

Towards Supporting the Architecture Design Process Through Evaluation of Design Alternatives

Lihua Xu, Scott A. Hendrickson, Eric Hettwer,
Hadar Ziv, André van der Hoek, and Debra J. Richardson
Donald Bren School of Information and Computer Sciences
Department of Informatics
University of California, Irvine
Irvine, California 92697-3455, U.S.A.
+1 949 824 6326

{lihuax, shendric, ehettwer, ziv, andre, djr}@ics.uci.edu

ABSTRACT

This paper addresses issues involved when an architect explore alternative designs including non-functional requirements; in our approach, non-functional requirements are expressed as statecharts. Non-functional requirements greatly impact the resulting design of a system because they naturally conflict with each other, crosscut the system at multiple points, and may be satisfied in a number of different ways. This makes correctly designing them early in the software lifecycle critical, since correcting them later can be extremely costly. Our approach supports an architect generating and evaluating many different design alternatives. This explorative process is not well supported by current techniques, which focus on documenting the *result* of this process, but not on assisting the designer *during* this process. We present an architecture-based approach that supports exploration of non-functional requirements expressed as statecharts. Our approach captures design alternatives of non-functional requirements separately, composes different system designs from these alternatives using a novel weaving technique, and analyzes the resulting design for specific qualities using simulation.

Categories and Subject Descriptors

D.2.2 [Design Tools and Techniques] – state diagrams; D.2.4 [Software/Program Verification] – validation; D.2.10 [Design] – representation; D.2.11 [Software Architectures] – languages.

General Terms

Design, Languages, Verification.

Keywords

Software architecture design, analysis, non-functional requirements, aspect-oriented design, state charts, simulation.

1. INTRODUCTION

Designing a software system is an explorative, creative process incorporating and balancing the functional and non-functional

requirements of a system. Functional requirements have received much attention and have been addressed both structurally and behaviorally. Non-functional requirements have not received as much attention, however, despite the fact that they greatly impact the resulting design of a system.

Non-functional requirements are especially difficult to address due to their unique qualities, particularly (1) *their conflicting nature*, as can be seen with the trade-off between making a system reliable vs. responsive, (2) *their crosscutting nature*, as can be seen when incorporating security, which requires modifying a system at multiple points, and (3) *their open-ended nature*, as can be seen with realizing security, where authentication could be used alone or with encryption. Consequently, correctly designing a system to satisfy non-functional requirements early in the software lifecycle is absolutely critical, since correcting them later can be extremely costly.

Typically, an architect must generate and evaluate many different design alternatives that address non-functional requirements – evaluating them, revisiting them, and refining them, until an adequate design is created. This is an explorative *process*, and is not well supported by current techniques. Previous techniques focus instead on documenting the *result* of this process (e.g., Promela [9], Rapide ADL[15], Statecharts [8], etc.) and analyzing the resulting document (e.g., SPIN [10], Rapide [14], Argus-I [19]).

In this paper, we propose a novel, state-based approach that:

1. *Explicitly and individually models design alternatives* of non-functional requirements using (partial) statecharts. This allows an architect to create and reason about individual design alternatives, rather than modeling each alternative system in its entirety.
2. *Composes these design alternatives into a single design* using an “architectural weaving” technique. This allows an architect to easily create and maintain different system designs consisting of desired combinations of design alternatives.
3. *Analyzes the composed system* against specific non-functional requirements. This allows an architect to evaluate different system designs during exploration.

We have begun implementing this approach in EASEL [11], a layered design tool that supports the modeling of components, connectors, and interfaces. Currently, we can model design alternatives and simulate simple statecharts within the context of an architecture. Composing design alternatives through architectural

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ROSATEA '06, July 17, 2006, Portland, Maine, USA.
Copyright 2006 ACM 1-59593-459-6/06/07...\$5.00.

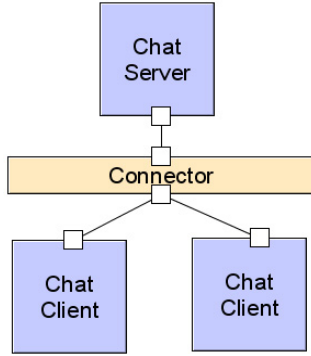


Figure 1. The chat system architecture

weaving is done manually. Thus, in this paper we focus on the approach and its underlying concepts.

The remainder of this paper is structured as follows. In Section 2, we present a motivating example that we use throughout the paper. Section 3 describes the scope of our approach, and Section 4 discusses in detail our approach of explicitly and separately modeling design alternatives of non-functional requirements and how they are composed into a system and verified. We conclude in Section 5 with related work and in Section 6 with an outlook at our future work.

2. MOTIVATING EXAMPLE

To understand the problem targeted by our approach, we introduce here a motivating example that we also use as the running example throughout the remainder of the paper. Consider a hypothetical “chat” system, the architecture of which consists of the server and client components shown in Figure 1. In this system, the server component routes messages between the clients.

To illustrate the problem of designing a system with non-functional requirements, consider the two non-functional requirements: “the system should be secure,” and “the system should be responsive.” To further illustrate these problems, suppose that the architect is considering authentication as a means of satisfying the security requirement. Incorporating authentication would require that clients authenticate with the server before sending messages to other clients in the system. This helps enforce security by preventing unauthorized users from participating. The architect further is considering incorporating encryption into the system to ensure that text messages sent by a client can only be read by the intended recipients.

These non-functional requirements exhibit the problems expressed in the introduction. They are *conflicting* because introducing encryption into the system will undoubtedly reduce its responsiveness. They are *crosscutting* because both the server and client components will have to be modified to make the system secure, and the response time can be negatively affected by any of the architectural components. Finally, they are *open-ended* because it may be sufficient to satisfying security using only authentication, or incorporating encryption may also be necessary and responsiveness could be measured in many different ways.

The architect must decide whether encryption can be included in the system without violating the responsiveness requirement.

With today’s technology, the architect must model these two alternatives in their entirety. When more than two alternatives are involved, or when additional non-functional requirements must be addressed, modeling each alternative separately quickly becomes unwieldy. Additionally, if these non-functional requirements change, the architect must first untangle from each modeled alternative the respective non-functional requirements in order to understand and change them. This, again, can be quite unwieldy.

To illustrate the process of modeling each alternative in its entirety, consider the example statecharts shown in Table 1, which must be created manually using traditional approaches. Table 1 shows the behavior of individual architectural components for the basic functional system, the system with authentication, and the system with both authentication and encryption. The behavior of the entire system is then derived by composing the respective component statecharts according to the architectures’ topology. In each statechart, a trigger event with an asterisk indicates that the event is received from other components in the system. Trigger events without an asterisk are triggered outside the system, such as being initiated by the user.

3. SCOPE

Non-functional requirements specify software quality attributes, such as accuracy, performance, security and modifiability that a system must exhibit. However, many non-functional requirements are difficult to address in a system because they typically interact with each other, broadly impact the system, and may be subjective [3]. As such, non-functional requirements cannot be fully or absolutely satisfied, rather at best they can be *satisfied* [17], i.e., satisfied within some acceptable limits.

Previous work [20] classifies most types of non-functional requirements into two distinct categories:

- *Operationalizable* – those that can be realized functionally in the software. For example, a system may be made secure by incorporating authentication and encryption mechanisms.
- *Checkable* – those that can be supported by design choices and verified, but cannot be implemented directly. For example, a number of different design decisions can improve system performance, which can then be verified, but implementing performance directly is difficult.

Our approach models operationalizable non-functional requirements behaviorally using statecharts and then verifies the resulting system against checkable non-functional requirements by simulating and monitoring the system to determine if the non-functional requirements are *satisfied*.

4. APPROACH

To support an architect in the creative design process of exploring different design alternatives for non-functional requirements, we build on the concepts introduced in our previous system, EASEL [11]. EASEL supports the incremental exploration of architectures by separating out different functional concerns on to different layers. These layers are then composed to create and explore different architectures exhibiting different concerns. To date, EASEL is limited to supporting the modeling of components, connectors, and interfaces.

In this section, we present our approach to a design and analysis extension to EASEL that aids a designer in exploring different design alternatives for non-functional requirements. Our approach also builds on the approach in Argus [19], which models component behaviors individually using statecharts [8] and simulates the behavior of the architecture by concurrently executing each component's statechart. Events emitted by one component's statechart are routed to other component statecharts according to the topological arrangement. This simulation is monitored to verify whether the non-functional requirements are being *satisfied* [17].

What is novel about our approach is that, rather than modeling operationalizable non-functional requirements implicitly within the individual component statecharts, we model them explicitly and individually as *behavior modifiers*. Each behavior modifier captures a different behavioral aspect of a non-functional requirement and can be optionally woven into an architecture at key

locations specified in *binders* [20]. Analogous to the layered composition of functional requirements provided by EASEL, the architect chooses which design alternative for non-functional requirements to weave into a system by selecting the design alternatives corresponding binders, then simulates the system to verify that they are being *satisfied*. By combining different design alternatives into the system and then verifying the system, the architect can easily explore different design alternatives, which was difficult using the traditional approach discussed above since each design alternative had to be specified in its entirety.

4.1 Behavior Modifiers

As stated, behavior modifiers capture the essential functional aspects of operationalizable non-functional requirements. Because of the crosscutting nature of non-functional requirements, behavior modifiers may need to be woven into multiple locations

Table 1. System designs exhibiting different design alternatives for operationalizable non-functional requirements.

Design Alternative	Server Component	Client Components
Basic		
Authenticating		
Authenticating & Encrypting		

* Events with an asterisk are triggered by other component actions. Those without an asterisk are triggered outside the system.

Table 2. Behavior modifiers for our example system

Behavior Modifier	Partial Statechart
Authenticate	
Verify Authentication	
Requires Authentication	
Encrypt & Send	
Receive & Decrypt	
Send Encrypted	
Receive Encrypted	

throughout the architecture. It is, therefore, necessary to specify behavior modifiers in a generic way, independent of any specific component.

Consider the motivating example presented in section 2, but now modeled using behavior modifiers. An architect would start with the *Basic* design alternative in Table 1 and create behavior modifiers to model the functional aspects of the operationalizable non-functional requirements; Table 2 shows one such collection. In Table 2, the *Authenticate* behavior modifier authenticates the user by prompting the user for their password and marking them as authenticated if it is valid. The transition to the next state in the targeted statechart (indicated by the terminal pseudostate) only occurs when the user has successfully authenticated or was already authenticated. When a user is unauthenticated, a transition back to the *Prompting* state allows the user to re-authenticate. The *Verify Authentication* behavior modifier transitions between a state where the user is not authenticated (indicated by the initial pseudostate) and one where the user is authenticated (indicated by the terminal pseudostate). In addition to transitioning to the appropriate state, it records whether the user is authenticated in the *authenticated* variable. The *Encrypt & Send* behavior modifier encrypts a users’ message and sends it as an event. Finally, the *Send Encrypted* behavior modifier simply sends an encrypted message as an event. The other behavior modifiers can be similarly interpreted. The behavioral aspects of operationalizable non-functional requirements may be modeled in many different ways and is largely up to the personal preferences of the architect; those shown in Table 2 are only one such possibility. For instance, one could simplify the *Encrypt & Send* behavior modifier so that it

only encrypts a message, but does not send it, instead utilizing the *Send Encrypted* behavior modifier to send the event.

Note that although a behavior modifier looks like a traditional statechart [8], it allows transitions to start from, and end at, both the initial and terminal pseudostates. This is only a minor variation on traditional statecharts, and reflects the fact that behavior modifiers do not capture stand-alone statecharts, but rather modifications that are woven into a set of statecharts. In our approach, the initial and terminal pseudostates of a behavior modifier are replaced by the neighboring states of the target statechart into which it is woven. The weaving process is discussed in the next section.

4.2 Architectural Weaving

Having modeled behavior modifiers, the architect needs to be able to weave them together into a single design that can be evaluated. For instance, to create the *Authenticating* design alternative shown in Table 1, the architect must weave into the *Basic* design, the *Authenticate* and *Verify Authentication* behavior modifiers of Table 2. To create the *Authenticating & Encrypting* design alternative, the architect would also include the remaining behavior modifiers. To accomplish this, it is necessary to be able to dynamically define *where* our crosscutting behavior modifiers are woven, and *how* to weave them into each location. The first question of *where* is specified by binders. Binders describe where behavior modifiers are to be woven into the component statecharts of the architecture in terms of trigger events, actions, and states.

To illustrate a binder, consider the *Authentication* binder shown in Figure 2. This binder advises that three behavior modifiers that should be woven into the system. Its first piece of advice states that the *Authenticate* behavior modifier should be woven into the *Client* component’s statechart before the *Idle* state. This modifies the client so that the user must be authenticated with the server before entering the *Idle* state. The second piece of advice states that the *Verify Authentication* behavior modifier should be woven into the *Server* component’s statechart around the *Idle* state. This allows the server to respond to authentication requests. The final piece of advice states that the *Requires Authentication* behavior modifier should be woven into the *Server* component’s statechart before it receives any **message* events. This further modifies the *Server* component so that it does not process messages from clients that are not authenticated. Refer to our previous work [20] for more information about binders.

The second question of *how* behavior modifiers are woven into the specific locations within a statechart is governed by a set of weaving rules (not shown in this paper) that describe how to weave a behavior modifier *before*, *after*, or *around* a specific *state*, *trigger event* or *action*. Figure 3 illustrates a specific case of what happens to the *Basic Client* statechart when the architect includes the *Authentication* binder presented in Figure 2. Recall that this binder specifies that the *Authenticate* behavior modifier should be woven into the *Client* statechart *before* the *Idle* state. Because this behavior modifier is to be woven *before* the *Idle* state, transitions that previously went to the *Idle* state are redirected to point to the initial pseudostate of the behavior modifier. Likewise, transitions that initially went from the terminal pseudostate of the behavior modifier are redirected to point to the *Idle* state. This intermediate statechart is shown in Figure 3. To pro-

```

<binder id = "authentication_design_alternative">
  <advice
    behaviorModifier = "Authenticate"
    type = "before">
    <pointcut
      target = "Client component"
      pattern = "Idle state" />
    </advice>
  <advice
    behaviorModifier = "Verify Authentication"
    type = "around">
    <pointcut
      target = "Server component"
      pattern = "Idle state" />
    </advice>
  <advice
    behaviorModifier = "Requires Authentication"
    type = "before">
    <pointcut
      target = "Server component"
      pattern = "*message trigger event"/>
    </advice>
</binder>

```

Figure 2. Binder for authentication design alternative

duce the final statechart, the behavior modifiers pseudostates are removed, and the transitions are combined. Consequently, transitions that initially went to the *Idle* state now go to a *Prompting* state, and transitions that initially involved the terminal pseudostate now involve the *Idle* state, as shown in the last step in Figure 3.

Other types of weaving are performed similarly. Weaving *after* a state is similar to *before*, except that the initial and terminal pseudostates of the behavior modifier play opposite roles. When weaving *around* a state, the initial and terminal pseudostates of the behavior modifier play the same roles: they are both replaced by the targeted state. Of interest is the weaving of the *Verify Authentication* behavior modifier into *server* component *around* the *Idle* State. Recall that this behavior modifier supports transitioning between two separate target states, one for being authenticated and one for being unauthenticated. However, in our *Server* component we use the *authenticated* variable to determine whether a client is authenticated. By weaving *around* the *Idle* state, transitions that involved the initial and terminal pseudostate are all redirect to the *Idle* state, and the *authenticated* variable is used instead to enforce authentication. This highlights an instance where the behavior modifier can be incorporated in different ways. Weaving *before* and *after* transitions and events are handled similarly to that of states. However, weaving *around* a transition or event differs in that the targeted transition or event is replaced.

4.3 Relationships

It is possible that two design alternatives conceptually depend on, or conflict with, each other. In our example system we have modeled the behavior modifiers for encryption separately from the behavior modifies for authentication. Therefore, it is technically possible to weave encryption into the system, without also weaving in authentication. However, as designers, let us suppose that

we made the conceptual decision to couple encryption with authentication, i.e. weaving encryption into the system requires that we also weave authentication into the system. Capturing this information explicitly is useful to aid a designer in exploring different alternatives as it allows the designer to focus on exploring combinations that are potentially valid.

It is also possible to have conceptual conflicts. Perhaps, after analyzing the system, the designer discovers that it is not possible to include encryption without violating the systems' desired responsiveness constraint. If this is the case, the designer would want to capture this information so that future effort will not be wasted by (accidentally) revisiting design alternatives that incorporate both of these requirements. It becomes more important to model these types of conflicts as the number of non-functional requirements that the architect is exploring increases.

Conflicts may also exist between design alternatives that modify the same join point, or design alternatives may have to be applied in a particular order. For example, one modifier may change the system, so that a certain join point referred to by another modifier no longer exists.

We plan to utilize and extend the relationships that are already present in EASEL. These relationships are used to model conceptual dependencies or conflicts between different features (captured as layers) in product line architectures [1]. We believe that these relationships can be directly applied to our binders and improved upon to capture such conceptual relationships between design alternatives for non-functional requirements.

4.4 Analysis

Our goal is to provide a variety of analysis methods that aid an architect in evaluating design alternatives. Our first step uses simulation. The architecture is simulated by concurrently executing all component statecharts, using the architecture topology to route emitted events between individual component statecharts.

In order to view the different impacts that each design alternative might have on the system, the architect would select one design alternative at a time and perform simulation on the composed system. To aid the architect in determining if the non-functional requirements are being *satisfied*, monitors associated with each non-functional requirement or design alternative observe the simulated system and inform the architect when the simulation does not execute as required.

Returning to our example chat system, we present one possible collection of monitors that could be used to verify that the resulting system *satisfices* the non-functional requirements in Table 3. Note that like the behavior modifiers presented in Table 2, we have identified monitors that verify *Authentication* and *Encryption* separately. To verify that a system *satisfices* both design alternatives, the architect would use both monitors together. The monitor for the *Responsiveness* design alternative is represented as number of simulation ticks. To determine which non-functional requirements conflict, the architect would simulate different systems composed of different design alternatives, observing which monitors are violated.

5. RELATED WORK

Modeling non-functional requirements at the level of software architecture has been the subject of other research. Several approaches have been proposed to systematically build software architectural models from functional and nonfunctional requirements [2, 4, 5, 7, 18]. However, their work does not differentiate between functional and non-functional requirements once the architecture is designed, making it difficult to explore different design alternatives that affect the design of the architecture.

The non-functional requirement framework presented by Mylopoulos et al. [3, 17, 21] provides a comprehensive methodology for the identification and refinement of non-functional requirements into software architecture components. Their work recognizes that an architect can never optimally satisfy non-functional requirements, but instead must explore different alternatives until a these requirements are *satisfied* [3] (i.e. acceptable, but not necessarily optimal). We consider our approach to be along the same lines as this philosophy; however, we additionally present an approach for supporting this exploration.

There has also been work on modeling the behavior of software architectures [6, 14, 16, 19]. These approaches explore a single architectural design, however, and are difficult to use when exploring many different design alternatives, especially design alternatives for which behavior may be modified at multiple points in the system. Our work follows the trend of Argus-I [19] using statecharts to describe the behaviors of software architectures, but

focuses on capturing non-functional requirements and weaving these throughout the system.

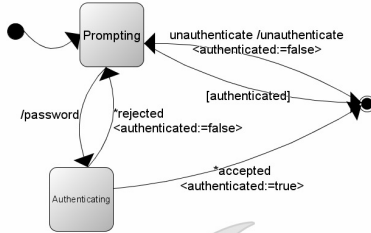
Aspect-oriented programming [12] is also related to our work. By their very nature, behavior modifiers are close to aspects in their compositional capabilities, but apply to statecharts. Binders [20] directly borrow the concepts of point cuts, join points, and advice from the aspect oriented community and apply them to statecharts. Further, as advocated in [13], we adopt the policy of explicitly specifying the weaving order of our design alternatives rather than using rules to specify this order.

6. CONCLUSION AND FUTURE WORK

This paper proposes a novel approach that supports the explorative design process of creating software architecture, with a specific focus on non-functional requirements. Our approach models non-functional requirements explicitly and individually from each other and the rest of the system as behavior modifiers. This allows an architect to directly reason about each non-functional requirement and its design alternatives, without needing to untangle it from an entire system. Behavior modifiers are modeled as partial statecharts, which are selectively woven into the system. The resulting system is simulated and monitors are used to observe whether the non-functional requirements are being *satisfied*.

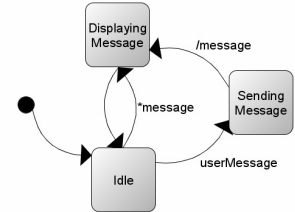
As stated, this paper presented our first step towards supporting the explorative architecture design process involving the evaluation of design alternatives. Our long-term goal is to investigate

Authentication Behavior Modifier

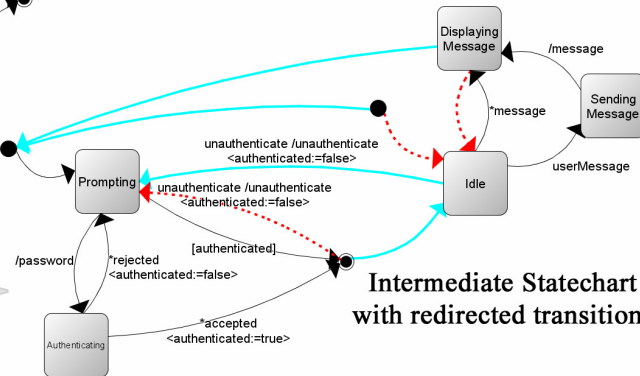


Transitions that initially went from the terminal pseudostate of the behavior modifier are redirected to point from the Idle state of the statechart.

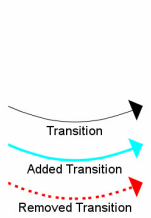
Basic Client Statechart



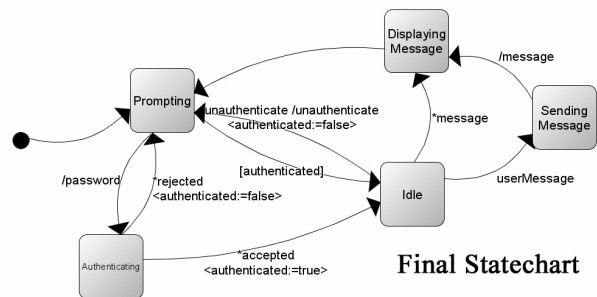
Transitions that initially went to the Idle state of the statechart are redirected to point to the initial pseudostate of the behavior modifier.



Intermediate Statechart with redirected transitions



The initial and terminal pseudostates of the behavior modifier are then collapsed to product the resulting statechart.



Final Statechart

Figure 3. Steps involved in weaving the *Authentication* behavior modifier into the *Basic Client* statechart before the *Idle* state.

Table 3. Monitors for each design alternative

Non-functional Requirement	Design Alternative	Monitor
Security	Authentication	Verify that no messages sent by an unauthenticated client will be delivered to other clients of the system.
	Encryption	Verify that no unencrypted messages are sent by the server or any clients.
Performance	Responsiveness	Verify that messages sent by a client are received by recipient clients within 50 simulation ticks.

additional modeling and analysis techniques for representing and analyzing design alternatives of non-functional requirements.

7. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments and suggestions.

Effort partially funded by the National Science Foundation under grant number CCR-0093489, DUE-0536203, and IIS-0205724.

8. REFERENCES

- [1] Bosch, J. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Wesley, A. ed. ACM Press, 2000.
- [2] Brandozzi, M. and Perry, D.E. From Goal-Oriented Requirements to Architecture Prescriptions: The Preskiptor Process. In *Proceedings of the Software Requirements to Architectures Workshop (STRAW)*. p. 107-113, May, 2003.
- [3] Chung, L., Nixon, B., et al. *Non-functional Requirements in Software Engineering*. Kulwer Academic Publishers, 2000.
- [4] Cysneiros, L.M., Leite, J.C.S.P., et al. A Framework for Integrating Non-functional Requirements into Conceptual Models. In *Proceedings of the Requirements Engineering*. 2001.
- [5] Cysneiros, L.M. and Leite, J.C.S.P. Nonfunctional Requirements: From Elicitation to Conceptual Models. *IEEE Transactions on Software Engineering*. May, 2004.
- [6] Egyed, A. and Wile, D. Statechart Simulator for Modeling Architectural Dynamics. In *Proceedings of the 2nd International Working Conference on Software Architecture*. Amsterdam, The Netherlands, August, 2001.
- [7] Gross, D. and Yu, E. From Non-Functional Requirements to Design Through Patterns. *Requirements Engineering Journal*. 6, 2001.
- [8] Harel, D. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*. 8, p. 231-274, 1987.
- [9] Holzmann, G.J. *Design and Validation of Computer Protocols*. Prentice Hall: Englewood Cliffs, N.J., 1991.
- [10] Holzmann, G.J. The Model Checker SPIN. *IEEE Transactions on Software Engineering*. 23(5), p. 279-295, May, 1997.
- [11] Institute for Software Research. *EASEL, An Extensible Architecting Support Environment with Layers*. <<http://www.isr.uci.edu/projects/easel/>>, University of California, Irvine.
- [12] Lopes, C.V., Kiczales, G., et al. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming*. Finland, 1997.
- [13] Lopez-Herrejon, R., Batory, D., et al. A Disciplined Approach to Aspect Composition. In *Proceedings of the Workshop on Partial Evaluation and Program Manipulation*. p. 68-77, Charleston, SC, January 9-10, 2006.
- [14] Luckham, D.C., Kenney, J.J., et al. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*. 21(4), p. 336-355, April, 1995.
- [15] Luckham, D.C. and Vera, J. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*. 21(9), p. 717-734, September, 1995.
- [16] Muccini, H., Bertolino, A., et al. Using Software Architecture for Code Testing. *IEEE Transactions on Software Engineering*. 30, p. 160-177, March, 2004.
- [17] Mylopoulos, J., Chung, L., et al. Representing and Using Non-Functional Requirements: A Process-Oriented Approach. *IEEE Transactions on Software Engineering*. 18, June, 1992.
- [18] van Lamsweerde, A. From System Goals to Software Architecture. In *Proceedings of the SFM*. p. 25-43, 2003.
- [19] Vieira, M.E.R., Dias, M.S., et al. Analyzing Software Architectures with Argus-I. In *Proceedings of the International Conference on Software Engineering (ICSE 2000)*. p. 758-761, Limerick, Ireland, June 4-11, 2000.
- [20] Xu, L., Ziv, H., et al. An Architectural Pattern for Non-functional Dependability Requirements. *Journal of Systems and Software*. to appear.
- [21] Yu, Y., Leite, J.C., et al. From Goals to Aspects: Discovering Aspects from Goal Models. In *Proceedings of the International Conference on Requirements Engineering*. 2004.