Proc. Of AOSD '02 (Enschede, the Netherlands, Apr. 2002), ACM, pp. 106-112.

[4] Wesley Coelho and Gail C. Murphy: "*Modeling Aspects: An Implementation-Driven Approach"*. Workshop on Best Practices for Model Driven Software Development at OOPSLA 2004.

[5] Kiczales, G, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm & W. Griswold: "*An Overview of AspectJ*". In Proceedings of ECOOP 2001, LNCS 2072, Springer Verlag, 2001.

[6] Boucke N., Holvoet T.: "*State-based join points: Motivation and requirements*". In Filman, R. E., Haupt, M., Hirschfeld, R. (eds): Proceedings of the Second Dynamic Aspects Workshop (2005) 1-4.

[7] Eduardo Barra Zavaleta, Gonzalo Génova Fuster, Juan Llorens Morillo: "*An approach to Aspect Modelling with UML 2.0*". Workshop on Aspect Oriented Modeling, October 11, 2004, Lisbon, Portugal, held in Conjunction with the 7th International conference on the Unified Modeling Language- UML 2004, October 10-15, 2004, Lisbon, Portugal.

[8] I. Nagy, Lodewijk Bergmans and Mehmet Aksit: "*Composing Aspects at Shared Join Points*". Proceedings of International Conference NetObjectDays, NODe2005.

[9] Stein, D., Hanenberg, S. and Unland, R.: "*On Representing Join Points in The UML*". In Proceedings of the 2nd Workshop on Aspect Modeling with UML at the Fifth International Conference on the Unified Modeling Language and its Applications (UML 2002), (Dresden, Germany, 30 September – 4 October, 2002).

[10] I. Nagy, L. Bergmans, M. Aksit.: "*Declarative Aspect Composition*". Technical Report, http://trese.ewi.utwente.nl/publications/publications.php?action=showPublication&pub_id=346 University of Twente, (April 2005).

[11] Anis Charfi, Michel Riveill, Mireille Blay-Fornarino, Anne-Marie Pinna-Déry: "*Transparent and Dynamic Aspect Composition*". In Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT), Bonn (Germany), 21 march 2006.

[12] A. Rashid, N. M. Ali: "*A State-based Join Point Model for AOP*". Workshop on Views, Aspects and Roles — VAR (held with ECOOP 2005).

[13] Stein, D.; Hanenberg, S.; Unland, R.: "Modeling Pointcuts. Early Aspects". Workshop on Aspect-Oriented Requirements Engineering and Architecture Design, AOSD 2004, Lancaster, UK, March 22, 2004.

[14] Eric Braude: Software Design: From Programming to Architecture. John Wiley & Sons, Inc. 2004.

[15] Object Management Group: "Unified Modeling Language". Superstructure, version 2.0 formal/05-07-04.

[16] Mohamed Mancona Kande, Jorg Kienzle and Alfred Strohmeier, "From AOP to UML- A Bottom-Up Approach", Swiss Federal Institute of Technology Lausanne, Switzerland. [2001].

[17] A. Rashid, Araujo J., A. Moreira, and I. Brito: "Aspect-Oriented Requirements with UML". Workshop on Aspect-Oriented Modeling with UML (held with UML 2002).

[18] Mahoney, M., Bader, A., Aldawud, O., Elrad, T.: "Using Aspects to Abstract and Modularize Statecharts." The 5th Aspect-Oriented Modeling Workshop in Conjunction with UML 2004. http://www.cs.iit.edu/~oaldawud/AOM/mahoney.pdf.

[19] Mark Mahoney: "Modeling Crosscutting Concerns in Reactive Systems with Aspect-Orientation". Doctoral Symposium at MoDELS/UML 2005, Montego Bay Jamaica, October 2005.

[20] Michelle Crane, Juergen Dingel: "*UML Vs. Classical Vs. Rhapsody Statecharts: Not All Models Are Created Equal*". ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2005).

[21] Stein, D.; Hanenberg, S.; Unland, R.: "*Position Paper on Aspect-Oriented Modeling: Issues on Representing Crosscutting Features*". 3rd International Workshop on Aspect-Oriented Modeling with UML, AOSD 2003, Boston, MA, March 18, 2003.