Requirements Trade-offs During UML Design
Anneliese Andrews¹, Per Runeson² and Robert France³

¹ Dept. of EE&CS, Washington State University, PO Box 642752 Pullman, WA 99164-2752, USA aandrews@eecs.wsu.edu
² Dept. of Communications Systems, Lund University, Box 118, SE-221 00 Lund, Sweden, per.runeson@telecom.lth.se
³ Dept. of Computer Science, Colorado State University, Fort Collins, CO 80523-1873, USA, france@cs.colostate.edu

Abstract

Designs almost always require trade-offs between competing design choices to meet pervasive system dependability requirements (e.g., security, performance and fault tolerance system goals). In some cases, dependability requirements are realized by functionality that cross-cuts designs. Aspect-Oriented Modeling (AOM) methods allow developers to localize such cross-cutting functionality in design modeling views called aspects. Aspects can be composed with other design views to obtain an integrated view of a design. This paper presents a technique that extends such methods to cover dependability requirements that are not directly realized by functional structures in a design. Performance goals provide examples of such requirements. We also present a trade-off mechanism to rank feasible solutions with respect to requirements priorities between different dependability requirements. The paper applies this technique to an example that has performance, fault-tolerance, and security requirements.

1. Introduction

When designing computer-based systems, the designers have to consider meeting a variety of dependability requirements besides basic functionality. Security, availability and fault tolerance goals provide examples of such requirements. Some of these requirements can be directly realized by functionality in a design. In these cases the pervasiveness of the requirement results in functionality that is distributed across a design. The Aspect-Oriented Modeling (AOM) method [1,2,3] provides techniques for describing the cross-cutting realizations of dependability goals in design views, called aspects. In this approach, designers specify core functional behavior in a primary model, and functionality that cross-cuts the primary model are described by aspects. Composing aspects with a primary model results in a more comprehensive and integrated view of the design that can be analyzed to determine the degree to which functional and dependability requirements are met. The approach uses the Unified Modeling Language (UML) [4] to describe designs.

Designing around separation of concerns has long been expected to produce more reliable artifacts, from requirements through the implemented system. This approach thus should work well with design requirements that can be directly realized by functionality expressible in the UML. But what about important requirements that cannot be expressed this way? Examples of such requirements include performance, quality of service, availability, and reliability requirements. In addition, requirements can compete with each other, forcing trade-offs when they cannot be met.

This paper explores two research questions related to designing with UML in the face of such requirements:

1. How should one express and evaluate “non-functional” requirements (i.e. those whose realizations that cannot be expressed in functional terms)?

2. How should one evaluate feasibility of combinations of functional and non-functional requirements (i.e. those expressible and those not expressible in UML) and perform trade-offs when they are not feasible?

We explore the principles of a solution to these two problems through the examples of security, fault-tolerance, and performance requirements. Section 2 discusses background material. This includes existing work on composing fault tolerance and security aspects into primary designs expressed in the UML. It also covers existing methods for trade-off evaluation. Section 3 describes our approach to solve research question 1, how to express and evaluate non-functional requirements. It uses performance requirements as an example. Section 4 presents our method to making trade-offs among competing requirements, using security and fault-tolerance as an example, i.e. our approach to solve research question 2. Section 5 presents an example case study on applying the method. Section 6 concludes with lessons learned and needs for future work.
2. Background

There are three areas of research that influences the work described here: (1) requirements prioritization, (2) AOM of non-functional requirements, and (3) software performance analysis. The first area of work is concerned with setting priorities between conflicting requirements that are consistent and provide at least one feasible solution. An extensive method to collect, quantify and analyze a multitude of requirements is presented by von Mayrhauser [5]. It includes specifying levels of functionality, reliability, performance, security, usability, portability, adaptability, etc. Each is ranked based on three levels of importance (hard requirement, important, nice-but optional). Next, constraints and benefits for each set of solutions are identified. Then priorities between types of requirements are identified to allow for trade-offs between them. Benefits for each set of solutions are determined, in addition to their feasibility with respect to project constraints. This analysis is done at the requirements level and does not include consideration of detailed design level issues.

Several methods exist to prioritize different requirements. One is simple lexicographic ordering as in [5]. The other is the analytic hierarchy process (AHP) [6]. It uses pair-wise comparison to order an established list. It has been used before for software requirements by Karlsson and Ryan [7]: requirements are evaluated based on value and cost. Shepperd and Cartwright [8] used it to estimate project effort when there was little objective data. AHP allows weighting the requirements items by quantifying their relative importance. This set of weights is then used with levels for individual requirements items to determine the value of a candidate solution. While AHP is not defined for rank ordering measures (use lexicographic ordering instead), it has the advantage that it calculates a consistency index that evaluates the degree to which priorities have been set consistently. The complexity of trading off a multiplicity of requirements items makes such a capability desirable.

There are numerous approaches to trade-off analysis in the literature. A rather comprehensive one is the Defect, Detection and Prevention process (DDP) [9,10,11]. This involves quantifying the comparative importance of requirements, failure modes, and their impact on requirements, as well as the impact of mitigation actions and their cost. Other approaches have been used to evaluate requirements threats, mitigation options and design choices include Quality Function Deployment (QFD) [12], i* [13], WinWin [14], KAOS [15], and constraint analysis [16]. None of these is directly applicable to serve as part of a design composition and analysis strategy, either because it is risk-driven (we are interested in finding a combination of solutions that together satisfy our requirements, rather than emphasizing cost or lead time risks), or because they only cover a small aspect of what we need in a trade-off evaluation strategy. We will emphasize priorities of requirements items and value of the design option with regards to achieving certain levels of a requirement (like performance, security, fault-tolerance).

Design and programming techniques that support Multi-Dimensional Separation of Concerns (MDSoc) allow developers to isolate functionality that cross-cuts the modular structure of design models or code implementations. Attempts at providing support for MDSoc at the programming level has given rise to aspect-oriented programming (AOP) [17,18] and subject-oriented programming [19]. In AOP an aspect is an implementation or design concern that cross-cuts the modular structure of a program [17,18]. A few researchers have started to address the problem of defining and weaving (composing) aspects at an abstraction level higher than the programming language level, see e.g. [20,21].

Gray et al. [21] use aspects to represent cross-cutting functionality (CCFs) in domain-specific models. Their research is part of the Model-Integrated Computing (MIC) initiative that targets embedded software systems specifically. MIC extends the scope and usage of models such that they form the backbone of a development process for building computer-based systems. Requirements, architecture and the environment of a system are captured in the form of formal high-level models that allow the representation of concerns.

In subject-oriented modeling [22,20], a design, called a Subject, is created for each system requirement. A comprehensive design is a composition of subjects. Subjects are expressed as UML model views, and weaving merges the views provided by the subjects. Merging is restricted to adding and overriding named elements in a model. Merging of constraints is not supported, nor is there support for deleting elements from models (except the implicit deletion that occurs when an element is overridden). Conflict resolution mechanisms are limited to defining precedence and override relationships between conflicting elements.

In prior work [2,24] we have shown how concerns can be modeled as aspects, expressed as structural and behavioral patterns specifications, and woven into designs expressed in the UML (e.g., security concerns [3,2], and authentication and auditing [23]). We have also shown that the order in which the aspects are woven is important. We have developed a prototype tool for weaving mapped aspects with primary models [24].

Software performance is one important property of a software system, for which the boundaries often are set by early architectural design decisions. In order to manage software performance requirements effectively and efficiently, a proactive approach is needed [25]. Smith and Williams [25] present an approach to performance analysis,
based on key performance scenarios expressed in terms of UML use case diagrams. They build queuing models for analysis of the performance of a system. An approach based on use case maps is presented by Petriu and Woodside [26]. In addition to the modeling approaches, data collection for performance evaluation is needed [27]. However, studies on reliability assessment show that results are quite robust with respect to estimation of input distributions [28]. Finally, performance evaluations depend on approaches to effectively choose which performance test cases to execute [29, 30]. The probably most well known architecture analysis method is the SAAM (Software Architecture Analysis Method) by Bass et al [31]. This method takes a broader, more general and qualitative approach to architecture analysis and the performance-specific methods can be used in conjunction with SAAM.


Integrating a value-based design analysis framework requires information and evaluation at three levels of representation. They are:

(1) Requirements level. This level shows information at the systems level, including actors and use cases, requirements and value of reaching certain levels of requirements. We also define priorities of requirements at this level.

(2) Logical design level. At this level, we find logical aspect-oriented design (AOD) models consisting of aspects and a primary design model. This is also the level at which trade-off decision are made. The logical level needs information from both the requirements and the physical level.

(3) Physical level. At this level we find the technology-specific platform models and models of performance characteristics and probabilities for faults to occur. Information on performance and fault probabilities is used with related design elements and knowledge of use cases and their frequency to evaluate the designs for feasibility and to make trade-off decisions between requirement and between design options.

Figure 1 shows a diagram with the type of information provided at which level and how the designer uses this information. The figure also relates the steps in our approach to the different levels.

Our approach will use performance, security, and fault tolerance requirements to illustrate how to express and evaluate constraints for UML designs, while its basic principles support trade-off analysis of various combinations of requirements. The approach is iterative and uses information from all three levels defined above. The performance models are refined continuously as the design models evolve, and hence the performance analyses can be performed for each iteration, each time with more precise models and better parameter estimates. The approach consists of the following eight steps, defined below and illustrated in an example case study in Section 5.

1. Define requirements, primary design and aspects. Define security and fault tolerance requirements. Define use cases for the design. Develop the primary design model and aspects for fault tolerance and security requirements. This first step produces articulated requirements, maps them into use cases, and links the requirements level information to a logical AOD model.

2. Associate performance measures with use cases. Each use case is associated with a performance measure derived from the requirements. For example, “configuring probes” in a sensor network is related to the time it takes to configure them (e.g. “minutes”). Or, “capturing data” on a sensor network is measured at a “kHz” or “MHz” rate. Thus, for each use case \( U_i \) \((i=1,...,n)\), there is an associated performance measure \( M_i \) \((i=1,...,n)\).

3. Parameterize use cases. Parameterize use cases with dimensions of “size” or “complexity”. For example a use case “buy a book online” and “buy 150 books online” differs in its performance, both with regards to response time and to memory requirements. This step determines a measure of size of the use cases. Thus, for each use case \( U_i \) \((i=1,...,n)\) we associate a size measure \( S_i \) \((i=1,...,n)\), including a range of values for \( S_i \) from low to high, i.e. \((low_i, high_i)\) for each size measure \( S_i \) \((i=1,...,n)\). For example, the size measure for the “buy books online” use case might be “number of books to be bought”, and the low and high values might be 1 and 150.

4. Develop a model to evaluate performance. A use case is graphically expressed as a set of Sequence Diagrams that each describes a sequence of messages. The messages are restricted to method calls. Each use case use case \( U_i \) \((i=1,...,n)\) is represented by a sequence of method calls \( \{ m_{i,1}, ..., m_{i,n} \} \). For each method, we must associate estimated individual performance characteristics as a function of the size or complexity of message involved. This requires developing a performance function for each message involved. We thus associate performance functions with each method referred to in a use case. This requires information from the physical level (performance characteristics) to be associated with elements of the design on a per method basis which is then aggregated to develop a performance function for the use case (parameterized by size). This analysis also has to determine how many method calls per use case and per size measure are expected. The best approach to represent the results of this analysis is probably a table. It lists in the first column the use cases. The second column
lists high and low ranges for each use case. The remaining columns are labeled with all method calls, grouped by class containing the method. The remaining table entries list the number of methods calls that are made for a given method $m_j$ for a use case $U_i$ for its high and low ranges of size. For example, to configure probes in the range $[5,100]$ requires a single message from the host system to request configuration, a single authentication, and as many configurations actions by the probe class as probes are to be configured, i.e. in the range $[5,100]$. The result of this analysis is now interpreted with respect to actual performance requirements.

5. **Determine levels of requirements.** The levels of requirements for each use case and size of a use case is a value function associated with the performance function developed in step 3. The value function can be of a variety of types. It is important to note that this value function can vary by use cases and that levels of performance required can vary by size level. Some uses may require practically instant response no matter what the size of the problem solved with the use case, while performance requirements may become less strict as size of the use increases. Since we are trying to establish trade-off between design choices, it is more important to establish trade-off by value of a design choice, rather than by risk, or cost. This approach is similar to the utility analysis for various assessment and trade-off situations that occur in software engineering as described by von Mayrhofer [4]. Utility assessment can be done either by the category method or by the direct method. It associates a utility level, usually between 0 (no value) and 1 (perfect), with each value of the performance measure and each security mechanism and fault tolerance aspect. The purpose is to map non-commensurate units (such as level of security, levels of fault tolerance, and performance indicator values) into commensurate units of value, so they can be compared. Utility functions can be discrete (by level of security and level of fault-tolerance) or continuous (as for performance). They can be stepwise, linear, concave, convex, or follow an S-curve. For example, if basic authentication were all that is needed for the security of the system, one would rank this as a value of 1.

6. **Set priorities between requirements.** Set priorities between use cases and between potentially competing requirements. Given that requirements are mapped to use cases, it is necessary to identify priorities for each use case with re-
gards to performance, fault tolerance, and security. This is, because not all use cases have security requirements, fault tolerance requirements, or performance requirements. For example, there may be performance requirements associated with capturing data (fast enough so no data is lost), the security requirements are related to authentication, and fault tolerance requirements are related to error checking. In such a case, on would prioritize the requirements types on a per use case basis.

7. Determine operational profile. This specifies frequency of each use case and, within each use case, frequency of various size levels. We also need to determine volume of use in a target period (such as hour, day, etc.). For example, configuring probes would occur once for every 10,000 measurements gathered. In addition, the most frequent number of probes to configure might be 25.

8. Compare alternatives. Determine a multidimensional matrix that compares for each use case, complexity level and frequency its performance versus the value of requirements levels.

4. Trade-offs Between Requirements

So far, we have shown how to trade-off and optimize performance requirements for a collection of use cases that form a workload for the application design. Now we add analysis of requirements for competing designs. Depending on how these requirements are expressed, there are a variety of ways to do this. We see two steps to this analysis:

1. Evaluate the effect of security and fault-tolerance levels on performance.
2. Trade-off analysis.

The evaluation of the effect of security levels on performance must be done for each use case. This is based on the version of the design that resulted from weaving the various security aspects into the primary UML design. Those use cases, that contain security aspects, will change their performance function, while those that do not, will remain unchanged. We expect that this would only affect a part of the use cases, depending where security activities (e.g. authentication, denial of service, error checking) are performed. This results in a comparison matrix. Each level of security is associated with the resulting performance function for the workload as represented by the performance of the use cases parameterized by their complexity/size.

Regarding the trade-off analysis, there are two ways to look at potential trade-offs:

(a) Determining a feasible design. In this case the analysis is finished if the performance effect of weaving security concerns into the design does not result in performance violations. If feasibility is an issue, priorities between the two types of requirements help to determine which requirement to relax. These priorities should be stated by type of use case, since it is likely that importance of security versus performance varies by type of use.

(b) Determining an optimal design. Optimality is defined by the overall utility function which can allow complete trade-off or include weights. Given n requirements with reached value levels \( v_1, \ldots, v_n \), minimum value levels \( v_{min_1}, \ldots, v_{min_n} \), the two possible optimization problems are:

\[
\text{utility} = \max \frac{1}{n} \left( \sum_{i=1}^{n} v_i \right) \quad \text{subject to} \quad v_i \geq v_{min_i} \quad \text{(EQ 1)}
\]

\[
\text{utility} = \max \frac{1}{n} \left( \sum_{i=1}^{n} a_i v_i \right) \quad \text{subject to} \quad v_i \geq v_{min_i} \quad \text{and}
\sum_{i=1}^{n} a_i = 1, \quad a_i > 0 \quad \text{(EQ 2)}
\]

We also assume that for all options value levels for each requirement can be determined.

5. Example Case Study

In this chapter we present an example case study for a simple system with a number of probes monitoring a network. The system is a scaled down version of a real system [3]. The probes need to be configured, and they need to perform data collection and on-the-fly analysis. In addition the host system will perform post analysis off line. The steps of the method defined in Section 3 are followed below.

1. Define requirements, primary design and aspects. The primary class diagram for the system is shown in Figure 2 and one part of the use case model is shown in Figure 3. The use cases include configuring n probes, collecting a certain amount of data at a frequency \( f \), filtering on the fly, and looking for patterns in the filtered data (i.e. post analysis). The first column in Table 1 lists all use cases. In addition to the primary design, aspects of security and fault tolerance are woven into the class diagrams and the use case models. Security aspects are related to authentication, and fault tolerance aspects have to do with error checking on the data. Performance requirements arise from the need to accept and process data at a rate no higher than the resolution of the probes. The shaded areas in the example below are the aspects woven into it.

2. Associate performance measures with use cases. Table 1 shows this association. The second column lists the performance requirements and the measures identified with each use case. For example, capturing data specifies performance measure as a rate, its value as MHz and allows a maximum data loss of \( 10^{-9} \) if the capture rate is not fast.
The requirements should be given with confidence intervals to enable verification, but in this example we only use mean values. The use case “Check probe” is shadowed since it originates from the fault tolerance aspect woven into the model. The security aspects do not add any new use cases, but add to existing use cases as seen in Figure 3.

3. Parameterize use cases. Each use case is parameterized in terms of “size”, indicating the range in terms of performance that are required. In the example, we specify two variants of each use case; see the rightmost column of Table 1. For $U_1$, configure probes; the low number of probes is 5 while the high number is 100 ($low_1=5$, $high_1=100$). Similarly data may arrive between a kHz and a MHz rate, hence ($low_2=kHz$, $high_2=MHz$). The probability of error for a bad channel and a good channel are also defined ($low_3=10^{-9}$, $high_3=10^{-6}$). Lastly, the size of the post analysis use case can range from analyzing the last hour’s data to a whole month’s worth of data ($low_4=hour$, $high_4=month$).

4. Develop a model to evaluate performance. The performance model consists of two parts, one model which associates each use case with a sequence of method calls in the object model, see Table 2, and one model which associates each method call with the processor or communication load related to executing that method, see Table 3. The load may in turn be a function of the parameters, see for example the method Host:configProbes(n) which requires 1000 cycles as an initial load, and then 200 cycles for each probe to configure.

5. Determine levels of requirements. For each of the use cases, utility functions are developed to capture the characteristics of the requirements of each use case. Given the nature of the requirements for this specific example, the value functions are binary. If the requirement is met, the value is
1, if it is not, the value is 0. Other options would be slowly degrading values as performance declines, but the specific requirements of this example do not warrant such a situation.

6. Set priorities between requirements. To prepare for the trade-off analysis, the requirements must be given priorities or order to enable strategy decisions. The priorities for the example study are as follows:

Prio 1: performance of capture data is twice as important as error check
Prio 2: error check and authentication for configure probes and check probes
Prio 3: performance of configure and post analysis

The most important is to capture the data. Secondary are the security and fault tolerance aspects, and thirdly comes the less frequently initiated use cases of configure probes and post analysis.

The operational profile is defined in terms of frequencies of use for each variant of the use cases; see Table 4. Note that the system has different number of actors \( n_\text{a} \) of the different types: 2 “Adm” actors and 10 “User” actors that have implication in the analysis later since they have different usage frequencies.

8. Compare alternatives. Now we are able to determine a multidimensional matrix that compares for each use case, complexity level and frequency its performance versus the value or requirements levels. In this simple example, we use only mean values, and hence we multiply the load for each use case, with the use frequency and the number of users. The resulting load and capacity is presented in Table 5. We analyze four different cases: 1) the basic functionality with the average number of nodes, i.e. 21; 2) authentication added with the same number of nodes, 3) error check added with the same number of nodes, and 4) the configuration of maximum size with 1000 nodes.

For each case, the load is calculated as the product of number of actors \( n_\text{a} \), frequency of use for each use case and actor \( f_{U_a} \) and the sum of the product of the number of method calls \( c_m \) and the load \( l_m \) for each method call. The total load is then summed up over all use cases:

\[
\text{Load} = \sum_{i=1}^{4} \left( n_{a_i} \cdot f_{U_i} \cdot \sum_{m=m_{i}}^{m_{i+1}} c_m \cdot l_m \right) = \sum_{i=1}^{4} \sum_{m=m_{i}}^{m_{i+1}} c_m \cdot l_m
\]  (EQ 3)

It can be concluded from the analysis in Table 5 that the basic configuration has a lower load than offered capacity \((3.8*10^{13} \text{ vs. } 5.2*10^{12})\). Adding authentication only adds marginally to the load on the system. However adding error check pushes the probes beyond their capacity \((2.3*10^{13} \text{ vs. })\).
5.2*10^{12}). In the max case, the host capacity is still sufficient while the probes are still a bottleneck. Given the priorities, we will relax the error check requirement to meet the more important performance requirement.

The analysis shows that either the capacity has to be increased, or the error checking function cannot be performed. Table 6 shows the value levels associated with each use case, based on having authentication added or not, and based on having error check added or not. Utility is modeled as a 1, if requirements are met, 0 otherwise. When a use case does not need either a security function or an error check, it is marked as N/A. Priorities are set as performance being twice as important as error check for capture data and check probe use cases. The rightmost column indicates overall value for each design choice. Then, the best option of the first use case is to implement authentication. For the second and third use case, ignoring error check gives a higher utility value, given the priorities. The fourth use case needs neither authentication nor error check for optimum value.

6. Conclusions

Designing software systems involves decisions about trade-offs between different requirements, such as functionality, performance and security. In this paper we present an ap-
The approach to systematically address the requirements and the trade-off between them.

The approach is based on weaving non-functional requirements modeled as aspects into the primary UML model. We add information regarding performance measures to the use cases, parameterize the use cases with “size”, map use case onto sequences of methods, determine the operational profile and hence are able to calculate the load each use case generates on the system. Then, trade-off analysis can take place between the competing requirements.

We present an example system consisting of probes monitoring a network. In addition to the basic model, we define security requirements and fault tolerance requirements as aspects. The analysis helps finding out that there is a conflict between the fault tolerance and the performance requirements. The analysis also identifies the bottlenecks of the design.

<table>
<thead>
<tr>
<th>Use case</th>
<th>Performance</th>
<th>Security</th>
<th>Fault tolerance</th>
<th>Utility</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>U1 Configure probes</strong></td>
<td>Basic model</td>
<td>1</td>
<td>N/A</td>
<td>0.5</td>
</tr>
<tr>
<td>Authentication</td>
<td>1</td>
<td>N/A</td>
<td>N/A</td>
<td>1.0</td>
</tr>
<tr>
<td>Error checking</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>0.67</td>
</tr>
<tr>
<td><strong>U2 Capture data</strong></td>
<td>Basic model</td>
<td>1</td>
<td>N/A</td>
<td>0.67</td>
</tr>
<tr>
<td>Authentication</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>0.67</td>
</tr>
<tr>
<td>Error checking</td>
<td>0</td>
<td>N/A</td>
<td>N/A</td>
<td>0.33</td>
</tr>
<tr>
<td><strong>U3 Check probe</strong></td>
<td>Basic model</td>
<td>1</td>
<td>N/A</td>
<td>1.0</td>
</tr>
<tr>
<td>Authentication</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>1.0</td>
</tr>
<tr>
<td>Error checking</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>0.33</td>
</tr>
<tr>
<td><strong>U4 Post analysis</strong></td>
<td>Basic model</td>
<td>1</td>
<td>N/A</td>
<td>1.0</td>
</tr>
<tr>
<td>Authentication</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>1.0</td>
</tr>
<tr>
<td>Error checking</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>0.33</td>
</tr>
</tbody>
</table>

The example case study shows that the method is feasible to use and therefore the next step is to scale it up to systems of more realistic size. Future work also includes selecting representations for the use case properties and the linking between use cases and methods to enable more efficient tool support for the calculations. We also intend to investigate more advanced performance models and investigate the sensibility of the estimates to uncertainties in the underlying estimates of cycle times and operational profiles.

Acknowledgement

We are grateful to the Fulbright commission for making Dr. Runeson’s sabbatical visit to Washington State University possible, during which this work was performed. Thanks to Mr. Edward Mancebo for checking the calculations.

References


